PIMLex: A High-Performance Learned Index with Processing-in-Memory

23rd USENIX Conference on File and Storage Technologies (FAST 2025)

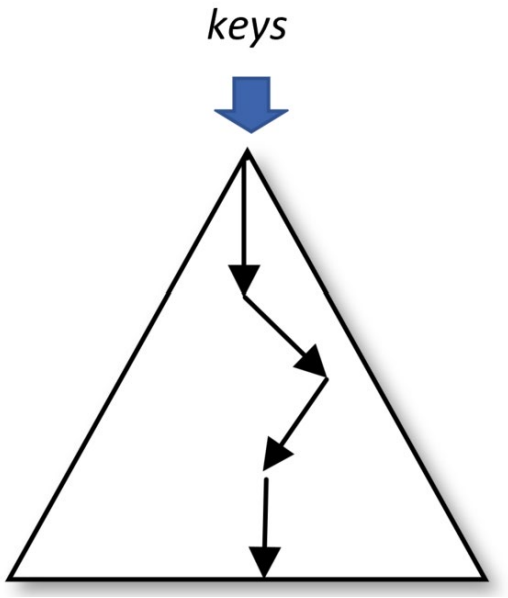
Lixiao Cui, Kedi Yang, Yusen Li, Gang Wang, Xiaoguang Liu

Nankai University

Learned Index

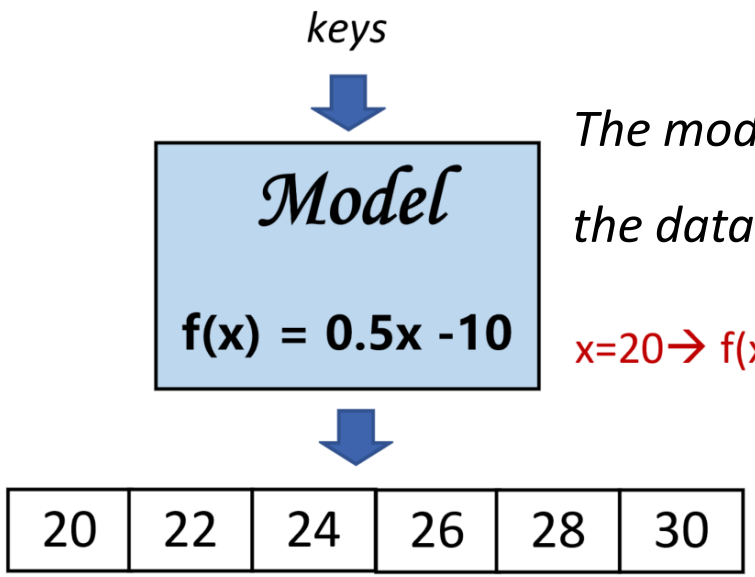
Ordered indexes play important roles in storage systems

Traditional trees (e.g., B+tree)



- **Lookup keys level-by-level**
- **Time Complexity - O(logn)**

Learned indexes



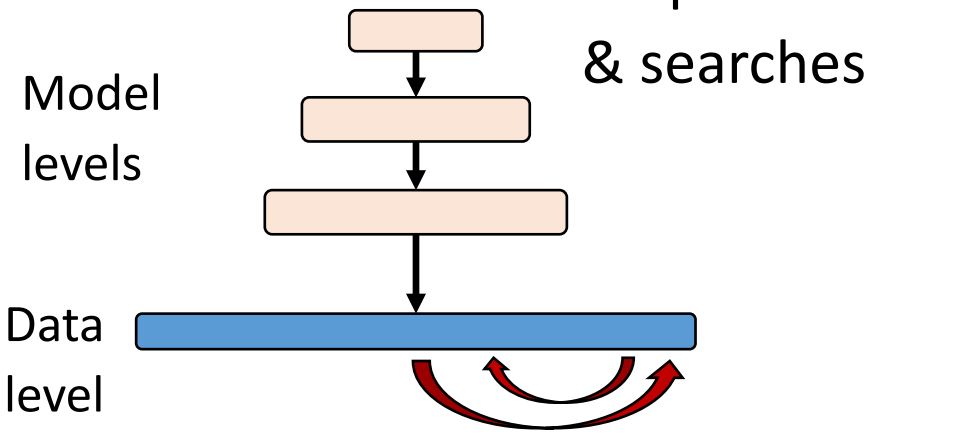
The model f(x) learns the data distribution

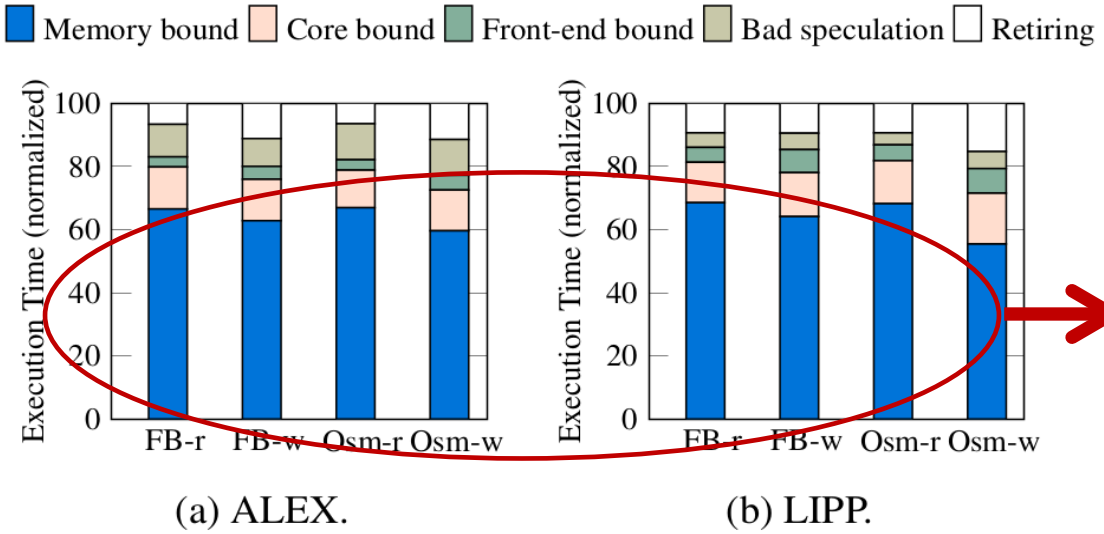
x=20→ f(x) = 0

- **Lookup keys by predictions**
- **Time Complexity - O(1) in best cases**

Learned Index

The model searches and the "last mile" search introduce numerous costly memory accesses.



