Why Are Learned Indexes So Effective but Sometimes Ineffective?

Qiyu Liu*† Southwest University qyliu.cs@gmail.com
Siyuan Han* HKUST shanaj@connect.ust.hk
Yanlin Qi HIT Shenzhen yanlinqi7@gmail.com
Jingshu Peng ByteDance jingshu.peng@bytedance.com
Jin Li Harvard University jinli@g.harvard.edu
Longlong Lin Southwest University longlonglin@swu.edu.cn
Lei Chen HKUST & HKUST (GZ) leichen@cse.ust.hk

ABSTRACT

Learned indexes have attracted significant research interest due to their potential to offer better space-time trade-offs compared to B+-tree variants. Among various learned indexes, the PGM-Index based on error-bounded piecewise linear approximation is an elegant data structure that has demonstrated *provably* superior performance over conventional B+-tree indexes. However, despite numerous efforts to optimize the design of the PGM-Index, few systematically study the root causes of performance mismatches observed in practice. In this paper, we explore two key research questions. **Q1**: *Why are PGM-Indexes theoretically effective?* and **Q2**: *Why do PGM-Indexes underperform in practice?* For **Q1**, we show that for a set of N sorted keys, the PGM-Index can achieve a lookup time of $O(\log \log N)$ while using $O(N)$ space. For **Q2**, we identify that querying PGM-Indexes is highly memory-bound, where the internal index search operations often become the bottleneck. To fill the performance gap, we propose PGM++, a *simple yet effective* extension to the original PGM-Index that employs a mixture of different search strategies, with hyper-parameters automatically tuned through a cost model calibrated by theoretical findings. Extensive experiments show that, at comparable space costs, PGM++ speeds up index lookup queries by up to **2.31**× and **1.56**× when compared to the original PGM-Index and SOTA baselines.

PVLDB Reference Format:
Qiyu Liu, Siyuan Han, Yanlin Qi, Jingshu Peng, Jin Li, Longlong Lin, and Lei Chen. Why Are Learned Indexes So Effective but Sometimes Ineffective?. PVLDB, 18(9): 2886 - 2898, 2025.
doi:10.14778/3746405.3746415

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/qyliu-hkust/bench_search.

1 INTRODUCTION

Indexes are fundamental components of DBMS and big data engines to enable efficient query processing [31, 37]. An emerging research tendency is to directly learn the storage layout of sorted data by using simple machine learning (ML) models, leading to the concept of *Learned Index* [8, 10, 16, 46, 51, 52]. Compared to traditional indexes like B+-tree variants [5, 14, 17, 18, 46], learned indexes have been shown to reduce the index memory footprint by 2–3 orders of magnitude while achieving comparable lookup performance.

Similar to B+-trees or other binary search tree (BST) variants, learned indexes address the classical problem of *Sorted Dictionary Indexing* [6]. Given a sorted set of N keys $\mathcal{K} = \{k_1, \cdots, k_N\}$, the goal of learned indexes is to find a compact projection function (i.e., an ML model) $f : k \mapsto \mathbb{N}^+$ that maps an arbitrary query key k to its corresponding index in the sorted array $\mathcal{K}$. However, as ML models inherently produce prediction errors, learned indexes usually employ error-bounded last-mile search (within the maximum prediction error ϵ) to ensure the correctness of query result. To eliminate the prediction error, an exact "last-mile" search, typically a standard binary search, is performed within the range $[f(k) - \epsilon, f(k) + \epsilon]$ to eliminate the model prediction errors. To balance model accuracy with complexity, existing learned indexes, such as RMI [16], PGM-Index [10], ALEX [8], LIPP [46], NFL [47], and DILI [18], opt to stack simple models, such as linear models or polynomial splines, in a hierarchical structure.

Among the various published learned indexes [8, 10, 16, 18, 46, 51, 52], the PGM-Index [10] stands out as a simple yet elegant data structure that has been proven to be *theoretically* more efficient than B+-trees [9]. As illustrated in Figure 1, the PGM-Index is constructed in a *bottom-up* fashion by recursively fitting the input keys using error-bounded piecewise linear approximation models (ϵ-PLA) until reaching a single line segment. For query processing, the PGM-Index performs a *top-down* traversal from the root to the leaf levels. At each level, an *error-bounded search* is invoked to identify either the line segment used to predict the index for the subsequent level or the exact location of the search key in the original sorted array. Recent theoretical analyses [9] have demonstrated that, compared to a B+-tree with fanout B, the PGM-Index can reduce memory footprint by a factor of B while preserving the same logarithmic query complexity.

Intuitively, the PGM-Index is structured as a hierarchy of line segments, where the index *height* is a key factor in determining the query time complexity. Existing results [9, 10] claim that the height of a PGM-Index grows logarithmically with respect to the data size (i.e., $O(\log N)$). However, our empirical investigations (Section 3.1) reveal that PGM-Indexes are highly *flat*, with over **99%** of the total index space cost attributed to the segments at the *bottom* level. This observation implies that the height of the PGM-Index grows much

*Both authors contributed equally to the paper.
†Correspondence to Dr. Qiyu Liu.

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746415

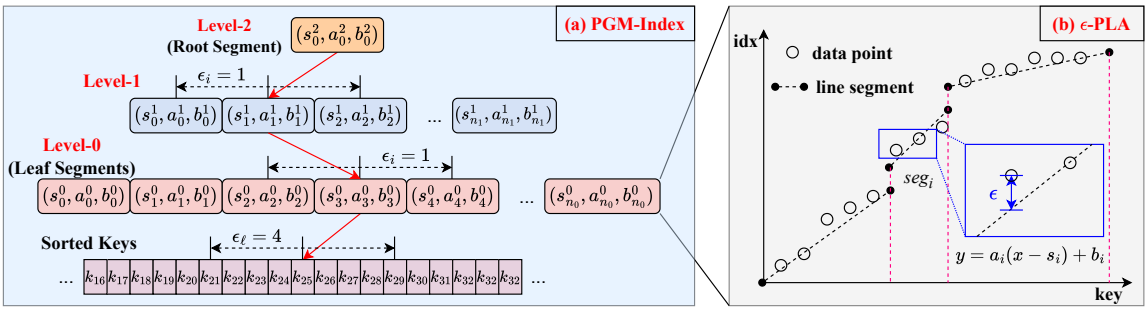


Figure 1: A toy example of a 3-level PGM-Index with $\epsilon_i = 1$ (i.e., internal search error range) and $\epsilon_\ell = 4$ (i.e., last-mile search error range). Processing a lookup query on such PGM-Index involves in total three linear function evaluations, two internal search operations in the range $2 \cdot \epsilon_i + 1$, and one "last-mile" search operation on the sorted data array in the range $2 \cdot \epsilon_\ell + 1$.

more slowly than $O(\log N)$, potentially at a *sub-logarithmic* rate. Motivated by this, we pose the first research question.

Q1: Why Are PGM-Indexes So Effective in Theory? To answer this question, we establish new theoretical results for PGM-Indexes based on a key finding: fitting PLA models becomes progressively easier at higher levels of the hierarchy. Specifically, we show that, with high probability (w.h.p.), the query time of the PGM-Index can be bounded by $O(\log_2 \log_G N) = O(\log \log N)$ using linear space of $O(N/G)$, where G is a constant determined by data distribution characteristics and the error constraint ϵ. To the best of our knowledge, this work presents the *tightest* bound for learned index structures compared to existing theoretical analyses [9, 50].

Interestingly, a BST can be viewed as a "materialized" version of the binary search algorithm, whose time complexity is $O(\log N)$. Similarly, the PGM-Index with piecewise linear approximation models can be regarded as a "materialized" version of the interpolation search algorithm, whose time complexity is $O(\log \log N)$ [29, 34], aligning with our theoretical findings.

Despite its theoretical superiority, recent benchmarks [24, 45] show that the PGM-Index falls short of practical performance expectations, often underperforming compared to well-optimized RMI variants [15, 16]. This mismatch between theoretical and practical performance motivates our second research question.

Q2: Why Are PGM-Indexes Ineffective in Practice? Our investigation, supported by extensive benchmark results across various hardware platforms (Section 4), reveals that PGM-Indexes are highly memory-bound. The internal error-bounded search operation, typically implemented as a standard binary search (e.g., `std::lower_bound` in C++), becomes a bottleneck when processing an index lookup query. According to our benchmark results, fewer than **1%** of the internal segments contribute to over **80%** of the total index lookup time.

To improve search efficiency, we adopt a hybrid internal search strategy that combines the advantages of linear search and optimized branchless binary search [36]. By carefully setting search range thresholds, this approach accelerates the standard binary search implementation by up to **1.6×**, resulting in an overall improvement of up to **2.3×** in total index lookup time.

Moreover, as illustrated in Figure 1, constructing a PGM-Index requires two hyper-parameters: ϵ_i, the error thresholds for internal index traversal, and ϵ_ℓ, the error thresholds for last-mile search on the sorted array. Our empirical findings (Section 5.2) reveal that ϵ_ℓ primarily controls the overall index size, while both ϵ_i and ϵ_ℓ affect the efficiency of index lookups. Building on these findings, we develop a query cost model grounded in theoretical analysis and calibrated through extensive benchmarking. Leveraging this cost model, we further introduce an automatic hyper-parameter tuning strategy to suggest configurations by minimizing the cost model while adhering to a specified index size budget.

In summary, our technical contributions are as follows. ❶ **New Bound.** We prove the sub-logarithmic index lookup time of the PGM-Index (i.e., $O(\log \log N)$). This result not only tightens the previously reported logarithmic bound but also reinforces the PGM-Index's theoretical superiority over conventional tree-based indexes. ❷ **Simple Methods.** We introduce PGM++, a *simple yet effective* improvement to the PGM-Index by replacing the costly internal search operations. We further propose an automatic hyper-parameter tuner for PGM++, guided by an accurate cost model. ❸ **Empirical Superiority.** Extensive experimental studies on real-world and synthetic datasets demonstrate the effectiveness of PGM++. For static workloads, PGM++ robustly outperforms the original PGM-Index and optimized RMI variants [24, 51] by up to **2.31×** and **1.56×**, respectively. For dynamic workloads, while less performant in read-heavy scenarios, PGM++ demonstrates robust and superior performance on highly write-intensive workloads.

The remainder of this paper is organized as follows. Section 2 overviews the basis of learned indexes. Section 2.3 details the microbenchmark used throughout this paper. Section 3 presents our core theoretical results showing the sub-logarithmic bound for PGM-Index. Section 4 investigates the practical limitations of PGM-Index. In Section 5, we introduce PGM++, an optimized variant of PGM-Index, featuring hybrid error-bounded search and automatic hyper-parameter tuning. Section 6 reports the experimental results. Section 7 surveys and discusses related works, and finally, Section 8 concludes the paper and discusses future research directions.

2 PRELIMINARIES

2.1 Basis of the PGM-Index

Given a set of N *sorted* keys $\mathcal{K} = \{k_1, k_2, \cdots, k_N\}$, the goal of learned indexes is to find a mapping $f : k \mapsto \mathbb{N}^+$ such that f can project a search key $k \in \mathcal{K}$ to its corresponding index $\text{rank}(k)$ with controllable error. Intuitively, learning f is equivalent to fitting $\mathcal{K}$'s cumulative distribution function (CDF) scaled by the data size N. The model selection considerations for f are threefold: ❶ **Model Complexity**: The model f should be compact to reduce

memory footprint, and its inference should introduce minimal computation overhead. ❷ **Error-Boundness**: The model f should be error-bounded, ensuring that an exact last-mile search can correct prediction errors, i.e., $|f(k) - \text{rank}(k)| \leq \epsilon$ holds for $\forall k \in \mathcal{K}$; ❸ **Monotonicity**: To ensure the correctness of queries for keys outside $\mathcal{K}$, the model f must be strictly monotonic, i.e., $f(k_1) \leq f(k_2)$ holds for any $k_1 \leq k_2$.

