

# DIndex: An Efficient On-Disk Learned Index for Memory-Constrained Environments

Jiahuan Shen\*, Chuzhe Tang\*, Haoning Lan\*, Ren Ren<sup>†</sup>, Zhaoguo Wang\*

\*Institute of Parallel and Distributed Systems, School of Computer Science, Shanghai Jiao Tong University

<sup>†</sup>Huawei Technologies Co., Ltd.

**Abstract**—B+trees, the de facto standard for on-disk database indexing, exhibit significant performance degradation in memory-constrained environments due to frequent disk I/O that accesses out-of-cache index nodes during tree traversal. The recent proposal of in-memory learned indexes promises substantial reductions in index footprint by replacing traditional hierarchical structures with compact learned models. However, they require sorted data layouts for accurate lookups, which are costly to maintain on disks, especially in the presence of writes. So far, attempts to adapt learned indexes for on-disk use have yet to fully realize the performance benefits of learned models: they either incur massive disruptive data movement to maintain data layout or introduce significant metadata overhead that negates the space advantage of learned models.

We present DIndex, a new on-disk learned index tailored for memory-constrained environments. DIndex efficiently supports write operations while maintaining high read performance and low index size. Key to DIndex’s design is the decoupling of logical ordering from physical data layout and a set of techniques that realize this decoupling with small metadata overhead. Specifically, DIndex proposes to maintain *model-compressed pivot keys* to guide error correction and a mapping table that associates data blocks’ physical locations using an *adaptive, variable-span mapping scheme*. Experiments show that DIndex outperforms B+tree and other learned indexes by up to 28 $\times$  in throughput and reduces index size by up to 99%.

**Index Terms**—Index structures, learned indexes, on-disk indexes.

## I. INTRODUCTION

The B+tree [1], [2], introduced over half a century ago, continues to be the dominant indexing structure in on-disk databases such as MySQL [3], PostgreSQL [4], and SQLite [5]. Ideally, with sufficient memory, all internal nodes of a B+tree should be cached in memory and a lookup operation requires only one disk I/O operation to fetch the leaf node. However, with the advent of large-dataset workloads and low-cost edge devices, memory-constrained environments are becoming increasingly prevalent. Prior work suggests that production databases nowadays can even work with memory-to-disk ratios that are less than 1:100 [6], making it often impossible to cache all internal nodes of a B+tree.

In memory-constrained environments, there can be significant performance degradation for B+tree lookups when the internal nodes are not fully cached. As Figure 1 shows, B+tree shows up to a 45% drop in throughput when the memory-to-disk ratio is reduced from 1/10 to 1/300. This drop is mainly due to the additional I/O operations incurred by accessing internal nodes. We refer to the total size of the internal nodes

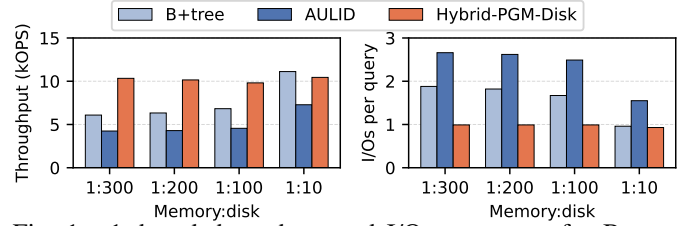


Fig. 1: 1-thread throughput and I/Os per query for B+tree, AULID, and Hybrid-PGM-Disk (fb dataset, 64-byte records, read-only workload).

as the index size in this paper. Even with fast storage like SSDs, I/O operations still account for >95% of the total search time. As a result, a single additional I/O operation can nearly double the search latency and reduce the throughput by half.

Recently, the learned index [7] and its variants [8], [9], [10], [11], [12], [13] have emerged, showing the potential of applying machine learning (ML) techniques to reduce index memory footprint. Notably, a two-orders-of-magnitude index size reduction compared to B+tree has been reported [7]. Unlike B+tree, which uses a hierarchical structure to locate a target key, a learned index uses learned models to directly predict the data location based on the query key. This approach eliminates the need for internal node hierarchies and requires only a small set of learned models, e.g., 150kB models for 200 million records [7], significantly reducing the index size. Therefore, although originally designed for in-memory use, learned indexes present an opportunity to avoid the I/O operations of on-disk B+trees in memory-constrained environments.

However, due to the performance disparity between disk I/O and in-memory operations, adapting the learned index for on-disk use faces one major challenge: *maintaining sorted data layout in the presence of write operations*. The model’s predicted location may not be the exact location of the target key due to the model’s error. Such inaccuracy necessitates a search procedure (sometimes called the last-mile search [14]) to pinpoint the exact location of the target key. Efficient execution of this search, typically using algorithms like binary search around the predicted location, requires the data to be contiguously sorted. Therefore, when naively porting learned indexes to disk, write operations would cause massive disk data movement to maintain sorted order, which would be prohibitively expensive even for MB-scale datasets.

Prior work has been proposed to support writes in on-disk learned indexes. However, new issues are introduced that

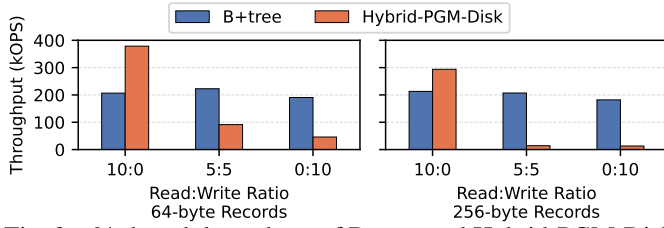


Fig. 2: 64-thread throughput of B+tree and Hybrid-PGM-Disk (fb dataset).

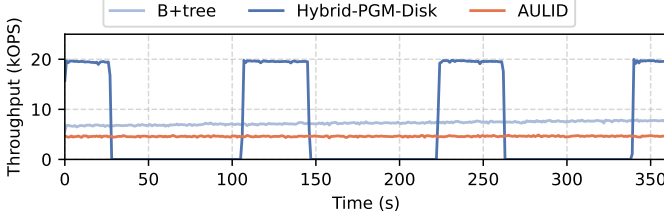


Fig. 3: 1-thread throughput over time (fb dataset, 64-byte records, read:write = 5:5).

cause significant performance degradation, in some cases even worse than a B+tree. Hybrid-PGM-Disk [13] is an on-disk learned index that targets SSD environments. It adopts designs from in-memory updatable learned indexes to use a dynamic index to handle writes. This dynamic index buffers writes and is periodically merged with the main index. However, these merge operations are costly: they require reading the entire on-disk index, integrating the buffered writes from the dynamic index, and rewriting the combined data to disk. Consequently, the performance of foreground reads and writes is significantly degraded during the merge process. This leads to low overall throughput up to  $26\times$  worse than a B+tree (Figure 2), and high performance fluctuations (Figure 3).

Alternatively, AULID [12] avoids the costly operations by partitioning the data layout into many small 4kB pages, such that data ordering is only maintained within individual pages. Since the underlying learned model still requires a globally sorted view of the data, AULID employs a layer of indirection to map the sorted logical order to scattered physical page locations. A drawback of scattering logically adjacent pages is the potential amplification of lookup costs when a search might access multiple such pages. To mitigate this, AULID stores a pivot key for each page along with its mapping entry, allowing unnecessary page lookups to be pruned by comparing the query key against the pivot keys. While AULID effectively avoids the write overhead and performance fluctuations caused by merging dynamic indexes, its additional metadata (pivot keys and mapping tables) incurs significant memory overhead. Figure 1 reveals that AULID is even 30% slower than B+tree in memory-constrained environments due to additional I/O operations to read its metadata.

This paper proposes DIndex, an on-disk learned index that efficiently supports write operations with no merge-related performance degradation and a small index size. Similar to AULID, DIndex utilizes additional metadata to localize data movement and avoid unnecessary page access during lookup.

However, instead of the plain pivot key representation and fixed-span mapping scheme, DIndex reduces the metadata size with the following two key design choices. First, we observe that only the relative order between pivot keys and the query key is required during error correction. For instance, given pages  $i$  and  $i+1$  with pivot keys  $p_i$  and  $p_{i+1}$ , determining if query key  $q$  falls into page  $i$  only requires checking if  $p_i \leq q < p_{i+1}$ , and the exact numerical difference is not necessary. Therefore, DIndex employs an order-preserving compressor based on learned models to map pivot keys to compact representations as the metadata. Second, we observe that a mapping entry does not necessarily need to have a fixed span, i.e., pointing to only one on-disk page. For example, if multiple logically adjacent pages are stored contiguously on disk, a single mapping entry can provide the mapping information for all these pages. Therefore, DIndex employs a variable-span mapping scheme and introduces runtime mechanisms to identify data regions with infrequent insertions and reorganize their physical layout to exploit this space-saving opportunity.

We evaluated the performance of DIndex against B+tree and state-of-the-art learned index baselines including Hybrid-PGM-Disk and AULID. DIndex achieves up to  $28\times$  the throughput of existing indexes and reduces index size by up to 99%. We have also integrated DIndex into MySQL, showing that it can achieve up to  $2.1\times$  the performance of InnoDB in the TPC-C benchmark [15]. DIndex is open-sourced at <https://ipads.se.sjtu.edu.cn:1312/opensource/dindex>.

