ALT-index: A Hybrid Learned Index for Concurrent Memory Database Systems

Yuxin Yang[†], Fang Wang[†], Mengya Lei[‡], Peng Zhang[†], Dan Feng[†]

[†]*Huazhong University of Science and Technology*
[‡]*Hubei University of Technology*

41st IEEE International Conference on Data Engineering, Hong Kong SAR, China, May 19-23, 2025

Outline

- ❖ **Background**
- ❖ **Motivation**
- ❖ **Design**
- ❖ **Evaluation**

Database System

- **Disk-Based Database**
  - Low performance with large capacity
  - Disk friendly index structure (e.g. B+tree)

- **Memory-Based Database**
  - High performance with limited space
  - Efficient in-memory index structure (e.g. Adaptive Radix Tree)

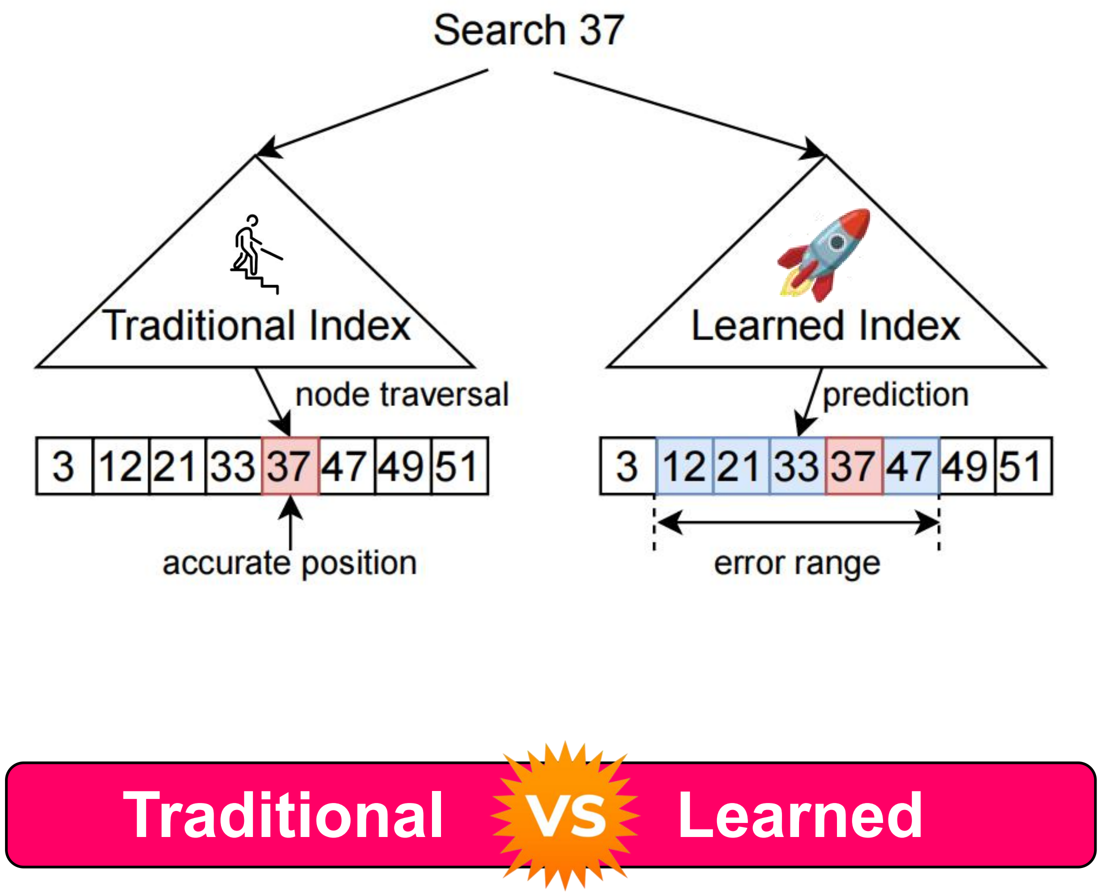
In-Memory Index Structure

❖ **Traditional Index (Node based)**

- **Slow** lookup performance
- **High** space overhead
- **Efficient** SMOs for insert

❖ **Learned Index (Model based)**

- **Fast** lookup performance
- **Low** space consumption
- **Costly** retraining for insert



C1: Can we burn the candle at both ends?

❖ Adaptive Radix Tree

- ➢ **Slow** lookup
- ➢ **High** memory consumption
- ➢ **Good** insert performance

	Lookup-only (MOPS/s)	Insert-only (MOPS/s)
★ALEX+	136.65	26.33
★LIPP+	173.54	4.02
★FINEdex	62.91	18.08
★XIndex	70.75	14.60
*ART-OLC	90.80	34.32

