# LOFT: A Lock-free and Adaptive Learned Index with High Scalability for Dynamic Workloads

**Yuxuan Mo,** Yu Hua

*Huazhong University of Science and Technology*

*20th European Conference on Computer Systems (EuroSys), 2025*

# Dynamic Workloads

- Contain **insert** operations
  - *Growth* in the data size
  - *Changes* in data distribution

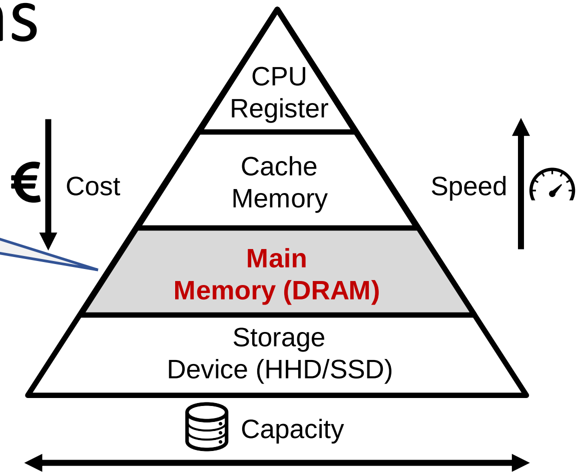
- Widely exist in real-world applications
  - e.g., Facebook, Twitter, etc.
  - Some are write-heavy[1]

[1] Dynamo: amazon's highly available key-value store, SOSP'07

# Memory Systems

- Memory systems play a critical role in compute systems
  - High-speed CPUs
  - Low-speed storage systems

