Learned Index: A Comprehensive Experimental Evaluation

Zhaoyan Sun
Tsinghua University
Beijing, China
szy22@mails.tsinghua.edu.cn

Xuanhe Zhou
Tsinghua University
Beijing, China
zhouxuan19@mails.tsinghua.edu.cn

Guoliang Li
Tsinghua University
Beijing, China
liguoliang@tsinghua.edu.cn

ABSTRACT

Indexes can improve query-processing performance by avoiding full table scans. Although traditional indexes (e.g., B+-tree) have been widely used, learned indexes are proposed to adopt machine learning models to reduce the query latency and index size. However, existing learned indexes are (1) not thoroughly evaluated under the same experimental framework and are (2) not comprehensively compared with different settings (e.g., key lookup, key insert, concurrent operations, bulk loading). Moreover, it is hard to select appropriate learned indexes for practitioners in different settings. To address those problems, this paper detailedly reviews existing learned indexes and discusses the design choices of key components in learned indexes, including key lookup (position inference which predicts the position of a key, and position refinement which re-searches the position if the predicted position is incorrect), key insert, concurrency, and bulk loading. Moreover, we provide a testbed to facilitate the design and test of new learned indexes for researchers. We compare state-of-the-art learned indexes in the same experimental framework, and provide findings to select suitable learned indexes under various practical scenarios.

PVLDB Reference Format:
Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. Learned Index: A Comprehensive Experimental Evaluation. PVLDB, 16(8): 1992 - 2004, 2023.
doi:10.14778/3594512.3594528

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

1 INTRODUCTION

Indexes are vital to improve query performance by avoiding full table scans. Traditional indexes (e.g., B+tree) build additional data structures to guide key search. However, additional indexes not only take additional space but also are inefficient due to pointer chasing and cache miss. To address those problems, learned indexes are proposed recently (e.g., one-dimensional index [1, 7, 9–13, 16–18, 24, 27–29, 38, 40, 41, 43, 46, 47, 49, 50, 53, 58, 59], multi-dimensional index [6, 8, 25, 34, 36, 54], Bloom filter [5, 26, 33, 37, 44]), which adopt machine learning models to replace the additional structures [19, 42, 45, 51, 56, 57] such that the models can not only reduce the index size but also improve the key lookup efficiency [23, 30, 48, 60].

Given a sorted list of key-position pairs, a learned index aims to use machine learning models to predict the position of a query key.

Proceedings of the VLDB Endowment, Vol. 16, No. 8 ISSN 2150-8097.
doi:10.14778/3594512.3594528

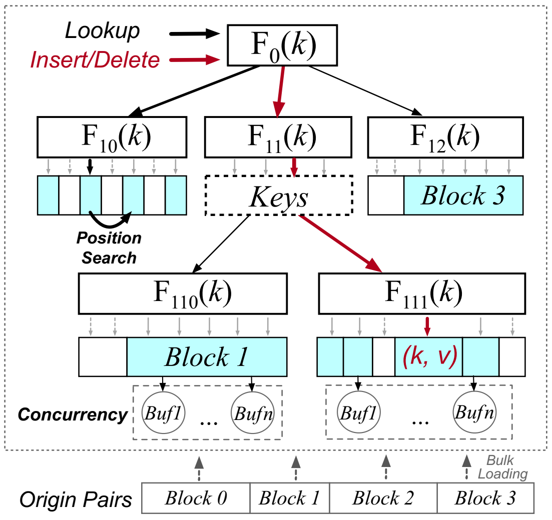


Figure 1: A General Learned Index Structure. Dotted rectangles denote some indexes have no keys in internal nodes.

Five important factors should be considered in designing a learned index (as shown in Figure 1).

(1) Key Lookup. It aims to identify the query key position efficiently. It includes position prediction and position search. The former adopts a machine learning model to predict the key position. If the prediction is correct, it can easily identify the key; otherwise it calls the latter step to re-search the key position based on the predicted position. There are different choices on model design for position prediction and position search algorithms. For model design, existing learned indexes mainly adopt a hierarchy of multiple models where each internal model predicts the position of a key in its child models and each leaf model predicts the real key position. For position search, there are various search methods (e.g., linear and binary searches), which are suitable for different scenarios (e.g., linear search works well for small ranges and small prediction errors, while binary search performs better for large ranges). Thus it is vital to select proper position search methods based on the prediction errors of machine learning models in learned indexes.

(2) Key Insert. Inserting a new key may change the index structure (e.g., node split) and lead to model retraining (for changes of key positions). Mutable learned indexes are proposed to address these problems in two ways. (*i*) In-place insert learned indexes reserve some gaps in the index in order to postpone index structure change and model retraining. (*ii*) Delta-buffer insert indexes use the pair-level/node-level/index-level buffer to allow key inserts.

(3) Key Delete. Deleting a key may change the index structure (e.g., node merge) and may also lead to model retraining. Key deletes can be handled similarly as key inserts.

(4) Concurrency. To support concurrent operations, learned indexes use different granularities of buffers to improve the throughput. On one hand, to enable intra-node concurrent queries, learned indexes maintain a buffer for each key in the node, and the updates across different keys can be concurrent. On the other hand, to enable concurrent queries across different nodes, learned indexes maintain a temporary buffer for each node and use the buffer to (1) support concurrent operations during nodes split/merge and (2) merge the buffer with the new node after the nodes splits/merge.

(5) Bulk Loading. It builds the learned index for a batch of key-position pairs. There are two types of methods. (i) Top-down methods initialize the root, split its pairs to child nodes, and process the pairs in the child nodes recursively. (ii) Bottom-up methods split the pairs to leaf nodes, extract the minimal or maximal keys from each node, and recursively process the extracted pairs. To decide how to split the pairs, there are many algorithms that consider the split overhead to effectively build the tree structure.

1.1 Our Motivation

(1) Existing Learned Indexes Are Not Evaluated Under The Same Evaluation Framework. Although there are dozens of learned indexes, there is a lack of a comprehensive evaluation framework to thoroughly compare them.

(2) There Is No Guideline To Help Practitioners Select Suitable Learned Indexes. There are multiple factors that affect the learned indexes, and it is rather hard for practitioners to select a suitable learned index in different scenarios.

(3) There Is No Testbed To Design New Learned Indexes. Designing a new learned index should implement multiple components and if there is no testbed, the researchers/practitioners have to re-design all the components which are tedious and needless.

1.2 Our Contribution

Some existing works have evaluated a subset of above index factors [2, 30, 48]. First, SOSD [30] evaluated immutable learned indexes, but did not compare the mutable learned indexes. Besides, although SOSD [30] and the workshop paper [2] evaluated learned indexes with micro-architectural metrics (e.g., cache misses), SOSD (i) did not cover some important metrics (e.g., instruction fetching, instruction encoding) and (ii) did not evaluate the execution time ratios of these metrics, which are vital to analyze the effects to index performance. And the evaluation in [2] only supported one learned index (ALEX) and did not evaluate other important learned indexes. Second, GRE [48] evaluated the mutable learned indexes, particularly in concurrency scenarios. However, GRE neither thoroughly summarized above index factors of learned indexes, nor conducted fine-grained evaluations of these factors. Different from those works, our main contributions are as follows:

(1) A Comprehensive Evaluation. We have constructed an evaluation framework and compared state-of-the-art learned indexes and traditional indexes on various datasets and workloads.

(2) An Extensive Set of Findings. We have extensively compared existing learned indexes from various aspects. We also summarize the evaluation results so as to guide practitioners to select proper indexes under various practical scenarios.

(i) Some learned indexes can outperform traditional indexes for simple data distributions (e.g., relatively smooth CDF without abrupt shifts) and read-heavy scenarios, for which they utilize machine learning models to quickly locate the key positions. However, existing learned indexes have no advantages for complicated data distributions (because the machine learning models cannot fit well) and write-heavy workloads (because tree structures should be updated and the models should be retrained).

(ii) Learned indexes have no significant advantage against traditional indexes for range queries, where most of the time is spent in scanning the sorted pairs in leaf nodes.

(iii) Learned indexes have no advantage on string keys, because it is rather hard to model and predict complicated string keys.

(iv) Learned indexes have no advantage on bulk loading, which need to iterate many times per node to determine the structure.

(v) Non-linear models often achieve higher prediction accuracy than linear models, with which learned indexes can reduce position searches and gain lower lookup latency. However, non-linear models take more training overhead and slow down the write operations during structural modifications.

(vi) Indexes often use additional structures (e.g., ART adopts hash tables in some nodes) to reduce insert/lookup latency, and thus involve large index sizes. However, learned indexes like XIndex and FINEdex gain both large index size and high insert/lookup latency, because they use extra space (e.g., pair-level buffers) to support concurrent operations, which may slow down insert/lookup operations as the search involves both the index and buffers.

(vii) The micro-architectural metrics like retiring (instruction count), bad speculation (branch-instruction misprediction), frontend bound (instruction fetching/encoding) and DRAM bound (cache miss) can reveal the read/write performances of learned indexes (e.g., reducing branch-instruction misprediction by searching only at the leaf nodes).

(ix) Learned indexes cannot outperform traditional indexes for concurrent lookups/writes, which need to (i) search both the index and delta buffers and (ii) retrain models during structural modification. However, learned indexes and traditional indexes achieve similar concurrency performance for range queries, which could incur thread collisions and are hard to optimize. For non-concurrent scenarios, DPGM has better performance for write-only workloads; LIPP is a better choice for workloads without range queries; ALEX has better performance for hybrid insert/lookup/range workloads.

(3) A Unified Testbed. We provide a testbed with many reusable components (e.g., workload generation, hyper-parameter tuning, performance evaluation), which can facilitate researchers to design and test new learned index structures with lower overhead on design, evaluation and implementation.