Though achieving success in various domains, deep learning (DL) models usually require a heavy runtime like PyTorch [32] or TensorFlow [41] that are costly and less flexible. Instead, existing learned index designs favor *stacking* simple models, such as linear functions [8, 10, 46], polynomial splines [16], and radix splines [15]. Among these, the PGM-Index [10] employs the error-bounded piecewise linear approximation (i.e., ϵ-PLA) to strike a balance between the model complexity and prediction accuracy.

Definition 2.1 (ϵ-PLA). Given a univariate set $\mathcal{X} = \{x_1, \cdots, x_N\}$, a corresponding target set $\mathcal{Y} = \{y_1, \cdots, y_N\}$, and an error constraint ϵ, an ϵ-PLA of m line segments on the point set in Cartesian space $(\mathcal{X}, \mathcal{Y}) = \{(x_i, y_i)\}_{i=1,\cdots,N}$ is defined as,

$$f(x) = \begin{cases} a_1 \cdot (x - s_1) + b_1 & \text{if } s_1 \leq x < s_2 \\ a_2 \cdot (x - s_2) + b_2 & \text{if } s_2 \leq x < s_3 \\ \cdots & \cdots \\ a_m \cdot (x - s_m) + b_m & \text{if } s_m \leq x < +\infty \end{cases} \tag{1}$$

such that for $\forall i = 1, 2, \cdots, N$, it always holds that $|f(x_i) - y_i| \leq \epsilon$.

Each segment in an ϵ-PLA can be expressed by a tuple $seg_i = (s_i, a_i, b_i)$ where s_i is the segment's starting point, a_i is the slope, and b_i is the intercept. To ensure the monotonic requirement, the segments in Eq. (1) should satisfy two conditions: (a) $a_i \geq 0$ for $i = 1, \cdots, m$, and (b) $s_i < s_j$ for $\forall 1 \leq i < j \leq m$. We then extend the original PGM-Index definition [10] by introducing separate error parameters for internal search and last-mile search.

Definition 2.2 ($(\epsilon_i, \epsilon_\ell)$-PGM-Index [10]). Given a sorted key set $\mathcal{K} = \{k_1, k_2, \cdots, k_N\}$ and error parameters ϵ_i and ϵ_ℓ ($\epsilon_i, \epsilon_\ell \in \mathbb{N}^+$), an $(\epsilon_i, \epsilon_\ell)$-PGM-Index is a multi-level structure:

❶ **Leaf Level:** The 0-th level is an ϵ_ℓ-PLA constructed on the dataset $(\mathcal{K}, \mathcal{I} = \{1, \cdots, N\})$, where $\mathcal{I}$ denotes the index set.

❷ **Internal Levels:** For the j-th level ($j \geq 1$), let $\mathcal{S}_{j-1}$ denote the set of segments in the previous level (i.e., level $j-1$), and let $\mathcal{K}_{j-1} = \{seg.s \mid seg \in \mathcal{S}_{j-1}\}$ and $\mathcal{I}_{j-1} = \{1, 2, \cdots, |\mathcal{K}_{j-1}|\}$. Then, the j-th level is an ϵ_i-PLA constructed on the dataset $(\mathcal{K}_{j-1}, \mathcal{I}_{j-1})$. Specifically, the topmost level consists of a single line segment, forming the root of the index structure.

We categorize the segments in the bottom level as *leaf segments* and the remaining segments as *internal segments*. For an $(\epsilon_i, \epsilon_\ell)$-PGM-Index, searching for a query key k can be performed in a *top-down* manner as follows:

❶ **Internal Index Traversal:** The search begins at the root level and proceeds downward, finding the appropriate segment at each level until reaching the leaf level (depicted by the red path in Figure 1). Specifically, let $seg^j = (s^j, a^j, b^j)$ denote the segment at the j-th level. The next segment to traverse at the $(j-1)$-th level is determined by searching for k within $a^j \cdot (k - s^j) + b^j \pm \epsilon_i$.

❷ **Last-Mile Search:** After reaching the 0-th level, an exact search is conducted on the raw sorted keys $\mathcal{K}$ within $\widehat{\text{rank}(k)} \pm \epsilon_\ell$, where

Results	Base Model	Lookup Time	Space Cost
ICML'20 [9]	Linear	$O(\log N)$	$O(N/\epsilon^2)$
ICML'23 [50]	Constant	$O(\log\log N)$	$O(N \log N)$
Ours	Linear	$O(\log\log N)$	$O(N/G)$

Table 1: Summary of theoretical results for *static* learned index. "Ours" refers to our tighter bounds for PGM-Index.

$\widehat{\text{rank}(k)} = a^0 \cdot (k - s^0) + b^0$ is the predicted rank for k, and $seg^0 = (s^0, a^0, b^0)$ is the *leaf segment* found during the internal index traversal.

Recall that the index construction introduced in Definition 2.2 ensures that the maximum errors for internal index traversal and last-mile search are bounded by ϵ_i and ϵ_ℓ, respectively. Therefore, the above search process guarantees to find the correct location (i.e., $\text{rank}(k)$) for an arbitrary query key k.

2.2 Existing Theoretical Results

From Section 2.1, two key sub-problems need to be addressed to determine the space and time complexities of the PGM-Index: ❶ *How many line segments are required to satisfy the error constraint for an ϵ-PLA model?* and ❷ *What is the height (i.e., the number of layers) of a PGM-Index?* In this section, we review existing theoretical studies [9, 50] regarding these two questions, with the major results summarized in Table 1.

The original PGM-Index [10] first provides a straightforward lower bound to determine the index height.

Theorem 2.3 (PGM-Index Lower Bound [10]). *Given a consecutive chunk of $2\epsilon + 1$ sorted keys $\{k_i, \cdots, k_{i+2\epsilon}\} \subseteq \mathcal{K}$, there always exists a horizontal line segment $\ell(x) = i + \epsilon$ such that $|\ell(k_j) - j| \leq \epsilon$ for $j \in \{i, \cdots, i + 2\epsilon\}$, implying that each line segment in an ϵ-PLA can cover at least $2\epsilon + 1$ keys.*

According to Theorem 2.3, w.l.o.g., for an (ϵ, ϵ)-PGM-Index, the height of the index is $O(\log_\epsilon N) = O(\log N)$. Thus, the index lookup takes time $O(\log N \cdot \log_2 \epsilon) = O(\log N)$, as ϵ can be treated as a pre-specified constant. Ferragina et al. [9] further refine the results in Theorem 2.3 by using a statistical model. Suppose that the key set to be indexed $\mathcal{K} = \{k_1, k_2, \cdots, k_N\}$ is a realization of a random process $k_i = k_{i-1} + g_i$ for $i \geq 2$ where g_i's are i.i.d. random variables (r.v.) following some unknown distribution. We refer to the r.v. g_i as the "gap", with $\mu = \mathbf{E}[g_i]$ and $\sigma^2 = \mathbf{Var}[g_i]$ representing its mean and variance, respectively. With such statistical model, the segment coverage can be further improved as follows.

Theorem 2.4 (Expected Line Segment Coverage [9]). *Given a set of sorted keys $\mathcal{K} = \{k_1, k_2, \cdots, k_N\}$ and an error parameter ϵ, let the gap be $g_i = k_i - k_{i-1}$. If the condition $\epsilon \gg \sigma/\mu$ holds, with high probability, the expected number of keys in $\mathcal{K}$ covered by a line segment $\ell(x) = \frac{1}{\mu} \cdot (x - k_1) + 1$ is given by:*

$$\mathbf{E}\left[\min\left\{i \in \mathbb{N}^+ \mid |\ell(k_i) - i| > \epsilon\right\}\right] = \mu^2\epsilon^2/\sigma^2, \tag{2}$$

where $\ell(k_i) = \mu \cdot (k_i - k_1) + 1$ is the predicted index for k_i.

By constructing a special line segment with slope $1/\mu$, Theorem 2.4 derives a tighter bound on segment coverage compared to Theorem 2.3. For a set of N sorted keys, the expected number of segments[1] in a *one-layer* ϵ-PLA can be derived as $N\sigma^2/\epsilon^2\mu^2$. In the

[1]This conclusion is not rigorous, as, in general, $1/\mathbf{E}[X] \neq \mathbf{E}[1/X]$ for an arbitrary random variable X. A better proof of this result can be found in Theorem 4 of [9].

Platform	OS	Compiler	CPU	Frequency	Memory (Bandwidth)	L1	L2	L3 (LLC)
X86-1	Ubuntu 20.04	g++ 11	Intel Core i7-13700K	5.30 GHz (P-core)	32 GB DDR4 (12.8 GB/s)	64 KB	256 KB	16 MB
X86-2	CentOS 9.4	g++ 11	AMD EPYC 7413	3.60 GHz	1 TB DDR4 (10.6 GB/s)	64 KB	1 MB	256 MB
ARM	macOS 14.4.1	clang++ 15	Apple M3	4.05 GHz (P-core)	16 GB LPDDR5 (35.6 GB/s)	320 KB	16 MB	N.A.

Table 2: Summary of three micro-benchmark platforms. For platforms X86-1 and ARM, which adopt the "big.LITTLE" architecture [2], the hardware statistics of the performance cores are reported. The listed L1/L2/L3 sizes represent the actual cache size accessible to a single physical core. Notably, for the Apple M3 chip, only L1 and L2 caches are available.

implementation, an *optimal online* ϵ-PLA fitting algorithm [27] is adopted to *minimize* the number of line segments while satisfying the error constraint ϵ, which ensures to find fewer segments than the special segment construction in Theorem 2.3. Therefore, the expected number of segments is bounded by $O(N/\epsilon^2)$. Combining Theorem 2.3 with Theorem 2.4, Ferragina et al. [9] claim that a PGM-Index using $O(N/\epsilon^2)$ space can process lookup queries in $O(\log N)$ time with high probability. By setting $\epsilon = \Theta(B)$, a PGM-Index achieves the same logarithmic index lookup complexity of a B+-tree with fanout B, while reducing the space complexity from B+-tree's $O(N/B)$ to $O(N/B^2)$.

Notably, the above results extend to any other *optimal* error-bounded PLA fitting algorithms. Besides the online algorithm [27] used by PGM-Index, other segment fitting methods include the OptimalPLR and GreedyPLR [49] used in Bourbon [7] and ShrinkingCone employed in FITing-Tree [11]. In [49], the OptimalPLR algorithm has been proven to be equivalent to [27], implying that the bound in Theorem 2.4 also applies to OptimalPLR. However, GreedyPLR and ShrinkingCone lack such optimality guarantees. In practice, however, these sub-optimal methods achieve comparable segment counts using only $O(1)$ space and are easy to parallelize. As linear function is a popular choice in learned index design, we leave benchmarking the impact of different segment fitting algorithms as an interesting topic for future study.

