IndeXY: A Framework for Constructing Indexes Larger than Memory

Chen Zhong*‡, Qingqing Zhou† ‖, Yuxing Chen† §, Xingsheng Zhao * ‡, Kuang He† ‡‡, Anqun Pan† §, Song Jiang* ¶
* *University of Texas at Arlington, US*
† *Tencent Inc., China*
§ {axingguchen,aaronpan}@tencent.com
‖ sendtoqq@gmail.com ‡‡ korenhe@outlook.com
‡ {chen.zhong,xingsheng.zhao}@mavs.uta.edu ¶ song.jiang@uta.edu

***Abstract*—Indexes in a database system can consume a large amount of memory. When they grow too large to be entirely held in the memory, selected portions of the indexes have to be unloaded to the secondary storage. There are a number of challenges in the design of an extensible index spanning memory and disk. First, the designs of in-memory portion and on-disk portion of the index must be decoupled so that the best choice for each device can be independently made. Second, selective unloading of in-memory portion to the disk must be carefully designed to maximize chance of memory access and to produce the most disk-friendly I/O access. Third, the strategy for index reloading from the disk and retaining in the memory must be optimized for the highest memory efficiency.**

In this paper, we proposed a memory-disk-spanning index design, named IndeXY, to effectively address the challenges. IndeXY distinguishes itself by being a framework that allows separate adoption of an in-memory index design and an on-disk data organization and access scheme that are deemed most efficient to its workloads. Instead of being just another one-size-fit-all index across memory and disk, the framework provides well-designed mechanisms and policies to integrate a selected in-memory index (Index X) and an on-disk index (Index Y) into one extensible index (IndeXY). We have implemented IndeXY with alternative in-memory indexes (ART tree or B+ tree) and alternative disk indexes (LSM tree or B+ tree). As an anecdotal example, experiments show that integrating the ART tree and an LSM tree in the framework can lead to a throughput improvement by as high as an 8.6X on a TPC-C workload over LeanStore that uses B+-tree indexes in the memory and disk, and can improve performance for almost all YCSB workloads.

***Index Terms*—index, larger than memory, B-Tree**

I. Introduction

In a database system, indexes are one of the most performance-critical components for fast and high-throughput data access. For online transaction processing (OLTP) workloads, the performance of the indexes is especially important. To this end, there has been a large amount of effort on the design of high-performance indexes. One major effort is to keep all indexes in the memory, including in-memory databases supporting OLTP, such as H-Store [1], in-memory OLTP SQL server database engine [2], and MonetDB [3]. While there have been many well-studied in-memory indexes, these databases can choose their indexes best suited to their target workloads for fast in-memory data access. However, the memory demand from the indexes can be too high. For example, about 55% of the memory is occupied by the indexes in H-Store, a state-of-the-art in-memory database, when it runs TPC-C workloads [1]. One reason for the high demand is that tuples are relatively small, and a table often requires multiple indexes.

The limited memory is competed for additionally by various other types of data. Once the available memory reduces to a threshold, the operating system (OS) starts to swap pages it deems less likely to be used soon out of the memory [4]. However, the OS does not have the insider knowledge of the index usage, and it carries out the replacement at the unit of pages. While access locality exists in the key space and accordingly on the index structure, the memory page often is not an appropriate vehicle to identify and group hot (or cold) data for migration between memory and disk (e.g., a page mixed with hot and cold keys). Instead, components in an index, such as leaf nodes or subtrees, are better candidate structures on which hot/cold keys can be consistently identified.

Therefore, indexes need to span memory and disk. This may happen temporally for in-memory databases during peak load periods when the memory cannot hold them entirely, and part of the indexes has to be spilled into the disk. It may happen constantly for conventional databases to cache their indexes in the buffer cache. While the OS is not effective in maintaining an index data structure across memory and disk for performance and semantic reasons [5], the database system takes the control of the management into its own hands at the user space. When indexes can be both in the memory and on the disk and may be constantly migrated between them, the database system must organize them in both places. Unlikely tuples in a table that are stored contiguously and can be easily reached via indexes, indexes are highly structured. There are several requirements on their effective storage and access.

First, they need to be organized in a highly accessible structure. As a counterexample, their storage on the disk as a log file is not acceptable as it doesn't support direct access. This implies that index nodes in both places should be indexed in more sophisticated data structures, such as tree variants. Second, nodes can be efficiently assembled and disassembled

to move between indexes in the two places. The efficiency is often compromised by their access pattern and storage granularity. While nodes can be flexibly organized in the byte-addressable memory, they have to be assembled into the blocks (e.g., 4KB) before being unloaded to the disk. When frequently accessed (a.k.a. hot) nodes are not contiguously placed on the in-memory index, they may be assembled into different blocks, resulting in a loss of efficiency. A similar issue arises when nodes are loaded into the in-memory index from the disk. Therefore, to achieve higher performance and better compatibility, the current practice is that an index's memory and disk structures must be co-designed. Third, there exist many highly performant legacy indexes designed either only for the memory or for the disk. New and more efficient indexes for data in the memory and on the disk keep emerging. Because memory and disks have very distinct access characteristics, it is unlikely to have one index design that fits both devices. An ideal design is to adopt the best indexes for memory and disk that respectively suit the devices and their workloads.

The current practices in response to these requirements are far from satisfactory. Being aware of disadvantages of using virtual memory for data migration and on-disk data management, database designers have resorted to the co-design approach, which is to develop customized disk or memory structures by themselves. For in-memory databases, the effort is on the migration of a selected subset of data out of the memory, either to a remote memory cache, such as Memcached [6], or to local disks. For the latter option, example systems are Siberia [7] and Anti-Caching [8]. However, they are highly customized to specific in-memory database systems (Microsoft Hekaton and H-Store, respectively). They developed their on-disk data management and access schemes for tuples spilled from the memory. While the indexes may consume half or even more of the memory space, it is necessary to study how to construct extensible indexes that inherently support memory extensions. For more traditional disk-based databases, the home location of their indexes is the disk. The optimization effort is on the management of the buffer cache where portions of the indexes are cached. Using the B+-tree indexed database, LeanStore [9], as an example, one of its major goals is to make the cached portion of the on-disk index be organized more efficiently by using the pointer swizzling technique. The efficiency issue arises as LeanStore maintains a single index and node structure across the two devices. Similar to the design of virtual memory, LeanStore's index co-design that keeps one index structure into two devices of vastly different characteristics can seriously leave out important performance optimization opportunities. This issue is especially critical in the resolution of the conflict between the demand for fine granularity on hot data detection and migration and the requirement of block-size data storage and access on the disk.

The goal of this paper is to provide a framework, named IndeXY, supporting development of extensible indexes across the memory and disk. It can effectively address the aforementioned issues. First, it allows a selected existing in-memory index (Index X) and a selected existing on-disk index (Index Y) to be integrated into one index structure. IndeXY can reap the benefits of each of its component indexes that have been designed for its target device and selected for the target workloads. Second, it provides a highly efficient hotness monitoring functionality in the key space in an adaptable granularity. Third, it automates unloading/loading index nodes to/from the disk in response to memory availability.

In summary, we make multiple contributions in the paper.

- Recognizing the lack of a virtual-memory-like framework for managing key-value data in databases (rather than memory pages), we propose a new framework, named IndeXY, to enable a consistent key space across the memory and disk with an index 'swappable' out to (in from) the disk. The framework facilitates quick construction of an extensible large index with high performance.

- We design a set of techniques for efficiently monitoring and recognizing segments of the key space with a consistent access locality and accordingly unloading selected segments when memory space limit is reached.

- We evaluate IndeXY with representative in-memory and on-disk indexes and compare its performance with designs that limit themselves to one index structure. After reaching the memory limit, experiment results demonstrate that IndeXY systems employing ART-LSM and ART-B+ outperform B+-B+ systems by up to a 30X increase of throughput in various micro-benchmarks and YCSB and TCP-C benchmarks.

II. The Design of IndeXY

There are several objectives in the design of the IndeXY framework. First, it can accommodate existing indexes that were designed exclusively for memory or disk. Second, it can add hotness monitoring capability to the in-memory index with minimal efforts. Third, detection of hot regions should consider spatial locality in addition to temporal locality. The spatial locality is defined on the continuous key space. For the purpose of unloading index nodes to the disk, IndeXY should identify cold key regions, each as long as possible, for high I/O efficiency. Fourth, the time overhead (e.g., that for monitoring access patterns) and space overhead (e.g., additional memory space for recording access history) must be small. In the design, we assume order-preserving indexes. For example, hash table is not supported in the framework because major database indexes require support range search. And only when keys in the disk are sorted can they be indexed in a course-grained (e.g., blocks) manner for memory efficiency. Therefore, we describe our design on tree-like index structures.