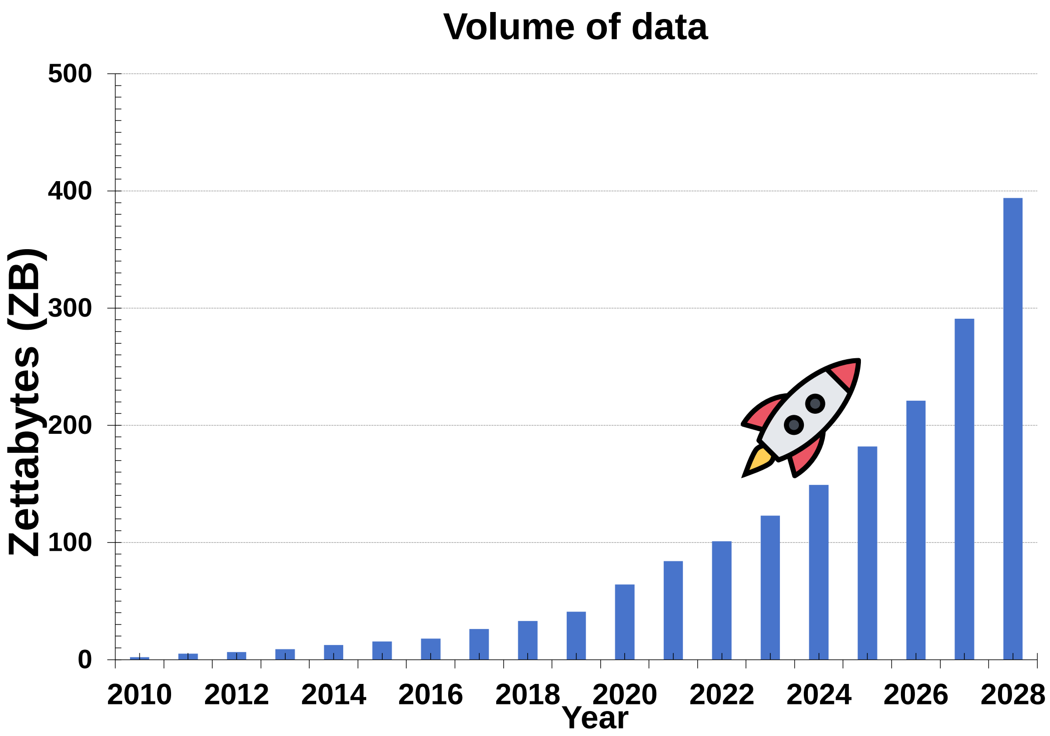
Bridge the performance gap



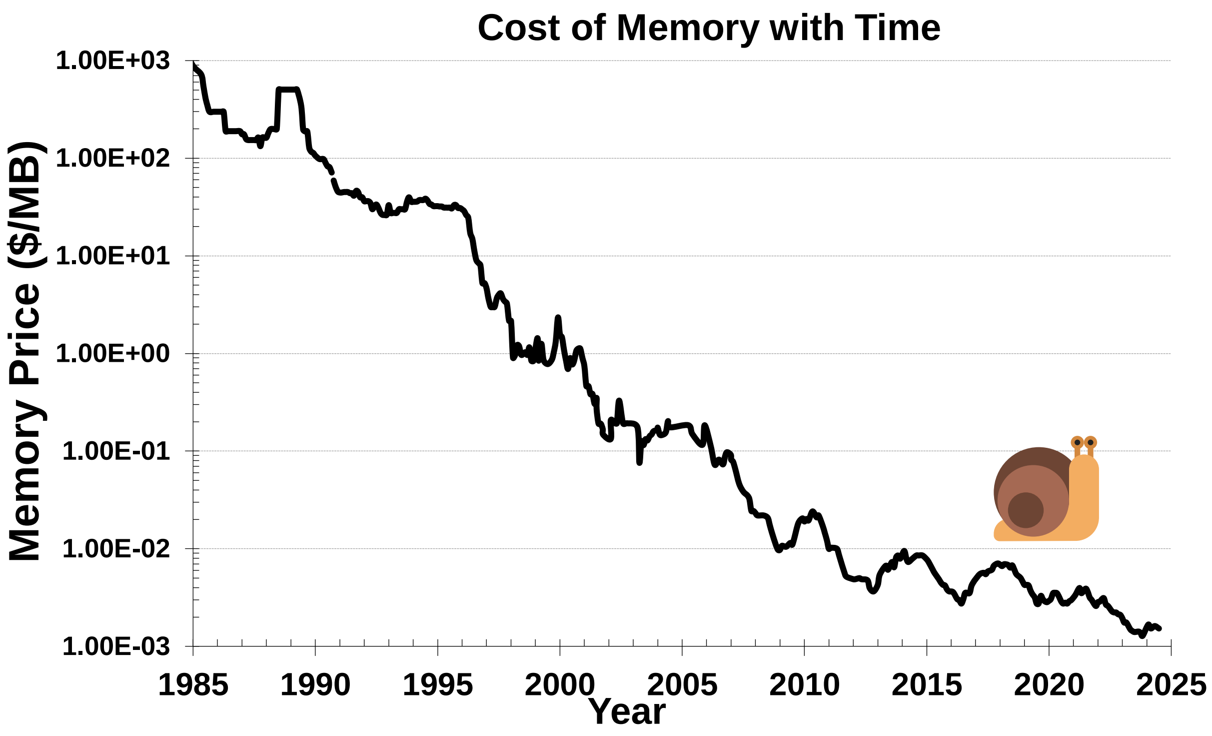

- In-memory index structures contribute to overall performance
  - e.g., B$^{+}$-tree and hash maps
  - Efficient data management with fast query performance

# Dilemma: Data Growth vs DRAM Scaling

- **Rapid growth** of stored data[1]



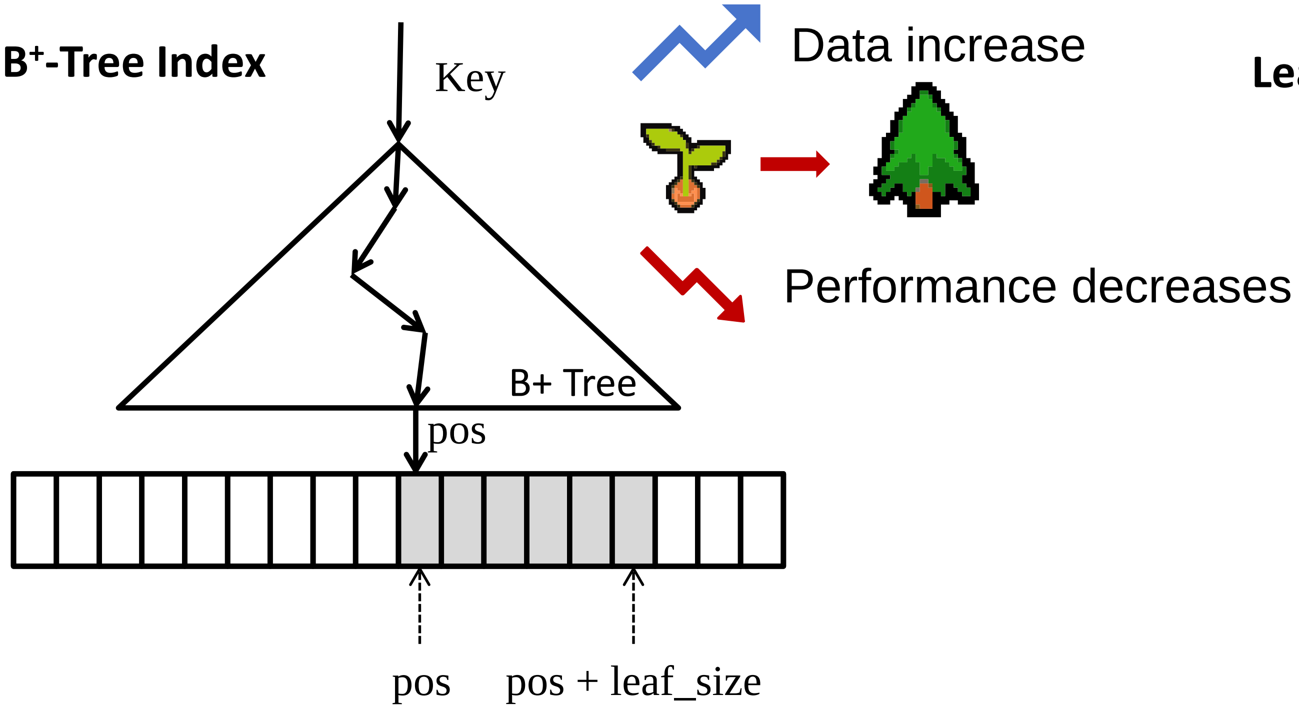

- **Slowdown** of DRAM scaling technology[2]

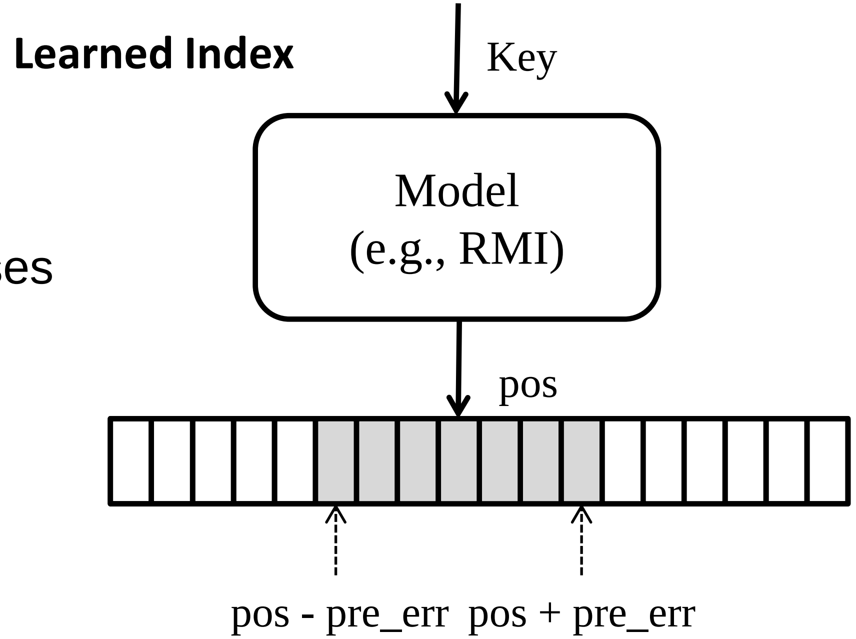



[1] https://www.statista.com/statistics/871513/worldwide-data-created/
[2] https://jcmit.net/memoryprice.htm

