Accelerating String-key Learned Index Structures via Memoization-based Incremental Training

Minsu Kim
KAIST
mskim@casys.kaist.ac.kr

Jinwoo Hwang
KAIST
jwhwang@casys.kaist.ac.kr

Guseul Heo
KAIST
gsheo@casys.kaist.ac.kr

Seiyeon Cho
KAIST
sycho@casys.kaist.ac.kr

Divya Mahajan
Georgia Tech
divya.mahajan@gatech.edu

Jongse Park
KAIST
jspark@casys.kaist.ac.kr

ABSTRACT

Learned indexes use machine learning models to learn the mappings between keys and their corresponding positions in key-value indexes. These indexes use the mapping information as training data. Learned indexes require frequent retrainings of their models to incorporate the changes introduced by update queries. To efficiently retrain the models, existing learned index systems often harness a linear algebraic QR factorization technique that performs matrix decomposition. This factorization approach processes all key-position pairs during each retraining, resulting in compute operations that grow linearly with the total number of keys and their lengths. Consequently, the retrainings create a severe performance bottleneck, especially for variable-length string keys, while the retrainings are crucial for maintaining high prediction accuracy and in turn, ensuring low query service latency.

To address this performance problem, we develop an algorithm-hardware co-designed string-key learned index system, dubbed SIA. In designing SIA, we leverage a unique algorithmic property of the matrix decomposition-based training method. Exploiting the property, we develop a memoization-based incremental training scheme, which only requires computation over updated keys, while decomposition results of non-updated keys from previous computations can be reused. We further enhance SIA to offload a portion of this training process to an FPGA accelerator to not only relieve CPU resources for serving index queries (i.e., inference), but also accelerate the training itself. Our evaluation shows that compared to ALEX, LIPP, and SIndex, a state-of-the-art learned index systems, SIA-accelerated learned indexes offer 2.6× and 3.4× higher throughput on the two real-world benchmark suites, YCSB and Twitter cache trace, respectively.

PVLDB Reference Format:
Minsu Kim, Jinwoo Hwang, Guseul Heo, Seiyeon Cho, Divya Mahajan, and Jongse Park. Accelerating String-key Learned Index Structures via Memoization-based Incremental Training. PVLDB, 17(8): 1802 - 1815, 2024.
doi:10.14778/3659437.3659439

Proceedings of the VLDB Endowment, Vol. 17, No. 8 ISSN 2150-8097.
doi:10.14778/3659437.3659439

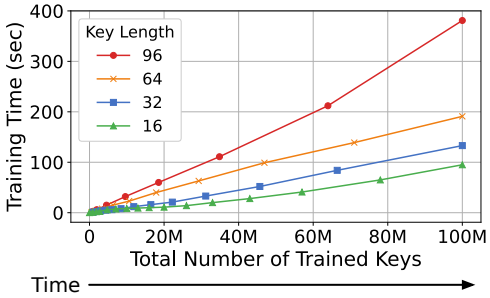

Figure 1: Increasing retraining time as the size of a learned index system grows, resulting from a stream of update queries. Markers on the same line represent sequential retraining runs, where leftward markers precede those on the right.

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

1 INTRODUCTION

Machine learning for system infrastructure is growing particularly in areas where data-driven decisions can make meaningful strides [7, 20, 62]. Efficient data access is one such avenue, where learning indexes have proven to be effective and practical [1, 11, 12, 17, 27, 29, 33–36, 38, 41–43, 51, 52, 54–57, 59–61, 64, 65, 73, 74, 76]. The pioneering work [27] proposed in this space uses a collection of machine learning models to create a read-only ordered index for integer keys. Due to its popularity and applicability, numerous follow-up research projects have extended the initial work to support read-write (updatable) indexes [11, 15, 29, 33, 35, 57, 58, 61, 64, 65, 68, 72], string keys [53, 61, 63], multi-dimensional indexes [12, 17, 43, 60], spatial indexes [34, 51, 73, 76], and other variants [36, 38, 54, 74]. This paper focuses on identifying performance challenges of *updatable string-key* learned indexes and addressing the challenges through an algorithm-hardware co-designed solution.

Regardless of the data types of keys, an algorithmic commonality among most existing learned indexes is that the indexes are constructed as a hierarchical structure where each node is a linear model [10–12, 27, 33, 35, 36, 57, 61, 64, 65, 74, 75]. These linear models are designed to collaboratively *learn* the mappings between keys and their corresponding positions, using this information as training data. The training process is inherently repetitive since the

key-position mappings constantly change due to the update queries (e.g., insert or delete), which necessitates *retrainings* to incorporate the changes into the models.

In learned indexes, training of linear models is essentially solving the following linear equation, $X\beta$=Y where X is a key matrix, β is a learnable parameter vector, and Y is the corresponding position vector. When learned indexes only support integer keys, the training process is computationally trivial since X is a vector of integer key values (i.e., n×1 matrix). However, when the keys are variable-length strings, X becomes a n×k-size matrix where k is the key length, which makes solving the equation a computationally non-trivial task. To algorithmically reduce the compute load of this training, existing string-key learned indexes [27, 53, 61, 63] employ a matrix factorization strategy known as *QR decomposition*, which enables training to be free from the burdens of matrix inversion.

Despite the algorithmic optimization, we observe that in the existing systems, the repetitive retrainings incur a severe performance bottleneck, since (1) the complexity of QR decomposition, although lower than matrix inversion, remains high, and (2) retrainings and index query servicing for existing keys (i.e., inference) compete for the limited CPU resource. Figure 1 shows that retraining time progressively grows as the number of keys and key lengths increase, on a state-of-the-art string-key learned index, SIndex [61]. Each point in the graph represents a retraining run. Increased retraining times negatively impact the inference throughput, as they result in an outdated index. This, in turn, lowers the index prediction accuracy and necessitates a costly linear search to locate the correct position. Thus, retraining is crucial for reducing service latency as well as improving index throughput.

To address the aforementioned bottlenecks, we introduce **SIA**: **S**tring-key Learned **I**ndex **A**cceleration. SIA enables efficient and scalable indexing by reducing the compute load of the retraining process through an algorithmic technique and judiciously offloads a portion of the training computation onto an FPGA accelerator. The challenge is that current learned indexes need to perform costly matrix decomposition using the *entire* key-position mappings as input to maintain model accuracy, which is pivotal for achieving high index performance. To tackle this challenge, SIA utilizes a modified parallel decomposition technique that allows for piecewise computation of matrix decomposition. In designing SIA, we leverage the insight that these retrainings occur on progressively updated indexes, thus offering an opportunity to reuse computations from prior results via memoization. It is important to note that training using the memoized decomposition results produces mathematically identical outcomes to those obtained if the models were fully retrained from scratch using the complete set of keys.

Building on the memoization-based decomposition, we develop a learned index training algorithm that incrementally retrains the models by leveraging the results of prior matrix decomposition. This enhanced algorithm reduces the computational complexity and retraining time, which in turn frees up CPU resources for servicing queries. However, our empirical analyses suggest that the algorithmic optimization, while helpful, offers a limited benefit since the retrainings still compete over the limited CPU resource. To further reduce the retraining time, we enable the retrainings to be accelerated using an FPGA. We choose FPGA over GPU owing to its customizability to index-specific algorithm configurations, leading to enhanced energy efficiency. SIA combines these elements to offer a novel learned index mechanism that aims to improve system query throughput through both algorithmic and hardware innovations. This work makes the following contributions:

- Identifies the system bottlenecks in current updatable learned index structures for string-keys, specifically, retraining the ensemble models in the hierarchical structure. We observe that as the retraining time grows, it progressively leads to lower performance of learned index systems.
- Introduces a novel learned index system, SIA, that accelerates the retraining process through an enhanced mathematical approach to matrix decomposition, enabling incremental training. With incremental training, only updated keys are used for computation, while the computation result for old keys is reused.
- Further accelerates SIA's incremental training process using an FPGA-based design that reduces training time and frees up CPU resources for index query servicing.

We demonstrate the effectiveness of SIA using two real-world benchmark suites, YCSB and Twitter cache trace. For YCSB, we use two datasets available to the public, Amazon review and MemeTracker datasets, as well as a synthetic dataset. We integrate SIA into the three updatable string-key learned indexes, including ALEX [11], LIPP [64], and SIndex [61]. Compared to baseline learned indexes, SIA provides 2.6× and 3.4× higher throughput for YCSB and Twitter cache trace workloads, respectively. From an in-depth ablation study using SIndex that breaks down the benefits of SIA, we observe that employing solely the memoization decomposition-based incremental learning algorithm offers 1.6× and 1.9× higher throughput. However, when the FPGA-based SIA accelerator is employed, it offers 2.8× and 4.3× higher throughput than the baselines, which are respectively 1.8× and 2.3× *additional* speedup, a substantial performance boost compared to the software-only counterpart. These results suggest that taking an algorithm-hardware co-design approach, SIA enables heterogeneous CPU-FPGA architecture to operate as a platform of choice to achieve high throughput for updatable string-key learned indexes. Our software and hardware code for SIA is available at https://github.com/casys-kaist/sia.