2 LEARNED INDEXES

In this section, we first give the definition of learned indexes. Next we describe the key factors in designing a learned index (Table 1). Note we focus on one-dimensional in-memory learned indexes [1, 7, 9, 11–13, 16, 18, 24, 40, 41, 43, 46, 47, 49, 50, 53, 58].

2.1 Learned Indexes

A learned index usually adopts a hierarchical structure, where all the original data (key-position pairs) are maintained in leaf nodes.

Table 1: Technical differences of evaluated indexes. The thick line separates immutable (top) and mutable indexes (down).

	Index	Insert		Lookup		Concurrency	Bulk Loading
		Insert Strategy	Structural Modification	Data Fitting Model	Position Search		
Learned	RMI [18]	No	No	Simple neural network	At leaf nodes	No	Top-down
	PLEX [41]	No	No	Non-linear model [4] Linear interpolation	At all nodes	No	Greedy split Bottom-up
	PGM [11]	No	No	Linear model	At all nodes	No	Greedy split Bottom-up
	DPGM (Dynamic PGM [11])	Delta-buffer	Buffer merge [35]	Linear model	At all nodes	No	Greedy split Bottom-up
	XIndex [43]	Delta-buffer	Buffer merge Error-based node split	RMI Piecewise linear regression	At all nodes	Temporary buffer	Even split Bottom-up
	FINEdex [24]	Delta-buffer	Fullness-based buffer train&merge	Piecewise linear regression	At all nodes	Pair-level buffer Buffer train&merge	Greedy split Bottom-up
	SIndex [46]	Delta-buffer	Buffer merge	(Piecewise) linear regression	At all nodes	Temporary buffer	Greedy split Bottom-up
	ALEX [7]	In-place	Fullness&cost based node expand/split/rebuild	Linear model	At leaf nodes	No	Cost-based split Top-down
	MAB+tree [1]	In-place	Fullness-based node split	Linear interpolation	At all nodes	No	Greedy split Bottom-up
	LIPP [49]	In-place	Conflict&fullness based subtree rebuild	Non-linear model	No	No	Conflict-based split Top-down
Traditional	FAST [14]	No	No	No	At all nodes	No	Bottom-up
	ART [20]	In-place	Fullness&prefix based node expand/split	No	At most nodes	No	Top-down
	B+tree [3]	In-place	Fullness-based node split	No	At all nodes	No	Bottom-up
	Wormhole [52]	In-place	Fullness-based node split	No	At all nodes	RCU [32] hash table	Bottom-up

Each node in the hierarchy contains a machine learning model that predicts the position of a key. Note, for simplicity, we assume there is no duplicated key in the original data and will explain how to solve the issue in Section 2.2.2.

Definition 2.1 (Learned Index). Let $D = \{(k_0, p_0), (k_1, p_1), \cdots\}$ be a sorted list of key-position pairs, where k_i denotes a key and p_i is the position value of k_i. Let K denote the set of keys and P denote the set of positions in D. Let $H : K \rightarrow P$ denote a mapping from K to P, where for a key k, $H(k)$ is the first position in D whose key is not smaller than k. A learned index I adopts a machine learning model F to fit the mapping H, and uses position search to correct the fitting error $|F(k) - H(k)|$.

Similar to traditional indexes, the learned index I also requires to resolve the following key factors:

(1) *lookup* (k): It uses the learned model f to identify the position of k. For each internal node from the root, it uses the model f to predict the child node k belongs to. For the leaf node, it uses the model f to predict the key position. Note that if the predicted position is incorrect, it needs to re-search the position, e.g., using the binary search algorithms as the keys are sorted (see Section 2.2).

(2) *range* (k_{left}, k_{right}): It first obtains the position of k_{left} via $lookup(k_{left})$. Note if k_{left} is not within D, it finds the first position whose key is larger than k_{left}. From the found position, it sequentially scans the leaf nodes until the key of the scanned pair is larger than k_{right}.

(3) *insert* (k_i, p_i) : It first obtains the leaf node that k_i belongs to. Then it inserts the key-position pair into the node. If the node is full, it needs to split the node and retrains the model (see Section 2.3).

(4) *delete* (k_i, p_i) : It first obtains the position of k_i via $lookup(k_i)$. Then if (k_i, p_i) exists, it removes (k_i, p_i) from D. Note that it may need to merge the nodes and retrain the model (see Section 2.4).

(5) *concurrency*: The concurrent operations on the index (e.g., insert, delete) may have conflicts, and it aims to process them in parallel while keeping transaction correctness (see Section 2.5).

(6) *bulk loading*: It builds the index for a batch of key-position pairs by utilizing the key distributions in the batch (see Section 2.6).

2.2 Lookup Design

Most of learned indexes use a hierarchical structure, where each node adopts a machine learning model to fit the key-to-position mapping (*data fitting model*). As shown in Figure 2 (a), in the lookup phase, the model of a leaf node predicts the position of the query key and the model of an internal node predicts its child node that contains the query key. Note that the complexity of the prediction is $O(1)$, which is better than binary search in a node. Next, if there exists an error in the predicted position, *position search* is used to re-search the key position.

2.2.1 *Data Fitting Model.*

It aims to fit the key-to-position mapping. Existing models can be broadly categorized into linear models [1, 7, 11, 24, 41, 43, 46] and non-linear models [18, 41, 43, 49]. The former is lightweight and most widely adopted, while the latter aims to achieve high prediction accuracy for a large node with many key-position pairs (which can reduce the index height).

(1) Linear Model. Most learned indexes adopt a linear model in each node, assuming linear relations between the keys and positions for leaf nodes (children IDs for internal nodes). There are two main types of linear models, i.e., the linear interpolation model and linear regression model. *The linear interpolation model* [1, 7, 41] extracts several pairs (e.g., the two endpoint pairs in a node and one median pair in MAB+tree [1]) and computes a piecewise linear model (e.g., a linear equation for every two adjacent pairs). *The linear regression model* [7, 24, 46] computes the minimal *sum of squared differences* of all the pairs in the node by adjusting the slope and intercept parameters (e.g., $y = 0.5x - 0.5$ for leaf node in Figure 2 (a)).

(2) Non-linear Model. There are two types of non-linear models, polynomial fitting models and neural network models. The former extends the linear regression model to polynomial models [49]. LIPP [49] extends the linear function as $F(x) = k \cdot G(x) + b$, where k is the slope, b is the intercept and $G(x)$ is any monotonically increasing function that helps to learn more complex key-to-position

mappings. The latter adopts neural-network models as the data fitting models, e.g., RMI [18], which are trained by gradient descent to minimize the prediction error.

(3) Hybrid Model. Some works adopt both linear and non-linear models in the learned indexes. For example, XIndex [43] adopts a two-layer hierarchical structure, where the first-layer uses an RMI model (as the first layer contains many pairs) and the second-layer nodes use piecewise linear models.

Linear vs Non-linear. First, non-linear models often achieve better prediction accuracy than linear models. However, the training and prediction of non-linear models often take more time. Second, non-linear models can support a larger number of keys than linear models, and thus each node can contain more keys and the tree depth can be smaller.

2.2.2 Position Search. Given a key, suppose its correct position is p. If a model predicts an incorrect position p', we need to re-search the true position p based on p'. There are three cases. (i) All model predictions are correct and it does not need to conduct re-search (e.g., LIPP [49]). (ii) The predictions of internal nodes are correct and the prediction of leaf nodes may be incorrect. Thus it only searches leaf nodes (e.g., RMI [18], ALEX [7]). (iii) The prediction of both leaf nodes and internal nodes may be incorrect. It needs to search both internal nodes and leaf nodes [1, 11, 24, 24, 41, 43, 46].

Position Search Methods. There are two steps in a position search. First, it needs to determine the search range. Given the sorted key-position pairs $D[0:N]$ and query key k, if the key of the predicted pair $D[p']$ is larger than k, the search range is $D[0:p']$; otherwise, the search range is $D[p'+1:N]$, where N is the number of pairs in D. Second, with the search range, its chooses proper search approaches. There are two types of methods. (i) Linear Search. If the error is relatively small, a linear search can perform well, since it can sequentially scan the pairs from the predicted position and quickly find the accurate position. (ii) Binary Search. If the error is large, it can use a binary search to find the key. Moreover, other variants of binary search, e.g., exponential search (searching positions of $1, 2, 4, \cdots$) and interpolation search (computing the probe position using interpolation on the two endpoints), are used to accelerate the binary search for some specific key distributions (e.g., interpolation search is better for uniform key distribution).

2.2.3 Supporting Range Query. Given a range query $[k_{left}, k_{right}]$, it first identifies the first key that is not smaller than k_{left}, and then scans the sorted pairs until reaching the key larger than k_{right}.

2.2.4 Supporting Duplicated Key. The difference between duplicated key and distinct key is that there may be multiple positions for a duplicated key. As a model predicts one position for each key, it requires to re-search positions even if the predicted position is correct, as it requires to find other positions for the key [1, 7, 11, 18, 41].

2.3 Insert Design

Given an insert pair (k, p), the learned index first finds the position of k, and then inserts the pair into the corresponding leaf node. If the number of pairs in the leaf node exceeds a threshold or the data fitting model is of low quality, the index has to perform structural modifications (e.g., splitting the node into two nodes and retrain the models) to keep high performance. Besides, since the structural modifications often take long time, learned indexes develop effective insert strategies to reduce the possibility of structural modification and model retraining.

2.3.1 Insert Strategies. There are two insert strategies. (i) The delta-buffer insert strategy keeps delta buffers, inserts a new pair into the buffers and periodically merges them into the existing index structure (Figure 2 (b)). (ii) The in-place insert strategy preserves some gaps in the leaf nodes and the new pairs can be directly inserted into the gaps (Figure 2 (c)).