A. The Architecture of IndeXY

Extensible index IndeXY is a framework integrating two independent indexes, namely in-memory Index X and on-disk Index Y. While it makes an in-memory index extensible to the disk and an on-disk index extensible into the memory, it fully decouples designs of the two indexes. They don't

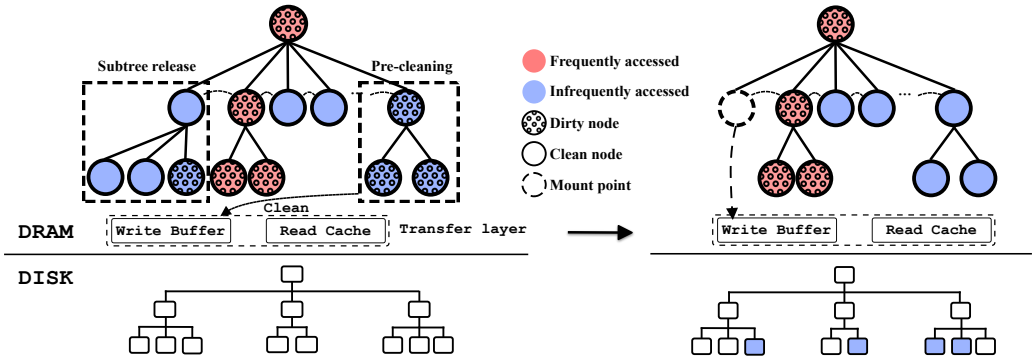


Fig. 1: The Architecture of IndeXY where its major components are illustrated. As illustrated, the subtree release and pre-cleaning operations on an index (the left graph) lead to space reclamation and a cleaned sub-tree, respectively (the right graph).

have to be of the same or similar structures for efficient data migration. Instead, their designs can focus on characteristics of their respective hosting devices. The framework is responsible for selecting and organizing keys for the migration with high efficiency. As shown in Figure 1, there is a transfer buffer layer in the memory that sits between the two indexes in the framework. The buffer is sized only to temporarily accommodate data ready for being immediately written to the disk or read from the disk. The role of actual write buffer and read cache is played by the Index X structure.

The framework's major functionalities are integrated into the in-memory Index X, including a module for proactively writing dirty keys to on-disk Index Y (the pre-cleaning module), a module for selecting and releasing index components (the subtree release module), and the module for moving keys between the two indexes (the data migration module). These modules' operations are coordinated, and carried out based on the distribution of accesses across the index and in response to current memory pressure. Unlike the virtual memory, which is a system-level memory-extension support, IndeXY is a framework to facilitate transforming user-provided indexes into highly efficient extensible indexes with ease. In the process, IndeXY aims to minimize changes to the existing indexes and make them least disruptive.

A memory size limit is set for the index managed in the framework, which monitors the index size that approaches the limit. There are two thresholds associated with the index size to control its memory usage. When the size reaches a threshold and becomes (very) close to the limit (the high watermark), the framework starts identifying the cold sub-trees for unloading to reduce its size to a lower threshold (the low watermark). The framework uses the two watermarks to minimize the memory size oscillation due to frequently triggering of index unloading.

B. Index Pre-cleaning

When the high watermark is reached, the index unloading is triggered to keep the in-memory index size from further growing and to reduce it under the low watermark. During the unloading, part of the index must be locked to prevent contention between the unloading and user-access operations. The locking for a long time period could seriously compromise the index performance. In the meantime, other part of the index can continue receiving inserts and keep growing the index. Therefore, it is important for this unloading process to be completed as quickly as possible so that the lock can be released quickly and the index size can be quickly reduced. However, this can be a challenging task. The index grows because of continuously inserted new keys. It leads to dirty data in the index and accordingly a slowdown of the unloading process due to writing of the dirty data to the disk.

The answer to this challenge in the framework is the periodic index pre-cleaning operation. It identifies dirty data in the index and then writes them back to the disk before an unloading with the hope that future unloading operation could simply discard (almost)-all-clean sub-trees. Compared to the unloading process involving the disk writes, unloading of pre-cleaned sub-trees is instantaneous. To achieve high efficiency for the pre-cleaning operation, the framework needs to avoid the key region receiving intensive inserts in its search for dirty data for two reasons. First, it is likely (some of) the inserts are overwrites of existing KV pairs in the region. Skipping the region for pre-cleaning reduces unnecessary writes to the disk. Second, we assume that an insert workload often has its spatial locality, leading to intensive updates in one key region in the index at a time (before it moves to another region). It is important to retain this locality in the write-back workload. As Index X is an order-preserving structure, Index Y is a sorted index. Writing keys in a limited range sequentially to the Index Y on the disk helps with the Index-Y's efficiency.

The pre-cleaning operation is carried out on the Index X, which is an ordered search tree. In the tree structure, each internal node represents a partition of the key space with its children covering its sub-partitions. The leaf nodes store tuples or pointers to tuples (if tuples are large) in the database. The framework adds a dirty bit to each internal node indicating if there are any inserts in the sub-tree rooted at the node. It also associates a dirty bit with each key in the leaf node. The framework relies on these bits in its search for dirty keys. The pre-cleaning operation is carried out on a key region covered by a sub-tree rooted at an inner node. Each of the key regions should be sufficiently large to accommodate a batch of dirty keys. The framework reasonably assumes that the index is a

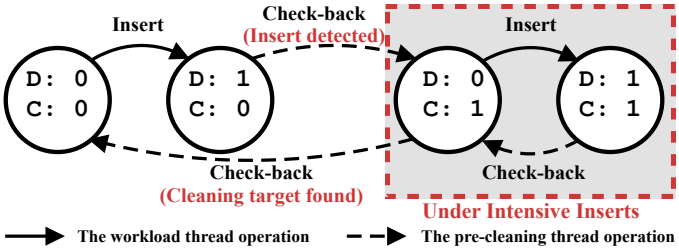


Fig. 2: Two-bit-based selection of a subtree for cleaning. The two bits ("DC") are associated with the subtree's root node. The pre-cleaning thread detects a subtree under intensive insert workload and avoids it until a "check-back" operation finds no more inserts at the node after the last scan.

balanced tree. Therefore, the inner nodes at the same level partition the entire key space into regions of about equal size. The choice of the level can be adjusted so that the key region covered by each of the inner nodes is sufficiently large to accumulate dirty keys for batching writes.

In the framework, the inner nodes are connected into a linked list. A pre-cleaning thread periodically scans the list to identify a node (the key region under the node) in which it searches for dirty keys. The thread's pre-cleaning operation is triggered by the insert requests. To this end, the framework maintains a counter that tracks number of inserted keys on the index. The counter is incremented upon a key insertion. To minimize the serialization in updating of the counter by multiple insert threads, the framework lets each of the threads keep the record of its own inserts locally without lock contention before updating the global counter (with a lock) in a much lower frequency. This optimization is acceptable as the delayed updating has little impact on the pre-cleaning thread. This counter enables a timer with a preset count of insert keys. When the timer expires, the framework activates the pre-cleaning thread to scan the list for a qualified inner node for cleaning. The timer is then reset. Note that this scan will be conducted on the list by only one pass. The next pass will start when the timer expires again.

Using the timer helps to control the pace of the cleaning operation, so that there are sufficient dirty keys for writing back. In the meantime, we propose a check-back approach to exploit the spatial locality in the insert workload. In each of the inner nodes on the list, in addition to the dirty bit (named D bit), there is a bit (named C bit) indicating if this node is a candidate for cleaning. Initially, both of the bits are 0. When the thread walks over the list and stops at a node, it checks if its D bit is 1 (dirty). If yes and its C bit is 0 (not yet a cleaning candidate), it clears its D bit and sets the C bit to 1, and continues its search for a node for cleaning. When it encounters a node whose DC bits are '11', it will simply reset its D bit back to 0 without selecting it for cleaning. When the D bit that was reset in the last pass of scan was set back to 1 by more key insertions, the framework considers the key region under the node is experiencing intensive insert workload. Accordingly, the scan skips this node in this pass and waits for it to be out of the intensive insert phase. When the thread sees a node with its DC bits of '01', the node does not receive any more insertions after the last pass (a check-back). This node is then selected for cleaning. And the thread suspends its scan on the list when this node's cleaning is completed to retain the spatial locality of the write-back operation. This check-back approach is depicted in Figure 2.

When a node is being pre-cleaned, it is locked so that access to the sub-tree rooted at the node is not accessible until the operation is completed. When the inner node list changes (removal or addition of nodes), the list will be re-constructed. Because the list is on a level not far away from the root, the list reconstruction occurs infrequently.