In addition to [9], a recent study [50] also explores the theoretical aspects of learned indexes. They show that a Recursive Model Index [16], using piece-wise constant functions as base models, can achieve a *sub-logarithmic* lookup complexity of $O(\log\log N)$. However, this comes at the cost of *super-linear* space of $O(N \log N)$.

Our Results. Inspired by the results of [50], we reasonably propose that PGM-Indexes, adopting ϵ-PLA as base models, can achieve the same *sub-logarithmic* lookup time complexity with *reduced* space overhead, given that a piecewise constant function can be regarded as a special case of a piecewise linear function. As summarized in Table 1, our analysis in Section 3 shows that the PGM-Index can search a query key in $O(\log\log N)$ time while requiring linear space $O(N/G)$, where G is a constant that depends on the error parameter ϵ and the gap distribution characteristics.

2.3 Microbenchmark Setting

To ensure consistency in presentation, we first detail the microbenchmark setups, including the hardware platforms and datasets for evaluation. Throughout the remainder of this paper, we adopt this microbenchmark to either motivate or validate the theoretical findings and proposed methodologies.

Platforms. We conduct the experiments on three platforms with different architectures: ❶ **X86-1:** an Ubuntu desktop equipped with an Intel© Core™ i7-13700K CPU (5.30 GHz) and 64 GB of memory; ❷ **X86-2:** a CentOS server with 2 AMD© EPYC™ 7413 CPUs (3.60 GHz) and 1 TB of memory; and ❸ **ARM:** a Macbook Air laptop with an Apple Silicon M3 CPU (4.05 GHz) and 16 GB of unified memory, which provides higher memory bandwidth compared to the X86 platforms. All the experiments are written in C++ and compiled using g++ 11.4 on X86-1 and X86-2 and clang++ 15 on ARM. Table 2 summarizes the specifications of the benchmark platforms.

Dataset	#Keys	Raw Size	Seg	$\overline{Cov}$	h_D
fb	200 M	1.6 GB	2.13×10^6	94	2.86×10^7
wiki	200 M	1.6 GB	2.28×10^5	877	1.29×10^3
books	800 M	6.4 GB	1.98×10^6	101	2.86
osm	800 M	6.4 GB	1.55×10^6	129	1.27×10^6

Table 3: Summary of datasets. Seg is the number of segments to fit an ϵ-PLA with $\epsilon = 16$. $\overline{Cov} = N/Segs$ is the average segment coverage. $h_D = \sigma^2/\mu^2$ according to Theorem 2.4.

Benchmark Datasets. We adopt 4 real datasets from SOSD [24] that have been widely used in previous studies [8, 15, 45, 46, 51, 52]: ❶ **fb:** a set of user IDs randomly sampled from Facebook [34]; ❷ **wiki:** a set of edit timestamp IDs from Wikipedia [43]; ❸ **books:** a dataset of book popularity from Amazon; ❹ **osm:** a set of cell IDs from OpenStreetMap [26]. Additionally, we generate three synthetic datasets by sampling from uniform, normal, and log-normal distributions, following a process similar to [24, 52]. To quantify the difficulty of indexing a given dataset, we adopt the approach from [45] and compute the minimum number of segments (obtained by building a one-layer PGM-Index) required to satisfy the error constraint of an ϵ-PLA (with $\epsilon = 16$ by default). All keys are represented as 64-bit unsigned integers (`uint64_t` in C++), and Table 3 summarizes the dataset statistics.

3 WHY ARE PGM-INDEXES SO EFFECTIVE?

3.1 Motivation Experiments

We construct both PGM-Indexes and B+-trees using various configurations, with the index statistics summarized in Table 4. Intuitively, a B+-tree with a fan-out of $B = \epsilon$ can be considered analogous to a PGM-Index with $\epsilon_i = \epsilon_\ell = \epsilon$, since a B+-tree index guarantees that a search key will be located within a data block of size B.

As shown in Table 4, we first fix $B = \epsilon = 16$ while varying the input data size from 10^3 to 10^9 using synthetic uniform keys. As the data size N increases, the height of the B+-tree index (H_B) follows a logarithmic growth pattern, adhering to the formula $H_B = \lceil 1 + \log_B \frac{N+1}{2} \rceil$. However, the PGM-Index height (H_{PGM}) grows at a much slower, sub-logarithmic rate. Moreover, when varying ϵ within $\{2^2, 2^3, \cdots, 2^9\}$ on dataset books, the results consistently demonstrate that $H_{PGM} \ll H_B$ across all ϵ configurations, and the decrease in H_{PGM} is also notably slower than that of H_B. In addition to index height, Table 4 also reports the numbers of leaf and internal segments. Unlike B+-trees or other BST variants, the

N	PGM Height	Leaf Segments	Internal Segments	% over Total	B+-tree Height
10^3	2	2	2	50.0%	4
10^4	2	16	2	88.9%	6
10^5	2	140	2	98.6%	7
10^6	2	1,388	2	99.9%	8
10^7	3	13,918	12	99.9%	9
10^8	3	139,376	109	99.9%	10
10^9	4	1,394,003	1,049	99.9%	11
ϵ (B)	PGM Height	Leaf Segments	Internal Segments	% over Total	B+-tree Height
8	4	16,859,902	46,572	99.7%	11
16	4	7,943,403	4,100	99.9%	9
32	3	2,464,229	272	99.99%	7
64	3	797,152	60	99.99%	6
128	3	267,966	25	99.99%	6
256	3	81,340	12	99.99%	5
512	3	22,684	7	99.99%	5

Table 4: Index statistics of PGM-Indexes ($\epsilon_i = \epsilon_\ell = \epsilon$) and B+-trees (fan-out $B = \epsilon$). ϵ is fixed to 8 while varying data size N (using synthetic uniform keys), and N is fixed to 800M while varying ϵ (using dataset books). The ratio in percentage refers to the proportion of *leaf* segments contributing to the total index memory footprint.

PGM-Index exhibits a highly *flat* structure, with up to **99.99%** of the line segments located at the bottom level, aligning with the guess on the sub-logarithmic growth in index height.

3.2 Theoretical Analysis

In this section, we aim to provide a tighter bound that refines the previous results with the following roadmap:
❶ Lemma 3.1 and Lemma 3.2 establish a lower bound for the expected segment coverage at each level of the PGM-Index;
❷ Theorem 3.3 derives the PGM-Index height as $O(\log\log N)$, demonstrating sub-logarithmic growth w.r.t. the data size N;
❸ Theorem 3.4 concludes the space and time complexities of the PGM-Index, summarized in Table 1.

Notably, unless otherwise stated, the subsequent analyses adhere to the core assumptions on *gaps* from Theorem 2.4 [9]. Specifically, the gaps are i.i.d. random variables following an unknown distribution with expectation μ and variance σ^2. As discussed in [9], the "i.i.d." assumption can be further relaxed to *weak correlation* without affecting the correctness of theoretical results.

Lemma 3.1 (Expected Coverage Recursion). *Given a set of N sorted keys $\mathcal{K} = \{k_1, \cdots, k_N\}$ and an error parameter ϵ, let a random variable C_i denote the number of keys in the $(i-1)$-th level that a segment in the i-th level can cover (i.e., satisfying the error constraint ϵ). Specifically, C_0 denotes the leaf segment coverage (i.e., level-0) for the input key set $\mathcal{K}$. Then, the following recursion holds for $\mathbf{E}[C_i]$:*

$$\mathbf{E}[C_i] = \frac{\mu^2 \cdot \epsilon^2}{\sigma^2} \cdot \mathbf{E}[C_0 \cdot C_1 \cdots C_{i-1}]. \tag{3}$$

Proof. According to the law of total expectation [4], we have,

$$\mathbf{E}[C_i] = \int \cdots \int \mathbf{E}[C_i \mid C_0 = c_0, \cdots, C_{i-1} = c_{i-1}] \times f(c_0, \cdots, c_{i-1})\, dc_0 \cdots dc_{i-1}, \tag{4}$$

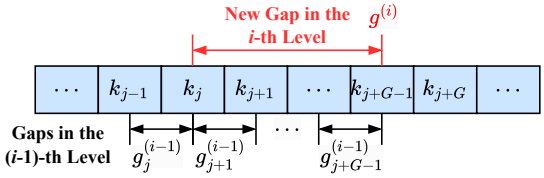


Figure 2: Illustration of gaps for the next level. Suppose G is the segment coverage at the i-th level. The new gap at the i-th level is given by $g^{(i)} = \sum_{j'=j+1}^{j+G-1} g_{j'}^{(i-1)}$, where $g_{j'}^{(i-1)}$ is the j'-th gap at the $(i-1)$-th level.

where $f(c_0, \cdots, c_{i-1})$ is the joint probability density function of the random variables $C_0, \cdots, C_{i-1}$.

When fixing $C_0, \cdots, C_{i-1}$ to values $c_0, \cdots, c_{i-1}$, as illustrated in Figure 2, w.l.o.g., an arbitrary gap at the i-th level, denoted by $g^{(i)}$, is the sum of $c_0 \cdot c_1 \cdots c_{i-1}$ consecutive gaps from the raw key set $\mathcal{K}$[2]. Thus, according to Theorem 2.4, on a key set with gaps as $g^{(i)}$, the expected segment coverage (conditioned on $C_0, \cdots, C_{i-1}$) at the i-th level for an ϵ-PLA is given by:

$$\begin{aligned}\mathbf{E}[C_i | C_0 = c_0, \cdots, C_{i-1} = c_{i-1}] &= \frac{\mathbf{E}\left[g^{(i)}\right]^2 \cdot \epsilon^2}{\mathbf{Var}\left[g^{(i)}\right]} \\ &= \frac{\mathbf{E}\left[\sum_{j'=j}^{j+c_0 \cdot c_1 \cdots c_{i-1}} g_{j'}\right]^2 \cdot \epsilon^2}{\mathbf{Var}\left[\sum_{j'=j}^{j+c_0 \cdot c_1 \cdots c_{i-1}} g_{j'}\right]} \\ &= (c_0 \cdot c_1 \cdots c_{i-1}) \cdot \frac{\mu^2 \cdot \epsilon^2}{\sigma^2},\end{aligned} \tag{5}$$

where μ and σ^2 are the mean and variance of the gaps on the *original key set* $\mathcal{K}$. Taking Eq. (5) into the integral in Eq. (4), we have,

$$\begin{aligned}\mathbf{E}[C_i] &= \frac{\mu^2 \cdot \epsilon^2}{\sigma^2} \int \cdots \int \prod_{j=0}^{i-1} c_j \cdot f(c_0, \cdots, c_{i-1})\, dc_0 \cdots dc_{i-1} \\ &= \frac{\mu^2 \cdot \epsilon^2}{\sigma^2} \cdot \mathbf{E}[C_0 \cdot C_1 \cdots C_{i-1}].\end{aligned} \tag{6}$$

Thus, we arrive at the statement in Lemma 3.1. □

Lemma 3.2 (Expected Coverage of Level-i). *The following lower bound holds for $\mathbf{E}[C_i]$:*

$$\mathbf{E}[C_i] \geq \left(\frac{\mu^2 \cdot \epsilon^2}{\sigma^2}\right)^{2^i}. \tag{7}$$

Proof. We prove Lemma 3.2 using mathematical induction.
❶ **Base Case ($i' = 0$):** According to Theorem 2.4, $\mathbf{E}[C_0] = \mu^2\epsilon^2/\sigma^2$, satisfying the inequality in Eq. (7).
❷ **Inductive Step:** Assume that the lower bound in Eq. (7) holds for $i' = i - 1$, i.e.,

$$\mathbf{E}[C_{i-1}] \geq \left(\frac{\mu^2 \cdot \epsilon^2}{\sigma^2}\right)^{2^{i-1}}. \tag{8}$$

Then, for the case of $i' = i$, according to Lemma 3.1, we have,

$$\begin{aligned}\mathbf{E}[C_i] &= \frac{\mu^2 \cdot \epsilon^2}{\sigma^2} \cdot \mathbf{E}[C_0 \cdot C_1 \cdots C_{i-1}] \\ &\geq \frac{\mu^2 \cdot \epsilon^2}{\sigma^2} \cdot \mathbf{E}[C_0 \cdot C_1 \cdots C_{i-2}] \cdot \mathbf{E}[C_{i-1}],\end{aligned} \tag{9}$$

[2]Here, we assume that all line segments within the same level exhibit equal coverage. A more rigorous analysis can be established by using concentration bounds [4].

