Two is Better Than One: The Case for 2-Tree for Skewed Data Sets

Xinjing Zhou
MIT CSAIL
xinjing@mit.edu

Xiangyao Yu
University of Wisconsin-Madison
yxy@cs.wisc.edu

Goetz Graefe
Google
goetzg@google.com

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

ABSTRACT

Real-world data sets almost always exhibit skew, i.e., a majority of the accesses go to a minority of the records. Obviously, Smith and Brown are more popular names than Stonebraker and Graefe. Traditional block-oriented index structures such as B-trees are sub-optimal for skewed data because an index block often has a small number of hot records and a larger number of cold ones. This results in poor main memory utilization and increased cost.

To alleviate this problem, we propose a 2-Tree architecture, where hot index records are in one tree and cold ones are in a second tree. Hot tree blocks are frequently accessed and likely to remain in main memory, resulting in improved main memory utilization. Our core idea is to employ a lightweight general migration protocol to move records between trees in both directions when appropriate and to maintain access statistics at low cost.

In addition, the two trees can be configured separately for hardware differences. One tree can be optimized for main memory while the second exploits secondary storage. Obviously, the 2-Tree idea can also be generalized to multiple storage levels and/or devices. We show how the 2-Tree idea and record migration can be applied to both B+trees and LSM-trees to improve their memory utilization significantly (by 15× and 20× respectively) on a highly skewed workload. We also observed up to 1.7× throughput improvement on a Zipfian-skewed IO-bound workload compared to traditional single B+tree or LSM-tree using the same amount of main memory. Unlike existing solutions for improving memory utilization at the cost of inferior range scan performance, 2-Tree refuses to make such a compromise.

1 INTRODUCTION

Real-world keyed data is invariably highly skewed. A subset (the working set) of the data has a much higher access frequency than the rest [3, 6, 7, 33, 39]. For example, celebrities on social media get orders of magnitude more page views than average users. On the NYSE 40 stocks account for 60 percent of the daily transaction volume. Generally, the hot records are spread across the entire key space [8] rather than clustered in a few subranges. Lastly, this working set is often not static [4]. For example, trending tweets and breaking news change over time.

The traditional approach to indexing keyed data sets is to employ a homogeneous data structure such as B+tree with a main memory buffer pool. In this way, hot blocks are cached in the main memory buffer pool and cold blocks reside on secondary storage. This approach manages and migrates both main memory and disk-resident data at block granularity. On skewed data sets main memory blocks might have only one hot record in a block containing hundreds of cold records. This results in poor memory utilization and sub-optimal performance.

In this paper, we advocate migrating data at the record-level. We study a 2-Tree architecture in which there are two separate tree data structures, one (top tree) for the hot records and a second (bottom tree) for the cold ones. When a hot record becomes cold, it is migrated from one tree to the other. Likewise, records can move in the other direction. At the core of this architecture is a general-purpose migration protocol, which can accurately detect and maintain hot records at low cost. It adds 3 bits per hot record and works with any tree data structures. With this clustering of hot records, 2-Tree significantly increases memory utilization on skewed data.

Another advantage of this architecture is that the two data structures can be optimized separately for their underlying storage medium. The hot structure can be optimized for main memory, while the cold structure can be optimized for secondary storage. Therefore, 2-Tree can be used as an indexing architecture for main-memory database indexing that extends to workloads larger than memory [11, 13].

We can also generalize 2-Tree architecture to an N-Tree architecture to adapt to systems with more than two distinct storage levels and/or devices. While this is somewhat similar to a well-known multi-tree structure, LSM-tree [28], N-Tree moves data upwards upon read as well. Such upward migration can help improve memory utilization under read-heavy workloads for which LSM-tree is not optimized.

In this paper, we present three case studies of applying 2-Tree architecture and record-level migration. First, we study an application of 2-Tree for indexing in main-memory database in a larger-than-memory setting [11, 13] and show that it can outperform the Anti-Caching [11] approach significantly. Second, we show that two buffer-managed B+trees combined with record migration can significantly outperform a state-of-the-art single B+tree implementation [19] by improving buffer pool memory utilization using the same amount of main memory. Lastly, we show a preliminary N-Tree implementation by simply augmenting LSM-tree with record-level upward migration. This improves their performance significantly versus vanilla LSM-trees on read-heavy workloads.

We summarize our contributions as follows:

- We propose the 2-Tree architecture to address the limitations of existing approaches for managing larger-than-memory indexes.
- We propose an efficient record migration protocol that works between any two tree structures.

 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

- We experimentally show that the 2-Tree approach can outperform existing single-structured B+tree as well as Anti-Caching.
- We present a preliminary N-Tree implementation by augmenting LSM-tree with upward migration and show that it delivers higher memory utilization across all workloads.
- We describe several future research directions.

2 Related Research

There have been many previous works on heterogeneous index structures. We discuss a few representative ones in both disk-based systems and main-memory-based systems.

LSM-tree. One well-known example of heterogeneous tree structures is the log-structured merge-tree (LSM-tree) [29]. The original proposal is a hierarchy of on-disk B+trees with exponentially increasing capacities. Writes are buffered in an in-memory B+tree. A rolling merge operation is used to batch-propagate updates from higher-level trees to lower-level ones. Due to the complexity of the rolling merge to B+trees, modern LSM-tree implementations [10, 34] replaced the B+tree for each on-disk level with an immutable block-based sorted-string-table (SST). A similar merge procedure called compaction is used to merge SST files with overlapping key ranges to produce a new SST file. To speed up reads, modern implementations typically cache blocks from SST files in memory. In a sense, LSM-tree is similar to N-Tree in that there are multiple tree structures. Records frequently written in the working set are naturally clustered on higher levels SSTs, resulting in good memory utilization for these records. However, records only receiving reads can still be scattered among levels, resulting in low utilization for the block cache. The key difference between an LSM-tree and the N-Tree architecture is that data only moves from higher levels to lower levels upon write in an LSM tree. On the other hand, the N-Tree architecture actively moves data in both directions upon read and write. Therefore, it can handle read-heavy workloads as well. Nevertheless, we show that 2-Tree migration protocol can be applied to a vanilla LSM-tree to increase memory utilization for read-heavy workloads.