2 A PRIMER ON LEARNED INDEX

Key-value stores are widely deployed in data management applications, where the index maps keys to their corresponding positions in a list of records. This pairing can be denoted as a function, $f(key,\ position)$, with key as the input and position as the output. Conventionally, hash-map and B-tree structures are commonly used to store this mapping in an array of records. Despite its popularity, they still have shown several limitations, which prevent their "one-size-fits-all" deployment. While hash-maps typically offer low average access time, they can be susceptible to hash collisions that may lead to unpredictable increases in lookup and construction time. Additionally, hash-maps may not perform as well as other data structures for range queries. On the other hand, B-trees and their variants do not have the same limitations as hash-maps, but their average-case performance, in terms of both latency and throughput, is generally lower than that of hash-maps. To overcome these limitations, the community has explored the use of machine learning

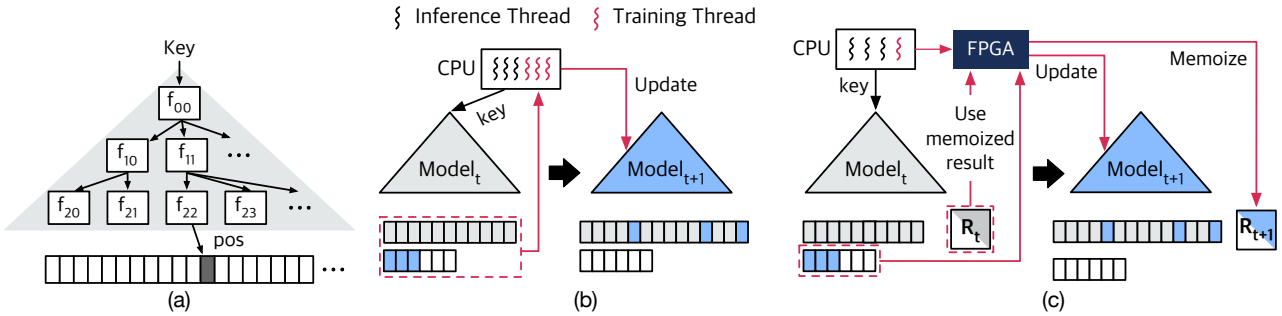

Figure 2: (a) Read-only learned index in a hierarchical structure, (b) updatable learned index, and (c) SIA: the proposed updatable string-key learned index that leverages computation reuse and hardware acceleration to improve the system throughput.

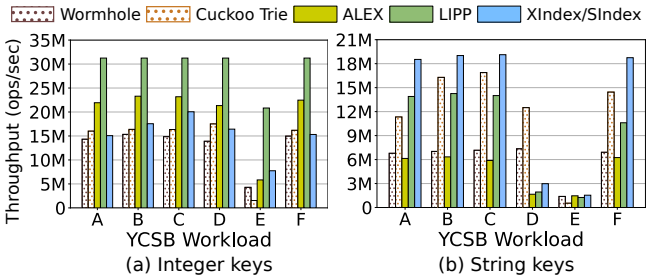

Figure 3: Throughput results of two conventional indexes and three learned indexes for YCSB workloads.

(ML) approaches to develop learned indexes as an alternative index structure [10–13, 27, 34, 41, 57, 61, 64].

Learning to index keys. The initial work on learned indexes [27] demonstrated that it is possible for ML models to learn the mappings between keys and their corresponding positions, using this information as training data. Unlike traditional machine learning models, which aim to generalize to unseen inputs, learned index models are intentionally overfit to the training data, as index structures mostly encounter keys that have been inserted. Despite the overfitting during training, inference-based indexing can still produce incorrect predictions due to the inherent approximation nature of machine learning. When the queried key is not found at the predicted position, learned indexes search for the key within a bounded range around the predicted position (i.e., $[p + err_{min}, p + err_{max}]$), ensuring accurate indexing functionality [27].

When designing a ML model for learned indexes, there are various alternatives that trade off accuracy (model size and architecture) against cost (inference latency and training time). Several works [13, 27, 57, 61] have shown that a hierarchical structure of linear models effectively balances this tradeoff. Each node in the hierarchical structure is a linear regression model that needs to be trained for a subset of the key-position mappings. These initial works focus on read-only indexes, and hence training is carried out once when building the indexes before deployment. The hierarchical structure for read-only indexing is depicted in Figure 2(a).

Updatable string-key learned index. Although restricting the scope to *read-only* indexes was an effective setting to demonstrate initial applicability of the "learning" approach, practical data management necessitates support for "update" queries (e.g., `insert` and `delete`). Follow-up works overcome this limitation and devise "updatable" learned indexes [11, 57, 61, 64]. ALEX [11] expands nodes with deliberately-reserved empty spaces for unseen future keys, which hold the newly inserted keys until the updated keys are retrained. LIPP [64] ensures precise model prediction results and removes costly local search usually used in other learned indexes. XIndex [57] is another variant that maintains reserved spaces for future keys, while unlike ALEX, the new keys are stored in separate temporary buffers. SIndex [61] is one of the initial efforts to support variable-length string keys in learned indexes. As string keys are an important datatype used in diverse applications such as web servers, sequence analysis, and genomics, modern key-value stores often have strong support for this datatype [3, 19, 21, 24, 28, 31, 37, 61, 66, 67, 70, 71]. Despite its importance, its performance implication on updatable string-key learned index systems remains under-examined in existing literature, which is the primary focus of this work.

Intertwinement of retraining and inference. Unlike traditional machine learning, training and inference phases in these updatable learned indexes are not clearly demarcated. Instead, learned indexes require iterative *retrainings*, because the training data is constantly changing due to update queries. Concurrently, the index systems must serve index queries by performing *inference*. This convergence of training and inference can influence each other's performance, potentially resulting in a marked degradation of overall efficiency. Figure 2(b) delineates the common execution flow where certain threads are dedicated to query servicing and certain to retraining.

Effectiveness of learned indexes. To better understand the effectiveness of learned indexes, we conduct preliminary experiments comparing the throughput of learned index structures with two *non*-learned indexes, Wormhole [66] and Cuckoo Trie [71]. We use the Yahoo! Cloud Serving Benchmark (YCSB) [8], a key-value store benchmark suite with six different workloads (see Section 7.1 for details). Figure 3 shows that learned indexes generally offer comparable or higher performance than the two baseline indexes for both integer and string key cases. However, the notable observation is that when keys are string, learned indexes perform much worse than the baselines for workload D and E. Workload D and E contain

`insert` queries, which necessitate the constant retrainings for index updates. While the retrainings impose marginal overhead when keys are integers, retrainings for string keys become severe performance bottleneck, cancelling the performance gains of learned indexes, as will be deeply analyzed in Section 3. This is the very challenge we aim to tackle in this work through SIA.

Our approach. SIA sets out to tackle the challenges posed by current updatable string-key learned indexes, with the following objectives: (1) SIA aims to reduce the cost of training linear models without any mathematical implication on model quality, and (2) it aims to enhance the system with an FPGA accelerator that can execute the compute-intensive portion of training, thus relieving CPU resources for inference. Figure 2(c) depicts SIA's system architecture, which is built upon existing learned index systems.

3 ANALYSES OF LEARNED INDEXES

We conduct in-depth performance characterizations through a set of experiments and obtain three main insights from the results. These insights form the key driving forces behind SIA. For these analyses, we use a SIndex system running on a 16-core server, the details of which are provided in Section 7.1. We use a workload with uniformly distributed keys, generating `read` and `insert` queries based on a predetermined ratio (e.g., 70% `read` and 30% `insert` queries). `Insert` queries raise the retraining complexity by adding more keys to the index. We initialize the index with 1M keys.

3.1 Retraining-Time Scalability Analysis

Existing updatable string-key learned indexes suffer from a limitation in that they aggregate all keys into a single dataset, making computations more demanding as the number of keys increases. To examine the scalability aspect of learned indexes, we measure the retraining time as we gradually increase the total number of keys from 1M to approximately 100M. Figure 1 shows the results with each marker representing a retraining invocation. The experiment shows that for total numbers of keys reaching 100M, the retraining time becomes prohibitively long. Retraining time for the shortest key length of 16 increases up to 100 seconds, while it exceeds 5 minutes for the case of key length 96. These extended retraining times for indexing are infeasible as they result in the index being significantly outdated. The results also show that progressively prolonged retraining time ends up leading to longer intervals between retraining invocations. This delay occurs because the growing retraining time increases the number of keys waiting for the next round of retraining, resulting in a lower frequency of model updates.

This analysis shows that the existing updatable string-key learned index systems face scalability issues. Thus, there is a need for a solution that minimizes the retraining time for linear models, especially when dealing with large index sizes and long key lengths.

3.2 Impact of Slow Retraining on Throughput

Given the aforementioned insight, a subsequent research question could be, "Why is retraining vital for the overall efficacy of the learned index system?" The response to this inquiry is that training influences throughput in two significant ways. (1) First, slow retraining causes the models to become outdated, resulting in reduced index prediction accuracy and requiring a costly linear search to locate the correct position. This, in turn, leads to prolonged index search latencies for more `read` queries, negatively impacting the overall system throughput. (2) Second, as retraining and inference run simultaneously on the same system and compete for CPU resources, the inference throughput is adversely affected. We discuss the first implication in this section and leave the discussion for the second effect to Section 3.3.

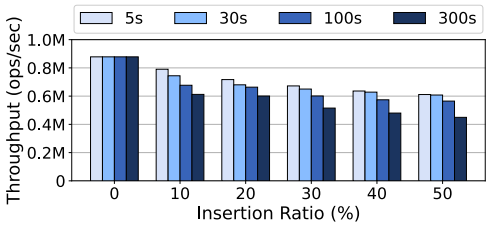

Figure 4: Throughput as training time varies from 5 to 300 seconds. Training does not utilize any CPU cycles. The insertion ratio sweeps from 0% (read-only) to 50%.