★learned indexes *traditional indexes

❖ Learned Index

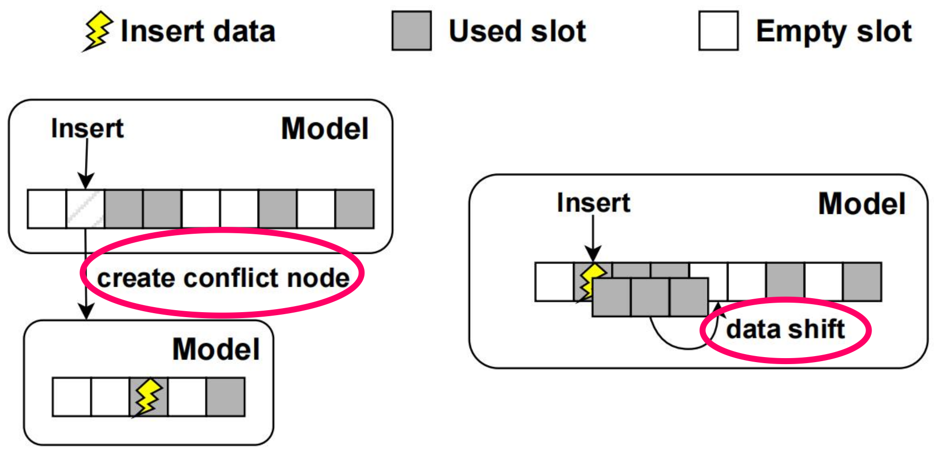
- ➢ **Fast** lookup
- ➢ **Low** memory consumption
- ➢ **Bad** insert performance

A: Hybrid Learned Index !

C2: Limitation of Learned Index

❖ Prediction Errors in Existing Learned Index

➢ Write bottleneck

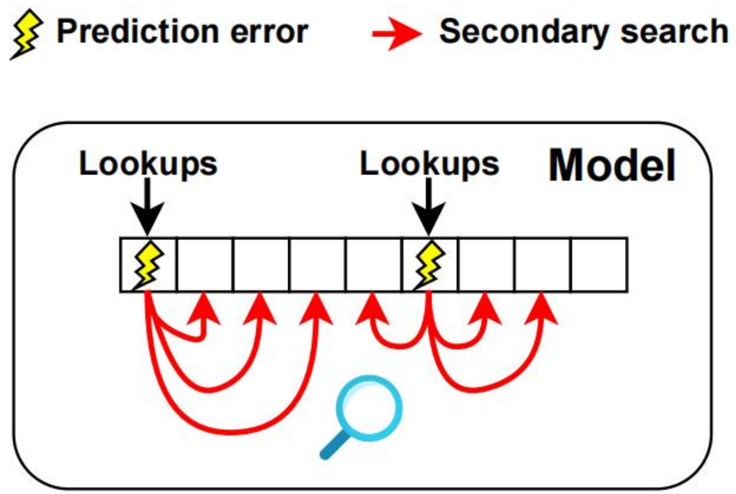


LIPP+ ALEX+

Write amplification

Create conflict node: 40.7% Data shift: 25.2%

➢ Secondary search bottleneck



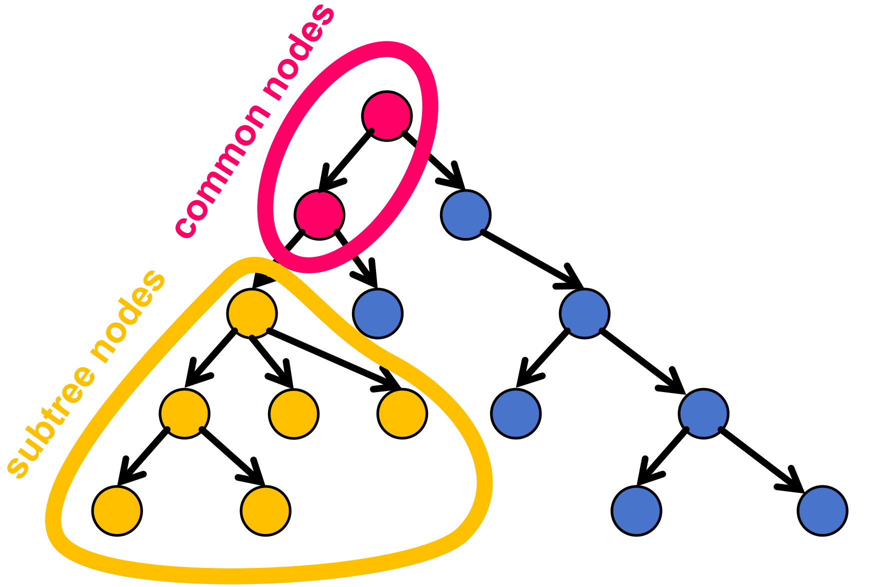
Secondary search

FINEdex & XIndex perform poor in lookups.
ALEX+ saturate the memory at 16~32 threads!

C3: Limitation of ART

❖ Complex Issues in ART

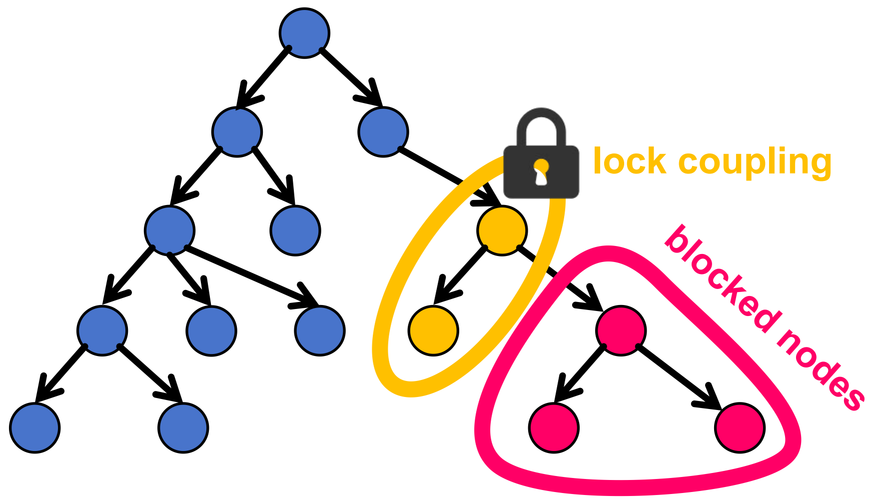
- ➢ Long traversal length



Long length traversal

All subtree nodes traverse common nodes!