Record Caching. One general approach for improving memory utilization of page-based storage engines is maintaining separate in-memory caches for hot records[23, 34, 40] extracted from the on-disk pages or blocks. The cache in these approaches is typically for serving read-only queries. To support modifications to an in-memory record cache, one needs to devise a new recovery mechanism that both handles the record cache and page-based storage manager [40]. 2-Tree, on the other hand, can operate through page-oriented access only. Therefore, it can easily reuse existing page-based recovery mechanisms, such as ARIES [27]. This simplifies system design and implementation.

The second issue with record caching is that it is not as versatile as block caching [38]. Record caching is only able to serve point queries. For example, RocksDB [34] features a read-only global row cache that keeps frequently-accessed records of SST files in an in-memory hash table. Hot records can be served from the global row cache, avoiding accessing SST files and increasing memory utilization. However, the row cache is not helpful to other important operations such as range queries and compactions that require scanning data in blocks. Moreover, record caching takes away the memory budget from block cache that could otherwise be used to accelerate range queries. As we will show in the experiments, the row cache in RocksDB enables high memory utilization for point reads at the cost of reducing the performance of other operations. Our proposal, LSM-tree with upward migration, caches only data blocks with high utilization and maintains competitive range scan performance.

Anti-Caching. Another example that follows a heterogeneous design philosophy is Anti-Caching [11], which manages hot memory-resident data at tuple-granularity without using a buffer pool interface to maximize memory utilization and keeps the cold data on disk at page granularity. However, Anti-Caching keeps metadata for every evicted record in main memory and requires all secondary indexes to be memory-resident, which could take up a large portion of the memory [41]. In fact, the memory overhead of Anti-Caching is $O(N)$, where N is the number of evicted records. In contrast, 2-Tree has a constant memory overhead for evicted records, which results in a more scalable system. Another major limitation is that Anti-Caching is unable to handle range scans efficiently. This is because the Anti-Caching architecture keeps evicted tuples unordered on disk. 2-Tree, on the other hand, maintains key order for evicted data. Therefore, 2-Tree can support efficient range scan.

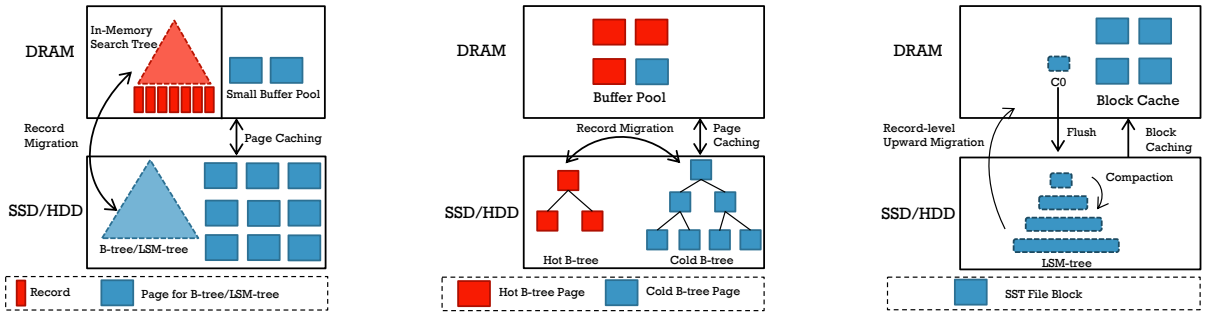
Siberia. Another system related to 2-Tree is Siberia [13], a part of Hekaton [12]. Siberia indexes hot tuples in a memory-optimized data store (hot store) and cold data in a traditional page-based storage manager (cold store). 2-Tree differs from Siberia in two major ways. First, Siberia does not migrate data from the cold store to the hot store upon read. This is sub-optimal for read-only or changing workloads. Second, the hot data identification mechanism in Siberia is offline through analyzing access logs. In contrast, 2-Tree's lightweight online approach requires fewer computing resources and results in more timely adaptation to workload changes.

3 2-Tree Design

In this section, we describe the 2-Tree architecture in detail. We start with an overview of the architecture in Section 3.1 followed by a discussion of the design principles of record migration in Section 3.2. We then describe how each basic tree operation works and how migration works in detail in Section 3.3. We end this section with a brief discussion of the durability and recovery in Section 3.4. For convenience, we use *record* throughout the rest of the paper in multiple contexts. In a relational database setting, we use *record* to represent a keyed database tuple or a secondary index entry that stores the primary key of the tuple it points to. In a key-value store setting *record* refers to a keyed binary payload.