To demonstrate the impact of slow training over throughput, we develop a "fictitious" system that can retrain linear models within a predetermined training time *without* using any CPU resources for training. This method allows us to isolate the impact of slow retraining separate from the implications of CPU resource contention. Figure 4 depicts the throughput of this fictitious system as the retraining duration shifts between 5s and 300s. The results show a consistent decline in throughput as the retraining time lengthens, since learned indexes must use outdated models during the retraining period, which would increase the frequency and degree of linear search to locate the correct position. Additionally, we observe that as the insertion ratio rises, the system sees a decline in throughput. This is because a greater number of inserted keys await in the buffer before integration into the learned index, which again requires more overhead on linear search at the non-trained key buffers. While the reported throughput averages over time, in a practical scenario, throughput would gradually drop as runtime progresses, because, unlike our hypothetical system, a real system would face an ever-increasing retraining time.

Our study suggests that a long retraining period hurts the end-to-end system throughput of updatable string-key learned index systems. Therefore, fast retraining of linear models is imperative.

3.3 Implication of CPU Resource Allocation

A straightforward solution to reduce training time would be to allocate more CPU resources. To better understand the correlation between throughput and CPU resources, we perform an experiment that measures the system throughput as we vary the number of threads allocated for inference (index serving) and training, while maintaining the number of threads assigned to the other task at 1. This approach allows us to determine the performance benefits that inference and training could achieve with additional CPU resources, respectively. As our system has 16 cores, we vary the number of

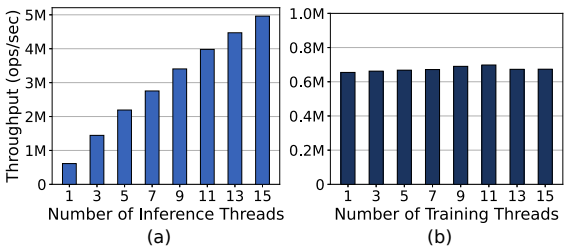

Figure 5: Throughput with varying threads for (a) inference and 1 for training, (b) training and 1 for inference.

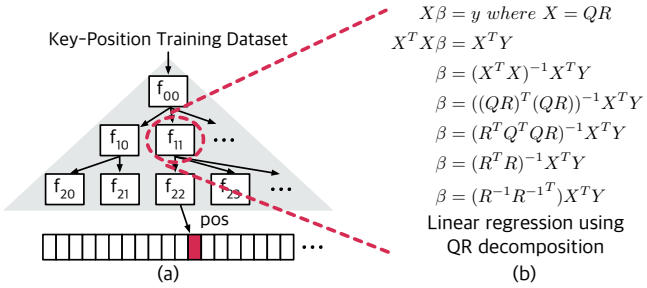

Figure 6: (a) Recursive Model Index, and (b) Linear regression training operations required per model.

cores allocated to either inference or training threads from 1 to 15, maintaining the insertion ratio at 50% and the key length at 32.

Figure 5(a) and Figure 5(b) show throughput trends. When the number of threads for training is 1, the additional CPU threads allocated for inference result in sub-linear yet substantial performance scaling. This is because inference is read-only and multiple inferences can be executed independently and in parallel across threads. However, when the inference process is restricted to a single thread while training utilizes an increasing number of cores, the additional resources only yield marginal benefits. The limited effectiveness of CPU for training can be attributed to the limited parallelism in the matrix decomposition algorithm used for linear regression training, as explained in further detail in Section 5.2.

We note that inference gains more from extra CPU resources compared to training. As a result, we propose a heterogeneous system that allocates CPU resources primarily for inference, while employing an FPGA accelerator for the training process.

4 SIA DESIGN PRINCIPLES

Building upon the insights, we propose a hardware-accelerated updatable string-key learned index system, dubbed SIA. First, SIA proposes a novel incremental index learning algorithm, which reduces the computing complexity and execution time of each retraining process. SIA then dedicates most of the CPU resource for inference serving by offloading the training to an FPGA accelerator, thus collaboratively achieving high throughput. This section outlines the design principles of each SIA component.

***Algorithm* design principle: Performing only necessary computations for learned indexes.** The fundamental challenge addressed in this work is the lack of scalability in learned index training since the compute operations for training *compounds* as the number of keys grows. In learned index systems, every retraining run necessitates the processing of the entire dataset. The current state-of-the-art approach involves performing matrix decomposition, matrix inverse, dot product, and transpose operations over the *entire* dataset to determine the parameters of the linear models. To reduce the computing complexity of the training, we devise an incremental index learning algorithm that memoizes the results of previous retraining computations and reuses them in combination with the new results obtained from the augmented training data. With this algorithm, the computational load is not determined by the total number of keys, but rather by the number of updated keys.

***Hardware* design principle: Designing the accelerator specifically for training to ensure high energy efficiency as index systems are often dedicated for the exclusive purpose and are consistently operational.** Updatable learned indexes must ceaselessly perform training to keep up with the changes made by update queries, which makes achieving high energy efficiency a primary concern in designing the systems. Although employing a GPU is seemingly a straightforward approach to attaining high throughput, the advantage is offset by substantial energy consumption. Thus, in this work, we choose FPGA as our platform. FPGA not only allows us to customize accelerators for diverse algorithm/system constraints and thus achieve high energy efficiency, but also it is already available in the form of off-the-shelf cards, which facilitates integration with the existing systems [25, 45]. To effectively utilize FPGAs for changing training configurations and index model sizes, we develop a hand-optimized design specifically for the proposed memoization-based incremental training algorithm.

***Software* design principle: Enabling plug-and-play based runtime software for generality and non-invasiveness.** While hardware acceleration can offer significant performance gains, the proposed technique needs to be integrated seamlessly with existing learned index systems. Thus, SIA cannot be specific to a certain updatable learned index. SIA's system software is built by determining the commonalities of existing updatable learned indexes and integrating the FPGA-based accelerator with minimal modifications to the existing software stack. To accomplish this objective, we utilize the fact that although various learned indexes may have different model structures and index management mechanisms, they all rely on linear regression models as the fundamental kernel, which can be readily separable from the other components of the index system. Given this insight, we encapsulate the accelerator and its driver as a linear model training library, which is customized for the case where the training data incrementally grows or shrinks.

5 INCREMENTAL INDEX LEARNING

Reducing the training workload of the updatable learned index structures is a key challenge tackled by this work. In this section, we first provide the background on training hierarchically structured learned indexes, which requires linear regression training using matrix decomposition. We then introduce SIA's novel index learning algorithm, which effectively reduces the computational load of the training process via reuse, without any changes to model quality.

Algorithm 1: Householder QR decomposition.

Input : X: Matrix of size $m \times n$
Output : R: Upper triangular matrix of size $n \times n$

for $(i \leftarrow 0$ *to* $n-2)$ **do**
  $col_i = X[i:m, i]$
  $d = \sqrt{\textbf{dot}\ (col_i, col_i)}$
  $ref_i = \textbf{cal\_reflector}\ (col_i, d)$
  $\gamma = -2\ /\ \textbf{dot}(ref_i, ref_i)$
  for $(j \leftarrow i$ *to* $n-1)$ **do**
    $col_j = X[i:m, j]$
    $\alpha = \gamma \times \textbf{dot}\ (ref_i, col_j)$
    $col_j = \textbf{axpy}\ (\alpha, ref_i, col_j)$
    $R[i, j] = X[i, j]$
  end
end

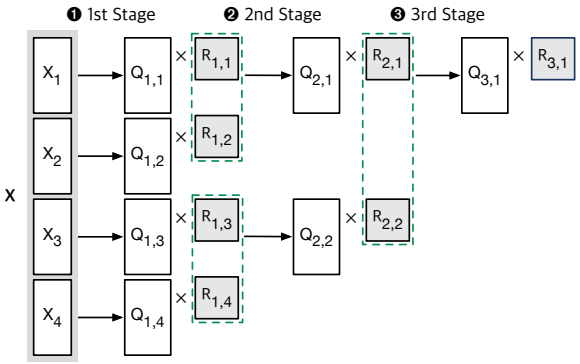

Figure 7: Parallel QR decomposition.

5.1 Hierarchical Model Index Training

Most learned indexes [11, 13, 26, 27, 33, 35, 36, 38, 57, 60, 61, 64, 65] share a unique commonality by employing a hierarchical model index structure, as illustrated in Figure 6(a). In the hierarchical structure, the internal and leaf nodes have different roles: *learned models at internal nodes* predict which node to traverse among the children and *learned models at leaf nodes* predict the positions for the queried keys. This structure splits the entire key range into a series of small and possibly overlapping ranges, where each range is assigned to a leaf node and learned with the associated model. Note that due to update queries, the index structure can potentially expand or shrink, as the total number of keys handled by the system increases and decreases.

The hierarchical index is trained in two main ways: (1) cold training from scratch, which is for new nodes created due to the index structure changes, and (2) updating pre-existing models within the existing nodes due to key additions or deletions without any alterations to the hierarchical index structure. Cold training is infrequent, typically triggered only when the prediction accuracy falls below a set threshold. Mostly, keys are updated without the need to add or remove any nodes. Hence, SIA focuses on optimizing the latter, reserving conventional training techniques for the former.

5.2 Linear Regression Training

Linear regression (LR) models the relationship between variables by fitting a linear equation to training data. Formally, given an input $X = ((x_{11}, .., x_{1p}), .., (x_{n1}, .., x_{np}))$ and output $Y = (y_1, .., y_n)$, a LR model is $Y = X\beta$ where $\beta = (\beta_1, .., \beta_p)$. Training determines β for a given dataset. In the context of learned index that uses variable-length string keys, the input to the models is a matrix X with n rows where each row is a numerically encoded key vector of length p, and Y (output) is a vector of integer values that represent the keys' positions in the sorted key array. Even for updating the pre-existing models, the entire X and Y are required to retrain all the models traversed in the hierarchical structure and determine their new βs. After retraining, the index for a given key can be predicted by performing a series of dot products between the traversed model input X and their corresponding βs.