In summary, our work makes the following contributions:

- We analyze the challenges of adapting learned indexes for on-disk usage in memory-constrained environments, and identify the limitations of existing solutions.
- We propose DIndex, an on-disk learned index that uses model-based pivot key compression and a variable-span mapping scheme to efficiently handle writes while maintaining high read performance and low index size.
- We implement and evaluate DIndex, showing that DIndex achieves up to  $28\times$  higher throughput than other learned indexes, and outperforms InnoDB by up to  $2.1\times$  when integrated into MySQL.

## II. BACKGROUND AND MOTIVATION

### A. The Learned Index

Recent work on the learned index [7], [8], [10], [9], [11] provides a new way to build fast and space-efficient indexes. Learned indexes use ML models (e.g., linear regression) to accelerate the routing procedure from a key to its position during index lookup. Specifically, given a sorted array of key-value pairs, the model is trained with  $(key, pos)$  pairs to learn the position of each key in the array. Since models may have prediction errors, to guarantee correctness, the learned index bounds the search space by recording the maximum error in both directions. It stores the maximum negative error  $max\_neg\_err$  and the maximum positive error  $max\_pos\_err$ , which represent the largest distances the model’s prediction can be from the actual position. Therefore, a lookup involves two steps. First, the trained model

TABLE I: Comparative analysis of index structures. Read I/O complexity refers to the asymptotic growth of I/O operations relative to data size  $n$ .

| Index Structure | Index Size | Write Handling | Read I/O Complexity |
|-----------------|------------|----------------|---------------------|
| B+tree          | High       | In-place       | $O(\log n)$         |
| Hybrid-PGM-Disk | Low        | Buffer + merge | $O(1)$              |
| AULID           | High       | In-place       | $O(\log n)$         |
| DIndex          | Low        | In-place       | $O(1)$              |

predicts a position  $pos\_pred$ . Second, it performs a search within the error range  $[pos\_pred - max\_neg\_err, pos\_pred + max\_pos\_err]$  to find the target record.

By replacing the hierarchical tree structure with an ML model, the learned index can significantly reduce the index size compared with B+tree. Ideally, even in memory-constrained environments, all models could be cached in memory, incurring no index-related I/O cost during lookups.

### B. Existing on-Disk Learned Indexes

The original learned index was designed as an in-memory index, and researchers have since proposed to adapt it for on-disk usage, including Hybrid-PGM-Disk [13], AULID [12], TreeLine [16] and FILM [17]. In this section, we analyze Hybrid-PGM-Disk and AULID as two representative solutions. Table I presents a comparative analysis of these indexes.

1) *Hybrid-PGM-Disk*: Figure 5(a) shows the architecture of Hybrid-PGM-Disk. It stores key-value pairs as a sorted array within a file and employs a two-layer PGM model to predict the file offset for a given query key. To mitigate the overhead associated with prediction errors, Hybrid-PGM-Disk performs a single large I/O operation to fetch all data blocks within the predicted range. This approach is more efficient than fetching each data block individually, as modern storage devices like SSDs support parallel reads. The read I/O complexity of Hybrid-PGM-Disk is  $O(1)$  as it has a constant-bounded error range irrelevant of the data size. Since the sorted array structure makes insertions difficult, Hybrid-PGM-Disk incorporates a dynamic index to buffer write operations. This dynamic index is limited to a maximum size (defined by a preset ratio relative to the on-disk data size) and is merged with the static index when this limit is reached.

Unfortunately, the periodic merge operation introduces significant overhead. A merge requires reading all data from both the dynamic and static indexes, sorting the combined data, and writing the result back to disk. During this process, the index becomes almost unavailable for new requests because the merge operation saturates disk bandwidth. The cost of this merge is proportional to the total data size (i.e., the number of records multiplied by the record size). As Figure 4 demonstrates, the merge duration increases dramatically with larger datasets (e.g., from 1s with 15M records to 10s with 150M records) and larger record sizes (e.g., from 10s with 64-byte records to 40s with 256-byte records). Data sizes of this scale are representative of practical workloads. For instance, the YCSB [18] benchmark uses 120M records with 1kB record size, and TPC-C [15] includes tables with record sizes ranging from 32 to 596 bytes.

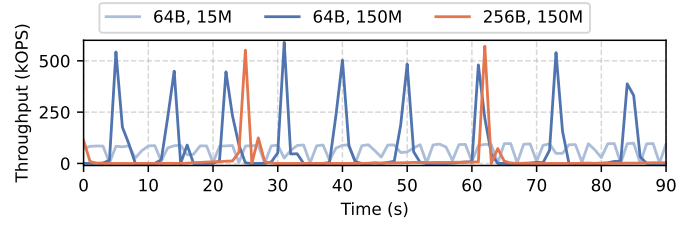


Fig. 4: 64-thread throughput of Hybrid-PGM-Disk over time (f1b dataset, read:write = 5:5). B=bytes per record; M=million records.

Key-value separation, employed by systems such as WisckKey [19] and HashKV [20], offers a potential solution to reduce the merge cost when dealing with large records. This technique stores values separately in a value log, leaving the index structure to manage only keys and pointers to value locations. Nevertheless, this separation introduces additional I/O overhead during read operations. Each read operation requires first looking up the index to find the value’s location, and then performing a second read from the value log to retrieve the actual data. According to our experiments, Hybrid-PGM-Disk with key-value separation achieves only 75% of B+tree’s throughput in read-only workloads, essentially negating the read performance advantages expected from the learned index.

2) *AULID*: Figure 5(b) shows the architecture of AULID. It organizes its index as a tree consisting of different types of nodes. Key-value pairs are stored in 4kB data blocks. Its read I/O complexity is  $O(\log n)$  as it has an  $O(\log n)$  tree height due to its use of the FMCD algorithm in LIPP [21], where  $n$  is the dataset size. To reduce update costs, AULID stores these leaf nodes unordered on disk and maintains a mapping (i.e., pointers) from logical leaf node IDs to their physical disk locations. Thus, when an insertion splits a full leaf node, only half its data needs to be moved to a new node, followed by an update to the mapping table. Compared with Hybrid-PGM-Disk, this design moves less data. However, storing leaf nodes unordered increases misprediction penalties, as locating the target record may require multiple random page accesses. To mitigate this, AULID maintains a pivot key for each leaf node, allowing for mispredictions to be corrected with pivot key comparisons before reading pages from disk.

This approach, however, sacrifices one of the primary advantages of learned indexes, the reduced memory footprint. The additional metadata for each data block results in an index size comparable to or even larger than a B+tree, effectively eliminating any memory efficiency benefits. We compare the performance and the number of I/O operations per read operation of B+tree and AULID in Figure 1. We also include the results of Hybrid-PGM-Disk in the figure for reference. In this read-only workload, Hybrid-PGM-Disk does not suffer from the merge overhead, so it achieves the highest throughput with the number of I/Os per query close to 1. However, due to the high memory usage of AULID, it performs even worse than the B+tree, with up to 35% performance degradation.

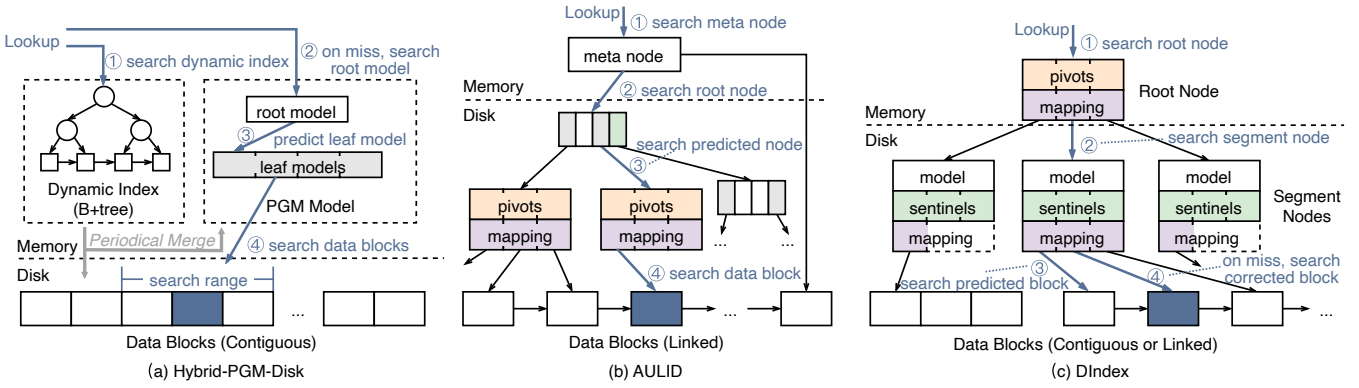


Fig. 5: Architectures of different on-disk learned indexes. The blue arrows show the lookup workflow for each index structure. Components in the memory section are always pinned in memory, while components in the disk section are stored on disk and may be cached in memory if possible.

### C. Observations

The following two key observations enable DIndex to reduce the memory overhead of AULID while avoiding the merge overhead of Hybrid-PGM-Disk.

*Correcting mispredictions does not require full pivot keys.* While pivot keys help correct prediction errors, it is not necessary to store the full pivot keys as the metadata. The core routing decision, i.e., selecting the correct child node for a query key, depends only on the relative order of the query and pivot keys, not their absolute values. Consequently, order-preserving compression can be utilized to reduce the space overhead of the pivot keys.

Interestingly, we find that the ML model can serve as such a compressor, provided the model guarantees monotonicity. The model takes a key as input and predicts its logical position. We can then scale the model output to a smaller integer than the original key to achieve compression. Compressing an 8-byte key into a 4-byte integer in this way achieves a 50% compression rate. Compared to traditional order-preserving compressors such as null suppression and delta encoding [22], the ML model offers a deterministic compression ratio.

