DALdex: A DPU-Accelerated Persistent Learned Index via Incremental Learning

Aoyang Tong
Huazhong University of Science and Technology
Wuhan, China
aoyangtong@hust.edu.cn

Yu Hua*
Huazhong University of Science and Technology
Wuhan, China
csyhua@hust.edu.cn

Menglei Chen
Huazhong University of Science and Technology
Wuhan, China
chenml@hust.edu.cn

Abstract

The Data Process Unit (DPU) specializes in offloading CPU-intensive tasks and provides efficient fault tolerance through hardware-level isolation. This brings unique opportunities to develop persistent indexes with high performance and availability in High Performance Computing (HPC) systems. The recent learned index exploits machine learning models to efficiently fit data distributions, exhibiting superior performance and low storage costs, which is a promising alternative to traditional tree-based range indexes. However, state-of-the-art persistent learned indexes suffer from costly model retrainings and inefficient recovery mechanisms based on Non-Volatile Memory (NVM), making them inefficient to be offloaded to DPUs.

To address these challenges, we propose DALdex, a CPU-DPU hybrid persistent learned index with high performance and availability. To mitigate model retraining overheads, DALdex offloads retraining tasks to DPU based on the incremental learning scheme. To minimize NVM amplifications, DALdex designs an NVM-friendly index structure that is decoupled into a DRAM-accelerated model layer and an NVM-aware block layer. Moreover, DALdex utilizes the hardware isolation feature of DPU to achieve seamless failover and instant recovery via the PCIe bus. Extensive evaluation results demonstrate that DALdex outperforms state-of-the-art persistent indexes by 1.07-6.34× with minimal DRAM and NVM overheads. The open-source code of DALdex is available at https://github.com/CitySkylines/DALdex.

*Corresponding Author: Yu Hua (csyhua@hust.edu.cn).

ICS '25, Salt Lake City, USA

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts

• **Information systems** → **Data structures**; • **Computer systems organization**;

Keywords

Data Process Unit, Learned Index, Non-Volatile Memory

ACM Reference Format:
Aoyang Tong, Yu Hua, and Menglei Chen. 2025. DALdex: A DPU-Accelerated Persistent Learned Index via Incremental Learning. In *Proceedings of the 39th ACM International Conference on Supercomputing(ICS '25), June 8–11, 2025, Salt Lake City, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The in-memory index is a crucial component in HPC systems for efficient data reads and writes [27]. However, traditional tree-based range indexes exhibit inefficient index operations and large memory consumption due to sophisticated structures [33, 57]. For example, state-of-the-art tree-based range indexes consume up to 55% of the total memory in popular HPC systems [70], which presents significant challenges to DRAM capacity. Moreover, due to the existence of the memory wall [62], DRAM exhibits limited scalability in bandwidth as well as capacity, which becomes insufficient for in-memory indexing in HPC systems.

Byte-addressable NVM (e.g., PCM [6, 29, 59], STT-RAM [3], ReRAM [1] and Intel Optane DC PMEM [19]) offers persistent storage, large capacity, and DRAM-comparable access latency, which can efficiently alleviate memory bottlenecks at a low cost. Therefore, there arise plenty of research works and practical applications on NVM, including indexes [4, 7, 8, 18, 36, 38, 46, 65, 74], key-value stores [9, 21, 23, 52] and file systems [20, 66, 72], etc. These works typically employ traditional tree-based range indexes with several NVM-optimized techniques such as unsorted leaf nodes [7], bitmaps [36], fingerprints [69], and selective consistency mechanisms [46, 65], etc. However, due to the unawareness of data distribution patterns, traditional tree-based range indexes fail to flexibly design the internal tree structure, thus leading to excessive NVM accesses in dynamic scenarios.

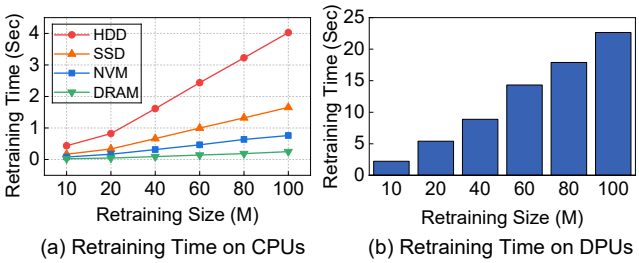

Figure 1: Model retraining time across various sizes.

The **Learned Index** leverages machine learning models to predict the position of a given key in sorted data nodes, offering high performance and low storage costs [27]. By fitting data distribution patterns, learned indexes can efficiently locate the target key-value pair without sophisticated structures. Therefore, learned indexes typically hold huge data nodes with a flat index structure of two or three layers. However, learned indexes are initially designed for volatile DRAM, making it inefficient to simply migrate them to NVM without further optimizations. Specifically, designing an efficient persistent learned index with high performance and availability based on NVM poses the following challenges:

Challenge 1: Expensive Model Retrainings. Learned indexes necessitate repeated model retrainings to preserve model accuracy, as data distributions constantly change with new insertions. However, the retraining process is extremely time-consuming, especially on low-speed storage devices. As shown in Figure 1(a), the model retraining time dramatically grows as the data size increases. For example, retraining a linear regression model with 100M data on disks consumes up to 4.1 seconds, which is about 16.1× than DRAM. Such delay would significantly block normal index operations and lead to high indexing latency. Therefore, reducing model retraining overheads is a key challenge for learned indexes to reduce indexing latency and improve performance.

Challenge 2: Excessive NVM Accesses. Byte-addressable NVM exhibits lower performance metrics than DRAM. For example, Intel Optane DC PMEM exhibits 2–3× higher access latency and 3–14× lower bandwidth compared to DRAM [37, 63, 64]. Therefore, excessive NVM accesses would rapidly saturate the limited NVM bandwidth and significantly decrease system performance. As a result, persistent learned indexes need to carefully manage NVM accesses, especially model accesses on the critical path of learned index operations. However, state-of-the-art persistent learned indexes fail to address this challenge due to inefficient management of learned models. For example, APEX [37] is a DRAM-NVM hybrid persistent learned index that stores learned models and key-value pairs in NVM for persistence, and stores partial metadata in DRAM for performance. PLIN [71] is an NVM-only persistent learned index that stores all components including learned models in NVM to achieve instant recovery. However, both schemes have to store learned models in NVM, resulting in excessive model accesses to NVM on the critical path.

Challenge 3: Inefficient Recovery Mechanisms. System crashes are among the most common failures in real-world applications due to various software bugs or hardware damages [12, 28, 68]. Therefore, persistent indexes are widely employed in HPC systems to enhance availability by instantly recovering from system crashes. However, state-of-the-art persistent learned indexes suffer from inefficient recovery mechanisms by rebuilding partial index structures or redoing heavy NVM logs [37, 71], which significantly increase the Mean Time To Repair (MTTR) across system crashes.

To address these challenges, we incorporate DPU into the persistent learned index to achieve high performance and availability. DPUs are equipped with dedicated computing and memory resources bypassing CPUs, making them efficient for offloading and accelerating CPU-intensive tasks [22, 58], especially model retraining tasks in persistent learned indexes. Besides, DPUs operate in independent systems isolated from CPUs, enabling DPUs to remain alive and continue running during CPU-side system crashes [22]. Therefore, by offloading the index structure to DPU during system crashes, the availability of persistent learned indexes can be further improved. Based on these unique hardware features of DPUs, we propose **DALdex**, a **D**PU-**A**ccelerated Persistent **L**earned In**dex** via Incremental Learning.

To mitigate model retraining overheads, we offload retraining tasks to DPU based on the incremental learning scheme, which removes model retrainings from the critical path. Unlike existing offline batched model retraining schemes [11, 13, 14, 32, 34, 37, 54, 61, 71], the incremental learning scheme dynamically retrains new models based on old models with low overheads, enabling DALdex to adaptively learn new data distributions during runtime.

To minimize NVM accesses, we decouple DALdex into a DRAM-accelerated model layer and an NVM-aware block layer, which frees fast model accesses from slow NVMs. The space-efficient model structure is maintained in DRAM to maximize its performance benefits. Meanwhile, to minimize NVM amplifications, we design an NVM-friendly block structure that mitigates the mismatched access granularity between the CPU cache and NVM medium.

