

# FILM: a Fully Learned Index for Larger-than-Memory Databases

Ma, Chaohong, et al. VLDB 2022

2024. 02. 28

Presentation by Yejin Oh, ZhuYongjie, Boseung Kim

yeojinoh@dankook.ac.kr, arashio1111@dankook.ac.kr, bskim1102@dankook.ac.kr

# Contents

## **1. Introduction**

## **2. Design**

## **3. FILM**

### **1) Design of piece**

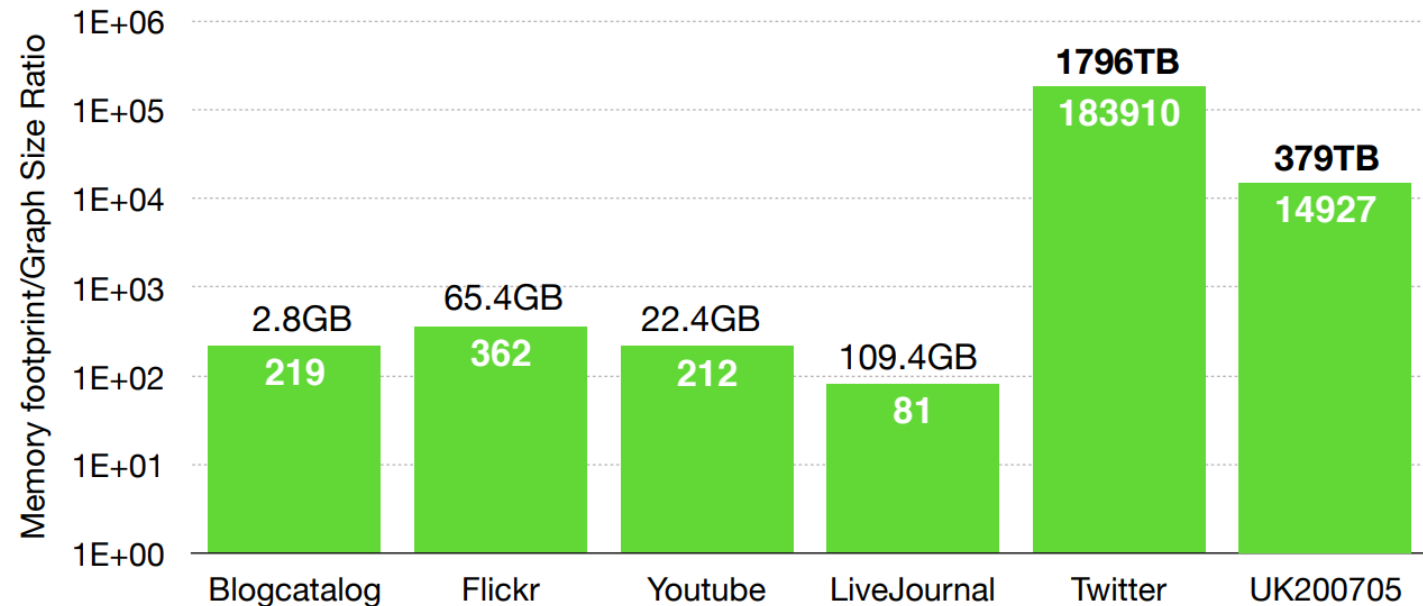
### **2) Dynamic learned index**

### **3) Adaptive LRU**

## **4. Experiments**

# Introduction

- Data characteristics
  - Large volume and high velocity
  - Append-only
  - Diverse workloads



Reference: Shao, Yingxia, et al. "Memory-aware framework for fast and scalable second-order random walk over billion-edge natural graphs." *The VLDB Journal* 30.5 (2021)