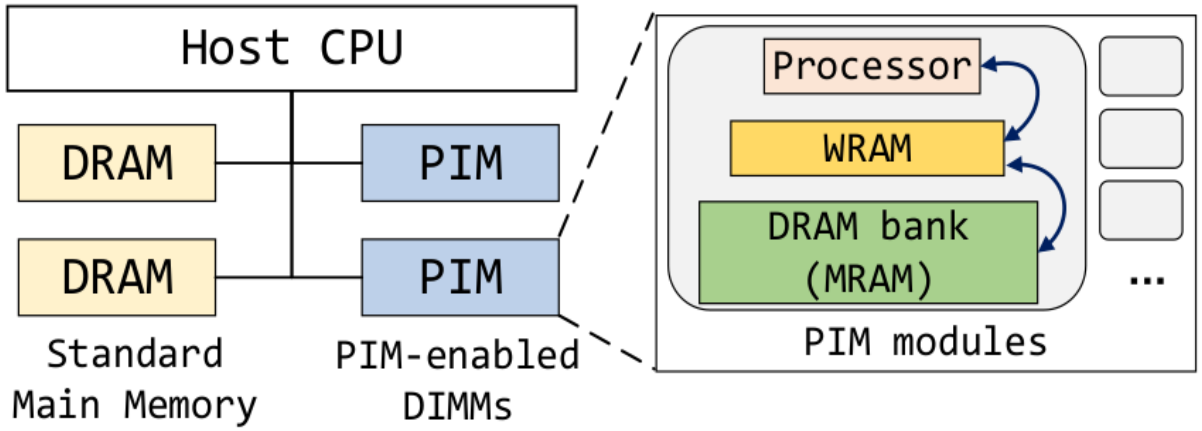


The majority of execution time is caused by **memory bound**

Execution time breakdown for learned index operations.
r represents Get and w represents Insert.

Promising Solution: Processing-in-Memory (PIM)

- Integrate processing capability within memory devices
- Solve the “memory wall” issue



The structure of the UPMEM PIM system.

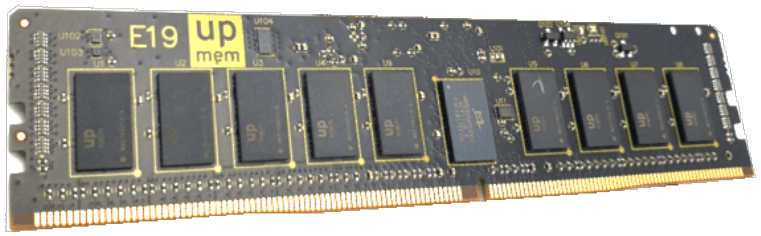
- Reduce costly data movements between memory and CPU
- Powerful memory access capability – 80GB/s per DIMM
- Comprises several PIM modules

Using powerful PIM to overcome memory bound of learned indexes

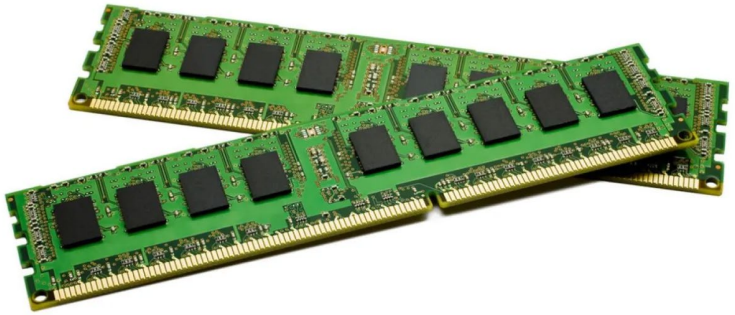
Challenge

① Large space demand vs. Limited capacity of PIM

- Learned indexes require **large space** to process massive amounts of data
- The capacity of PIM is small



UPMEM PIM: only 8GB per DIMM



Conventional DRAM: 64GB per DIMM

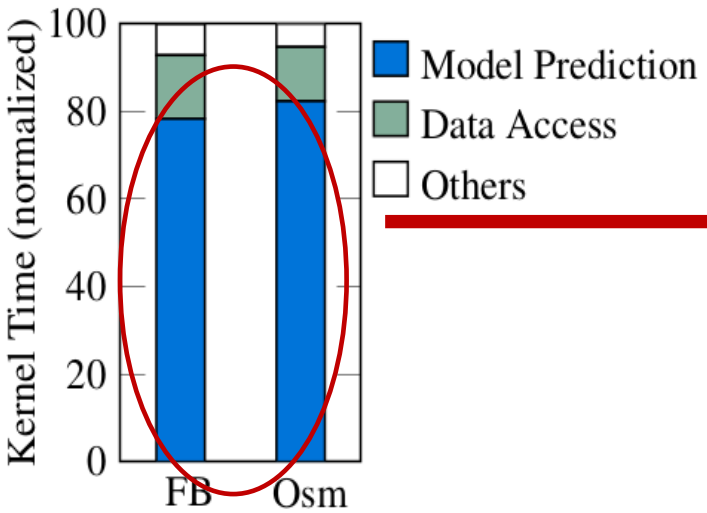
Challenge

② The mismatch between model structure and PIM features

- Model prediction needs multiple **floating-point operations**
- PIM features **high memory bandwidth** but **limited computational capabilities**

Mem	FLOAT ADD	FLOAT SUB	FLOAT MUL	FLOAT DIV
630 MB/s	4.91 MOPS	4.59 MOPS	1.91 MOPS	0.34 MOPS

Throughput of an UPMEM PIM module.

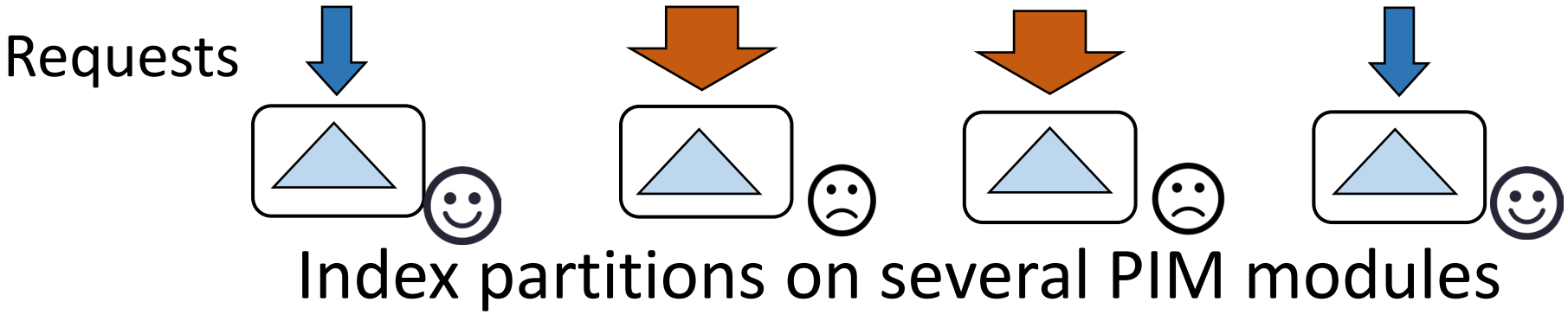


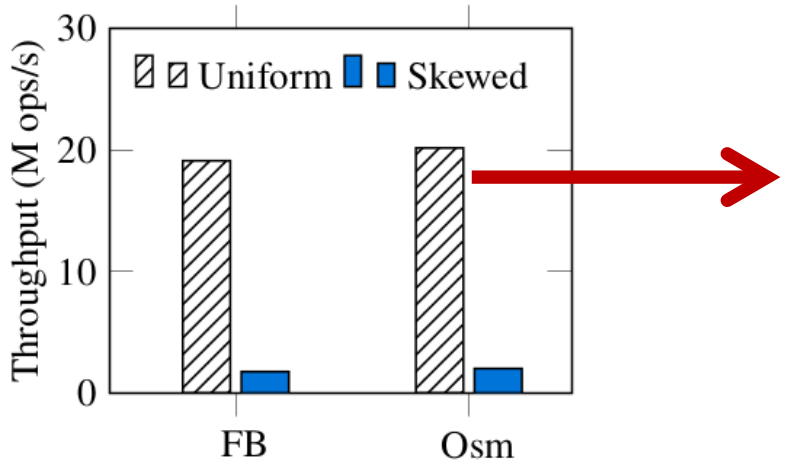
About 80% of the time is dedicated to model prediction

