Making In-Memory Learned Indexes Efficient on Disk

JIAOYI ZHANG, Institute for Interdisciplinary Information Sciences, Tsinghua University, China
KAI SU, Institute for Interdisciplinary Information Sciences, Tsinghua University, China
HUANCHEN ZHANG*, Institute for Interdisciplinary Information Sciences, Tsinghua University, China

Learned indexes have been demonstrated to outperform traditional ones in memory-resident scenarios. However, recent studies show that they fail to outperform B+tree when extended to disks directly. In this paper, we argue that it is feasible to create efficient disk-based learned indexes by applying a set of general transformations and optimizations to existing in-memory ones. Through theoretical analysis and controlled experiments, we propose six transformation guidelines applicable to various state-of-the-art learned index structures to fully leverage the characteristics of disk storage. Our evaluation shows that the indexes developed by applying our guidelines achieve a Pareto improvement in both throughput and space efficiency compared to the traditional B+tree and previous implementations of disk-based learned indexes.

CCS Concepts: • **Information systems** → **Data access methods**.

Additional Key Words and Phrases: Learned Indexes, Disk-based Indexes

ACM Reference Format:
Jiaoyi Zhang, Kai Su, and Huanchen Zhang. 2024. Making In-Memory Learned Indexes Efficient on Disk. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 151 (June 2024), 26 pages. https://doi.org/10.1145/3654954

1 INTRODUCTION

Learned indexes have been a major research topic for database indexing for the last five years [6, 8, 9, 11–13, 22, 23, 27, 30–33, 35]. These indexes claim to reduce index lookup latency by an order of magnitude and save the memory footprint by up to three orders of magnitude compared to traditional B+trees [13, 31]. To the best of our knowledge, however, none of the major online transaction processing (OLTP) database management systems (DBMSs) choose to adopt learned indexes. One of the key reasons is that most commercial OLTP DBMSs are still disk-based [2, 10, 14, 20, 21, 37] but existing learned indexes were designed to reside purely in memory.

Such a mismatch makes learned indexes less attractive to disk-based OLTP DBMSs. First, the nano-second latency improvement from learned indexes will likely be overshadowed by the much slower I/Os in a disk-based system [2]. Moreover, it is still impractical to fit all indexes in memory for these systems with learned indexes. Previous work has shown that index structures are often as large as the original tables in an OLTP DBMS, and most of them are secondary (i.e., non-clustering) indexes [34]. Although learned indexes are reported to be up to 2000× smaller than traditional tree-based ones [13, 31], their memory savings are insignificant for secondary indexes: the 2000×

*Huanchen Zhang is also affiliated with Shanghai Qi Zhi Institute.

Authors' addresses: Jiaoyi Zhang, jy-zhang20@mails.tsinghua.edu.cn, Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China; Kai Su, suk23@mails.tsinghua.edu.cn, Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China; Huanchen Zhang, huanchen@tsinghua.edu.cn, Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China.

ACM 2836-6573/2024/6-ART151
https://doi.org/10.1145/3654954

compression is only with respect to the internal nodes of a traditional index (e.g., B+tree), which is almost negligible compared to the size of the leaf nodes.

Therefore, to make learned indexes useful in the more general disk-based DBMSs, a natural idea is to only keep the learned models (i.e., internal nodes) in memory while storing the leaf nodes on disk – the so-called disk-based learned indexes. Note that most DBMSs lay out B+trees in a similar way by pinning non-leaf nodes in the buffer pool. However, recent experiments show that by simply locating the leaf nodes on disk, none of the state-of-the-art learned indexes can outperform a disk-resident B+tree across a broad range of workloads and datasets [14]. The conclusion seems to be that without a complete redesign from scratch, learned indexes are hardly preferable for databases operating on disk.

In this paper, we argue that efficient disk-based learned indexes can be obtained by performing a series of general transformations and optimizations to the existing in-memory ones. To facilitate this process, we propose the following SGACRU guidelines for converting a state-of-the-art in-memory learned index into a fast and memory-efficient disk-based learned index:

- **G1**: (**S**earch) Determine the leaf-page fetching strategy during the “last-mile” search based on disk and workload characteristics.
- **G2**: (**G**ranularity) Determine the prediction granularity (e.g., item vs. page) of the learned models.
- **G3**: (**A**lignment) Expand[1] the error bound (by leveraging page alignment) for index training to reduce the number of models while keeping the same expected I/O pages.
- **G4**: (**C**ompression) Reduce memory usage by compressing model parameters, accepting a negligible increase in CPU overhead compared to dominant I/O time.
- **G5**: (**R**obustness) For datasets with CDFs challenging to learn, fall back to an efficiently compressed zone-map index.
- **G6**: (**U**pdate) Use hybrid indexes [34] for efficient updates.

A disk-based learned index consists of in-memory models and on-disk leaf pages. The index’s performance is dominated by the I/O efficiency of fetching the leaf pages, while its memory efficiency depends on the model storage. The first two guidelines target minimizing disk I/Os. For **G1**, we found that whether to fetch the leaf pages “one-by-one” or “all-at-once” during the “last-mile” search could affect the learned index’s performance significantly. We recommend using the all-at-once strategy (i.e., fetching all the pages within the error range in one I/O request) by default because it leads to a lower lookup latency than the one-by-one strategy for disk-based learned indexes in most cases. However, with higher-end NVMe/Optane SSDs and more concurrent threads, the one-by-one strategy gains advantages.

For **G2**, conventional wisdom is to set the prediction granularity for disk-based learned indexes to the page level (i.e., the predicted value is a page ID) [14, 20]. Although we proved that the page-level prediction granularity could reduce the I/Os by up to one page per index lookup compared to the item-level prediction granularity, we found empirically that the page-level prediction granularity requires more models to guarantee the same error bound. And the performance gain (due to I/O reduction) is often not high enough to justify the extra memory consumption (due to more models).

The next two guidelines target reducing the memory footprint of a disk-based learned index. In **G3**, we observe that the fixed-length error bound used for index construction typically does not align with the page boundaries, causing the portion of the fetched page outside the error bound wasted. We, therefore, propose to expand the error bound dynamically to align with page boundaries during training to fully utilize the content in each fetched page. With a looser error bound, the learned index requires fewer models to achieve the same number of expected I/O pages per lookup.

[1]We use the term “expand” instead of “relax” because there is no reduction of precision.

We further propose to compress the in-memory learned models using lightweight compression algorithms in **G4**. Specifically, we apply the recent "learned compression" (LeCo) [18] which supports fast random access to the compressed elements in which we store our model parameters (i.e., keys separating the partitions, intercepts of the linear models). Such compression can reduce the model memory by up to 96% with a slight decoding overhead negligible in the end-to-end lookup latency.

Prior studies have reported datasets with unfriendly cumulative distribution functions (CDFs) for existing learned indexes to capture [31, 35]. In **G5**, we propose a fall-back design of learned indexes that are robust against such challenging workloads. The key idea is to compress the plain zone map (i.e., the minimum value of each leaf page) using the aforementioned learned compression (LeCo) technique. With further optimizations inspired by G3, our LeCo-based zone map exhibits superior performance and memory efficiency compared to a regular disk-based learned index, especially when dealing with challenging datasets.

Finally, we handle index updates with **G6**. Although several updatable learned indexes [6, 8, 9, 12, 32] have been proposed to support dynamic operations (i.e., inserts, updates, and deletes), their write performance is inferior to that of a B+tree when the leaf nodes are on disk [14]. Interestingly, we found that applying the general hybrid-index framework [34] to the disk-based learned index solves its update problem efficiently, outperforming the B+tree by 5.1× and 1.8× for write-only and read-write-balanced workloads, respectively with the same memory footprint. More importantly, the hybrid-index framework does not require the underlying learned index to be updatable, thus bypassing the challenge of designing updatable learned indexes specifically for disk.

Putting everything together, we developed the hybrid-PGM-disk and the hybrid-LeCo-disk as the example "end products" of applying the SGACRU guidelines to state-of-the-art in-memory learned indexes. Our evaluation using the YCSB benchmark [4] shows that hybrid-PGM-disk and hybrid-LeCo-disk can achieve the same performance as the B+tree for read-only workloads while consuming up to 83.5% less memory. For read-write-balanced workloads, our indexes outperform B+tree and the on-disk PGM index [14] by 1.8× and 16.8×, respectively with the same memory footprint. Besides, our two "end products" scale well, achieving up to 5× the throughput of the concurrent B+tree on write-only workloads.

We make the following contributions in this paper. First, we propose a set of guidelines that can make an existing in-memory learned index operate efficiently when located on disk. Second, we apply the guidelines progressively to various learned indexes and provide detailed analyses of the performance and memory trade-offs for each guideline. Finally, we affirm that the disk-based learned indexes developed in accordance with our guidelines are faster and more memory-efficient compared to traditional B+trees and previous implementations of disk-based learned indexes. Table 1 summarizes the notation used in this paper. Our implementation and experiment data are publicly available.[2]

2 BACKGROUND AND RELATED WORK

2.1 In-Memory Learned Indexes

Learned indexes address the challenge of indexing by focusing on its key operation: rank. Essentially, the rank operation is a function that maps a given key to a corresponding memory address. This can be conceptualized further as a Cumulative Distribution Function (CDF) operating over the distribution of keys.

Although the designs in various learned indexes are different, they generally adhere to a common search process and overall structure. Given the keys to be indexed, these learned indexes use their

[2]https://github.com/JiaoyiZhang/Efficient-Disk-Learned-Index

Table 1. Notations.

Notation	Meaning
P	number of data points (i.e., items) within a disk page
R_i	number of items in the last-mile search
G	prediction granularity (#keys sharing the same y)
ϵ	maximum model error (in items)
ϵ'	maximum model error (in G)
y	true position of the given key (in items)
$\hat{y}$	predicted position of the given key (in items)
R_p	number of pages in the last-mile search
$\mathbb{E}[R_p]$	average of R_p over all queries

own construction algorithm to obtain tree-like structures, comprising a root node, single or multiple layers of internal nodes, and leaf nodes used to manage data. Note that some updatable learned indexes store data alongside models in leaf nodes, whereas static learned indexes only store models that manage data points in external sorted arrays. Each node typically consists of a linear model[3] employed to capture the distribution of a subset of the dataset, along with associated parameters (e.g., slope, intercept). Detailed designs may vary across different learned indexes.

When a query arrives, the search process begins at the root node, where the model determines the location of the next node to be visited in the subsequent layer. This process continues until a leaf node is reached, and the model in the leaf node predicts the location of the querying key, denoted as $\hat{y}$. Finally, a search for the precise location (i.e., last-mile search) is conducted within this data partition (the sub-dataset managed by this leaf node). Typically, with the maximum-allowed model error ϵ, the search range is $[\hat{y}-\epsilon, \hat{y}+\epsilon)$, with the number of involved items $R_i = (\hat{y}+\epsilon)-(\hat{y}-\epsilon) = 2\epsilon$. In contrast to the $O(\log n)$ complexity of the binary search process in a B+tree, model prediction offers $O(1)$ lookup complexity to quickly determine the location of the next node or data point.