Results	Lookup Complexity	Insertion Complexity
LIPP [46]	$O(\log N)$	$O(\log^2 N)$
ALEX [8]	$O(\log N)$	$O(\log^2 N)$
DPGM-Index [10]	$O(\log^2 N)$	$O(\log N)$
Ours	$O(\log N \log\log N)$	$O(\log N)$

Table 5: Summary of existing theoretical results for dynamic learned indexes. "Ours" refers to the improved complexity results we establish for DPGM-Index.

considering that C_{i-1} is positively correlated with $C_0 \cdot C_1 \cdots C_{i-2}$. By the inductive hypothesis (i.e., Eq. (8)), we have,

$$\mathbf{E}[C_i] \geq \mathbf{E}[C_{i-1}]^2 \geq \left(\frac{\mu^2 \cdot \epsilon^2}{\sigma^2}\right)^{2^i}, \tag{10}$$

which satisfies the lower bound for $i' = i$. Thus, by induction, we conclude that Lemma 3.2 holds for all i. □

Theorem 3.3 (PGM-Index Height). *Given a set $\mathcal{K}$ of N sorted keys, denote the constant $G = \mu^2\epsilon^2/\sigma^2$, w.h.p., the height of a PGM-Index with error parameter $\epsilon_i = \epsilon_\ell = \epsilon$ is bounded by*

$$H_{PGM} = O(\log_2 \log_G N) = O(\log\log N). \tag{11}$$

Proof. Due to page limits, we provide only an intuitive proof sketch. A more rigorous proof can be derived using a technique similar to that in Theorem 4 of [9]. According to Definition 2.2, the construction of a PGM-Index terminates when the current level consists of exactly one line segment (i.e., reaching the root level). Intuitively, the index height H_{PGM} can be solved by letting

$$\frac{N}{\prod_{i=0}^{H_{PGM}-1} \mathbf{E}[C_i]} = O(1). \tag{12}$$

According to Theorem 3.2, we have,

$$\prod_{i=0}^{H_{PGM}-1} \mathbf{E}[C_i] \geq \prod_{i=0}^{H_{PGM}-1} G^{2^i} \geq G^{\sum_{i=0}^{H_{PGM}-1} 2^i} \geq G^{2^{H_{PGM}-1}}. \tag{13}$$

Thus, Eq. (12) can be solved by $H_{PGM} = O(\log_2 \log_G N)$. □

Theorem 3.4 (Space and Time Complexity). *Given a set $\mathcal{K}$ of N sorted keys, a PGM-Index with $\epsilon_i = \epsilon_\ell = \epsilon$ can process an index lookup query in $O(\log\log N)$ time using $O(N/G)$ space.*

Proof. According to Definition 2.2, querying a PGM-Index requires H_{PGM} times search operations, each within a range of $2\cdot\epsilon+1$. According to Theorem 3.3, the total index lookup time should be $O(H_{PGM} \cdot \log_2(2 \cdot \epsilon + 1)) = O(\log\log N)$.

We further analyze the space complexity of a PGM-Index, specifically the total number of line segments required to satisfy the error constraint ϵ. According to Definition 2.2 and Lemma 3.2, the h-th level contains at most $N/\prod_{i=0}^{h} G^{2^i}$ line segments. Thus, the upper bound on the total number of segments can be derived as,

$$\begin{aligned} \sum_{h=0}^{H_{PGM}-1} \frac{N}{\prod_{i=0}^{h} G^{2^i}} &\leq \sum_{h=0}^{H_{PGM}-1} \frac{N}{G^{h+1}} \leq N \cdot \frac{1 - \frac{1}{G^{H_{PGM}}}}{G-1} \\ &\leq G/(G-1) = O(N/G), \end{aligned} \tag{14}$$

considering that $\prod_{i=0}^{h} G^{2^i} \geq \prod_{i=0}^{h} G^{2^0} \geq G^{h+1}$. □

Note that the $O(N/G)$ space complexity of the PGM-Index is tight, as a **one-layer** PGM-Index requires $O(N/G)$ segments according to Theorem 2.4.

Key Range	ϵ	Height	Segments	Memory	$\overline{Cov}$
$[0, 10^8]$	4	4	129,503	2,078 KB	77
$\mu = 10$	8	3	37,732	604 KB	265
$\sigma^2 = 100.19$	16	3	10,224	163 KB	978
$[0, 10^9]$	4	3	129,659	2,080 KB	77
$\mu = 100$	8	3	37,601	602 KB	266
$\sigma^2 = 10007.7$	16	3	10,124	162 KB	988
$[0, 10^{10}]$	4	3	129,586	2,079 KB	77
$\mu = 1000$	8	3	37,597	602 KB	266
$\sigma^2 = 999750$	16	3	10,217	164 KB	979

Table 6: PGM-Index statistics on 10 million synthetic uniform keys with different ranges.

We then investigate the complexities of the PGM-Index under *dynamic* workloads. To support updates, the dynamic PGM-Index [9] (DPGM-Index) employs an LSM-tree-like [28] approach, maintaning a sequence of PGM-Indexes over key sets $S_0, \cdots, S_b$ of sizes at most $2^0, \cdots, 2^b$, where $b = \Theta(\log N)$. To insert a new key k, it finds the first empty set S_i and builds a new PGM-Index over a merged key set $S_0 \cup \cdots S_{i-1} \cup \{k\}$. After the construction, S_i becomes the merged set and $S_0, \cdots, S_{i-1}$ are emptied. Deleting a key k is processed by inserting a special key called "tombstone" to indicate the deletion. As there are at most $\Theta(\log N)$ PGM-Indexes, the amortized complexity for inserting N keys to a DPGM-Index is $O(\log N)$. For index lookup, in the worst case, DPGM-Index requires scanning $\Theta(\log N)$ sub-indexes, with a total complexity of $O(\log N \log\log N)$ according to Theorem 2.3. Table 5 summarizes the complexity results of updatable learned indexes.

3.3 Case Study and Discussions

To further validate the correctness of our theoretical results, we now provide a case study on uniformly distributed *keys*.

Given a key set $\mathcal{K}$, assume that all keys $k \in \mathcal{K}$ are i.i.d. samples drawn from a uniform distribution $\mathbf{U}(\alpha, \beta)$. In this case, the i-th gap on $\mathcal{K}$ can be defined as $g_i = k_{(i)} - k_{(i-1)}$, where $k_{(i)}$ and $k_{(i-1)}$ are the i-th and $(i-1)$-th *order statistics* of $\mathcal{K}$ (i.e., the i-th and $(i-1)$-th smallest values in $\mathcal{K}$). Then, for an arbitrary $i = 2, \cdots, N$, it can be shown that g_i follows a beta distribution, $g_i \sim (\beta - \alpha) \cdot \mathbf{Beta}(1, N)$, with the following mean $\mathbf{E}[g_i] = (\beta - \alpha)/(N+1)$ and variance $\mathbf{Var}[g_i] \approx (\beta - \alpha)^2/(N+1)^2$. Thus, the constant $G = \mu^2 \cdot \epsilon^2/\sigma^2 \approx \epsilon^2$, which is notably independent of the original key distribution. By Theorem 2.3 and Theorem 3.4, this result implies that, for uniform keys, a PGM-Index should have the *same* index height and memory footprint as long as N and ϵ are fixed. Table 6 reports the statistics of PGM-Indexes constructed on three synthetic uniform key sets. The results further validate the correctness of the aforementioned analysis, given that the index height and segment count remain consistent across different data ranges, depending solely on the error constraint ϵ.

4 WHY ARE PGM-INDEXES INEFFECTIVE?

The theoretical findings in Section 3 reveal that PGM-Indexes can achieve the best space-time trade-off among existing learned indexes. However, recent benchmarks [24, 39, 45] show that PGM-Index cannot outperform learned indexes without rigorous theoretical guarantees, such as RMI [33] and ALEX [8]. Such performance

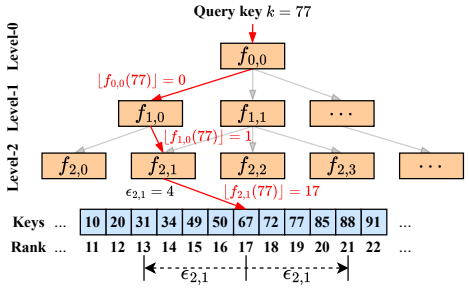


Figure 3: Illustration of a 3-layer RMI [16]. $f_{i,j}$ denotes the j-th model in the i-th level. The path in red denotes the index traversal from the root model $f_{0,0}$ to leaf. Notably, the root level is specified as level-0, opposite to the PGM-Index.

mismatch motivates us to answer the second question: Why do PGM-Indexes underperform in practice?

A Simple Cost Model. We begin by introducing a simplified cost model for an arbitrary $(\epsilon_i, \epsilon_\ell)$-PGM-Index. Recalling the PGM-Index structure as shown in Figure 1, the total index lookup time for a search key k can be modeled as the summation of the internal search cost with error constraint ϵ_i and the last-mile search cost with error constraint ϵ_ℓ:

$$Cost = (H_{PGM} - 1) \cdot C_S(\epsilon_i) + C_S(\epsilon_\ell) + H_{PGM} \cdot C_L, \quad (15)$$

where H_{PGM} is the index height, $C_S(\epsilon)$ represents the search cost within the range of $2 \cdot \epsilon + 1$, and C_L is the overhead to evaluate a linear function $y = a \cdot x + b$.

Bottleneck: Error-bounded Search. According to Theorem 2.3, the index height $H_{PGM} = O(\log \log N)$, implying that very few internal searches are required (generally fewer than **5** for 1 billion keys). Moreover, our benchmark results across various datasets and platforms indicate that evaluating a linear function typically takes **less than 10 ns**. In contrast, performing a search with $\epsilon = 64$ takes time **more than 200 ns** by adopting a standard binary search implementation (e.g., `std::lower_bound`). Based on this observation, the cost model in Eq. (15) can be simplified by neglecting the segment evaluation overhead:

$$Cost \approx (H_{PGM} - 1) \cdot C_S(\epsilon_i) + C_S(\epsilon_\ell). \quad (16)$$

The revised cost model in Eq. (16) indicates that searching a PGM-Index is dominated by performing H_{PGM} times error-bounded searches, which are generally known as *memory-bound* operations [44]. Specifically, an ϵ-bounded binary search typically involves $\lceil \log_2(2 \cdot \epsilon + 1) \rceil$ comparisons and memory accesses. Each comparison generally requires a few nanoseconds, whereas each memory access, if both cache missed and TLB missed, can take approximately 100 nanoseconds due to the asymmetric nature of the memory hierarchy in computer systems [12].