C. Selection of Subtrees for Release

When the size of the in-memory Index X approaches its limit, a memory release thread is activated to select subtrees for removal from the index so that its size is smaller than the low-watermark value. While the pre-cleaning thread has exploited the write locality, the release thread exploits the read locality to identify the least likely accessed subtrees in the index for release. In the meantime, it also needs to consider the subtree size in the selection. A frequently accessed subtree could be a good candidate for release if its size is large, because its release can free a large memory space and quickly bring the index size below the low watermark.

There are three challenges in the design of an effective subtree selection policy. First, there are two factors (namely, subtree access frequency and space size) that must be considered in an integrated manner. A consistent criterion is required. Second, subtrees in an index tree are not necessarily exclusive. A subtree can be part of a larger subtree and covers multiple smaller subtrees. They may not be independent of each other. Thus they cannot be simply compared against each other in the selection process. Third, to free a given amount of memory, we prefer a smaller number of larger subtrees to many small subtrees. Note that a released subtree is still logically connected to the Index X. It is physically moved to the Index Y as an extension of the Index X. On the Index X there are mount points where the released subtrees are logically connected. It is desired to minimize the number of the mount points because a search of a key that reaches a mount point needs to continue into Index Y (if it is not yet known if the key is in Index Y). This would compromise performance of workloads with many accesses to non-existent keys.

To address the challenges, we propose the concept of access density and a density-based node ranking algorithm. Any access of keys in the leaf nodes is preceded with a search on the index starting from the root. A search leaves a path from the root to a leaf node. The *access density* of a subtree rooted at an inner node is the ratio between the number of searches that have crossed the node and the number of keys in the subtree (on the Index X). This density takes both the loss (miss penalty) and benefit (space reclamation) into account. Multiple accesses can be collected along a search path (at different inner nodes). And the size of a subtree rooted at

an inner node can be estimated by counting key insertions passing through the node. The major concern on the counting method is the overhead of updating the two counters in an inner node (an access counter and an insertion counter). The framework uses two approaches to minimize the overhead. First, it consistently applies sampling to reduce the update frequency (e.g., one updating every 100 accesses at any inner node). Second, the framework has managed not to select small subtrees for release. To this end, it chooses a threshold tree level such that only inner nodes on or above it have their two counters updated. So that their corresponding subtrees are sufficiently large. The minimal granularity of index space reclamation (or the smallest subtree for release) determines the level choice. The framework decides this granularity based on the amount of memory space it normally needs to release, which can be estimated by the index memory limit and the watermarks. The framework initially only monitors the index memory size by tracking effective inserts and space release. When the index memory size reaches the low watermark for the first time and the index size approaches its limit, the framework determines its threshold level and starts to collect the statistics for selected inner nodes. The access counters will be reset after a node release.

When the framework wakes up the release thread for space reclamation, the thread first runs a density-based node ranking algorithm to select inner nodes of low access density. The ranking algorithm ranks candidate nodes in the order of their *increasing* density in a list with a target size of memory for release. Initially, only the index's root node is in the list. In each iteration, one of the node(s) in the list is chosen and replaced by its child nodes. Specifically, it scans the list from the head (with the smallest density). At each node, it tracks (1) the total size of the nodes (each for a sub-tree) it has scanned, including this node. (2) the densities of its child nodes. If the total size is smaller than the target size, the scan continues to the next node. If it reaches the target size (or larger than it with a small margin), all the nodes from the head to this one are selected for release, and the algorithm is done. Otherwise, if it is larger than the target size by more than the margin, the list is updated by a split-and-replace operation. The algorithm then starts over on the updated list. In a split-and-replace operation on a node, the node is removed from the list, and its child nodes are inserted into the list at positions determined by their respective densities. The split-and-replace node is selected in the following way. The nodes are ordered into another list according to their sizes. Starting from the largest node, we evaluate variation of its child nodes' densities. If the variation (measured as the gap between the lowest density and the highest density) is more than a threshold (20% of the parent node's density by default), it is selected as the split-and-replace node. Otherwise, the next largest node on the list is evaluated. The process continues until such a node is found or the end of the list is reached. In the latter case, the largest node is simply selected. The pseudocode is at Algorithm 1. This algorithm effectively minimizes both number of misses and number of released subtrees. The selected inner nodes for release are locked. Because of pre-cleaning the nodes are more likely to be clean and can be quickly released.

Algorithm 1: Selection of Subtrees for Release

```
1  Function SpaceReclamation(target_size):
       // select inner nodes of low access density.
2      list1 = [root]// Ordered by density
3      while True do
4          pos = 0
5          total_size = 0
6          while pos < list1.length do
7              cur_size = total_size + list1[pos].size
8              if cur_size < target_size then
9                  total_size += list1[pos].size
10             else if cur_size <= target_size + margin then
11                 release all nodes from the head to current node
12                 return
13             else
14                 list1 = SplitAndReplace(list1)
15                 break
16             pos++
17 Function SplitAndReplace(list1):
18     list2 = sort(list1)// Ordered by node size
19     sp_node = largest node in list2// split-and-replace node
20     for each node in list2 do
21         variation = VariationOfDensities(node.children)
22         if variation > threshold then
23             sp_node = node
24             break
25     replace sp_node in list1 with its child nodes and maintain the order
         according to the density
26     return list1
27
```

D. Moving Data between Indexes X and Y

After Index X reaches its size limit, data movement between Index X in the memory and Index Y on the disk takes place, including writing-back dirty data to the disk and loading keys from the disk to the memory in response to misses in the Index X. As a common practice in the design of an on-disk index, a write buffer and a read cache are set up in the memory to improve the disk access efficiency. Often the more the memory allocation to the buffer/cache is, the more effective they are. However, large buffer/cache can consume a significant amount of memory. Another issue is that it is not clear how large (relative to the Index X size) they should be set to maximize the memory efficiency, because keys in the index and the buffer/cache are separated into two data structures and their access locality cannot be consistently evaluated.

The framework addresses the issue by considering temporal locality and spatial locality separately. Exploitation of temporal locality often requires a large cache space, as a re-access of a key is likely preceded by accessing many other keys. This is why increasing cache/buffer helps reduce the miss ratio. The framework uses the Index X itself as the buffer/cache. Regarding the write buffer, the pre-cleaning operation attempts to avoid cleaning the subtrees that are being intensively written. Repeated updates on a key can be absorbed in the write buffer. Regarding the read cache, the framework inserts keys loaded from Index Y directly to the Index X. Note that these inserted keys are marked as clean ones in the index as their copies are not removed in the Index Y. Integrating Index X and the read cache makes the locality consistently compared, and the temporal locality be better exploited.

The spatial locality is also critical to the performance of accessing block devices. The framework keeps the write buffer and read cache designed with Index Y to exploit the locality. However, their sizes are configured to the minimal (e.g., a few Megabytes). For the write buffer, its main purpose is to receive and aggregate the dirty keys from the Index X to form sequences of disk writes. For the read cache, its main purpose is to keep recently loaded blocks from the disk. When a key search misses in the Index X, the key along with the other keys stored in the same block is read from the disk. Instead of also inserting these other keys immediately into the Index X, the framework leaves them in the read cache. For workloads with strong spatial locality, such as sequential read, the design effectively exploits the spatial locality without taking the risk of polluting Index X with non-accessed keys.

III. Performance Evaluation

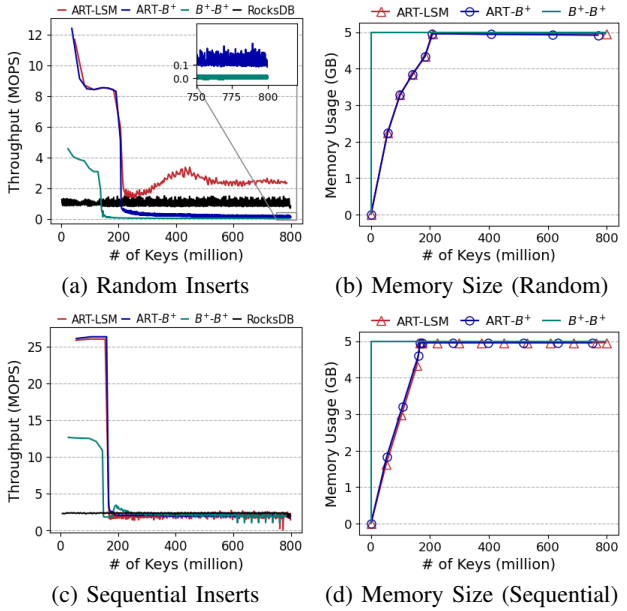
In this section, we experimentally evaluate the benefits brought by the framework. In the experiments, we choose two alternatives for Index X. One is B+ tree. And the other is the ART index [10], an adaptive radix tree that has been well recognized and used for efficient indexing in the main memory [11]–[14]. We also choose two alternatives for Index Y. One is the B+ tree for the disk, and the other is LSM tree [15]. The LSM tree is designed to optimize the insert operations on the block device by always sequentially writing a large file containing sorted key-value pairs.