Different variants of learned indexes have been proposed since the pioneering work by Kraska et al. [13] FITing-Tree [9] replaces the leaf nodes of B+tree [3] with linear models, while PGM-Index [8] uses an optimal algorithm to build the index structure with the minimum number of models. Updatable learned indexes such as ALEX [6] and LIPP [32] employ gapped arrays and handle prediction collisions by growing to a tree of models. RadixSpline [12] uses buckets (as in bucket sort and radix sort) to optimize index build and lookup time with the cost of higher memory consumption. Other learned index variants [7, 11, 16, 17, 19, 24–26, 28, 29, 33, 36] are designed for specific application scenarios.

2.2 Disk-Based Learned Indexes

Google Bigtable first proposed a disk-based learned index in their system. However, it does not allow inserts and introduces storage gaps due to misaligned page boundaries [2]. FILM [20] is another attempt at building a disk-based learned index, but it requires cold data detection and append-only inserts, limiting its applicable scenarios. To build general-purpose on-disk learned indexes, a recent study by Lan et al [14] extends existing in-memory learned indexes to the disk environment. Their approach is to pin the smaller learned models in memory while storing the data points (i.e., leaf nodes) on disk. As shown in Figure 1, according to the search range given by the model, the range of pages to be retrieved from the disk are $\left[\left\lfloor \frac{\hat{y}-\epsilon}{P} \right\rfloor, \left\lceil \frac{\hat{y}+\epsilon}{P} \right\rceil\right)$, where P is the

[3]The original learned index [13] and its subsequent work [23] are composed of simple neural networks, but the state-of-the-art learned indexes all use linear models.

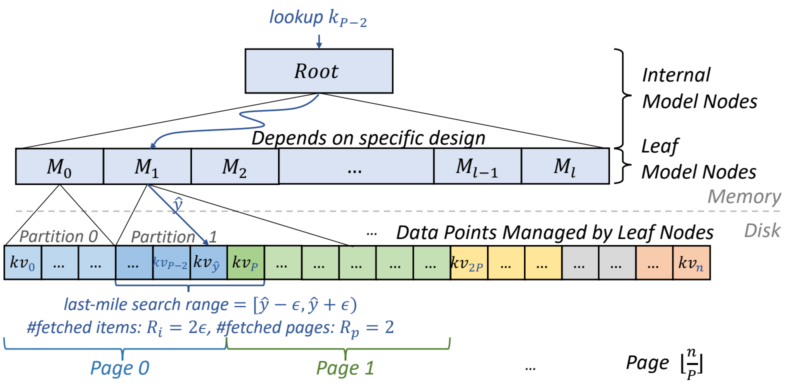

lookup k_{P-2}
Root
Internal Model Nodes
Depends on specific design
Leaf Model Nodes
M_0
M_1
M_2
...
M_{l-1}
M_l
Memory
Disk
$\hat{y}$
Partition 0
Partition 1
... Data Points Managed by Leaf Nodes
kv_0
kv_{P-2}
$kv_{\hat{y}}$
kv_P
kv_{2P}
kv_n
last-mile search range = $[\hat{y}-\epsilon, \hat{y}+\epsilon)$
#fetched items: $R_i = 2\epsilon$, #fetched pages: $R_p = 2$
Page 0
Page 1
Page $\lfloor \frac{n}{P} \rfloor$

Fig. 1. The Structure of Learned Indexes.

number of data points within a disk page. Then the number of I/O pages for a lookup query is:

$$R_p = \left\lceil \frac{\hat{y}+\epsilon}{P} \right\rceil - \left\lfloor \frac{\hat{y}-\epsilon}{P} \right\rfloor \quad (1)$$

However, the experimental results by [14] indicate that none of the disk versions of existing learned indexes can outperform a regular B+tree. We argue that the primary reason behind the above results is that the current design of disk-based learned indexes fails to leverage the characteristics of the disk storage. Unlike DRAM, SSDs and HDDs are block-addressable devices with a larger performance gap between sequential accesses and random accesses. Considering these device characteristics, we show in this paper that by applying a series of general transformations and optimizations to the existing in-memory learned indexes, they are able to outperform traditional B+trees and current on-disk learned indexes while consuming less memory at the same time.

3 MICROBENCHMARK SETUP

In this section, we outline the experimental setup for our microbenchmark to maintain clarity and consistency. In the following sections, we use this microbenchmark to validate our proposed general guidelines and follow the settings explained here unless otherwise stated. The index structures (e.g., learned models) are kept in memory because they are small and frequently accessed. The ordered data points are stored sequentially on disk. We enable direct I/O to bypass the OS page cache when fetching the data points.

Hardware and Platform. Our microbenchmarks are conducted on a machine with 128 Intel® Xeon® Platinum 8358 CPUs (2.60GHz) and 503GB of RAM. Besides, we also provide four disks on this machine, as shown in Table 2. The random read performance across these disks displays differences that span orders of magnitude. The differences are also pronounced when compared to memory, which exhibits sub-microsecond latency.

Datasets and Workloads. We use four real-world datasets in SOSD [22] commonly employed in current studies of learned indexes [6, 14, 31, 32, 35]. Additionally, we generate five synthetic datasets using the generator provided by GRE [31]. The generator creates datasets that satisfy both local hardness (l) and global hardness (g). The local hardness l refers to the number of models required in PGM-Index to construct an index for the dataset, and the maximum errors of all these models do not exceed ϵ_l. The global hardness g is the number of needed models under a more relaxed requirement of the maximum model error ϵ_g. Here, we fix ϵ_l to be 8 and ϵ_g to be 512 and then vary the values of l and g to generate synthetic datasets. Each dataset is composed of 200 million 8-byte

Table 2. Information of Four Disks.

	Disk	**4KB Random Read**		**Sequential Read**
		IOPS	**Latency**	**Bandwidth**
Optane SSD	Optane P5800X	1.5 M	11.7 μs	7.2 GB/s
NVMe SSD	Samsung PM9A3	1.1 M	70 μs	6.9 GB/s
SATA SSD	Samsung 870 QVO	98 K	211.6 μs	560 MB/s
HDD	Seagate ST2000NX0253	7293	4.8 ms	141 MB/s

unsigned integer keys, with an 8-byte payload. Then, each page (4KiB) can accommodate 256 data points. Except for the wiki dataset, none of the datasets contain duplicate keys. We provide a brief description of these datasets: (1) **amzn**: the popularity of each book from Amazon; (2) **wiki**: the time ID of each edit from Wikipedia; (3) **osm**: the ID of embedded positions from Open Street Map; (4) **fb**: user IDs randomly sampled from Facebook; (5) **syn_g10_l1**: l is 1,000,000 and g is 100,000; (6) **syn_g10_l2**: l is 2,000,000 and g is 100,000; (7) **syn_g10_l4**: l is 4,000,000 and g is 100,000; (8) **syn_g12_l1**: l is 1,000,000 and g is 120,000; (9) **syn_g12_l4**: l is 4,000,000 and g is 120,000. These datasets are sorted by data difficulty, where synthetic datasets are used to assess the ability of learned indexes to handle more complex datasets.

Because we delay the handling of index updates in Section 9, the microbenchmark only includes lookups. A more comprehensive evaluation can be found in Section 10. For this microbenchmark, We first build the index with 200M keys and then randomly select 10M keys for lookup. We measure the throughput and memory usage (without the data array) for the index under evaluation.

Learned Indexes. We use the state-of-the-art PGM-Index [8] and RadixSpline [12] as representatives in Sections 4 to 8, as no updates are involved here. Updatable learned indexes are not included in this microbenchmark because they typically introduce gaps and additional functionality to support writes, leading to increased space usage and potential degradation in read performance.

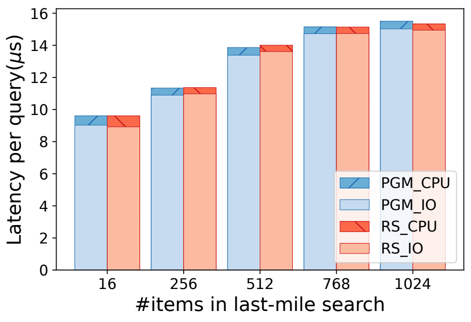
4 LEAF-PAGE FETCHING STRATEGY (G1)

The bottleneck of in-memory learned indexes typically arises from the last-mile search process, which involves multiple memory accesses [31, 35]. Consequently, rather than focusing on optimizing the internal structure and algorithms of learned indexes, our initial investigation focuses on understanding the impact of the last-mile search strategies on disk.

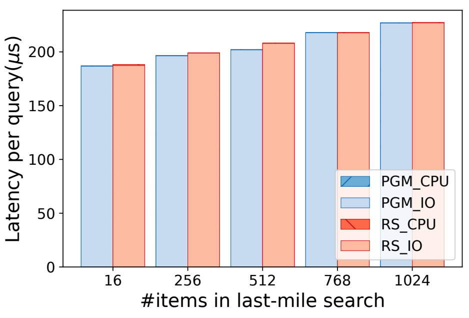
It is worth noting that the transition from in-memory to disk-based learned indexes shifts performance bottlenecks towards the I/O operations[14, 20]. We performed an experiment on the Optane and SATA SSDs: we built two indexes, PGM-Index and RadixSpline, on the Amazon dataset, then executed 10M read-only queries and recorded the CPU and I/O latency, respectively. As shown in Figure 2, the latency of these two indexes with different ϵ guarantees are all dominated by the I/O time on our disks. Notably, even on the highest-performing disk examined in our experiments, the CPU time remains significantly less than the time consumed by I/O operations. This shift implies a reordering of priorities, with I/O latency being more prominent than memory access latency. Most I/O operations come from the last-mile search in the bottom level (i.e., disk pages of leaf nodes) because the internal nodes are typically cached/pinned in memory.