# Demand: Space-efficient and Scalable Index Structures

- Tree-like range-query indexes
  - Multiple pointer chasing operations
  - Large space overhead
- Learned indexes
  - Model-based calculation
  - Computation memory

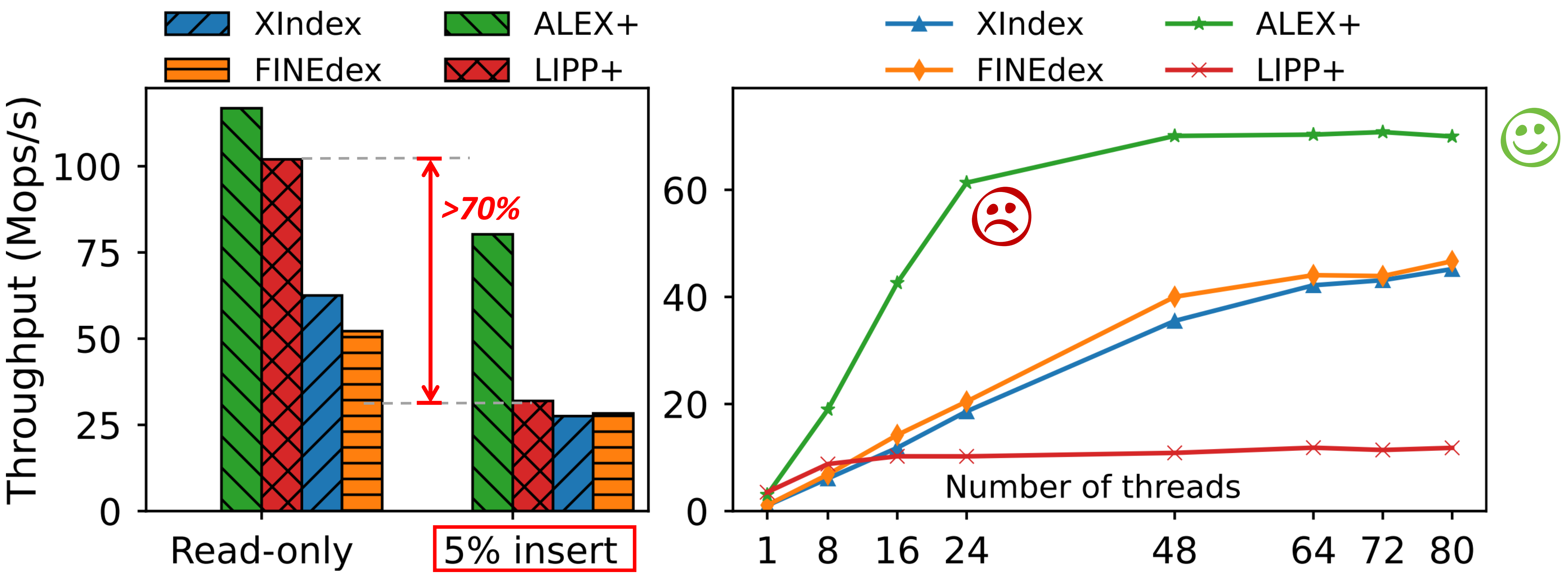



**Is the learned index the optimal solution?**

# Existing Learned Indexes in Dynamic Workloads

- Fail to scale to dynamic workloads
- Fail to simultaneously achieve high throughput and high scalability




- The used workloads generated from YCSB.
- XIndex@PPoPP'20, FINEdex@VLDB'21, ALEX+@VLDB'22, LIPP+@VLDB'22

# Existing Learned Indexes in Dynamic Workloads

- ✔ ➢ Space-efficient

  Small number of parameters

- ✔ ➢ Efficient query performance

  Model-based calculation

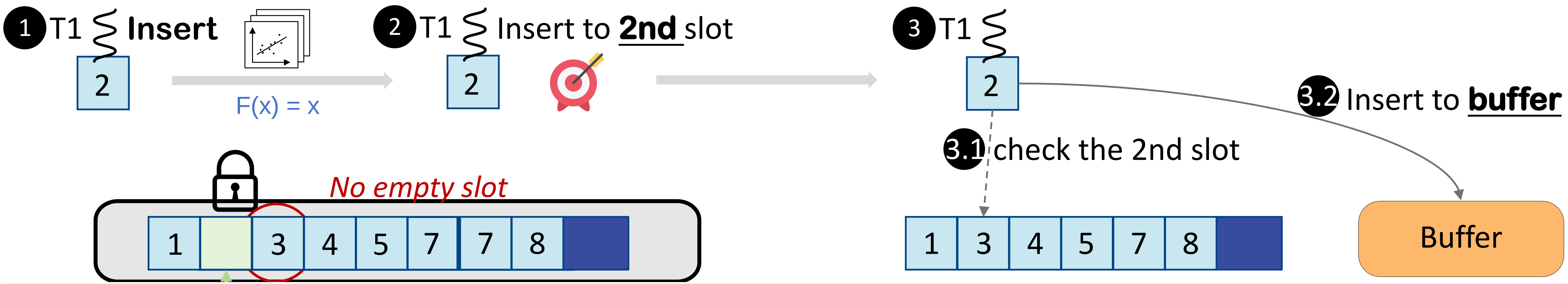
- ✘ ➢ High performance in dynamic workloads