A. Experiment Setup

Because B+ tree is an index that has been well studied and used across the memory and disk, we use it to represent an index design that couples its memory and disk components in the same structure. In particular, we choose LeanStore [9] and its opened source code (`git:#d3d83143`) [16], a high-performance database, in the evaluation to reveal potential limitations of the one-index-for-two-devices design. LeanStore has extensively optimized its in-memory B+-tree component as well as its B+-tree-indexed database on the disk. It adopts pointer swizzling and a low-overhead replacement technique to improve the memory access efficiency and memory hit ratio. The ART index has been shown to consistently perform better than B+ tree. Designed as a memory-only index, it doesn't come with a disk component. We integrate the index into the IndeXY framework by adding the framework's capabilities, such as pre-cleaning and subtree release, to its opened source code [17]. For this Index X, we choose either B+ tree or LSM tree as its paired Index Y. For the B+-tree Index Y, we use LeanStore with its on-disk key-value pairs organized in a B+-tree index. We minimize its buffer pool (512 MB) to serve as the write buffer and read cache of the framework for exploiting spatial locality. For the LSM-tree Index Y, we use RocksDB (`git:#v8.1.1`) [18]. Its in-memory MemTable becomes the framework's write buffer (256MB). We minimize its read cache (256 MB) for recently read blocks. The three systems are named B+-B+, ART-B+, ART-LSM, respectively. Note that B+-B+ is just the LeanStore itself. As RocksDB also has its indexes for in-memory buffer and on-disk data organization, we include it in the evaluation. Table I summarizes their compositions.

Systems	Index X	Index Y
B+-B+	B+ Index	B+ Index
ART-B+	ART Index	B+ Index
ART-LSM	ART Index	LSM-tree Index
RocksDB	RocksDB Buffer	LSM-tree Index

TABLE I: The four systems in comparison



(a) Random Inserts (b) Memory Size (Random)

(c) Sequential Inserts (d) Memory Size (Sequential)

Fig. 3: Throughput and memory sizes of the index systems under the workload of random or sequential key inserts.

In the evaluation, we first uses a set of micro-benchmarks to extensively examine the performance improvements that can be enabled by the framework. We then use the YCSB benchmark suit and the TPC-C benchmark to quantitatively evaluate the difference the framework can make on the widely used benchmarks. The experiments were conducted on a server equipped with two 24-core Intel Xeon Platinum 8255C CPU processors, with two NUMA nodes, each with 128 GB DRAM. We installed one 512 GB SAMSUNG MZ7LH480 SSD. Since our focus is not on the NUMA effect, to avoid remote memory access, all worker threads are pinned to NUMA node 0 by using `numactl`. In the evaluation of the YCSB and micro-benchmarks, we set the index size limit at 5GB and use four threads to serve queries, while setting the overall memory limit to 30GB when evaluating the TPC-C benchmark.

B. Write Performance

To observe how the systems' performance responds to intensive write, we generate inserts of distinct 8-byte keys (along with their respective 8-byte values). The keys are uniformly distributed, and are inserted either in a random order or in a sequential order. We insert 800 million keys with a total size of around 12GB. The systems' page/block size is 4KB.

Figures 3(a) and (c) show the throughput with the insertion of keys. There are several interesting observations. First, the Index X should be selected solely based on its in-memory performance. When Index X can be all held in the memory, the systems using the ART index as Index X (ART-B+ and ART-LSM) perform much better than B+-B+ (by 2.2X-3.1X). However, for a design that demands the same data structure in the two places (memory and disk) such as B+-B+, it lacks the flexibility to adopt a newly proposed high-performance in-memory index. Second, when more and more keys are inserted, the memory runs out, and the throughput drops dramatically. In comparison, ART-B+ and ART-LSM can hold more keys than B+-B+ before their memory limit is reached. This is because the ART index has a more compact structure and more efficient use of memory. One can easily take advantage of it by selecting the index as Index X in the framework. Third, the two different insert patterns also impact the systems' performance behaviors. Random inserts cause a throughput drop even before the memory limit is reached as inserts become more expensive with a larger Index X. The throughput is less sensitive to the size of the Index X with sequential inserts. The impact is more pronounced on Index Y's performance. When Index Y is an LSM tree and keys are randomly inserted, the throughput of ART-LSM is over 30X higher than that of systems choosing B+ tree as their Index Y (after the memory limit is reached). This is because LSM tree is designed to accommodate random writes with its log-style writes. In contrast, using B+ index as Index Y causes constant node split/merge, leading to significantly more disk access. Furthermore, if we look into the enlarged part in Figure 3(a), ART-B+ has a higher throughput than B+-B+ when Index Y is intensively involved in receiving keys. This is because with ART as its Index X, the framework manages to use the pre-cleaning operation to produce batched write-backs of keys in one subtree at a time. These more localized writes are more friendly to the B+-tree Index Y.

Without an Index X supported by the framework, RocksDB uses its fixed-size MemTable as the write buffer and provides a consistent write throughput. Because MemTable is much smaller than the available memory, its throughput is much lower initially. When the memory is used up, with the random inserts RocksDB has a higher throughput than the indexes using B+ as its Index Y. This is expected as RocksDB is designed to optimize random writes.

Figures 3(b) and (d) show the memory consumption of the systems. The B+-B+ system (LeanStore) allocates the memory according to its limit size at the beginning and keeps its in-memory index within the limit. The memory sizes of ART-LSM and ART-B+ are regulated by the framework. We can see that their memory size is well maintained at the limit and is very stable after the limit is reached. This suggests that the framework can responsively release memory once the limit is exceeded. Its pre-cleaning operations make clean subtrees more available, enabling quick releases.

While ART-B+ and B+-B+ organize and access their keys in pages, we vary the page size to understand its impact on the performance. Table II shows the throughput of the entire workload (inserting 800 million keys) of ART-B+ and B+-B+. With a larger page size, the throughput of B+-B+ becomes lower as the cost of node split and merge increases. In contrast, ART-B+'s effort on producing more and longer sequential writes is better translated into performance advantage with a large page size. This echos the throughput difference between sequential and random inserts shown in Figures 3(a) and (c).

Page Size	4KB	8KB	16KB
B+-B+	44.68 KOPS	33.18 KOPS	27.21 KOPS
ART-B+	313.89 KOPS	458.1 KOPS	583.96 KOPS

TABLE II: Random write throughput with different page sizes

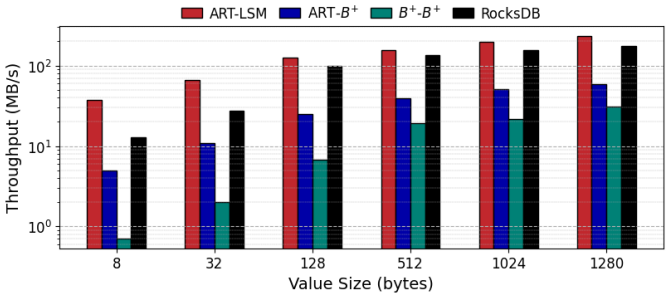


Fig. 4: Throughput with different value sizes

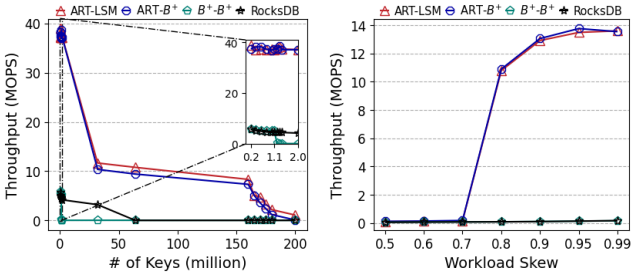


Fig. 5: Read throughput with different working set sizes

Fig. 6: Read throughput with various Zipfian distributions

To understand the impact of insert size on the performance, we increase the value size from the default value of 8B in the random insert workload. To better reveal the performance difference, we use the amount of KV data per second, rather than MOPS, as the metric to show the results in Figure 4. Understandably, with large KV sizes each leaf node stores fewer KV pairs. Fewer nodes are involved in a node split/merge, leading to lower read and write amplification. Accordingly, B+-B+ receives the largest performance increase. ART-LSM also increases its throughput but at a much smaller scale. While the framework can make the writes to the disk more sequential, using larger KV pair sizes further improves the sequentiality, which helps with the I/O efficiency and makes LSM-tree's internal compaction operation more efficient. RocksDB's behavior is similar to that of ART-LSM, as they both use RocksDB to process write requests to the disk.