3.1 Overview

The core idea of the architecture is physically separating hot records from cold ones. Figure 1 illustrates the three cases we consider. Figure 1a shows a 2-Tree version of the Anti-Caching architecture. Note that the data of the hot tree is kept in memory completely and indexed at record granularity without using a buffer pool. Data evicted are stored in an on-disk tree structure that is accessed through a small buffer pool. The migration works at record-level between the in-memory hot tree and the on-disk tree. Figure 1b gives the two B-tree architecture using a shared buffer pool. The same record migration is used. Lastly, Figure 1c shows an LSM-tree



(a) 2-Tree for Anti-Caching (b) 2-Tree for B+tree (c) LSM-tree with 2-Tree Upward Migration

Figure 1: Applications of 2-Tree Architecture

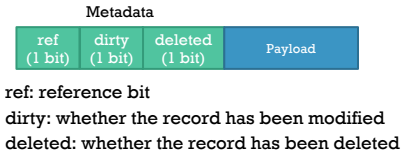


Figure 2: Top Tree Record Format

augmented with upward migration protocol. Note Figure 1b and Figure 1c are disk-based systems. Therefore, data is still stored on disk. However, pages or blocks stored at higher levels of the hierarchy are expected to be accessed more likely, taking a larger portion of the buffer pool which results in better memory utilization.

3.2 Migration Design Principles

In this section, we present the design principles for record migration in 2-Tree.

Downward Migration. When the allowable memory budget is exhausted, we must decide what records to evict from the top tree. A classic approach is to choose the least recently used (LRU) records for eviction. However, the classic LRU algorithm must maintain an ordered list of records, and the resulting per-record memory overhead is high. Instead, we exploit the range scan operation commonly supported by tree data structures to devise a variant of the clock replacement algorithm [9], which approximates LRU. Shown in Figure 2, a record in the top tree is extended with metadata containing a reference bit which is set upon record access. A clock handle, i.e., a key value indicating the current progress of the eviction scan, is maintained in memory. When eviction is needed, the system cycles through every record starting after the clock handle. It collects records with the reference bit off for eviction. It also clears the reference bit of records examined. The scan stops when the desired number of records has been collected.

Probabilistically Deferred Upward Migration. The next decision to make is when to move data upwards. A simple policy is to always move data upwards upon reading from the bottom tree. However, such an approach might promote cold data that forces the eviction of hot data in the top tree. A classic example of this effect is a sequential scan that accesses all data exactly once. There have been many proposed caching strategies that guard against such thrashing. These strategies typically employ some form of cache partitioning and/or frequency tracking that incur non-trivial per-record bookkeeping.

Instead, we adopt a sampling-based approach where we move only a sample of accessed records upwards. We define a sampling rate as $D(0 < D \leq 1)$. For data that is becoming hot, its frequency of access increases, and therefore 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 up the cache quickly while providing little thrash resistance. In contrast, a small sampling rate delivers thrash resistance by sacrificing the warm-up rate. Notice that with sampling rate $D = 1$, our approach is identical to "always move upward on access". Finding the best sampling rate is a typical knob-tuning task, and there are many automatic techniques [35, 36, 42] available. For simplicity, in this paper, we manually find the sampling rate that works well in our evaluation. We leave systematic tuning of such parameter as future work.

Inclusive versus Exclusive Migration Policy. Another important decision to make when migrating records is how many copies of a record to keep in the system. In an exclusive policy, the system only keeps one copy of the record in either the top or bottom tree. When migrating data from the bottom tree upwards, the record is erased from the bottom tree. In an inclusive policy, the record is retained in the bottom tree when it is copied to the top tree. An inclusive policy avoids modifying the bottom tree during migration, which is beneficial for IO-bound workloads. However, an inclusive policy results in a larger overall tree size as duplication increases. In this paper, we focus on an inclusive migration policy. For completeness, we cover both inclusive and exclusive in the description of algorithms. However, we leave the experimental exploration of exclusive policy as future work.

3.3 Base Operations

We next describe specific tree operations under the 2-Tree architecture with record migration, adhering to the principles described above. We assume that all the records are uniquely identified by a key.

Lookup Operation. As shown in Figure 4, the lookup operation first searches for the key in the top tree. If the record found in the top tree is undergoing migration, the operation is retried (Line 4

```
UPMIGRATION Input: key, op, new_value
# record does not exist in the top tree
record = find key in bottom tree
if record exists:
    if op == OP::UPDATE:
        record.payload = new_value
        update record in bottom tree
    if random(0, 1) <= D: # Lazy Upward Migration
        insert record into top tree
        if exclusive policy:
            delete record from bottom tree
    return (SUCCEEDED, record)
return (NON_EXISTENT, null)
```

Figure 3: Upward Migration Procedure

```
LOOKUP Input: key
record = find key in top tree
if record exists:
    if record.deleted: return null
    if record.ref == false:
        set record.ref to true in top tree # retain hotness
    return record.payload

# record does not exist in the top tree
status, record = UPMIGRATION(key, OP::LOOKUP, null)

if status == NON_EXISTENT: return null
else: return record.payload
```

Figure 4: Lookup Procedure

```
UPDATE Input: key, new_value
record = find key in top tree for write
if record exists:
    if record.deleted: return false
    update record with reference = true, dirty = true, payload = new_value
    ↪ in top tree
    return true

status, record = UPMIGRATION(key, OP::UPDATE, new_value)

if status == NON_EXISTENT: return false
else: return true
```

Figure 5: Update Procedure

of Figure 4). Otherwise, we update the reference bit of the record to retain hotness and return the value (Line 5 of Figure 4). If not found in the top tree, the search continues in the bottom tree. If found, the upward migration (Figure 3) algorithm is performed, which probabilistically moves the record to the top tree. In addition, the record is erased from the bottom tree if the system is configured with an exclusive policy. The value found in the bottom tree is returned to the user.

Update Operation. Similar to Lookup operation, shown in Figure 5, the update operation first searches for the key in the top tree. If the key is found, we update the reference bit, dirty bit, and contents of the record. The update operation continues to bottom tree if the key is not found in the top tree. A similar upward migration mechanism is performed.