  - High throughputs
  - High scalability

# Challenge 1: Interference from Insertions

- In-place Insertion
  - Model-based insertion
  - Good query performance
- ➢ Lock-based design(ALEX+@VLDB'22)
  - Poor scalability

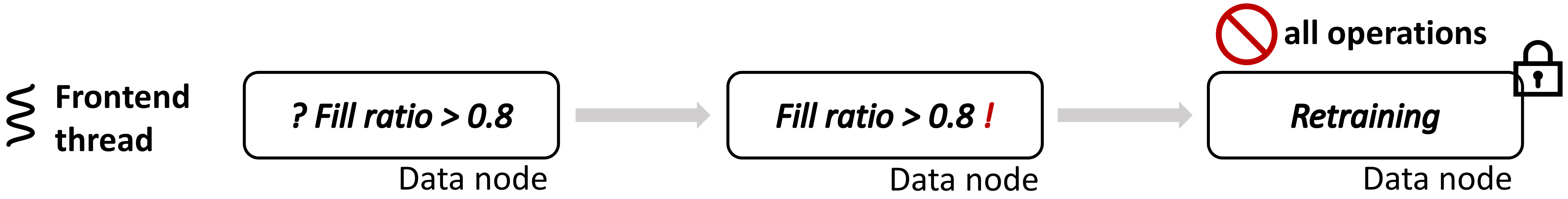
- Out-of-place Insertion
  - Buffer-based insertion
  - Good scalability
- ➢ Buffer-based design(XIndex@PPOPP'20 )
  - Poor query performance




**Existing schemes interfere with concurrent or subsequent reads.**

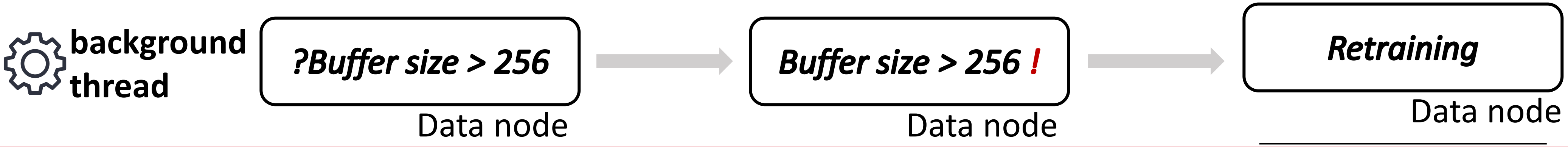
# Challenge 2: Collisions between Indexing and Retraining

- Blocking retraining scheme (ALEX+@VLDB'22)
  - Actively trigger retraining once the condition is met — timely
  - Block the following operations to the retrained node
  - In the critical path — Long tail latency



# Challenge 2: Collision between Indexing and Retraining

- Blocking retraining scheme (ALEX+@VLDB'22)
  - Actively trigger retraining once the condition is met — **Timely**
  - Block the following operations to the retrained node
  - In the critical path — **Long tail latency**
- Non-blocking retraining scheme (XIndex@PPOPP'20)
  - Periodically check the data nodes using background threads — **Non-blocking**
  - Unable to handle heavy tasks in write-intensive workloads — **Long average latency**




**How to achieve in-time and lightweight retraining?**

# Challenge 3: Fixed Parameters vs Diverse Access Patterns

Write-intensive

- **Static triggering mechanism:**
  - Perform retraining once the predefined condition is met

**Trigger retraining in advance?**

Write-intensive

- **Static retraining parameters**
  - Use fixed parameters based on preliminary experiments

**Preserve more free slots in advance?**

# Our Solution: LOFT

- **To achieve high performance in dynamic workloads:**
  - **C1**: Interference introduced by insertions

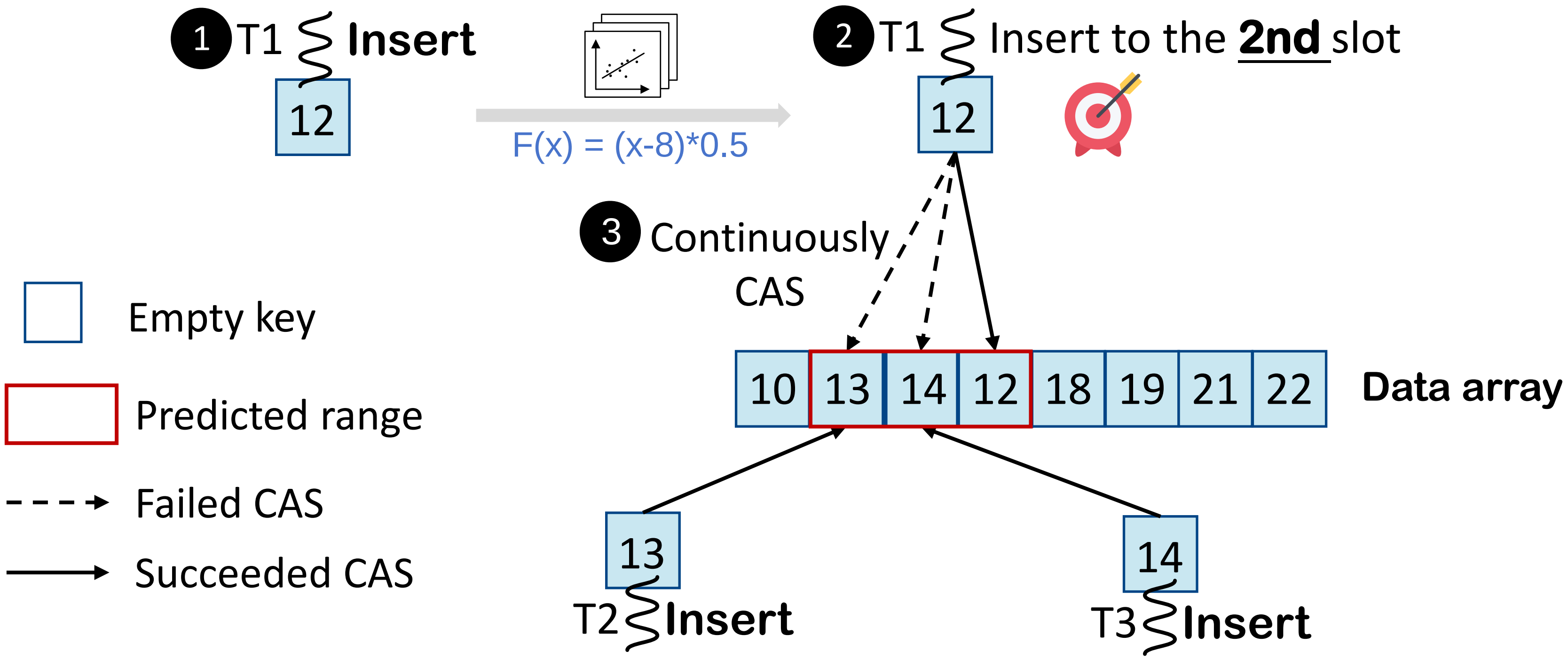
    **Error-bounded insertion**
  - **C2**: Collisions between indexing and retraining