C. Read Performance

For the read performance, we experiment with a key read workload that has a well-defined working set. Keys in the

working set are accessed repeatedly and uniformly in a random order. The keys are uniformly distributed in a key space containing 320 million KV pairs. In LeanStore they occupy about 16GB disk space in a B+ tree. In the experiment, we first use the read workload to warm up the memory (loading KV pairs into the Index X) before we start to collect the measurement data. Figure 5 shows the throughput of the four systems with different working sets (in terms of the number of distinct keys accessed). The system with ART tree as their Index X receives much higher throughput than B+-B+. There are two reasons. First, when the working set is very small (less than around 1.1 million keys), the 5GB memory is sufficient to keep the working set of any of the systems. Because the ART tree is an index with a higher performance than B+ tree, the throughput of ART-LSM and ART-B+ is about 7X of that of B+-B+. Second, when the working set becomes larger, only ART-LSM and ART-B+ can keep it in the memory. While B+-B+ also has 5GB memory, it stores its in-memory keys in the cached B+ pages that are generated as leaf nodes of the on-disk B+ tree. When keys are accessed sparsely within a page, to keep one or a few hot keys in a page in its Index X, B+-B+ has to keep the entire page in the memory. Due to the coupled index design, B+-B+ leaves the memory seriously underutilized. If Index X is an index designed for memory (e.g., the ART tree), this embarrassing situation is avoided. The throughput of ART-LSM and ART-B+ has a drop at around of 25 million keys in Figure 5 because the tree grows into another new level. The throughput of RocksDB is also much lower than that of ART-LSM and ART-B+, but is higher than that of B+-B+ with a smaller working set, as we enable its Row Cache with a smaller caching granularity than the block.

If Index X is considered as an in-memory extension of Index Y, the effectiveness of the in-memory Index X relies on the locality strength of accesses on Index Y. To reveal its impact, we create a read workload with skewed access on keys in the 320-million-key Index Y. The access has a Zipfian distribution. We vary its skewness parameter (S) in the distribution. The larger the S is, the more the skewness. Figure 6 shows the throughput with different S (from 0.5 to 0.99). With a smaller skewness, there are more hot keys spread out into many hot pages. The 5GB memory in any of the systems is too small to hold the hot keys. When S increases beyond 0.7, by caching the hot keys in the ART tree ART-LSM and ART-B+ become increasingly capable of capturing the working set in the memory. In contrast, B+-B+ caches hot pages, rather than hot keys. The 5GB memory is not large enough to keep the hot pages even with an S of 0.99, because of its use of page granularity for memory allocation. The framework enabling the decoupled index design frees Index X from the constraint.

D. Workload with Shifting Working Set

While the IndeXY framework supports an extensible index across the memory and disk, its Index X must selectively store KV pairs that are in the current working set. Furthermore, this selection must be updated in response to working set change

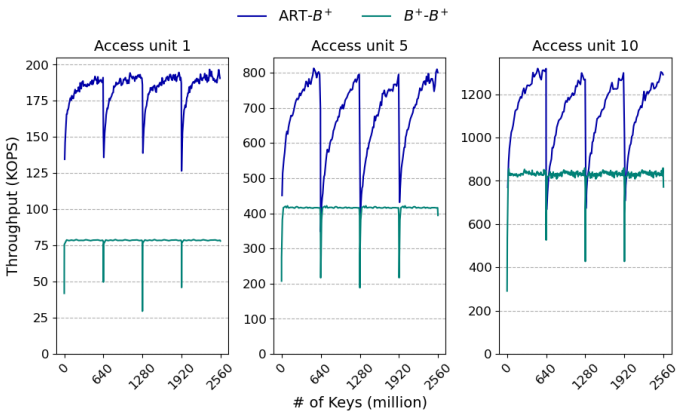


Fig. 7: Lookup performance with shifting workload (4 threads, 5 GB dataset, 5 GB buffer pool, skew 0.7)

so that the memory is always well utilized. The framework has its subtree release and data migration modules for this purpose. To evaluate its efficacy, we design a read workload with its working set shifting over the key space. Specifically, the read workload has a Zipfian distribution with a skewness factor S of 0.7 over a key space of 320 million KV pairs in Index Y. The workload has four phases, each with 640 million KV reads. After a phase, we rotate the key space by 1/4 of the space to serve the read requests. For example, a read of the key at the 1/4 of the key space in the last phase reaches the key at the beginning of the key space in this phase. Therefore, the working set keeps shifting across the key space.

Figure 7 shows the throughput of ART-B+ and B+-B+ with the workload. In the experiment, we pre-warm the memory before we start to collect throughput data. In addition, to observe the impact of spatial access locality, we change the access unit from 1 key to 5 then 10 continuous keys in one read. As shown in the figure, with the access unit of one key ART-B+ experiences a brief throughput drop at the time of entering a phase. It then quickly adapts its Index X to the new working set and recovers its throughput. When the access unit increases, there are two major differences. One is that the throughput is significantly increased (by 4.3X and 7.2X for the unit of 5 and 10 keys, respectively). This indicates that the framework's write buffer effectively exploits the spatial locality. The other is that the throughput recovery during the transition period takes a longer time. That is, the framework becomes less responsive to the working set shift and keeps its last working set in the memory for a longer time. Note that after a key space rotation, the last working set is accessed only less than the current working set. With a larger access unit, each read causes access of multiple keys in the same subtree, which immediately makes the subtree more difficult to be released. Therefore, it takes a longer time for the distinction of access density between the old and current working sets to be revealed to the subtree release thread. As established working set usually lasts for a much longer time, this longer transition period is not expected to be an issue.

In comparison, B+-B+ can almost instantly complete the

Workloads	Description
Load	100% Random write
A	50% Read, 50% Update
B	95% Read, 5% Update
C	100% Read
D	95% Read Latest, 5% Update
E	95% Scan (average scan length 50, maximum 100), 5% Update
F	50% Read-Modify-Write, 50% Read

TABLE III: Description of the YCSB Benchmarks

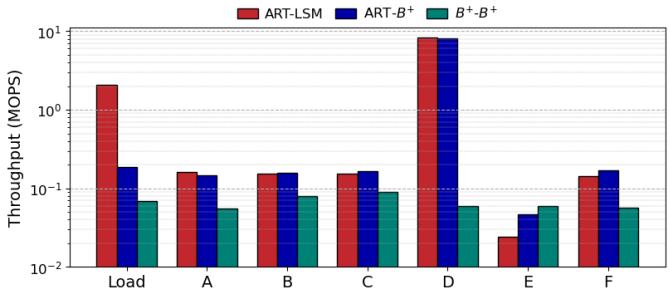


Fig. 8: Throughput of with the YCSB benchmarks

transition. It conducts its replacement consistently at the page granularity, rather than release of the subtrees of variable size. This simplistic approach is more effective for a well-regulated access pattern. However, its indexing of keys at the page granularity compromises the memory space efficiency, leading to a much lower throughput than ART-B+.

E. The YCSB Benchmarks

The YCSB suite [19] is a popular set of benchmarks to evaluate the performance of indexes and databases. We wrote a test bench for the IndeXY framework to support YCSB benchmarks, which generates Zipfian-distributed accesses with a skewness factor of 0.7 in each of the benchmarks. The initial Load phase is completed with random writes (320 million 16-byte KV pairs on the Index Y). The other benchmark (A, B,..F) starts after the "Load". Each benchmark uses four threads and sends 320 million requests. The index memory limit is still 5GB. A description of the YCSB benchmarks is in Table III.

The throughput of the three systems with the benchmarks is shown in Figure 8. Note that its Y axis is in a logarithmic scale. As shown, the Load phase reveals the largest throughput difference between ART-LSM/ART-B+ and B+-B+(over 30X). This is an observation consistent with that in Section III.B. After the memory limit is reached, the I/O operation becomes the performance bottleneck. Random writes to the B+-tree-based Index Y causes frequent read-modify-write operations on the leaf node and structural modification operations (SMO) such as leaf node split/merge and tree-rebalancing operations. All these operations lead to write amplification. The more random the writes are, the more significant the amplification is. Facilitated by the framework, ART-B+ makes the issue less serious. In ART-B+, most of writes are produced by the pre-cleaning operations on its Index X, which have made every effort to form batched writes in a limited key region. The reduced randomness makes ART-B+'s throughput improved by 3X over B+-B+.