4.1 Last-Mile Search On Disk

The last-mile search in leaf nodes usually consists of two phases: obtaining the search range from the linear model in the learned index and performing a binary search (or other search methods)



(a) Optane (single thread)



(b) SATA (single thread)

Fig. 2. CPU Time vs. I/O Time.

within this range. As previously shown in Equation 1, the number of I/O pages for each query is denoted as R_p. Here, we also define the number of pages between the start position $\hat{y} - \epsilon$ of the search range and the actual position y as *dist*, which is calculated by

$$dist = \left\lfloor \frac{y}{P} \right\rfloor - \left\lfloor \frac{\hat{y} - \epsilon}{P} \right\rfloor + 1 \tag{2}$$

With these two page-related statistics, we can more accurately control the I/O operations of disk-based learned indexes:

- ***dist***: When scanning from $\hat{y} - \epsilon$ and arrives at the page containing k_y, we only need to fetch the first *dist* pages, thus saving traffic for each query.
- $\boldsymbol{R_p}$: R_p is the maximum number of fetched pages per query. If R_p is small, we can take full advantage of sequential I/O by fetching all pages (containing $[\hat{y} - \epsilon, \hat{y} + \epsilon]$) in a single I/O, thereby reducing latency.

4.2 One-by-One vs. All-at-Once

Based on these two metrics, we have two leaf-page fetching strategies tailored to different scenarios: an **all-at-once** strategy and a **one-by-one** strategy. The former involves fetching all R_p pages from disk to memory in a single I/O operation, while the latter indicates fetching only the first few pages over a total of *dist* I/Os, with each I/O obtaining a single page.

Next, through simple experiments, we elucidate the particular scenarios where each leaf-page fetching strategy is the most preferable. We vary *dist* and R_p and assess the throughput of these two distinct strategies across multiple disks. There is no need to use a particular index here, as R_p and *dist* can be controlled via the length of a given range and selecting the correct position within this range, respectively. We use 1, 16, and 256 threads to complete these 10M queries on the machine equipped with four different disks as described in Section 3. The preferences for the two strategies at different combinations of *dist* and R_p are shown in Figure 3, where the color of each square corresponds to the ratio of the throughput of the one-by-one strategy to that of the all-at-once strategy and the throughput is the number of queries completed per second. A darker red indicates that the ratio is closer to or even exceeds 2.5, in which case the one-by-one strategy is preferred. Conversely, the darker blue color with a smaller throughput ratio suggests that the all-at-once strategy is more suitable. White in the middle signifies that both strategies are approximately equal in performance.

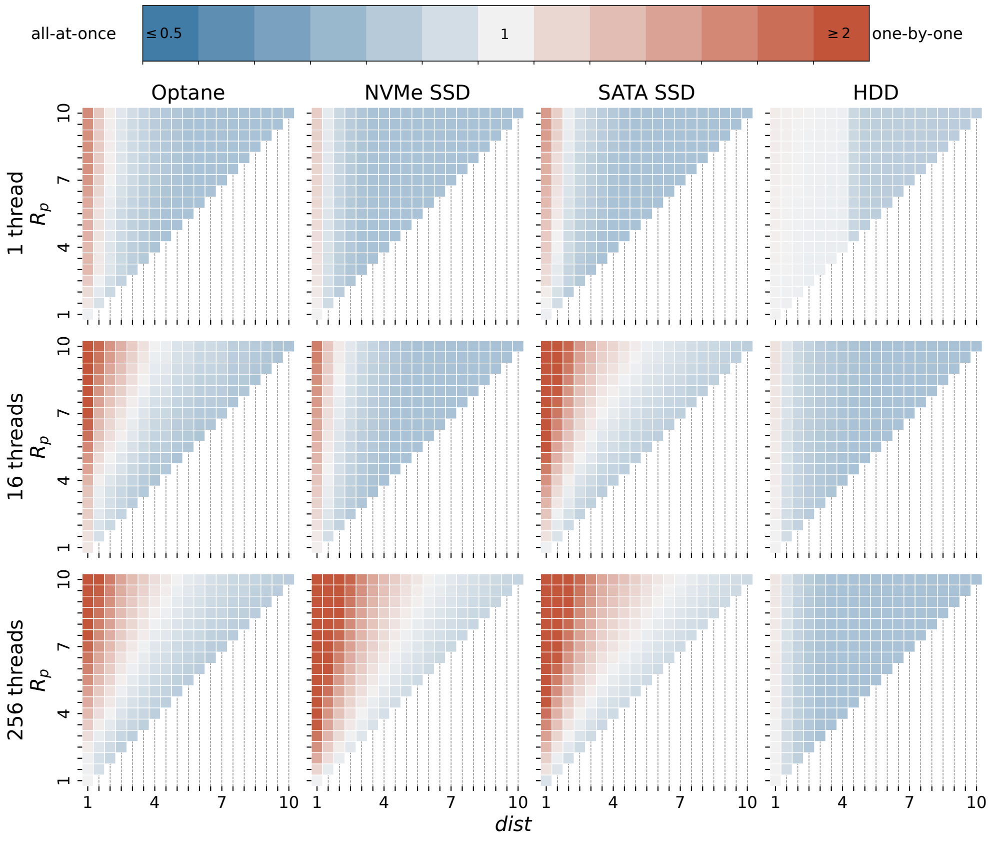


Fig. 3. Disk Performance: One-by-One Strategy (red) vs. All-at-Once Strategy (blue) – Darker color indicates that the corresponding strategy is more dominant in the corresponding pair of *dist* and R_p.

The diverse preferences for leaf-page fetching strategies can be attributed to various performance characteristics of disks and the number of threads. In scenarios where the access latency of individual I/Os is high and disk bandwidth is relatively "larger" compared to the latency, the all-at-once strategy is better. This strategy effectively mitigates latency through a single large I/O and ultimately results in higher throughput on the HDD. Note that SSDs have multiple channels and chips inside to implement various levels of parallelism mechanisms, allowing SSDs to read multiple pages in a single large I/O in parallel. Consequently, the all-at-once strategy can have higher throughput than the one-by-one strategy when the number of threads is small and *dist* is large. On the other hand, the one-by-one strategy excels when taking advantage of the low random access latency inherent to SSDs, especially when dealing with a small number of I/O pages that are less likely to saturate disk bandwidth even in multi-threaded scenarios.

In current disk-based learned indexes [14], the one-by-one strategy is used to retrieve data from disk, but it is often suboptimal. For example, we measure the throughput of PGM with the Facebook dataset on Optane after transitioning to the all-at-once strategy with all other settings unchanged. The results show that the throughput increased from 28.9 Kops/sec to 52.4 Kops/sec, with an increase of up to 1.81×, which verifies that selecting a suitable fetching strategy for a real-world scenario can significantly improve performance.

Discussion: We propose that when making decisions regarding leaf-page fetching strategies, it is necessary to consider specific workloads and performance characteristics of disks. In most cases, opting for the all-at-once strategy is an effective choice. However, when operating on SSDs with more threads, we recommend selecting the one-by-one strategy in scenarios where *dist* of a learned index is less than 1.25 pages or much smaller than R_p.

5 PREDICTION GRANULARITY (G2)

In the context of disk-based learned indexes, existing works usually choose to predict page IDs [13, 20]. Interestingly, according to our experiments, the intuition that page-level prediction granularity produces comparable or superior results than item-level predictions doesn't always hold. In this section, we explore the impact of varying prediction granularities on the space cost and throughput of learned indexes from theoretical analysis and experiments.

5.1 Theoretical Analysis

Prediction granularity, denoted as G, represents the number of data points sharing the same y. Then, the new error bound ϵ' in terms of G is calculated by $\epsilon' = \frac{\epsilon}{G}$, and the number of data points searched in the last mile can be represented as:

$$R_i = 2\epsilon = 2\epsilon' \cdot G \tag{3}$$

This indicates that when R_i is constant, the increase in G is traded by a decrease in the ϵ'. We assume that the queries follow a uniform distribution for simplicity, and we adopt the all-at-once page-fetching strategy. The distribution of the true positions within the predicted range also follows a uniform distribution. With these assumptions, we can establish a connection between G and the number of I/O pages R_p of a given range.

As R_i is a multiple of G, the predicted range is divided into $\frac{R_i}{G}$ parts, with each part typically introducing an expected $\frac{G}{P}$ new pages, except for the first part, which consistently introduces a new page. Then, the expected number of I/O pages $\mathbb{E}[R_p]$ is:

$$\mathbb{E}[R_p] = 1 + \sum_{t=1}^{\frac{R_i}{G}-1} \frac{G}{P} = 1 + \frac{R_i - G}{P} \tag{4}$$

According to Equation 4, we can deduce that as the prediction granularity increases, the average number of I/O pages to be visited decreases under the same last-mile search range. Besides, in the following experiments, we can set different parameters based on this formula to get the desired average number of I/O pages.

5.2 Experimental Results

To comprehensively evaluate the impact of prediction granularity on learned indexes, we vary G under the same R_i and test on different R_is. There are three categories of G in our microbenchmark:

(1) **Item Level ($G = 1$)**: The finest granularity possible.
(2) **Mini-Page Level ($1 < G < 256$)**: The intermediate category.
(3) **Page Level ($G = 256$)**: Data points on the same disk page have the same y.

We build RadixSpline, a learned index, for the Amazon and OSMC datasets on the NVMe SSD and execute the read-only workload using a single thread and 32 threads. The results are shown in Figures 4, where the x-axis represents the memory usage of RadixSpline, and the y-axis represents the throughput. We observe similar results with PGM-Index.

As illustrated in Figure 4, the throughput demonstrates an upward trend as the value of G increases under the same R_i. The reason is that if G is not an integer multiple of P, the accessed data

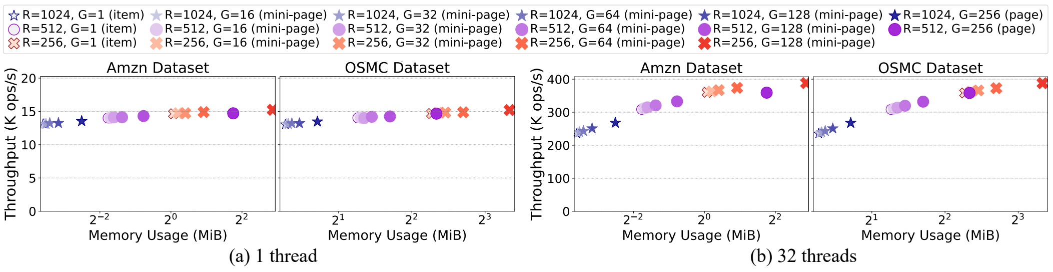


Fig. 4. Throughput and Space of RadixSpline with Different Prediction Granularity – Each symbol denotes a set of results with the same R_i. Within each set, the colors, transitioning from light to dark, indicate increasing values of G.

points do not align perfectly with the page boundaries, resulting in waste. Thus, the page-level G tends to obtain the highest throughput since it perfectly aligns with page boundaries. Nonetheless, according to Equation 4, the expected number of I/O pages between page-level G and item-level G differs by a maximum of one page. This marginal difference has a small impact on throughput in a single-threaded scenario, as the disk can provide similar throughput on these two settings with the all-at-once page-fetching strategy. However, in a multi-threaded scenario, where the bandwidth is saturated, the page-level prediction granularity can enhance throughput by a factor of 1.16 compared to the other option.