    **Lock-free retraining**
  - **C3**: Mismatch between fixed parameters and diverse access patterns

    **Self-tuning retraining**

# LOFT: Error-bounded Insertion

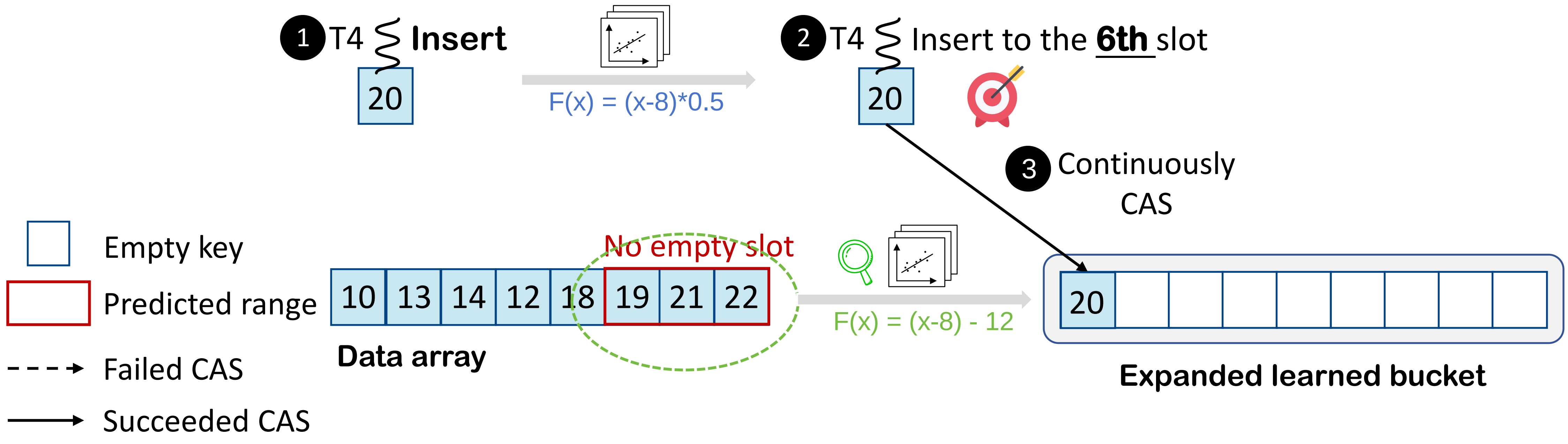
- Using **CAS*** to compete for an empty slot within the predicted range
  - No shifting for sorting
  - No duplicate keys




* Compare-and-Swap

# LOFT: Error-bounded Insertion

- **Expanded Learned Bucket** for possible overflows
  - A small data array with expanded models
  - Increase the expansion factor as the bucket level rises