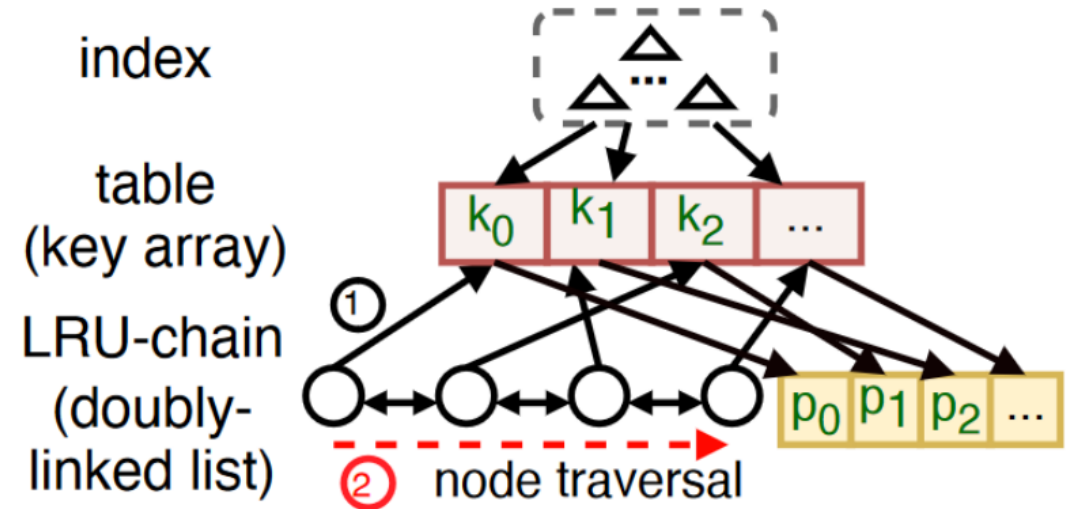
# Introduction

- Challenge

- 1) Existing index has **high disk I/O**
- 2) Existing index occupies **a large portion of memory**
- 3) In 'anti-caching', computation **overhead in LRU is too high**
- 4) Learned index assumes data is stored in **a contiguous array**
- 5) Learned index is **for in-memory** data

# Introduction

- Anti-caching
  - To reduce the consumption of main memory
  - High memory consumption
  - High LRU maintenance costs



**(a) anti-caching's LRU**

# Introduction

- Goal
  - Using Learned Indexes in Heterogeneous storage
  - Cost Reduction of Cold Data Identification with Adaptive LRU

# Design

## 1. Lightweight Machine learning model

- Capturing data distribution to reduce memory usage

## 2. Adaptive LRU

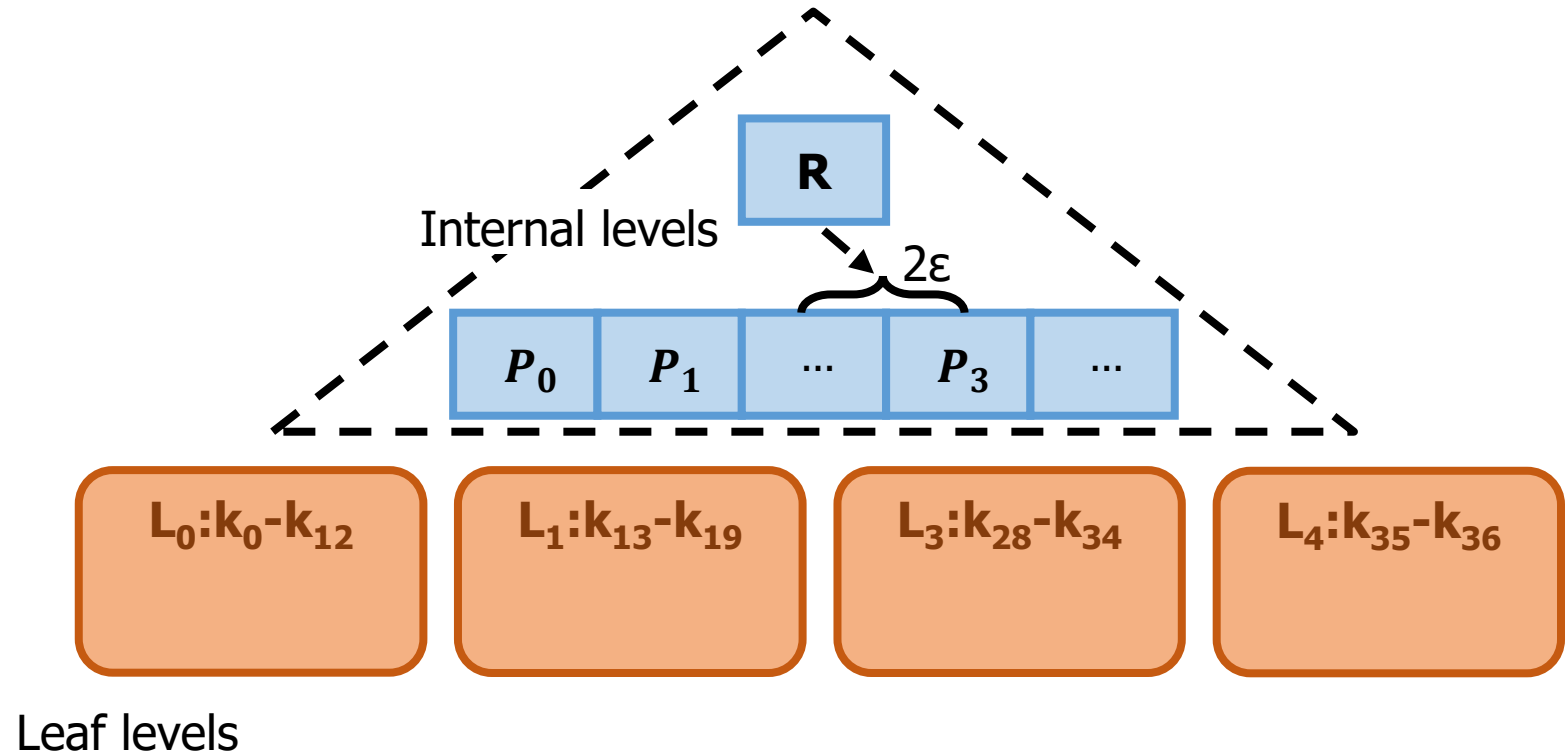
- Adaptive LRU for efficient cold data identification, cost-effective management