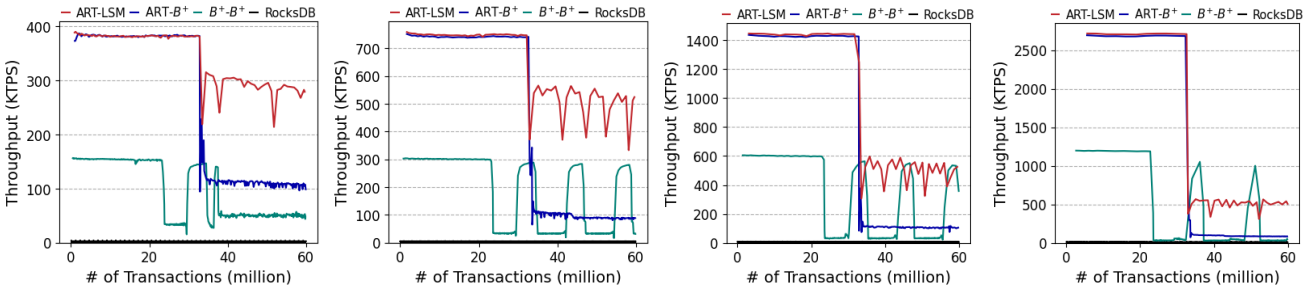
Comparing the throughput with Benchmarks A, B, and C with increasingly fewer updates, B+-B+'s throughput improves (although it is still consistently lower than that of ART-LSM and ART-B+). Because of the Zipfian updates, the writes are not sequential, leading to the write amplification. That's why Benchmark A with 50% update receives the lowest throughput. Another reason why B+-B+ has lower throughput is that it cannot keep a larger portion of the working set in the memory to serve more reads from the memory because of its use of page granularity for in-memory index organization. With 50% of the reads followed by writes, Benchmark F is another workload that compromises B+-B+'s throughput.

We have an interesting observation on Benchmark E, where ART-LSM has a substantially lower throughput. 95% of the requests in Benchmark E are scans. While none of the systems can keep its working set entirely in the memory, some of the scans must be carried out on the disk. As the LSM tree is notoriously inferior in its support of scan operation with its multi-level structure [20]–[22], ART-LSM's throughput is lower than the other two's throughput by more than 40%. As the framework provides the means and convenience for adopting various indexes as Index X or Index Y, for scan-intensive workloads one should consider other scan-friendly indexes to replace LSM-tree or select an LSM tree design optimized for scan operation [23].

Benchmark D is a benchmark that reads the latest "Load" keys (the most recent 20% of all "Load" keys). In the systems that use ART tree as their Index X, these latest keys are still held in the memory. These reads are all hits in the Index X. However, B+-B+ cannot keep them all in memory with its page-based memory data organization, leading to read misses and much lower throughput.

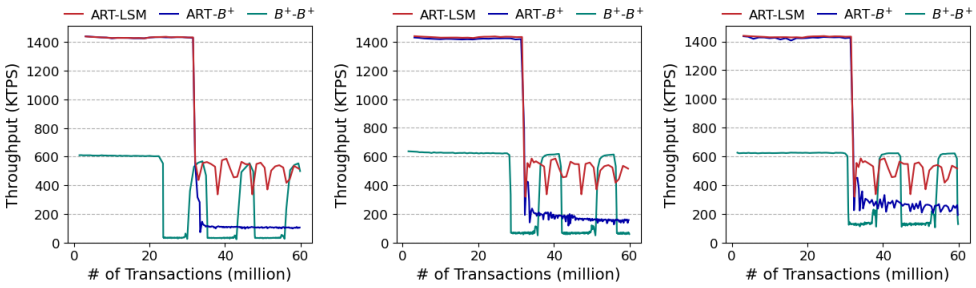
F. The TPC-C Benchmark

The TPC-C benchmark is widely used for evaluating the performance of OLTP databases [24]. It simulates a realistic mix of read, insert, update, and scan operations across various tables. We integrated the IndeXY implementation into the TPC-C engine in LeanStore's codebase and experimented with two of its five types of transactions (New Order and Payment). The New-Order and Payment transactions account for nearly 90% of the workload in TPC-C [25, Page 70]. These two transactions are sufficient to represent the characteristics of the TPC-C workload. We use 100 warehouses in TPC-C workload, which initially have around 10GB. 50% of all the transactions in the workload are from New Order and the remaining 50% are from Payment. Of the nine tables accessed in the service of the transactions the orderline table is the largest one (its index size is over 10X larger than any other table's index). Therefore, in this evaluation we make only this index to be swappable and apply the framework to it. As the framework's Index X, the index is implemented either as an ART tree or a B+ tree. We limit all the memory used by the workload to be 30GB. The orderline index keeps receiving insert requests and growing its size. When the memory limit is reached, the



(a) Two Threads (b) Four Threads (c) Eight Threads (d) Sixteen Threads

Fig. 9: Throughput of the three systems under the TPC-C workload with 100 warehouses and various number of threads



(a) Four KB Page Size (b) Eight KB Page Size (c) Sixteen KB Page Size

Fig. 10: Throughput of the three systems under TPC-C workload with different page sizes

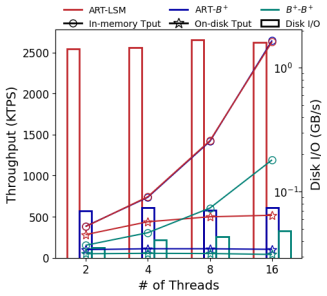


Fig. 11: Throughput and disk I/O under TPC-C workload

framework reduces orderline's Index X to keep the memory size for the workload within the limit.

In the first set of experiments, we configure the B+ tree page size of ART-B+ and B+-B+ to 4KB and vary the number of threads from 2, 4, 8, to 16. The throughput of the index systems (in terms of thousand transactions per second) is shown in Figure 9. Transactions from Payment do not access the orderline index. They only access indexes that have been kept in the memory. In contrast, transactions from the New Order intensively insert new keys into the orderline index and extend it into Index Y on the disk in ART-LSM and ART-B+. The performance difference between the systems mainly is attributed to these transactions.

Consistent to what we have observed on ART-LSM and ART-B+ under write-intensive workloads, in the first phase of the execution when memory is not yet filled, the throughput stays at a high level. When the thread number is increased, the peak throughput also increases (by around 8X when the thread count increases from 2 to 16), indicating that the CPU is not yet fully utilized. However, when the memory use reaches to its limit, the execution enters its second phase where the pre-cleaning and subtree release take place. In this phase, increasing thread count barely leads to throughput increase (the only exception is ART-LSM from 2 threads to 4 threads), as the performance bottleneck has moved to the disk. Furthermore, in this phase, the two systems behave very differently. First, ART-LSM has a much higher throughput. The inserts in the orderline Index X take place randomly in the key space. However, at each random place, there are multiple keys (around 5-15 KV pairs with a total size of 0.5KB-1KB). The ART-LSM's Index Y is resistant to the negative impact of random writes. In comparison, the ART-B+'s Index Y is sensitive even to these half-random-half-sequential writes. Second, ART-LSM's throughput fluctuates. With non-sequential writes, the compaction operations in the LSM tree index are triggered in the background, which impacts the quality of the frontend service.

The performance behavior of B+-B+ in response to the mixed workload is interesting. First, the access pattern is locally sequential and globally random. Second, it is mixed with CPU-intensive transactions from Payment and I/O-intensive transactions from New Order (due to constant write-backs). As we have indicated, B+-B+ is actually LeanStore. In the LeanStore's replacement strategy, non-accessed pages are selected as replacement candidates for writing back. The write-backs are triggered and carried out intensively once memory limit is reached to lower the memory size well below the limit. This corresponds to a time period of low throughput on the B+-B+ plots. It then takes another time period to receive inserts without immediately writing these dirty to the disk. This corresponds to the time period with small a number of writes and thus higher throughput. In comparison, ART-LSM/ART-B+ uses pre-cleaning to spread out the write-backs constantly to the entire second execution period. Accordingly, they achieve a relatively stable performance (especially for ART-B+ with the compaction operations).