Learning the parameters. Every β can be obtained by inversing the matrix X and multiplying it with the output vector Y (i.e., $\beta = X^{-1}Y$). However, computing the inverse matrix X^{-1} can be computationally prohibitive, especially when the matrix size is large. To tackle the challenge, an existing alternative approach commonly and widely used in practice is to employ a matrix factorization method, known as QR decomposition (QRD) technique. QRD decomposes a matrix X into a multiplication of two matrices: Q, an nxp-sized matrix with $Q^TQ = QQ^T = I$, and R, a pxp-sized upper triangular matrix.

Figure 6(b) illustrates the linear algebra operations required to determine β, leveraging the QRD technique. At first, the QRD of the input dataset X is performed, which produces Q and R. After the decomposition, the following operations are performed: (1) computing the inverse of the upper triangular matrix (R^{-1}), (2) transposing matrices (R^{-1^T} and X^T), (3) multiplying the resulting small matrices ($R^{-1}R^{-1^T}$), and (4) matrix-vector multiplication (X^TY). Note that during training, only the R matrix is required.

Householder QR decomposition. QR decomposition can be computed using various algorithmic methods [14, 18, 23]. Among these methods, we base SIA on the Householder algorithm [23] owing to its relatively enhanced numerical stability, while SIA remains compatible with other alternatives due to their algorithmically similar traits. Algorithm 12 illustrates the Householder algorithm [23]. The algorithm has two loops. In the outer loop, the algorithm iterates over the columns of the input matrix and calculates a vector, called a reflector (ref_i), and a scalar value γ. For each column, the inner loop visits all the columns located on the right of the current column one by one, and updates the visiting column while producing the $R[i][j]$ values. The nature of this process is fundamentally serial.

Parallelizing QR decomposition. Vanilla QRD algorithms, including Algorithm 12, execute sequentially by sweeping through the columns of an input matrix and gradually filling the rows and columns of the Q and R matrices, respectively. Thus, QRD can be slow for large matrices, as is the case with learned index. As the number of keys grows, the height of the key matrix X also increases ($n \times p$ matrix where $n \gg p$), making it a *tall-and-skinny* matrix.

Prior works [6, 16, 47] offer a parallelization mechanism customized for tall-and-skinny matrices. The parallelization mechanism exploits a mathematical property of orthogonal matrix Q that its transpose is equal to its inverse matrix, as depicted with an example in Figure 7. Let X be an input matrix for an LR model within

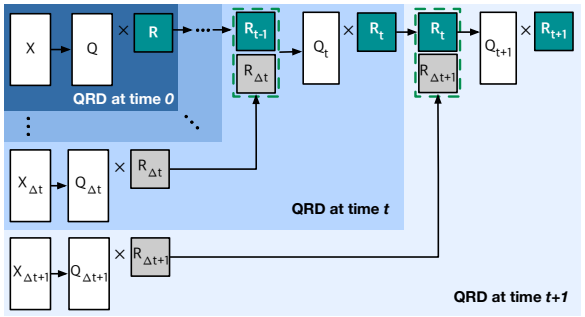

Figure 8: Memoized QR decomposition.

Algorithm 2: Incremental index learning algorithm.

Input : M_{old}: Current linear models
X_{old}: Current key matrices
X_Δ: Newly inserted key matrices
Y_{old}: Current index vectors
R_{old}: Memoized R matrices

Output : M_{new}: Updated linear models
X_{new}: Updated key matrices
Y_{new}: Updated index vectors
R_{new}: Newly memoized R matrices

Initialize $M_{new} \leftarrow \emptyset$, $X_{new} \leftarrow \emptyset$, $R_{new} \leftarrow \emptyset$
while $(m \in M_{old})$ **do**
  $mid \leftarrow m.model\_id$
  $X_{new}[mid] \leftarrow$ concat $(X_{old}[mid], X_\Delta[mid])$
  $Y_{new}[mid] \leftarrow$ calc_index$(Y_{old}[mid], X_{new}[mid])$
  $tmp = (X_{new}[mid])^T \times Y_{new}[mid]$
  $R_\Delta \leftarrow \mathbf{QR}(X_\Delta[mid])$
  $R_{tmp} \leftarrow$ concat$(R_{old}[mid], R_\Delta)$
  $R_{new}[mid] \leftarrow \mathbf{QR}(R_{tmp})$
  $\beta = ((R_{new}[mid])^{-1} \times ((R_{new}[mid])^{-1})^T) \times tmp$
  $M_{new}[mid].\beta \leftarrow \beta$
end

the tree. X is decomposed through three steps: (1) X is vertically split into smaller sub-matrices (X_1, X_2, X_3, X_4) and decomposed into QR matrices in parallel; (2) the QR decomposition is performed on the vertically concatenated R matrices (concat$(R_{1,1}, R_{1,2})$ and concat$(R_{1,3}, R_{1,4})$); (3) finally, the last QR decomposition is applied over concat$(R_{2,1}, R_{2,2})$ to produce $R_{3,1}$. The resulting $R_{3,1}$ is mathematically equivalent to R, obtainable by decomposing the X as a whole without parallelization.

5.3 SIA's Incremental Index Learning

Memoized QRD via computation reuse. Exploiting the mathematical insight of parallelized QRD, we modify the vanilla QRD that incurs a heavy amount of computation and devise a memoized QRD. Figure 8 shows the memoized QRD algorithm. We exclusively consider the case that the number of keys *grows* due to the `insert` queries[1]. When a learned index is retrained, we require the R matrix corresponding to the current X. To do so, we memoize the computed R matrix in memory at every retraining invocation (R_t). When a retraining is invoked, the rows of collected additional

[1]We will discuss the `delete` query handling in Section 6.4.

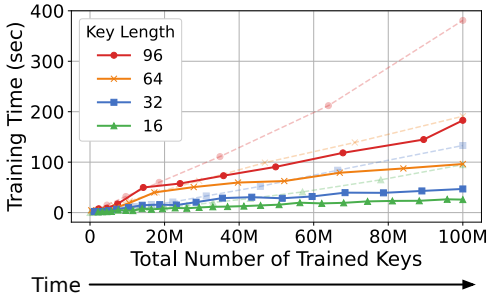

Figure 9: Increasing retraining time as the total number of keys increases with CPU-based memoized QRD on SIndex. For comparison, the shaded lines depict the results presented in Figure 1.

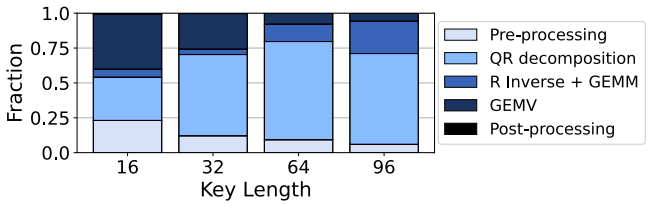

Figure 10: Breakdown of linear model training runtime.

keys $X_{\Delta t+1}$ is decomposed. Then, similar to the parallelized QRD, we concatenate the R_t and $R_{\Delta t+1}$, and perform one more QRD to obtain the final R_{t+1}. Now, R_{t+1} is used for linear model training and cached in memory for the next retraining run. Note that SIA's QRD algorithm involves only two small QR decompositions, which significantly reduces the compute load by reusing the performed computations. Moreover, the size of each R matrix is $p \times p$ where p is the key length, thus is very small and does not incur large memory footprint overhead. For instance, with a key length of 96, the size of R is merely 72 KB (=96×96×8).

SIA's incremental index learning algorithm. SIA's incremental index learning algorithm uses the memoized QRD to train the models in the updatable learned indexes. Algorithm 12 describes SIA's training process. The algorithm loops over the list of linear models in the hierarchical structure, which need to be updated. It concatenates the existing keys X_{old} with new keys X_Δ to obtain X_{new}, calculates indexes for the new keys to update Y_{old} with Y_{new}, and computes the $(X_{new})^T Y_{new}$. Then, the algorithm performs the memoized QRD, which results in R_{new}. Using R_{new}, the algorithm obtains the β and updates the model parameters with the new β. The obtained R_{new} is memoized for next retrainings. The same training process is repeated until the models of all leaf and internal nodes in the index structure are updated.