However, this improvement in throughput comes at a significant cost in space, resulting in a 6.94× increase in memory footprint, as shown in Figure 4. The reason is that an increase in G corresponds to a decrease in ϵ', as illustrated in Equation 3, imposing higher demands on model accuracy. Note that the marginal cost of learned models is high when $\epsilon' \longrightarrow 0$, which is the case at the page level. Thus, regarding space efficiency, a page-level granularity is often not a practical choice as it inevitably results in increased space costs, and it is better to maintain the item-level prediction granularity.

Discussion: Considering all these factors, we do not recommend practitioners replace item IDs with page IDs when transitioning from memory to disk. Instead, we advocate choosing the item level in most cases. When the saved page brings significant speedup in multi-threaded scenarios, then the mini-page and page-level prediction granularity are suitable choices.

6 ERROR BOUND ALIGNMENT (G3)

The build process for most learned indexes follows a common pattern[4]: given an error bound ϵ representing the maximum distance between the actual and predicted positions of a key, group all data points satisfying this error bound into the same partition and use a linear model to approximate their cumulative distribution function. In other words, a new model is needed when the error of a data point exceeds 2ϵ.

This section introduces a general modification to the core build algorithms of learned indexes following the aforementioned pattern to adapt to disk characteristics. The key principle is to expand the error bound without compromising throughput performance.

6.1 Disk-based Zero Intervals

As the scenario transitions from memory to disk, throughput mainly depends on the average number of I/O pages. Given that each I/O reads an entire page, the error bound may not align precisely with

[4]Here we set the prediction granularity to be item-level, i.e., G is 1 and ϵ is $\frac{R_i}{2}$.

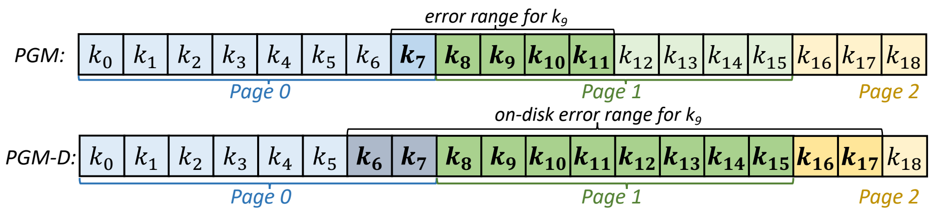

error range for k_9
PGM:
k_0 k_1 k_2 k_3 k_4 k_5 k_6 k_7 k_8 k_9 k_{10} k_{11} k_{12} k_{13} k_{14} k_{15} k_{16} k_{17} k_{18}
Page 0
Page 1
Page 2
on-disk error range for k_9
PGM-D:
Page 0
Page 1
Page 2

Fig. 5. An Example for Disk-based Error Range.

page boundaries, and some fetched pages may be partially wasted. Thus, fully utilizing the wasted portion of pages offers an opportunity to reduce memory usage while maintaining a constant average number of I/O pages. Intuitively, a construction algorithm that integrates page information can make full use of the wasted portion. Specifically, we can expand the error bounds during training to dynamically align with page boundaries, which reduces the chance of adding new models when dealing with partitions.

We propose a new strategy to achieve this, which can be applied to many existing learned indexes. The crucial step is incorporating page information into the original build algorithms, i.e., altering the error calculation process. First, let us introduce the concept of ***disk-based zero intervals***. When the predicted and true positions reside on the same page, the error between them should be recognized as zero instead of the actual difference. Then, the zero interval contains any position within the page, represented as $[z_l, z_r] = \left[\lfloor \frac{y}{P} \rfloor * P, \left(\lfloor \frac{y}{P} \rfloor + 1\right) * P - 1\right]$, and the new error is:

$$error = \begin{cases} z_l - \hat{y}, & if\ \hat{y} < z_l \\ 0, & if\ z_l \le \hat{y} \le z_r \\ \hat{y} - z_r, & if\ \hat{y} > z_r \end{cases} \tag{5}$$

With this disk-based zero interval, we can obtain a new maximum tolerable error range for each key of $[z_l - \epsilon, z_r + \epsilon]$, within which the model can make predictions.

We integrate this looser error bound into the core build algorithm in a learned index **X**. Equipped with the disk-based zero interval, the original build algorithm in **X** can then partition data points with fewer models and obtain a more compact **Disk**-based index, denoted as **X-Disk**. Note that we only alter the length of the error range during construction rather than the length of the last-mile search range when responding to queries. Hence, the original search method remains capable of accurately locating the page containing the given key within 2ϵ, requiring no further modifications.

It is sufficient to prove that the set of fetched pages contains the query k_y. Let the predicted position be $\hat{y}$ and the set of fetched elements covers $[w_l, w_r]$, where $w_l = \left\lfloor \frac{\hat{y}-\epsilon}{P} \right\rfloor * P$ and $w_r = \left(\left\lfloor \frac{\hat{y}+\epsilon}{P} \right\rfloor + 1\right) * P - 1$. If $\hat{y} > z_r$, then $error = \hat{y} - z_r \le \epsilon \implies w_l \le \lfloor \frac{z_r}{P} \rfloor * P = z_l \le y \le z_r < \hat{y} \le w_r$. The case $\hat{y} < z_l$ is similar and the case $z_l \le \hat{y} \le z_r$ is trivial.

We provide an example illustrating the change in the error range in Figure 5, where the ϵ for the PGM-Index is 2 and keys with the same color indicate that they are on the same disk page. The original algorithm assigns a tolerance range of $[k_7, k_{11}]$ to k_9, which are the keys with darker colors in Figure 5. This requires the position predicted by the current model in the build process to fall within the range of 5 keys; otherwise, a new model would be required to handle it. However, with our expanded errors, the predicted position can fall within a broader range of $[k_6, k_{17}]$, which can reduce the chance of adding a new model.

Disk-based zero intervals might seem no different from the effect of the page-level prediction granularity(G2) at first glance, both of which regard the distance between items within the same

Table 3. Models Saving of Learned Indexes Equipped with Disk-based Zero Intervals – Each column represents the results for the same expected number of I/O pages, which means that the corresponding throughput is guaranteed to be similar.

Expected Number of I/O Pages		**PGM-Index** 1.01	1.5	2	3	**RadixSpline** 1.01	1.5	2	3
Facebook Dataset	#Models (w/o)	27,442,813	531,431	258,373	120,493	31,451,334	1,023,185	504,121	245,665
	#Models (w/)	713,368	240,058	151,847	88,439	2,562,804	581,190	344,880	197,122
	% of Saving	97.4%	54.8%	41.2%	26.6%	91.8%	43.2%	31.6%	19.8%
Amazon Dataset	#Models (w/o)	25,088,692	82,804	22,914	5,997	27,378,554	230,602	68,357	18,705
	#Models (w/)	660,556	42,872	15,758	4,965	2,406,374	130,356	48,616	15,485
	% of Saving	97.4%	48.2%	31.2%	17.2%	91.2%	43.5%	28.9%	17.2%

page as zero. The substantial difference is that G3 expands ϵ without relaxing R_p, but in G2, $R_p \propto \epsilon$ according to Equation 3 & 4. To illustrate this, consider Figure 5, where $\epsilon = 2$. The distance between k_8 and k_{15} is zero, implying that $G = 8$. When the predicted position $\hat{y} = 7$ (which is in Page 0), the maximum possible page is $\left\lfloor \frac{\hat{y}+\epsilon}{G} \right\rfloor = 1$, so the granularity-level error in G2 is $\epsilon' \geq 1$. This means that, even if $\hat{y} = 5$, the rightmost page we need to search for is $\left\lfloor \frac{\hat{y}}{G} \right\rfloor + \epsilon' \geq 1$, while in G3 we only need to search for Page 0.

6.2 Applying to Learned Indexes

We apply this simple yet effective strategy to both PGM-Index and RadixSpline. Table 3 shows the number of models with and without this strategy on the Facebook and Amazon datasets[5], along with the percentage reduction in the number of models. The columns represent the results under various constraints on the expected number of I/O pages, which can be controlled by Equation 4.

As shown in Table 3, in most cases, the indexes with expanded errors yield a decrease in the number of models ranging from 17.2% to 97.4%. This verifies that fully utilizing page information during the build process significantly contributes to model reduction, and importantly, it does not come at the cost of an additional increase in the average number of I/Os. Additionally, this effect diminishes as the average number of I/O pages increases, because the larger ϵ is, the smaller the wasted proportion of the last-mile search range.

Notably, for all scenarios examined, the number of models required for PGM-Index equipped with disk-based zero intervals is consistently lower than that for RadixSpline under the same setup. This is expected because the **X-Disk** versions inherit the fundamental construction algorithm from their original version **X** while achieving a more space-efficient layout. It's worth noting that our algorithm demonstrates remarkable potential for reducing the number of models in PGM-Index in disk-based environments, a learned index known for its minimal number of models [8, 31].

7 COMPRESSION TECHNIQUES (G4)

In this section, we discuss the application of compression techniques on models to reduce their memory usage. Typically, each model consists of a partitioning key, slope, and intercept, where the partitioning key is the smallest key within the partition of each leaf node and serves to locate the correct leaf node. While compression for slope and intercept has been discussed in the compressed PGM [8], we focus on the uncompressed part: the partitioning keys of models. Although various general compression techniques can be employed for key compression, even with lightweight

[5]Facebook dataset is harder for learned indexes than Amazon dataset [31].

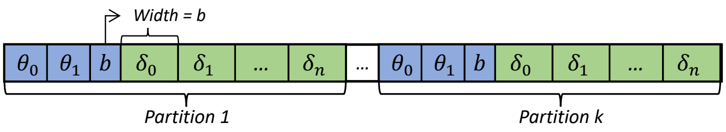

Width = b
θ_0 θ_1 b δ_0 δ_1 ... δ_n ... θ_0 θ_1 b δ_0 δ_1 ... δ_n
Partition 1
Partition k

Fig. 6. Storage Format of LeCo [18]

compression algorithms (e.g., Delta Encoding [1, 15]), decompression still requires a certain amount of CPU time, resulting in a decrease in throughput for in-memory scenarios with random access. This is why key compression was omitted in the compressed PGM, and it is preferable to choose the uncompressed ones.