(1) Delta-buffer Insert Strategy. There are three buffering granularities, including index-level, node-level, and pair-level. First, the index-level method (e.g., DPGM [11]) shares one buffer for all the insert operations. Second, the node-level method (e.g., XIndex [43], SIndex [46]) maintains a delta buffer for each leaf node to cache the inserted pairs. Third, the pair-level method (e.g., FINEdex [24]) has finer-grained delta buffers, and allocates a buffer for each key in the leaf node. Compared with node-level buffer, pair-level buffer achieves higher concurrency performance, but may incur extra storage overhead.

(2) In-place Insert Strategy. To accommodate an insert, we can reserve some gaps for each node (e.g., inserting one key occupies two positions). For each insert, if there exists a gap in the target position, the pair can be directly inserted; otherwise, there are two strategies to resolve the conflict. First, ALEX [7] shifts the pairs between the target position and the closest gap to make space for the new pair. (Note that if the insert cost is too large, it may trigger structural modification). Second, LIPP [49] creates a new node with both the inserted pair and the existing pair in the target position, and replaces the target position with a pointer to the new node.

2.3.2 Structural Modification. If a node cannot accommodate the inserted pairs, it needs to update the index structure (e.g., splitting the node). There are four cases to trigger the update. (i) The fullness-based method triggers the structural modification if the number of pairs in the node or buffer exceeds a threshold. (ii) The error-based method triggers the structural update if the model prediction error exceeds a threshold. (iii) The cost-based method uses a cost model to trigger the structural update, which estimates the cost value of an index node based on both the lookup costs (position searches) and insert costs (position searches and pair shifts). It updates the local structures if the cost of a node exceeds a threshold. (iv) The conflict-based method records the number of pairs that are mapped to the same position, and triggers the structural update if the number of conflict pairs in the subtree rooted at the node exceeds a threshold.

(1) *Fullness-based Structural Modification.* It can be further divided into three classes. First, for the *delta-buffer insert indexes* [11, 24, 43, 46], if the buffer has no space left, they should merge the buffer with the index node and retrain the model. Second, if the number of pairs in the node exceeds a threshold, it will split the node. For instance, MAB+tree [1] splits the target index node into two nodes, which become the child nodes of the target node's parent. Third, if the number of pairs in the subtree rooted at the node exceeds a threshold, LIPP will adopt the bulk loading algorithm to rebuild the subtree with all of pairs under this node (see Section 2.6).

(2) *Error-based Structural Modification.* If the prediction error of a node exceeds a threshold, XIndex will split the node. The keys in the node are evenly divided into two child nodes, and each child node trains a piecewise linear model with linear regression.

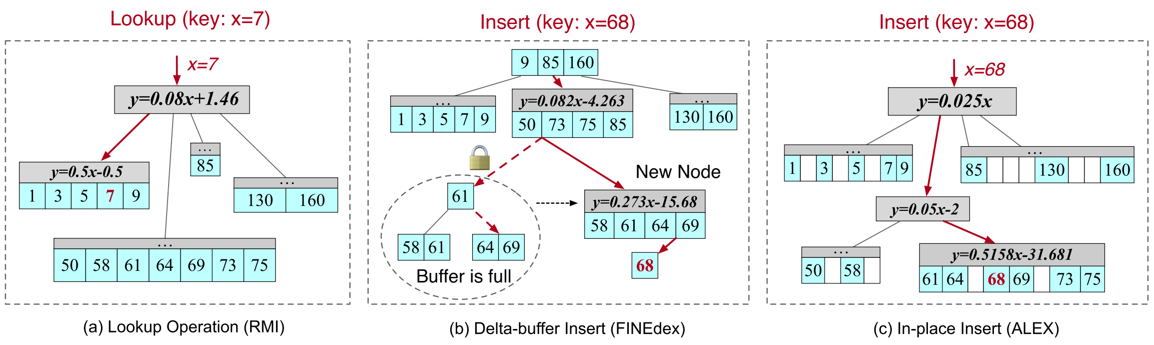
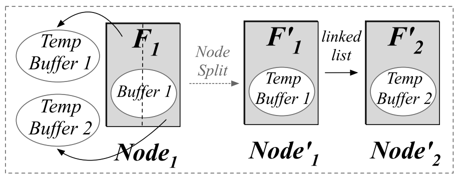
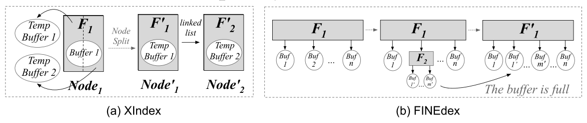


Figure 2: Examples of Lookup and Insert on Learned Indexes.



(a) Temporary Buffer (XIndex).



(b) Buffer-Train-Merge (FINEdex).

Figure 3: Concurrency in Learned Indexes.

(3) *Cost-based Structural Modification.* ALEX proposes a cost model to estimate the average latency of lookups/inserts for each leaf node. If the cost of a leaf node exceeds a threshold, ALEX adopts three strategies to modify the index structure. (*i*) Expand the node. It first doubles the capability of the node, scales the linear model (e.g., doubling the slope and the intercept), and maps the keys to a wider range of positions. The pairs are re-arranged to their predicted positions, and more gaps are preserved among them. (*ii*) Split the node. Given a leaf node, it splits the keys into two nodes, trains models for the two nodes, and replaces the leaf node with the two nodes, which are rooted at the same parent node. (*iii*) Rebuild the node. It adopts the bulk loading algorithm to rebuild the subtree for pairs under the node, and then replaces the original node. ALEX selects the best one from the three strategies with the lowest cost.

(4) *Conflict-based Structural Modification.* Given a node, if two pairs are mapped to the same position by the model, the node and all its parents record the conflict. If the number of conflicts for a node exceeds a threshold, LIPP will adopt the bulk loading algorithm to rebuild the subtree of this node.

Note that each node contains a small number of pairs, thus retraining the models of a few nodes is acceptable.

2.4 Delete Design

Delete is similar to the insert operation. For delete, the index first finds the position and removes it from the node. There are also trigger conditions to determine when to modify the structure. For example, in XIndex [43], if there are too few pairs in the buffer and too small model prediction error, two consecutive nodes will be merged into one node and the corresponding model is retrained.

2.5 Concurrency Design

When a thread inserts/deletes a pair, it locks the involved object (e.g., key-position pair or node). Other threads requiring to access that object are blocked until the lock is released. There are two concurrency design granularities. (*i*) Intra-node concurrency. It will not modify the index structure, and the updates across different key-position pairs can be concurrent with their pair-level buffers. However, updates on the same pair cannot be concurrent. As shown in Figure 2 (b), when FINEdex inserts a pair (*key: x=68*), it first locks and then inserts into the buffer. (*ii*) Inter-node concurrency. It needs to modify the index structures. There are two strategies. *(a) Temporary-Buffer.* It locks the node and its node buffers when splitting the node into two nodes, e.g., XIndex [43]. Then it creates two temporary buffers for the two split nodes (Figure 3 (a)). New pairs from other concurrent threads are inserted into the temporary buffers during splitting the node. After the split, the temporary buffers become the delta buffers of the new nodes. *(b) Buffer-Train-Merge.* It performs two types of structural modifications, e.g., FINEdex [24] in Figure 2 (b) and Figure 3 (b). First, when a pair-level buffer is full (e.g., with 4 pairs), it trains a sub-node (*y=0.273x-15.68*) with the buffered pairs. During this procedure, updates across other pair-level buffers can be concurrent. Second, when a buffer of a sub-node is full, it merges the sub-nodes with the node to reduce the height. It locks the node's model, and retrains a new model with the pairs in the node and sub-nodes. During retraining, updates across the buffers of the node and sub-nodes can be concurrent. After retraining, the pair-level buffers of the original node and sub-nodes become buffers of the new node.

2.6 Bulk Loading Design

Bulk loading aims to construct the learned index structure for a batch of key-position pairs, which can utilize the data distribution to effectively build the index. Most of learned indexes adopt a tree structure and there are two bulk-loading strategies.

Top-down Bulk Loading. It first takes all the pairs as the root node. It then decides whether to split the pairs to generate multiple child nodes. If it decides not to split, this node will be a leaf node and it trains a model to predict the key-to-position mapping in this leaf node; otherwise, it trains a key-to-child model in this internal node, and splits the pairs to multiple child nodes in accordance with the model prediction. Next it recursively processes the child nodes. There are two challenges in this framework. The first is whether to

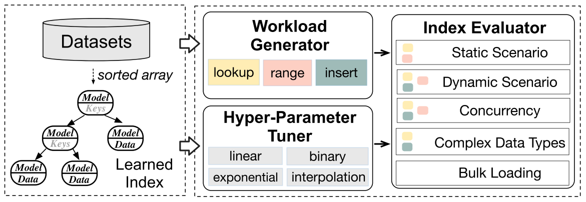


Figure 4: The Evaluation Testbed.

split a node. The second is to decide the number of split child nodes and how to split the pairs into different child nodes (discuss later).

Bottom-up Bulk Loading. It first splits the pairs into different leaf nodes and trains a model to predict the positions for each key in each leaf node. Then it extracts the minimal or maximal keys in each leaf node and generates a subset of pairs with those selected keys. Next it decides whether to further split the subset of those selected pairs. If it decides not to split, it constructs a root node with those pairs and trains a key-to-child model for that node, which predicts the child that a key belongs to; otherwise, it splits these pairs to different nodes, trains a key-to-child model for each split node, and extracts the minimal or maximal keys for each split node. Next it iteratively evaluates those selected keys and builds the tree structure. This method also needs to address the above challenges.

Decide Whether To Split A Node. There are several strategies to decide whether to split a node. The first is to manually decide. For example, the learned indexes decide the number of levels. Then the nodes except the last level will be split. The second is to use some conditions (e.g., prediction accuracy) to decide. For instance, it evaluates the prediction accuracy of this node. If the prediction error is larger than a threshold, then it splits the nodes; not otherwise.