Comparison to RMI. We then investigate why RMI practically outperforms the PGM-Index. As shown in Figure 3, the major structural difference between RMI and PGM-Index lies in their internal search mechanisms. In RMI, the model prediction $f_{i,j}(k)$ directly determines the *model index* for the next level (i.e., the $(i+1)$-th level), thereby bypassing the expensive internal error-bounded search required by PGM-Index. To ensure correctness, the bottom-layer models materialize the *maximum search error* to perform a last-mile error-bounded search, similar to the PGM-Index. ALEX [8] employs a similar model-based navigation approach to avoid intermediate search, while LIPP [46] takes a step further by eliminating both internal and last-mile search errors to achieve better performance.

Data	Index	Size	Max Err.	Internal Time	Last-mile Time	Total
fb	PGM	16.1 MB	**32**	675 ns	**300 ns**	975 ns
	RMI	24.0 MB	568	**185 ns**	614 ns	**799 ns**
wiki	PGM	1.3 MB	**32**	606 ns	**270 ns**	876 ns
	RMI	1.0 MB	63	**95 ns**	317 ns	**412 ns**

Table 7: Index and query processing details. We adopt CDF-Shop [25] to find an optimized RMI configuration that matches the space cost of a (16, 32)-PGM-Index.

Data	Index	Min/Max Err.	Mean Time	Max Time	Variance
books	PGM	**32/32**	694 ns	**1720 ns**	**15802**
	RMI	41/627	**573 ns**	1985 ns	18743
osm	PGM	**32/32**	952 ns	**1813 ns**	**17487**
	RMI	63/1765584	**903 ns**	14212 ns	534086

Table 8: Distribution of latencies on 1K uniform query keys. The index configurations are consistent with Table 7.

Table 7 presents a detailed breakdown of the overheads when querying RMI and PGM-Index. Consistent with prior benchmark results [24], RMI outperforms the PGM-Index in terms of total index lookup time across all datasets. Specifically, for the PGM-Index, the internal search time constitutes a significant **69%–81%** of the total index lookup overhead. In contrast, for RMI, this ratio is considerably lower, at just **19%–27%**, supporting our earlier claim.

Is RMI the Best Choice? While RMI generally outperforms the PGM-Index, it suffers from a critical limitation: RMI cannot guarantee a maximum error *before* index construction, making its performance highly *data-sensitive*. For instance, as shown in Table 8, on dataset osm, RMI's prediction error varies significantly from **63** to $1.77 \times \mathbf{10^6}$, resulting in non-robust query latencies ranging from **440 ns** to **14212 ns**. In contrast, on dataset books, RMI's last-mile error is much narrower, leading to a much more stable query latency between **380 ns** and **1985 ns**. Such performance variability makes RMI's query latency highly dependent on the specific query key and complicates the precise cost estimation, which is critical for DBMS query optimization [13].

5 PGM++: OPTIMIZATION TO PGM-INDEX

This section introduces PGM++, a *simple yet effective* optimization to the PGM-Index by incorporating a hybrid error-bounded search strategy (▷ Section 5.1) and an automatic parameter tuner guided by well-calibrated cost models (▷ Section 5.2).

5.1 Search Strategy

PGM++ employs a *hybrid search strategy* to replace the standard binary search. To start, we discuss the impact of branch misses in standard binary search implementations.

Branch Prediction and Branch Miss. Modern CPUs rely on sophisticated branch predictors to enhance pipeline parallelism by forecasting the outcomes of conditional jump instructions (e.g., `JLE` and `JAE` on `X86`). These predictors are highly effective for simple,

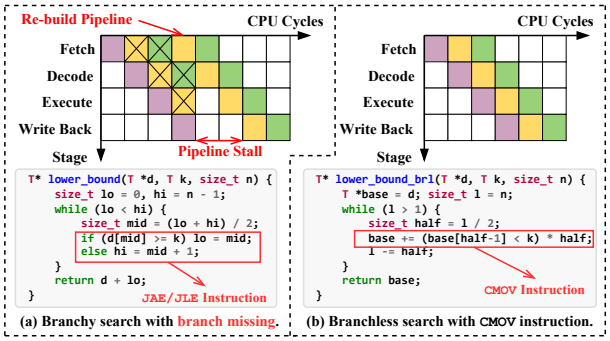
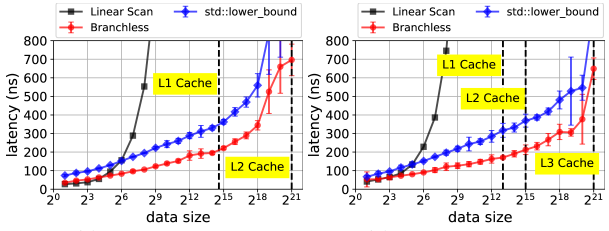


Figure 4: Illustration of the CPU pipeline status when executing: (a) standard binary search, and (b) branchless binary search enabled by the CMOV instruction.



(a) Platform ARM. (b) Platform X86-1.

Figure 5: Latency w.r.t. data sizes for linear search, standard binary search, and branchless binary search. We generate keys of varying sizes (from 2^1 to 2^{21}) by uniformly sampling from the wiki dataset and evaluate them using 10K of search keys, also randomly sampled from wiki.

repetitive tasks, such as for loops or pointer chasing, where the execution patterns are straightforward [12]. However, standard binary search implementations, like the widely used `std::lower_bound`, branching patterns are quite random, leading to a high branch miss rate of approximately **50%** [35]. As illustrated in Figure 4(a), a branch miss stalls the entire CPU pipeline until the branch condition is resolved (e.g., `d[mid]>=k` in line 5 of function `lower_bound`).

Branchless Search. A simple optimization [35] to the standard binary search is to remove the branches by conditional move instructions (e.g., `CMOV` on `X86` and `MOVGE` on `ARM`), which executes *both* sides of a branch and keeps the valid one based on the evaluated condition. As illustrated in Figure 4(b), eliminating branches (function `lower_bound_brl`) improves CPU pipeline utilization, thus reducing total search time.

Notably, `CMOV` is not a "silver bullet". It disables the native branch predictor and introduces additional overhead [35]. On large datasets (>LLC size), the performance advantage of branchless searches diminishes as memory access latency dominates the total overhead. However, this extra cost is *negligible* particularly when the search range fits within the L2 cache, making `CMOV` performance-worthy for PGM-Index (usually $\epsilon \leq 1024$).

Benchmark Results. Figure 5 presents the benchmark results for three search methods on synthetic `uint64_t` key sets of varying sizes: branchy binary search (`std::lower_bound`), branchless binary search (similar to that in Figure 4(b)), and linear scan. The results indicate that branchless search consistently demonstrates superior performance across most data sizes, except for very small datasets (e.g., $N \leq 16$), where the linear scan proves more efficient.

Dataset	PGM++	PGM++ w/o ❶	PGM++ w/o ❷	PGM
fb	545 (1×)	782 (↑43%)	636 (↑17%)	911 (↑67%)
wiki	428 (1×)	685 (↑60%)	522 (↑22%)	828 (↑93%)
books	501 (1×)	757 (↑51%)	562 (↑12%)	857 (↑71%)
osm	696 (1×)	967 (↑39%)	807 (↑16%)	1078 (↑55%)

Table 9: Ablation study results for PGM++. Both PGM and PGM++ use a fixed configuration where $\epsilon_i = 16$ and $\epsilon_\ell = 64$. The reported figures represent the average query latency (in nanoseconds) measured over 1K random queries.

Compared to `std::lower_bound`, the branchless search achieves a performance improvement of approximately **1.2×** to **1.6×**.

It is noteworthy that other search algorithms, such as k-ary search and interpolation search [35], are not considered in this work. This is because, the search range in the PGM-Index is typically small (e.g., $\epsilon \leq 1024$), where more advanced search algorithms do not *consistently* outperform a branchless binary search. Based on the above discussions, PGM++ adopts two simple yet effective optimizations to boost lookup query processing:

Optimization ❶: Hybrid Branchless Search. PGM++ adopts a *hybrid search strategy* for both internal and last-mile searches: for search ranges exceeding a pre-defined threshold δ, PGM++ employs the *branchless binary search*, while for smaller search ranges, it switches to a simple linear scan. In contrast, the original PGM-Index implementation [30] also adopts a hybrid approach, but combines linear scan with the *branchy binary search* implemented via std::lower_bound.

Optimization ❷: Layer Bypassing. Recall that PGM-Index exhibits a highly *flat* hierarchical structure where the *non-bottom* layers contain very few line segments. Thus, instead of recursively searching from the root, PGM++ skips all layers until reaching the first layer whose next layer is considered *dense* (segment count $> \delta$). This strategy, named *layer bypassing*, effectively reduces search overhead, particularly in a *cold-cache* environment.

Ablation Study. To further demonstrate the effectiveness of the proposed optimizations, we conduct an ablation study by reporting the query performance of PGM++ with either **Optimization ❶** or **❷** disabled. As shown in Table 9, the results indicate that the hybrid branchless search and the skipping of unnecessary layers reduce index lookup costs by up to **60%** and **22%**, respectively, compared to the original PGM-Index. Additionally, a more low-level evaluation is provided by reporting CPU's branch misses (measured by `Linux perf`) for PGM++ with and without the aforementioned hybrid branchless search (HBS) operator. As shown in Figure 6, by employing HBS, the total branch misses can be reduced from **1.51**×to **9.88**×compared to the branchy baseline, strongly positively correlated with the reduction in query latencies.

5.2 Calibrated Cost Model

To efficiently and effectively determine the error bounds for internal search (ϵ_i) and last-mile search (ϵ_ℓ), we first establish cost models that estimate the space and time overheads without the need for physically constructing the PGM-Index.

Space Cost Model. As discussed earlier in Section 3.1 and Table 4, the space overhead of a PGM-Index is dominated by the number of segments in the bottom layer (denoted by $L(\epsilon_\ell)$), which accounts

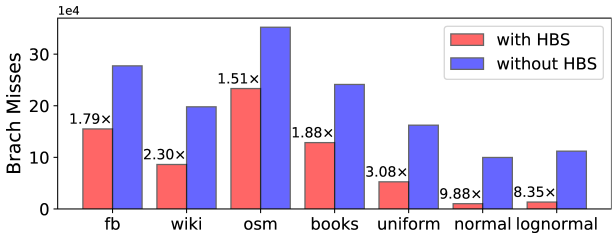


Figure 6: Branch misses of processing 1K lookup queries for PGM++ with and without hybrid branchless search (HBS). The error parameters are fixed to $\epsilon_i = 16$ and $\epsilon_\ell = 64$.

for up to > **99.9%** of the total space cost. Therefore, we focus *solely* on the leaf segments and ignore the internal segments. The total space cost of an $(\epsilon_i, \epsilon_\ell)$-PGM-Index is then given by $M = L(\epsilon_\ell) \cdot \text{sizeof}(seg)$, where $\text{sizeof}(seg)$ is the number of bytes required to encode a line segment $seg = (s, a, b)$. Typically, $\text{sizeof}(seg) = 16$ for `uint64_t` keys and `float` slope and intercept.