Challenge

③ The load imbalance caused by skewed workloads

- PIM-based indexes divide indexes into several PIM modules
- Skewed workload can cause load imbalance





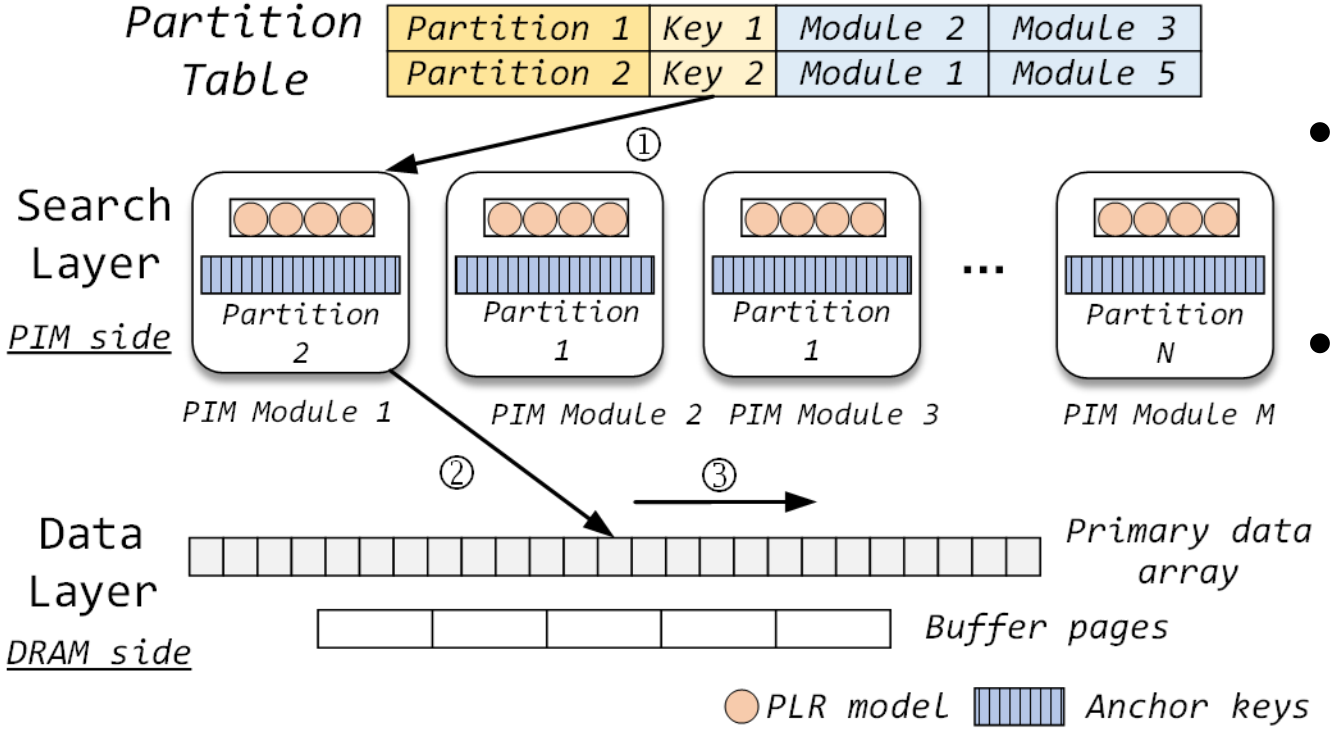
The throughput under skewed workload achieves only 9.2%~9.9% of the uniform workload

Outline

- ☐ Background and Motivation
- ☐ **Design of PIMLex**
- ☐ Evaluation
- ☐ Conclusion

PIMLex Overview

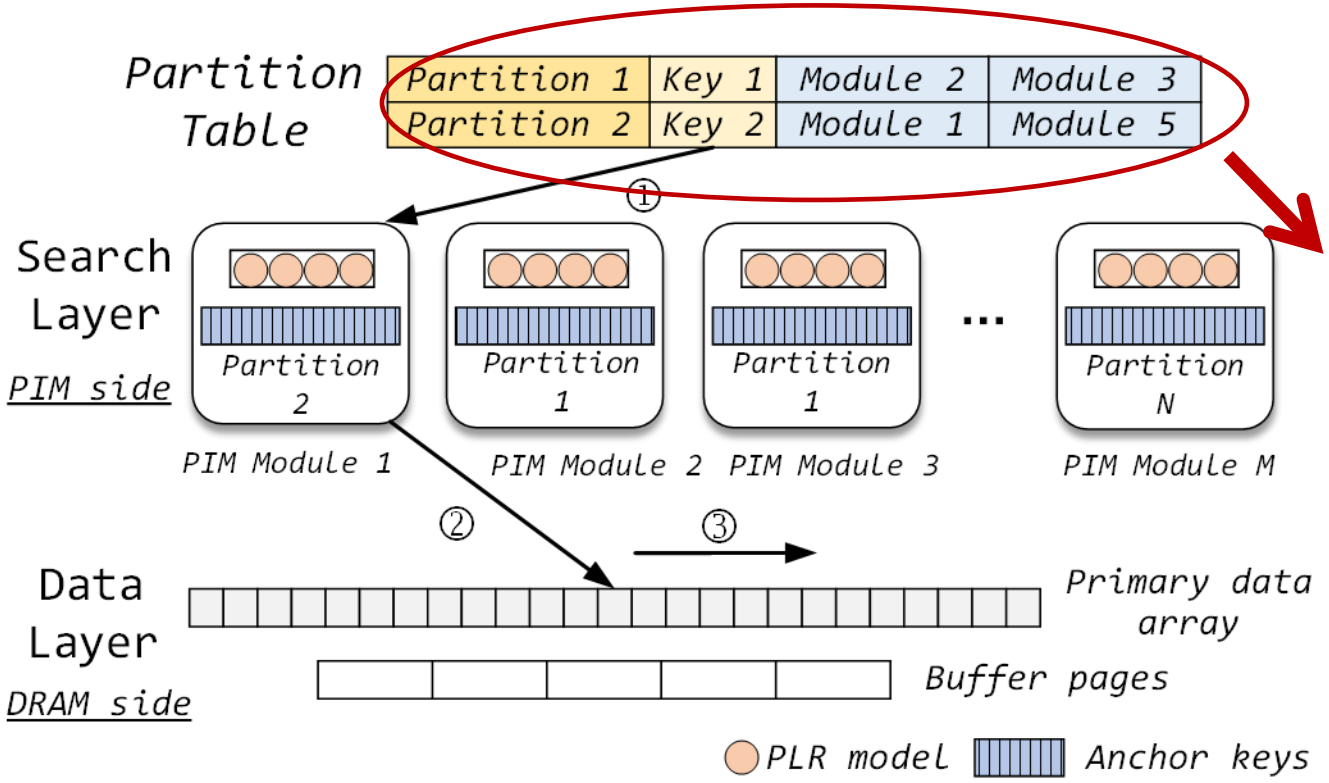
Two-Layer Decoupled Index Structure



- **Search Layer** on PIM side - consists of sampled anchor keys
- **Data Layer** on DRAM side - manages complete set of key-value pairs