Put Operation. A put operation performs an upsert that does not check the existence of the record in both tree structures. This is useful when loading data or performing blind writes. Put operation always inserts records first in the top tree with the reference bit and dirty bit set. They get migrated down to the bottom tree via eviction as the top tree fills up.

Insert Operation An insertion operation first checks the existence of a keyed record by using the Lookup operation. This is

```
EVICTION Input:
victims = {}
for record in range(clock_hand, ∞) of top tree for write:
    record.ref == true: # toggle reference bit
        record.ref = false
    else:
        victims = victims ∪ record
        if victims set is large enough:
            break
update clock_hand

for record in victims:
    if record.deleted: # apply the deletion to the bottom tree
        delete record from bottom tree
    elif record.dirty: # write back the latest copy of the record
        upsert record into bottom tree
    else: # not dirty
        if exclusive policy:
            insert record into bottom tree
    delete record from top tree
```

Figure 6: Eviction Procedure

useful for implementing the primary key in relational databases. If the record does not exist, it inserts a record into the top tree. If the system is employed with inclusive policy, both reference and dirty bit are set. Otherwise, only the reference bit will be set.

Delete Operation. A delete operation first searches the top tree for the key. 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 only applied to the bottom tree 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.

Range Scan Operation. The scan of a 2-Tree structure merges the scans of the individual trees. This has been done in many LSM-tree implementations. When merging results, we need to handle two cases. **Case #1:** a record exists in only one tree. In this case, the system returns the record. **Case #2:** the record is in both trees. In this case, the system returns the record in the top tree as it reflects the latest version.

Eviction. We trigger eviction by periodically checking if the memory consumption of the system has exceeded a threshold. As shown in Figure 6, the eviction process starts with a range scan of the top tree starting after the clock handle. The process iterates through every record. If the reference bit of a record is on, it is toggled off. Otherwise, it is added to the eviction set. We end the scan operation after the eviction set has collected the desired number of records. We first apply deletions to the bottom tree for those delete-marked records. Then, all dirty records are upserted into the bottom tree. For non-dirty records, they get inserted into the bottom if the system is configured with an exclusive migration policy. Lastly, we erase records in the eviction set from top tree.

3.4 Durability and Recovery

We notice that migration does not change logical user content. It merely changes the representation of the physical data structures. Therefore, we use system transactions [15] as a general approach to ensure the atomicity and durability of migration. 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 tree operation or multiple records during the eviction. For disk-resident page-based 2-Tree systems (Figure 1b), ARIES [27] is employed for durability and recovery. 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 stable storage. For 2-Tree for Anti-Caching (Figure 1a), the durability and recovery mechanism described in the original paper [11] is compatible with 2-Tree. Therefore, we omit the discussion. For LSM-tree with upward migration (Figure 1c), we can apply the same system transaction idea to guarantee the durability and recoverability of the upward migration operation at low-cost. Since modern LSM-tree implementations perform writes out-of-place and employ write-ahead-log for its memory-resident write buffer, we may represent the upward migration as a normal write operation. Similarly, such a write operation is lightweight as it does not need to guarantee durability.

4 Implementations

We present four implementations following the principles of 2-Tree ranging from on-disk to in-memory structures.[1]

4.1 Disk-Based Structures

2B+tree. We use two on-disk LeanStore B+trees [19] sharing a single buffer pool with record migration. The eviction process is activated when the hot B+tree consumes more than 90% of the buffer pool capacity. We chose this threshold to make sure the index nodes of the cold B+tree are memory-resident [16]. We found that the replacement policy in LeanStore randomly picks a page as eviction candidate, which is undesirable as we want to keep hot tree pages in memory. Although LeanStore gives each eviction candidate a grace period before real eviction, hot tree pages will still often be chosen for eviction because they take up the vast majority of the buffer pool frames. We fix this problem by ensuring pages from the cold tree are 10× more likely to be chosen for eviction than the hot-tree ones.

UpLSM-tree. As discussed in Section 2, vanilla LSM-tree suffers from low memory utilization when the workload on the working set is read-only. We leave existing downward migration in LSM-tree as is and apply the upward migration from 2-Tree to address this problem. Specifically, we consider migrating a record upwards to the highest level (C0). This happens when a point read operation incurs a miss in the block cache. This suggests that the I/O likely occurred in the lower levels as higher levels are typically small and cached by block cache. Therefore, migrating such a record upwards helps cluster records frequently read. We apply the same probabilistic approach to ensure only warm records get migrated and to reduce excessive writes.

4.2 Memory-Optimized Structures

IM-2B+tree. We use a memory-optimized B+tree [5] as the top tree that eschews buffer pool interface and a LeanStore B+tree as the bottom tree. We dedicate 90% of the available memory to the top tree that stores data directly in memory and 10% of the memory to the LeanStore buffer pool. The eviction process is activated when the top tree exceeds its memory budget. We chose this threshold to make sure there is enough memory to cache the index nodes of the cold B+tree on-disk.

Trie+B+tree. This is similar to IM-2B+tree, except that the top tree is indexed using the adaptive radix tree [20].

[1]Source code: https://github.com/zxjcarrot/2-Tree

5 EXPERIMENTAL RESULTS

In this section, we conduct preliminary experiments with a single worker aimed to answer the following questions:

- How effective is the 2-Tree concept at improving state-of-the-art disk-based indexing schemes, namely LeanStore for B-trees and RocksDB for LSM trees, in terms of buffer pool memory utilization under point operations?
- How does 2-Tree compare to Anti-Caching in terms of data set scalability?
- How much overhead does 2-Tree approach impose on range scan operations?