Estimation of Leaf Segment Count. Theorem 2.4 implies a straightforward estimation to L, which is $L \approx N\sigma^2/\epsilon_\ell^2\mu^2$. However, extreme gap values can easily influence the ratio of $h_D = \sigma^2/\mu^2$, leading to an overestimation of the necessary segment count. For example, according to Table 3, on dataset `osm`, $h_D = 1.27 \times 10^6$, and according to Theorem 2.4, the segment count when $\epsilon_\ell = 16$ can be estimated as, $\mu^2 \cdot \epsilon^2/\sigma^2 = 16^2/1.27 \times 10^6 \approx 0.0002$, which is far away from 129, the observed segment coverage. To mitigate this, we clip the observed gaps at the 1%- and 99%-quantiles and re-calculate h_D based on the clipped gaps (as reported in column h_D of Table 3). After removing the extreme gaps, the revised $h_D = 5.39$ on dataset `osm`, and the corresponding estimated segment coverage is $16^2/5.39 \approx 71.3$, which is much closer to the observed value.

However, this clipping-based estimation, which still relies on the *global* picture of gap distribution, remains *coarse* for practical datasets due to the inherent heterogeneity in gap distributions. To offer a better estimation of L, we partition the gaps into a set of consecutive and disjoint chunks $\mathcal{P}$ (such that $\sum_{P\in\mathcal{P}} |P| = N$) using a kernel-based change-point detection (KCPD) algorithm [1]. The refined estimator of L then becomes:

$$L(\epsilon_\ell) \approx \sum\nolimits_{P\in\mathcal{P}} N_P\sigma_P^2/\epsilon_\ell^2\mu_P^2, \tag{17}$$

where N_P, μ_P, and σ_P^2 represent the size, mean, and variance of gaps within partition $P \in \mathcal{P}$, respectively. Notably, the original dynamic programming-based KCPD algorithm [1] yields a complexity of $O(N^2)$, which is prohibitive for large key sets. Thus, in implementation, we uniformly pre-sample a small portion (0.05%) of each dataset, on which the KCPD algorithm is invoked.

Time Cost Model. As discussed in Section 4, the total index lookup cost includes two parts: internal search cost and last-mile search. As we adopt a hybrid search strategy, the simplified cost model introduced in Eq. (16) can be refined as follows,

$$Cost(\epsilon_i, \epsilon_\ell) = Cost_{\text{internal}} + Cost_{\text{last-mile}} \tag{18a}$$

$$Cost_{\text{last-mile}} = \lceil \log_2(2 \cdot \epsilon_\ell + 1) \rceil \cdot C_{\text{miss}} \tag{18b}$$

$$Cost_{\text{internal}} = \max\{(H_{PGM} - 1), 1\} \cdot (C_S(\epsilon_i) + C_{\text{segment}}) \tag{18c}$$

$$C_S(\epsilon_i) = \begin{cases} C_{\text{linear}} & \text{if } 2 \cdot \epsilon_i + 1 \le \delta \\ \lceil \log_2(2 \cdot \epsilon_i + 1) \rceil \cdot C_{\text{hit}} & \text{if } 2 \cdot \epsilon_i + 1 > \delta \end{cases} \tag{18d}$$

$$H_{PGM} = \lceil \log_2 \log_{\mu^2\epsilon_i^2/\sigma^2} L(\epsilon_\ell) \rceil \tag{18e}$$

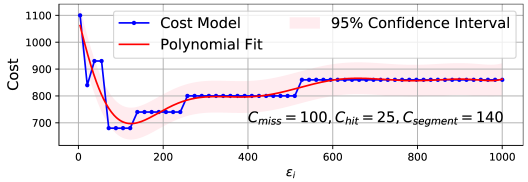
Data: the input key set $\mathcal{K}$, a space budget B
Result: the optimal configuration of ϵ_ℓ and ϵ_i
`/* Invoke KCPD on a sampled subset of` $\mathcal{K}$ `*/`
Step ❶ $\mathcal{P} \leftarrow$ KCPD(Sample($\mathcal{K}$, ratio=0.05%));
`/* Configure` ϵ_ℓ `to satisfy the index space budget` B `*/`
Step ❷ $\widetilde{\epsilon_\ell} \leftarrow \sqrt{\frac{\text{sizeof}(seg)}{B} \sum_{P\in\mathcal{P}} N_P\sigma_P^2/\mu_P^2}$;
`/* Configure` ϵ_i `to minimize the lookup cost model */`
Step ❸ $\widetilde{\epsilon_i} \leftarrow \arg\min_{\epsilon_i \in \mathcal{E}} Cost(\epsilon_i, \widetilde{\epsilon_\ell})$;

Algorithm 1: PGM++ Parameter Tuning

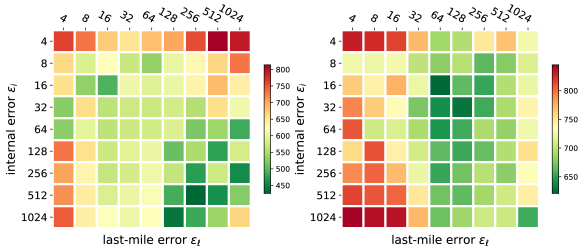
where (a) C_{miss} and C_{hit} are the memory access costs when missing or hitting L1/L2 cache; (b) C_{segment} refers to the overhead of evaluating a linear function; (c) C_{linear} is the cost of performing a linear search within the range of $2 \cdot \epsilon_i + 1$; and (d) $L(\epsilon_\ell)$ is the leaf segment count estimated by Eq. (17). All constants in the cost model can be estimated by probe datasets for different platforms. Notably, Eq. (18b) assumes a *cold-cache* environment, as the key set $\mathcal{K}$ is large enough and the access to $\mathcal{K}$ is highly random for hardware prefetchers. Conversely, Eq. (18d) assumes a *hot-cache* environment, since the non-bottom layers contain very few segments, which are highly likely to be cache-resident after processing a few queries.

PGM++ Parameter Tuning. With the space and time cost models, the two error parameters ϵ_i and ϵ_ℓ can be automatically configured. As detailed in Algorithm 1, the parameter tuning process minimizes the estimated lookup cost while satisfying a pre-specified space budget B. Specifically, Step ❶ partitions the input key set to provide more fine-grained estimations of the gap mean and variance; based on the space cost model $L(\epsilon_\ell)$, Step ❷ estimates ϵ_ℓ by solving the equation $L(\epsilon_\ell) \cdot \text{sizeof}(seg) = B$; with a determined ϵ_ℓ, Step ❸ derives ϵ_i by minimizing the index lookup cost estimated by the cost model in Eq. (18a).

Intuitively, to minimize Eq. (18a) in Step ❸, the value of ϵ_i should neither be too large nor too small. A larger ϵ_i increases the overhead of $C_S(\epsilon_i)$ (i.e., Eq. (18d)), while a smaller ϵ_i results in more layers to traverse (i.e., H_{PGM} in Eq. (18e)). Unfortunately, deriving an analytical solution for this optimization problem is impossible, as it is non-convex and involves ceiling functions. We visualize the cost



(a) Lookup time cost model on `wiki` with $\epsilon_\ell = 16$ on `X86-1`. The red line is a polynomial fitting to the cost model.



(b) Lookup cost on `wiki`. **(c) Lookup cost on `books`.**

Figure 7: Observed index lookup overhead (unit: ns) of PGM++ on `X86-1` w.r.t. different combinations of $(\epsilon_i, \epsilon_\ell)$.

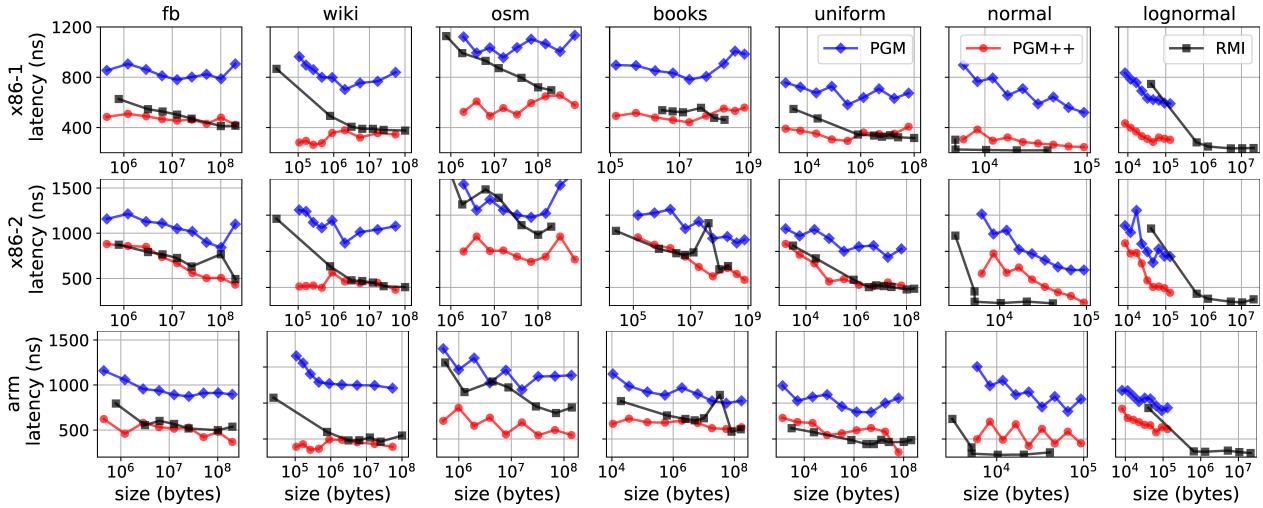


Figure 8: Space and time tradeoffs for seven datasets on three platforms (static workload).

model computed by Eq. (18a) on dataset `wiki` in Figure 7a, with constants $C_{\text{miss}}, C_{\text{hit}}, C_{\text{segment}}$ estimated on platform `X86-1`. Although non-convex, from the whole picture, the overall cost model exhibits an *approximate* "U"-shape, aligning with the true query latencies observed in Figure 7bc. In practice, we simply enumerate all possible ϵ_i within a discrete candidate set $\mathcal{E} = \{2^2, \cdots, 2^{10}\}$ to find the near-optimal value, which is efficient as $|\mathcal{E}|$ is typically small. A detailed evaluation of the accuracy and effectiveness of our cost model is provided in Section 6.4.

Remarks. Our cost model for PGM++ can be easily extended to *any* PGM-Index variants like [10, 52]. In contrast to existing cost models for learned indexes (mostly based on RMI) like [51], our cost model is *workload-independent*, relying solely on gap distribution characteristics and platform-aware cost constants. These features enhance the robustness of parameter tuning, as the cost is optimized for *all* queries rather than being tailored to a specific workload.

6 EXPERIMENTAL STUDY

To answer the question of whether PGM++ can reverse the "ineffective" scenario of PGM-Indexes, we perform experimental studies under the micro-benchmark detailed in Section 2.3.

6.1 Experiment Setups

Query Workloads. Following the settings of recent experimental studies [39, 45], we evaluate learned indexes under query workloads with varying read-to-write ratios, including ❶ **`read-only`** (0% writes), ❷ **`ready-heavy`** (20% writes), ❸ **`balanced`** (50% writes), ❹ **`write-heavy`** (80% writes), and ❺ **`write-only`** (100% writes). For `read-only` workload (a.k.a. static workload), we bulk-load the entire key set and issue 100M lookup queries. For workloads involving writes, we first bulk-load 100M keys, then issue lookup and insertion queries with varying ratios using the remaining keys.