PIMLex Overview

PIMLex Execution Procedure

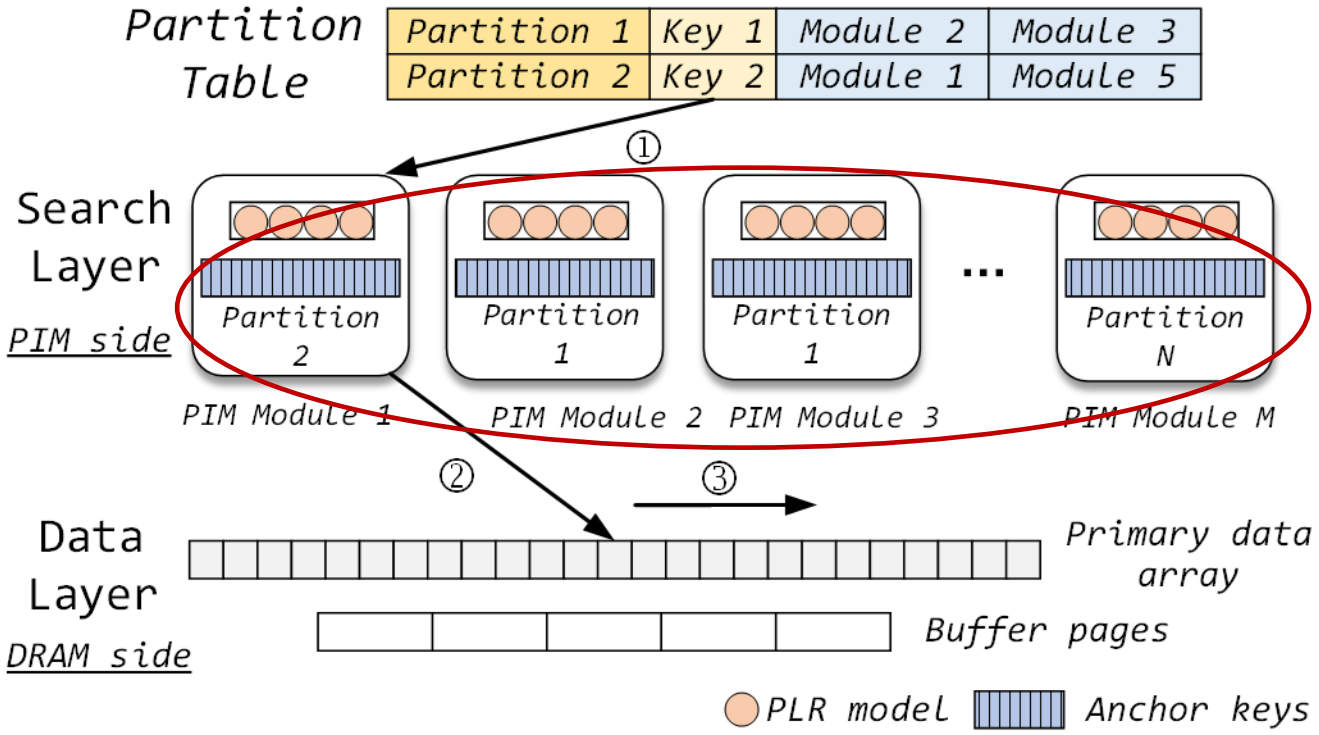


①

Check the partition table to identify the PIM module to which the requested key K_t belongs

PIMLex Overview

PIMLex Execution Procedure

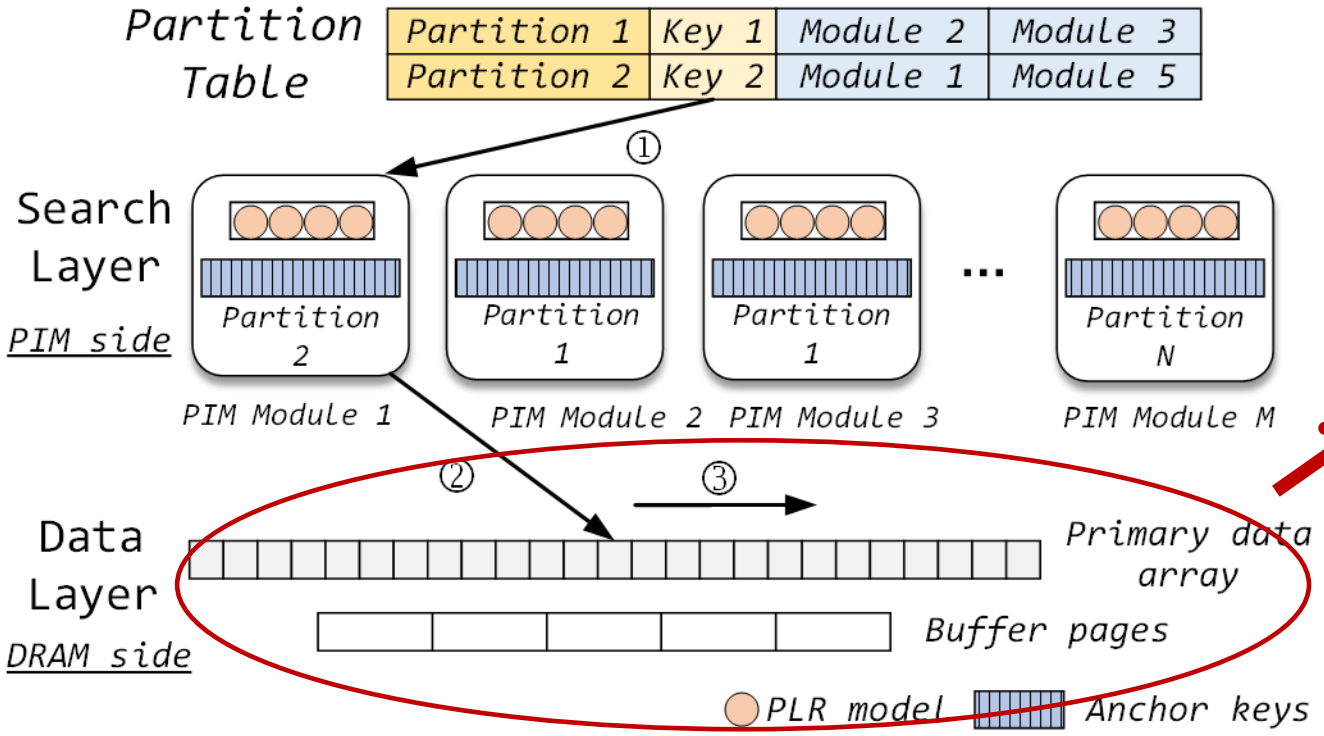


②

Perform lookup on search layer to obtain the approx. position of K_t in data layer