To further enhance availability, we utilize the hardware isolation feature of DPU to achieve seamless failover and instant recovery. In DALdex, the model retraining tasks are offloaded to DPU for incremental learning. Therefore, the offloaded model structure on the DPU side naturally serves as a backup of the CPU side. By promoting the offloaded model structure to the primary structure, DALdex can be entirely offloaded to DPU, allowing seamless failover during CPU-side system crashes. Besides, upon system restores,

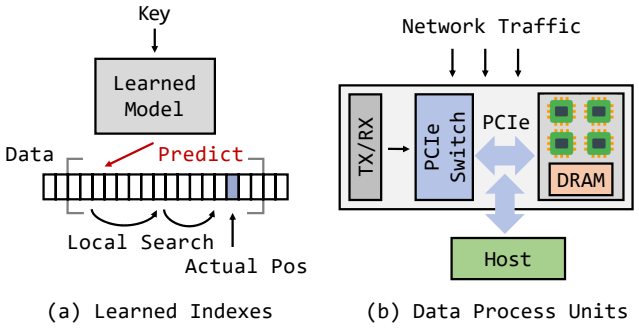

Figure 2: The structures of Learned Indexes and DPUs.

DALdex can be instantly recovered from the offloaded model structure on the DPU side through DMA via the PCIe bus.

In summary, we make the following contributions:

- We identify the bottleneck of existing model retraining schemes in the offline batched training process and propose an incremental learning scheme to mitigate model retraining overheads for learned indexes.
- We offload the incremental learning scheme to DPU to remove model retrainings from the critical path and utilize the hardware isolation feature of DPU to achieve seamless failover and instant recovery across system crashes.
- We analyze the inefficient NVM access patterns of learned models in state-of-the-art persistent learned indexes and design an NVM-friendly index structure to minimize NVM amplifications.

Our evaluation demonstrates that DALdex outperforms state-of-the-art persistent indexes by 1.07-6.34× with minimal DRAM and NVM overheads. Besides, DALdex achieves instant recovery in 0.5 seconds on a single thread. We also integrate DALdex into Redis [49] and achieve the best read and write performance compared to existing persistent learned indexes. To the best of our knowledge, DALdex is the first persistent learned index with the DPU-offloaded scheme.

2 Background

2.1 Learned Index

Learned indexes leverage machine learning models to learn the mapping from keys to data positions by fitting the Cumulative Distribution Function (CDF) of the input data sequence [27]. As illustrated in Figure 2(a), given a target key, learned indexes first leverage machine learning models to predict its approximate position in the sorted data node, and then locally search to locate the actual position. In general, learned indexes adopt the linear regression algorithm (i.e., $pos = a*key+b$ where a refers to the slope and b refers to the intercept) to train machine learning models [11, 14, 37, 71].

However, this retraining scheme is inefficient due to its offline batched training process. Specifically, during model retrainings, learned indexes need to process the entire training data in a single batch, involving massive data accesses to the memory device. Such repeated accesses would rapidly exhaust memory bandwidth and lead to substantial I/O overheads. Moreover, due to severe data dependencies between index operations and learned models, reads and writes are completely blocked until new models are fully retrained [37]. Therefore, reducing model retraining overheads is a key challenge to improve the performance of learned indexes.

Table 1: Hardware Configuration of BlueField-2 [43].

Component	Hardware Configuration
Cores	8× ARMv8 A72 processors (64-bit)
DRAM	16GB on-board DDR4-1600 DRAM
Network	1× Ethernet port 200Gb/s
PCIe	8× PCIe Gen 4.0

2.2 Data Process Unit

DPU is an off-path SmartNIC that is located out of the critical path of data transmissions in the network, as depicted in Figure 2(b). By processing in-network packets bypassing CPUs at a low cost, DPU can eliminate extra data movements in the network, thus improving resource utilization and power efficiency in HPC systems. In general, DPU is connected to the CPU via the PCIe bus, allowing fast data exchange through DMA. Besides, DPU runs in a full network stack with a dedicated network interface. By leveraging NVIDIA DOCA SDK tools [45], DPU can communicate with CPUs through a secure and network-independent DOCA communication channel [44]. Moreover, DPU runs in a popular Linux, making it easy for developers to deploy and develop applications. Therefore, applications based on DPUs can be easily migrated across various DPUs with high portability.

The unique off-path architecture of DPU makes it efficient for offloading and accelerating CPU-intensive tasks [15, 16]. However, due to the limited onboard computing and memory resources as listed in Table 1, DPU is insufficient to handle heavy offloaded tasks [17]. Therefore, developers need to carefully design a lightweight offloading scheme to exploit the capabilities of DPUs. Besides, the hardware-level isolation between the CPU and DPU enables the DPU to remain alive and continue running during CPU-side system crashes. Therefore, DPU can provide additional fault tolerance for CPUs and enhance the availability of DPU-driven HPC architecture across system crashes.

2.3 Non-Volatile Memory

Byte-addressable NVM offers storage-like persistence and DRAM-like performance, and can be easily integrated into

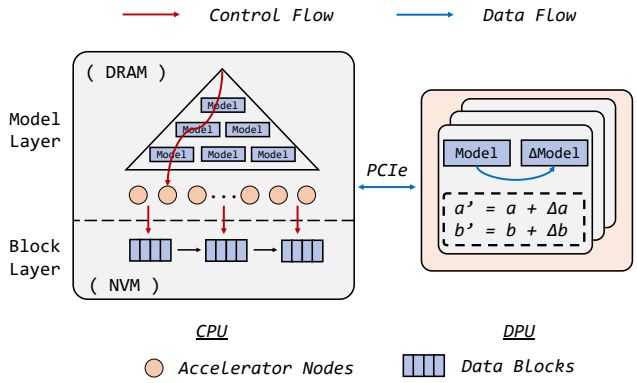

Figure 3: The structure of DALdex.

HPC systems without complicated configurations [19]. In general, NVM is installed into a memory socket via the memory bus, allowing the CPU to directly access it with *load/store* instructions. However, programming based on NVM is not straightforward due to the following reasons. (1) Data in the CPU cache is volatile and writes may be reordered by CPUs. Therefore, developers need to explicitly flush (CLWB, CLFLUSH, or CLFLUSHOPT) the CPU cacheline followed by memory fence instructions (SFENCE or MFENCE) to ensure writes actually reach the NVM medium. (2) The data access granularity in the CPU cache does not match the NVM medium. For example, Intel Optane DC PMEM accesses data at the granularity of an XPLine (256B) in the 3DXPoint medium. Therefore, small reads or writes in a CPU cacheline (64B) would trigger a 4× amplification to the 3DXPoint medium. This mismatch would rapidly saturate NVM bandwidth during runtime and significantly decrease system performance. Hence, it is necessary to design an NVM-friendly index structure to minimize NVM amplifications.

3 DALdex Design

3.1 Overview

DALdex is a CPU-DPU hybrid persistent learned index with high performance and availability. On the CPU side, DALdex is decoupled into a DRAM-accelerated model layer and an NVM-aware block layer to minimize NVM amplifications. On the DPU side, DALdex offloads model retraining tasks from the CPU based on the incremental learning scheme to mitigate model retraining overheads. Moreover, DALdex achieves seamless failover and instant recovery by leveraging the hardware isolation feature of DPU. The overall structure of DALdex is illustrated in Figure 3.

3.2 DALdex Structure

DRAM-Accelerated Model Layer. DALdex stores the model structure in DRAM to eliminate model accesses on the critical path in NVM. Specifically, the model layer consists of inner and accelerator nodes embedded with learned models.

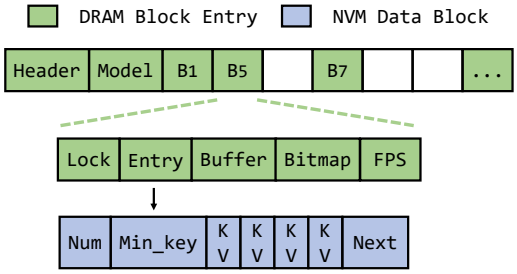

Figure 4: The accelerator node structure of DALdex.

Similar to leaf nodes in traditional tree-based range indexes, accelerator nodes reside at the bottom level of the model layer, while key-value pairs are separately stored in NVM for persistence. As the name implies, accelerator nodes target to speed up reads and writes in DALdex by expediting both model and data accesses during runtime.

Figure 4 details the accelerator node structure of DALdex. The accelerator node comprises a metadata header and a gapped array of block entries [11, 37]. The node header contains model parameters (i.e., slope and intercept) and metadata in DRAM. Each slot in the gapped array is either a block entry or a free slot reserved for future insertions. The block entry holds a persistent pointer that points to a corresponding data block in NVM. To further minimize NVM accesses for data blocks, the block entry also maintains several block metadata such as locks, bitmaps, and fingerprints in DRAM. Similar to accelerator nodes, the inner node contains a metadata header embedded with model parameters, followed by its child nodes (i.e., inner nodes or accelerator nodes).