The major source of performance loss with the use of B+ as the Index Y is the disk read/write amplification due to on-disk lead node split/merge. A common belief is that using a larger page size would aggravate the issue. To validate the

belief, we increase the page size from 4KB for the systems considered in Figure 9(c) to 8KB and then further to 16KB and see how the throughput of ART-B+ and B+-B+ change. The results are shown in Figures 10(b) and (c). Note that the plot for ART-LSM does not change with the page size variation as it uses LSM tree as its Index Y. Surprisingly, we find out that their throughput increases significantly. For example, the lower throughput of B+-B+ in the second phase correspondingly increases from 30 KTPS (Kilo Transactions Per Second) to 60 KPTS and then to 130 KPTS. The throughput of ART-B+ is similarly increased. We look into the write-back strategy of LeanStore (both of them use LeanStore's write buffer management), where a page that is more dirtied (containing more dirty data) is more likely to be selected for early write-back. A smaller page for the orderline index makes it easier to be filled with dirty (locally sequential) data and then evicted to the disk. Note that inserts to the index are globally random. When future inserts come back to a key space covered by a leaf node and the leaf node has been evicted to the disk, a large performance penalty (on-disk leaf node split/merge) occurs. However, with a leaf node of a larger page size, it is more likely it is still in the memory and has spare space in the page-sized node to quickly in-place accommodate the new insert keys without any overhead disk I/O. The particular access pattern of the benchmark causes this seemingly surprising outcome. The uncertainty on the impact of adjusting page size in different access patterns illustrates the value of this framework that enables flexible and decoupled index choices. O simply tuning an index (or indexing system) to adapt to workload characteristics is adequate.

In contrast, RocksDB as a data structure optimized for on-disk random writes, its in-memory performance (on its cached data) and caching efficiency are way worse than an IndeXY index. With a not-all-write TPC-C workload RocksDB's throughput is much lower. (only 2-3KTPS).

To clearly observe scalability of the indexes, we show the aggregate throughput of ART-LSM, ART-B+, and B+-B+ during the phase before the memory limit is reached (in-memory throughput) and when the limit has reached (on-disk throughput) with different thread counts at Figure 11. While the in-memory throughput scales well, their on-disk throughput does not scale. To understand the bottleneck, we measure their disk I/O throughput (see in Figure 11). With more sequential writes ART-LSM's disk throughput is higher than ART-B+'s, which is then higher than B+-B+'s. However, the disk throughput of ART-LSM and ART-B+ barely grows with number of threads. The fact that disk I/O is the performance bottleneck explains why their key-insert throughput does not scale. In contrast, the peak throughput of B+-B+ in the phase scales more. Correspondingly, Figure 11 shows that its disk throughput scales better.

G. Guidelines on Selection of Indexes X and Y

The IndeXY framework turns the design of a new extensible index over memory and disk into a process of selecting two indexes, each for one device. Still, a judicious selection could make a difference in the resulting extensible index's performance and its stability. According to our experience, here are some informative guidelines for the selection.

- For Index X, the framework expects a balanced tree structure whose inner node has an extra 2-4 unused bytes for the framework's use (e.g., D/C bits, sampled access frequency, and subtree size). Some in-memory indexes have assigned their node size and placement with the 64 cacheline [26], [27]. It is important not to break the design consideration to retain its high performance. For the sake of space and time efficiency, the framework only requires this space from the inner nodes at the higher levels.
- Indexes X and Y should be selected according to expected workloads and access requirements. For example, to support scan, hash table cannot be used. If scan is frequently used, RocksDB isn't a good choice for Index Y.
- For the Index Y candidate, it is desirable for it to come with its own write buffer and read cache, such as RocksDB's MemTable and block cache. So that the framework can directly use them as its transfer buffer.
- Index Y's selection highly depends on the expected workloads. If the workload only grows the Index X occasionally into a size larger than its limit, any on-disk index would suffice. However, if it likely receives intensive inserts to constantly exceed the limit and/or shift its working set to the cold data (that is mostly only available on the disk), one must carefully analyze its access patterns in his selection, such as read or write dominant, sequential or random, access size, and stable over time or not. The challenge is on the decision for a workload with mixed access patterns, such as random write and scan, that makes any single choice, such as LSM tree, to be suboptimal. As a future extension of the framework, we will consider the co-existence of more than one Index Y, each optimized for one access pattern. Access to different Index Xes or different key regions in an Index X is directed into the most-friendly Index Y.

IV. Related Works

It has been well aware that any on-disk index needs to be extended into the memory to take advantage of the DRAM's high speed, and any in-memory index may grow into a size larger than the available memory and spill into the disk to take advantage of its high capacity. Works in the two directions are abundant and related to the IndeXY framework.

A. Extension of On-disk Index into the Memory

Most of the on-disk data structures may simply use the OS-managed buffer cache for its extension. However, database systems usually bypass the system buffer. Instead, they manage their own buffer cache to leverage their better understanding of data access pattern in the replacement policy design [28]–[30]. Some recent works attempt to optimize the self-managed cache to better exploit the DRAM's high performance. One performance deficiency is due to a straightforward reuse of one-disk data organization in the memory [31], [32]. For

example, a pointer in an on-disk index page is a disk page address. When the page is read to the memory cache, the disk address needs to be translated into a memory page address upon each access via the pointer. To address the issue, LeanStore uses the pointer unswizzling technique to remove the indirection [9]. This is a telling example illustrating the negative impact of keeping an on-disk index organization in its in-memory presence on memory performance.

Another such example is to keep the on-disk page organization directly in the cache. When only one or a few keys in a page are hot, the entire page must be cached, compromising the memory space efficiency. This is an issue in LeanStore and in the read cache of LSM-tree KV store design (e.g., RocksDB [15]). The RocksDB addresses this issue partially in its separately managed write buffer (the MemTable), where dirty keys are organized in an in-memory index (a skip list), before they are serialized and written to the disk. This issue motivates the proposal of the 2-tree architecture where hot keys are organized into a memory index at the key granularity for high memory efficiency [33]. Another effort in this direction is the AC-Key that maintains a KV cache in addition to the existing block cache in RocksDB [34]. However, while these are timely efforts in the right direction, without a systematic solution their designs leave many questions unanswered. For example, the 2-tree design simply transforms a block-based replacement strategy into a record-based (or key-based) strategy. This requires tracking access to each individual key and scanning a very large number of keys for replacement. This is likely not affordable. By simultaneously maintaining a KV cache and a block cache, AC-Key faces the challenge of allocating cache space between them. Furthermore, often neither individual keys nor blocks (pages) are the right granularity to batch the writing back with an ever-changing access pattern. Realizing the issue and challenge, Umbra uses variable-size pages to organize data in the buffer manager [35]. However, deciding how to vary the page size itself becomes a problem.

In contrast, the IndeXY framework uses an inner node (or subtree) covering a set of keys in a local key region. Its size is variable adapting to the current access pattern. The access tracking and replacement design are also carried out at the dynamically adjustable granularity. Additionally, as a framework, this solution provides more flexibility and a larger space for further expansion and optimization.

B. Extension of In-memory Index into the Disk

In-memory databases have become popular in recent years in pursuit of the highest performance possible for a database system [36]–[38]. However, these systems are mostly designed by assuming the systems will not run out of memory. They provide warning to their users asking them to keep their database well below the available memory [39], [40]. Otherwise, the virtual memory would starting swapping pages to the disk even leading to page thrashing. There have been works to add a new component to the databases so that the out-of-memory situation can be more effectively handled.

While data in the in-memory databases are usually managed in granularity that is much smaller than page, a major effort to address is on how to track and identify small data items in a lightweight way for moving them to the disk. Anti-Caching is such a design [8]. It manages hot memory resident data at the tuple granularity. To this end, each relation table has an LRU list across the tuples. Each tuple has additional 8 bytes as space overhead for recording access history, which is too expensive. Accordingly, it doesn't support out-of-memory index and all indexes must be fully in the memory. Indexes in an in-memory database may consume over 50% of the memory [41]. Siberia [7], a component of the Hekaton database [42], keeps hot tuples in the in-memory hot store, and cold data in a traditional page-based on-disk cold store. Knowing that it can be prohibitive for tracking hot data at a small granularity, it resorts to offline access log analysis. However, it cannot quickly respond to access pattern changes. In contrast, IndeXY represents the first effort to develop a generic framework supporting index integration across memory and disk.

A variant of VoltDB has attempted to leverage OS's virtual memory to support large-than-memory databases [43]. It moves hot tuples to memory-pinned pages and cold tuples to the unpinned pages. However, it still needs to always track access at the tuple granularity. Reorganizing data in different pages is not a straightforward task. Aware of the gap of data representation granularity between memory and disk, Ma et al. consider introduction of a hierarchy of storage device (DRAM, NVRAM, 3D XPoint SSD,and SSD) with different access granularity and tailor the data migration strategy for each storage device [44]. The IndeXY framework addresses the challenge of tracking access with small data granularity by using access-pattern-aware size-variable subtrees as the tracking objects. It doesn't need additional hardware support and provide a framework for flexibility and future expansion.