**Concurrent insert operations are executed in a lock-free manner.**

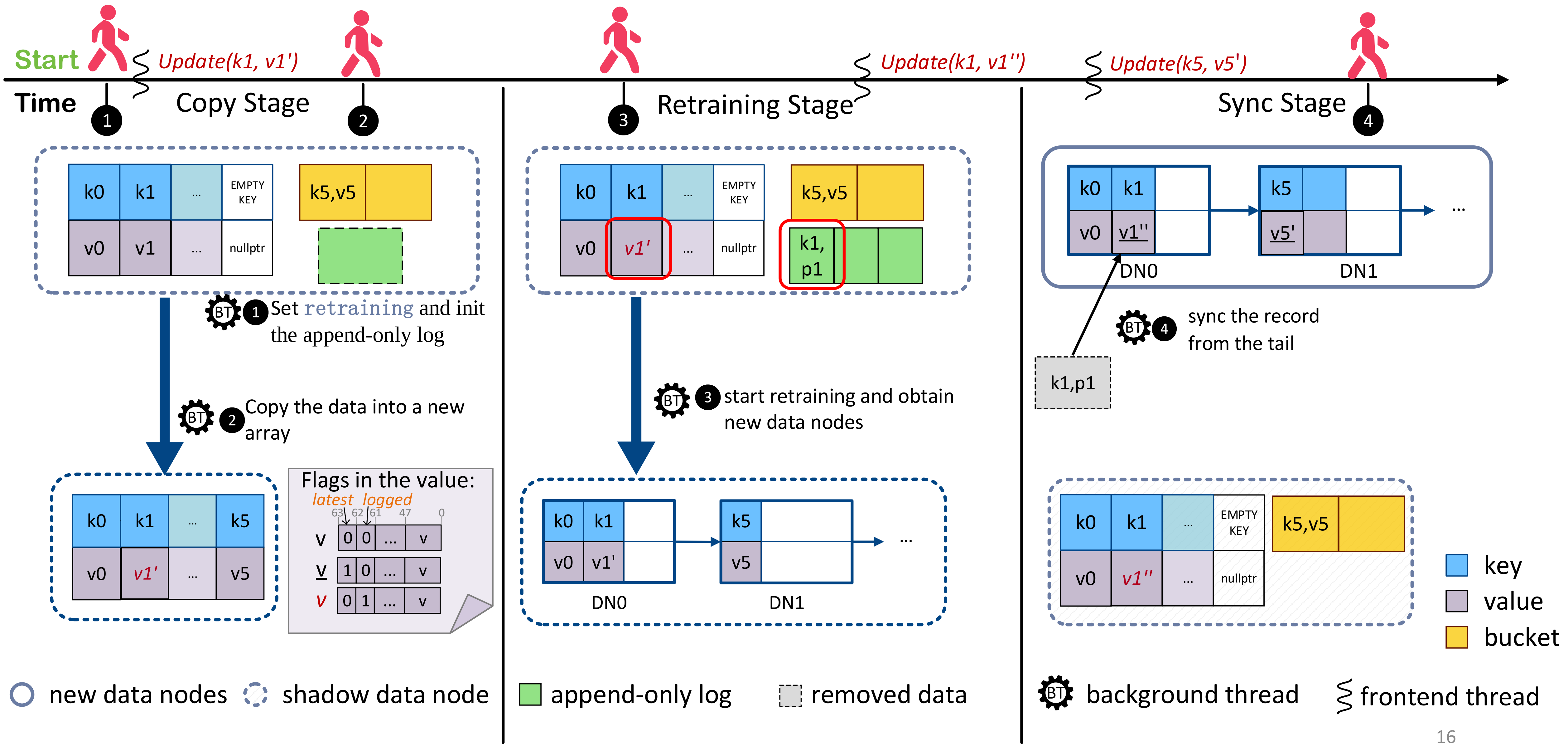
# LOFT: Lock-free Index Operations

➢ <u>Decrease read performance to minimize operation interference</u>

- Read
  - Linear search within the predicted range
  - Reasonable overheads
- In-place update
  - Atomically update the 8-byte value pointers
- Soft delete
  - Maintain the key in the data array
  - Invalidate the value

**All index operations are executed in a lock-free manner.**

# LOFT: Non-blocking Retraining Process



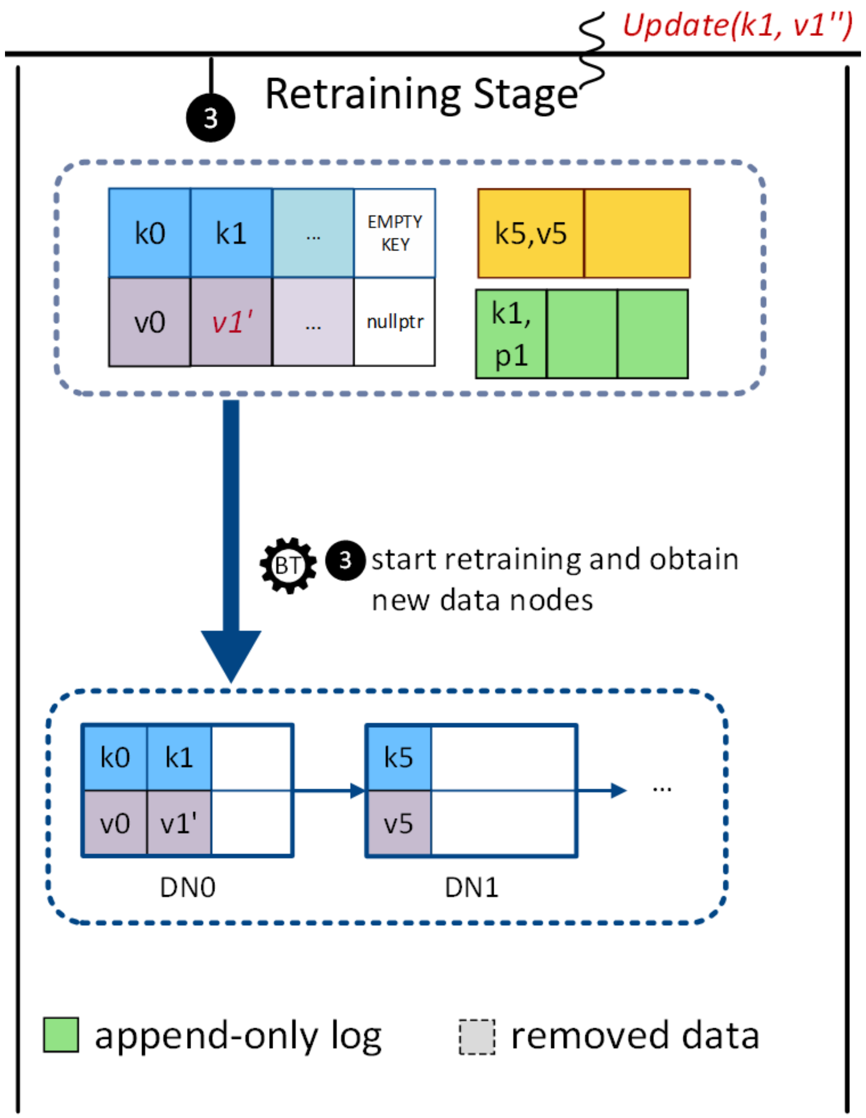
# LOFT: Self-tuning Retraining

- **Write-intensive**
  - Increase the expansion factor of data nodes
  - Increase the predicted range