Decide The Number of Split Nodes And How To Assign Pairs Into Split Nodes. There are several strategies to split the *sorted pairs* into multiple nodes.

(1) Greedy Split [1, 11, 24, 41, 46]. It initializes a node with the first pair. Then it incrementally checks whether to add the next pair into that node. Assume it adds that pair into the node and trains a model for the pairs in that node. If the prediction error of the trained model for the pairs in that node exceeds a threshold [11, 41], it will not add that pair into that node and takes it as the next node; otherwise it adds that pair into that node. However it is expensive to add the pairs one by one, and thus some methods [24, 46] add the pairs in a batch (e.g., adding 1K pairs each time) and train a model until the prediction error is larger than a threshold. Then they remove some pairs backward until the prediction error is not larger than a threshold.

(2) Even Split [43]. It first sets a number τ of split nodes, e.g., 2, and then it evenly splits the pairs into τ sub-nodes and trains the models. If the prediction error of any sub-node exceeds a threshold, it increases the number (e.g., to 2τ) and repeats the above steps; otherwise it terminates the split.

(3) Cost-based Split [7]. It first trains a model with the pairs. Then it exponentially splits the pairs to different numbers (e.g., 2, 4, 8, $\cdots$), and the key range of the pairs is evenly split to the child nodes. It adopts a cost model to estimate the average latency of lookups/inserts for the index, and selects the best split number. Then it adopts an auxiliary full binary tree, and maps each leaf node to a split node. Next it wants to merge some nodes in a bottom-up manner in order to get a concise structure. If the cost becomes smaller by merging two siblings, it merges them; otherwise, it skips them. It repeats the process until no siblings can be merged.

(4) Conflict-based Split [49]. It first trains a model with the pairs. For the pairs, if more than two pairs are mapped to a position, they will be assigned to a child node and the pointer to the child node will be stored in that position; otherwise, the pair is mapped to a distinct position, and will be directly stored in that position.

Bottom-up vs Top-down. The bottom-up methods have prediction errors in internal nodes, while top-down methods do not. Because for the internal node, the bottom-up method extracts the minimal or maximal key from the child node and trains a key-to-child model in it. The model may not fit the mapping perfectly and have prediction errors. On the other hand, the top-down method first trains a key-to-child model with the pairs in the internal node, and splits the pairs to multiple child nodes in accordance with model prediction (i.e., the key must belong to its predicted child node). Thus the top-down method may have no prediction error.

Note that the top-down method adopts the monotonic model in the internal node [7, 49]. If the internal node adopts the non-monotonic model for the sorted list of keys, the mapped children IDs of them may not keep sorted. Then two problems arise. (i) Distant keys in the sorted list are mapped to the same node. It is difficult to train a model to fit those distant keys, which may produce larger prediction error. (ii) The prediction for a non-existing key can be far from the position of its closest existing key, which causes extra position search overhead.

Other Methods. RMI [18] uses a non-tree structure (two layers) for bulk loading. It first takes all pairs as the root, and the number of leaf nodes is a hyper-parameter. For the root node, it first builds a model to predict the position of each key, and it can obtain the corresponding leaf node based on the position and the number of leaf nodes. For example, suppose the predicted position p', the number of pairs N and the number of leaf nodes M, the corresponding leaf node of this key position will be the i-th node, where $i = \lfloor \frac{Mp'}{N} \rfloor$. RMI then splits the root into the leaf nodes using the model predictions. Next it trains a model by stochastic gradient descent for each leaf node that predicts the position of each key in the leaf node.

3 A TESTBED FOR LEARNED INDEXES

We design a testbed that can be used for (1) practitioners to select a learned index for their applications and (2) researchers to design new learned indexes with less development overhead.

3.1 Testbed Architecture

Figure 4 shows the architecture of our testbed, which is composed of three main reusable modules, i.e., *workload generator*, *hyper-parameter tuner*, and *index evaluator*.

Workload Generator. It generates different workloads for static/-dynamic/concurrency scenarios (see Section 3.2).

Hyper-Parameter Tuner. It explores the most suitable hyper-parameter settings for each learned index to ensure fair evaluation. First, we support five main search methods, i.e., linear search, SIMD accelerated linear search, binary search, exponential search, interpolation search. We also provide APIs similar to the *standard template library* (STL) API: (i) the *lower_bound* function returns the first location in the list where the key is not smaller than query key k_q

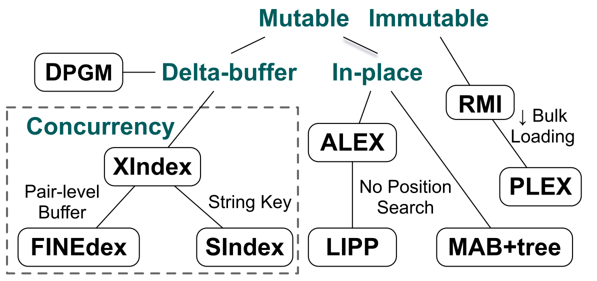


Figure 5: Lineage of Learned Indexes.

and (ii) the *upper_bound* function returns the first location where the key is larger than k_q. The API is easily compatible with existing (and potentially future) indexes. Second, for other hyper-parameters (e.g., prediction error threshold), we either utilize existing tuning tools (e.g., RMI uses the automated tuning framework CDFShop to tune the hyper-parameters [31]) or tune major hyper-parameters discussed in their papers to report the best performance. Specifically, we conduct grid search for the hyper-parameters and try out all the five search methods. Then, we run sample workloads (e.g., 1 million insert/lookup only) with every hyper-parameter setting from the grid search, and record the throughput. For the workload with the read ratio of r, we can estimate the throughput as $r\frac{lookup\ throughput}{max\ lookup\ throughput} + (1-r)\frac{insert\ throughput}{max\ insert\ throughput}$. For any workload, we sort the throughput of the hyper-parameter samples in a descending order, and select the best hyper-parameter setting to run our experiments.

Index Evaluator. It evaluates the throughput/latency/index size of learned indexes by varying different factors, e.g., static or dynamic scenarios, concurrent operations, and bulk loading.

Workflow. To evaluate a learned index, we first prepare the test datasets from real scenarios. Then the *workload generator* generates test workloads. Next, the *hyper-parameter tuner* selects and tunes the hyper-parameters of the index. Finally, the *index evaluator* executes the workloads on the index, evaluates the performance during workload execution, and provides empirical analysis based on the evaluation results.

3.2 Workload Generator

To effectively evaluate different index designs, we separately provide workloads in *static scenario* (with lookup/range operations), *dynamic scenario* (with lookup/range/insert operations), and *concurrency scenario* (with parallel lookup/range/insert operations).

Static Scenario only involves lookups and range queries. We control the proportions of existing keys and non-existing keys. Assume the keys in the dataset are within $[k_{min}, k_{max}]$, we separately extract existing and non-existing keys from $[k_{min}, k_{max}]$ with uniformly random sampling. For non-existing keys, we repeatedly sample until the sampled key does not exist. In our experiments, the proportions of (non)-existing keys are set 50%. Besides, to generate the range query of keys in $[k_{left}, k_{right}]$, we first uniformly and randomly sample k_{left} from $[k_{min}, k_{max}]$. Then we choose k_{right} to ensure the number of target keys in the range is fewer than 100 (too many keys will significantly increase the scan overhead).

Dynamic Scenario. Compared with static scenarios, there are inserts in dynamic scenarios. To make our evaluation more practical, we design three distribution patterns of inserted pairs: (i) the *uniform* mode where the keys of inserted pairs are evenly and randomly sampled from the dataset; (ii) the *delta* mode where the keys of inserted pairs are larger than those of bulk-loaded pairs, and the inserted pairs are sequentially appended by the key values; (iii) the *hotspot* mode where the keys are evenly and randomly sampled from a small part of the dataset D. The sample range divided by the range of D is defined as *hotspot ratio*. The smaller *hotspot ratio*, the more skewed the inserted keys distribute. Furthermore, to better test the index factors (e.g., structural modification), we also provide a parameter *mix*. If *mix* is set $false$, different types of operations are separately executed in batch (e.g., first execute inserts and then execute lookups); otherwise, they are mixed together to execute.

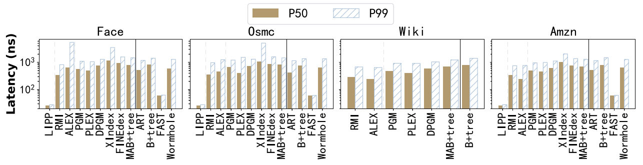


Figure 6: P50 and P99 lookup latency. The dotted and solid lines divide the chart into (i) learned indexes without position search; (ii) learned indexes that only search at leaf nodes; (iii) learned indexes that search at both internal and leaf nodes; (iv) traditional indexes (listed from left to right). Note some indexes cannot support duplicated keys on `Wiki`.

Concurrency Scenario involves concurrent lookups and inserts, and its design is similar to that of dynamic scenarios.

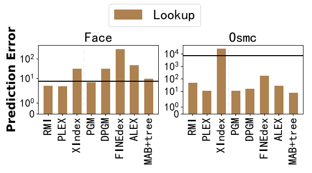
4 EXPERIMENTS

In this section, we first discuss the experimental setup (see Section 4.1), and then evaluate the learned indexes in static scenarios (see Section 4.2), dynamic scenarios (see Section 4.3), hybrid lookup/range/insert scenarios (see Section 4.4), concurrency scenarios (see Section 4.5), bulk loading (see Section 4.6). Next, we evaluate string keys (see Section 4.7) [1]. Finally, we provide guidance for learned index selection (see Section 4.8).