PIMLex Overview

PIMLex Execution Procedure



③

- Locate the actual position of K_t within data layer
- Perform reading and writing on the primary data array and buffer pages

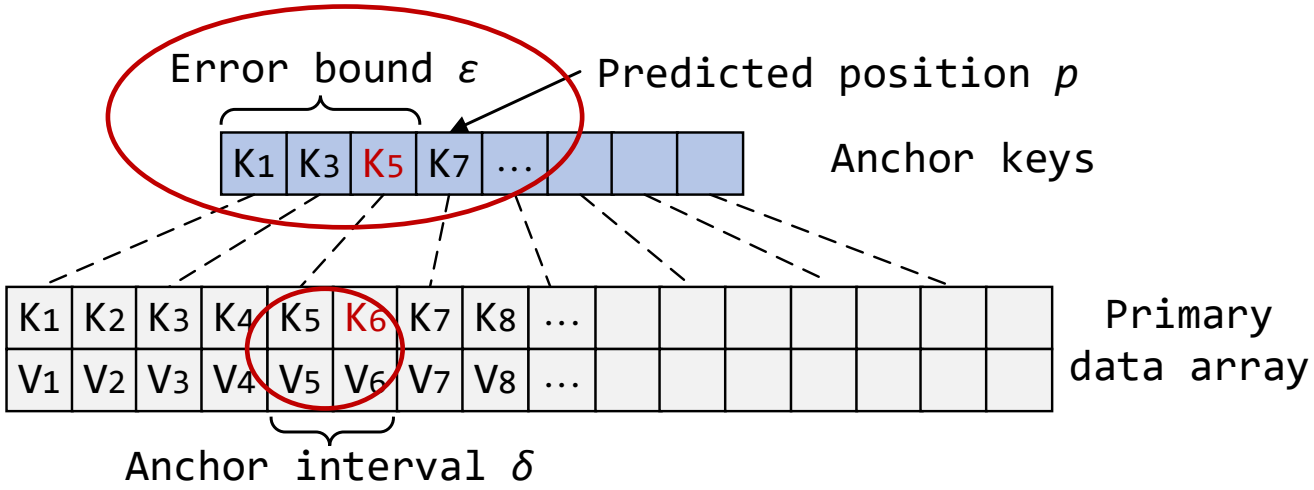
Two-Layer Decoupled Structure

Minimizing PIM space usage

- $|\text{Anchor Keys}| = \frac{|\text{Primary Keys}|}{\delta}$

Handle memory accesses in PIM

- To find K_6, first find K_5 in search layer. Search range is [p – ε, p + ε].
- Check data layer, search keys within anchor interval δ.



Address Challenge 1:

Large space demand vs. The limited capacity of PIM

PIM-friendly Model Structure

PIM: powerful memory capability ☺

limited computing capability ☹

- **Key idea**: Trading more memory access for less computation
  - ✓ Model Search Method Selection
  - ✓ Lookup-table based Model

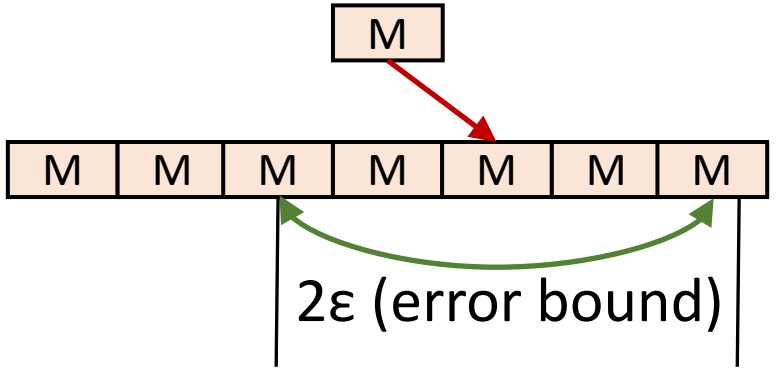
Address Challenge 2:

The mismatch between the model structure and PIM features

Model Search Method Selection

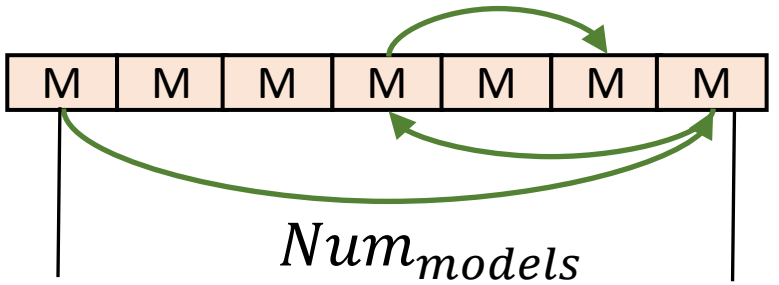
Learned indexes usually apply multi-level model structures

Model-based search (original)



- With model computation
- Less memory accesses
- $Cost_{prediction} + \log(2\varepsilon) * Cost_{search}$

Global binary search



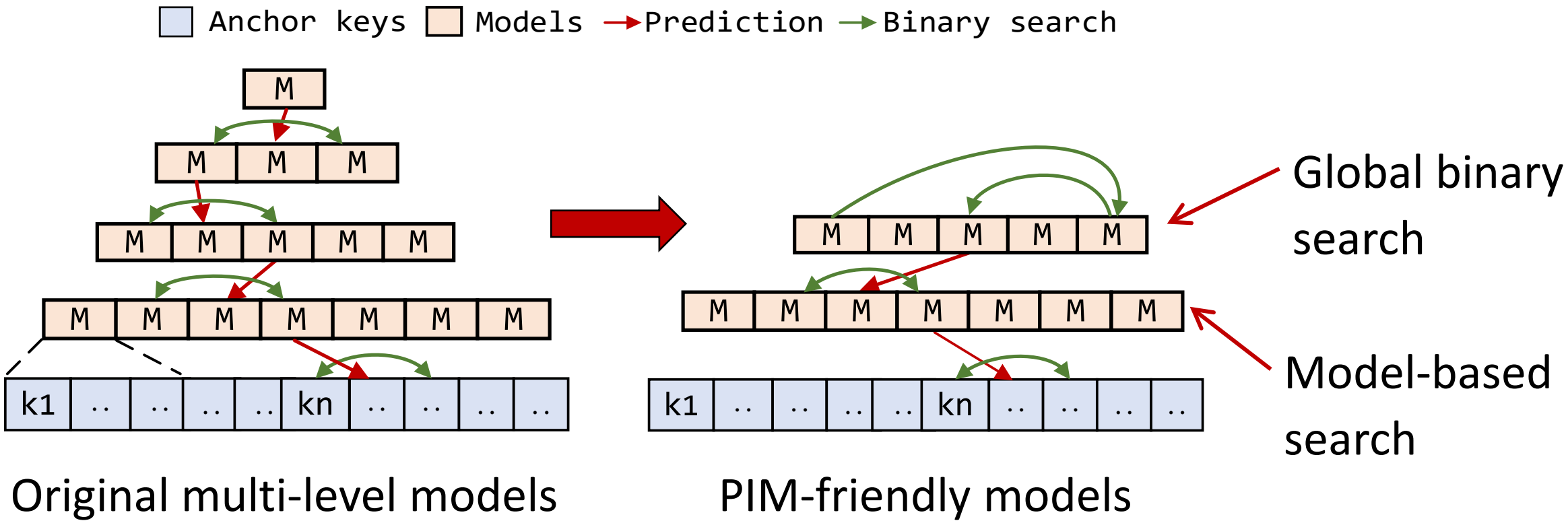
Prediction

Binary search

- Without model computation
- More memory accesses
- $\log(Num_{models}) * Cost_{search}$