NVM-Aware Block Layer. DALdex organizes key-value pairs into NVM-aware data blocks with a fixed size to minimize NVM amplifications. As illustrated in Figure 4, each data block contains the number of keys, the minimum key, and the next block pointer in its metadata to ensure crash consistency. By default, the size of data blocks is configured as 256B to match the access granularity of the XPLine in Intel Optane DC PMEM. Note that the data blocks can be configured to various sizes based on different characteristics of the storage device deployed in the system. Besides, key-value pairs in data blocks are unsorted to eliminate data movements during insertions [69, 71]. To enable scans, data blocks are linked as a singly linked list in NVM. With matched access granularity, the NVM-aware block layer allows DALdex to restrict most NVM accesses to a single data block for reads and writes.

3.3 DPU-Offloaded Incremental Learning

Reducing model retraining overheads is a key challenge for learned indexes. Prolonged model retrainings would block normal index operations and severely degrade the Quality of Service (QoS) in HPC systems [5]. However, state-of-the-art learned indexes fail to address this challenge due to the offline batched model retraining scheme [11, 32, 34, 37, 54, 61, 71].

Offloading model retraining tasks from the CPU to DPU is a promising approach to address this challenge. However, naively offloading the offline batched model retraining scheme to DPU is inefficient due to the limited onboard computing and memory resources in DPUs [58]. As shown in Figure 1(b), retraining a linear regression model with 100M data on the DPU side consumes up to 22.6 seconds, which is about 13.7× than on the CPU side. Such a straightforward offloading scheme fails to yield performance gains since the offloaded model retraining tasks on the DPU side become a new bottleneck. This motivates us to revisit the model retraining process and design a more efficient offloading scheme for DPUs.

Through an in-depth analysis of the offline batched model retraining scheme, we identify its inefficiency primarily stems from the static nature of learned models, which fail to dynamically adapt to evolving data distributions. In learned indexes, the static model is trained on a fixed dataset with an established data distribution that always falls behind with new data distributions. Therefore, learned indexes require frequent model retrainings to learn new data distributions. However, the offline batched retraining scheme adopts an inefficient training pattern to learn new data distributions from scratch. This not only ignores the knowledge (i.e., the old data distributions) encoded in old models, but also wastes valuable computing and memory resources.

However, we observe that new models can be incrementally retrained with low overheads by updating old models based on new data distributions. Inspired by this insight, we propose an online incremental learning scheme to bridge the gap between static learned models and evolving data distributions [55, 73]. The core idea behind it is to *incrementally train new models by dynamically retraining old models based on new data distributions.*

Online Incremental Learning Scheme. Unlike existing offline batched model retraining schemes, the online incremental learning scheme is a dynamic approach that tracks data distribution changes during runtime [40, 42]. To demonstrate this, we decompose the offline batched model retraining scheme and extract common factors from it, denoted as *intermediate results* $\{S_k, S_{kk}, S_p, S_{kp}\}$, as detailed in Equations 1 and 2. Due to the equivalence of mathematical transformations, the model parameters (i.e., slope a and intercept b) can be uniquely derived from the intermediate results $\{S_k, S_{kk}, S_p, S_{kp}\}$ without additional errors. Therefore, by maintaining intermediate results during runtime, we can dynamically retrain new models through several lightweight floating-point operations. This approach effectively decouples model parameters from extensive iterations over the training data required in the offline batched model retraining scheme, thereby significantly mitigating model retraining overheads for learned indexes.

$$a = \frac{n\sum_0^{n-1} k_i p_i - \sum_0^{n-1} k_i \sum_0^{n-1} p_i}{n\sum_0^{n-1} k_i^2 - (\sum_0^{n-1} k_i)^2}$$
$$b = \frac{\sum_0^{n-1} k_i^2 \sum_0^{n-1} p_i - \sum_0^{n-1} k_i \sum_0^{n-1} k_i p_i}{n\sum_0^{n-1} k_i^2 - (\sum_0^{n-1} k_i)^2} \quad [41] \tag{1}$$

$$S_{k_n} = \sum_0^{n-1} k_i \qquad S_{kk_n} = \sum_0^{n-1} k_i^2$$
$$S_{p_n} = \sum_0^{n-1} p_i \qquad S_{kp_n} = \sum_0^{n-1} k_i p_i \tag{2}$$

To efficiently fit new data distributions, we need to dynamically update intermediate results $\{S_k, S_{kk}, S_p, S_{kp}\}$ during runtime. Formally, suppose a model LM_k is trained based on a data sequence $\{(k_0, p_0), (k_1, p_1), ..., (k_{n-1}, p_{n-1})\}$ with intermediate results $\{S_{k_n}, S_{kk_n}, S_{p_n}, S_{kp_n}\}$, where k_i denotes the key and p_i denotes its position. When a new data (k_m, p_m) is inserted into the data sequence $(0 \le m \le n)$, the intermediate results are updated from $\{S_{k_n}, S_{kk_n}, S_{p_n}, S_{kp_n}\}$ to $\{S_{k_{n+1}}, S_{kk_{n+1}}, S_{p_{n+1}}, S_{kp_{n+1}}\}$, as detailed in Equation 3. To ensure precision and avoid overflow, we allocate four double-precision floating point numbers for intermediate results, which only consume 32B space for each model.

$$S_{k_{n+1}} = S_{k_n} + k_m \qquad S_{kk_{n+1}} = S_{kk_n} + k_m^2$$
$$S_{p_{n+1}} = S_{p_n} + n \qquad S_{kp_{n+1}} = S_{kp_n} + k_m p_m + S_{k_n} - S_{k_m} \tag{3}$$

Once new intermediate results $\{S_{k_{n+1}}, S_{kk_{n+1}}, S_{p_{n+1}}, S_{kp_{n+1}}\}$ are calculated, we can incrementally train a new model LM_{k+1} on top of the old model LM_k, as detailed in Equation 4. Note that the online incremental learning scheme is mathematically equivalent to the offline batched retraining scheme, both of which converge to the optimal solution with a closed form [27].

$$a_{new} = \frac{(n+1)S_{kp_{n+1}} - S_{k_{n+1}} S_{p_{n+1}}}{(n+1)S_{kk_{n+1}} - (S_{k_{n+1}})^2}$$
$$b_{new} = \frac{S_{kk_{n+1}} S_{p_{n+1}} - S_{k_{n+1}} S_{kp_{n+1}}}{(n+1)S_{kk_{n+1}} - (S_{k_{n+1}})^2} \tag{4}$$

DPU-Offloaded Incremental Learning Scheme. Simply employing the online incremental learning scheme on the CPU side is still inefficient, since it remains on the critical path of DALdex operations, as depicted in Figure 5(a). To further remove model retrainings from the critical path, we offload the incremental learning scheme to the DPU side. As depicted in Figure 5(b), the offloaded incremental learning tasks on the DPU side overlap with DALdex operations on the CPU side, which further improves the overall performance of DALdex. Unlike naively offloading the offline batched model retraining scheme, offloading the incremental learning scheme to DPU is much more efficient with several lightweight floating-point operations (Equation 4), which consumes negligible DPU cycles.

However, eagerly offloading the online incremental learning scheme to the DPU side without proper synchronizations may lead to lost updates across system crashes. To resolve this problem, a naive approach is to employ locks to ensure strong synchronizations between the CPU and DPU for each model update. However, this would significantly block normal index operations on the CPU side due to the huge performance gap between the CPU and DPU. To address this challenge at a low cost, we further offload the model structure of DALdex apart from intermediate results to the DPU side for incremental learning. New models are asynchronously retrained and updated on the DPU side bypassing the CPU to maximize the parallelism of DALdex. When model retrainings are triggered on the CPU side, DALdex then notifies the DPU and fetches new model parameters from the DPU to CPU through the DOCA communication channel to ensure crash consistency [44].

3.4 DPU-Extended Failover

The CPU-DPU hybrid structure dramatically enhances the availability of DALdex across system crashes. Specifically, in the event of DPU-side system crashes, DALdex on the CPU side remains unaffected, since DALdex can locally retrain new models without DPUs. Moreover, when the system crashes on the CPU side, the DPU subsystem remains alive due to the hardware-level isolation between the CPU and DPU. In this scenario, DALdex can be entirely offloaded to the DPU side without interruptions.

As mentioned previously, to enable efficient incremental learning, the model structure of DALdex is offloaded to the DPU side, which is a natural backup of the CPU side. Therefore, by promoting the offloaded model structure to the primary structure, DALdex can be entirely offloaded to the DPU side without inconsistencies. Similar to redo logs, the offloaded model structure retains new model updates, since new models are first incrementally trained on the DPU side. This enables DALdex to achieve seamless failover by consistently migrating to the DPU side during CPU-side system crashes. However, since key-value pairs are persisted in NVM on the CPU side, the offloaded DALdex on the DPU side needs to access them through DMA via the PCIe bus.