5.1 Baselines

We compare our proposals against the following baselines:

B+tree. For a single-structured B+tree, we use LeanStore [19] as the baseline, which is a state-of-the-art B+tree implementation backed by a fast buffer manager.

LSM-tree. For an LSM-tree, we use RocksDB as the baseline, which is a popular production-grade implementation.

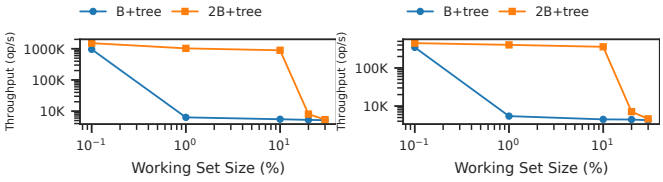
LSM-tree with Row Cache. RocksDB comes with Row Cache feature that keeps frequently-accessed records of SST files in an in-memory global cache. The RocksDB lookup procedure consults the global cache before searching each file, thereby increasing memory utilization. For a given memory budget, we configure 90% of the budget to row cache. We then leave the rest of the 10% budget to Block Cache to make sure that all blocks containing fence pointers and bloom filters are cached in memory. Therefore, this allows at most one I/O per SST file.

Anti-Caching. We implemented Anti-Caching [11] for main-memory DBMS. We use the same main-memory optimized B+tree [5] for indexing records, which are also chained in a doubly-linked LRU list. We use a similar sampling-based approach to reduce the maintenance overhead of the LRU list introduced in the original paper [11]. When memory is exhausted, we assemble a 1KB block consisting of the coldest records taken from the LRU list. We use another in-memory B+tree to index the metadata of each evicted tuple. The metadata includes block id and record offset within a block. We store blocks in a LeanStore B+tree on-disk keyed by block id. Upon access to an evicted block, all records in the block are attached to the LRU list in memory as the coldest.

5.2 Experimental Setup

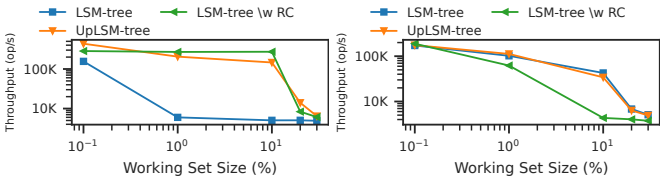
Workload. We use the Yahoo! Cloud Serving Benchmark (YCSB) as our main workload. Specifically, we use a 5 GB data set with 20 million records, each with an 8-byte key and 248 random bytes of payload. Records are clustered within each index, except for trie, which stores pointers to the record in-memory. We evaluated 2-Tree under the following access distributions:

Hotspot. This distribution has a fixed working set with randomly chosen records. Each record in the working set is accessed uniformly. This measures how big a working set each of the approaches can maintain in-memory.



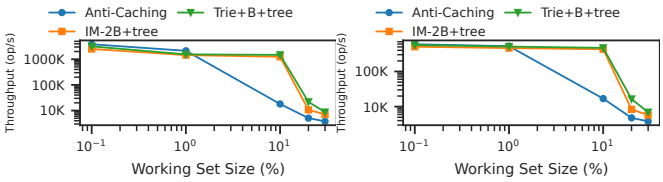
(a) Read-Only (b) Update

Figure 7: Throughput of 2B+tree and Single B+tree on Hotspot Request Distribution with Varying Working Set Size.



(a) Read-Only (b) Update

Figure 8: Throughput of UpLSM-tree and LSM-tree Variations on Hotspot Request Distribution with Varying Working Set Size.



(a) Read-Only (b) Update

Figure 9: Throughput for 2-Tree Variations and Anti-Caching on Hotspot Request Distribution with Varying Working Set Size.

Zipfian. We also evaluated a Zipfian access distribution that is typically seen in real-world workloads.

All experiments are conducted on a Google Cloud instance with 16 2.30 GHz virtual CPUs and 14.8 GB RAM. The instance has a local SSD with a read latency of around 250 us. To focus solely on buffer pool memory utilization, we disable write-ahead logging and the Linux page cache for all experiments. Therefore, the only I/O are the misses in the buffer pool for the B+tree and compactions for the LSM-tree.

To highlight the importance of memory utilization, we fixed the memory budget to be 1 GB which is about 20% of the raw data set size. In such a memory-constrained setting, an improvement in utilization translates to better performance. We use a page size of 16 KB for LeanStore and 32 KB for RocksDB. For all experiments, we first sequentially load the data set and then warm up the systems by running the target workload until the system throughput stabilizes before measuring the average throughput. By default, 2-Tree variations are configured with inclusive policy and lazy upward migration with a sampling rate D of 0.5, which we found performs reasonably well. All experiments were run with a single worker thread that performs normal operations as well as migration. We leave concurrent migrations as future work.

5.3 Point Operations

We first evaluate 2-Tree architecture with point operations including lookup and read-modify-write (update). We ran the experiments under Hotspot and Zipfian access distribution separately.

5.3.1 ***Hotspot Results*** In this experiment, we vary the working set size from 0.1% to 30% of the entire data set. All accesses go to the working set uniformly.