  Retraining frequency
- **Cold**
  - Decrease the expansion factor
  - Increase the predicted range

  Index size
- **Read-intensive**
  - Decrease the predicted range

  Search length



# More Details about LOFT

- Concurrency correctness
- Structure modification operations
- Informed decision making
- ……


EuroSys '25, March 30-April 3, 2025, Rotterdam, Netherlands

Yuxuan Mo and Yu Hua

the set conditions are met. For example, ALEX+ triggers retraining when the fill ratio of the data node exceeds the threshold. The blocking retraining process decreases the performance, because the index is unable to access items in the retrained data node until the new models are ready. To reduce the overhead of retraining, FINEdex [32] proposes a fine-grained retraining technique, which only retrains the records in the level-bin (a small B⁺-tree) to obtain a new small data node and blocks all reads and insertions to the retrained level-bin. ALEX+ incurs 6.2× tail latency and FINEdex has 2.3× with 24 threads due to the blocking retraining, while XIndex only shows 1.5× tail latency by using non-blocking retraining (Figure 3). To reduce the tail latency, we need to remove the retraining process from the critical path.

Non-blocking retraining in XIndex however becomes a hurdle to gain high performance, when the retrained tasks are not processed promptly. XIndex retrains data nodes with buffer sizes larger than 256 records, which is easily achievable even in read-intensive workloads. XIndex needs to retrain almost all data nodes using a single worker thread as shown in Figure 3. However, the computing resources of background threads are limited. A single background thread is unable to complete such heavy retraining tasks in a short time. Hence, the large buffer size leads to long read latency. As a result, the relative P99.9 read latency of XIndex is 5.3×, when the thread number is 1, as shown in Figure 3.

**Mismatch between Fixed Parameters and Various Access Patterns.** All the parameters for retraining are preset and fixed. For example, the training prediction error is a tradeoff between prediction accuracy and the model numbers based on the evaluation results. ALEX+ determines whether to retrain by comparing the fill ratio of the data node with a fixed threshold. However, different workloads have different request distributions. We need to customize the parameters for each data node under different workloads with reasonable overheads and thereby achieve higher performance. For instance, the hot data node with frequent read operations can obtain higher prediction accuracy for higher read throughput, while the cold data node with rare data accesses can preserve fewer free slots for memory saving. However, it is inefficient to pause the clients' requests and then manually modify the parameters. In order to be easy-to-use and adaptive, the learned index needs to be self-tuning depending on the access patterns. Unfortunately, existing schemes fail to achieve these design goals.

## 3 The LOFT Design

### 3.1 Overview

We propose LOFT, an adaptive and lock-free learned index designed for high scalability in dynamic workloads. Figure 4 shows the overall architecture of LOFT, which contains two layers: one root node and multiple data nodes. The root node

**Figure 4.** The overall architecture of LOFT.

consists of a two-stage RMI model and a collection of pointers to the data nodes. Each data node handles distinct key ranges without overlaps. For each client request, LOFT utilizes the RMI in the root node to locate the appropriate data node, followed by performing the index operation within the data node using the corresponding linear models. Since all index operations follow a uniform process at the root node level, our primary focus lies on the structures and techniques related to the data nodes. Specifically, to mitigate the interference brought by insertions, LOFT employs an error-bounded insertion mechanism that places new items into their predicted positions and uses expanded learned buckets to manage the overflowed items so that all index operations can be executed in a lock-free manner. We present the structure of the expanded learned bucket and demonstrate how LOFT carries out index operations concurrently and correctly without locks in §3.2. To alleviate the collision between the indexing and retraining, LOFT introduces a shadow data node to serve the clients' requests, while allowing clients to contribute to the retraining process. §3.3 outlines the retraining workflow and describes how index operations proceed during retraining. Moreover, LOFT maintains essential statistics at low costs, making it workload-aware and enabling adaptive retraining. §3.4 presents how to handle retraining tasks based on an informed decision-making strategy. Finally, we demonstrate the concurrency correctness of index operations in §3.5.

### 3.2 Lock-free Index Operations

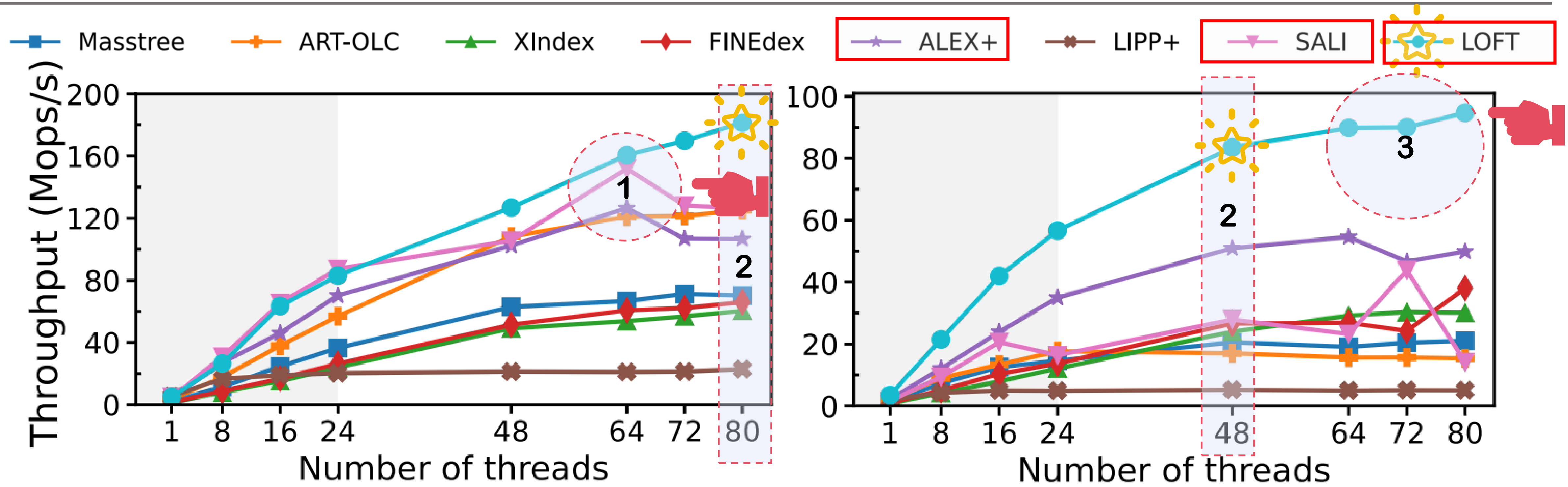
LOFT supports common operations in traditional index structures, including `read`, `insert`, `update`, `delete` and `scan`. We omit the repeated details of using the RMI model in the root node to reach the data node. We present the procedures of these operations upon data nodes without performing structure modification operations (SMOs) in this subsection and with performing SMOs in §3.3.