In addition, simultaneous system crashes on both CPU and DPU sides typically imply catastrophic failures (e.g., hardware damages), which are extremely rare in practice. In such scenarios, DALdex needs to rebuild the model structure from scratch after system-level emergency repairs.

3.5 DPU-Assisted Instant Recovery

When the system restores, DALdex can either migrate to the CPU side or proceed on the DPU side. However, due to the limited onboard computing and memory resources in DPUs, the performance of DALdex on the DPU side is much lower than on the CPU side. Therefore, for performance considerations, DALdex prioritizes recovery to the CPU side upon restorations. During recovery, unlike APEX and PLIN that need to rebuild partial index structures or redo heavy NVM logs [37, 71], DALdex can be instantly recovered by retrieving the offloaded model structure from the DPU to CPU through DMA via the PCIe bus. Compared to naively rebuilding the model structure on the CPU side, the DMA-based scheme achieves minimal recovery overheads due to the space-efficient model structure in DALdex. Besides, once the DMA task is finished, DALdex reverts to a consistent state without performance degradation, since the offloaded model structure keeps up-to-date during failover.

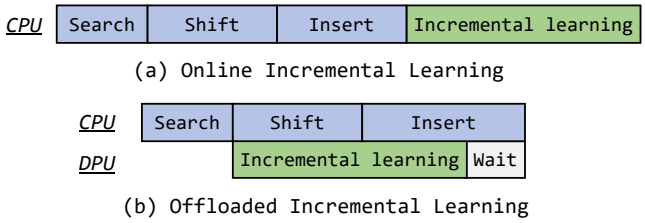

Figure 5: DPU-offloaded incremental learning scheme.

4 Implementation

4.1 Index Operations

Search & Scan. Searching for a target key-value pair starts in the model layer in DRAM. First, DALdex (1) searches at the root node and traverses in the inner nodes using embedded models until reaching the accelerator node. Then DALdex (2) searches within the accelerator node from the predicted position to locate the target block entry. If the predicted slot does not contain the target block entry due to prediction errors, DALdex needs to exponentially search the following slots, until reaching the actual slot (i.e., the last slot that is smaller than the target key). To ensure ordered exponential searches, free slots in the accelerator node are assigned with a free flag that is the maximum key in the numerical range. After locating the target block entry, DALdex (3) checks the bitmaps and fingerprints in the block entry in DRAM, and then linearly searches within the corresponding data block in NVM to find the target key-value pair. If the target key-value pair is not found, DALdex (4) proceeds to search the next data block in the singly linked list. For scans, DALdex further searches through subsequent data blocks until sufficient records are collected.

However, accessing the data block in NVM for the first time typically results in a cache miss due to separated address spaces in DRAM and NVM. To mitigate this problem, we maintain a data buffer in DRAM as a primary cache for the corresponding data block in NVM. Therefore, DALdex first searches in the data buffer if it hits, and then searches in the data block if the data buffer misses.

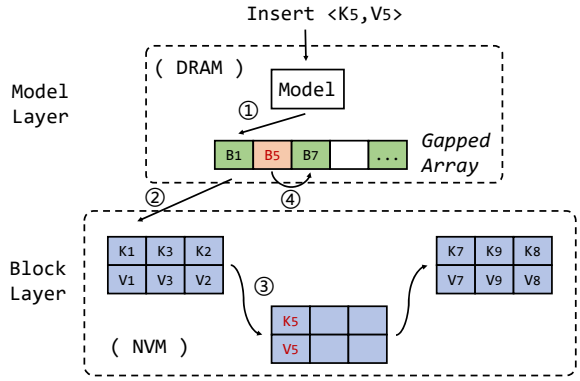

Figure 6: The insertion process of DALdex.

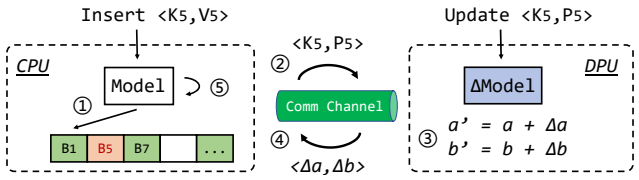

Figure 7: The incremental learning process of DALdex.

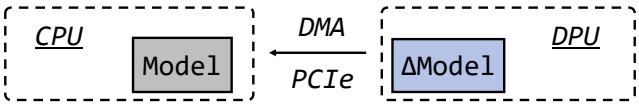

Figure 8: The recovery process of DALdex.

Insert. Similar to searching, inserting a new key-value pair starts by searching for the corresponding block entry in the model layer in DRAM. After locating the target data block in NVM, DALdex linearly searches within the data block to find a deleted or free slot for insertion, since key-value pairs in data blocks are unsorted. To ensure crash consistency, DALdex first inserts the value and then inserts the key followed by flush and fence instructions [37, 69, 71]. Once the key is persisted, the new insertion remains consistent across system crashes [69, 71]. Finally, DALdex updates the associated metadata in DRAM.

However, if the target data block is full, instead of splitting it into two child blocks, DALdex allocates a new data block, inserts the new key-value pair into it, and links the new data block into the singly linked list to eliminate expensive data movements in NVM. This does not introduce extra memory overheads since the size of data blocks is accurately configured to match the access granularity in NVM. Then DALdex inserts the new block entry into the accelerator node based on the model prediction. However, if the predicted position is already occupied, DALdex needs to locate the nearest free slot, and sequentially shift occupied slots to make space for the new insertion. Finally, DALdex in-place inserts the new block entry into the free slot to ensure sorted order in the accelerator node. For example, as depicted in Figure 6, when inserting a new key-value pair $< K_5, V_5 >$, DALdex ❶ searches within the accelerator node to find the target data block B_1 with the key K_1 smaller than K_5, and ❷ tries to insert $< K_5, V_5 >$ into B_1. However, the insertion fails since B_1 is already full. Then DALdex ❸ allocates a new data block B_5, inserts $< K_5, V_5 >$ into B_5, and links B_5 into the singly linked list. Next, DALdex attempts to insert B_5 into the accelerator node at position 1 based on the model prediction. However, since the predicted position 1 is already occupied by B_7, DALdex needs to locate the nearest free slot at position 2, and ❹ shifts B_7 from position 1 to 2 to make space for B_5. Finally, DALdex inserts B_5 at position 1 in the accelerator node.

Structural Modification Operations. When an accelerator node fills up with new insertions, DALdex needs to either expand or split it to accommodate more data [11, 37]. To determine which mechanism to apply, DALdex employs a lightweight decision scheme based on the model accuracy. Specifically, DALdex checks the Root Mean Square Error (RMSE, which can be easily derived from intermediate results) of the incremental model on the DPU side. If the RMSE does not exceed a predefined threshold, demonstrating the model accuracy is sufficient, DALdex expands the accelerator node at a low cost. Otherwise, it implies that a single model would underfit the data distribution, since the training data size exceeds the expressivity of a linear regression model. In this scenario, DALdex splits the accelerator node to reduce the training data size for each model.

For accelerator node expands, DALdex (1) fetches the incremental model from the DPU to CPU as a new model, (2) allocates a larger accelerator node, (3) inserts block entries from the old node to the new node based on new model predictions, and (4) updates the corresponding node pointers. For accelerator node splits, DALdex even splits it into two child nodes and locally retrains two models based on the partitioned data in each child node. Subsequently, the new child node is inserted into its parent node with possible cascade splits. To avoid frequently splitting inner nodes, DALdex adopts an opportunistic insertion strategy as PLIN [71], which ensures that the child node is inserted into its parent node only if there are free slots available. Otherwise, the child node is not inserted and becomes an orphan node linked to its left sibling node. However, if the number of orphan nodes exceeds a predefined threshold, DALdex needs to rebuild all inner nodes to merge orphan nodes into their parent nodes for performance.

Delete & Update. Deleting an existing key-value pair in DALdex is implemented by invalidating the target key with a free flag. For updates, DALdex in-place modifies the old value with the new value atomically. Different from insertions, deletions and updates do not pollute actual key distributions or learned models [60]. Therefore, DALdex do not retrain new models in these scenarios.

Offloaded Incremental Learning. DPU offloads incremental learning by processing requests from the CPU side. Specifically, DPU dynamically updates intermediate results based on new insertions during runtime (Equation 3). When model retrainings are triggered on the CPU side during SMOs, DALdex sends an incremental learning request from the CPU to DPU for model retrainings. Once receiving the request, DPU incrementally retrains a new model on top of intermediate results (Equation 4). Then DPU sends new model parameters back to the CPU through the DOCA communication channel. For example, as illustrated in Figure 7, during insertions, DALdex ❶ inserts a new data block B_5 into the accelerator node, and ❷ sends the new key as well as its position $< K_5, P_5 >$ from the CPU to DPU through the DOCA communication channel. Then DPU updates intermediate results based on $< K_5, P_5 >$ to learn new data distributions. Once DPU receives the incremental learning request from the CPU, it ❸ incrementally retrains a new model based on the intermediate results, and ❹ sends new model parameters back to the CPU through the DOCA communication channel. Finally, DALdex ❺ in-place updates the old model using new model parameters on the CPU side.