- ➢ Block problem



Blocked sibling nodes

Insert with lock coupling scheme blocks nodes!

Our Solution: ALT-index

❖ **ALT-index Overview**

① Hybrid Construction

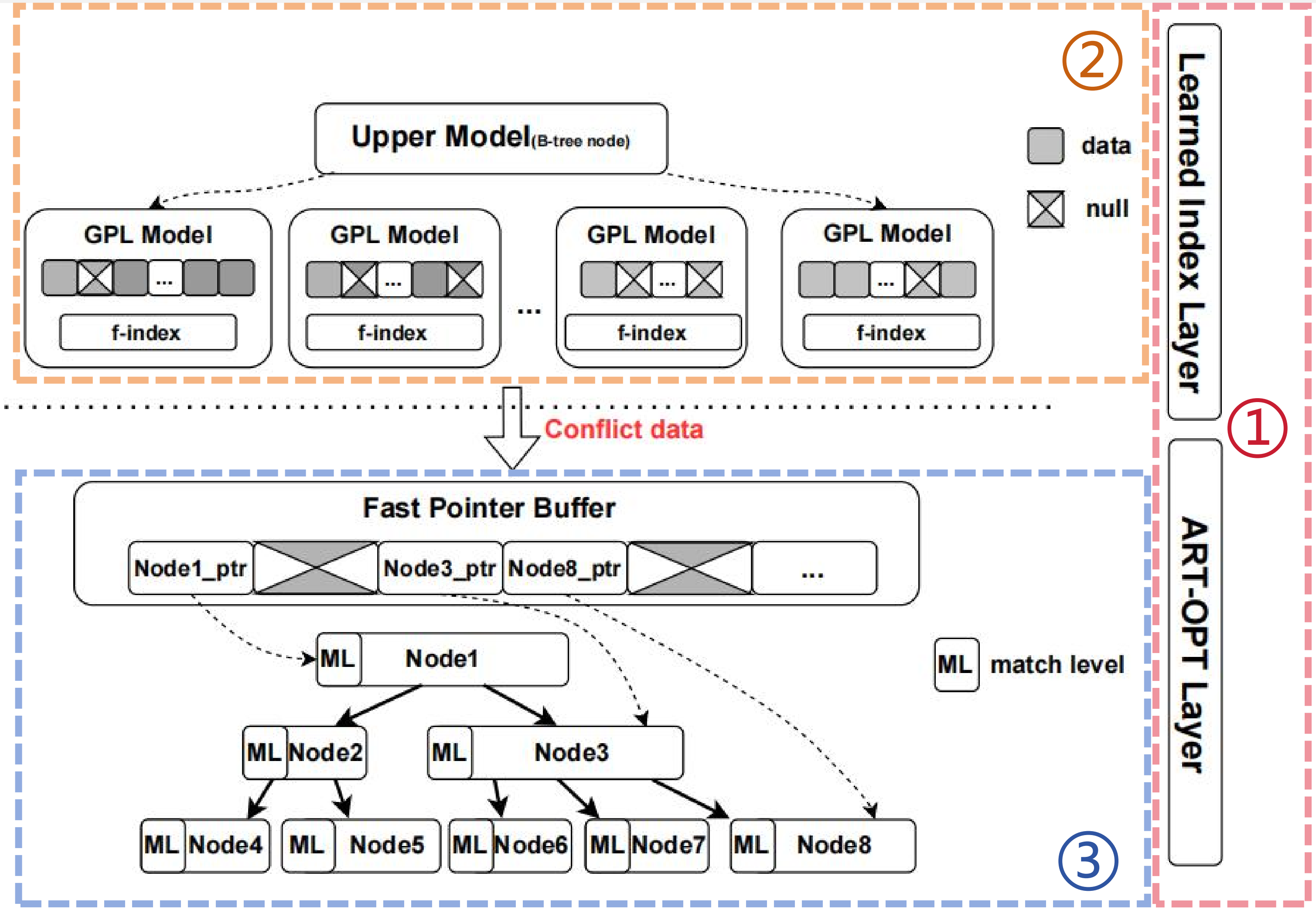
- ➢ Two layer design
- ➢ Conflict dataflow

② Learned Index Layer

- ➢ GPL algorithm
- ➢ Dynamic retraining

③ Optimized ART Layer

- ➢ Fast pointer buffer

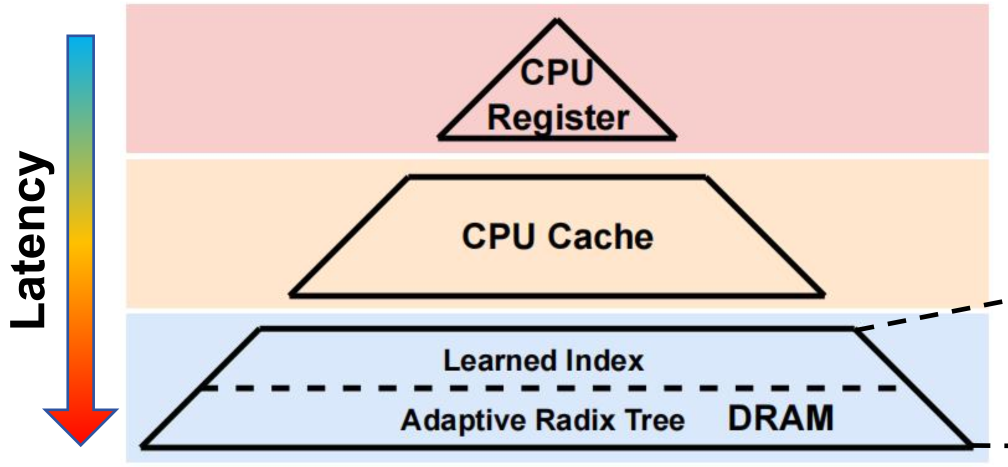


O1: Hybrid Construction

❖ Two-layer Design

- ➢ Learned Index Layer
  - ✓ **Lookup** operations
  - ✓ **In-place** insert operations