*Fixed-span mapping entries can be redundant.* The mapping table enables leaf node retrieval by logical ID, decoupling the logical node order from physical storage locations and thus permitting non-contiguous disk layouts. However, we observe that maintaining per-leaf mapping entries can be redundant if a span of leaf nodes is reorganized to be sorted and stored contiguously on disk. In such cases, a single mapping entry pointing to the start of the span suffices, and the physical locations of subsequent nodes can be calculated using offsets from the first node's location. If a subsequent insert operation requires a leaf node split within this contiguous span, the per-leaf mapping can be rebuilt to avoid data movement. Therefore, variable-span mapping entries give us an opportunity to reduce the mapping table size at the cost of data movement during reorganization.

Furthermore, we find that access patterns are helpful in reducing the performance impact of reorganization. Specifically, for a range of leaf nodes that are read-only, or only experience in-place updates that do not cause node splits, the

reorganization would be beneficial since the per-leaf mapping may not be rebuilt shortly. This type of access pattern is prevalent in many OLTP workloads. For instance, in relational tables with auto-increment primary keys, new records are always appended to the last leaf node, ensuring that previous leaf nodes remain unchanged and can be stored contiguously.

## III. DINDEX DESIGN AND IMPLEMENTATION

### A. Key Designs

*1) Model-Based Pivot Key Compression for Error Handling:* DIndex uses the ML model to map pivot keys to compact integer representations, which we term *sentinels*, and store the sentinels to correct mispredictions. The model first predicts a logical position for a key, and this position is then scaled to the entire range of a smaller integer space. However, the model can thus only guarantee a non-decreasing property due to the loss of resolution when mapping a large key space to a smaller integer space: for any two keys  $k_1 < k_2$ , the model's scaled output  $f$  ensures  $f(k_1) \leq f(k_2)$ , but not the strict inequality  $f(k_1) < f(k_2)$ . This limitation implies that comparing the scaled output  $f(q)$  for a query key  $q$  against stored sentinels might fail to pinpoint the correct data block. For instance, if  $q$  belongs to the  $i$ -th leaf node ( $p_i \leq q < p_{i+1}$ ), it is possible that the scaling process results in  $f(p_i) = f(q) = f(p_{i+1})$ . In such cases, it is impossible to determine from the sentinels whether  $q$  belongs to the  $i$ -th or the  $(i+1)$ -th leaf node. We term this issue *sentinel collision*. We handle such collisions in two ways. First, we design the learned model to reduce sentinel collisions. Second, if a collision happens, DIndex has a verification step to check if  $q$  lies within the range of the candidate block. If not, DIndex redirects the search to the correct, adjacent block.

*2) Variable-Span Mapping Scheme for Write Handling:* DIndex uses a variable-span mapping scheme where some mapping entries point to a single data block while others point to a contiguous range of data blocks. As shown in Figure 6, DIndex performs dynamic transitions between these two types based on the access patterns. It employs a background thread to identify ranges that have been static (no insertions) for a predefined period. For such ranges, the thread physically sorts

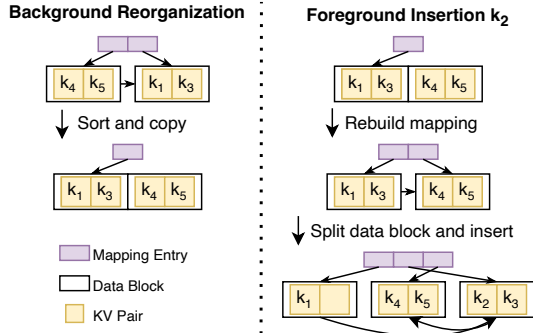


Fig. 6: The variable-span mapping scheme.

the blocks and replaces their mapping entries with a single entry pointing to the first block. Conversely, if an insertion operation (e.g., key  $k_2$ ) occurs within a contiguous range of data blocks and requires a data block split, DIndex rebuilds the mapping entries to allow for data block splits. This reverse process is efficient as it only requires rebuilding the relevant mapping entries without data block movement.

### B. Architecture and Workflow

Figure 5(c) illustrates the three-layer architecture of DIndex, consisting of a single root node at the top, multiple segment nodes in the middle, and numerous data blocks at the bottom. The root node routes query keys to the appropriate segment node. Each segment node, responsible for a specific range of data blocks, then uses an ML model to direct the query key to the correct data block within its range.

The root node contains an array of pivot keys and a mapping table. Each pivot key represents the smallest key within a segment, and the mapping table stores the on-disk address of each segment node. The root node is pinned in memory since it is used in every read and write operation. When looking up the root node, binary search is used to find the correct segment node for the given query key without involving an ML model, since in-memory binary search incurs negligible latency overhead compared to disk I/Os.

Each segment node contains a learned model, an array of sentinels, and a mapping table. Given a query key as input, this model predicts the position of the target data block within the segment. The sentinel array is computed using the model and the pivot key (the smallest key) of each data block in the segment. Each segment node manages either physically contiguous, sorted data blocks (termed *sorted segments*), or logically ordered, unsorted but linked data blocks (termed *unsorted segments*). For sorted segments (such as the left and the right segments in Figure 5(c)), DIndex uses a single mapping entry to store the start address of the data blocks. For unsorted segments (such as the middle segment in Figure 5(c)), DIndex keeps a mapping entry for each data block it manages. DIndex adopts 4-byte `uint32_t` as the sentinel type. While smaller sentinels are possible, we find that 4-byte sentinels strike a balance between memory overhead and collision rate. As will be shown in Section V-D6, there is no collision with 4-byte sentinels for all the integer key datasets, but the collision of 2-byte and 1-byte sentinels could cause up to 0.2 and

even 8.0 additional disk I/Os per read operation, respectively. Segment nodes are fixed at 4kB, stored on disk and cached in memory as needed.

Data blocks are analogous to B+tree leaf nodes and store the actual key-value pairs. DIndex adopts the well-known slotted page organization [23] to support variable-length keys and values. Each data block includes a pointer to the next logical block to support range scans. Similar to segment nodes, each data block is fixed at 4kB, and the frequently accessed data blocks are cached in memory.

### C. Model Design

DIndex maintains a piecewise linear model in each segment to predict data block positions and compress sentinels. The model is trained on a list of  $(key, pos)$  pairs, where each key is the pivot key of a data block, and the position is the index of the data block in the segment node. The learned model stores a subset of the pairs to serve as the knots, which are connected by straight lines to form a linear spline. To predict a position, DIndex performs linear interpolation between the two consecutive knots a key falls between. If a key is greater than all knots, the predicted position is that of the last knot.

As discussed in Section III-A1, sentinel collisions arise when distinct keys are mapped to the same integer. To mitigate this, our knot selection process minimizes these collisions within a predefined memory budget for the model. We employ an iterative greedy algorithm to choose the set of knots. It begins with two knots: the first and last  $(key, pos)$  pairs. The algorithm then identifies ranges with sentinel collisions. In each iteration, the algorithm identifies the longest range of keys with sentinel collisions and adds its start and end keys to the knot set. After adding these new knots, it performs a pruning step, removing any existing knots that have become redundant to conserve memory. This process is repeated until the model space is exhausted or no sentinel collisions remain.

For non-numeric keys such as string keys and composite keys, DIndex requires a numerical representation for model computation. DIndex interprets the key's raw byte representation as a large, big-endian integer in base-256. Furthermore, when the dataset has a known, limited character set (e.g., email addresses), DIndex uses a more compact base to reduce skewness among transformed integers.

### D. Basic Operations

DIndex provides five basic index operations: `get`, `put`, `delete`, `scan`, and `bulk load`. In this paper, we focus on the algorithms for `get`, `put`, and `bulk load`. We omit detailed discussion of the other two operations: `delete` is similar to `put` because we use logical deletion, and `scan` is straightforward to implement since the data blocks are chained together. To simplify the presentation, we assume all segment nodes are unsorted segments and describe the algorithms in a single-threaded setting without crash recovery support. Extending the operations to support sorted segments is straightforward. Concurrency and crash recovery support are discussed in Sections III-F and III-G, respectively.