To enable efficient incremental learning, DPU needs to dynamically update intermediate results to capture new data distributions during runtime. However, updating S_{kp} is inefficient since it involves additional prefix sums (i.e., S_{k_m} in Equation 3), which incurs substantial computing and memory overheads on the DPU side. To mitigate this issue, DALdex does not update prefix sums during insertions and approximately computes S_{kp} on the DPU side. This approximation significantly reduces the incremental learning latency and further improves the overall performance of DALdex at the cost of minimal retraining errors. However, this trade-off is necessary due to the limited onboard computing and memory resources in DPUs.

4.2 Bulk Load

Similar to state-of-the-art learned indexes [13, 71], DALdex is constructed recursively from the bottom up using the OptimalPLR algorithm [53]. During the bulk load process, we first use the OptimalPLR algorithm to partition the input data sequence into multiple segments, and organize the partitioned data into fixed-size data blocks in NVM. Then we build accelerator nodes in DRAM on top of data blocks and recursively build inner nodes until reaching the root node. Compared to the top-down structure that has to propagate model updates upwards to the root node during SMOs [53, 60], the bottom-up structure in DALdex offers more flexibility to independently update the model structure in each level. Therefore, DALdex significantly reduces the model dependency in the index structure and thus achieves better scalability in concurrent scenarios.

4.3 Concurrency

DALdex utilizes the popular optimistic locking scheme in both accelerator nodes and data blocks to offer high concurrency [32, 37, 54, 71]. Readers only need to check the lock version without acquiring it, but need to retry reading if the lock version changes during reads. However, writers need to exclusively acquire the lock and increment the lock version by one if the write succeeds. To minimize lock contentions, DALdex employs a fine-grained concurrency control scheme that decouples the locking of model structure in DRAM from data blocks in NVM. Therefore, DALdex allows more reads and writes to be performed asynchronously in DRAM and NVM. Moreover, DALdex offloads expensive model retrainings from the CPU to DPU, which significantly narrows the critical section in writes and thus achieves high concurrency in concurrent scenarios.

5 Performance Evaluation

5.1 Experimental Setup

We conduct experiments on a Linux server (Ubuntu 18.04 LTS with kernel 5.4.0) equipped with two Intel Xeon Gold 6230R CPUs. Each CPU has 26 cores with 32KB L1D cache, 32KB L1I cache, 1MB L2 cache, and 35.75MB L3 cache. The system is equipped with 384GB DRAM, 768GB NVM (6×128 GB Intel Optane DC PMEM on the AppDirect mode), and an NVIDIA Mellanox BlueField-2 MT42822 DPU on the DPU mode with DOCA 2.6.0. The hardware configuration of BlueField-2 DPU is listed in Table 1.

Datasets. We select three real-world datasets: the Books, Genome, and OSM datasets for evaluation [25, 60]. The Books dataset is a set of Amazon book sales rank data [2], the Genome dataset contains autosomal and sex chromosome sequences from the human genome [48], and the OSM dataset is a one-dimensional projection of uniformly sampled multi-dimensional locations from OpenStreetMap [47]. Each dataset consists of 200M unique key-value pairs, where both keys and values are 8B integers. Among these datasets, the OSM dataset exhibits a complex data distribution, making it the most challenging dataset for model retraining and inference [60].

Competitors. We compare DALdex with five state-of-the-art persistent learned indexes and tree-based range indexes, including APEX [37], PLIN [71], TLBTree [38], ROART [39] and PACTree [24]. Among these persistent indexes, APEX is a DRAM-NVM hybrid persistent learned index that stores partial metadata in DRAM to improve performance. PLIN is an NVM-only persistent learned index that stores all components in NVM to achieve instant recovery. TLBTree is an NVM-only B+tree-style persistent range index that divides the index into a read-optimized top layer and a write-optimized bottom layer to trade off between read and write

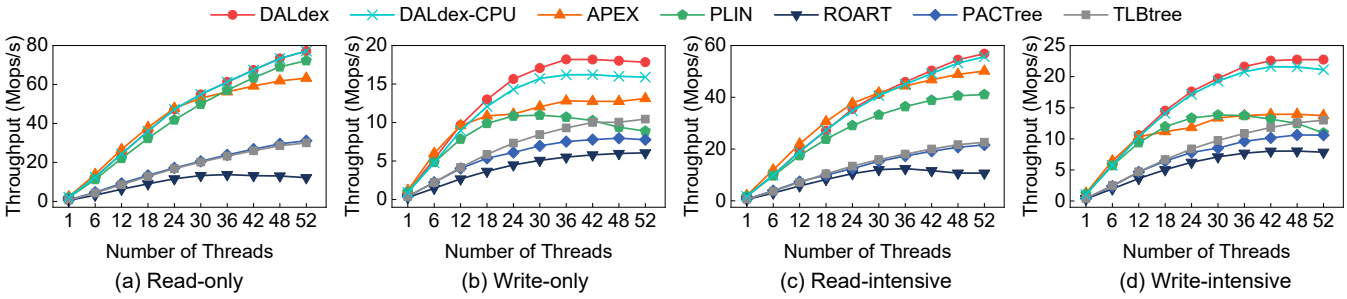

Figure 9: Throughputs on the Books dataset.

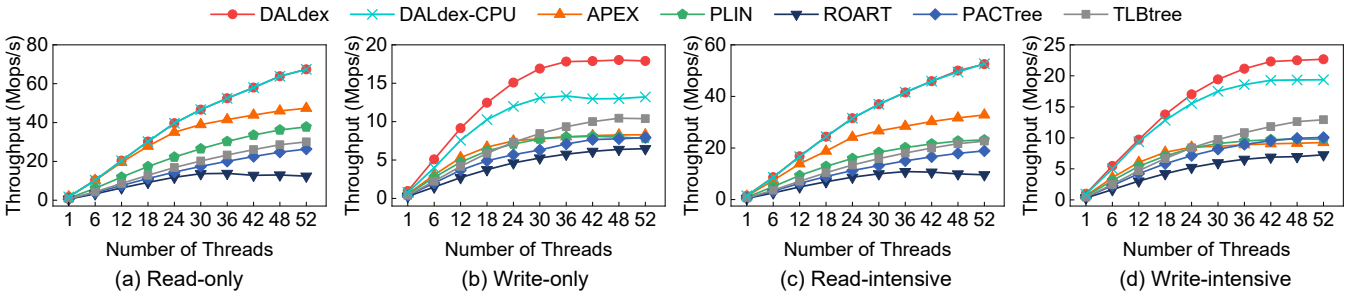

Figure 10: Throughputs on the Genome dataset.

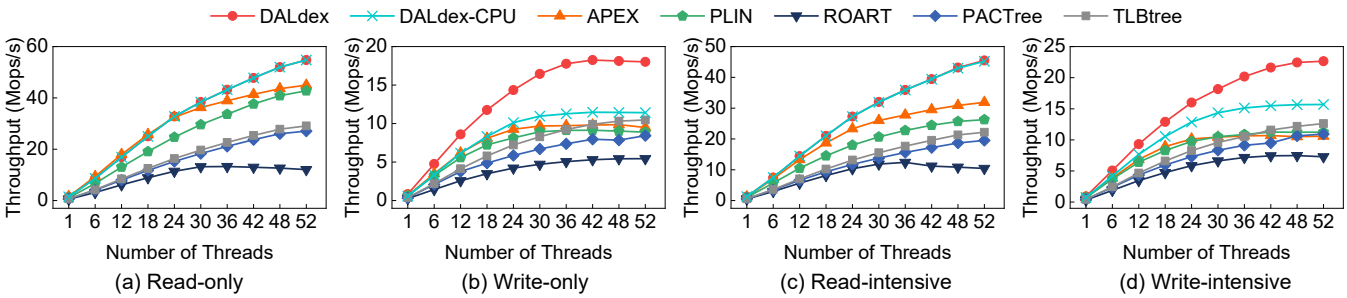

Figure 11: Throughputs on the OSM dataset.

performance. PACTree and ROART are two NVM-only trie-style persistent range indexes built based on the Adaptive Radix Tree (ART) [31]. The implementations of all persistent indexes are available on GitHub and we directly use their open-source codes with recommended parameters in their original papers for comparisons. However, these persistent indexes are designed for CPUs and unaware of DPUs. Therefore, it is challenging to implement CPU-DPU hybrid variants for them. To ensure fairness, we also include a CPU-oriented DALdex (i.e., DALdex-CPU) for comparison to comprehensively exhibit the benefits of our proposed design.