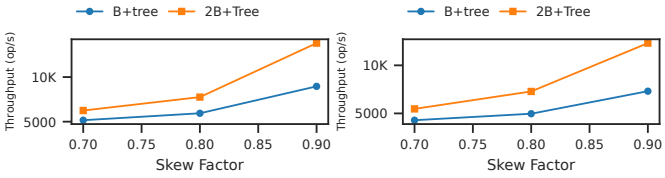
B+tree. With the read-only workload shown in Figure 7a, 2B+tree can keep a working set of up to 15% of the data set in memory. For traditional B+tree, the throughput plummeted after the working set reaches 1% of the data set. For this workload, 2B+tree can successfully manage a working set more than 15× bigger than on a B+tree. Even when the working set is 0.1% of the date set and all approaches can maintain the working set in main memory, 2B+tree still outperforms B+tree by 1.6×. The reason concerns the capacity of the top tree which is capped by the memory budget (20% of the data set). Therefore its index nodes are much smaller than that of single B+tree, resulting in improved CPU cache efficiency. We observe similar improvements for 2B+tree over B+tree on the update workload in Figure 7b.

LSM-tree. Similarly, UpLSM-tree and LSM-tree with row cache can keep a much bigger (up to 20× bigger) working set in memory than vanilla LSM-tree on the read-only workload shown in Figure 8a. However, for update workload shown in Figure 8b, we observe no significant memory utilization difference between UpLSM-tree and vanilla LSM-tree. This is because an update workload naturally clusters records in the working set at the top levels of an LSM-tree, resulting in high memory utilization. Surprisingly, LSM-tree with row cache was outperformed by vanilla LSM-tree and UpLSM-tree by up to 9×. The reason is as follows. An update operation consists of a read followed by a write on the same key. Since RocksDB performs writes out-of-place, each write will effectively invalidate the record in the row cache which likely results in an I/O for the following read on the same key. Therefore, there is effectively no reuse for a record in the row cache. In contrast, a bigger block cache, as in the case with UpLSM-tree and vanilla LSM-tree, can help reduce I/Os by caching more SST files which explain the performance difference.

Anti-Caching. The results of the comparison of 2-Tree against Anti-Caching are shown in Figures 9a and 9b. All systems have similar throughput when the working set can be kept in memory. However, Anti-Caching is only able to keep a working set up to 3% of the data set. Whereas all 2-Tree variations can cache up to 17% of the working set. This is because the in-memory eviction table in Anti-Caching takes too much memory (54%) for indexing the evicted data.

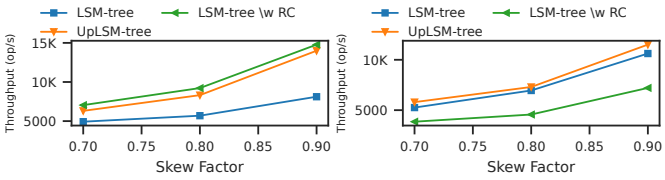
5.3.2 ***Zipfian Results*** We next evaluate the 2-Tree architecture for a Zipfian distribution, varying the skew factor from 0.7 to 0.9. For a skew factor of 0.8, 80% of the accesses go to 32% of the records.

B+tree. The results comparing 2-Tree against single B+tree structures are shown in Figures 10a and 10b. For 2B+tree versus B+tree, we observed 1.54× throughput increase for a read only workload at skew factor 0.9. This is due to the increased buffer pool memory utilization when clustering hot records in the top tree compared to a single B+tree. For the update workload, we observed



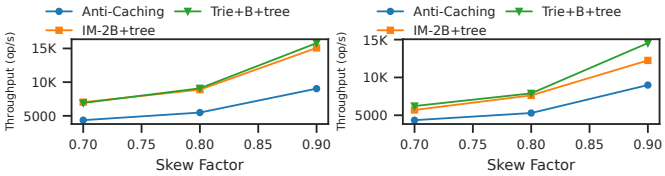
(a) Read-Only (b) Update

Figure 10: Throughput of 2B+tree and Single B+tree on Zipfian Request Distribution.



(a) Read-Only (b) Update

Figure 11: Throughput of UpLSM-tree and LSM-tree Variations on Zipfian Request Distribution.



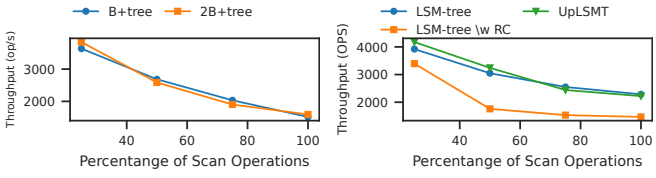
(a) Read-Only (b) Update

Figure 12: Throughput for 2-Tree Variations and Anti-Caching on Zipfian Request Distribution

up to 1.7× throughput improvement. This is because 2B+tree helps reduce write amplification. Therefore, it pays lower disk write overhead per operation compared to a single B+tree.

LSM-tree. The results comparing LSM-tree variations are shown in Figures 11a and 11b. UpLSM-tree outperformed vanilla LSM-tree by up to 1.73× on read-only workload thanks to the increased utilization of the block cache. UpLSM-tree is competitive against LSM-tree with row cache on read-only workload. On update workload, we did not observe improvements over LSM-tree using UpLSM-tree because such a workload already maintains high memory utilization for a vanilla LSM-tree. However, UpLSM-tree outperformed LSM-tree with row cache by 1.6× for the same reasoning explained in Section 5.3.1.

Anti-Caching. The results of comparison of 2-Tree against Anti-Caching are shown in Figures 12a and 12b. Anti-Caching is consistently outperformed by other 2-Tree variations by up to 4× and 3.1× respectively for read-only and update workloads. This is mainly because the evicted table in Anti-Caching occupies 54% of the memory budget, leaving little memory for hot records. Whereas 2-Tree variations do not keep metadata for the evicted record. Hence, they can keep more hot records in memory. The difference between Trie+B+tree and IM-2B+tree is not significant (within 10%) because the performance gain from in-memory indexing is dwarfed by I/O latency in this workload.