4.1 Experimental Setup

Datasets. We test the learned indexes on five real datasets with different cumulative distribution function (CDF) curves, which can well reflect different distributions of keys [15, 22]. `Amzn` is a collection of Amazon book IDs with their sale popularity, which includes 200M non-duplicated integer keys. `Face` is a collection of Facebook user IDs, which includes 200M non-duplicated integer keys. `Wiki` is a collection of wikipedia article IDs with their edit timestamps, which includes 200M pairs, and at most 21,026 duplicated integer keys. `Osmc` is a collection of cell IDs from OpenStreetMap, which includes 200M non-duplicated integer keys. Since the keys in `Osmc` are the projections of the high dimensional locations, the CDF of `Osmc` is much more complicated than other datasets [15]. `Url` is a collection of URLs from Memetracker, which includes 90M non-duplicated string keys. The string keys in `Url` are of variable lengths (15∼128 bytes). As the keys need to be converted into high-dimensional integers, it takes more processing time in learned indexes [46]. *Note that some learned indexes do not support strings, and thus we use it to evaluate those supporting strings.*

[1]Some supplementary experiments at *github.com/curtis-sun/TLI/tree/main/report*


(a) Model prediction error.

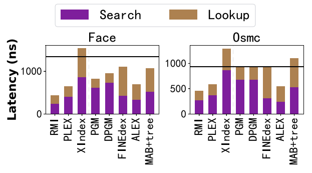

(b) Ratio of position search latency in lookup latency.

Figure 7: Model prediction and position search in lookup. The horizontal line represents (i) the average distance between the start and end positions in searches of B+tree (a); (ii) the lookup latency of B+tree (b). The vertical line divides learned indexes by data fitting model, i.e., the left adopt non-linear/hybrid models, and the right adopt linear models.

Evaluated Indexes. We evaluate ten learned indexes and four traditional indexes. First, we explain the techniques of learned indexes in Section 2 and the relations of the selected learned indexes are shown in Figure 5. Second, we select four representative traditional indexes, `B+tree`, `FAST`, `ART`, `Wormhole` [3, 14, 20, 52]. `B+tree` [3] is a widely used disk-based index. `FAST` [14] is a variant of binary search tree, which is well optimized for the hardware characteristics, e.g., SIMD, cache line. `ART` [20, 21] is a variant of radix tree, which is a well-optimized in-memory index. `Wormhole` [52] is a hybrid structure of B+tree, radix tree and hash table, which supports variable-length keys and optimizes concurrency.

Performance Metrics. We evaluate latency, throughput, and index size. Note that some work only measures the space of internal nodes as the index size [18, 30]. However, for secondary indexes, the leaf nodes must be included in the index. Thus, in our evaluation, the index size includes both internal nodes and leaf nodes.

Experimental Setting. We conduct all experiments on a Linux server with 128GB RAM and two 10-core Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz.

4.2 Static Scenario Evaluation

In this section, we evaluate the lookup and range query performances in static scenarios, e.g., only point and range queries without inserts and concurrency.

4.2.1 Lookup Evaluation. We first evaluate the performances for point queries. For each dataset, we initialize the indexes with 200M keys in total, and evaluate the indexes using 20M lookups as the workload, which are generated as described in Section 3.2.

Lookup Performance. We evaluate the lookup performance from P50 and P99 lookup latency, and Figure 6 shows the results. First, for each dataset, there is at least one learned index that outperforms traditional indexes; but there is no learned index that outperforms traditional indexes on all datasets. On `Face`/`Osmc`/`Amzn` datasets, LIPP performs best because it does not conduct position searches; while on `Wiki` dataset, ALEX performs best because ALEX only conducts position searches at leaf nodes (LIPP does not support duplicated keys in wiki data). Second, P99 latency is close to P50 latency, indicating lookups of most keys have similar latency.

Position Search. From Figure 6, we find the lookup performance is affected by the position searches. (i) Those with no search have best lookup performances. (ii) Those that only search at leaf nodes achieve middle performances. (iii) Those with searches at both internal and leaf nodes have the worst performances (except for concurrent learned indexes XIndex and FINEdex). Because fewer position searches require less overhead in the phase of lookups.

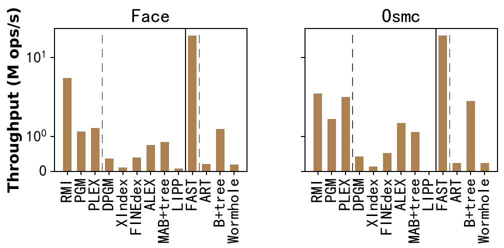


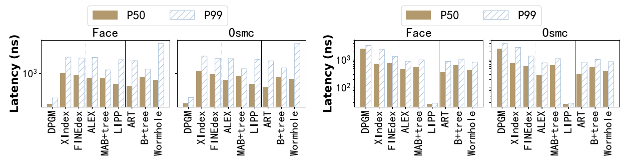
Figure 8: Throughput of range queries. The solid line divides the chart into learned indexes (left) and traditional indexes (right), for which the dotted lines separately divide into immutable (left) and mutable indexes (right).

Data Fitting Model. We evaluate the position search latency and the prediction error. As shown in Figure 7, we have two observations. First, in most cases, the prediction error of learned indexes is smaller than the search distance of B+tree, which verifies that data fitting model can reduce the search distance by properly fitting the key-to-position mapping. Second, in terms of model design, we find that indexes using non-linear/hybrid models (except XIndex) have smaller prediction errors than those using linear models on `Face` dataset, with which learned indexes can reduce position searches and gain lower lookup latency. Differently, all indexes (except XIndex) have similar prediction errors on `Osmc` dataset, for which ALEX using linear models can gain similar lookup latency with those using non-linear models. That is because the key-position distribution of `Osmc` data is the most complicated, and all models cannot fit well.

Finding 1. *For lookup-only scenarios, learned indexes LIPP and ALEX perform the best on the four datasets. In terms of position search, learned indexes without search and those which only search at leaf nodes perform better. For model design, non-linear models have lower prediction errors than linear models on simple datasets (e.g., relatively smooth CDF without abrupt shifts), and thus achieve better lookup performances.*

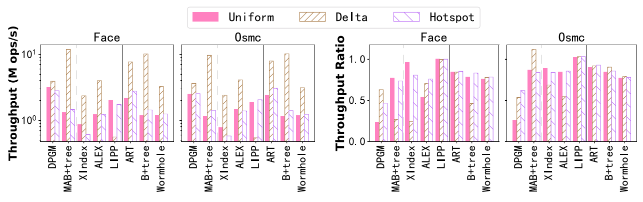
4.2.2 Range Query Evaluation. Figure 8 shows the performances of range queries. For each dataset, we initialize the indexes with 200M keys in total, and evaluate the indexes using 20M range queries as the workload, which are generated as described in Section 3.2.

We have three observations. First, immutable indexes mostly perform better than mutable indexes, because immutable indexes store the original list of key-position pairs together outside the index, and can directly scan the pair list for range queries. Instead, mutable indexes split the original data into different index nodes, which limits the scan throughput. Second, among the mutable indexes, the traditional index B+tree and learned indexes ALEX, MAB+tree perform best on both datasets. Because they all link a sorted list among leaf nodes. So during the scan of range query, they can scan the pairs along the list and reduce the data access overhead. Third, LIPP has the worst range-query throughput. Although LIPP achieves best lookup performance, the pairs of LIPP are scattered in both the internal and leaf nodes, and LIPP has no linked list among them. As a result, during the scan of range query, it takes a long time for LIPP to find the target consecutive pairs in different nodes.



(a) P50 and P99 insert latency. (b) P50 and P99 lookup latency after inserts.

Figure 9: Insert and lookup latency. The solid line divides the indexes into learned (left) and traditional indexes (right). The dotted line divides learned indexes in terms of the insert strategy: the left is delta-buffer, and the right is in-place.



(a) Insert throughput. (b) Lookup throughput ratio of after insert to before insert.

Figure 10: Performance of different insert patterns. The solid line divides into learned (left) and traditional (right) indexes. The dotted line divides learned indexes by structural modification, i.e., fullness-based (left), and others (right).

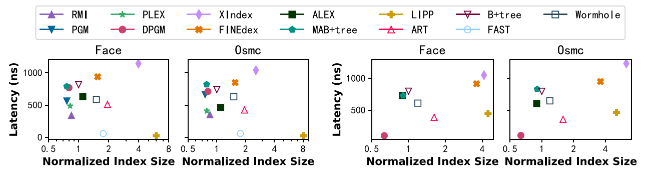
Finding 2. *For range queries only, learned indexes do not outperform traditional indexes. Indexes that maintain a sorted list of key-position pairs can efficiently process range queries while others cannot as they cannot sequentially obtain the pairs.*

4.3 Dynamic Scenario Evaluation

Figure 9 (a) shows the P50 and P99 insert latency, and Figure 9 (b) shows those of lookup latency after insert. The P50 latency represents the insert/lookup performance. In terms of the insert strategy, we have two observations. First, in-place insert provides better lookup performances than delta-buffer insert. For instance, the delta-buffer index DPGM performs worst lookup performances on both datasets. Because DPGM adopts the buffer and multiple PGMs (similar to LSM-tree [35]), which require lots of position searches during lookups. Second, we also find that DPGM performs best insert performances on both datasets, because it adopts one shared delta buffer and can directly insert into the buffer.

Structural Modification Overhead. From Figure 9 (a), we have two observations. First, most indexes have similar P99 insert latency, which reflects that they have similar structural modification overhead. Second, DPGM has the lowest P99 insert latency. That is because DPGM adopts the delta-buffer insert strategy and the logarithmic merge method [35], so that most of its structural modifications only involve a small number of key-position pairs and consume low overhead.

Insert Factor. Figure 12 shows the insert throughput under varying insert factor, which is defined as the inserted data size divided by the bulk-loaded data size. To fairly compare the results, we ensure that the bulk-loaded data and inserted data cover all the original data pairs. We observe the learned index DPGM performs best in



(a) Trade-off of lookup latency and normalized index size. (b) Trade-off of insert latency and normalized index size.