5.2 Overall Performance

In this experiment, we evaluate the performance of all persistent indexes on read-only, write-only, read-intensive (80% read and 20% write), and write-intensive (20% read and 80% write) workloads. All read and write operations follow a uniform distribution. The number of threads scales from 1 to 52 with hyper-threading over 26 threads. To eliminate the NUMA effect, we bind each thread to different physical cores within a single socket under 26 threads. For all configurations, DALdex exhibits higher performance than other indexes, especially on complex datasets with more threads.

Figures (9-11)(a) show the performance on the read-only workload. DALdex scales almost linearly for reads and outperforms other persistent indexes by 1.07-6.34× under 52 threads. The main reason comes from the minimized NVM accesses and fine-grained concurrency control enabled by the decoupled index structure in DALdex. Specifically, both APEX and PLIN require at least two NVM accesses for reads. One is for accessing the learned model in the node header, and the other is for retrieving the target key-value pair in the data node. In comparison, DALdex requires only one NVM access for retrieving the target key-value pair in the NVM-friendly data block. This is because learned models in DALdex are maintained in DRAM to eliminate NVM accesses on the critical path. However, the read performance of DALdex shows marginal benefits and falls short of APEX with fewer threads, since DALdex needs to locally search within internal nodes due to prediction errors introduced by the OptimalPLR algorithm [13]. While APEX avoids internal searches since

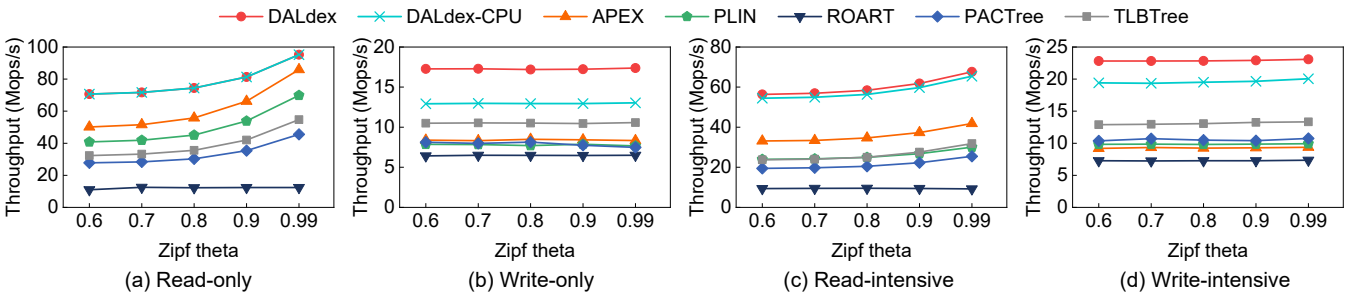

Figure 12: Throughputs under Zipfian distribution across various skewness (Genome dataset, 52 threads).

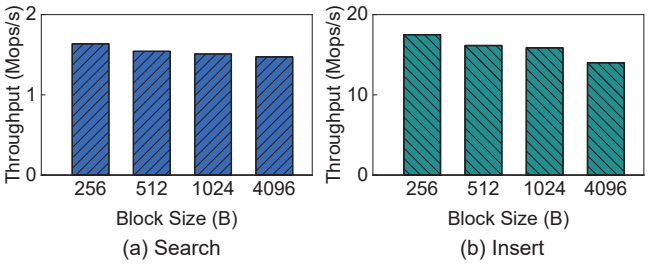

Figure 13: Impact of block size.

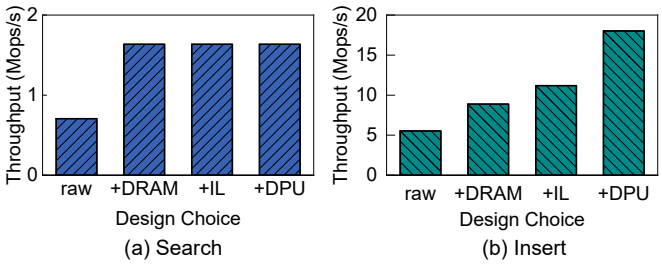

Figure 14: Factor analysis (IL: Incremental Learning).

inner node predictions are accurate due to its top-down structure [11, 37]. However, as the number of threads increases, DALdex eventually surpasses APEX in reads and exhibits better scalability than APEX due to the fine-grained concurrency control. Besides, DALdex-CPU performs nearly the same as DALdex since there are few model retrainings in read-only and read-intensive workloads. Therefore, DALdex gains limited performance benefits from the DPU-offloaded scheme in these scenarios. The non-learned indexes, TLBTree, ROART and PACTree, exhibit much lower performance than learned indexes due to massive NVM accesses in internal tree structures without using learned models.

Figures (9-11)(b) show the performance on the write-only workload. DALdex outperforms other persistent indexes by 1.36-3.31× under 52 threads. In the Books dataset, there are few model retrainings during insertions since the data distribution is simple. Therefore, DALdex obtains limited performance improvements from the offloaded incremental learning scheme. However, DALdex still achieves the highest write performance among other persistent indexes since the NVM-friendly index structure significantly minimizes write amplifications in NVM. Besides, the unsorted key-value pairs in data blocks avoid extensive data movements during insertions, which further reduces NVM writes and saves NVM bandwidth. In Genome and OSM datasets, the write performance of APEX and PLIN is significantly constrained by frequent model retrainings due to complex data distributions. Unlike them, DALdex achieves the best write performance by mitigating model retraining overheads through the offloaded incremental learning scheme on the DPU side. Similarly, DALdex outperforms DALdex-CPU by 1.12-1.58× under 52 threads, with higher performance improvements on more complex datasets. The non-learned indexes exhibit consistently low performance across all datasets, since they are unaware of different data distribution patterns.

Figures (9-11)(c) and (d) show the performance on the read-intensive and write-intensive workloads. DALdex outperforms other persistent indexes by 1.13-5.46× and 1.65-3.12× under 52 threads, respectively. In the read-intensive workload, DALdex allows more reads without locks in DRAM due to the optimistic concurrency control in the decoupled index structure. In the write-intensive workload, DALdex significantly mitigates model retraining overheads and minimizes lock contentions in writes, thus achieving the best performance with high concurrency.

5.3 Performance of Varying Access Patterns

In this experiment, we evaluate the performance of all persistent indexes under the Zipfan distribution across various skewness. Figure 12 shows the evaluation results on the Genome dataset with 52 threads. Compared to Figure 10, all indexes achieve higher read performance with higher skewness, since more reads focus on a small set of hot keys in the CPU cache rather than NVM. However, the write performance achieves minimal improvements since writes are eventually flushed from the CPU cache to NVM. DALdex outperforms other persistent indexes under all skewness, exhibiting robust performance under diverse access patterns.

5.4 Effects of Each Design

In this experiment, we study the impact of different data block sizes on the overall performance and analyze the effect of each design choice in DALdex.

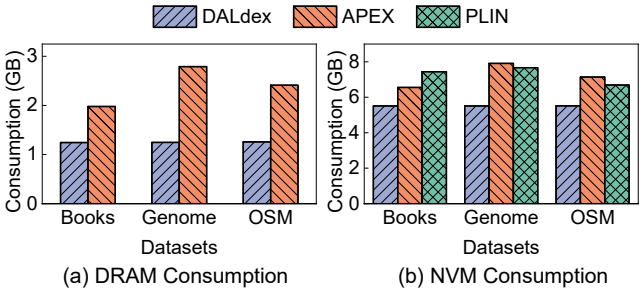

Figure 15: Memory consumption of DRAM and NVM.

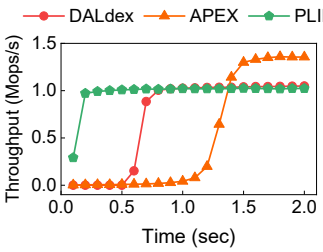

Figure 16: Single-thread recovery performance.

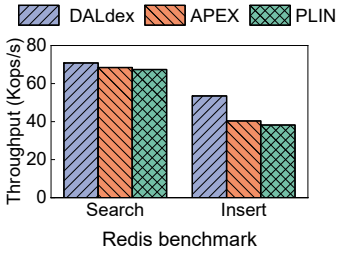

Figure 17: Redis benchmark performance.