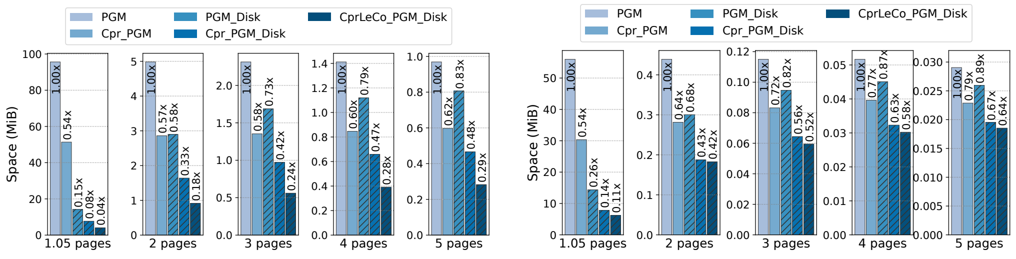
However, we argue that the situation changes when dealing with disk-based indexes, and that judiciously choosing the appropriate compression algorithm for each part can help improve performance. As depicted in Figure 2, CPU time accounts for a negligible fraction of the overall query time, and even if CPU time doubles to support random access, the performance would still be primarily determined by I/O operations. This observation gives compression techniques a greater advantage. Meanwhile, we still aim to ensure that CPU time remains within reasonable limits.

During the investigation, we discovered a new learned compressor, called LeCo, that aligns well with our requirements. First, let's provide a brief overview of LeCo, and more details can be found in [18]. LeCo, as a learned compressor, follows a similar construction process of learned indexes: the dataset is divided into multiple partitions, and a linear model (denoted as θ_0 and θ_1) is applied to each partition. To ensure lossless compression, which allows for the accurate recovery of each key, LeCo must retain specific error values of the learned model for each key. As shown in Figure 6, the error of each element is stored in an array of δ, each occupying b bits. The better the linear model fits, the smaller b can be set, thus leading to better compression. LeCo exhibits strong compression capabilities for monotonic value sequences and supports random access efficiently. Consequently, LeCo can be readily applied to compress the array of keys in our models (i.e., partitioning keys), making it a superior choice compared to other compression techniques. Thus, we recommend employing LeCo to compress the smallest key of each model, thus saving memory usage without sacrificing search performance.

In summary, the methods that can be used for each part are:

- **Partitioning Key:** Use LeCo to compress the smallest keys used to locate leaf nodes.
- **Slope:** If the index permits the slope to be any value in an interval within which all values have the same ϵ, the compression method in compressed PGM is a good choice. Otherwise, the general compression methods can be applied.
- **Intercept:** Use the strategy in the compressed PGM to make intercepts increase, and then, succinct data structures or LeCo can be applied to compress them.

Application Examples: Following the guidelines for compression methods, we apply compression techniques to PGM indexes, which is currently the space-optimal learned index. The effectiveness is then validated through five indexes: (1) the original PGM, (2) the compressed PGM-Index in [8], (3) PGM_Disk, (4) PGM_Disk with **Com**pr**essed slopes and intercepts (denoted as Cpr_PGM_Disk), (5) and the fully compressed index (CprLeCo_PGM_Disk). More specifically, PGM_Disk comprises models derived from the construction algorithm of PGM-Index but enhanced with the dynamically expanded error bounds (G3) without compression. Cpr_PGM_Disk employs the compression methods utilized in the compressed PGM-Index to compress slopes and intercepts, and CprLeCo_PGM_Disk adopts LeCo to compress the key of each model. The memory usage of these indexes is presented in Figure 7 with five groups of expected I/O pages on both the Facebook



(a) Facebook Dataset (b) Amazon Dataset

Fig. 7. Memory Usage of All Indexes – The number above each bar is the ratio of index memory usage to the memory usage of PGM.

Table 4. Memory Usage: Compressed Learned Indexes vs. Zone Maps – The throughput is similar for all these indexes in this experiment.

	fb	**amzn**	**wiki**	**osmc**
Cpr_PGM	51.36 MiB	30.34 MiB	9.48 MiB	34.58 MiB
CprLeCo_PGM_D	4.05 MiB	6.20 MiB	3.17 MiB	6.58 MiB
Zone Map	5.96 MiB	**5.96 MiB**	5.96 MiB	**5.96 MiB**
LeCo-based Zone Map	**2.09 MiB**	**4.22 MiB**	**1.05 MiB**	**4.59 MiB**

and Amazon datasets. Overall, CprLeCo_PGM_Disk consistently maintains the smallest memory footprint within these five indexes, and its minimum memory usage is only 4% of the original size in PGM. This illustrates the effectiveness of our general guidelines of leveraging the disk-based zero intervals (G3) and applying compression techniques (G4) to all components of models. Our guidelines are able to significantly reduce both the number of models by extending their error ranges based on disk information and the average amount of memory footprint needed per model through compression techniques.

8 FALL-BACK DESIGNS OF LEARNED INDEXES (G5)

The combined use of compression techniques and expanded error bounds based on disk pages provides a practical method for reducing the memory usage of disk-based learned indexes, and our CprLeCo_PGM_D[6] demonstrates superior spatial efficiency. Nevertheless, in certain scenarios, CprLeCo_PGM_D may still occupy more space than the zone map that directly stores the minimum key value of each page. Furthermore, we propose a LeCo-based zone map index that makes this phenomenon even more apparent.

First, let's introduce our fall-back design of learned indexes. The key idea is to apply LeCo to compress the zone map, which contains the minimum keys of each leaf page. Specifically, we use LeCo to store these minimum key values of leaf pages in a compressed manner and implement the binary searches by utilizing the interface provided by LeCo. In other words, when we need to compare the key at position i of the LeCo-based zone map with the given key, we input i to LeCo and then obtain the recovered key used for the comparison, thus completing a step of the binary search process. Given that the zone map index requires $\log_2\lceil \frac{n}{P} \rceil$ comparisons for a query, the LeCo-based zone map also needs $\log_2\lceil \frac{n}{P} \rceil$ decompression, where $\lceil \frac{n}{P} \rceil$ is the number of leaf

[6]We use CprLeCo_PGM_D to denote CprLeCo_PGM_Disk in the following description.

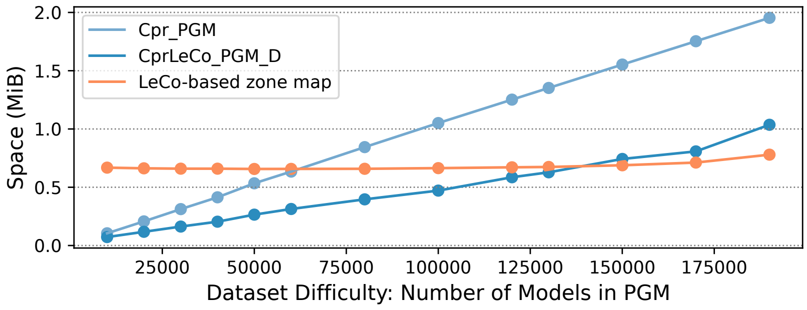


Fig. 8. Memory Usage on Datasets of Varying Difficulty: Learned Indexes vs. LeCo-based Indexes – The expected number of I/O pages in each index is three.

pages. As LeCo supports fast random access, this fall-back index can provide compact structures with only a negligible increase in CPU latency. As shown in Table 4, LeCo-based zone map achieves a more compact structure, further lessening the competitive edge of learned indexes.

8.1 Challenging Scenarios

To investigate the challenging scenario for learned indexes, we leverage the generator introduced in Section 3 to obtain numerous synthetic datasets that require different numbers of models for PGM to handle, since the memory footprint of learned indexes is basically determined by the number of models. We then use two compressed learned indexes and the LeCo-based zone map to build them and show their space cost in Figure 8.

Even with our CprLeCo_PGM_D, the advantages in memory usage still diminish when the number of models grows significantly. As shown in Figure 8, in scenarios where the expected number of I/O pages is 3, if the PGM demands fewer than 140,000 linear models for a dataset, then CprLeCo_PGM_D is a suitable option. Conversely, these are challenging cases for learned indexes, where LeCo-based zone map becomes the preferable choice. Thus, this guideline is intended to help practitioners make more informed decisions when using learned indexes and to provide a robust fallback design for learned indexes when they require many models to handle a dataset.

8.2 Further Optimization on LeCo-Zonemap

To further optimize LeCo-based zone maps, we also apply the idea of expanded error bounds to it.

8.2.1 LeCo-Zonemap-D. When we use LeCo to compress the minimum key within each page, our primary requirement is that LeCo can provide us with the value at a given position. We then use this value to compare with the input key to identify the block that stores the given key. An interesting observation is that we do not actually require LeCo's compression to be completely lossless here; our main concern is to ensure that the final results of binary searches remain consistent with the outcome of the uncompressed zone map. Hence it is sufficient to store some "rough" separators.

Consider a sequence of keys to be compressed as an example: $S = \{1, 4, 9, 16, 17\}$. Let's assume that the model predicts a sequence of values: $\{0, 5, 10, 15, 20\}$, and the maximum error is calculated as 20 - 17 = 3. Then, LeCo requires $\log_2(3 + 1) * 5 = 10$ bits to store all the errors. Now, consider a scenario where the predicted sequence is unchanged, and S is modified to $S' = \{1, 5, 10, 16, 20\}$, which remains the same searching results as S when finding the first position greater than or equal to the value for each value in S. In this case, only $\log_2(1 + 1) * 5 = 5$ bits are required to store the errors, resulting in significant memory savings.

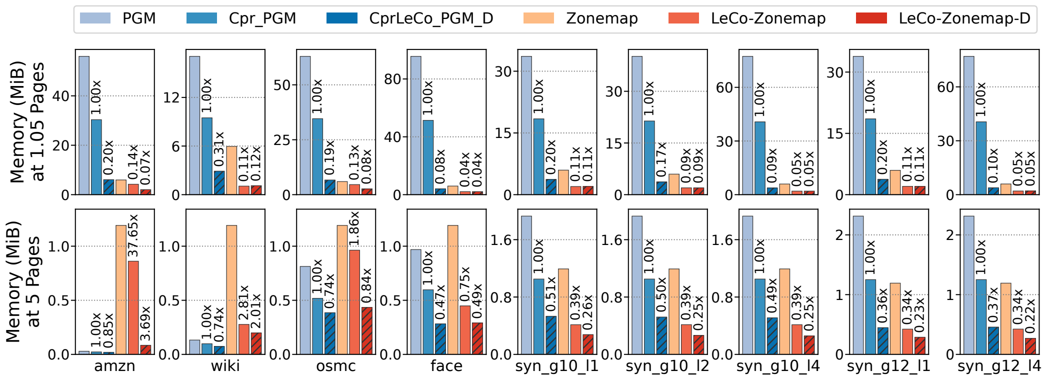


Fig. 9. Memory Usage: PGM-based Indexes vs. Zonemap-based Indexes – The difficulty of datasets increases from left to right. The first row shows the results for indexes with expected I/O pages of 1.05, and the second row shows the results at 5 pages.