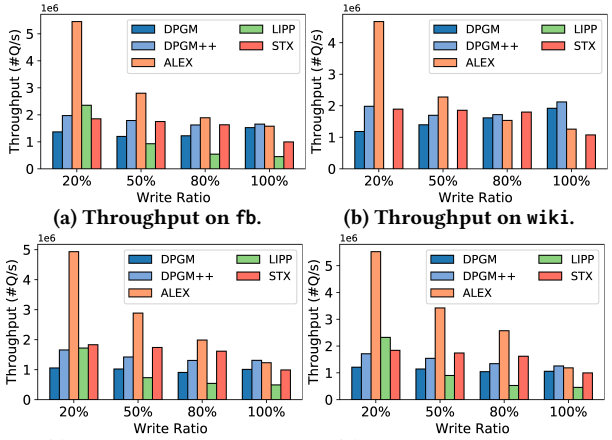
Compared Methods. For static workload, we evaluate three learned indexes: ❶ RMI, the optimized recursive model index [16, 24] tuned by CDFShop [25]; ❷ PGM, the original PGM-Index implementation [10]; and ❸ PGM++, our optimized PGM-Index variant. For PGM, we construct 9×9 PGM-Indexes with $(\epsilon_i, \epsilon_\ell) \in \mathcal{E} \times \mathcal{E}$, where $\mathcal{E} = \{2^2, \cdots, 2^{10}\}$. Then, for each $\epsilon_\ell \in \mathcal{E}$, the fastest PGM-Index is reported. Similarly, for PGM++, we adopt the $(\epsilon_i, \epsilon_\ell)$ configuration tuned by our cost model (Section 5.2) for each $\epsilon_\ell \in \mathcal{E}$.

For dynamic workloads, we evaluate four updatable learned indexes: ❶ DPGM, the dynamic PGM-Index variant [10] by employing a LSM-tree-like structure; ❷ DPGM++, our optimized version to DPGM by implementing hybrid branchless search and optimal error configuration; ❸ ALEX [8], a learned index using gapped array to handle updates; and ❹ LIPP [46], a learned index that improves read efficiency by eliminating last-mile search errors. We tune DPGM and DPGM++ in the same way as PGM and PGM++. For ALEX, we tune index parameters to achieve a similar amount of memory as LIPP. In addition to learned baselines, we also include the popular STX B+-tree [38] in the experiments.

6.2 Results on Static Workloads

Overall Evaluation. Figure 8 presents the trade-offs between index lookup overhead and storage cost across all seven datasets and three platforms on static workloads. The results show that, in terms of index lookup time, PGM++ consistently outperforms PGM by a factor of **1.2×** ∼ **2.2×** with the same index size (ϵ_ℓ), supporting our bottleneck analysis for PGM-Indexes (Section 4). Compared to the optimized RMI, PGM++ delivers better or, in some cases, comparable lookup efficiency, achieving speedups of up to **1.56×**. An outlier case is on dataset `normal` and `lognormal`, RMI outperforms PGM++ and PGM. The reason is that the optimized RMI implementation adopts non-linear models (e.g., cubic spline interpolation [25]), which can fit normal and lognormal keys very well (maximum error < 4). However, on real-world datasets, RMI fell short in fitting the data with bounded error, leading to non-robust latencies as reported in Table 8 and discussed in Section 4.

Space-time Trade-off. In most cases, PGM++ offers the best space-time trade-off. However, interestingly, unlike RMI, whose performance generally improves with increased index memory usage, PGM++ exhibits an "irregular" pattern in its time-space relationship. This is because we always report the performance of the *optimal* index configuration for each ϵ_ℓ. When allocating more memory budget by decreasing ϵ_ℓ, the number of leaf segments increases, leading to a higher index height. While a smaller ϵ_i reduces last-mile search costs, the increase in internal index traversal costs



(a) Throughput on `fb`. (b) Throughput on `wiki`.

(c) Throughput on `osm`. (d) Throughput on `books`.

Figure 9: Query throughput (#Queries/s) across different write ratios for dynamic workloads on `X86-1`.

may dominate, implying that the total query latency does not necessarily improve with more space. An important takeaway here is that our parameter tuner can find configurations to provide competitive query efficiency, even under very limited space constraints. For example, on dataset `wiki`, PGM++ uses 0.1 MB of memory to outperform an RMI with over 100 MB space.

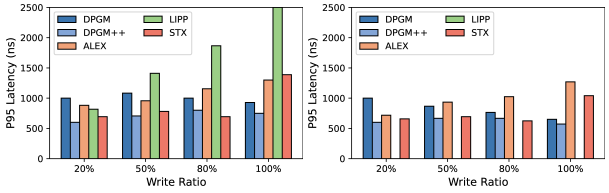
Influence of Architecture. From Figure 8, the comparison results vary across different platforms. For dataset `osm`, compared to PGM, PGM++ achieves an average speedup ratio of **1.78**× on `x86-1` and `arm`. However, such a speedup decreases to **1.32**× on platform `x86-2`. This is because the memory access latency on `x86-2` is much higher than that on `x86-1`, which reduces the improvement brought by adopting the hybrid search strategy.

6.3 Results on Dynamic Workloads

In Figure 9, we present experimental results on dynamic workloads across varying write ratios.

Comparison with DPGM. For read-heavy workloads (20% writes), DPGM++ improves the query throughput by up to **1.68**× compared to DPGM, demonstrating the effectiveness of our proposed optimizations (hybrid branchless search and parameter tuner) on both static and dynamic workloads. However, as the write ratio increases, the performance improvement becomes less significant. This is because both DPGM and DPGM++ employ an LSM-tree-like approach to handle insertions, where the major overhead comes from the compaction operation triggered when a sub-index is full.

Comparison with ALEX and LIPP. Except for write-heavy and write-only workloads, ALEX always achieves the highest throughput, consistent with recent benchmarks [45]. LIPP achieves the second-highest throughput in read-heavy workloads, following ALEX. DPGM and DPGM++ exhibit lower performance in read-heavy scenarios due to the underlying LSM-like structure, where each lookup must traverse multiple levels to locate the relevant sub-index, incurring additional overhead. According to the theoretical results in Table 5, DPGM++ has a lookup cost of $O(\log N \cdot \log\log N)$, while ALEX achieves $O(\log N)$. Although the additional $\log\log N$ factor is asymptotically small, in an **in-memory** index setting, where each lookup typically takes only a few hundred nanoseconds, even one additional cache miss caused by error-bounded search (on the order of ~100ns) can introduce noticeable latency.



(a) P95 latency on `fb`. (b) P95 latency on `wiki`.

Figure 10: Long-tail latency across different write ratios for dynamic workloads on `X86-1`.

However, when the write ratio exceeds 50%, the throughput of both ALEX and LIPP drops rapidly due to the high update overhead on their internal structures. In contrast, DPGM++ excels in high-write workloads, achieving up to **1.68**× the throughput of ALEX, thanks to the LSM-tree-like update strategy.

While DPGM++ is noticeably inferior to ALEX on read-heavy workloads, the error-bounded design of PGM++ offers more robust performance across a variety of workload patterns. As shown in Figure 10, we report the **long-tail latency (P95)** under different write ratios for each index. While ALEX demonstrates higher throughput in read-heavy scenarios, its P95 latency is worse than that of DPGM++ across different write ratios. This characteristic is particularly valuable in real-world DBMS environments, where query optimizers rely on predictable and stable index access costs for accurate plan selection.

6.4 Cost Model and Parameter Tuner

Space Cost Model. As discussed earlier, the leaf segment count (L) dominates the PGM-Index space cost. Here, we evaluate three segment count estimators introduced in Section 5.2. (a) SIMPLE: Directly apply Theorem 2.4 on the *entire* gap distribution. (b) CLIP: Apply Theorem 2.4 while excluding extreme gap values (those outside the P1 and P99 quantiles). (c) ADAP: Partition gaps into disjoint chunks and aggregate the segment count estimates from each chunk (Eq. (17)). As shown in Figure 11, ADAP consistently achieves more accurate estimations across all datasets, nearly overlapping the ground truth (TRUE). SIMPLE performs the worst on real datasets, validating our discussion in Section 2.3 that extreme gap values significantly affect estimation accuracy. CLIP also delivers accurate results on real datasets (especially `fb`), as the gaps are *nearly* identically distributed after removing the extreme values.

Time Cost Model. For each pair of $(\epsilon_i, \epsilon_\ell) \in \mathcal{E} \times \mathcal{E}$, where $\mathcal{E} = \{2^j \mid j = 2, \cdots, 10\}$, we estimate the index lookup overhead using our time cost model (Eq. (18a)–Eq. (18e)) and then measure the actual lookup time by constructing the corresponding $(\epsilon_i, \epsilon_\ell)$-PGM-Index. Figure 12 plots the true index lookup overhead against the cost model's estimation (both normalized). The closer the points are to the line $y = x$, the more accurate the estimation. Our results show that the cost model closely approximates the true lookup overhead, particularly for synthetic datasets. This is because synthetic datasets strictly follow i.i.d. gaps assumptions, leading to more precise estimates of the index height (Eq. (18e)). Although the estimation on real-world datasets is less accurate than on synthetic datasets, the overall trend remains reliable, making it sufficient for subsequent parameter tuning.

Parameter Tuning Strategy. We finally evaluate the effectiveness of PGM++'s parameter tuning strategy (Algorithm 1). For each ϵ_ℓ,

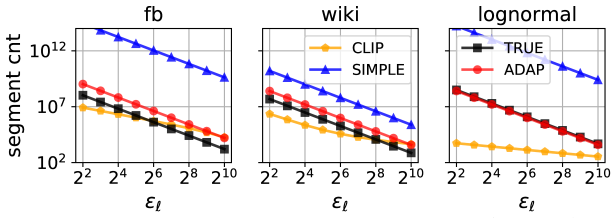


Figure 11: Evaluation of the space cost model (Eq. (17)). For each ϵ_ℓ, we compare three leaf segment count estimators: (a) SIMPLE, (b) CLIP, and (c) ADAP.

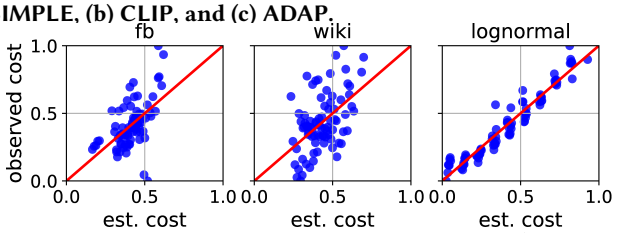


Figure 12: Evaluation of the time cost model (Eq. (18a)–Eq. (18e)). We plot the true index lookup costs (normalized) against the estimated costs (normalized) on x86-1, where each point represents a unique $(\epsilon_i, \epsilon_\ell)$ configuration.

we record the index lookup overhead for PGM-Indexes with different ϵ_i configurations: (a) PGM++, where ϵ_i is tuned by Algorithm 1; and (b) Random, where ϵ_i is randomly picked. Figure 13 reports the *relative* index lookup overhead w.r.t. different ϵ_ℓ settings. From the results, in **46%** of cases, PGM++ successfully picks the **optimal** ϵ_i, and in **91%** of cases, PGM++ finds a configuration that is only $<$ **10%** worse than the optimal one in terms of actual index lookup overhead. Moreover, our parameter tuner (<1 µs) is much more efficient than CDFShop [25], which takes over 10 minutes to optimize RMI (>10 minutes). This is because CDFShop requires physically constructing the index, while our method only depends on gap mean and variance, which can be pre-computed and re-used.