Block Size. The data block size in DALdex is configurable to adapt to different characteristics of the storage device employed in the system. In this experiment, we assign different data block sizes to evaluate the impact on the overall performance. As shown in Figure 13(a), the read performance of DALdex on a single thread keeps dropping as the data block size increases, since larger data blocks incur more local searches in NVM. Similarly, as shown in Figure 13(b), the write performance of DALdex on 52 threads continues to decrease due to more block-level lock contentions in concurrent scenarios. Therefore, we set the data block size as 256B by default, achieving both the best read and write performance among other alternative sizes.

Factor Analysis. To demonstrate the effect of each design choice in DALdex, we conduct a factor analysis on DALdex. We first implement a baseline DALdex by maintaining all components of DALdex in NVM and retraining learned models based on the offline batched model retraining scheme. Then we gradually incorporate each design choice into the baseline implementation and evaluate the performance of DALdex on the OSM dataset. As shown in Figure 14(a), by building DRAM-accelerated learned model structure, DALdex achieves 2.31× speedup in reads on a single thread compared to the baseline implementation. However, the read performance of DALdex cannot further benefit from our proposed DPU-offloaded incremental learning scheme, since there is no model retraining in the read-only workload. As shown in Figure 14(b), DALdex achieves 1.61-3.26× speedup in writes on 52 threads compared to the baseline implementation, demonstrating the effectiveness of each design choice in DALdex.

Table 2: Runtime statistics of intermediate results.

Datasets	Number of Learned Models	Size of Intermediate Results
Books	1691	52.84 KB
Genome	9987	312.09 KB
OSM	41281	1.26 MB

5.5 Memory Consumption

In this experiment, we measure the end-to-end DRAM and NVM consumption of each learned index on various datasets by monitoring DRAM and NVM allocators during runtime. First, we initialize each learned index with 100M data followed by inserting another 100M data on the write-only workload, and then collect DRAM and NVM consumption of each index for comparison. As depicted in Figure 15, DALdex reduces DRAM consumption by 58.9-123.7% compared to APEX on different datasets, since APEX needs to maintain auxiliary fanout trees and cost models during runtime. However, DALdex only maintains the space-efficient model structure in DRAM, which incurs minimal DRAM overheads. In comparison, PLIN does not consume DRAM at all due to its NVM-only structure. Besides, DALdex reduces NVM consumption by 18.9-43.4% compared to APEX and PLIN on various datasets, since NVM-friendly data blocks with matched access granularity effectively reduce NVM fragmentation in DALdex. We also measure the runtime statistics of intermediate results on the DPU side, as shown in Table 2. The intermediate results incur negligible DRAM overheads on the DPU side due to the flat index structure in DALdex.

5.6 Recovery Performance

In this experiment, we evaluate the recovery performance of each persistent learned index on the OSM dataset. We first initialize each index with 100M data followed by inserting another 100M data on the write-only workload. During insertions, we randomly crash the system on the CPU side following the methodology in LineFS [22]. Then we evaluate the recovery performance of each index upon system restores. Figure 16 shows the recovery performance of each persistent learned index on a single thread. DALdex exhibits faster recovery than APEX since APEX adopts a lazy recovery scheme that rebuilds DRAM metadata during runtime. Therefore, the initial recovery performance of APEX is low due to the lack of necessary DRAM metadata. However, the recovery performance of DALdex falls short of PLIN, since PLIN is an NVM-only persistent learned index that maintains all components in NVM. Therefore, PLIN does not need to rebuild the model structure during recovery and thus achieves the fastest recovery. In comparison, DALdex only requires lightweight DMA read requests to fully recover the model

structure on the CPU side. However, the current DMA-based recovery scheme still incurs nearly 0.5s latency limited by the PCIe bandwidth in our system. We estimate that the emerging CXL (Compute Express Link) technique [10, 67] with higher PCIe bandwidth and faster DMA transfer rate could further improve the recovery performance of DALdex.

5.7 Real-World Applications

To evaluate the performance of each persistent learned index in real-world applications, we leverage a modified version of Redis to support multi-thread execution and replace the hash index in Redis with each learned index for evaluation [49]. Since learned indexes do not support inserting data into an empty index, we first bulk load each learned index with 1000K random keys in a batch, and then use Redis-benchmark to *Get* or *Set* 1000K random keys for evaluation. For fairness, the keys are 8B random integers as in the micro benchmarks. As shown in Figure 17, DALdex achieves 1.03-1.05× higher read performance and 1.32-1.40× higher write performance than APEX and PLIN in the Redis benchmark, demonstrating the efficiency of DALdex in real-world applications.

6 Discussion

CPU-Oriented Scheme vs. DPU-Offloaded Scheme. DALdex leverages the DPU-offloaded scheme to remove model retrainings from the critical path for performance and ensure crash consistency through hardware-level isolation. However, the CPU-oriented scheme is crash-unsafe without the fault-tolerance support provided by DPU. To avoid this, DALdex-CPU has to write NVM logs for each model update during runtime, which incurs excessive NVM accesses and significantly decreases system performance. Therefore, we design to offload incremental learning to DPU, thus enhancing the performance and availability of DALdex.

Cross-NVM Compatibility. DALdex is built based on Intel Optane DC PMEM as a case study since it is the only commercially available NVM. Unfortunately, for some operational and business reasons, Intel announced to discontinue the Optane DC Persistent Memory business in July 2022 [35]. However, the design choices of DALdex can easily migrate to other types of NVMs if they exhibit similar mismatched access granularity between the CPU cache and NVM medium. In practice, with several parameter adjustments (i.e., the NVM-aware data block size), DALdex can also be extended to other NVMs with minimized NVM amplifications.

7 Related Work

SmartNIC Offloads. SmartNICs are specialized to offload compute-intensive tasks from CPUs. LineFS [22] offloads NVM-based distributed file system operations to SmartNICs to improve performance and availability. PMNet [51] offloads the data plane to SmartNICs equipped with NVM to extend the persistence domain from servers to the network. Xenic [50] offloads transaction processes to SmartNICs to improve the efficiency of distributed transactions. Different from these schemes, DALdex targets persistent learned indexes and offloads model retrainings to DPUs based on the incremental learning scheme.

Learned Indexes. RMI (Recursive Model Index) is the first learned index leveraging machine learning models [27]. However, RMI is a static learned index and does not support insertions during runtime. ALEX [11] proposes a gapped array structure to reserve free slots in data nodes for future insertions. FITing-Tree [14] proposes a delta buffer structure apart from data nodes for insertions. PGM-index [13], LIPP [61], DILI [34] and RadixSpline [26] mainly focus on optimizing model structures during the bulk load process. However, they still employ expensive offline batched model retraining schemes during runtime. Unlike them, DALdex mainly focuses on optimizing model retrainings by offloading the incremental learning scheme to DPU.

Persistent Indexes. Persistent indexes have been widely studied over the last decade, even before the first commercial Intel Optane DC PMEM was available. uTree [8] decouples B+Tree leaf nodes into an array layer and a list layer to reduce the tail latency of SMOs. LB+Tree [36] introduces a 3DXPoint-aligned node structure to reduce NVM writes during insertions. DPTree [74] combines multiple writes in DRAM and merges them into NVM in a batch to reduce persistence overheads. Besides, there are also several automatic persistent index conversion frameworks, such as RECIPE [30] and Nap [56]. However, they target traditional tree-based range indexes and hash indexes, which fail to work for persistent learned indexes.

8 Conclusion

In this work, we propose DALdex, a CPU-DPU hybrid persistent learned index with high performance and availability. DALdex in-depth analyzes the bottleneck of existing model retraining schemes and proposes an incremental learning scheme to mitigate model retraining overheads. Besides, DALdex exploits the hardware characteristics of DPU to offload model retrainings and achieves instant recovery via the PCIe bus. Moreover, DALdex designs an NVM-friendly index structure to optimize the access pattern for learned models on the critical path. The evaluation results demonstrate that DALdex significantly outperforms state-of-the-art persistent indexes with minimal DRAM and NVM overheads.

9 Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022. We are grateful to anonymous reviewers for their constructive suggestions and feedback.

References

[1] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010), 2237–2251.

[2] Amazon. 2024. Amazon sales rank data for print and kindle books. https://www.kaggle.com/datasets/ucffool/amazon-sales-rank-data-for-print-and-kindle-books.

[3] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *J. Emerg. Technol. Comput. Syst.* 9, 2 (May 2013), 1–35.

[4] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: a high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.* 11, 5 (Oct. 2018), 553–565.

[5] Quan Chen, Zhenning Wang, Jingwen Leng, Chao Li, Wenli Zheng, and Minyi Guo. 2019. Avalon: towards QoS awareness and improved utilization through multi-resource management in datacenters. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 272–283.

[6] Shimin Chen, Phillip B Gibbons, Suman Nath, et al. 2011. Rethinking database algorithms for phase change memory. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR '21, Vol. 11)*. 5th.

[7] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797.