Limitation of software-only solution. We observe that the SIA's learning algorithm already substantially reduces the computational cost of training the learned index, even when implemented in software without hardware acceleration. Figure 9 shows the improved training time with the proposed memoized algorithm. Compared to the baseline reported in Figure 1, which is presented as dimmed lines in Figure 9, the retraining time is reduced for all the evaluated

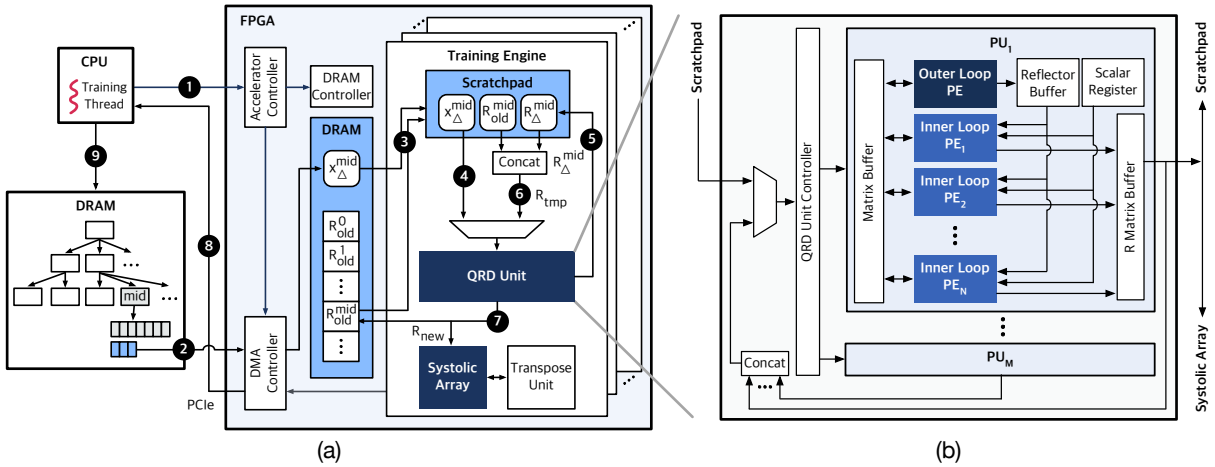

Figure 11: (a) SIA's system where the CPU runs the training thread to issue jobs to the accelerator. Accelerator executes the training operations on the training engine; (b) Microarchitecture of QR decomposition unit.

key lengths, as reflected by the slopes of line graphs. Moreover, this shortens the retraining interval, as shown in Figure 9 reporting a greater number of data points (markers), each corresponding to a retraining. However, Figure 9 also shows that the resulting reduction in training time is insufficient, still extending up to 200s.

Acceleration target determination. This observation motivates us to devise an efficient and performant hardware accelerator for training. However, the first crucial step is to determine the acceleration targets for offloading to the hardware. For this purpose, we first characterize the core compute kernels of training. Figure 10 shows the results as we vary the key length from 16 to 96. We look into four kernels: (1) training data matricization, (2) QR decomposition, (3) R matrix inverse calculation and matrix-matrix multiplication, and (4) matrix-vector multiplication. As the data matricization is mostly memory copy, it needs to be performed by CPU. We also rule out matrix-vector multiplication from the acceleration targets since it requires a memory copy for the entire X matrix from host to FPGA. To this end, this work focuses on accelerating QR decomposition, R inverse, and GEMM operations on the FPGA.

6 SIA SYSTEM DESIGN

While SIA employs the incremental learning algorithm to reduce the computation load, we enhance this algorithmic approach by incorporating an FPGA accelerator and customized runtime software to further accelerate SIA's training. We first describe the overview of SIA's system, and then, elaborate each component in detail.

6.1 FPGA-Accelerated Training Infrastructure

FPGAs have been commonly used as a successful platforms for acceleration [39, 40, 46, 50] and are even deployed in cloud datacenters [48]. Figure 11(a) depicts the SIA system accelerated using FPGA. As in existing learned index systems, SIA employs a multi-core CPU that can serve both inference and training. However, SIA also comes with an FPGA accelerator to offload training computation. We chose FPGA as the acceleration platform owing to its customizability to index-specific algorithms and high energy efficiency, which is crucial for index systems since trainings are consistently conducted throughout their lifespan. Unlike the existing systems, SIA only runs a single training thread to not only compute the non-accelerated memory-bound kernels, but also manage the data transfer between host and FPGA and control the accelerator invocations. The training thread iterates over a list of linear models within the hierarchical structure and initiates the retrainings of these models one by one on available Training Engines (TEs). To train a model, the newly inserted keys accumulated in the buffer (X_Δ) are first copied from host to FPGA. FPGA's off-chip memory maintains an array of R_{old} matrices, which are memoized from the previous retraining runs. In the figure, the superscript *mid* on the R and X matrices refers to the model ID. After the memory copy from the host to FPGA is completed, the training thread sets a control register in the accelerator controller, scheduling the training computation to an available TE. The training thread is also responsible for updating the model parameters, which occurs repeatedly during runtime, allowing the index to integrate new keys.

6.2 Accelerator Architecture

Training Engine. Figure 11(a) also depicts the TE architecture. The first computation performed by TE is the SIA's QRD algorithm described in Section 5.2. The TE feeds X_Δ to the QRD unit. It then obtains the R_Δ, which is concatenated with the memoized R_{old} in the scratchpad memory to produce the R_{tmp}. This R_{tmp} is then fed to the QRD unit as an input that produces R_{new}. The next step is to perform R_{new} matrix inversion and matrix-matrix multiplication (i.e., GEMM) between the inverse and its transpose. We exploit a parallelized matrix inverse algorithm, Heller's algorithm [22], which effectively converts a matrix inverse into a series of recursive GEMMs. As we transform all needed operations into a series of GEMMs, a systolic-array accelerator equipped with a transpose unit can complete all the necessary kernel executions. Once the computation is completed, the accelerator controller uses a control flag to inform the training thread about the completion.

QRD Unit. Due to its computational intensity in mathematical problems, QRD has been a target for hardware acceleration [6, 30, 49]. We devise the architecture of our QRD unit inspired by an existing QRD accelerator [6], which executes the Householder algorithm described in Algorithm 12. Figure 11(b) shows the microarchitecture of the QRD unit in each Training Engine. QRD unit constitutes an array of Processing Units (PUs), each of which executes a QRD. The results of PUs are concatenated and stored back to the matrix buffer for the next stage of QRD (Figure 7). Each PU first gets its input data from scratchpad memory (X_Δ or R_Δ) and stores them in the matrix buffer. Then, the outer loop in Algorithm 12 is performed at the "Outer Loop PE", which calculates the reflector and γ. These two inputs are sent to a set of "Inner Loop PEs", which are responsible for calculating $R_{new}[i][j]$ for different columns in parallel. Each "Outer Loop PE" and "Inner Loop PE" is equipped with a vector of multiply-and-accumulate (MACC) units for dot products. The resulting R_{new} matrix is sent to the scratchpad memory and replaces R_{old} for future retrainings.

6.3 Runtime Software Interface

As emphasized in Section 5.1, in designing the SIA system, we leverage a commonality of most learned indexes that they use linear regression as their backend machine learning models. This unique property enables us to build an abstraction between various learned indexes and our hardware accelerator solution. Hence, SIA could be readily adopted by any linear model-based learned indexes.

To transparently develop the abstraction and facilitate the use of underlying acceleration solution, we encapsulate the SIA accelerator along with its device driver and accelerator invocation runtime software as a library. In fact, as existing learned index systems often employ LAPACK, a famous linear algebra library, we propose SIA's interfaces to be equivalent to the LAPACK's, so that the integration of SIA with the existing systems becomes straightforward. The runtime interface of SIA includes two functions: (1) `cold_train`: a function for full model training with key matrix and key's position vector, and (2) `incre_train`: a function for incremental learning with memoization that takes the memoized R matrix as an additional argument. These two functions closely resemble the LAPACK's *gels* function, enabling existing updatable learned indexes to leverage SIA's incremental index learning algorithm and hardware acceleration with minimal software modifications.

6.4 Lazy Delete Query Handling

While this paper has focused on the `insert` query handling thus far, updatable learned indexes must be able to handle `delete` queries as well. Conventional updatable learned indexes handle these `delete` queries through retraining, similarly to the `insert` queries. In contrast, our incremental learning algorithm exploits a memoization technique, which relies on the assumption that the existing keys used to compute the memoized R matrix are not changed. Therefore, the removal of keys from the index inevitably forces SIA to discard the memoized R matrix and necessitates a cold training, which undercuts the advantages of our proposed technique.

To tackle this problem, we employ a *lazy* delete handling technique where the deleted keys are simply flagged as "deleted", yet the information of these deleted keys still remains in the memoized R matrices. This way, our incremental training method remains effective during retraining. It is important to note, however, that upon marking as "deleted", the key string and associated value data are immediately erased from the indexes for security purposes. Memoized R matrices for the "deleted" keys are eliminated during cold training, where the models are trained from scratch without utilizing the memoized matrices. Note that our lazy deletion technique does not affect the functionality of indexes, but only influences performance, since deleted-yet-unremoved information would lower the prediction accuracy and end up increasing the linear search cost for mispredicted accesses. However, we observe that lazy deletion has a marginal impact on performance, with less than 5% overhead.

6.5 Implication of Node Split and Merge

The hierarchical structure of learned index undergoes structural modifications through either *split* or *merge*, as new keys are inserted or deleted. Model *split* involves partitioning the keys assigned to a node into two nodes when the accuracy of the corresponding model drops, while model *merge* combines two nodes into one when both have sufficiently high accuracy. SIA employs the same threshold determination mechanism for split and merge as the default learned index system, without any modifications. Note that SIA should perform cold trainings for split nodes as they lack memoized R matrices, while for merged nodes, SIA can merge the R matrices and use the merged R matrix for further incremental training.

7 EVALUATION

To evaluate the effectiveness of SIA, we use two open-source benchmark suites, YCSB and Twitter cache trace, using two real-world datasets, Amazon review and MemeTracker. We evaluate throughput, system-level energy efficiency, and memory usage of SIA-accelerated learned indexes, compared to other index structures.

7.1 Methodology

YCSB. To evaluate SIA, we primarily use a real-world key-value store benchmark suite, YCSB [8]. YCSB contains six diverse workloads (A-F), each characterized by its unique mix of query types. As SIA is for updatable learned index systems, we focus on the two workloads among the six, which include *insert* queries: (D) *read latest* that tend to have `read` queries for recently inserted keys, along with roughly the 5% of `insert` queries, and (E) *short ranges* that consists of 95% `range` queries, and 5% of `insert` queries. Note that while YCSB's query compositions mirror real-world application patterns, the key lengths do not. To better emulate real-world key-value stores, we employ two genuine string datasets: Amazon review data and the MemeTracker dataset. Amazon review data (*amaz*) [44] is collected from user reviews on products from Amazon with the user IDs as keys of length 12. MemeTracker dataset (*meme*) [32] comprises quotes and phrases collected from the web and online news URLs referring to them with the URLs as keys of length 128. We use these datasets since they are widely used in prior works [61, 66, 70] to evaluate the string-key key-value stores. Additionally, we use a randomly synthesized dataset (*rand*) with a uniform key distribution.