Figure 11: Trade-off of performance and normalized index size (i.e., index size divided by that of B+tree).

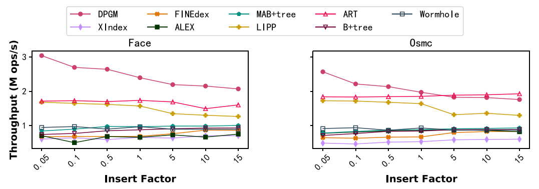


Figure 12: Insert throughput varying with insert factor (inserted data size divided by bulk-loaded data size).

most cases. However, the traditional index ART performs best on Osmc dataset when the insert factor is large. There are two reasons. (i) Osmc data has the most complicated data distribution, which often causes inaccurate model predictions of learned indexes and negatively affects the insert throughput. (ii) For large insert factors, learned indexes trigger more structural modifications during inserts, which further reduces the insert throughput.

Insert Pattern. Figure 10 shows throughput under three insert patterns, i.e., uniform, delta and hotspot. We evaluate the insert throughput in Figure 10 (a). For the uniform pattern, we find that DPGM performs best on all datasets. For the delta pattern, we find that MAB+tree and B+tree perform best on all datasets. That is because in the insert phase, the new key is always larger than all keys in the index. For MAB+tree and B+tree, the new key can be directly inserted into the end of the last leaf node, which avoids the overhead to shift existing keys. When the inserted data follows the hotspot pattern, we find that DPGM performs best on Face dataset, while the traditional index ART performs best on Osmc. There are two reasons. (i) Osmc data has more complicated CDF, which reduces the insert throughput of learned indexes. (ii) Since many pairs are inserted into a small region, learned indexes trigger more structural modifications during inserts, which further reduces the insert throughput.

Next, we evaluate the lookup throughput change before/after inserts with 1.0 insert factor in Figure 10 (b). We have two observations. First, the lookup throughput ratios of most indexes are larger than 0.5, which reflects that their structural modifications perform well to improve the lookup throughput. Second, indexes using cost-based and conflict-based modifications are more robust than those using fullness-based modifications. We explain the performances of DPGM and MAB+tree respectively. (i) For uniform pattern, we observe that DPGM's lookup throughput decreases most greatly on both datasets. That is because whenever the inserted keys accumulate to a particular number, DPGM will rebuild a PGM index. Since the inserted keys of uniform pattern are randomly distributed in a large range, PGM index's linear models have difficulty in fitting the key-to-position mapping and require a higher height, which reduces the lookup throughput. (ii) For delta pattern, MAB+tree's

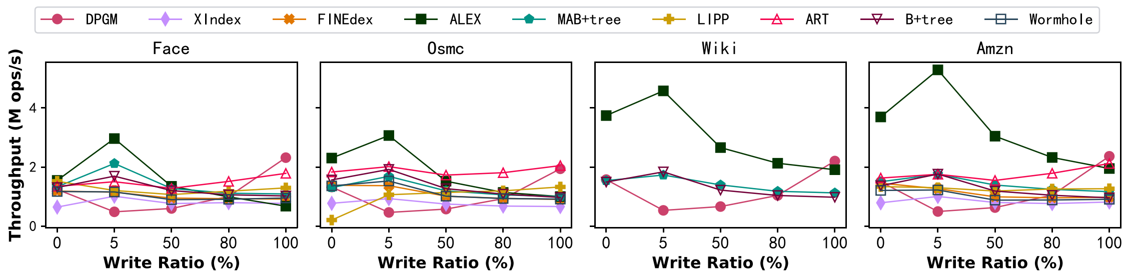


Figure 13: Throughput of different read-write ratios. Some indexes cannot support duplicated keys on Wiki.

lookup throughput decreases extremely greatly on `Face` dataset, because MAB+tree's models are not updated in time, which increases the prediction error and reduces the lookup throughput.

Trade-off between Performance and Index Size. Figure 11 shows the trade-off between the insert/lookup performance and index size. We have three observations. First, indexes often use additional structures (e.g., ART adopts hash tables in some nodes) to reduce insert/lookup latency, and thus involve large index sizes. Second, we also find that some learned indexes like XIndex and FINEdex gain both large index size and high insert/lookup latency, because they use extra space (e.g., pair-level buffers) to support concurrent operations, which may slow down insert/lookup operations as the search involves both the index and buffers. Third, most indexes have similar index sizes (within 0.5 ~2× that of B+tree). Because the index size is mainly affected by the original data size (except for structures like Bloom filters that only involve keys).

Finding 3. *For uniform insert pattern, learned indexes gain high insert throughput; while for skewed pattern or many inserts, learned indexes have no advantage over traditional indexes. In terms of insert strategies, the learned indexes using in-place inserts have higher lookup performances, while the indexes using delta-buffer inserts can achieve highest insert performance. For structural modification, most indexes (except DPGM) have similar overhead, and those using fullness-based modification perform less robustly. Indexes often use additional structures to reduce insert/lookup latency, and thus involve large index sizes.*

4.4 Hybrid Lookup/Range/Insert Evaluation

Figure 13 shows the workload throughput with different read-write ratios. The read operations include lookups and range queries (with a ratio of 19:1). For the read-only workload, we first initialize the index with 105M random dataset samples, and then execute 100M lookups. For the other four workloads, we first initialize the index with 5M random samples, and then execute 100M operations.

For read-only (0% write) and read-heavy workloads (5% write), ALEX performs best on all datasets. There are two reasons. (*i*) ALEX searches only at leaf nodes, and decreases the position search overhead in inserts and lookups. (*ii*) ALEX links the leaf nodes in a sorted sequence to facilitate range queries. Note in lookup evaluation (Section 4.2.1), LIPP performs best on `Face`, `Osmc` and `Amzn` datasets, because, for range queries, LIPP needs to search from difference leaf nodes (not linked together). For balanced workloads (50% write), the traditional index ART performs best on `Osmc` dataset, while ALEX performs best on `Face`, `Wiki` and `Amzn`. For write-heavy workloads (80% write), ART performs best on `Face` and `Osmc` datasets, while ALEX performs best on `Wiki` and `Amzn`. There are two reasons. (*i*) Compared with ALEX, ART involves fewer nodes (no more than 3 in most cases) in structural modifications, and gains higher insert throughput. (*ii*) DPGM adopts a delta buffer and multiple PGMs (similar to LSM-tree [35]), which requires lots of position searches during lookups. For write-only workloads (100% write), DPGM performs best on most datasets, which is similar to the experimental results in Section 4.3.

Furthermore, to analyze the experimental results in finer granularity, we break down the execution cycles (∝ latency) into underlying metrics in the Intel's Top-down Micro-architecture Analysis Method (TMAM), including retiring (cycles used to retire instructions, mainly affected by the instruction count), bad speculation (stalls from branch-instruction mispredictions), frontend bound (stalls from instruction fetching/encoding), DRAM bound (stalls from cache misses), and other minor metrics [39, 55]. From Figure 14, we have four observations. (*i*) For read-only workloads, the metrics that consume most time are DRAM bound (30.0%), retiring (24.5%), frontend bound (14.0%) and bad speculation (9.4%). These metrics are closely related to lookup/range operations. For write-only workloads, DRAM bound (30.7%), retiring (21.6%) occupy most part of the latency, and thus are the bottlenecks of insert operations. (*ii*) For lookup operations, learned indexes use model predictions to reduce position searches, and thus reduce cache misses (DRAM bound). Besides, some learned indexes further improve the performance by avoiding position searches (e.g., LIPP) or searching only at leaf nodes (e.g., ALEX), which eases branch-instruction prediction (bad speculation) and avoids extra instruction re-fetching between model prediction and position search (frontend bound). (*iii*) For range operations, some indexes link the leaf nodes in a sorted list (e.g., ALEX) and allow direct list scanning, which results in fewer instructions (retiring) and cache misses (DRAM bound). (*iv*) For insert operations, some learned indexes utilize shared delta buffers (e.g., DPGM) to reduce cache misses (DRAM bound). Besides, for structural modifications during inserts, learned indexes often involve more tree nodes than traditional indexes (e.g., ART) and require to retrain the data fitting models in these nodes, which increases instructions (retiring) and cache misses (DRAM bound).

Finding 4. *Learned indexes are efficient for read-heavy workloads, but traditional indexes better balance the throughput of lookups/ranges/inserts in write-heavy workloads. Besides, micro-architectural metrics like retiring (instruction count), bad speculation (branch-instruction misprediction), frontend bound (instruction fetching/encoding), DRAM bound (cache miss) can indicate the read/write performances of learned indexes (e.g., reducing branch-instruction misprediction by searching only at the leaf nodes).*

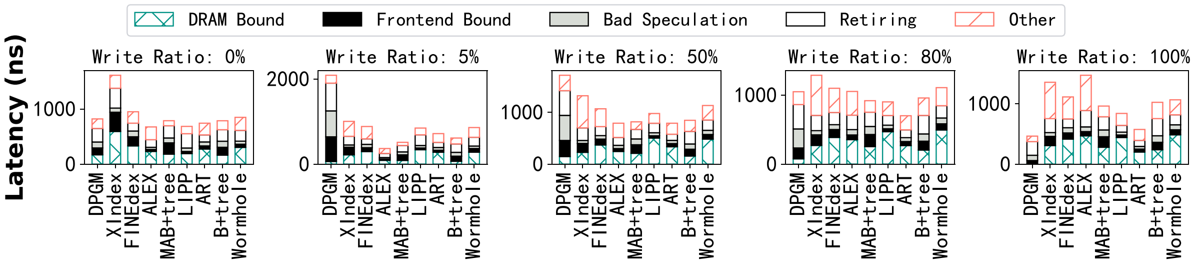


Figure 14: Average latency of different workloads on Face (broken down by micro-architectural metrics).

Table 2: Bulk Loading Time on Amzn Dataset.