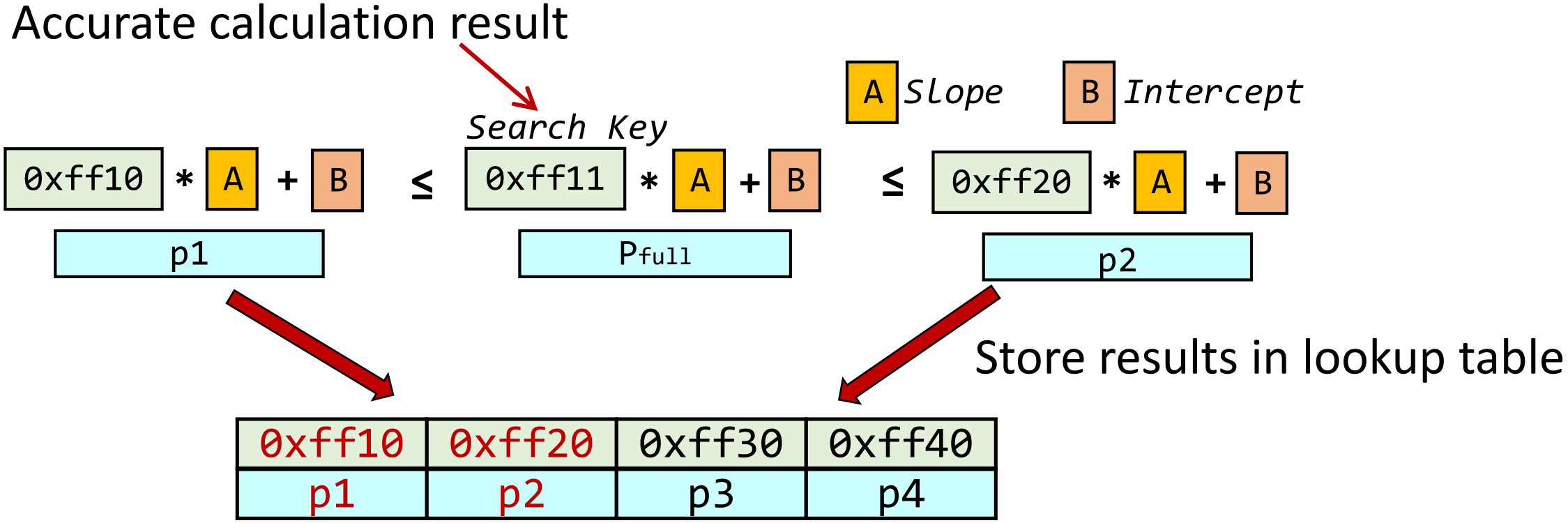
Model Search Method Selection

Select the model search method with the minimal cost



Lookup-table based Model

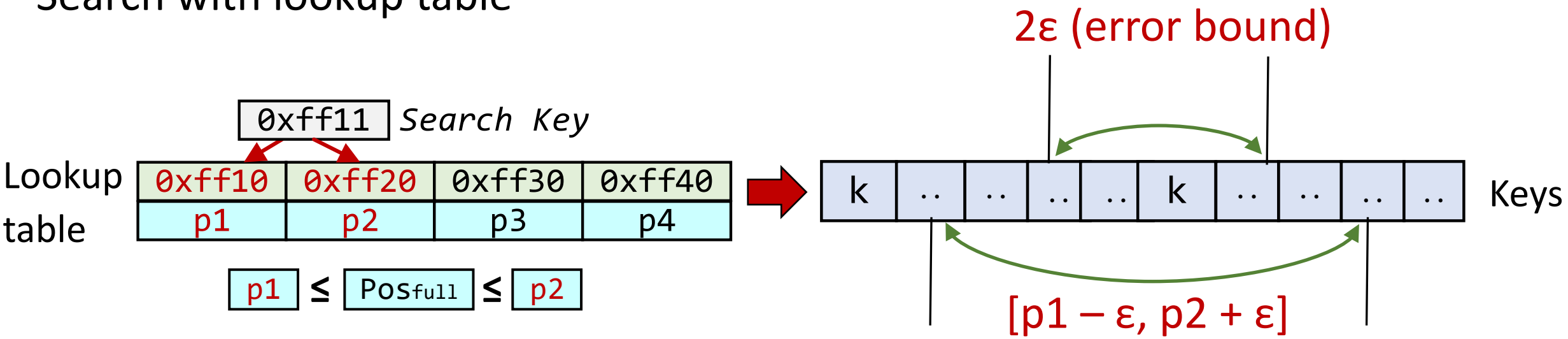
- Eliminate floating-point operations during model calculations
- Pre-calculate and store the results in a lookup table



- Memory space and accuracy tradeoff
- Constrain the number of slots in the lookup table to S

Lookup-table based Model

Search with lookup table



- Lookup-table based Model:

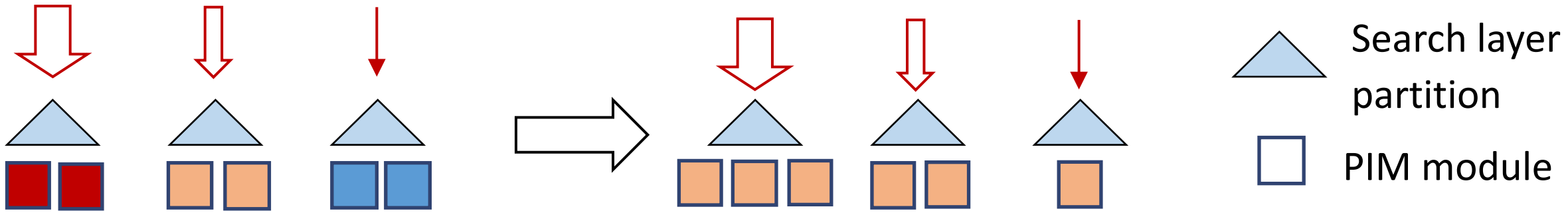
$$Cost_{search} + \log(Pos_{upper} - Pos_{lower} + 2\varepsilon) * Cost_{search}$$

- Floating-point based Model:

$$Cost_{prediction} + \log(2\varepsilon) * Cost_{search}$$

- Select the model with lower cost

Hotness Aware Replication Mechanism



- Some PIM modules are overloaded due to skewed workloads
- **Key idea:** Create additional replicas on more PIM modules for hot partitions

Address Challenge 3:

The load imbalance issues caused by skewed workloads

Hotness Aware Replication Mechanism

Determine the number of replicas R_i for partition i

- $R_1 + R_2 + \ldots + R_n = M$, M is the total number of PIM modules
- T_i is the number of tasks (queries) handled by partition i

Load factor $L_i = \frac{T_i/R_i}{T_{total}/M}$ $L_{global} = max(L_i)\ \ i = 1,2,\ldots,n$

- **Load Balancing Goal:** Configure the values of R_i that minimize L_{global}
- **Load Balancing algorithm:**
  - Collect T_i that processed by partition i in the past period of time
  - Obtain R_i by comparing T_i with T_{avg} (i.e., T_{total}/M)
  - More details in our paper.