(a) B-tree Variations (b) LSM-tree Variations

Figure 13: Range Scan Performance of 2-Tree Architecture Applied to B-tree and LSM-tree

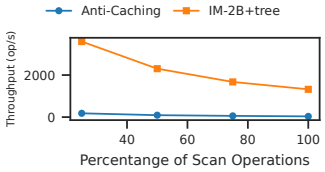


Figure 14: Range Scan Performance of Memory-Optimized 2-Tree and Anti-Caching

5.3.3 ***Summary*** To summarize, we showed that 2B+tree significantly increased the buffer pool memory utilization over a single B+tree. We also showed UpLSM-tree increased the block cache memory utilization for read-only workloads over vanilla LSM-tree without hurting the performance of update-heavy workloads. Lastly, 2-Tree variations can keep a working set much larger than Anti-Caching in memory.

5.4 Range Scan Operations

We next evaluate 2-Tree architecture under range scan operations. We ran a workload with a mixture of point lookup and range scan operations. We vary the percentage of range scan operations from 25% to 100%. The scan operation examines 100 consecutive records starting at a specific key. We use a Zipfian access distribution with the skew factor set to 0.8.

B+tree. As shown in Figure 13a, 2B+tree performs similarly compare to a single B+tree as we increase the percentage of scan operations. This is because the top tree in 2B+tree is mostly cached in the buffer pool. Therefore, most of the I/Os for scans are from the bottom B+tree which has a similar number of leaves compared to that of single B+tree.

LSM-tree. For LSM-tree variations shown in Figure 13b, LSM-tree and UpLSM-tree preform indistinguishably as we increase the percentage of scan operations since they have the same amount of block cache. However, we observe 83% lower throughput for LSM-tree with Row Cache (LSM-tree with RC). This is because a range scan operation in LSM-tree requires scanning blocks in SST files. The block cache can help with reducing I/Os in such case whereas the row cache cannot. Therefore, caching at the block-level is more versatile than caching at the record-level.

Anti-Caching. The results are shown in Figure 14. IM-2B+tree significantly outperforms Anti-Caching by up to 35× because IM-2B+tree keeps evicted data ordered on disk. However, the Anti-Caching design optimizes for eviction and keeps evicted data unordered on disk. This results in many random I/O operations per scan. Therefore, it cannot support efficient scan over data on disk.

6 FUTURE DIRECTIONS

We also see at least five promising future research directions related to 2-Tree:

Physical Concurrency Control. This paper only covers the case where there is only one thread accessing a 2-Tree data structure. This only works well in single-threaded architecture such as H-Store/VoltDB [17, 31, 32] and Redis [30]. In future work, we plan to address the problem of coordinating concurrency within a 2-Tree structure. Specifically, we need to guarantee that data structures stay consistent in the face of concurrent migrations and normal accesses. We will leverage fruitful research results in concurrent tree data structures [22, 26] and synchronization techniques [2, 18, 21].

Adapting to Non-tree-based Structures. There is no fundamental reason why we cannot extend this architecture to more data structures to optimize for skews. For example, widely-used structures such as extendible/linear [14, 25] hashing and heap file fit easily in this architecture by having two structures instead of one. As a future work, we plan to generalize 2-Tree to handle non-tree-based database index and storage structures.

Applications in Cloud-Native Databases. 2-Tree architecture can be directly applied to modern cloud-native databases where storage is disaggregated from compute [1, 24, 37]. This can help increase the memory utilization of the buffer pool on the compute node, reducing round trips to the storage node which is typically more costly than accessing local storage devices. Furthermore, 2-Tree architecture and the fact that storage nodes can do computation open up more optimizations. For example, we can migrate data at record granularity instead of page between compute and storage, reducing network traffic.

Multi-Tier Memory Hierarchies. Several multi-tier solutions have been proposed, ranging from local memory (Persistent Memory) to remote memory (memory expansion and pooling using Compute Express Link and/or RDMA) with varying performance and cost characteristics. We expect to extend the 2-Tree architecture to multiple levels. New caching and data migration techniques are likely needed in such a diverse design space.

Compressed DRAM. Compression is a promising approach to keeping more data in DRAM. For example, the latency of decompressing a B+tree leaf page of 4 KB filled with random data in main memory using LZ4 is 5-50× lower than reading the same 4 KB block from several NVMe SSDs we tested. Hence, we expect to explore this optimization tactic. Besides using compressed DRAM as the top tier, it could also form a new middle tier of block storage in between uncompressed DRAM and compressed SSD. Open questions include how to properly manage, provision, and index compressed DRAM and how to identify non-compressible data and avoid placing them in compressed DRAM; and the possible role of 2-Tree for these open questions.

7 CONCLUSION

In this paper, we proposed the 2-Tree architecture for improving memory utilization in skewed working sets. The core idea is to maintain two trees and migrate records between them: in both directions, based on hotness, and at low cost. Our results show that for skewed data, 2-Tree significantly increases buffer pool memory utilization for B+trees while maintaining competitive range scan performance. We also demonstrated that migration can be applied to LSM-tree to help increase block cache memory utilization for read-heavy workloads without hurting the performance of range scan and update operations. 2-Tree is also very competitive for indexing in Anti-Caching architecture as it significantly improves data set scalability and range scan performance. Since workloads are invariably skewed, we believe 2-Tree is a promising step towards better memory utilization in database indexing.

Acknowledgments

We thank the anonymous reviewers and Çağatay Demiralp for their insightful feedback on the early draft of this work. This research is supported by Google.