- ➢ Adaptive Radix Tree Layer
  - ✓ Lookup operations **after insert**
  - ✓ **Out-of-place** insert operations



	Lookup	In-place insert	Out-of-place insert
Learned Index	Fast √	Fast √	Slow ×
ART	Slow ×	None ×	Fast √

O1: Hybrid Construction

❖ **Conflict Dataflow**

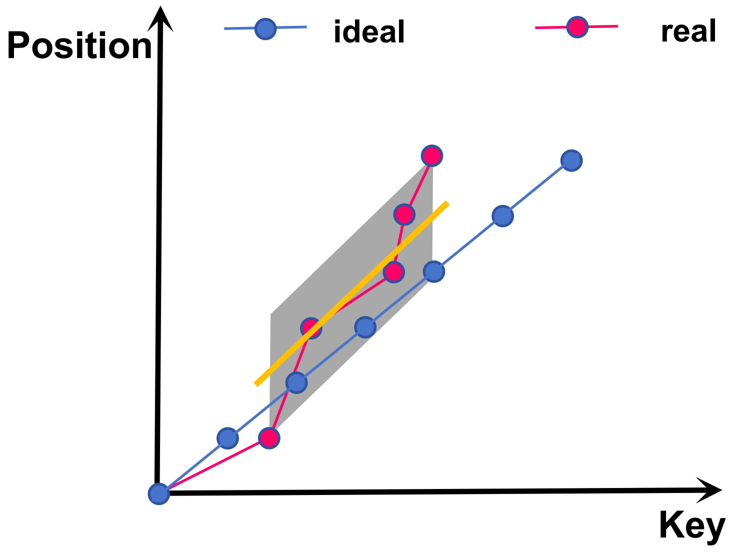
- Learned Index Layer
- Adaptive Radix Tree Layer

Decouple operations to the corresponding layer

O2 : Learned Index Layer

❖ Segment Algorithm

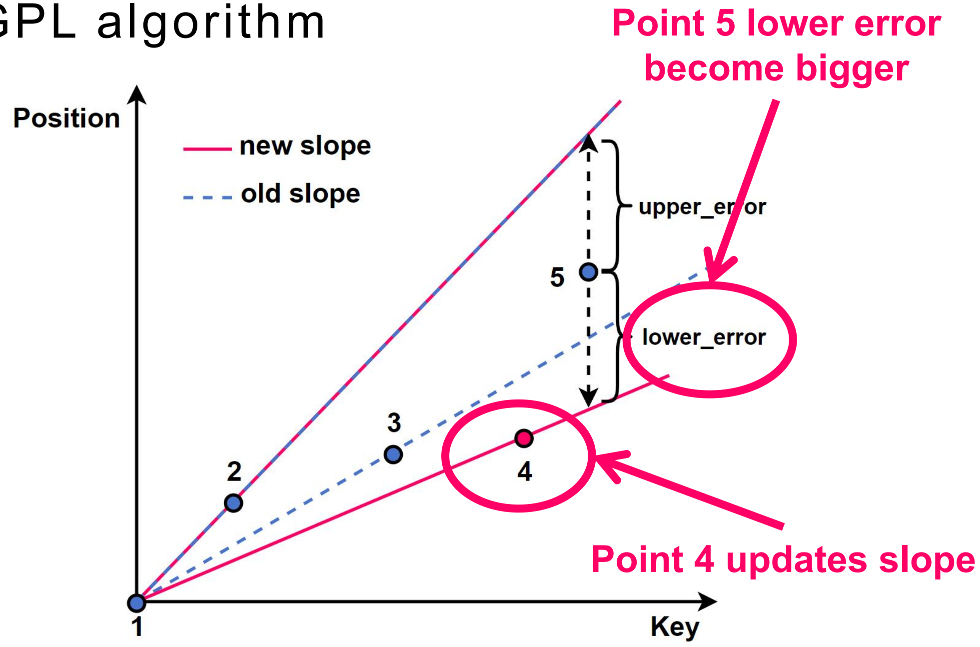
➢ Old algorithm



Consider max error

Over 50% data have prediction conflicts on avg.