---

**Algorithm 1: Get Operation**

---

```
1 Function LocateDataBlk (key) :  
  // ① Root node lookup  
2  seg_id ← BinarySearch (root.pivot_keys, key)  
3  seg_location ← root.mapping[seg_id]  
  // ② Segment node lookup  
4  seg_node ← LoadSeg (seg_location)  
5  pred_val ← Predict (seg_node.model, key)  
6  pred_blk_id ← ⌊pred_val⌋  
7  pred_scaled ← ScaleToUInt32 (pred_val)  
8  blk_id ← ExpSearch (seg_node.sentinels, pred_scaled,  
  pred_blk_id)  
9  blk_location ← seg_node.mapping[blk_id]  
  // ③ Data block lookup  
10 data_blk ← Loadblk (blk_location)  
  // ④ Collision handling  
11 while key < MinKey (data_blk) or key >  
  MaxKey (data_blk) do  
12   if key < MinKey (data_blk) then  
13     blk_id ← blk_id - 1  
14   else  
15     blk_id ← blk_id + 1  
16   blk_location ← seg_node.mapping[blk_id]  
17   data_blk ← Loadblk (blk_location)  
18 return (seg_node, data_blk, blk_id)  
19 Function Get (key) :  
20 (seg_node, data_blk, blk_id) ← LocateDataBlk (key)  
21 value ← BinarySearch (data_blk.kv_pairs, key)  
22 return value
```

---

1) *Get*: The get operation, detailed in Algorithm 1, traverses the three-layer index structure of DIndex as follows. ① *Root node lookup*. First, DIndex performs a binary search on the pivot keys to identify the target segment node. Since the root node resides entirely in memory (as discussed in Section III-B), the cost of this binary search is negligible compared to subsequent disk I/O operations. Next, DIndex consults the root node’s mapping table to retrieve the disk location of the identified segment node. ② *Segment node lookup*. Segment nodes are stored on disk and managed by a buffer pool. If the required segment node is not present in the buffer pool, DIndex fetches it from disk. Once the segment node is in memory, DIndex uses its learned model to predict a candidate data block ID (*pred\_blk\_id*) and also scale the predicted value to an integer (*pred\_scaled*) to compare with the sentinel values. An exponential search, starting from *pred\_blk\_id*, compares *pred\_scaled* against the sentinel values to locate the target data block ID (*blk\_id*). This search identifies the block *blk\_id* satisfying the following: 1) If *pred\_scaled* matches one or more sentinel values, *blk\_id* is the index of the first block whose sentinel equals *pred\_scaled*. 2) Otherwise, *blk\_id* is the index of the block corresponding to the sentinel with the largest value strictly less than *pred\_scaled*. Finally, DIndex uses the segment node’s mapping table to find the disk location of this target data block. ③ *Data block lookup*. Data blocks also reside on disk and are managed by the buffer pool. DIndex fetches the target data block from disk if it is not already buffered. Once the block is in memory,

---

**Algorithm 2: Put Operation**

---

```
1 Function Put (key, value) :  
  // ① Data block lookup and insertion  
2  (seg_node, data_blk, blk_id) ← LocateDataBlk (key)  
3  if HasSpace (data_blk, key, value) then  
4    UpdateOrInsert (data_blk, key, value)  
5  else  
6    // ② Data block split  
7    (new_data_blk, new_blk_location) ←  
      AllocNewDataBlk ()  
8    new_sentinel ← SplitDataBlk (data_blk,  
      new_data_blk, key, value)  
9    // Update parent segment node  
10   if HasSpace (seg_node) then  
11     UpdateSegment (seg_node, new_sentinel,  
      new_blk_location, blk_id+1)  
12     ScheduleRetrain (seg_node)  
13   else  
14     // ③ Segment node split  
15     (new_seg_node, new_seg_location) ←  
      AllocNewSegmentNode ()  
16     new_pivot ← SplitSegmentNode (seg_node,  
      new_seg_node, new_sentinel, new_blk_location)  
17     ScheduleRetrain (seg_node)  
18     ScheduleRetrain (new_seg_node)  
19     UpdateRoot (root, new_pivot, new_seg_location)
```

---

DIndex verifies if the query key lies within the block’s key range [*min\_key*, *max\_key*]. If the key is within the range, DIndex performs a binary search within the block to find the corresponding value. ④ *Collision handling*. Otherwise, if the key is outside the expected range, DIndex reads adjacent blocks until the block containing the key is found. The three-layer structure of DIndex ensures its read I/O complexity  $O(1)$  as it has a fixed index depth, and I/O for collision handling are always contained in a single segment node. Such collisions are rare in practice. Section V-D6 demonstrates there is no collision for all the integer key datasets and on average 0.1 collisions per query for the string key datasets.

2) *Put*: The put operation is responsible for both update and insert operations, detailed in Algorithm 2. It performs the following steps. ① *Data block lookup and insertion*. It first locates the target data block for the given key, similar to a read operation. If this block has sufficient capacity, DIndex simply performs the update or insertion directly within that block. ② *Data block split*. Otherwise, a data block split is necessary. DIndex allocates a new data block and redistributes approximately half of the key-value pairs from the original block to the new one. The new key-value pair is then inserted into the appropriate block. This split involves updating the parent segment node to include a new sentinel key and a reference to the newly allocated block. To maintain model quality, DIndex schedules a retraining task for the segment node when the number of insertions to the segment since the last retraining exceeds a threshold. DIndex employs a background thread to handle these retraining tasks asynchronously, ensuring they do not block the main execution threads. ③ *Segment node split*. If the segment node is also full during this update, DIndex

recursively splits the segment node, schedules retraining tasks for both the original and the newly created segment nodes, and inserts the pivot key and mapping entry of the newly created segment node into the root node.

3) *Bulk load*: DIndex supports bulk loading to efficiently initialize the index from a given set of key-value pairs via a bottom-up approach. First, the key-value pairs are sorted and DIndex fills one data block with key-value pairs in an ascending order until it becomes 70% full, and then continues with the next data block. The remaining 30% of the space is reserved for future insertions. These data blocks are then grouped into segment nodes, where each segment node also reserves 30% of empty space. The model training procedure occurs during the construction of these segment nodes. Finally, DIndex constructs the root node using the generated segment nodes, completing the index structure.

#### E. Variable-Span Mapping Support

As described in Sections III-A2 and III-B, DIndex utilizes a background thread to identify unsorted segments with no insertions for a period. This thread reorganizes the data blocks within the identified segments into sorted and contiguous data blocks, and transforms the segment into a sorted segment. However, since segment nodes have a fixed 4kB size, eliminating the mapping table only frees up internal space rather than reducing the index size. To overcome this inefficiency, DIndex implements a merging strategy. It checks for adjacent segment nodes that are both eligible for reorganization. If found, DIndex merges them into a single sorted segment. DIndex updates the root node to remove the previous two segment nodes and add the new segment. This merging process is designed to be conservative (e.g., when the two segment nodes have not had insertions for 10 minutes), aiming to minimize the write overhead associated with data movement. DIndex also includes a restoration mechanism when an insertion is performed on a merged segment node, and the data block does not have enough space to accommodate the new key-value pair. In this case, DIndex rebuilds the per-block mapping entries to support data block splits. The restoration is more lightweight than merge since it does not incur data block copy.

As mentioned in Section II-C, the data access pattern in OLTP applications makes the reorganization infrequent and the variable-span mapping scheme beneficial. Even in database systems using multi-version concurrency control (MVCC), where logical updates are implemented as physical insertions at the storage layer, reorganization may still be infrequent. This is because data blocks typically contain reserved space to accommodate new insertions without triggering splits, similar to B+trees. As old versions become obsolete and are reclaimed through garbage collection, the freed space continues to buffer future insertions, further reducing the likelihood of splits that would necessitate reorganization.

#### F. Concurrency Support

DIndex supports concurrent `get` and `put` operations by employing the standard latch coupling protocol [24] for coor-

inating access to its hierarchical structure. Each node within the index is equipped with its own read-write lock (latch). To ensure safe traversal during concurrent operations while preventing deadlocks, DIndex strictly adheres to the latch coupling rule: a latch on a child node is always acquired before the latch on its parent node is released.

Specifically, for a `get` request, DIndex acquires the lock in the following order: ① Acquire a read lock on the root node and identify the target segment node. ② Acquire a read lock on the identified segment node, release the read lock on the root node, and determine the target data block. ③ Acquire a read lock on the predicted target data block. If the key is not within the current data block's range (due to a sentinel collision), DIndex needs to examine adjacent data blocks. This is achieved through releasing the lock on the current data block and acquiring a read lock on the appropriate neighboring data block. This process repeats until the data block containing the key's range is found and locked. The read lock on the segment node is not released until the correct data block is locked. ④ Release the lock on the data block after the key is found.

The `put` requests follow a similar lock acquisition pattern, but DIndex adopts a write lock at the data block level. If the data block has sufficient space, the request inserts the record and releases the write lock. However, if the data block is full and requires a split, the current attempt is aborted, and the `put` operation is restarted with an escalated locking strategy. Specifically, the restarted `put` operation attempts to acquire write locks on both the segment node and the data block. Furthermore, if the segment node itself is full, the operation must be restarted again. This time, it acquires write locks on the root node, the segment node, and the data block to safely perform the cascading split.

#### G. Crash Recovery

As an on-disk index, DIndex ensures crash recovery using the Write-Ahead Logging (WAL) protocol, with a recovery algorithm based on ARIES [25]. This process guarantees that all completed operations are durably persisted, while any modifications from incomplete operations are rolled back. Specifically, during write operations, DIndex generates log records for all index modifications, such as node insertions, node allocations, and data movement between nodes. Each operation is bracketed by begin and commit log records to delineate its boundaries. In accordance with the WAL protocol, DIndex forces log records to disk before the modified index pages are written back, and before acknowledging completion to the user. DIndex also performs periodic ARIES-style checkpointing to bound the amount of work required during recovery. The recovery process follows the standard three phase approach of ARIES, including analysis, redo, and undo.

#### H. Implementation

We implemented DIndex as a C++ library with approximately 6k lines of code that runs on POSIX-compliant file systems such as Ext4. For disk I/O operations, DIndex uses the `pread` and `pwrite` syscalls with the `O_DIRECT` flag

to bypass the operating system’s page cache. Instead of relying on OS caching, DIndex maintains its own buffer pool with a configurable maximum size for caching pages and managing page latches. This buffer pool employs a clock-based page replacement algorithm and writes dirty pages back to disk upon eviction. To handle background maintenance tasks, DIndex employs two dedicated threads: one for model retraining and another for data block reorganization. These threads execute their tasks when disk utilization is low to avoid interfering with the main execution threads. To support model computations on non-numeric keys, DIndex utilizes `boost::multiprecision::cpp_int` as the big integer type mentioned in Section III-C.