[8] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: a persistent B+-tree with low tail latency. *Proc. VLDB Endow.* 13, 12 (July 2020), 2634–2648.

[9] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1077–1091.

[10] Grigory Chirkov and David Wentzlaff. 2023. Seizing the Bandwidth Scaling of On-Package Interconnect in a Post-Moore's Law World. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 410–422.

[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.

[12] Bo Fang, Hassan Halawa, Karthik Pattabiraman, Matei Ripeanu, and Sriram Krishnamoorthy. 2019. BonVoision: leveraging spatial data smoothness for recovery from memory soft errors. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 484–496.

[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 (April 2020), 1162–1175.

[14] 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.

[15] Anqi Guo, Yuchen Hao, Chunshu Wu, Pouya Haghi, Zhenyu Pan, Min Si, Dingwen Tao, Ang Li, Martin Herbordt, and Tong Geng. 2023. Software-Hardware Co-design of Heterogeneous SmartNIC System for Recommendation Models Inference and Training. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 336–347.

[16] Pouya Haghi, Cheng Tan, Anqi Guo, Chunshu Wu, Dongfang Liu, Ang Li, Anthony Skjellum, Tong Geng, and Martin Herbordt. 2024. SmartFuse: Reconfigurable Smart Switches to Accelerate Fused Collectives in HPC Applications. In *Proceedings of the 38th ACM International Conference on Supercomputing* (Kyoto, Japan) *(ICS '24)*. Association for Computing Machinery, New York, NY, USA, 413–425.

[17] Mert Hidayetoglu, Simon Garcia De Gonzalo, Elliott Slaughter, Yu Li, Christopher Zimmer, Tekin Bicer, Bin Ren, William Gropp, Wen-Mei Hwu, and Alex Aiken. 2024. CommBench: Micro-Benchmarking Hierarchical Networks with Multi-GPU, Multi-NIC Nodes. In *Proceedings of the 38th ACM International Conference on Supercomputing* (Kyoto, Japan) *(ICS '24)*. Association for Computing Machinery, New York, NY, USA, 426–436.

[18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies* (Oakland, CA, USA) *(FAST'18)*. USENIX Association, USA, 187–200.

[19] Intel. 2024. Intel Optane Persistent Memory Overview. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[20] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. 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, 804–818.

[21] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-Ri Choi. 2019. SLM-DB: single-level key-value store with persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies* (Boston, MA, USA) *(FAST'19)*. USENIX Association, USA, 191–204.

[22] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. 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, 756–771.

[23] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young-ri Choi, Alan Sussman, and Beomseok Nam. 2022. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation (OSDI '22)*. 161–177.

[24] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. 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, 424–439.

[25] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems (NeurIPS 19)* (2019).

[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.

[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.

[28] R. Madhava Krishnan, Diyu Zhou, Wook-Hee Kim, Sudarsun Kannan, Sanidhya Kashyap, and Changwoo Min. 2023. TENET: memory safe and fault tolerant persistent transactional memory. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'23)*. USENIX Association, USA, 247–264.

[29] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-change technology and the future of main memory. *IEEE Micro* 30, 1 (2010), 143–143.

[30] 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.

[31] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 38–49.

[32] 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.

[33] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: a scalable RDMA-oriented learned key-value store for disaggregated memory systems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'23)*. USENIX Association, USA, 99–114.

[34] 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 (May 2023), 2212–2224.

[35] Tianxi Li, Yang Wang, and Xiaoyi Lu. 2023. On the Discontinuation of Persistent Memory: Looking Back to Look Forward. In *Workshop on Hot Topics in System Infrastructure June 18, 2023, Orlando, Florida, USA Co-located with ISCA 2023.*

[36] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+Trees: optimizing persistent index performance on 3DXPoint memory. *Proc. VLDB Endow.* 13, 7 (March 2020), 1078–1090.

[37] 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.

[38] Yongping Luo, Peiquan Jin, Qinglin Zhang, and Bin Cheng. 2021. TLB-tree: A Read/Write-Optimized Tree Index for Non-Volatile Memory. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1889–1894.

[39] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 1–16.

[40] Meghana Madhyastha, Robert Underwood, Randal Burns, and Bogdan Nicolae. 2023. DStore: A Lightweight Scalable Learning Model Repository with Fine-Grain Tensor-Level Access. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 133–143.

[41] Steven J Miller. 2006. The method of least squares. (2006), 1–12.

[42] Lin Ning and Xipeng Shen. 2019. Deep reuse: streamline CNN inference on the fly via coarse-grained computation reuse. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 438–448.

[43] Nvidia. 2021. NVIDIA BLUEFIELD-2 DPU, Data Center Infrastructure on a Chip. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf.

[44] Nvidia. 2024. DOCA Comm Channel. https://docs.nvidia.com/doca/archive/2-6-0/doca+comm+channel/index.html.

[45] Nvidia. 2024. DOCA Documentation v2.6.0. https://docs.nvidia.com/doca/archive/doca-v2-6-0/index.html.

[46] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 371–386.

[47] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How good are modern spatial analytics systems? *Proc. VLDB Endow.* 11, 11 (July 2018), 1661–1673.

[48] Suhas SP Rao, Miriam H Huntley, Neva C Durand, Elena K Stamenova, Ivan D Bochkov, James T Robinson, Adrian L Sanborn, Ido Machol, Arina D Omer, Eric S Lander, et al. 2014. A 3D map of the human genome at kilobase resolution reveals principles of chromatin looping. *Cell* 159, 7 (2014), 1665–1680.

[49] Redis. 2024. Redis - The Real-time Data Platform. https://redis.io/.

[50] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. 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, 740–755.

[51] Korakit Seemakhupt, Sihang Liu, Yasas Senevirathne, Muhammad Shahbaz, and Samira Khan. 2021. PMNet: in-network data persistence. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 804–817.

[52] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Prism: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 588–602.

[53] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 16, 8 (April 2023), 1992–2004.

[54] 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.

[55] Gido M Van de Ven, Tinne Tuytelaars, and Andreas S Tolias. 2022. Three types of incremental learning. *Nature Machine Intelligence* 4, 12 (2022), 1185–1197.

[56] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. 2021. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *Proceedings of the 15th USENIX Conference on Operating Systems Design and Implementation (OSDI '21)*. 93–111.

[57] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based ordered key-value store using remote learned cache. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, 117–135.

[58] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *Proceedings of the 17th USENIX Conference on Operating Systems Design and Implementation (OSDI '23)*. 987–1004.

[59] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. 2010. Phase change memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.

[60] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are updatable learned indexes ready? *Proc. VLDB Endow.* 15, 11 (July 2022), 3004–3017.

[61] 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 (April 2021), 1276–1288.

[62] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24.

[63] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the performance of intel optane persistent memory: a close look at its on-DIMM buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 488–505.

[64] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'20)*. USENIX Association, USA, 169–182.

[65] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'15)*. USENIX Association, USA, 167–181.

[66] Jifei Yi, Mingkai Dong, Fangnuo Wu, and Haibo Chen. 2022. HTMFS: Strong Consistency Comes for Free with Hardware Transactional Memory in Persistent Memory File Systems. In *20th USENIX Conference on File and Storage Technologies (FAST '22)*. 17–34.

[67] Sungmin Yun, Hwayong Nam, Kwanhee Kyung, Jaehyun Park, Byeongho Kim, Yongsuk Kwon, Eojin Lee, and Jung Ho Ahn. 2024. CLAY: CXL-based Scalable NDP Architecture Accelerating Embedding Layers. In *Proceedings of the 38th ACM International Conference on Supercomputing* (Kyoto, Japan) *(ICS '24)*. Association for Computing Machinery, New York, NY, USA, 338–351.

[68] Jianping Zeng, Shao-Yu Huang, Jiuyang Liu, and Changhee Jung. 2024. Soft Error Resilience at Near-Zero Cost. In *Proceedings of the 38th ACM International Conference on Supercomputing* (Kyoto, Japan) *(ICS '24)*. Association for Computing Machinery, New York, NY, USA, 176–187.

[69] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: a lock-free PM-friendly persistent B+-tree for eADR-enabled PM systems. *Proc. VLDB Endow.* 15, 6 (Feb. 2022), 1187–1200.

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

[71] 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.

[72] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. 2022. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation (OSDI '22)*. 179–193.

[73] Da-Wei Zhou, Qi-Wei Wang, Zhi-Hong Qi, Han-Jia Ye, De-Chuan Zhan, and Ziwei Liu. 2024. Class-Incremental Learning: A Survey. *IEEE Trans. Pattern Anal. Mach. Intell.* 46, 12 (Dec. 2024), 9851–9873.

[74] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: differential indexing for persistent memory. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 421–434.