Hence, we can apply a similar procedure in Section 6 to mitigate the overall space cost of LeCo-based zone map. This involves providing new error ranges for each data point during constructing LeCo, thereby reducing the magnitude of the largest error in each model. Among them, the lower bound of each zero interval is the value of the current key, and the upper bound is smaller than the value of the next key to be compressed. We refer to this optimized version as LeCo-Zonemap-D, which incorporates disk-page information.

8.2.2 Results. Until now, we have obtained the most space-efficient indexes among the two types: CprLeCo_PGM_D for learned indexes and LeCo-Zonemap-D for zone maps. This makes it easy to handle both model-friendly and challenging datasets. It is worth noting that LeCo is a learned method, so both types are fundamentally learned indexes designed for different scenarios.

We evaluate them with two different R_p (1.05 and 5 pages) on nine datasets, and the results in Figure 9 clearly illustrate two challenging scenarios for learned indexes. First, when strict high-throughput requirements exist, learned indexes need more models to ensure accurate predictions, leading to a larger memory footprint. In situations where the expected number of I/O pages is as low as 1.05, zonemap-based indexes dominate due to the large marginal cost of learned indexes when $\epsilon \longrightarrow 0$. Second, datasets that are unfriendly to linear models pose difficulties to learned indexes because a small number of linear models cannot effectively capture the data distribution. Even when the error bound is increased to five pages (i.e., 1280 data points), CprLeCo_PGM_D, which has the smallest space cost among learned indexes, still requires 1.68× the memory footprint of LeCo-Zonemap-D on the syn_g12_l4 dataset.

It is worth noting that the two indexes, CprLeCo_PGM_D and LeCo-Zonemap-D excel in scenarios that align with their favorable index categories: SOSD datasets for learned indexes[7] and five synthetic datasets for zone maps. Compared to Cpr_PGM, CprLeCo_PGM_D reduces the memory footprint by up to 92% (greater than 45% in all but three of the 18 cases). LeCo-Zonemap-D, developed based on expanded error bounds, also reduces the space cost of LeCo-Zonemap, especially amplifying the advantages of LeCo-Zonemap in its dominant cases. These results reflect that judicious adoption of indexes can effectively improve the robustness.

[7]The real-world datasets in [31] are also friendly to learned indexes.

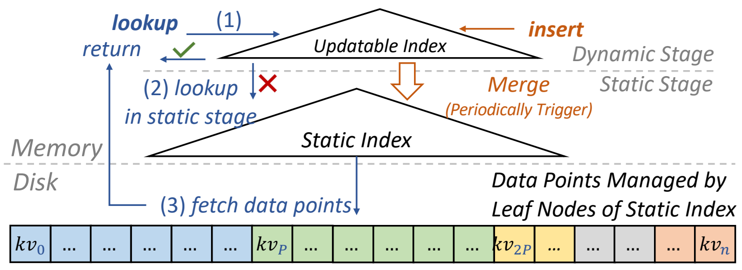


Fig. 10. Hybrid Learned Index Framework.

9 SUPPORTING UPDATES (G6)

9.1 Hybrid Learned Index Framework

The original hybrid index aims to improve the memory efficiency of an in-memory database with a minimal performance compromise [34]. It batches newly inserted/updated index items in a write-optimized structure (i.e., the dynamic stage). When the size of the dynamic stage reaches a threshold, the items are merged into the much larger static stage where the index structure is highly compact and optimized for lookups.

We extend the original in-memory hybrid index to a framework to support updates efficiently for an arbitrary on-disk learned index. As shown in Figure 10, the dynamic stage includes a regular in-memory index (e.g., an in-memory B+tree) to hold both keys and values of recent updates. The static stage is made of an on-disk learned index described in the previous sections (e.g., CprLeCo_PGM_D) where the index key-value pairs are stored compactly on disk, while the learned models are kept in memory. To process a lookup, the hybrid learned index first checks the dynamic stage and returns the result if the given key is found. Otherwise, it searches the static stage by using the in-memory learned models to locate the leaf page(s) on disk that might contain the querying key.

All write operations (i.e., insert, update, delete) are carried out in the dynamic stage. As the size of the dynamic stage grows, a merge routine is triggered periodically to merge all the newly inserted data to the disk. The merge process works as follows. It scans the leaf pages of the static learned index on disk and merges the key-value pairs from the dynamic stage with a simple linear algorithm similar to the merge step in a merge-sort. New learned models are then trained on the merged leaf nodes before they are written back to the disk. Finally, the dynamic stage is cleared and continues to accept queries and updates. We next discuss two important aspects of the merging strategy in our hybrid learned index framework.

Merge Trigger. A *constant trigger* refers to a fixed predefined memory budget for the dynamic stage, while a *ratio-based trigger* allows the maximum size of the dynamic stage to grow with respect to the size of the static stage to maintain a constant ratio (e.g., 1:100). [34] proved that the amortized merge overhead remains constant over time with a ratio-based trigger, and it is thus a preferred merge trigger for our hybrid learned index framework.

To ensure a fair comparison to the B+tree (whose inner nodes are pinned in memory) in the evaluation, we set the merge ratio to R so that the maximum size of the dynamic stage + the model size in the static stage never exceeds the size of the B+tree inner nodes (i.e., the memory footprint of the hybrid learned index is less than or equal to that of the B+tree). Given that the B+tree we use has a fanout of 256, we configure $R = 178$ by default in the subsequent experiments unless specified otherwise.

Merge Concurrency. In a single-threaded environment, normal index operations are blocked during a merge, causing a high tail latency for queries. We, therefore, implemented a non-blocking

Table 5. Throughput and Space Cost of Disk-based Indexes Under Balanced-Write Workload on Facebook Dataset –Memory usage of hybrid indexes includes space for the dynamic stage and the index in the static stage. A smaller static index allows more space for newly inserted keys under the same memory usage.

	Hybrid-Index Structure		**Baseline**
Dynamic Type in Hybrid	ALEX	ALEX	B+tree
Static Type in Hybrid	CprLeCo_PGM_D	Cpr_PGM	
Throughput (k ops/s)	25.66	24.28	14.11
Disk Space (MiB)	2358.22	2358.7	3291.79
Memory Usage (MiB)	9.20	9.26	9.79

and lock-free merge algorithm for our hybrid learned index framework with multiple threads. When the index enters the merge phase, we first make the current dynamic stage immutable and create a new empty dynamic stage in front of it to continue accepting writes. The background threads then start rebuilding the learned index in the static stage to incorporate the new items in the immutable dynamic stage, as described above. The key range is partitioned equally according to the number of threads so that each thread works on a single partition independently. Because the merge follows copy-on-write instead of update-in-place, it does not need to acquire locks to synchronize with the query threads. Once the concurrent merge completes, it updates the globally visible version number using a Compare-And-Swap (CAS) operation and garbage collects the old stages when their reference counts become zero. Note that a query needs to look up three stages (i.e., the new dynamic stage, the immutable dynamic stage, and the static stage) in sequence during the merge phase.

Currently, the dynamic stage is not persistent. It must rely on the write-ahead-log (WAL) of the underlying database to recover upon crash. Alternatively, the index can have its own WAL to speed up recovery.

9.2 Experimental Results

We conduct single-threaded experiments using two examples on a balanced workload (i.e., 50% inserts and 50% reads). The first hybrid learned index utilizes ALEX, an in-memory updatable learned index, as the dynamic index and CprLeCo_PGM_D as the static index, while the other employs the compressed PGM as the static index. The search range R_p for static learned indexes is 1.5 pages. Since the existing work [14] has concluded that none of the learned indexes implemented on disk can compete with the disk-resident B+tree and the reserved gaps of updatable disk-based learned indexes do not offer better performance, we only select B+tree as the baseline here. More details and experiments can be found in Section 10.

Based on the results in Table 5, we observe that the hybrid learned indexes consistently outperform the baseline, demonstrating superior performance in terms of both time and space efficiency. Notably, the hybrid index using CprLeCo_PGM_D in the static phase achieves a throughput of 1.82× that of the B+tree, while consuming only 71% of the space cost of the baseline. The primary distinction between these two hybrid learned indexes lies in their static phases. When the memory usage is the same, the smaller CprLeCo_PGM_D in the static phase saves more space for the insert buffer in the dynamic phase, thereby enhancing throughput.

In summary, our hybrid framework offers practitioners a valuable opportunity to concentrate their efforts on designing more efficient in-memory learned indexes or specialized index structures tailored to other specific scenarios. This eliminates the need for extensive efforts in adapting in-memory learned indexes to disk, allowing for more efficient and effective index development.

Table 6. Summary of Parameters in YCSB Evaluation

Parameters	**Meaning/Implications**
Fetching Strategy	All-at-once strategy is adopted (G1)
Prediction Granularity	Item-level G is preferred based on G2
Search Range R_p	Smaller $R_p \Rightarrow$ faster lookup but more models
Merge Trigger Ratio (Single-Threaded YCSB)	Large ratios save memory via more merges. Set to 178, 222, 296, 445, and 890.

10 EVALUATION

In this section, we provide a comprehensive evaluation of all the general guidelines discussed above. We first present the YCSB-based experimental setup in Section 10.1. Then, we evaluate the hybrid learned indexes in single-threaded and multi-threaded scenarios in Sections 10.2 and 10.3, respectively. The space efficiency is shown in Section 10.4. We compare against FILM, a learned index designed for larger-than-memory scenarios, in Section 10.5. Finally, we evaluate the hybrid learned indexes using the TPC-C benchmark in Section 10.6.

10.1 Experimental Setup

We use the Facebook, Amazon, and OSM datasets from SOSD. The evaluation is conducted on the same machine in Section 3. The disk is PM9A3 NVMe SSD. Each dataset consists of 200 million 8-byte unsigned integer keys, each with an 8-byte pointer as the payload.

10.1.1 Workloads. Three YCSB benchmarks [4, 34] are used in our experiments: (1) **Read-Only:** YCSB-C workload with 100% read. (2) **Write-Only:** Obtained by YCSB-C workload, but with 100% insert. (3) **Balanced:** 50% insert, 50% read. This workload is derived from YCSB-A, and we replace updates with inserts since update operations are similar to read operations. We establish a one-to-one mapping between the YCSB keys and the keys from our datasets. We then replace the YCSB keys with our keys in the generated workloads, as in [34].

We first initialize the indexes with 150 million keys from the randomly shuffled 200 million keys in each dataset. Then, we carry out 10 million queries from YCSB following a uniform distribution. Note that all lookup keys already exist in the index, and all inserted keys are not duplicated. As described in Section 3, we use direct I/O to fetch the data points to bypass the OS page cache.

10.1.2 Metrics. Our evaluation includes three metrics: disk space, memory usage, and throughput. Memory usage is the peak memory consumption during the execution of 10 million queries for each index, and the disk space is the space used on disk after completing all queries, both measured in MiB. These two metrics indicate the space efficiency of indexes. Throughput is calculated as the total number of operations divided by the execution time.