#### IV. DISCUSSION

**Rationale for the Three-Layer Design.** While it might seem appealing to use a deeper hierarchy than the current three-layer structure, we find our current design sufficient for a wide range of dataset sizes. Replacing the large flat root node with more tree-like layers has two benefits: 1) helps shorten the time spent on searching in memory and 2) allows paging in and out the index metadata at finer granularity so that the index can work with extremely large datasets. However, read latency is dominated by disk I/O, and additional layers will only increase the contention for cache. Regarding indexing capability, we find that our current design is sufficient for indexing petabyte-scale datasets with 16GB memory (to host the root node). Since it takes a decade for premium SSDs to grow from 15TB to 122TB [26], [27], we believe DIndex will remain applicable in the near future, except for rare extreme settings where the storage capacity grows drastically (e.g., via SSD arrays) yet the memory capacity remains customer-grade.

**Benefits to Other Database Systems.** While our evaluation shows DIndex’s performance in the context of a relational database (MySQL), DIndex could potentially benefit other database systems. For LSM-tree-based key-value stores, DIndex can be used as the data organization within each SSTable. The root and segment nodes serve as the index of an SSTable, which has smaller metadata overhead compared to the original index block. This could reduce the additional I/O cost of reading index blocks. For column stores, which organize data into large blocks called chunks that are primarily accessed through scans, DIndex is better suited for indexing metadata (e.g., chunk locations) than for organizing the data within the chunks. For vector database systems, a second-stage retrieval is required to fetch the attributes for the top-K IDs. This process involves batch point lookups on random IDs, and DIndex’s small memory footprint can also potentially reduce the number of I/Os in this stage and improve the end-to-end performance.

#### V. EVALUATION

We evaluate DIndex to answer the following questions:

- How does DIndex’s performance and index size compare to B+tree and other learned indexes?
- Does DIndex bring end-to-end performance improvement when integrated into a complete data management stack?

- How is the performance of DIndex affected by different factors (e.g., record size, dataset)?
- How effective are DIndex’s core design choices?

##### A. Setup

1) *Testbed:* All experiments were conducted on a server equipped with two 12-core Intel Xeon E5-2650 v4 CPUs. We utilized an NVMe Solid State Drive (Samsung 970 EVO Plus, 2TB) as the storage device and Ext4 as the file system. We limited the available memory for each index to 1/75 of the total dataset size by default.

2) *Configuration:* DIndex uses 4-byte sentinels across all datasets and configures model space limits for segment nodes based on key types. For integer keys, it reserves 32 bytes for the model space (as a single-piece linear spline introduces no sentinel collision). For other key types, it reserves 1,024 bytes for the model space.<sup>1</sup>

3) *Datasets:* Throughout the experiments, we used 8 integer datasets and 3 string datasets. The integer datasets are of type `uint64`, with 6 datasets (`fb` [28], `amzn` [28], `osm` [28], `covid` [29], `planet` [29] and `genome` [29]) having 200 million keys and 2 datasets (`amzn-l` [28] and `osm-l` [28]) having 800 million keys. The 3 string datasets (`reddit` [30], `email` [31] and `wiki` [32]) contain 26 million (average length 11 bytes), 100 million (average length 24 bytes) and 47 million (average length 20 bytes) keys, respectively. We varied value sizes across experiments, creating record sizes (key + value) that ranged from 16 to 256 bytes.

4) *Counterparts:* DIndex is compared against the traditional B+tree, 4 learned index counterparts for integer keys (Hybrid-PGM-Disk [13], AULID [12], TreeLine [16], FILM [17]) and 1 learned index counterpart for string keys (LITS-D [33]). For the learned index structures, the implementations are obtained from their open-source repositories. These indexes are configured as follows.

**B+tree:** We used the B+tree implementation from [34], which incorporates fine-grained concurrency [2]. It uses a fixed-size 4kB page for each node and the branching factor is determined by the number of keys and pointers that can fit into the page. Approximately, the fan-out is 120 when indexing integers, and that of string datasets ranges from 77 to 109. We modified this implementation to bypass the OS page cache and use a self-managed buffer pool with configurable memory budget. **Hybrid-PGM-Disk:** Hybrid-PGM-Disk has two configurable parameters *ratio* and  $R_p$ . By default, we set *ratio* = 1 : 200 and  $R_p$  = 1.05. For experiments with different memory budgets (Section V-D3), the *ratio* parameter is adjusted proportionally. Any memory not allocated to the dynamic index or the model serves as a cache for static data. **AULID:** The open-source implementation of AULID relies on the OS page cache to speed up read operations. Therefore, we used `cgroup` to limit the maximum memory usage during experiments. **TreeLine:** We ensured the size

<sup>1</sup>When the memory budget and key distribution are unknown in advance, a simple exhaustive search can be performed to determine the optimal combination of sentinel size and model space allocation.

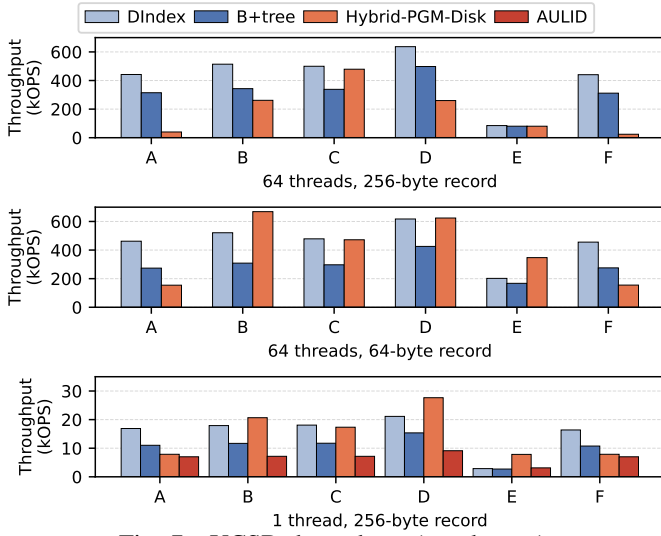


Fig. 7: YCSB throughput (*fb* dataset).

of its record-level cache and in-memory data structures do not exceed the memory budget. We chose the configuration parameters of TreeLine as suggested by its paper (*goal* = 44 and *epsilon* = 5). **FILM**: We configured FILM so that its memory index structure and the cached values for hot keys do not exceed the memory budget. **LITS-D**: Since LITS is designed for in-memory settings, we modified it to store all nodes on disk and use a buffer pool for caching frequently accessed pages.

In the following experiments, the results for AULID are only presented for the single-thread configuration since it does not support concurrency. TreeLine and FILM are only presented in Section V-D2 where larger memory budgets are used, as their minimum memory requirements exceed our default budget.

### B. YCSB Benchmark

We first compare each index structure using the YCSB [18] benchmark on the *fb* dataset. The YCSB workloads exhibit non-uniform access patterns: YCSB-D demonstrates temporal locality by reading recently inserted keys, while other workloads follow skewed access patterns using a Zipfian distribution for read and write operations.

We compare the throughput of different indexes under three types of configurations: 1) 64 threads with 256-byte records; 2) 64 threads with 64-byte records; and 3) 1 thread with 256-byte records. Due to its lack of concurrency support, results for AULID are presented only for the single-thread configuration. Figure 7 presents the throughput results for all experiments. First, in the configuration using 64 threads and 256-byte records, DIndex achieves the highest throughput among the evaluated indexes across all six workloads. Specifically, DIndex improves throughput by up to 50% compared to B+tree and outperforms Hybrid-PGM-Disk by up to 17%. The performance advantage over B+tree is primarily attributed to DIndex’s smaller index size, which reduces I/O overhead during lookups. The smallest performance gain for DIndex over B+tree occurs in YCSB-E (short-range scans), where

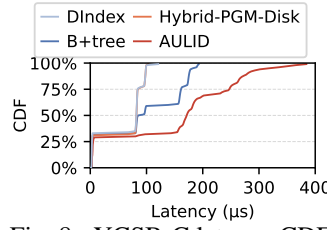


Fig. 8: YCSB-C latency CDF (*fb* dataset, 64-byte records).

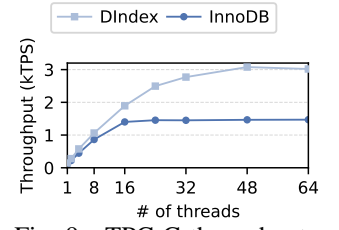


Fig. 9: TPC-C throughput.

the performance is not primarily limited by index traversal. Hybrid-PGM-Disk exhibits poor performance, particularly in update-intensive workloads like YCSB-A and YCSB-F. This is because high update rates trigger frequent merge operations, and its merge overhead is higher for large records. In the read-only workload (YCSB-C), Hybrid-PGM-Disk incurs no merge overhead and achieves performance comparable to DIndex, as both typically require at most one disk I/O per lookup. Next, we examine performance under the second configuration (64 threads, 64-byte records). Under this setting, DIndex maintains a similar performance advantage over B+tree. Hybrid-PGM-Disk, however, benefits from lower merge overhead with smaller records. Consequently, it achieves higher throughput than in the 256-byte record setting, especially in workloads with high update or insert rates. Finally, we analyze single-threaded performance using 256-byte records, including AULID in this comparison. In this setting, DIndex achieves up to 53% higher throughput than B+tree and up to 1.5 $\times$  higher throughput than AULID across the six workloads. Hybrid-PGM-Disk performs best in certain low-write-rate workloads (e.g., YCSB-B) in this setting, as its effective merge overhead is lower compared to the 64-thread configurations.