➢ GPL algorithm

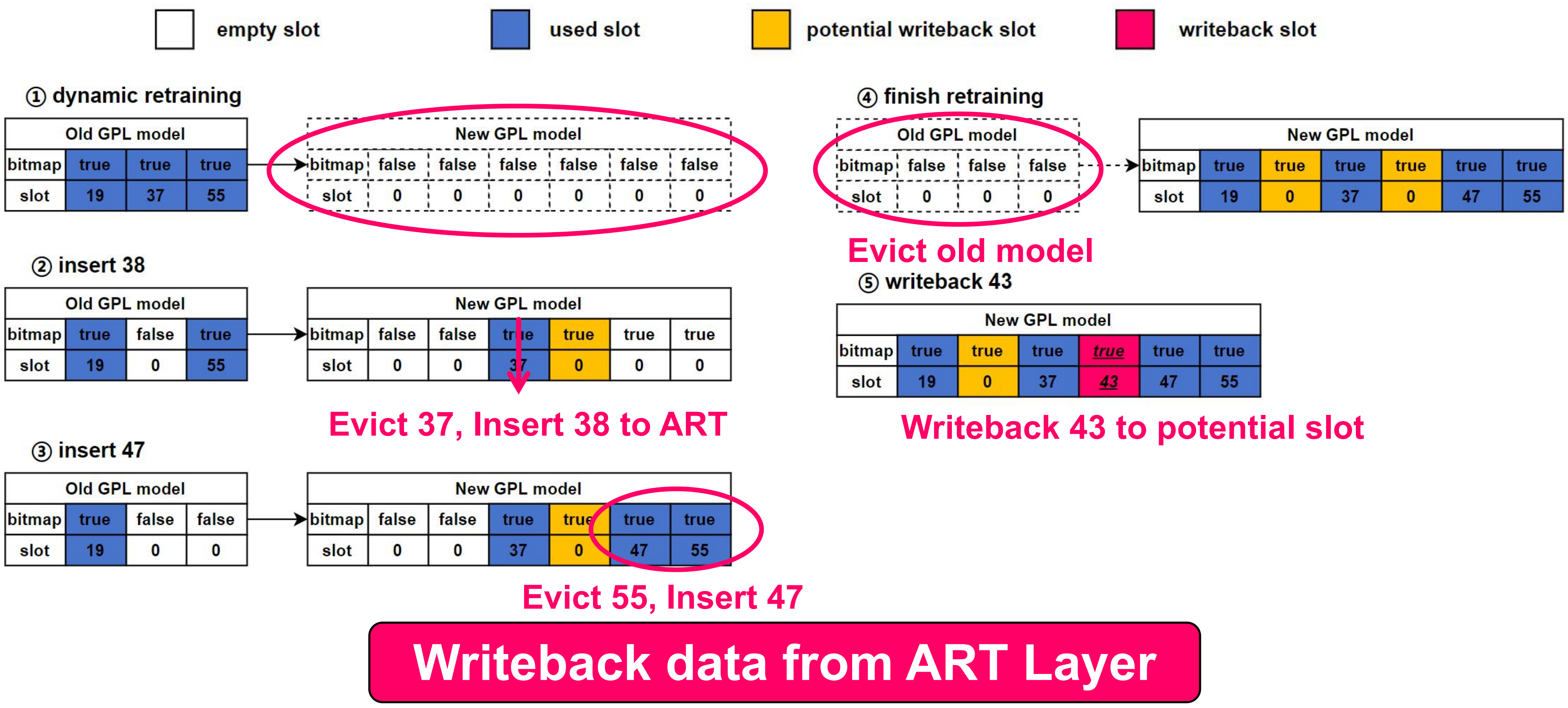


Consider every error

Only 30% data have prediction conflicts on avg.