10.1.3 Our All-in-One Disk-Based Learned Indexes. Following the general guidelines outlined earlier, we have developed two representative all-in-one index structures: **hybrid-PGM-disk** and **hybrid-LeCo-disk**. As illustrated in Table 6, we employ the all-at-once fetching strategy (G1) and item-level prediction granularity (G2) for them. Besides, in the static stage of these indexes, we use CprLeCo_PGM_D (G3, G4) and LeCo-Zonemap-D (G3, G5), both of which are equipped with expanded error bounds. The dynamic phases (G6) use B+tree to focus more on static structures since B+tree has the same performance on different datasets.

10.1.4 Baselines. We compare our two indexes with two baseline indexes, consisting of a learned index extended to disk by [14], **PGM**, as well as a traditional tree index, **B+tree**. We exclude other disk-extended learned indexes because previous study [14] showed that they are inferior to a regular disk-resident B+tree. We use open-source implementations for these indexes, which are implemented in C++. We modify the original implementation of B+tree to provide users with the capability to adjust the number of layers that are stored on disk. In our experimental setup, B+tree comprises four layers: three layers of internal nodes and one layer of leaf nodes. Consequently, B+tree offers three distinct settings in our YCSB evaluation to represent which layers are stored on disk: (1) Leaf nodes only. (2) Leaf nodes and the bottom layer of internal nodes. (3) All layers except the root node. In our evaluation of PGM, we employ the same parameters as those utilized for our all-in-one indexes. We measure PGM at five settings of $\mathbb{E}[R_p]$, including 1.05, 2, 3, 4, and 5, respectively. For multi-threaded experiments, we try our best to extend the single-threaded **B+tree** to support concurrency through reader-writer locks and optimistic insertion with a bitmap.

10.2 Single-Threaded YCSB Evaluation

10.2.1 Performance on Static Workload. We first focus on evaluating the lookup performance of indexes without the impact of updates. As shown in Figure 11, on the read-only workload, both our all-in-one indexes provide a Pareto improvement to the throughput and memory usage on all the datasets.

First, our indexes consistently have higher throughput than PGMs for the same length of the last-mile search. This is due to our choice of the all-at-once strategy based on **G1**, enabling our hybrid indexes to achieve higher throughputs. In contrast, PGM experiences multiple I/O operations in the last-mile search, leading to performance degradation. Second, compared to the B+tree, our indexes can achieve the same maximum throughput with far less memory usage than its internal nodes. The reason is that we have reduced memory usage in two ways: by reducing the number of models and compressing the representation of models. As a result, our all-in-one learned indexes necessitate a minimum of 16.4% of the memory footprint of B+tree to deliver the highest throughput.

The reason why the results of hybrid-PGM-disk and hybrid-LeCo-disk look similar is that the gap between them is relatively small compared to the entire range on the x-axis. eActually, hybrid-LeCo-disk utilizes more space than hybrid-PGM-Disk in most cases. Furthermore, LeCo also employs linear models to learn data distribution, which is essentially the same approach as learned indexes. Hence, it is not surprising that they exhibit similar performance.

10.2.2 Performance on Dynamic Workloads. Two dynamic workloads are used to test the update performance of indexes.

Write-Only Workload. We allocate various memory budgets to accommodate future inserts for our hybrid learned indexes, and $\mathbb{E}[R_p]$ in the static phase is 5 pages since there are no read queries here. As depicted in Figure 11, our recommendation to use a hybrid-index structure, rather than expanding or designing a dedicated updatable learned index for disk, effectively addresses the inefficiency in supporting updates of existing disk-based ones [14]. Consequently, the learned indexes derived from our general guidelines can achieve up to 14.04× the throughput of the B+tree without consuming additional memory space. The throughput of our indexes already incorporates the effect of a single-threaded merge process that would block other queries. This implies that a batch of inserts can perform better than multiple smaller inserts. Note that the throughput of our indexes might decrease as the memory budget decreases, which is reasonable because the reduced space allocated to hybrid indexes necessitates more merge operations. Nevertheless, it's worth noting that even with a small memory usage of only about 2MiB, both the hybrid-PGM-disk and the hybrid-LeCo-disk can still maintain approximately the same or even higher throughput as

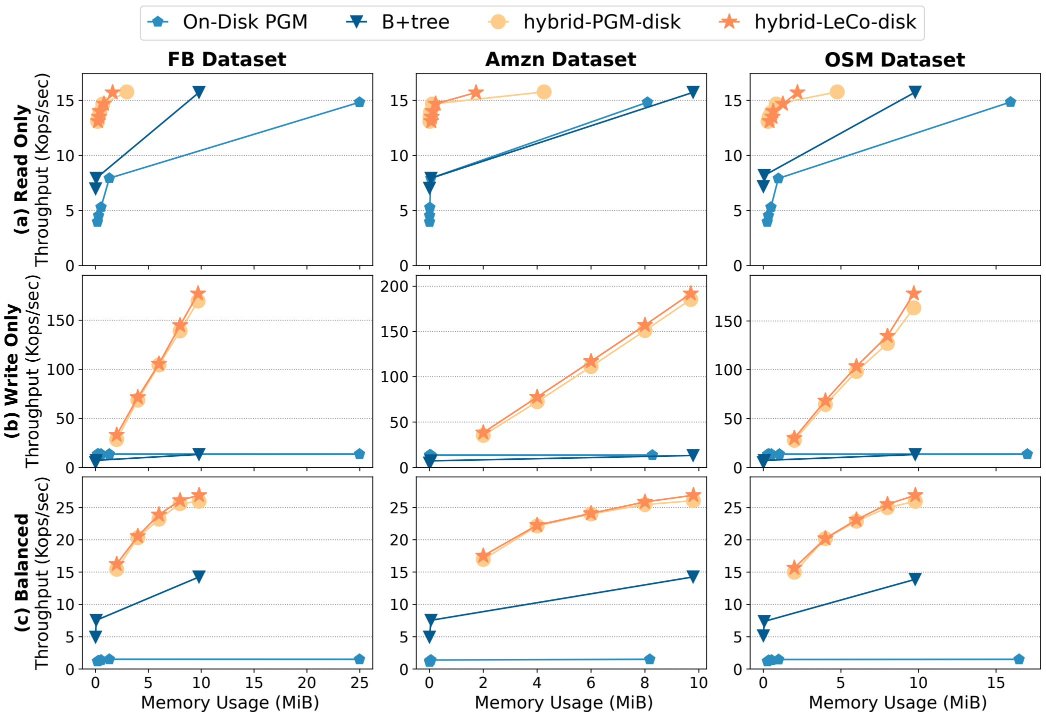


Fig. 11. Throughput and Memory Usage of Four Indexes on Three Single-Threaded Workloads – The expected number of I/O pages for learned indexes are as follows: **(a) Read-Only**: 1.05, 2, 3, 4, and 5 pages, respectively; **(b) Write-Only**: all five pages. **(c) Balanced**: 1.5 pages when the memory usage exceeds 2MiB, increasing to 5 pages when below 2MiB. Some points of PGM overlap because its insert throughput is independent of memory usage and the memory usage is close in this figure's scale when ϵ is large.

the baseline. Besides, the throughput of PGM on this workload is constant and independent of its own parameters, as it uses LSM-style multi-layer buffers to support inserts, but this can have a significant negative impact on mixed read/write workloads, as discussed below.

Mixed Workloads. On mixed workloads, our two indexes demonstrate remarkable performance, achieving up to 1.81× speedup than the B+tree and 16.76× speedup than the PGM while consuming the same memory footprint as the B+tree. The main reason for this performance boost is that we allocate some space for newly inserted keys and ensure that read performance remains efficient by allocating memory wisely between the two phases. In this way, read operations using the all-at-once page-fetching strategy only require a single fast I/O and disk-related write operations can be efficiently amortized. In contrast, due to its LSM-style design but without proper size ratios, PGM tends to perform poorly on mixed read/write workloads. Lookups in the PGM require traversing multiple layers of files throughout the index structure, resulting in a throughput of only around 1 Kops/sec. This validates our idea that existing implementations of disk-based learned indexes miss numerous optimization opportunities.

Moreover, indexes with smaller static stages can allocate more space for inserts when memory budgets are limited, consequently enhancing overall throughput. In scenarios with constrained memory, the hybrid-PGM-disk with a smaller memory footprint offers higher throughput than the hybrid-LeCo-disk.

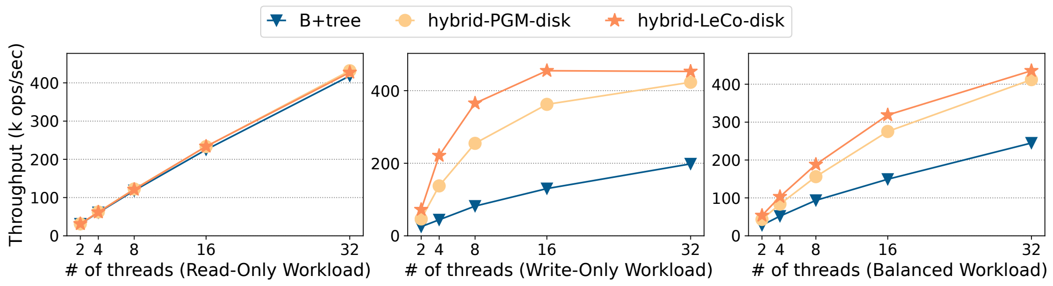


Fig. 12. Throughput of Three Multi-Threaded Indexes under Varying Thread Counts – The parameter settings for the indexes all use the settings with their highest throughput in the single-threaded experiments.

Table 7. Space Cost (MiB) of Four Indexes on Facebook Dataset, Including Memory Usage and Disk Space.

	PGM	B+tree	hybrid-PGM-d	hybrid-LeCo-d
Read-Only	2313.8	3301.58	2290.43	2291.77
Write-Only	2451.75	3301.58	2444.16	2443.86
Balanced	2387.54	3301.58	2299.34	2297.71

10.3 Multi-Threaded YCSB Evaluation

In this section, we repeat the above experiments on the Facebook dataset using multiple threads. Note that we disabled the parallel training of PGM in hybrid-PGM-disk to ensure a fair comparison. As shown in Figure 12, both hybrid learned indexes and B+tree exhibit excellent scalability in the read-only and balanced workload. For the write-only workload, the hybrid learned indexes can scale up to 16 threads before they saturate the disk bandwidth. The non-blocking, lock-free merge algorithm described in Section 9.1 allows the hybrid learned indexes to scale even with write-intensive workloads and to achieve a notable speedup over the B+tree.