Index operations are closely related to data node initialization since this process determines the record placement. We hence start with node initialization. Piecewise Linear Approximation (PLA) algorithm [18] is employed to obtain the linear models within the data nodes. Consider a linear model for $N$ keys, where $a$ represents the slope, $K_1$ is the smallest key, and $x$ denotes the given lookup key. This model

# Experimental Setup

- **Testbeds**
  - Two 26-core Intel(R) Xeon(R) CPU @2.10GHz
  - Assign one background thread to every twelve worker threads
- **Workloads**
  - YCSB with Zipfian distribution
  - Multiple real-world datasets
- **Comparisons**
  - *Conventional:* Masstree [Eurosys'12], ART-OLC [DaMoN '16]
  - *Learned:* DyTIS [Eurosys'23], XIndex [PPOPP'20], FINEdex[VLDB'21], ALEX+ [VLDB'22], LIPP+ [VLDB'22], SALI [SIGMOD'23]

# Evaluation on Scalability




Read-intensive workload

Write-intensive workload

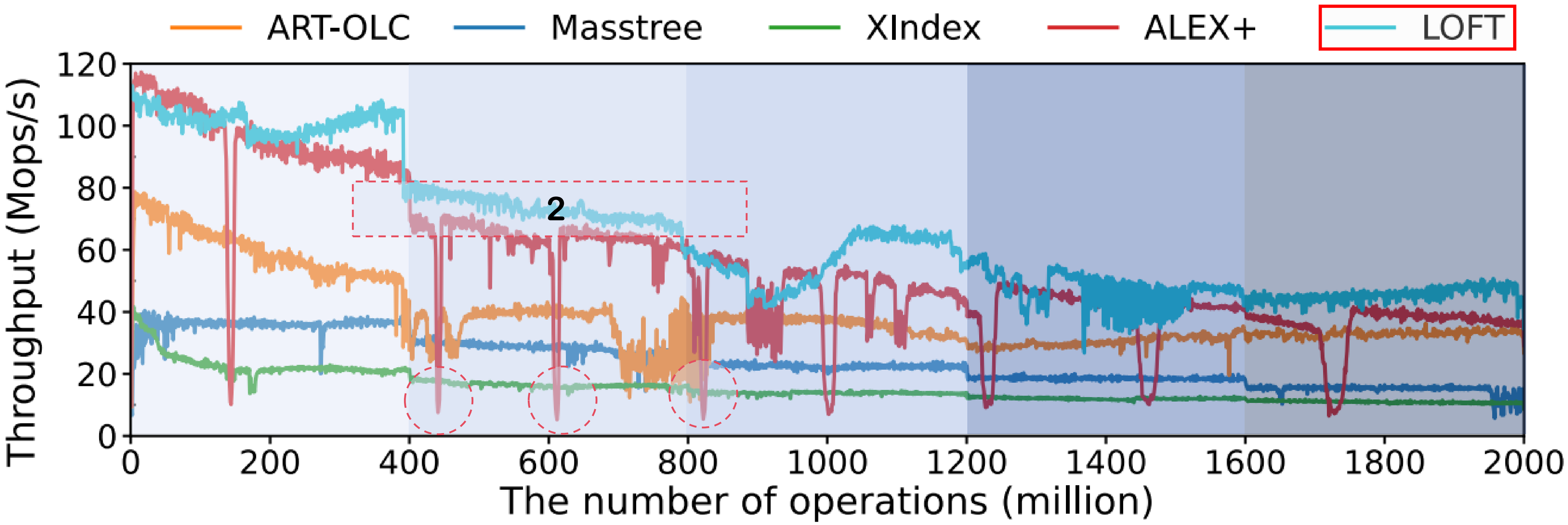
1. Due to the **in-place insertion** design, ALEX+, SALI and LOFT achieve higher throughput.

2. Due to the **lock-free** design, LOFT achieves the best scalability.

3. LOFT improves the throughput by **1.7x – 14x** on average.

IA node

# Evaluation on Adaptiveness




1. The average throughputs of all indexes **decline** as the proportion of insertions **increases**.

2. Our **lock-free retraining** scheme enables LOFT to avoid severe performance jitter.

3. LOFT illustrates long-term stability thanks to **self-tuning retraining** mechanism.

# Summary

- Existing learned indexes show limited scalability in dynamic workloads.
  - Display sharp performance degradation
  - Fail to simultaneously achieve high throughput and high scalability
- **LOFT**: a Lock-free and scalable learned index.
  - Error-bounded insertion scheme
  - Lock-free index operations and retraining process
  - Self-tuning retraining mechanism
- **LOFT** significantly improves the throughput with high scalability compared with state-of-the-art schemes.

Open-source address: https://github.com/yuxuanMo/LOFT.git

# Thanks!

## Q & A

yuxuanmo@hust.edu.cn