Type	Learned (Greedy)					Learned (Not Greedy)				Traditional			
Index	MAB+tree	PLEX	PGM	DPGM	FINEdex	XIndex	LIPP	RMI	ALEX	B+tree	FAST	ART	Wormhole
Time (s)	2.258	6.634	11.233	12.476	18.336	25.162	25.463	29.378	77.867	4.717	5.788	25.094	57.535

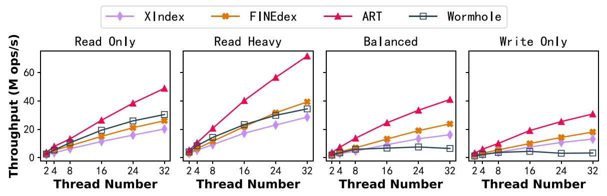


Figure 15: Throughput of Concurrency on Face Dataset.

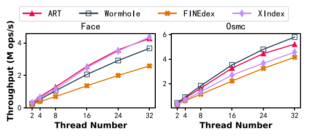


Figure 16: Throughput of Concurrent Range Workloads (95% range + 5% insert).

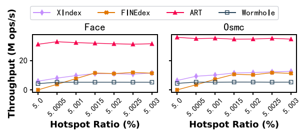


Figure 17: Concurrent insert throughput varying with hotspot ratio.

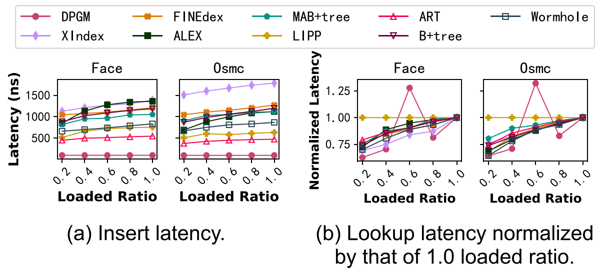


Figure 18: Evaluation on Block-wise Loading.

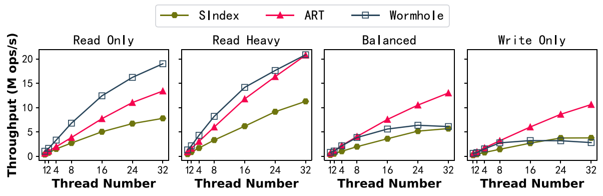


Figure 19: Evaluation on String Keys.

4.5 Concurrency Scenario Evaluation

We evaluate the four indexes that support concurrency scenarios, and Figure 15 shows the throughput of four workloads with different ratios of lookups and inserts (i.e., read only (0% insert), read heavy (5% insert), balanced (50% insert) and write only (100% insert)). We observe that learned indexes do not outperform traditional indexes in all cases. There are three reasons. (i) The learned indexes adopt data-buffer insert, which may slow down lookups as the search involves both the index and buffers. (ii) For structural modifications during inserts, learned indexes should retrain models, which decreases the insert throughput. Besides, structural modifications should lock involved nodes, and more thread collisions occur. (iii) The learned index FINEdex does not adopt advanced lock mechanisms (e.g., optimistic lock coupling [21]), which negatively affects its insert/lookup throughput.

Range Query Evaluation. Figure 16 shows the throughput of range workloads (95% range + 5% insert). We find the traditional index Wormhole performs best on Osmc dataset, while XIndex and the traditional index ART perform best on Face dataset. Because (*i*) Osmc data has the most complicated data distribution, which causes inaccurate model predictions of XIndex and negatively affects the insert/range throughput. (*ii*) XIndex links its leaf nodes and buffers in a sorted list, which facilitates range queries.

Skewed Insert. Figure 17 shows the insert throughput under different hotspot ratios (smaller hotspot ratio indicates more skewed inserted data). We find learned indexes cannot outperform traditional indexes, particularly for small hotspot ratios. There are two reasons. First, the inserted keys are within a small range (several buffers) and cause frequent thread collisions. Second, the skewed inserted data quickly takes up the buffers and triggers frequent structural modifications.

Finding 5. *Learned indexes cannot outperform traditional indexes for concurrent lookups/writes, which need to* (*i*) *search both the index and delta buffers and* (*ii*) *retrain models during structural modification. However, learned indexes and traditional indexes achieve similar concurrency performance for range queries, which could incur thread collisions and are hard to optimize.*

4.6 Bulk Loading Evaluation

Since the bulk loading latencies are similar among datasets, Table 2 showcase the bulk loading time for Amzn dataset. For each index, we choose the hyper-parameter with the best lookup throughput. We have three observations. First, learned indexes do not outperform traditional indexes in bulk loading time. Because during the bulk loading, learned indexes train the models and split the key-position pairs into nodes according to the model prediction, which takes much time. Second, learned indexes using greedy split have lower bulk loading time than others. That is because greedy split has the smallest time complexity. On contrary, ALEX using cost-based split has the longest bulk loading time, since it iterates many times per node to determine the index structure. Third, recall the lookup evaluation results (see Section 4.2.1), learned indexes that do not using greedy split have higher lookup throughput than others. Because greedy split is relatively simple. Instead, more effective index structure will be built if it considers the cost and the pair conflict for the nodes.

Block-wise Loading. Figure 18 shows the insert and lookup latency during inserting pairs block by block. The loaded ratio is the loaded pair number divided by the dataset size. We have three observations. (*i*) The insert/lookup latency of most indexes increases slowly during block-wise loading, because their structural modification methods can adapt to the increasing data size (e.g., maintaining low prediction error and index height). (*ii*) As the loaded ratio increases, the insert latency of ALEX increases most greatly in `Face` and `Osmc` datasets. Because ALEX splits the nodes and increases the index height, such that maintaining accurate model predictions and enough gaps for in-place inserts. (*iii*) DPGM's lookup latency varies greatly during block-wise loading. Because DPGM adopts the logarithmic merge method, and its PGM components can greatly change during inserts.

Finding 6. *Learned indexes do not outperform traditional indexes for bulk loading time. Among learned indexes, those using the greedy split method gain faster bulk loadings, but at the cost of worse lookup performances. The insert/lookup latency of most indexes increases slowly during block-wise loading, while DPGM's lookup latency could significantly changes.*

4.7 String Index Evaluation

Figure 19 shows the throughput of indexes supporting strings keys on `Url` workloads. We observe that learned indexes do not outperform traditional indexes in all cases. There are two reasons. (*i*) Since the string keys are much longer than the integer keys, the model prediction and position search require more overhead. (*ii*) SIndex deals with the string key as the high-dimensional integers, and adopts multi-variant linear models to fit the key-to-position mapping. However, the high-dimensional data has more complicated distribution, and the prediction error is relatively large. In contrast, `Wormhole` adopts a trie to deal with the keys with variable lengths, and adopts hash table to accelerate the search of the keys, which accelerate both the lookup and insert throughput.

Finding 7. *Learned indexes cannot outperform traditional indexes on string keys, because the string keys are of long length and the distribution is complicated to predict.*

4.8 Learned Index Selection Guidance

Based on the experimental results, we summarize how to select learned indexes (Figure 20). First, learned indexes can replace traditional indexes for simple datasets (e.g., relatively smooth CDF without abrupt shifts), uniform inserted pairs, and read-heavy workloads. However, learned indexes have no advantage against traditional indexes for complex datasets, skewed insert patterns, and workloads with many range queries, heavy writes, concurrent operations, or string keys. Second, for learned indexes in single-thread scenarios, we consider three cases. (*i*) If there are only inserts, DPGM [11] is the best choice, which efficiently supports inserts with delta buffers. (*ii*) If there are both inserts and lookups, LIPP [49] is the most suitable, which locates keys for inserts/lookups without position searches. (*iii*) If there are inserts, lookups and range queries, ALEX [7] is most suitable, which conducts minor positions searches and uses a linked list of leaf nodes for range queries.

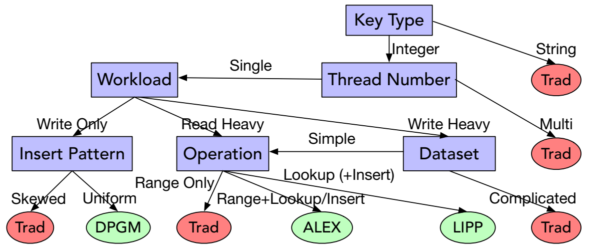


Figure 20: Learned index selection ('Trad': traditional).

5 CONCLUSION AND FUTURE WORK

We have systematically reviewed existing learned indexes and evaluated them under the same evaluation framework. We have tested three typical scenarios (static, dynamic, concurrency) with real datasets. The results and findings can guide researchers and practitioners in selecting appropriate learned indexes for their applications. We have also provided a unified testbed to help researchers design new learned indexes that can reduce the overhead of design and implementation.

Based on our findings, there are still some open problems and research challenges. First, existing learned indexes struggle to perform well in different scenarios. For example, (*i*) for static scenarios, learned indexes have similar or even worse range query performance than traditional indexes; (*ii*) for dynamic scenarios, learned indexes sometimes fail to outperform traditional indexes in write-heavy workloads; (*iii*) for concurrency scenarios, most of existing learned indexes adopt delta-buffer to store inserted pairs, which slows down the lookups. Besides, the insert performance of learned indexes drop greatly for highly skewed inserted data, while traditional indexes achieve more stable performance. It requires new index designs to solve above performance issues (e.g., designing in-place insert learned index to efficiently support concurrent lookups/inserts). Second, learned indexes lack practical features, such as recovery from system failures and persistence [27, 28, 59]. Third, there is a need for guidelines and best practices for implementing learned indexes in real databases (e.g., estimating index benefits based on model retraining issues).

ACKNOWLEDGEMENTS

This paper was supported by NSF of China (61925205, 62232009, 62102215), Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist). Guoliang Li is the corresponding author.