Twitter cache trace. Complementing YCSB, we also utilize the Twitter cache trace [69] to enrich our experimental methodology.

Table 1: Hardware specifications and resource utilization of the Intel Arria 10 with the configuration of with 4 TEs, each having 2 PUs, and each PU containing 3 Outer Loop PEs.

		ALM	RAM Blk	PLL	Blk RAM	DSP
Intel Arria 10 GX-1150	Used	278,759	2,123	48	4,648,464	244
	Total	427,200	2,713	176	55,562,240	1,518
	Utilization	65.3%	78.3%	27.3%	8.4%	16.1%

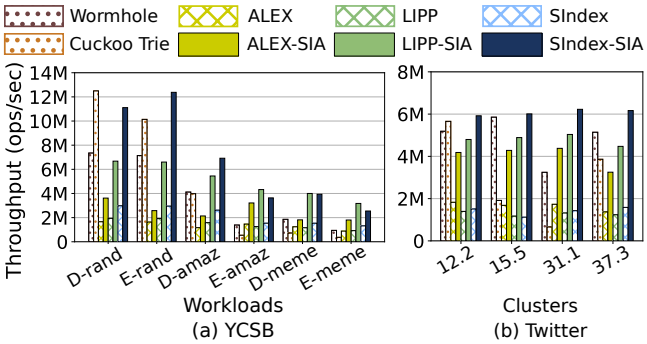

Figure 12: Throughput comparison of *non*-learned (conventional) and learned indexes for YCSB and Twitter cache trace.

Twitter cache trace constitutes a pile of indexing traces collected from Twitter clusters, which allows it to concurrently serve as a workload and a dataset. Among the provided 54 cluster traces, we specifically select four cluster traces with a relatively significant volume of update queries, each of which exhibits a distinct query composition, represented by the following tuples of (cluster ID, update query ratio): (12.2, 43%), (15.5, 59%), (31.1, 56%), (37.3, 42%).

Baselines. As baselines, we use three state-of-the-art learned indexes, ALEX [11], LIPP [64], and SIndex [61], all of which are chosen for their open-source implementations available at our disposal. We added the variable-length string key and multi-threading support on top of ALEX and LIPP, as they lack the features. We built their corresponding SIA-accelerated counterparts by integrating the implementation with our SIA library. Note that while the three systems have disparities in how to initially build the indexes through bulk loading (e.g., top-down vs. bottom-up), it does not affect our performance evaluations because the index building only requires cold retrainings, which cannot exploit the proposed incremental index learning algorithm.

Furthermore, we include comparisons between SIA-accelerated learned indexes and two state-of-the-art *non*-learned indexes, Wormhole [66] and Cuckoo Trie [71], all of which support variable-length string keys. Wormhole [66] is an optimized B-tree in which part of the tree is replaced with a trie utilizing hashes. Cuckoo Trie [71] is a hash-based trie index that achieves high performance through overlapping memory accesses.

System specifications. The SIA-accelerated learned index systems are equipped with a 16-core Intel Xeon Gold 6226R and 128 GB DRAM. For building SIndex-GPU, GPU-accelerated variant of SIndex, we employ NVIDIA GeForce RTX 2080 TI GPU along with the same CPU and memory configuration. SIndex-GPU uses CuSolver library in CUDA version 11.7. For the runtime measurement of the baseline learned index systems, we use a highly-optimized, parallel linear algebra library, Intel Math Kernel Library (MKL) 2019.0.

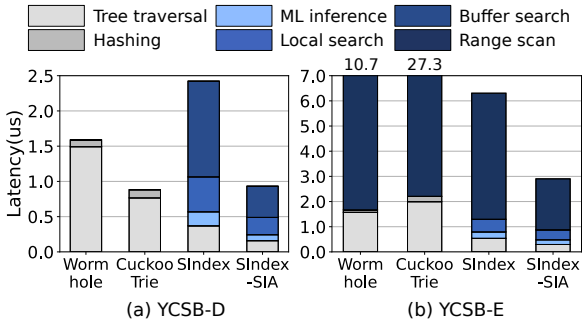

Figure 13: Latency breakdown for YCSB D/E workloads using *rand* dataset. Range scan includes buffer search for YCSB-E.

FPGA platform details. Table 1 shows the hardware resource specification of the evaluated FPGA, Intel Arria 10 GX-1150, and its utilization when we program our accelerator on it. We develop a custom accelerator controller on the programmable logic to interface with the device's main memory. We synthesize the hardware with Quartus II v20.1, and achieve a frequency of 272 MHz.

Power measurement. To measure the end-to-end system power, we use an off-the-shelf power meter, WATTMAN HPM-100A [2]. This power meter is placed between the power outlets and the server, which are configured with various processor combinations, including CPU-only, CPU-GPU, and CPU-FPGA setups. The measured power can be monitored per each second through the vendor-provided software, which we average over the experiment runtime.

7.2 Experimental Results

7.2.1 Throughput. Figure 12 shows the throughput comparison results among two *non*-learned indexes (Wormhole and Cuckoo Trie), three learned indexes (ALEX, LIPP, SIndex), and their SIA-accelerated counterparts (ALEX-SIA, LIPP-SIA, SIndex-SIA).

YCSB results. Figure 12(a) illustrates the results using two YCSB workloads across three datasets: *rand*, *amaz*, and *meme*. Although there is some variability in the results, we observe a consistent trend that the SIA-accelerated indexes outperform the learned index baselines, as well as the conventional, *non*-learned index baselines. This translates to approximately an average 2.6× throughput improvement over CPU-only learned index systems. This substantial enhancement is attributed to SIA's utilization of both iterative learning algorithm and customized hardware accelerator. This approach dedicates the majority of CPU cores to inferences, while the system allocates only one training thread for memory-bound kernels and accelerator management, not performing any expensive operations.

Twitter cache trace results. Figure 12(b) reports the throughput results for Twitter cache trace. Twitter cache trace has diverse key lengths that range from 19 to 82. As the key length directly affects the computational load, there are variations among clusters in the throughput results. On average, the SIA-accelerated learned indexes offer 3.4× throughput improvement over CPU-only systems,

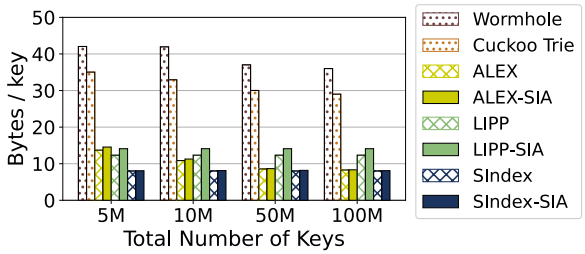

Figure 14: Memory consumption of traditional (non-learned) indexes, baseline learned indexes, and learned indexes with SIA. Key and value data is excluded.

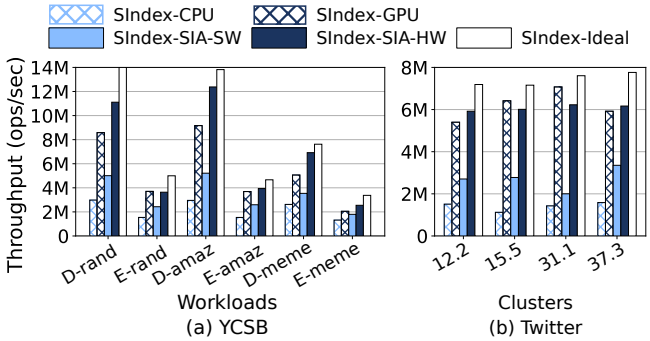

Figure 15: Ablation study results using SIndex variants.

representing a more substantial performance improvement than observed in the YCSB scenario. The larger gain comes from that the dataset of Twitter cache trace has generally longer keys, making the key matrix larger, which can be better parallelized by the accelerator. Overall, the results suggest that SIA is an effective solution for enabling updatable string-key learned indexes without suffering from performance bottlenecks caused by training computations.

7.2.2 Query Latency. To understand the source of performance improvements, we further analyze the query latency for YCSB workload (D) and (E), and present the breakdown results in Figure 13. Non-learned indexes, Wormhole and Cuckoo Trie, require traversal through their tree structures, which often involve multiple DRAM accesses, leading to high query latency. In contrast, learned indexes (SIndex and SIndex-SIA) require much fewer memory accesses for graph traversal. In fact, the depth of hierarchical learned index structure of SIndex is only two, which imposes significantly lower memory access overhead than the alternatives. As the cost of these benefits, the learned indexes must pay other costs such as ML inference, local search in case of misprediction, and buffer search for seeking the "not-yet-trained" keys. The outcomes of the study reveal that the buffer search is the largest overhead, especially for SIndex, because it piles up a large number of keys in the buffer due to slow retraining. On the contrary, SIA accelerates the retrainings and frequently empties the buffers of SIndex-SIA, which substantially reduces the buffer search latency, directly leading to the total latency reduction.

7.2.3 Memory Usage. Figure 14 reports the memory usage of five baselines (learned and *non*-learned) and three SIA-accelerated learned indexes. To specifically assess the memory usage difference among the indexes, we exclusively measure the memory usage for indexes, not key and values. Learned indexes typically require significantly less memory because they efficiently compress the key-position mapping information from hierarchical data structures into a series of compact machine learning models. SIA incurs marginal overhead in memory usage as it must additionally store the R matrices for memoized computation. However, the average overhead measured in our experiments is merely 6.0%, which is negligible and justifiable with the significant performance improvements.