O2 : Learned Index Layer

❖ Dynamic Retaining

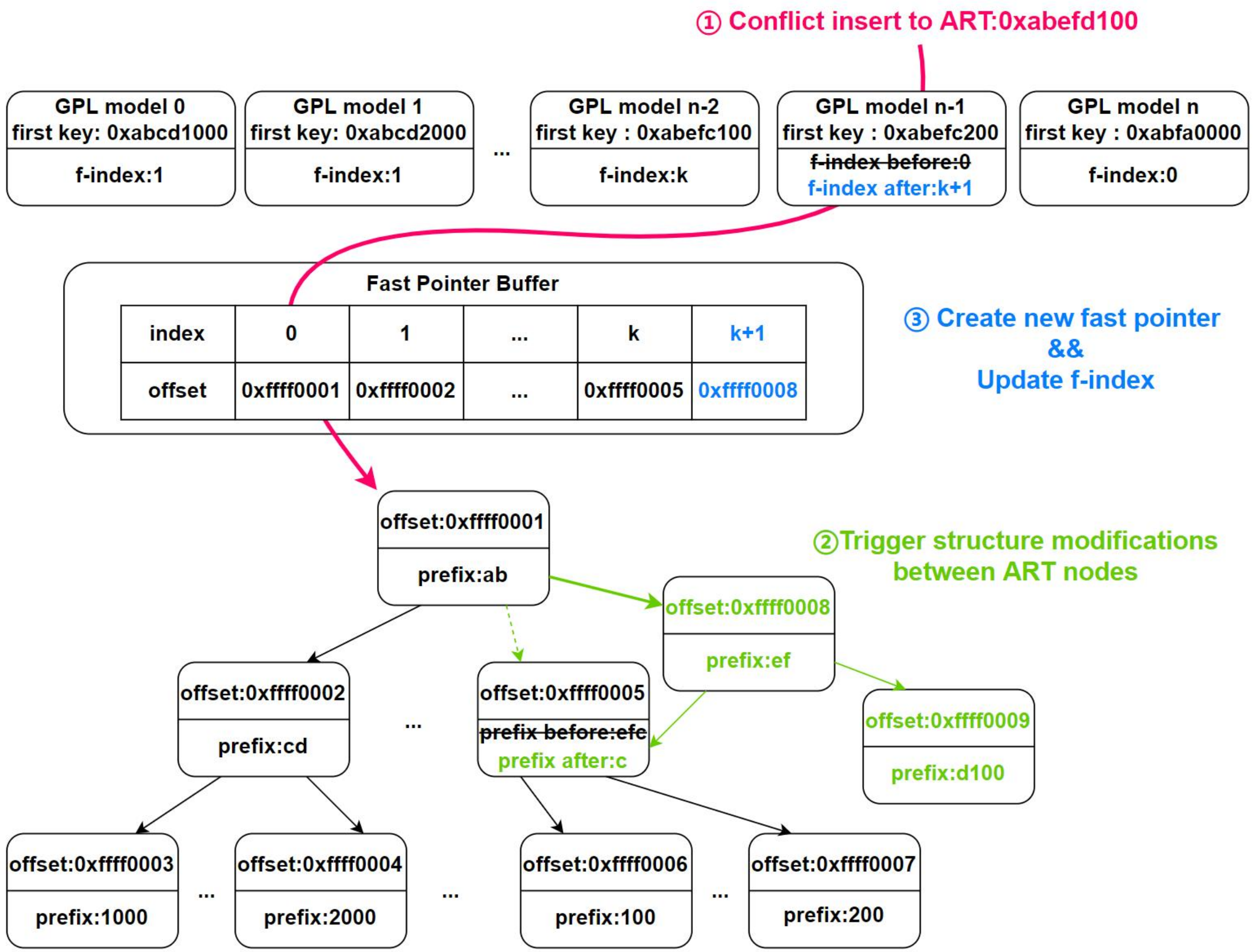


O3 : Optimized ART Layer

❖ Fast Pointer Buffer Scheme

- ➢ **Build** fast pointers
  - ✓ Cut down traversal length
- ➢ **Merge** duplicated fast pointers
  - ✓ Space efficient
  - ✓ Data consistency for ART SMOs