To understand the latency characteristics of different indexes, we measure the *get* latency CDF for each index under the read-only YCSB-C workload. Figure 8 presents the results for the single-thread experiment with 64-byte records. The CDF displays distinct steps, which primarily correspond to the discrete number of disk I/Os required by different queries. For approximately the fastest 30% of requests, all four indexes exhibit query latencies below 10 $\mu$ s, as these requests are served directly from the cache, incurring no disk I/Os. Beyond these cache hits, both DIndex and Hybrid-PGM-Disk complete nearly all remaining requests within 100 $\mu$ s. This suggests that most lookups require only a single disk I/O, with no additional I/Os incurred during the index traversal itself. In contrast, approximately 40% of requests for B+tree necessitate at least two disk I/Os, indicating that the internal nodes are not always cached. AULID shows the highest latency among the four indexes, correlating with its larger index size.

### C. TPC-C Benchmark

To evaluate DIndex’s performance within a complete data management stack, we integrated it into MySQL using MySQL’s pluggable storage engine architecture [35]. We ran the TPC-C benchmark [15] to compare DIndex’s performance against InnoDB. TPC-C is a complex OLTP benchmark with

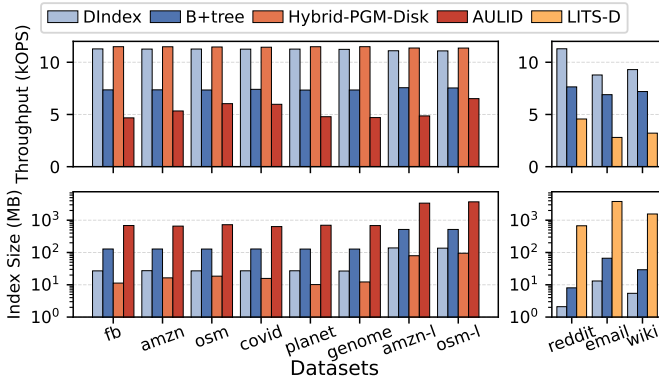


Fig. 10: 1-thread throughput and index size across datasets.

skewed access patterns and temporal locality caused by newly placed orders and non-uniform customer selection. We used the open-source TPC-C implementation from BenchBase [36], [37]. Since DIndex currently does not support transactions, we modified the benchmark by removing the remote new order and payment transactions to avoid conflicts that would require concurrency control mechanisms, and we ran the benchmark under the read-uncommitted isolation level.

We configured the benchmark to use 1,000 warehouses with 500MB memory budget, and varied the number of threads from 1 to 64 and compared the performance of DIndex and InnoDB. Figure 9 presents the performance results. DIndex shows consistently higher throughput than InnoDB across all thread counts due to the smaller index size. Under 64 threads, DIndex can achieve  $2.1\times$  the performance of InnoDB.

#### D. Microbenchmarks

We conducted a series of microbenchmarks with diverse configurations to compare the performance of DIndex with other indexes, and also to better understand how various factors affect the performance of DIndex. In these microbenchmarks, all read and write operations are uniformly random without locality patterns. This allows us to isolate and analyze the performance characteristics of each index structure without the confounding effects of access locality.

1) *Performance under different datasets*: We first compare the performance of DIndex and other indexes across different datasets, including both integer key and string key datasets. For integer datasets, we used 56-byte values to form 64-byte records. For string datasets, we chose 8-byte values since LITS-D currently only supports 8-byte values. We measure the single-thread throughput and index size of each index. The results are presented in Figure 10, with integer datasets on the left and string datasets on the right. On the first six integer datasets, DIndex consistently achieves approximately 53% higher throughput and requires 79% less index space compared to B+tree. The index size of DIndex remains relatively constant across these datasets due to the same number of keys, which results in a similar number of segment nodes. This small index size of DIndex ensures almost no additional I/O overhead during index traversals. Besides small index size, achieving high throughput with DIndex also requires a low

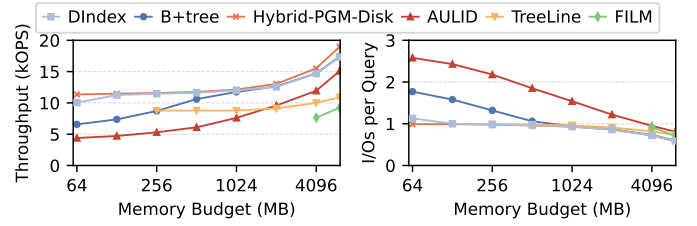


Fig. 11: 1-thread throughput and I/Os per query across memory budgets (fb dataset, 64-byte records, read-only).

sentinel collision rate. Our statistics indicate that DIndex incurs no sentinel collisions for the integer key datasets, and therefore has no impact on performance. While Hybrid-PGM-Disk exhibits an even smaller index size than DIndex, we contend that DIndex’s index is already sufficiently small to be fully cached within the memory budget. We compare the performance of DIndex under different memory constraints in Section V-D2. AULID has the largest index size among all the indexes, and consequently, it has the lowest throughput due to higher I/O overhead during index traversals. For the two datasets with 800M keys, DIndex shows similar performance improvement. Compared to B+tree, DIndex still achieves 47% higher throughput while requiring 73% smaller index size. For string datasets, DIndex shows 27% to 48% performance improvement compared to B+tree and a  $1.5\times$  to  $2.1\times$  higher than LITS-D. LITS-D fails to work as an efficient on-disk index though it performs well in the in-memory settings. The reason is that its index size is extremely large, causing 2-3 additional I/Os for each query to read the index. Take the reddit dataset as an example. Its total index size is 672MB, where 72% of the index size in LITS-D is due to its large and sparse model nodes, and the remaining 19% and 9% belong to the compact leaf nodes and trie nodes, respectively. DIndex reduces the index size by 99% compared to LITS-D.

2) *Performance under different memory constraints*: To evaluate how DIndex performs under different memory constraints, we varied the allocated memory budget. Specifically, we tested budgets ranging from 1/150 (64MB) to approximately 2/3 (6144MB) of the total dataset size. We compare the performance of each index using a single-threaded, read-only workload to assess the impact of the memory budget on read performance, primarily focusing on how it affects the number of required I/O operations. The results are presented in Figure 11. When the memory budget exceeds 1/75 (128MB) of the dataset size, DIndex’s throughput remains largely stable since its index can be fully cached in memory, eliminating I/O overhead during queries. This is corroborated by the I/O measurements (shown in the right part of Figure 11), which indicate only 1 I/O per query under these conditions. With an extremely low memory budget (1/150 of the dataset size), DIndex’s throughput decreases by 11% as parts of the index fail to be cached. Under this constrained budget, DIndex incurs an average of 1.13 I/Os per query. The performance of Hybrid-PGM-Disk is stable across the tested memory budgets due to its smaller memory footprint. However, as previously noted, this efficiency comes at the cost of higher merge overhead.

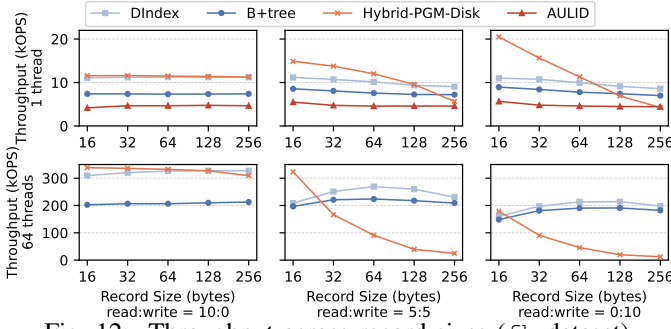


Fig. 12: Throughput across record sizes (fb dataset).

The performance of B+tree and AULID is more sensitive to available memory owing to their larger index sizes. B+tree only achieves performance comparable to DIndex when the memory budget reaches approximately 1/10 of the dataset size. AULID performs worse than B+tree due to its even larger index size. TreeLine and FILM fail to work under low memory constraints since they need to keep a large size of metadata in memory. Specifically, FILM must store all keys and associated directory entries in memory, and thus requires a >2GB memory budget. While TreeLine has a smaller memory footprint, its use of B+tree for organizing pivot keys (each segment’s lower boundary) still imposes significant memory overhead. Besides, when the memory is sufficient for these two indexes, they still perform worse than DIndex. TreeLine employs record-level caching, which incurs high CPU overhead upon cache misses, as it performs tens of cache insertions to put all records in a page into the cache. FILM, however, has a larger I/O granularity (64kB) and thus the read cost for a single I/O is higher.

3) *Performance under different record sizes:* We evaluate the performance of different indexes using record sizes ranging from 16 to 256 bytes. The results, presented in Figure 12, also show performance under varying read-write ratios and number of threads. From these results, we have the following findings. First, under read-only workloads, DIndex and Hybrid-PGM-Disk exhibit similar performance. This is because both indexes typically require only one disk I/O per operation, without incurring additional I/Os during index traversal. In contrast, B+tree and AULID show lower performance because they incur additional I/Os during index traversal. Second, under mixed read-write and write-only workloads, Hybrid-PGM-Disk’s performance degrades significantly as the record size increases. This performance degradation becomes more pronounced under high concurrency. Under 64 threads, Hybrid-PGM-Disk’s throughput with 256-byte records falls to 12 kOPS, which is only 6% of the throughput achieved by DIndex under the same conditions. The performance of DIndex remains relatively stable across the evaluated record sizes.