7.2.4 Ablation Study. For a more thorough analysis of the factors contributing to performance improvements, we focus on the SIA-accelerated SIndex and conduct an ablation study. Figure 15 compares the throughput of the five SIndex variants. SIndex-GPU is a system offloading retraining to GPU, while SIndex-Ideal is a system equipped with an infinitely fast accelerator that trains models in zero time. On the other hand, SIndex-SIA-SW and SIndex-SIA-HW are the SIA-accelerated SIndex systems with algorithm-only and algorithm-hardware co-designed SIA solutions, respectively. SIndex-GPU achieves 2.3× throughput improvement compared to the default CPU baseline, SIndex-CPU. While SIndex-SIA-SW offers 1.7× improvement over SIndex-CPU, the benefit is 56.5% lower than that of SIndex-GPU, which demonstrates the limitation of the software-only solution. However, SIndex-SIA-HW achieves 2.0× additional improvement over SIndex-SIA-SW, closely approaching to SIndex-Ideal, 11.6% higher than what SIndex-GPU offers, which presents the effectiveness of hardware acceleration. These results show the effectiveness and necessity of SIA as a solution that synergizes algorithm and hardware designs for acceleration.

7.2.5 System Power Consumption. We choose FPGA due to its capability to tailor the hardware architecture for the given task, incremental training, delivering notably higher energy efficiency compared to GPU. Figure 16 illustrates the system-level power consumption of SIndex variants: SIndex-CPU, SIndex-GPU, and SIndex-SIA, which demonstrates the advantages of FPGA acceleration in power efficiency. We observe that the CPU-only system, SIndex-CPU, operates at 150W, with a significant portion of this power attributed to CPU-based training. SIndex-GPU operates at 203W, dissipating 79W for training at GPU and the remaining 123W for the CPU-based system. In contrast, SIndex-SIA, a CPU-FPGA heterogeneous system, consumes only 126W as the FPGA accelerator adds only 3W to the CPU-only system, which demonstrates the power efficiency of the FPGA.

7.2.6 Throughput-per-watt. As noted in the power consumption analysis, if we only consider the accelerator itself instead of the entire system, FPGA offers 28× less power consumption compared to GPU. However, when we consider the system-level power consumption with their throughput together, SIndex-SIA achieves only 1.76× higher throughput-per-watt compared to SIndex-GPU. While the gain may be deemed modest, the 76% gap could translate into substantial cost disparities in terms of actual monetary expenditure since index systems tend to remain operational continuously,

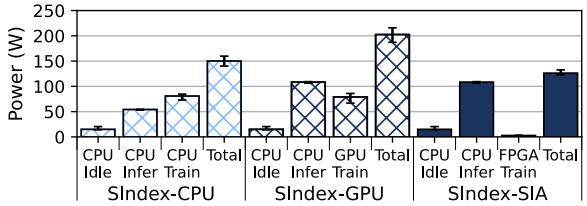

Figure 16: Average power consumption of SIndex-CPU, SIndex-GPU, and SIndex-SIA end-to-end systems. Vertical lines indicate minimum and maximum power consumption.

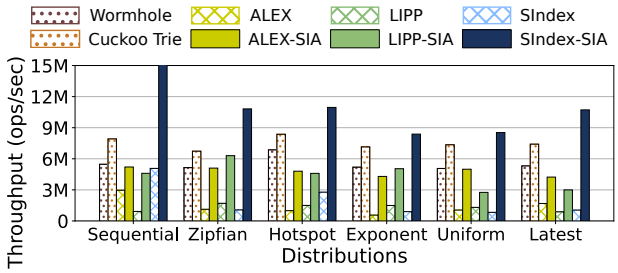

Figure 17: Throughput of *non*-learned and learned indexes for queries with different request distributions.

consistently dissipating considerable amounts of energy. These results suggest that for the given task, the continuous retrainings of updatable learned indexes, FPGA is a more attractive option as an acceleration platform compared to GPU.

7.2.7 Implication of Request Distributions. Figure 17 illustrates the throughput of each index across six different request distributions as used in prior works [5, 10]: *sequential*, *zipfian*, *hotspot*, *exponent*, *uniform* and *latest*. Across all query distributions, learned indexes accelerated with SIA consistently show significant performance improvement, which ranges from 3.9× to 6.2× compared to the baselines. Note that *zipfian*, *hotspot*, and *exponent* distributions exhibit skewed patterns, resulting in certain key ranges being accessed more frequently than others, causing more node splits. As node splits trigger cold trainings, it imposes performance overhead, while we observe that its impact on the end throughput is negligible.

7.2.8 Implication of Lazy Delete Query Handling. We analyze the impact of lazy delete query handling on the performance of SIA-accelerated learned indexes. We configure the cold training interval to various durations: 5, 30, 100, and 300 seconds, and sweep the `delete` query ratio from 5% to 15%, filling the remaining queries with `read` queries. We observe that at a deletion ratio of 5%, there is a performance degradation of 3.2% when the training interval is 300 seconds. Meanwhile, with a deletion ratio of 10% and 15%, the larger number of unhandled keys results in a greater performance loss, which increases up 4.1% to 4.6%, respectively. Nonetheless, the performance degradation remains at a marginal level, which validates the viability of the lazy approach, particularly considering the significant costs associated with complete cold training.

8 ADDITIONAL RELATED WORK

Learned index structures. There has been a large body of prior works [1, 9, 10, 12, 13, 26, 34, 41, 43, 54, 64, 73, 75, 76] for learned index systems. RadixSpline [26] and PLEX [54] further optimize learned index construction. Flood [43] and Tsunami [12] exploit the learning approach for multi-dimensional indexes to automatically optimize the index structure for the given data and query distributions. On the other hand, SIA optimizes training via memoized QRD algorithm enhanced by an accelerator and builds a system for integration with learned index structure.

Learned index acceleration. Colin [75] builds and manages CPU cache-friendly learned index structure on top of PGM-index, with performing key insertions in place to better utilize the caching. Anderson et al. [4] perform microarchitectural analysis of ALEX on commodity CPU and show the impact of memory hierarchy on read/write latency. Unlike these works that aim to benefit from microarchitectural optimizations on the CPU, SIA devises iterative QRD to leverage computation reuse and further enhances the index system by offloading the training process to a separate accelerator.

QR decomposition accelerator. As the QR decomposition makes up an essential building block of many modern applications, several architectural design for accelerators has been studied in the literature [6, 30]. Although the QRD unit is motivated from past works, none of them use these in the context of learned index systems. Moreover, SIA's accelerator is designed to execute multi-dimensional parallelism in the context of retraining models in learned indexes, while QRD accelerator is a small function unit.

9 CONCLUSION

This work offers SIA, an accelerated string-key learned index system. These index structures require constant retraining of their machine learning models to determine the mapping between keys and their positions. SIA mitigates the bottleneck of the current systems that incur huge overhead of training when the keys are updated. Training observes multi-fold issues, where it is inefficient to execute on the CPU, is serial across runs as it writes to the model, and cannibalizes CPU resources from inference queries. Based on these insights, SIA enhances the learned index training by leveraging the mathematical property that keys can be updated incrementally, and thus, can benefit from computation reuse via memoization. SIA further boosts this training on an energy-efficient FPGA accelerator and relieves CPU resources for inference, collaboratively offering significant speedup.

ACKNOWLEDGMENTS

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) (No.2024-00396013, No.2022-0-01037, No.2018-0-00503) under the Graduate School of Artificial Intelligence Semiconductor (IITP-2024-RS-2023-00256472), Information Technology Research Center (ITRC) support program (IITP-2024-2020-0-01795), and Artificial Intelligence Graduate School Program (KAIST) (No.2019-0-00075), all funded by the Korea government (MSIT).

REFERENCES

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

[2] ADpower. 2023. Wattman (HPM-100A). http://adpower21com.cafe24.com/shop2/product/wattman-hpm-100a/17.

[3] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. 2016. ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys. *IEEE Trans. Comput.* 65, 3 (2016), 902–915.

[4] Mikkel Møller Andersen and Pinar Tözün. 2022. Micro-Architectural Analysis of a Learned Index. In *Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Philadelphia, Pennsylvania) *(aiDM '22)*. Association for Computing Machinery, New York, NY, USA, Article 5, 12 pages. https://doi.org/10.1145/3533702.3534917

[5] Esmail Asyabi, Yuanli Wang, John Liagouris, Vasiliki Kalavri, and Azer Bestavros. 2022. A New Benchmark Harness for Systematic and Robust Evaluation of Streaming State Stores. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 559–574. https://doi.org/10.1145/3492321.3519592

[6] Jose M. Rodriguez Borbon, Junjie Huang, Bryan M. Wong, and Walid Najjar. 2021. Acceleration of Parallel-Blocked QR Decomposition of Tall-and-Skinny Matrices on FPGAs. *ACM Trans. Archit. Code Optim.* 18, 3, Article 27 (may 2021), 25 pages. https://doi.org/10.1145/3447775

[7] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3393–3404.

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

[9] Andrew Crotty. 2021. Hist-Tree: Those Who Ignore It Are Doomed to Learn. In *Conference on Innovative Data Systems Research (CIDR'21)*.

[10] 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 *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 9, 17 pages.