REFERENCES

[1] A. H. 0001 and T. Heinis. MADEX: Learning-augmented Algorithmic Index Structures. In B. He, B. Reinwald, and Y. Wu, editors, *AIDB@VLDB 2020*, 2020.

[2] M. M. Andersen and P. Tözün. Micro-architectural analysis of a learned index. In *Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–12, 2022.

[3] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *SIGMOD*, SIGFIDET '70, page 107–141, New York, NY, USA, 1970. Association for Computing Machinery.

[4] A. Crotty. Hist-Tree : Those Who Ignore It Are Doomed to Learn. In *11th Annual Conference on Innovative Data Systems Research (CIDR)*, 2021.

[5] Z. Dai and A. Shrivastava. Adaptive learned bloom filter (Ada-BF): Efficient utilization of the classifier with application to real-time information filtering on the web. In *Advances in Neural Information Processing Systems*, volume 2020-Decem, pages 1–11, 2020.

[6] A. Davitkova, E. Milchevski, and S. Michel. The ML-index: A multidimensional, learned index for point, range, and nearest-neighbor queries. In *EDBT*, volume 2020-March, pages 407–410, 2020.

[7] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. Lomet, and T. Kraska. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*, pages 969–984, 2020.

[8] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. In *Proceedings of the VLDB Endowment*, volume 14, pages 74–86, 2020.

[9] J. Dittrich, J. Nix, and C. Schön. The next 50 Years in Database Indexing or: The Case for Automatically Generated Index Structures. In *VLDB*, volume 15, pages 527–540, 2021.

[10] P. Ferragina, F. Lillo, and G. Vinciguerra. Why are learned indexes so effective? *ICML 2020*, PartF16814:3104–3113, 2020.

[11] P. Ferragina and G. Vinciguerra. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. In *VLDB*, volume 13, pages 1162–1175, 2020.

[12] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. Fiting-tree: A data-aware index structure. In *SIGMOD*, SIGMOD '19, page 1189–1206, New York, NY, USA, 2019. Association for Computing Machinery.

[13] A. Hadian and T. Heinis. Interpolation-friendly B-trees: Bridging the gap between algorithmic and learned indexes. In *EDBT*, volume 2019-March, pages 710–713, 2019.

[14] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: Fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, SIGMOD '10, page 339–350, New York, NY, USA, 2010. Association for Computing Machinery.

[15] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. SOSD: A Benchmark for Learned Indexes. In *MLForSystems@NeurIPS 2019*, 2019.

[16] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. Radixspline: A single-pass learned index. In *Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[17] E. M. Kornaropoulos, S. Ren, and R. Tamassia. The price of tailoring the index to your data: Poisoning attacks on learned index structures. In *SIGMOD*, SIGMOD '22, page 1331–1344, New York, NY, USA, 2022. Association for Computing Machinery.

[18] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *SIGMOD*, pages 489–504, 2018.

[19] H. Lan, Z. Bao, and Y. Peng. A survey on advancing the DBMS query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Sci. Eng.*, 6(1):86–101, 2021.

[20] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings - International Conference on Data Engineering*, pages 38–49. IEEE, 2013.

[21] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The art of practical synchronization. In *Data Management on New Hardware*, pages 1–8, 2016.

[22] J. Leskovec, L. Backstrom, and J. Kleinberg. Meme-tracking and the dynamics of the news cycle. In *SIGKDD*, KDD '09, page 497–506, New York, NY, USA, 2009. Association for Computing Machinery.

[23] G. Li, X. Zhou, and L. Cao. AI meets database: AI4DB and DB4AI. In *SIGMOD*, pages 2859–2866. ACM, 2021.

[24] P. Li, Y. Hua, J. Jia, and P. Zuo. FINEdex: A Fine-grained Learned Index Scheme for Scalable and Concurrent Memory Systems. In *VLDB*, volume 15, pages 321–334, 2021.

[25] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan. LISA: A Learned Index Structure for Spatial Data. In *SIGMOD*, pages 2119–2133, 2020.

[26] Q. Liu and L. Zheng. Stable Learned Bloom Filters for Data Streams. In *Proceedings of the VLDB Endowment*, volume 13, pages 2355–2367, 2020.

[27] B. Lu, J. Ding, E. Lo, U. F. Minhas, and T. Wang. APEX: A High-Performance Learned Index on Persistent Memory. In *VLDB*, volume 15, pages 597–610, 2021.

[28] C. Ma, X. Yu, Y. Li, X. Meng, and A. Maoliniyazi. Film: A fully learned index for larger-than-memory databases. *VLDB*, 16(3):561–573, 2022.

[29] M. Maltry and J. Dittrich. A critical analysis of recursive model indexes. In *Proceedings of the VLDB Endowment*, volume 15, pages 1079–1091, 2022.

[30] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, sep 2020.

[31] R. Marcus, E. Zhang, and T. Kraska. CDFShop: Exploring and Optimizing Learned Index Structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2789–2792, 2020.

[32] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, volume 509518, 1998.

[33] M. Mitzenmacher. A model for learned bloom filters, and optimizing by sandwiching. In *NIPS*, volume 2018-Decem, pages 464–473, 2018.

[34] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning Multi-dimensional Indexes. In *SIGMOD*, pages 985–1000, 2020.

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

[36] J. Qi, G. Liu, C. S. Jensen, and L. Kulik. Effectively learning spatial indices. In *VLDB*, volume 13, pages 2341–2354, 2020.

[37] J. W. Rae, S. Bartunov, and T. P. Lillicrap. Meta-learning neural bloom filters. In *ICML 2019*, volume 2019-June, pages 9188–9197, 2019.

[38] I. Sabek, K. Vaidya, D. Horn TUM, A. Kipf, M. Mitzenmacher, T. Kraska, D. Horn, and T. Kraska Can. Learned Models Replace Hash Functions. 16(1):532–545, 2022.

[39] U. Sirin, A. Yasin, and A. Ailamaki. A methodology for oltp micro-architectural analysis. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, pages 1–10, 2017.

[40] B. Spector, A. Kipf, K. Vaidya, C. Wang, U. F. Minhas, and T. Kraska. Bounding the Last Mile: Efficient Learned String Indexing. In *AIDB@VLDB 2021*, 2021.

[41] M. Stoian, A. Kipf, R. Marcus, and T. Kraska. Towards Practical Learned Indexing. In *AIDB@VLDB 2021*, 2021.

[42] J. Sun, J. Zhang, Z. Sun, G. Li, and N. Tang. Learned cardinality estimation: A design space exploration and A comparative evaluation. *Proc. VLDB Endow.*, 15(1):85–97, 2021.

[43] C. Tang, Y. Wang, Z. Dong, G. Hu, Z. Wang, M. Wang, and H. Chen. Xindex: A scalable learned index for multicore data storage. In *PPoPP*, pages 308–320, 2020.

[44] K. Vaidya, E. Knorr, M. Mitzenmacher, and T. Kraska. Partitioned learned bloom filters. In *International Conference on Learning Representations*, 2021.

[45] J. Wang, C. Chai, J. Liu, and G. Li. FACE: A normalizing flow based cardinality estimator. *Proc. VLDB Endow.*, 15(1):72–84, 2021.

[46] Y. Wang, C. Tang, Z. Wang, and H. Chen. SIndex: A scalable learned index for string keys. In *APSys 2020*, pages 17–24, 2020.

[47] Z. Wang, H. Chen, Y. Wang, C. Tang, and H. Wang. The Concurrent Learned Indexes for Multicore Data Storage. *ACM Transactions on Storage*, 18(1), 2022.

[48] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang. Are updatable learned indexes ready? *Proc. VLDB Endow.*, 15(11):3004–3017, jul 2022.

[49] J. Wu, Y. Zhang, S. Chen, J. Wang, Y. Chen, and C. Xing. Updatable learned index with precise positions. In *VLDB*, volume 14, page 1276, 2021.

[50] S. Wu, Y. Cui, J. Yu, X. Sun, T.-W. Kuo, and C. J. Xue. Nfl: Robust learned index via distribution transformation. *Proc. VLDB Endow.*, 15(10):2188–2200, jun 2022.

[51] S. Wu, Y. Li, H. Zhu, J. Zhao, and G. Chen. Dynamic index construction with deep reinforcement learning. *Data Sci. Eng.*, 7(2):87–101, 2022.

[52] X. Wu, F. Ni, and S. Jiang. Wormhole: A fast ordered index for in-memory data management. In *Proceedings of the 14th EuroSys Conference 2019*, 2019.

[53] Y. Wu, J. Yu, Y. Tian, R. Sidle, and R. Barber. Designing succinct secondary indexing mechanism by exploiting column correlations. *SIGMOD*, pages 1223–1240, 2019.

[54] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P. Å. Larson, D. Kossmann, and R. Acharya. Qd-tree: Learning Data Layouts for Big Data Analytics. In *SIGMOD*, number 2, pages 193–208, 2020.

[55] A. Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.

[56] X. Yu, C. Chai, G. Li, and J. Liu. Cost-based or learning-based? A hybrid query optimizer for query plan selection. *Proc. VLDB Endow.*, 15(13):3924–3936, 2022.

[57] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement learning with tree-lstm for join order selection. In *ICDE*, pages 1297–1308. IEEE, 2020.

[58] J. Zhang and Y. Gao. Carmi: A cache-aware learned index with a cost-based construction algorithm. *Proc. VLDB Endow.*, 15(11):2679–2691, jul 2022.

[59] Z. Zhang, Z. Chu, P. Jin, Y. Luo, X. Xie, S. Wan, Y. Luo, X. Wu, P. Zou, C. Zheng, et al. Plin: a persistent learned index for non-volatile memory with high performance and instant recovery. *VLDB*, 16(2):243–255, 2022.

[60] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. *IEEE Trans. Knowl. Data Eng.*, 34(3):1096–1116, 2022.