REFERENCES

[1] Antonopoulos, P., Budovski, A., Diaconu, C., Hernandez Saenz, A., Hu, J., Kodavalla, H., Kossmann, D., Lingam, S., Minhas, U. F., Prakash, N., et al. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data* (2019), pp. 1743–1756.

[2] Arulraj, J., Levandoski, J., Minhas, U. F., and Larson, P.-A. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment 11*, 5 (2018), 553–565.

[3] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *SIGMETRICS* (2012), pp. 53–64.

[4] Berg, B., Berger, D. S., McAllister, S., Grosof, I., Gunasekar, S., Lu, J., Uhlar, M., Carrig, J., Beckmann, N., Harchol-Balter, M., et al. The {CacheLib} caching engine: Design and experiences at scale. In *OSDI* (2020), pp. 753–768.

[5] Bingmann, T. STX B+ tree C++ template classes. http://panthema.net/2007/stx-btree/.

[6] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM* (1999), vol. 1, IEEE, pp. 126–134.

[7] Cao, Z., Dong, S., Vemuri, S., and Du, D. H. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *FAST* (2020), pp. 209–223.

[8] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *SoCC* (2010), pp. 143–154.

[9] Corbato, F. J. A paging experiment with the multics system. Tech. rep., MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[10] Dean, J., and Ghemawat, S. LevelDB. http://leveldb.googlecode.com.

[11] DeBrabant, J., Pavlo, A., Tu, S., Stonebraker, M., and Zdonik, S. Anti-caching: A new approach to database management system architecture. *Proc. VLDB Endow. 6*, 14 (Sept. 2013), 1942–1953.

[12] Diaconu, C., Freedman, C., Ismert, E., Larson, P.-A., Mittal, P., Stonecipher, R., Verma, N., and Zwilling, M. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD* (2013), pp. 1243–1254.

[13] Eldawy, A., Levandoski, J., and Larson, P.-Å. Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB 7*, 11 (2014), 931–942.

[14] Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems (TODS) 4*, 3 (1979), 315–344.

[15] Graefe, G. A survey of b-tree logging and recovery techniques. *ACM Transactions on Database Systems (TODS) 37*, 1 (2012), 1–35.

[16] Graefe, G., et al. Modern b-tree techniques. *Foundations and Trends® in Databases 3*, 4 (2011), 203–402.

[17] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *VLDB* (2008), pp. 1496–1499.

[18] Karnagel, T., Dementiev, R., Rajwar, R., Lai, K., Legler, T., Schlegel, B., and Lehner, W. Improving in-memory database index performance with intel® transactional synchronization extensions. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (2014), IEEE, pp. 476–487.

[19] Leis, V., Haubenschild, M., Kemper, A., and Neumann, T. Leanstore: In-memory data management beyond main memory. In *ICDE* (2018), IEEE, pp. 185–196.

[20] Leis, V., Kemper, A., and Neumann, T. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)* (2013), IEEE, pp. 38–49.

[21] Leis, V., Scheibner, F., Kemper, A., and Neumann, T. The art of practical synchronization. In *DaMoN* (2016), pp. 1–8.

[22] Levandoski, J., Lomet, D., and Sengupta, S. The Bw-Tree: A B-tree for new hardware platforms. *ICDE* (2013), 302–313.

[23] Levandoski, J., Lomet, D. B., Sengupta, S., Stutsman, R., and Wang, R. High performance transactions in deuteronomy. In *CIDR* (2015).

[24] Li, F. Cloud-native database systems at alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment 12*, 12 (2019), 2263–2272.

[25] Litwin, W. Linear hashing: a new tool for file and table addressing. In *VLDB* (1980), vol. 80, pp. 1–3.

[26] Mao, Y., Kohler, E., and Morris, R. T. Cache craftiness for fast multicore key-value storage. In *EuroSys* (2012), pp. 183–196.

[27] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS 17*, 1 (1992), 94–162.

[28] O'Neil, E. J., O'Neil, P. E., and Weikum, G. The lru-k page replacement algorithm for database disk buffering. In *SIGMOD* (1993), ACM, pp. 297–306.

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

[30] Redis. Redis. https://redis.io.

[31] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. The end of an architectural era: (it's time for a complete rewrite). In *VLDB* (2007), pp. 1150–1160.

[32] Stonebraker, M., and Weisberg, A. The voltdb main memory dbms. *IEEE Data Eng. Bull. 36*, 2 (2013), 21–27.

[33] Sundarrajan, A., Feng, M., Kasbekar, M., and Sitaraman, R. K. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *CoNEXT* (2017), pp. 55–67.

[34] Team, R. RocksDB. http://rocksdb.org/.

[35] Van Aken, D., Pavlo, A., Gordon, G. J., and Zhang, B. Automatic database management system tuning through large-scale machine learning. In *SIGMOD* (2017), pp. 1009–1024.

[36] Van Aken, D., Yang, D., Brillard, S., Fiorino, A., Zhang, B., Bilien, C., and 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.

[37] Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., and Bao, X. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), pp. 1041–1052.

[38] Wu, F., Yang, M.-H., Zhang, B., and 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.

[39] Yang, J., Yue, Y., and Rashmi, K. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *OSDI* (2020), pp. 191–208.

[40] Yu, G. X., Markakis, M., Kipf, A., Larson, P.-Å., Minhas, U. F., and Kraska, T. Treeline: an update-in-place key-value store for modern storage. *Proceedings of the VLDB Endowment 16*, 1 (2022), 99–112.

[41] Zhang, H., Andersen, D. G., Pavlo, A., Kaminsky, M., Ma, L., and Shen, R. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *SIGMOD* (2016), pp. 1567–1581.

[42] Zhou, X., Arulraj, J., Pavlo, A., and 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.