Efficient ART traversal



Evaluation

❖ **Environment**

➢ **Hardware**

- ✓ Intel Xeon Gold 6240@2.60GHz × 2
- ✓ 186GB DDR4 Memory

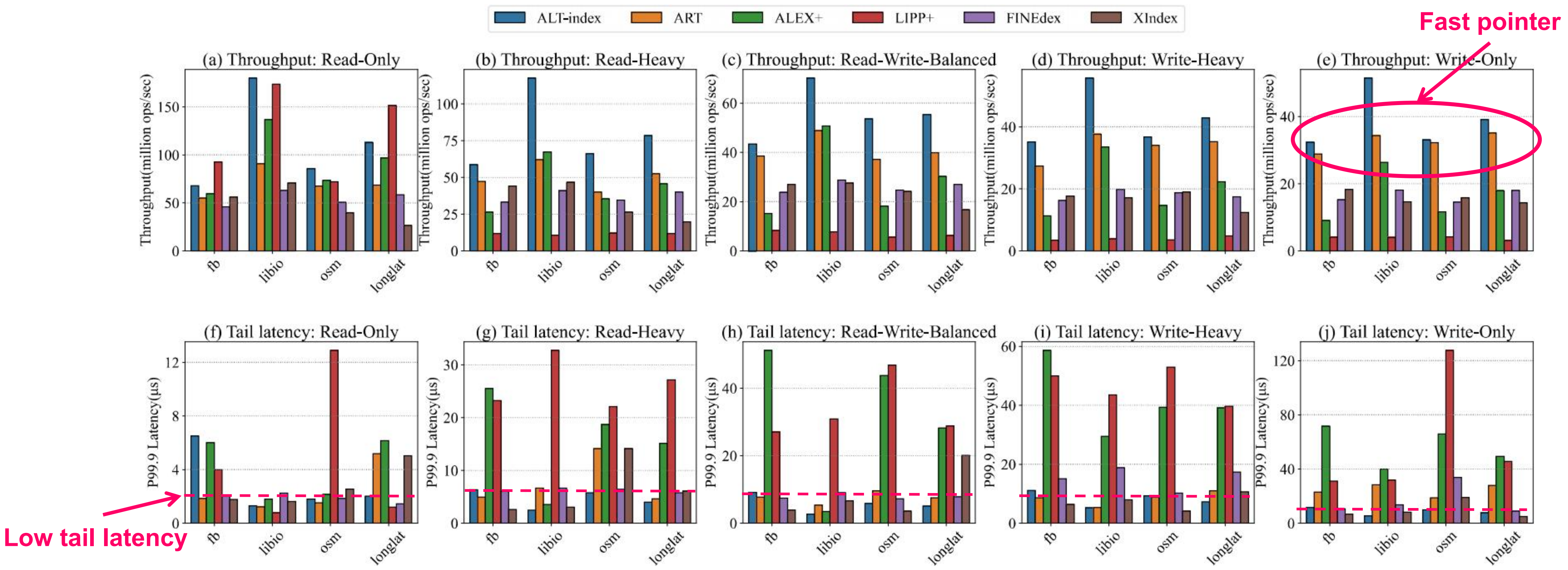
➢ **Software**

- ✓ GCC 9.4.0, CMAKE 3.16 with O3 optimization
- ✓ 4 real-world datasets

➢ **Competitors**

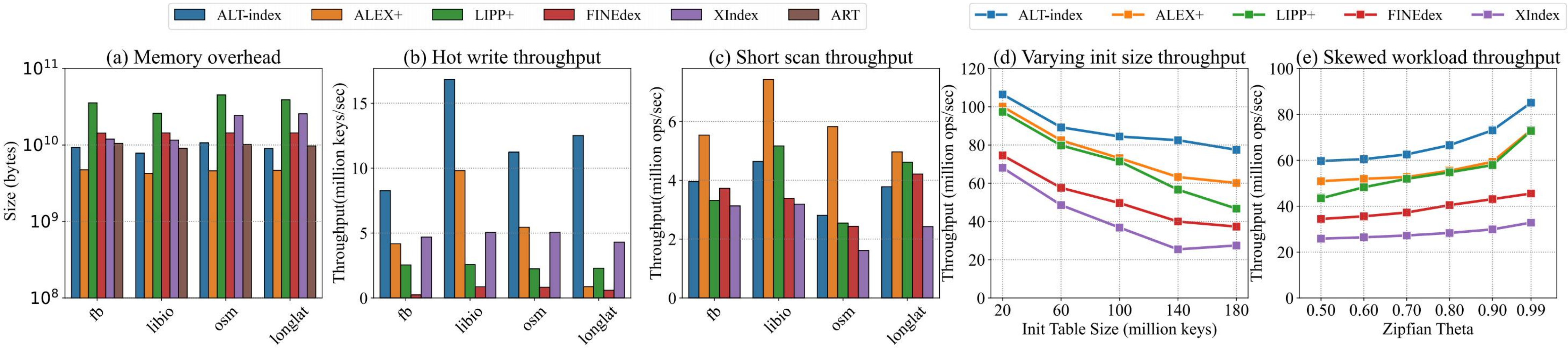
- ✓ ART [DaMoN ‘16]
- ✓ ALEX+ [VLDB’22]
- ✓ LIPP+ [VLDB’22]
- ✓ FINEdex [VLDB’21]
- ✓ XIndex [PPoPP’21]

Throughput & Tail Latency



ALT-index improves the throughput by 1.9-2.3x on average. 😇

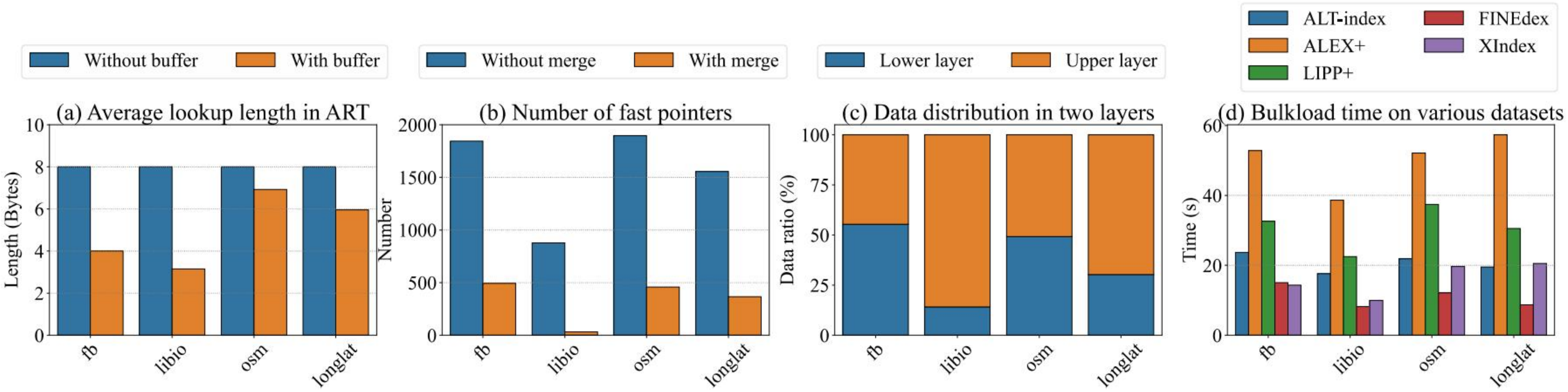
Robustness



Memory Overhead | **Hot Write** | **Short Scan** | **Init Size** | **Skewness**

ALT-index performs <u>Good Robustness</u> under different scenarios.