4) *Performance under different number of threads:* We compare the scalability of different indexes across multiple read-write ratios. To achieve this, we measure the throughput of each index while increasing the number of concurrent threads from 1 to 64. The results are presented in Figure 13. Under read-only workloads, DIndex and Hybrid-PGM-Disk

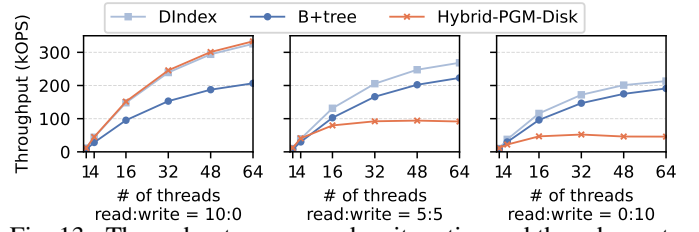


Fig. 13: Throughput across read-write ratios and thread counts (fb dataset, 64-byte records).

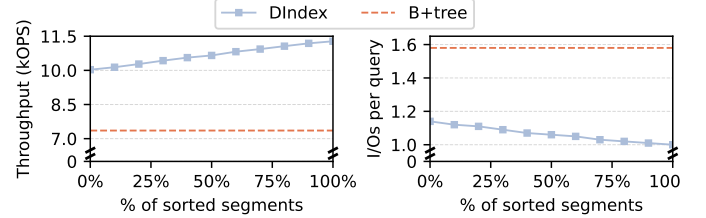


Fig. 14: 1-thread throughput and I/Os per query with different ratios of sorted segments (fb dataset, 64-byte records, read-only).

scale similarly as the thread count increases, since both require only a single disk I/O per operation to fetch data. Consequently, their performance becomes bottlenecked by the disk IOPS limit at 64 threads. However, under workloads involving writes (mixed read-write and write-only), DIndex consistently outperforms both Hybrid-PGM-Disk and B+tree, achieving higher throughput. This demonstrates the effectiveness of DIndex’s fine-grained locking mechanism for managing concurrent operations. In contrast, Hybrid-PGM-Disk’s scalability suffers significantly due to its high merge overhead.

5) *Performance under different ratios of sorted segments:* We then investigate how DIndex’s performance is affected by the ratio of sorted segments, which is determined by the insert pattern. We simulated different insert patterns by varying this ratio and measured the resulting throughput and I/O per query (Figure 14). The result shows that in the worst case, DIndex’s throughput is 11% lower than the best case, but still 36% higher than B+tree. Furthermore, the average I/O count per query reaches 1.14 in the worst case, meaning about 14% of queries require an extra I/O. Overall, Figure 14 demonstrates a nearly linear relationship: DIndex’s performance improves as the proportion of sorted segments increases.

6) *Performance under different sentinel sizes:* We analyze the impact of sentinel size on DIndex’s performance. In addition to the default 4-byte sentinels, we evaluate the use of smaller 1-byte (`uint8_t`) and 2-byte (`uint16_t`) sentinels across different integer datasets. The results are presented in Figure 15. The throughput of DIndex with 4-byte sentinels remains mostly consistent across datasets since the sentinel collision rate is low (<0.01%). In contrast, DIndex with 1-byte and 2-byte sentinels exhibits lower or unstable throughput due to higher sentinel collision rates. For 2-byte sentinels, the performance degradation is up to 16%, resulting from an additional 0.2 I/Os per query caused by collisions. For 1-byte sentinels, the throughput is even 49% to 87% lower than that

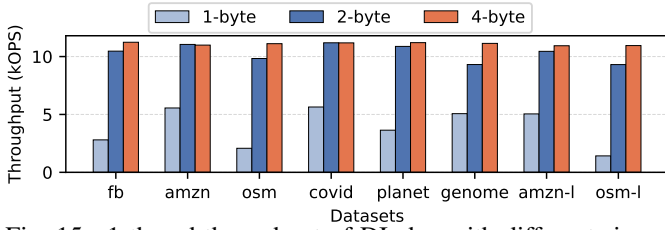


Fig. 15: 1-thread throughput of DIndex with different size of sentinels (64-byte records, read-only).

of 4-byte sentinels due to 1.0 to 8.0 additional I/Os per query.

7) *Recovery procedure analysis*: We simulated crashes during put operations and invoked the recovery procedure to check the correctness of DIndex’s recovery mechanism. After recovery, DIndex ensures the index structure is in a consistent state, and all completed operations are durably persisted. We also measured the recovery time using a trace with 1 million log entries (68MB) generated since the last checkpoint. The recovery process took 30 seconds, with the majority of the time dominated by disk I/O (page reads and writes).

## VI. RELATED WORK

**Space-Efficient Traditional Disk Indexes.** Block compression [38], [39], [40] is the most intuitive way to reduce the storage overhead. It trades extra computation for smaller storage consumption, which is orthogonal to our work. Other research efforts have concentrated on compressing individual keys. For example, null suppression reduces data size by compactly representing or omitting nulls and insignificant leading/trailing zeros or spaces, while delta encoding saves space by storing only the differences between consecutive data elements [22]. There are also dictionary-based methods [6], [41], [42], [43] that use a dictionary to compress keys and operate under the assumption of a relatively static workload. Perfect hashing [44], [45], [46] is another popular strategy for efficient memory indexing. However, it is hindered by lengthy construction times and does not maintain the order of keys after hashing. Another line of research focuses on developing hybrid structures [44], [47]. These indexes utilize compact index structures for indexing cold and static data while employing write-friendly index structures for managing hot data. There has also been research aiming to reducing memory footprint by evicting cold tuples from memory based on the access pattern. For example, the anti-caching architecture [48] moves cold data to disk as the database grows in size. VoltDB [49] relies on the operating system’s virtual memory system to move cold data to disk.

**Learned Indexes and Extensions.** There have been numerous advancements in learned indexes [7]. Research has primarily focused on enhancing the core capabilities or adapting them to new scenarios. There has been a list of solutions adapting learned indexes for on-disk storage. Hybrid-PGM-Disk and AULID were discussed in Section II-B in detail. TreeLine organizes multiple contiguous disk pages as a segment and uses a B+tree to index the segments. Inside each segment, it uses learned models for routing. FILM targets larger-than-

memory databases, using a unified tree structure to handle data spanning memory and disk, and includes an adaptive LRU mechanism for cold data identification and eviction. The original learned index structure [7] was limited to single-threaded, read-only lookups. Consequently, one major line of research has focused on write support. Solutions either reserve space within nodes for future insertions (e.g., ALEX [9], LIPP [21]) or use a delta index to buffer write operations and perform periodic merges with the main index (e.g., XIndex [8], PGM-Index [10], FINEdex [50]). DIndex adopts the former approach and is specifically optimized to minimize disk I/O during data block splits. Some indexes, like XIndex [8] and FINEdex [50], also support concurrent write operations via fine-grained optimistic locking. Another line of research attempts to optimize the model structure in learned indexes for better performance. The original learned index [7] proposed the hierarchical model structure named Recursive Model Index (RMI) for routing read requests. FITing-Tree [11] uses piecewise linear functions to approximate data distribution with an error bound guarantee. PGM-Index [10] constructs a recursive index structure with an optimal number of models. RadixSpline [51] uses a linear spline as the indexing function to reduce the cost of model construction and uses a radix table to index the spline points. Learned indexes have also been adapted for modern hardware platforms beyond main memory. XStore [52] uses the learned index as the client-side cache for the RDMA-based key-value store and leverages a hybrid structure to handle dynamic workloads. ROLEX [53] uses the learned index in the disaggregated memory scenario to speed up index retrieval. APEX [54] targets persistent memory, delivering high performance and crash consistency. Finally, research extends learned indexes to support different data types. SIndex [55], RSS [14] and LITS [33] are learned indexes specially optimized for string keys. LISA [56] and Tsunami [57] are two learned indexes built for multi-dimensional data.

## VII. CONCLUSION

This paper proposes DIndex, an on-disk learned index designed for memory-constrained environments. DIndex maintains model-compressed pivot keys called sentinels to guide error correction and uses a variable-span mapping scheme to track the physical locations of data blocks with low overhead. Our evaluation shows that DIndex outperforms B+tree and other state-of-the-art on-disk learned indexes by up to  $28\times$  in throughput and reduces index size by up to 99%.

## ACKNOWLEDGEMENT

This work was supported by the National Natural Science Foundation of China (Grant Nos. 62422209, 62272304), the National Key R&D Program of China (Grant No. 2023YFB3002000), and Huawei. We thank Youyun Wang for his contribution in the early stage of this work. Zhaoguo Wang is the corresponding author (zhaoguowang@sjtu.edu.cn).

## AI-GENERATED CONTENT ACKNOWLEDGEMENT

Google Gemini is used for proofreading and polishing this paper. No AI-generated content is directly incorporated without manual examination.