7 RELATED WORK

Learned Indexes. Indexing one-dimensional sorted keys has been extensively studied for decades. While tree-based indexes (e.g., B+-tree [5], FAST [14], ART [3], Wormhole [48], HOT [3], etc.) remain the backbone of modern DBMS, a new class of index structure, known as *learned index*, has recently gained significant attention in both academia and industry [7, 8, 10, 16, 18–23, 40, 42, 46, 47, 47, 51–53]. Learned indexes directly fit the CDF of input keys with controllable error to perform an exact last-mile search. By properly organizing the model structure, learned indexes offer the potential for superior space-time trade-offs compared to conventional tree-based indexes [24, 45]. In this work, we delve deeply into the theoretical and empirical aspects of the popular PGM-Index, demonstrating its potential to be *practically* embedded into real DBMS.

Learned Index Theories. Unlike tree-based indexes supported by well-established theoretical foundations, the effectiveness of learned indexes has largely been demonstrated through *empirical results*. Ferragina et al. [9, 10] first prove that the expected time and space complexities of a PGM-Index with error constraint ϵ on N keys should be $O(\log N)$ and $O(N/\epsilon^2)$, respectively. In parallel, another recent work [50] focuses on an RMI variant with *piece-wise constant* models, achieving an index lookup time of $O(\log \log N)$

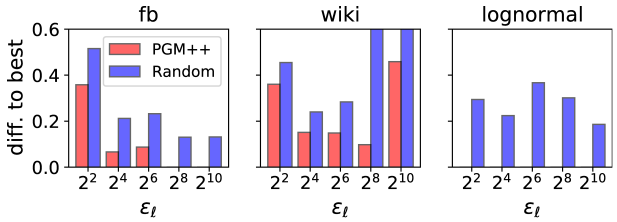


Figure 13: Evaluation of PGM++ parameter tuner. The y-axis is the relative difference compared to the optimal configuration (i.e., $t_{\text{observed}}/t_{\text{opt}} - 1$). The closer to 0, the better.

but using *super-linear* space of $O(N \log N)$. In this work, we tighten the results of [9], demonstrating that PGM-Indexes can achieve a sub-logarithmic lookup time of $O(\log \log N)$ while maintaining *linear* space complexity of $O(N/\epsilon^2)$. To the best of our knowledge, this is the tightest bound among all existing learned indexes.

Learned Index Cost Model. Modeling the space and time overheads of an index structure is crucial for both index parameter configuration and DBMS query optimization. Existing learned indexes mainly adopt a workload-based cost model, which assumes prior knowledge of the query distribution [25, 51]. In contrast, by extending the theoretical results, we establish a cost model for PGM-like indexes without *any* assumptions on query workloads. As our cost model is simple, parameter tuning based on it is much more efficient than workload-driven approaches, making it more feasible to be integrated into practical DBMS.

8 CONCLUSION AND FUTURE WORK

This work provides an in-depth theoretical and experimental revisit to the PGM-Index. We establish a new bound for the PGM-Index by showing the $O(\log \log N)$ index lookup time while using $O(N/G)$ space. We further identify that costly internal error-bounded search operations have become a bottleneck in practice. To address this, we propose PGM++, a simple yet effective variant of the PGM-Index, which optimizes the internal search operator and optimally configures index parameters using an accurate cost model. Extensive experimental results demonstrate that PGM++ significantly enhances the original PGM-Index across both static and dynamic workloads, making it a promising index for practical DBMS.

ACKNOWLEDGMENTS

Dr. Qiyu Liu is supported by the Fundamental Research Funds for the Central Universities (No. 5330501376). Lei Chen's work is partially supported by National Key Research and Development Program of China Grant No. 2023YFF0725100, National Science Foundation of China (NSFC) under Grant No. U22B2060, Guangdong-Hong Kong Technology Innovation Joint Funding Scheme Project No. 2024A0505040012, the Hong Kong RGC GRF Project 16213620, RIF Project R6020-19, AOE Project AoE/E-603/18, Theme-based project TRS T41-603/20R, CRF Project C2004-21G, Key Areas Special Project of Guangdong Provincial Universities 2024ZDZX1006, Guangdong Province Science and Technology Plan Project 2023A0505030011, Guangzhou municipality big data intelligence key lab, 2023A03J0012, Hong Kong ITC ITF grants MHX/078/21 and PRP/004/22FX, Zhujiang scholar program 2021JC02X170, Microsoft Research Asia Collaborative Research Grant, HKUST-Webank joint research lab and 2023 HKUST Shenzhen-Hong Kong Collaborative Innovation Institute Green Sustainability Special Fund, from Shui On Xintiandi and the InnoSpace GBA.

REFERENCES

[1] Sylvain Arlot, Alain Celisse, and Zaïd Harchaoui. 2019. A Kernel Multiple Change-point Algorithm via Model Selection. *J. Mach. Learn. Res.* 20 (2019), 162:1–162:56.

[2] biglittle [n.d.]. big.LITTLE: Balancing Power Efficiency and Performance. https://www.arm.com/en/technologies/big-little. Accessed: 2024-06-12.

[3] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD Conference*. ACM, 521–534.

[4] Kai Lai Chung. 2000. *A course in probability theory*. Elsevier.

[5] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137.

[6] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[7] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *OSDI*. USENIX Association, 155–171.

[8] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD Conference*. ACM, 969–984.

[9] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why Are Learned Indexes So Effective?. In *ICML (Proceedings of Machine Learning Research)*, Vol. 119. PMLR, 3123–3132.

[10] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175.

[11] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *SIGMOD Conference*. ACM, 1189–1206.

[12] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[13] Matthias Jarke and Jürgen Koch. 1984. Query Optimization in Database Systems. *ACM Comput. Surv.* 16, 2 (1984), 111–152.

[14] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD Conference*. ACM, 339–350.

[15] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *aiDM@SIGMOD*. ACM, 5:1–5:5.

[16] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD Conference*. ACM, 489–504.

[17] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. IEEE Computer Society, 302–313.

[18] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A Distribution-Driven Learned Index. *Proc. VLDB Endow.* 16, 9 (2023), 2212–2224.

[19] Qiyu Liu, Maocheng Li, Yuxiang Zeng, Yanyan Shen, and Lei Chen. 2025. How good are multi-dimensional learned indexes? An experimental survey. *The VLDB Journal* 34, 2 (2025), 1–29.

[20] Qiyu Liu, Yuxin Luo, Mengke Cui, Siyuan Han, Jingshu Peng, Jin Li, and Lei Chen. 2025. BitTuner: A Toolbox for Automatically Configuring Learned Data Compressors. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 4548–4551.

[21] Qiyu Liu, Yanyan Shen, and Lei Chen. 2021. LHist: Towards Learning Multi-dimensional Histogram for Massive Spatial Data. In *ICDE*. IEEE, 1188–1199.

[22] Qiyu Liu, Yanyan Shen, and Lei Chen. 2022. HAP: An Efficient Hamming Space Index Based on Augmented Pigeonhole Principle. In *SIGMOD Conference*. ACM, 917–930.

[23] Qiyu Liu, Libin Zheng, Yanyan Shen, and Lei Chen. 2020. Stable Learned Bloom Filters for Data Streams. *Proc. VLDB Endow.* 13, 11 (2020), 2355–2367.

[24] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.

[25] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *SIGMOD Conference*. ACM, 2789–2792.

[26] openstreetmap [n.d.]. OpenStreetMap. https://www.openstreetmap.org/. Accessed: 2024-06-12.

[27] Joseph O'Rourke. 1981. An On-Line Algorithm for Fitting Straight Lines Between Data Ranges. *Commun. ACM* 24, 9 (1981), 574–578.

[28] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[29] Yehoshua Perl, Alon Itai, and Haim Avni. 1978. Interpolation Search - A Log Log N Search. *Commun. ACM* 21, 7 (1978), 550–553.

[30] pgm [n.d.]. PGM-Index. https://github.com/gvinciguerra/PGM-index. Accessed: 2024-06-12.

[31] postgresql [n.d.]. PostgreSQL. https://www.postgresql.org/docs/current/indexes.html. Accessed: 2024-06-12.

[32] pytorch [n.d.]. PyTorch. https://pytorch.org/. Accessed: 2024-06-12.

[33] rmi [n.d.]. rmi. https://github.com/learnedsystems/RMI/. Accessed: 2024-06-12.

[34] Peter Van Sandt, Yannis Chronis, and Jignesh M. Patel. 2019. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?. In *SIGMOD Conference*. ACM, 36–53.

[35] Lars-Christian Schulz, David Broneske, and Gunter Saake. 2018. An Eight-Dimensional Systematic Evaluation of Optimized Search Algorithms on Modern Processors. *Proc. VLDB Endow.* 11, 11 (2018), 1550–1562.

[36] Lars-Christian Schulz, David Broneske, and Gunter Saake. 2018. An eight-dimensional systematic evaluation of optimized search algorithms on modern processors. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1550–1562.

[37] sparksql [n.d.]. Spark SQL. https://spark.apache.org/sql/. Accessed: 2024-06-12.

[38] stx [n.d.]. STX B+-tree. https://panthema.net/2007/stx-btree/. Accessed: 2024-06-12.

[39] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned index: A comprehensive experimental evaluation. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1992–2004.

[40] 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 *PPoPP*. ACM, 308–320.

[41] tensorflow [n.d.]. TensorFlow. https://www.tensorflow.org/. Accessed: 2024-06-12.

[42] Zhaoguo Wang, Haibo Chen, Youyun Wang, Chuzhe Tang, and Huan Wang. 2022. The Concurrent Learned Indexes for Multicore Data Storage. *ACM Trans. Storage* 18, 1 (2022), 8:1–8:35.

[43] wikidata [n.d.]. Wikidata. https://www.wikidata.org/wiki/Wikidata:Main_Page. Accessed: 2024-06-12.

[44] Samuel Williams, Andrew Waterman, and David A. Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[45] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are Updatable Learned Indexes Ready? *Proc. VLDB Endow.* 15, 11 (2022), 3004–3017.

[46] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proc. VLDB Endow.* 14, 8 (2021), 1276–1288.

[47] Shangyu Wu, Yufei Cui, Jinghuan Yu, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2022. NFL: Robust Learned Index via Distribution Transformation. *Proc. VLDB Endow.* 15, 10 (2022), 2188–2200.

[48] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A Fast Ordered Index for In-memory Data Management. In *EuroSys*. ACM, 18:1–18:16.

[49] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum error-bounded Piecewise Linear Representation for online stream approximation. *VLDB J.* 23, 6 (2014), 915–937.

[50] Sepanta Zeighami and Cyrus Shahabi. 2023. On Distribution Dependent Sub-Logarithmic Query Time of Learned Indexing. In *ICML (Proceedings of Machine Learning Research)*, Vol. 202. PMLR, 40669–40680.

[51] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: A Cache-Aware Learned Index with a Cost-based Construction Algorithm. *Proc. VLDB Endow.* 15, 11 (2022), 2679–2691.

[52] Jiaoyi Zhang, Kai Su, and Huanchen Zhang. 2024. Making In-Memory Learned Indexes Efficient on Disk. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[53] 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. *Proc. VLDB Endow.* 16, 2 (2022), 243–255.