Hotness Aware Replication Mechanism

How to maintain load balance when hotspot distribution changes?

- **Normal Replication Adjustment**
  - ✓ Reallocate all replicas for all partition
  - ✓ Change M replicas, M is the total number of PIM modules
  - ✓ Slow but effective

- **Quick Replication Adjustment**
  - ✓ Reassign PIM modules from idle partition to busy partition
  - ✓ Change replicas for at most *N* (N < M) PIM modules
  - ✓ Fast but ineffective

Outline

- ☐ Background and Motivation
- ☐ Design of PIMLex
- ☐ **Evaluation**
- ☐ Conclusion

Experimental Setup

Compared Counterparts

- BasicLex (PIM-based learned index baseline, without any optimizations)
- Four DRAM-based learned indexes
  - ✓ ALEX (SIGMOD'20)
  - ✓ LIPP (VLDB'21)
  - ✓ FINEdex (VLDB'22)
  - ✓ SALI (SIGMOD'24)
- ALEX and LIPP use optimistic lock to achieve concurrency (VLDB'22)

Five real-world datasets. Zipfian 0.99 distribution (skewed workloads)

Books	the book sales popularity from Amazon	800M int64
Osm	the randomly sampled cell IDs on OpenStreetMap	800M int64
FB	the randomly sampled Facebook user IDs	200M int64
Genome	the local pairs in human chromosomes	200M int64
Planet	the Planet IDs in OpenStreetMap	200M int64

Experimental Setup

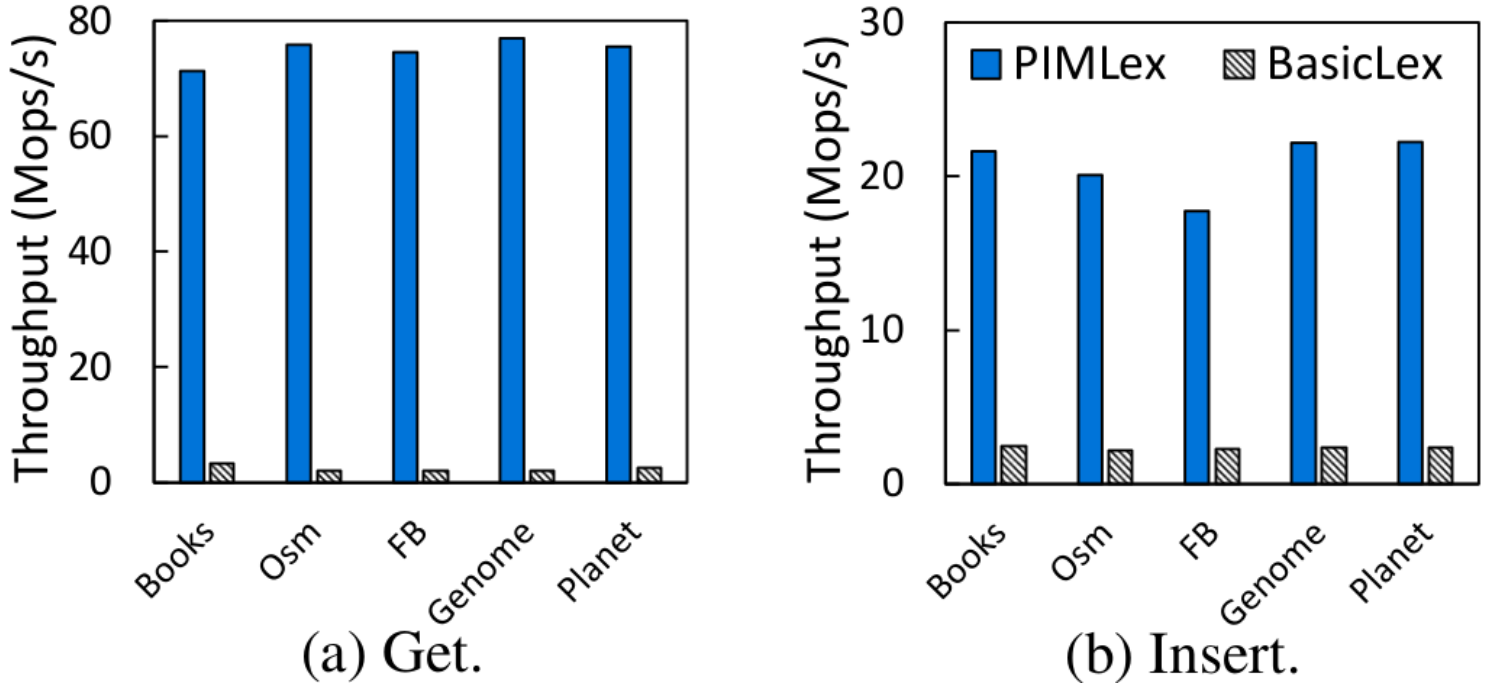
PIM-based Indexes Hardware Platform

CPU	2 * Intel Xeon Silver 4110 CPUs, 16 logical cores per node
DRAM	4 * 32GB DDR4 DIMMs (64GB per node)
PIM	4 * 8GB UPMEM DIMMs (512 PIM modules)

DRAM-based Indexes Hardware Platform

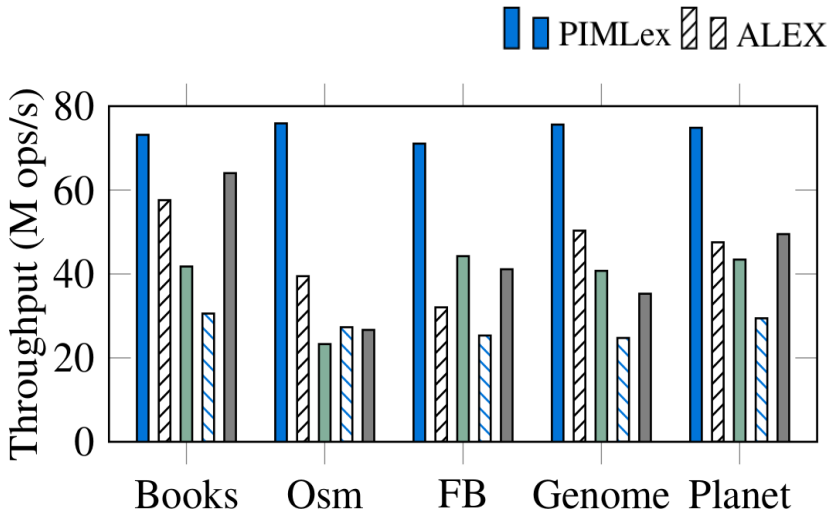
CPU	2 * Intel Xeon Silver 4210 CPUs, 20 logical cores per node
DRAM	8 * 32GB DDR4 DIMMs (128GB per node)