## REFERENCES

- [1] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET '70. New York, NY, USA: Association for Computing Machinery, 1970, p. 107–141. [Online]. Available: <https://doi.org/10.1145/1734663.1734671>
- [2] P. L. Lehman and S. B. Yao, "Efficient locking for concurrent operations on b-trees," *ACM Trans. Database Syst.*, vol. 6, no. 4, p. 650–670, dec 1981. [Online]. Available: <https://doi.org/10.1145/319628.319663>
- [3] "MySQL," <https://www.mysql.com>, 2025.
- [4] "PostgreSQL," <https://postgresql.org>, 2025.
- [5] "SQLite," <https://www.sqlite.org>, 2025.
- [6] H. Zhang, X. Liu, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "Order-preserving key compression for in-memory search trees," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1601–1615. [Online]. Available: <https://doi.org/10.1145/3318464.3380583>
- [7] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 489–504. [Online]. Available: <https://doi.org/10.1145/3183713.3196909>
- [8] C. Tang, Y. Wang, Z. Dong, G. Hu, Z. Wang, M. Wang, and H. Chen, *XIndex: A Scalable Learned Index for Multicore Data Storage*. New York, NY, USA: Association for Computing Machinery, 2020, p. 308–320. [Online]. Available: <https://doi.org/10.1145/3332466.3374547>
- [9] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. Lomet, and T. Kraska, *ALEX: An Updatable Adaptive Learned Index*. New York, NY, USA: Association for Computing Machinery, 2020, p. 969–984. [Online]. Available: <https://doi.org/10.1145/3318464.3389711>
- [10] P. Ferragina and G. Vinciguerra, "The pgm-index: A fully-dynamic compressed learned index with provable worst-case bounds," *Proc. VLDB Endow.*, vol. 13, no. 8, p. 1162–1175, apr 2020. [Online]. Available: <https://doi.org/10.14778/3389133.3389135>
- [11] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1189–1206. [Online]. Available: <https://doi.org/10.1145/3299869.3319860>
- [12] H. Lan, Z. Bao, J. S. Culpepper, R. Borovica-Gajic, and Y. Dong, "A fully on-disk updatable learned index," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, 2024, pp. 4856–4869.
- [13] J. Zhang, K. Su, and H. Zhang, "Making in-memory learned indexes efficient on disk," *Proc. ACM Manag. Data*, vol. 2, no. 3, May 2024. [Online]. Available: <https://doi.org/10.1145/3654954>
- [14] B. Spector, A. Kipf, K. Vaidya, C. Wang, U. F. Minhas, and T. Kraska, "Bounding the last mile: Efficient learned string indexing," *arXiv preprint arXiv:2111.14905*, 2021.
- [15] "TPC-C," <http://www.tpc.org/tpcc>, 2025.
- [16] G. X. Yu, M. Markakis, A. Kipf, P.-r. Larson, U. F. Minhas, and T. Kraska, "Treeline: an update-in-place key-value store for modern storage," *Proc. VLDB Endow.*, vol. 16, no. 1, p. 99–112, Sep. 2022. [Online]. Available: <https://doi.org/10.14778/3561261.3561270>
- [17] C. Ma, X. Yu, Y. Li, X. Meng, and A. Maolinyazi, "Film: A fully learned index for larger-than-memory databases," *Proc. VLDB Endow.*, vol. 16, no. 3, p. 561–573, Nov. 2022. [Online]. Available: <https://doi.org/10.14778/3570690.3570704>
- [18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: <https://doi.org/10.1145/1807128.1807152>
- [19] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wiskey: Separating keys from values in ssd-conscious storage," *ACM Trans. Storage*, vol. 13, no. 1, mar 2017. [Online]. Available: <https://doi.org/10.1145/3033273>
- [20] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "Hashkv: enabling efficient updates in kv storage via hashing," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018, p. 1007–1019.
- [21] J. Wu, Y. Zhang, S. Chen, J. Wang, Y. Chen, and C. Xing, "Updatable learned index with precise positions," *Proc. VLDB Endow.*, vol. 14, no. 8, p. 1276–1288, Apr. 2021. [Online]. Available: <https://doi.org/10.14778/3457390.3457393>
- [22] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 671–682. [Online]. Available: <https://doi.org/10.1145/1142473.1142548>
- [23] R. Ramakrishnan and J. Gehrke, *Database management systems*. McGraw-Hill New York, 2003, vol. 3.
- [24] G. Graefe, "A survey of b-tree locking techniques," *ACM Trans. Database Syst.*, vol. 35, no. 3, jul 2010. [Online]. Available: <https://doi.org/10.1145/1806907.1806908>
- [25] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, p. 94–162, Mar. 1992. [Online]. Available: <https://doi.org/10.1145/128765.128770>
- [26] "Samsung Introduces World's Largest Capacity (15.36TB) SSD for Enterprise Storage Systems," <https://news.samsung.com/global/samsung-now-introducing-worlds-largest-capacity-15-36tb-ssd-for-enterprise-storage-systems>, 2025.
- [27] "The Incredible Path to 122TB: Decades of Engineering Feats Led to this Milestone," <https://www.solidigm.com/products/technology/solidigm-path-to-122tb-ssd.html>, 2025.
- [28] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "Soss: A benchmark for learned indexes," *NeurIPS Workshop on Machine Learning for Systems*, 2019.
- [29] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang, "Are updatable learned indexes ready?" *Proc. VLDB Endow.*, vol. 15, no. 11, p. 3004–3017, Jul. 2022. [Online]. Available: <https://doi.org/10.14778/3551793.3551848>
- [30] "Reddit Usernames," <https://www.kaggle.com/datasets/colinmorris/reddit-usernames>, 2025.
- [31] "The Faker Library," <https://faker.readthedocs.io/en/master/>, 2025.
- [32] "Wikipedia Titles," <https://dumps.wikimedia.org/enwiki/latest/>, 2025.
- [33] Y. Yang and S. Chen, "Lits: An optimized learned index for strings," *Proc. VLDB Endow.*, vol. 17, no. 11, p. 3415–3427, Jul. 2024. [Online]. Available: <https://doi.org/10.14778/3681954.3682010>
- [34] K. Malbrain, "A blink tree latch method and protocol to support synchronous node deletion," 2014.
- [35] "Overview of MySQL Storage Engine Architecture," <https://dev.mysql.com/doc/refman/9.4/en/pluggable-storage-overview.html>, 2025.
- [36] "BenchBase," <https://github.com/cmu-db/benchbase>, 2025.
- [37] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "Oltpbench: An extensible testbed for benchmarking relational databases," *PVLDB*, vol. 7, no. 4, pp. 277–288, 2013. [Online]. Available: <http://www.vldb.org/pvldb/vol7/p277-difallah.pdf>
- [38] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte, "The implementation and performance of compressed databases," *SIGMOD Rec.*, vol. 29, no. 3, p. 55–67, sep 2000. [Online]. Available: <https://doi.org/10.1145/362084.362137>
- [39] "LZ4," <https://lz4.github.io/lz4>, 2025.
- [40] "Snappy," <https://github.com/google/snappy>, 2025.
- [41] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 283–296. [Online]. Available: <https://doi.org/10.1145/1559845.1559877>
- [42] G. Antoshenkov, D. Lomet, and J. Murray, "Order preserving string compression," in *Proceedings of the Twelfth International Conference on Data Engineering*, 1996, pp. 655–663.

- [43] B. Schlegel, R. Gemulla, and W. Lehner, “Fast integer compression using simd instructions,” in *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, ser. DaMoN ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 34–40. [Online]. Available: <https://doi.org/10.1145/1869389.1869394>
- [44] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, “Silt: A memory-efficient, high-performance key-value store,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1–13. [Online]. Available: <https://doi.org/10.1145/2043556.2043558>
- [45] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna, “Monotone minimal perfect hashing: Searching a sorted table with  $o(1)$  accesses,” in *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA ’09. USA: Society for Industrial and Applied Mathematics, 2009, p. 785–794.
- [46] E. A. Fox, L. S. Heath, Q. F. Chen, and A. M. Daoud, “Practical minimal perfect hash functions for large databases,” *Commun. ACM*, vol. 35, no. 1, p. 105–121, jan 1992. [Online]. Available: <https://doi.org/10.1145/129617.129623>
- [47] 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>
- [48] 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>
- [49] 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>
- [50] P. Li, Y. Hua, J. Jia, and P. Zuo, “Finedex: a fine-grained learned index scheme for scalable and concurrent memory systems,” *Proc. VLDB Endow.*, vol. 15, no. 2, p. 321–334, Oct. 2021. [Online]. Available: <https://doi.org/10.14778/3489496.3489512>
- [51] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, “Radixspline: A single-pass learned index,” in *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, ser. aiDM ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: <https://doi.org/10.1145/3401071.3401659>
- [52] X. Wei, R. Chen, and H. Chen, “Fast RDMA-based ordered Key-Value store using remote learned cache,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 117–135. [Online]. Available: <https://www.usenix.org/conference/osdi20/presentation/wei>
- [53] P. Li, Y. Hua, P. Zuo, Z. Chen, and J. Sheng, “{ROLEX}: A scalable {RDMA-oriented} learned {Key-Value} store for disaggregated memory systems,” in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 99–114.
- [54] B. Lu, J. Ding, E. Lo, U. F. Minhas, and T. Wang, “Apex: a high-performance learned index on persistent memory,” *Proc. VLDB Endow.*, vol. 15, no. 3, p. 597–610, Nov. 2021. [Online]. Available: <https://doi.org/10.14778/3494124.3494141>
- [55] Y. Wang, C. Tang, Z. Wang, and H. Chen, “Sindex: A scalable learned index for string keys,” in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 17–24. [Online]. Available: <https://doi.org/10.1145/3409963.3410496>
- [56] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, *LISA: A Learned Index Structure for Spatial Data*. New York, NY, USA: Association for Computing Machinery, 2020, p. 2119–2133. [Online]. Available: <https://doi.org/10.1145/3318464.3389703>
- [57] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, “Tsunami: A learned multi-dimensional index for correlated data and skewed workloads,” *Proc. VLDB Endow.*, vol. 14, no. 2, p. 74–86, oct 2020. [Online]. Available: <https://doi.org/10.14778/3425879.3425880>