V. Conclusion

In this paper, we propose a framework that allows two indexes, one designed for memory and the other for the disk, to work together as one integrated extensible index across the memory and disk. Compared to other efforts on making an index extensible across the boundary of memory and disk, the advantage of the framework comes from two of its features, which are decoupling and integration. Decoupling the design of in-memory Index X and on-disk Index Y gives the index designers the freedom of index choice and space for device-customized optimization. Integration support for Indexes X and Y, including framework built-in capabilities of light-weight access monitoring, identifying subtrees for write-backs and memory release, gives the designers the convenience to obtain a well-coordinated extensible index.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was partially supported by Tencent America LLC. with a generous gift fund.

REFERENCES

[1] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: A high-performance, distributed main memory transaction processing system," *Proc. VLDB Endow.*, vol. 1, no. 2, p. 1496–1499, aug 2008. [Online]. Available: https://doi.org/10.14778/1454159.1454211

[2] Wikimedia. (2002) Microsoft sql server. https://en.wikipedia.org/wiki/Microsoft_SQL_Server.

[3] ——. (2005) Monetdb. https://en.wikipedia.org/wiki/MonetDB.

[4] R. Stoica and A. Ailamaki, "Enabling efficient OS paging for main-memory OLTP databases," in *Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN 2013, New York, NY, USA, June 24, 2013*, R. Johnson and A. Kemper, Eds. ACM, 2013, p. 7. [Online]. Available: https://doi.org/10.1145/2485278.2485285

[5] M. Stonebraker, "Operating system support for database management," *Commun. ACM*, vol. 24, no. 7, pp. 412–418, 1981. [Online]. Available: https://doi.org/10.1145/358699.358703

[6] B. Fitzpatrick. (2004) Distributed caching with memcached. https://www.linuxjournal.com/article/7451.

[7] A. Eldawy, J. Levandoski, and P. Larson, "Trekking through siberia: Managing cold data in a memory-optimized database," in *International Conference on Very Large Databases (PVLDB Vol. 7, Issue. 11), June 2014*. VLDB - Very Large Data Bases, September 2014.

[8] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik, "Anti-caching: A new approach to database management system architecture," *Proc. VLDB Endow.*, vol. 6, no. 14, p. 1942–1953, sep 2013. [Online]. Available: https://doi.org/10.14778/2556549.2556575

[9] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann, "Leanstore: In-memory data management beyond main memory," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018, pp. 185–196. [Online]. Available: https://doi.org/10.1109/ICDE.2018.00026

[10] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, C. S. Jensen, C. M. Jermaine, and X. Zhou, Eds. IEEE Computer Society, 2013, pp. 38–49. [Online]. Available: https://doi.org/10.1109/ICDE.2013.6544812

[11] D. De Leo and P. Boncz, "Packed memory arrays - rewired," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 830–841.

[12] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis, "Hot: A height optimized trie index for main-memory database systems," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 521–534. [Online]. Available: https://doi.org/10.1145/3183713.3196896

[13] V. Alvarez, S. Richter, X. Chen, and J. Dittrich, "A comparison of adaptive radix trees and hash tables," in *2015 IEEE 31st International Conference on Data Engineering*, 2015, pp. 1227–1238.

[14] Z. Xie, Q. Cai, G. Chen, R. Mao, and M. Zhang, "A comprehensive performance evaluation of modern in-memory indices," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 641–652.

[15] M. D. E. Team, "Rocksdb," 2018. [Online]. Available: https://rocksdb.org

[16] A. Alhomssi and Demian, "Leanstore," 2018. [Online]. Available: https://github.com/leanstore/leanstore

[17] F. Scheibner, C. Lou, and M. Markakis, "Leanstore," 2018. [Online]. Available: https://github.com/flode/ARTSynchronized

[18] M. D. E. Team, "Rocksdb," 2018. [Online]. Available: https://github.com/facebook/rocksdb/tree/v8.1.1

[19] M. Mutsuzaki, "Ycsb," 2018. [Online]. Available: https://github.com/brianfrankcooper/YCSB/

[20] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 133–148. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

[21] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "HashKV: Enabling efficient updates in KV storage via hashing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 1007–1019. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/chan

[22] A. Papagiannis, G. Saloustros, G. Xanthakis, G. Kalaentzis, P. Gonzalez-Ferez, and A. Bilas, "Kreon: An efficient memory-mapped key-value store for flash storage," *ACM Trans. Storage*, vol. 17, no. 1, jan 2021. [Online]. Available: https://doi.org/10.1145/3418414

[23] W. Zhong, C. Chen, X. Wu, and S. Jiang, "REMIX: Efficient range query for LSM-trees," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 51–64. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/zhong

[24] T. P. P. Council, "$tpc - c\ \ home$," 2018. [Online]. Available: https://www.tpc.org/tpcc/

[25] ——, "$tpc\ \ benchmark\ \ c$," 2010. [Online]. Available: https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[26] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, P. Felber, F. Bellosa, and H. Bos, Eds. ACM, 2012, pp. 183–196. [Online]. Available: https://doi.org/10.1145/2168836.2168855

[27] X. Zhao, C. Zhong, and S. Jiang, "Turbohash: A hash table for key-value store on persistent memory," in *Proceedings of the 16th ACM International Conference on Systems and Storage, SYSTOR 2023, Haifa, Israel, June 5-7, 2023*, Y. Moatti, O. Biran, Y. Gilad, and D. Kostic, Eds. ACM, 2023, pp. 35–48. [Online]. Available: https://doi.org/10.1145/3579370.3594766

[28] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The bw-tree: A b-tree for new hardware platforms," in *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, C. S. Jensen, C. M. Jermaine, and X. Zhou, Eds. IEEE Computer Society, 2013, pp. 302–313. [Online]. Available: https://doi.org/10.1109/ICDE.2013.6544834

[29] ——, "LLAMA: A cache/storage subsystem for modern hardware," *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 877–888, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p877-levandoski.pdf

[30] H. Kimura, "FOEDUS: OLTP engine for a thousand cores and NVRAM," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 691–706. [Online]. Available: https://doi.org/10.1145/2723372.2746480

[31] A. Kemper and D. Kossmann, "Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis," *VLDB J.*, vol. 4, no. 3, pp. 519–566, 1995. [Online]. Available: http://www.vldb.org/journal/VLDBJ4/P519.pdf

[32] S. J. White and D. J. DeWitt, "Quickstore: A high performance mapped object store," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*, R. T. Snodgrass and M. Winslett, Eds. ACM Press, 1994, pp. 395–406. [Online]. Available: https://doi.org/10.1145/191839.191919

[33] X. Zhou, X. Yu, G. Graefe, and M. Stonebraker, "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, The Netherlands, January 8-11, 2023*. www.cidrdb.org, 2023. [Online]. Available: https://www.cidrdb.org/cidr2023/papers/p57-zhou.pdf

[34] F. Wu, M.-H. Yang, B. Zhang, and D. H. Du, "AC-Key: Adaptive caching for LSM-based Key-Value stores," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 603–615. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/wu-fenggang

[35] T. Neumann and M. J. Freitag, "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*. www.cidrdb.org, 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[36] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA database – an architecture overview," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012. [Online]. Available: http://sites.computer.org/debull/A12mar/hana.pdf

[37] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *2011 IEEE 27th International Conference on Data Engineering*, 2011, pp. 195–206.

[38] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 18–32. [Online]. Available: https://doi.org/10.1145/2517349.2522713

[39] Oracle, "Oracle timesten products and technologies," 2008. [Online]. Available: https://www.oracle.com/technetwork/products/timesten/tt70-wp-timesten-tech-133125.pdf

[40] M. Colgan, "Oracle database in-memory population," 2019. [Online]. Available: https://blogs.oracle.com/in-memory/post/oracle-database-in-memory-population

[41] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory oltp databases with hybrid indexes," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1567–1581. [Online]. Available: https://doi.org/10.1145/2882903.2915222

[42] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: SQL server's memory-optimized OLTP engine," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, K. A. Ross, D. Srivastava, and D. Papadias, Eds. ACM, 2013, pp. 1243–1254. [Online]. Available: https://doi.org/10.1145/2463676.2463710

[43] R. Stoica and A. Ailamaki, "Enabling efficient os paging for main-memory oltp databases," in *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, ser. DaMoN '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2485278.2485285

[44] L. Ma, J. Arulraj, S. Zhao, A. Pavlo, S. R. Dulloor, M. J. Giardino, J. Parkhurst, J. L. Gardner, K. Doshi, and S. Zdonik, "Larger-than-memory data management on modern storage hardware for in-memory oltp database systems," in *Proceedings of the 12th International Workshop on Data Management on New Hardware*, ser. DaMoN '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2933349.2933358