10.4 Space Efficiency

Table 7 shows the total space cost for these indexes on the Facebook dataset. We only present the space cost for the configuration with the highest throughput for each workload, as the results align with those obtained from other setups and datasets. The hybrid-index design without gaps at the bottom level proves to be more space-efficient than the design that incorporates gaps throughout the index structure. This observation further highlights the effectiveness of our guidelines for utilizing a hybrid-index framework to facilitate updates, which not only help index structures achieve high throughput but also make them more compact.

10.5 Comparison to FILM

FILM [20] is a learned index designed for larger-than-memory scenarios. Its main idea is to incorporate an in-memory data cache based on adaptive LRU. We compare our approach to FILM using the above single-threaded YCSB-based workload on the Facebook dataset. We set the memory budget for our hybrid-PGM-disk to 10 MiB, according to the "best-throughput" configurations in Figure 11. Because FILM must keep a 16-byte (page #, offset) pair in memory for every data item on disk, it is impossible to force a small memory footprint with FILM. As shown in Figure 13, FILM has 1.9× higher throughput than hybrid-PGM-disk for the read-only workload, but it consumes 5 GiB of the memory space (the on-disk portion is only 0.94 GiB since 58% of data points are in memory).

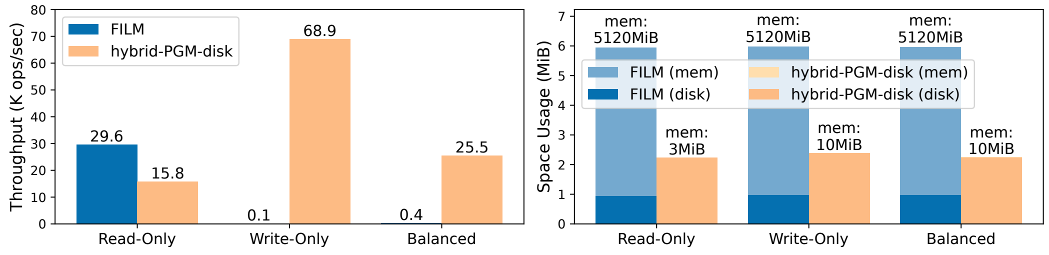


Fig. 13. Throughput and Space Cost Comparison to FILM on the Facebook Dataset

Table 8. TPC-C Workload – Index space includes both memory usage and disk space, and all indexes have similar memory usage.

100 warehouses	B+tree	hybrid-PGM-d	hybrid-LeCo-d
Throughput (txn/s)	443.65	545.86	533.21
Index Space (MiB)	1100.37	764.111	765.076

Meanwhile, FILM exhibits extremely low performance in the write-heavy workloads because it has to evict data frequently to disk to maintain the LRU chains. Given the poor write performance and high memory usage, we conclude that FILM is less efficient compared to our approach as a general-purpose on-disk index.

10.6 TPC-C Benchmark

We evaluate the hybrid learned indexes using the TPC-C benchmark [5] in this section. To focus on the performance differences of the on-disk indexes, we store the tuples in memory. Each table has a primary-key index, and we construct two secondary indexes: last name + first name for the `CUSTOMER` table and customer ID for the `ORDER` table. Each transaction involves multiple index lookups/insertions (on disk) followed by tuple fetches/insertions (in memory). The transaction throughput, therefore, is dominated by the index performance. We initialize the workload with 100 warehouses and then execute 200,000 transactions with a single thread. We report the transaction throughput and space consumption for using B+tree and the two different hybrid learned indexes. As shown in Table 8, a hybrid learned index can speed up transaction processing by over 20% in TPC-C compared to a B+tree while consuming 30% less index space.

11 CONCLUSION

In this paper, we delve into the issue of sub-optimal performance exhibited by learned indexes when transitioning from memory to disk scenarios. To address this issue, we propose several general guidelines to facilitate the mem-to-disk transformation. We assess our guidelines by developing and evaluating two disk-based learned indexes using the YCSB benchmark. The results demonstrate that indexes constructed in accordance with our guidelines can achieve superior throughput compared to the B+tree (implying supremacy over other implementations as well), while consuming a much smaller storage footprint. We hope these guidelines can facilitate practitioners and enhance the functionality and efficiency of learned indexes in a disk-based environment.

ACKNOWLEDGMENTS

The authors thank Shanghai Qi Zhi Institute for supporting this work.

REFERENCES

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 671–682. https://doi.org/10.1145/1142473.1142548

[2] Hussam Abu-Libdeh, Deniz Altınbüken, Alex Beutel, Ed H. Chi, Lyric Pankaj Doshi, Tim Klas Kraska, Xiaozhou (Steve) Li, Andy Ly, and Chris Olston (Eds.). 2020. *Learned Indexes for a Google-scale Disk-based Database.* https://arxiv.org/pdf/2012.12501.pdf

[3] Rudolf Bayer. 1972. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta informatica* 1, 4 (dec 1972), 290–306. https://doi.org/10.1007/BF00289509

[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[5] The Transaction Processing Council. 2007. TPC-C Benchmark. http://www.tpc.org/tpcc/

[6] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 969–984. https://doi.org/10.1145/3318464.3389711

[7] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads. *Proceedings of the VLDB Endowment* 14, 2 (oct 2020), 74–86. https://doi.org/10.14778/3425879.3425880

[8] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. *Proceedings of the VLDB Endowment* 13, 8 (apr 2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[9] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1189–1206. https://doi.org/10.1145/3299869.3319860

[10] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS.. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR '20)*. CIDR Conference, Amsterdam, The Netherlands, 8 pages.

[11] Andreas Kipf, Dominik Horn, Pascal Pfeil, Ryan Marcus, and Tim Kraska. 2022. LSI: A Learned Secondary Index Structure. In *Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Philadelphia, Pennsylvania) *(aiDM '22)*. Association for Computing Machinery, New York, NY, USA, Article 4, 5 pages. https://doi.org/10.1145/3533702.3534912

[12] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A Single-Pass Learned Index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Portland, Oregon) *(aiDM '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/3401071.3401659

[13] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

[14] Hai Lan, Zhifeng Bao, J. Shane Culpepper, and Renata Borovica-Gajic. 2023. Updatable Learned Indexes Meet Disk-Resident DBMS - From Evaluations to Design Choices. *Proceedings of the ACM on Management of Data* 1, 2, Article 139 (jun 2023), 22 pages. https://doi.org/10.1145/3589284

[15] Daniel Lemire and Leonid Boytsov. 2015. Decoding Billions of Integers per Second through Vectorization. *Software: Practice and Experience* 45, 1 (jan 2015), 1–29. https://doi.org/10.1002/spe.2203

[16] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A Fine-Grained Learned Index Scheme for Scalable and Concurrent Memory Systems. *Proceedings of the VLDB Endowment* 15, 2 (oct 2021), 321–334. https://doi.org/10.14778/3489496.3489512

[17] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2119–2133. https://doi.org/10.1145/3318464.3389703

[18] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. *Proc. ACM Manag. Data* 2, 1, Article 65 (mar 2024), 28 pages. https://doi.org/10.1145/3639320

[19] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A High-Performance Learned Index on Persistent Memory. *Proceedings of the VLDB Endowment* 15, 3 (nov 2021), 597–610. https://doi.org/10.14778/3494124.3494141

[20] Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maoliniyazi. 2022. FILM: A Fully Learned Index for Larger-Than-Memory Databases. *Proceedings of the VLDB Endowment* 16, 3 (nov 2022), 561–573. https://doi.org/10.14778/3570690.3570704

[21] Lin Ma, Joy Arulraj, Sam Zhao, Andrew Pavlo, Subramanya R. Dulloor, Michael J. Giardino, Jeff Parkhurst, Jason L. Gardner, Kshitij Doshi, and Stanley Zdonik. 2016. 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* (San Francisco, California) *(DaMoN '16)*. Association for Computing Machinery, New York, NY, USA, Article 9, 7 pages. https://doi.org/10.1145/2933349.2933358

[22] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proceedings of the VLDB Endowment* 14, 1 (sep 2020), 1–13. https://doi.org/10.14778/3421424.3421425

[23] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2789–2792. https://doi.org/10.1145/3318464.3384706

[24] Mayank Mishra and Rekha Singhal. 2021. RUSLI: Real-Time Updatable Spline Learned Index. In *Fourth Workshop in Exploiting AI Techniques for Data Management* (Virtual Event, China) *(aiDM '21)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3464509.3464886

[25] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 985–1000. https://doi.org/10.1145/3318464.3380579

[26] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *Proceedings of the VLDB Endowment* 13, 12 (jul 2020), 2341–2354. https://doi.org/10.14778/3407790.3407829

[27] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proceedings of the VLDB Endowment* 16, 8 (jun 2023), 1992–2004. https://doi.org/10.14778/3594512.3594528

[28] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: A Scalable Learned Index for Multicore Data Storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 308–320. https://doi.org/10.1145/3332466.3374547

[29] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. 2020. SIndex: A Scalable Learned Index for String Keys. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems* (Tsukuba, Japan) *(APSys '20)*. Association for Computing Machinery, New York, NY, USA, 17–24. https://doi.org/10.1145/3409963.3410496

[30] Zhaoguo Wang, Haibo Chen, Youyun Wang, Chuzhe Tang, and Huan Wang. 2022. The Concurrent Learned Indexes for Multicore Data Storage. *ACM Transactions on Storage* 18, 1, Article 8 (jan 2022), 35 pages. https://doi.org/10.1145/3478289

[31] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are Updatable Learned Indexes Ready? *Proceedings of the VLDB Endowment* 15, 11 (jul 2022), 3004–3017. https://doi.org/10.14778/3551793.3551848

[32] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proceedings of the VLDB Endowment* 14, 8 (apr 2021), 1276–1288. https://doi.org/10.14778/3457390.3457393

[33] Shangyu Wu, Yufei Cui, Jinghuan Yu, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2022. NFL: Robust Learned Index via Distribution Transformation. *Proceedings of the VLDB Endowment* 15, 10 (jun 2022), 2188–2200. https://doi.org/10.14778/3547305.3547322

[34] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1567–1581. https://doi.org/10.1145/2882903.2915222

[35] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: A Cache-Aware Learned Index with a Cost-Based Construction Algorithm. *Proceedings of the VLDB Endowment* 15, 11 (jul 2022), 2679–2691. https://doi.org/10.14778/3551793.3551823

[36] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, Guoan Wu, and Andy Rudoff. 2022. PLIN: A Persistent Learned Index for Non-Volatile Memory with High Performance and Instant Recovery. *Proceedings of the VLDB Endowment* 16, 2 (oct 2022), 243–255. https://doi.org/10.14778/3565816.3565826

[37] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine Using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 685–699. https://doi.org/10.1145/3514221.3526187

Received October 2023; revised January 2024; accepted February 2024