Comparison with PIM-based Baseline

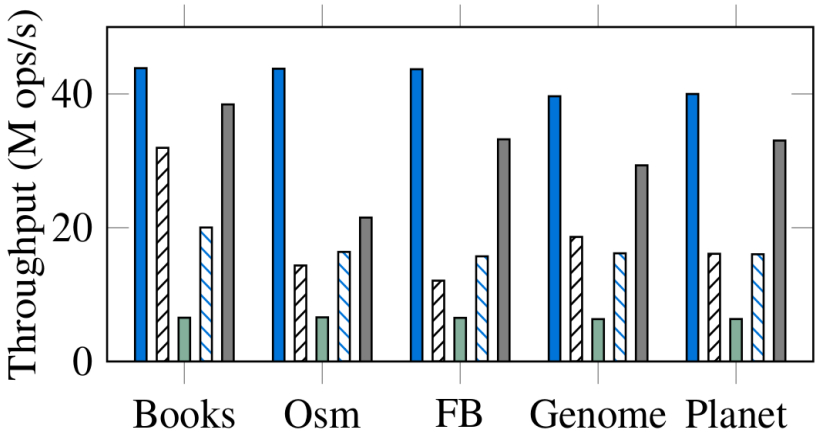


- ***Get*** throughput outperforms BasicLex by up to **36.5×**
- ***Insert*** throughput outperforms BasicLex by up to **9.5×**

Comparison with DRAM-based Indexes



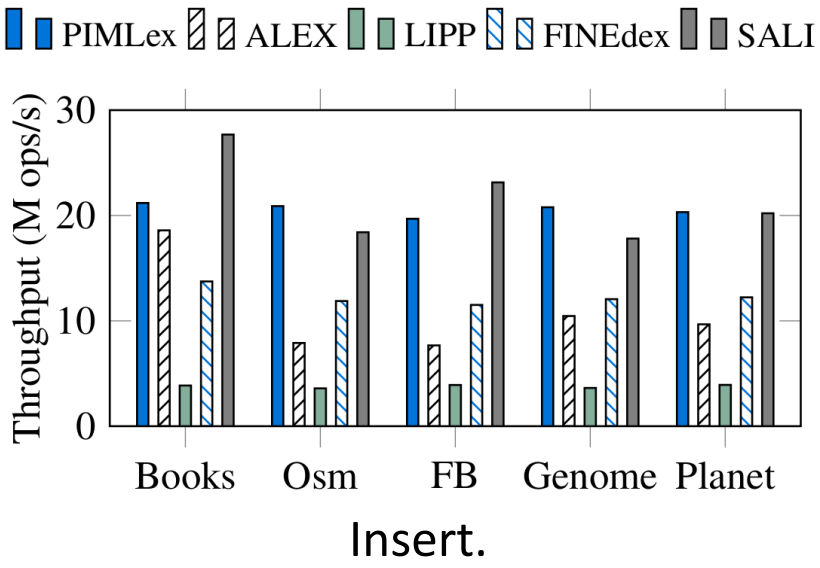
Get.



50% Get + 50% Insert (Mixed).

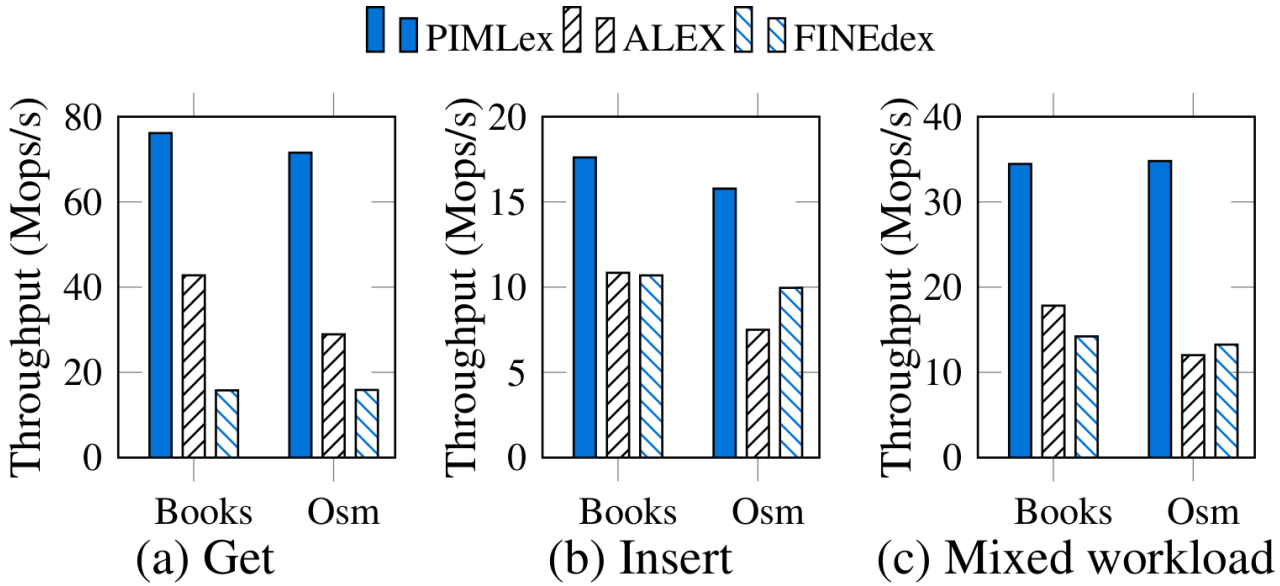
- ***Get-only:*** Outperform others by up to **2.2×, 3.25×, 3.05× and 2.84×**
- ***Mixed:*** Outperform others by up to **3.6×, 6.7×, 2.73× and 2.03×**

Comparison with DRAM-based Indexes



- ***Insert:***
  - Outperform ALEX, LIPP and FINEdex across all datasets
  - Inferior to SALI in certain datasets (e.g., Books)

Large Datasets



Performance with large-scale datasets.

Each dataset contains 800M keys

- Outperform ALEX by up to **2.47×, 2.09× and 2.93×**
- Outperform FINEdex by up to **4.83×, 1.65× and 2.62×**
- The index size of LIPP and SALI exceeds the total memory capacity

Outline

- ☐ Background and Motivation
- ☐ Design of PIMLex
- ☐ Evaluation
- ☐ **Conclusion**

Conclusion

- PIMLex preliminarily explores the potential of PIM to address the memory-bound issues of learned indexes
  - ✓ Decoupled Index Structure
  - ✓ PIM-friendly Model Structure
  - ✓ Hotness Aware Replication Mechanism
- On real-world PIM hardware, PIMLex outperforms PIM-based baseline and DRAM-based learned indexes
- More details (e.g. search/insert processing) are in our paper.

Thank you!

Q & A