Other Analysis

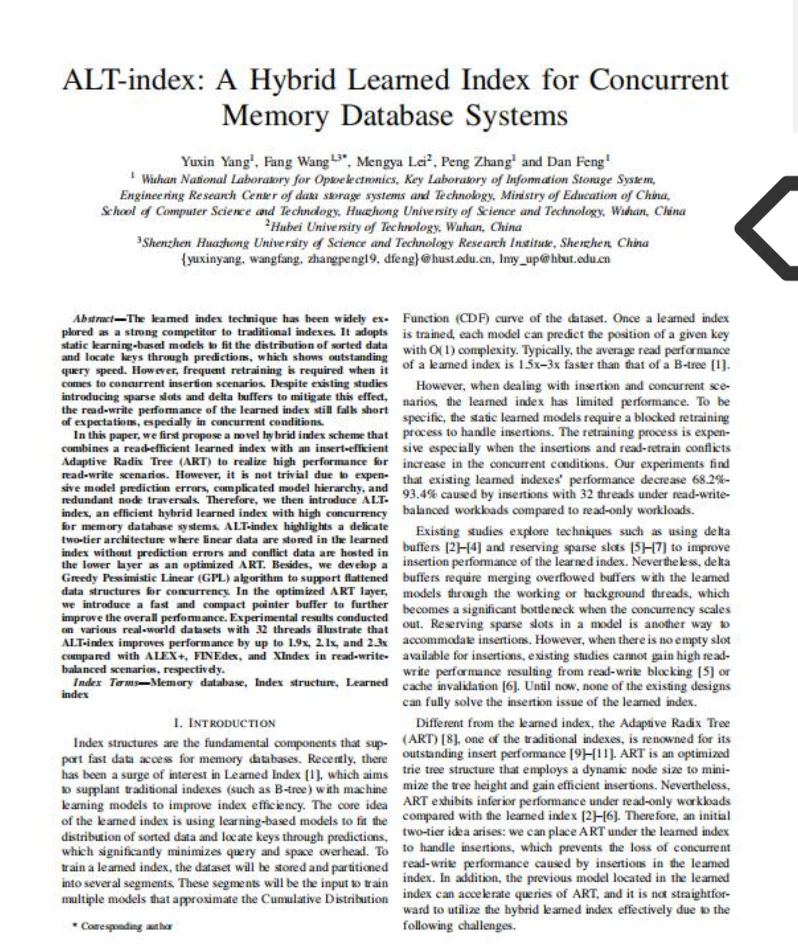


Lookup Length | **Merge Scheme** | **Data Distribution** | **Bulkload Time**

ALT-index minimizes the expense of hybrid design.

Conclusion

- ❖ **Traditional Index 🤝 Learned Index**
  - ➢ ART has good write performance
  - ➢ Learned Index has good read performance
- ❖ **ALT-index Design: A Hybrid Learned Index**
  - ➢ Two-layer construction
  - ➢ Optimized learned index layer
  - ➢ Optimized ART layer
- ❖ **ALT-index** can take full advantage of both **Learned Index** and **ART**



ALT-index: A Hybrid Learned Index for Concurrent Memory Database Systems

Yuxin Yang[1], Fang Wang[1,3*], Mengya Lei[2], Peng Zhang[1] and Dan Feng[1]

[1] *Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Engineering Research Center of data storage systems and Technology, Ministry of Education of China, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China*

[2] *Hubei University of Technology, Wuhan, China*

[3] *Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen, China*

{yuxinyang, wangfang, zhangpeng19, dfeng}@hust.edu.cn, lmy_up@hbut.edu.cn

***Abstract*—The learned index technique has been widely explored as a strong competitor to traditional indexes. It adopts static learning-based models to fit the distribution of sorted data and locate keys through predictions, which shows outstanding query speed. However, frequent retraining is required when it comes to concurrent insertion scenarios. Despite existing studies introducing sparse slots and delta buffers to mitigate this effect, the read-write performance of the learned index still falls short of expectations, especially in concurrent conditions.**

In this paper, we first propose a novel hybrid index scheme that combines a read-efficient learned index with an insert-efficient Adaptive Radix Tree (ART) to realize high performance for read-write scenarios. However, it is not trivial due to expensive model prediction errors, complicated model hierarchy, and redundant node traversals. Therefore, we then introduce ALT-index, an efficient hybrid learned index with high concurrency for memory database systems. ALT-index highlights a delicate two-tier architecture where linear data are stored in the learned index without prediction errors and conflict data are hosted in the lower layer as an optimized ART. Besides, we develop a Greedy Pessimistic Linear (GPL) algorithm to support flattened data structures for concurrency. In the optimized ART layer, we introduce a fast and compact pointer buffer to further improve the overall performance. Experimental results conducted on various real-world datasets with 32 threads illustrate that ALT-index improves performance by up to 1.9x, 2.1x, and 2.3x compared with ALEX+, FINEdex, and XIndex in read-write-balanced scenarios, respectively.

***Index Terms*—Memory database, Index structure, Learned index**

I. INTRODUCTION

Index structures are the fundamental components that support fast data access for memory databases. Recently, there has been a surge of interest in Learned Index [1], which aims to supplant traditional indexes (such as B-tree) with machine learning models to improve index efficiency. The core idea of the learned index is using learning-based models to fit the distribution of sorted data and locate keys through predictions, which significantly minimizes query and space overhead. To train a learned index, the dataset will be stored and partitioned into several segments. These segments will be the input to train multiple models that approximate the Cumulative Distribution Function (CDF) curve of the dataset. Once a learned index is trained, each model can predict the position of a given key with O(1) complexity. Typically, the average read performance of a learned index is 1.5x–3x faster than that of a B-tree [1].

However, when dealing with insertion and concurrent scenarios, the learned index has limited performance. To be specific, the static learned models require a blocked retraining process to handle insertions. The retraining process is expensive especially when the insertions and read-retrain conflicts increase in the concurrent conditions. Our experiments find that existing learned indexes' performance decrease 68.2%-93.4% caused by insertions with 32 threads under read-write-balanced workloads compared to read-only workloads.

Existing studies explore techniques such as using delta buffers [2]–[4] and reserving sparse slots [5]–[7] to improve insertion performance of the learned index. Nevertheless, delta buffers require merging overflowed buffers with the learned models through the working or background threads, which becomes a significant bottleneck when the concurrency scales out. Reserving sparse slots in a model is another way to accommodate insertions. However, when there is no empty slot available for insertions, existing studies cannot gain high read-write performance resulting from read-write blocking [5] or cache invalidation [6]. Until now, none of the existing designs can fully solve the insertion issue of the learned index.

Different from the learned index, the Adaptive Radix Tree (ART) [8], one of the traditional indexes, is renowned for its outstanding insert performance [9]–[11]. ART is an optimized trie tree structure that employs a dynamic node size to minimize the tree height and gain efficient insertions. Nevertheless, ART exhibits inferior performance under read-only workloads compared with the learned index [2]–[6]. Therefore, an initial two-tier idea arises: we can place ART under the learned index to handle insertions, which prevents the loss of concurrent read-write performance caused by insertions in the learned index. In addition, the previous model located in the learned index can accelerate queries of ART, and it is not straightforward to utilize the hybrid learned index effectively due to the following challenges.

* Corresponding author

Q & A