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

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

[13] 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 (apr 2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[14] L. FOX, H. D. HUSKEY, and J. H. WILKINSON. 1948. NOTES ON THE SOLUTION OF ALGEBRAIC LINEAR SIMULTANEOUS EQUATIONS. *The Quarterly Journal of Mechanics and Applied Mathematics* 1, 1 (01 1948), 149–173. https://doi.org/10.1093/qjmam/1.1.149 arXiv:https://academic.oup.com/qjmam/article-pdf/1/1/149/5322943/1-1-149.pdf

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

[16] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. 1990. Parallel Algorithms for Dense Linear Algebra Computations. *SIAM Rev.* 32, 1 (mar 1990), 54–135. https://doi.org/10.1137/1032002

[17] Jian Gao, Xin Cao, Xin Yao, Gong Zhang, and Wei Wang. 2023. LMSFC: A Novel Multidimensional Index Based on Learned Monotonic Space Filling Curves. *Proc. VLDB Endow.* 16, 10 (aug 2023), 2605–2617. https://doi.org/10.14778/3603581.3603598

[18] Gene H. Golub and Charles F. Van Loan. 1996. *Matrix Computations* (third ed.). The Johns Hopkins University Press.

[19] Tim Gubner, Viktor Leis, and Peter Boncz. 2021. Optimistically Compressed Hash Tables & Strings in TheUSSR. *SIGMOD Rec.* 50, 1 (jun 2021), 60–67. https://doi.org/10.1145/3471485.3471500

[20] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning Memory Access Patterns. arXiv:1803.02329 http://arxiv.org/abs/1803.02329

[21] Steffen Heinz, Justin Zobel, and Hugh E. Williams. 2002. Burst Tries: A Fast, Efficient Data Structure for String Keys. *ACM Transactions on Information Systems* 20, 2 (2002), 902–915.

[22] Don Heller. 1978. A Survey of Parallel Algorithms in Numerical Linear Algebra. *SIAM Rev.* 20, 4 (1978), 740–777. https://doi.org/10.1137/1020096

[23] Alston S. Householder. 1958. Unitary Triangularization of a Nonsymmetric Matrix. *J. ACM* 5, 4 (oct 1958), 339–342. https://doi.org/10.1145/320941.320947

[24] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind Arvind, and Sungjin Lee. 2020. PinK: High-Speed in-Storage Key-Value Store with Bounded Tails. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 12, 15 pages.

[25] Intel. 2023. Intel FPGA. https://www.intel.com/content/www/us/en/products/details/fpga.html.

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

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

[28] Branimir Lambov. 2022. Trie Memtables in Cassandra. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3359–3371. https://doi.org/10.14778/3554821.3554828

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

[30] Martin Langhammer and Bogdan Pasca. 2018. High-Performance QR Decomposition for FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 183–188. https://doi.org/10.1145/3174243.3174273

[31] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

[32] Jure Leskovec, Lars Backstrom, and Jon Kleinberg. 2009. Meme-Tracking and the Dynamics of the News Cycle. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Paris, France) *(KDD '09)*. Association for Computing Machinery, New York, NY, USA, 497–506. https://doi.org/10.1145/1557019.1557077

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

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

[35] 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 (jul 2023), 2212–2224. https://doi.org/10.14778/3598581.3598593

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

[37] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2071–2086. https://doi.org/10.1145/3318464.3389731

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

[39] Divya Mahajan, Joon Kyung Kim, Jacob Sacks, Adel Ardalan, Arun Kumar, and Hadi Esmaeilzadeh. 2018. In-RDBMS hardware acceleration of advanced analytics. *Proc. VLDB Endow.* 11, 11 (jul 2018), 1317–1331. https://doi.org/10.14778/3236187.3236188

[40] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. 2016. TABLA: A unified

template-based framework for accelerating statistical machine learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 14–26. https://doi.org/10.1109/HPCA.2016.7446050

[41] 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 (sep 2020), 1–13. https://doi.org/10.14778/3421424.3421425

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

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

[44] Jianmo Ni, Jiacheng Li, and Julian McAuley. 2019. Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, Hong Kong, China, 188–197. https://doi.org/10.18653/v1/D19-1018

[45] Nvidia. 2023. Nvidia Flex NIC with FPGA. https://www.nvidia.com/en-us/networking/ethernet/innova-2-flex/.

[46] Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. 2017. Scale-Out Acceleration for Machine Learning. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 367–381.

[47] Robert J Plemmons. 1988. *Parallel Block Schemes for Large-Scale Least-Squares Computations*. Urbana: University of Illinois Press.

[48] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2015. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *IEEE Micro* 35, 3 (2015), 10–22. https://doi.org/10.1109/MM.2015.42

[49] Abid Rafique, Nachiket Kapre, and George A. Constantinides. 2012. Enhancing performance of Tall-Skinny QR factorization using FPGAs. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 443–450. https://doi.org/10.1109/FPL.2012.6339142

[50] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783720

[51] Yufan Sheng, Xin Cao, Yixiang Fang, Kaiqi Zhao, Jianzhong Qi, Gao Cong, and Wenjie Zhang. 2023. WISK: A Workload-Aware Learned Index for Spatial Keyword Queries. *Proc. ACM Manag. Data* 1, 2, Article 187 (jun 2023), 27 pages. https://doi.org/10.1145/3589332

[52] Jiachen Shi, Gao Cong, and Xiao-Li Li. 2022. Learned Index Benefits: Machine Learning Based Index Performance Estimation. *Proc. VLDB Endow.* 15, 13 (sep 2022), 3950–3962. https://doi.org/10.14778/3565838.3565848

[53] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. 2021. Bounding the Last Mile: Efficient Learned String Indexing. arXiv:2111.14905 [cs.DB]

[54] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. Towards Practical Learned Indexing. arXiv:2108.05117 [cs.DB]

[55] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. 2023. LeaFTL: A Learning-Based Flash Translation Layer for Solid-State Drives. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 442–456. https://doi.org/10.1145/3575693.3575744

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

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

[58] Yulai Tong, Jiazhen Liu, Hua Wang, Ke Zhou, Rongfeng He, Qin Zhang, and Cheng Wang. 2023. Sieve: A Learned Data-Skipping Index for Data Analytics. *Proc. VLDB Endow.* 16, 11 (jul 2023), 3214–3226. https://doi.org/10.14778/3611479.3611520

[59] Shengzhe Wang, Zihang Lin, Suzhen Wu, Hong Jiang, Jie Zhang, and Bo Mao. 2023. LearnedFTL: A Learning-based Page-level FTL for Improving Random Reads in Flash-based SSDs. arXiv:2303.13226 [cs.AR]

[60] Yifan Wang, Haodi Ma, and Daisy Zhe Wang. 2022. LIDER: An Efficient High-Dimensional Learned Index for Large-Scale Dense Passage Retrieval. *Proc. VLDB Endow.* 16, 2 (oct 2022), 154–166. https://doi.org/10.14778/3565816.3565819

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

[62] Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901. https://doi.org/10.1109/JPROC.2018.2817118

[63] Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. 2021. XStore: Fast RDMA-Based Ordered Key-Value Store Using Remote Learned Cache. *ACM Trans. Storage* 17, 3, Article 18 (aug 2021), 32 pages. https://doi.org/10.1145/3468520

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

[65] 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 (jun 2022), 2188–2200. https://doi.org/10.14778/3547305.3547322

[66] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A Fast Ordered Index for In-Memory Data Management. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 18, 16 pages. https://doi.org/10.1145/3302424.3303955

[67] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. 2021. Parallax: Hybrid Key-Value Placement in LSM-Based Key-Value Stores. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 305–318. https://doi.org/10.1145/3472883.3487012

[68] Jin Yang, Heejin Yoon, Gyeongchan Yun, Sam H. Noh, and Young-ri Choi. 2023. DyTIS: A Dynamic Dataset Targeted Index Structure Simultaneously Efficient for Search, Insert, and Scan. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 800–816. https://doi.org/10.1145/3552326.3587434

[69] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 11, 18 pages.

[70] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: An Update-in-Place Key-Value Store for Modern Storage. *Proc. VLDB Endow.* 16, 1 (sep 2022), 99–112. https://doi.org/10.14778/3561261.3561270

[71] Adar Zeitak and Adam Morrison. 2021. Cuckoo Trie: Exploiting Memory-Level Parallelism for Efficient DRAM Indexing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 147–162. https://doi.org/10.1145/3477132.3483551

[72] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: a cache-aware learned index with a cost-based construction algorithm. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2679–2691. https://doi.org/10.14778/3551793.3551823

[73] Songnian Zhang, Suprio Ray, Rongxing Lu, and Yandong Zheng. 2021. SPRIG: A Learned Spatial Index for Range and KNN Queries. In *17th International Symposium on Spatial and Temporal Databases* (virtual, USA) *(SSTD '21)*. Association for Computing Machinery, New York, NY, USA, 96–105. https://doi.org/10.1145/3469830.3470892

[74] 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 (oct 2022), 243–255. https://doi.org/10.14778/3565816.3565826

[75] Zhou Zhang, Peiquan Jin, Xiaoliang Wang, Yanqi Lv, Shouhong Wan, and Xike Xie. 2021. COLIN: A Cache-Conscious Dynamic Learned Index with High Read/Write Performance. *Journal of Computer Science and Technology* 36 (2021), 721–740.

[76] Zejian Zhang, Yan Wang, and Shunzhi Zhu. 2021. LIDUSA – A Learned Index Structure for Dynamical Uneven Spatial Data. In *Algorithms and Architectures for Parallel Processing: 21st International Conference, ICA3PP 2021, Virtual Event, December 3–5, 2021, Proceedings, Part III*. Springer-Verlag, Berlin, Heidelberg, 737–753. https://doi.org/10.1007/978-3-030-95391-1_46