## 3. Unified Tree Structure

- Stable tree structure minimizes maintenance despite data swapping

# Design

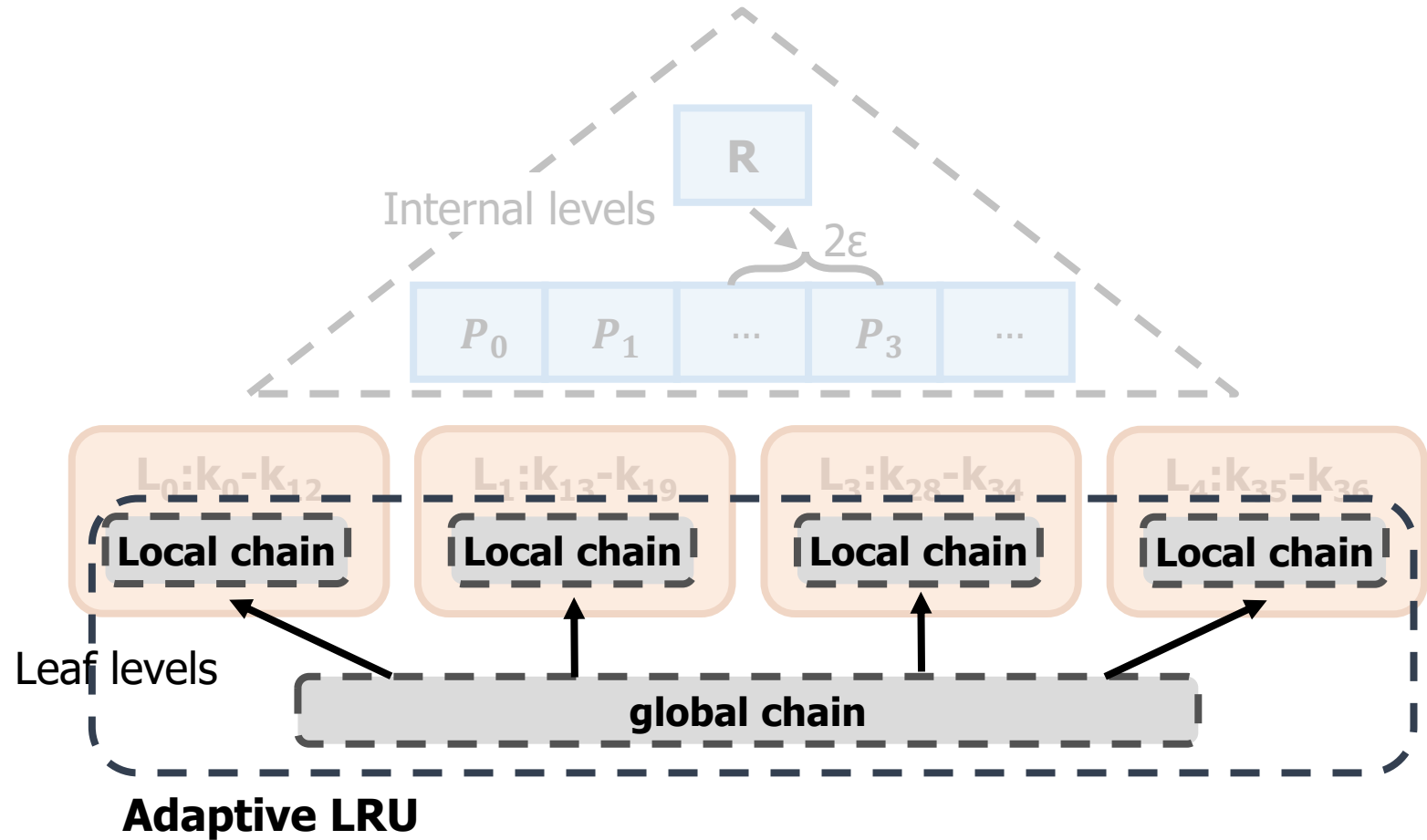
- **Learned index**
- adaptive LRU
- Unified tree structure





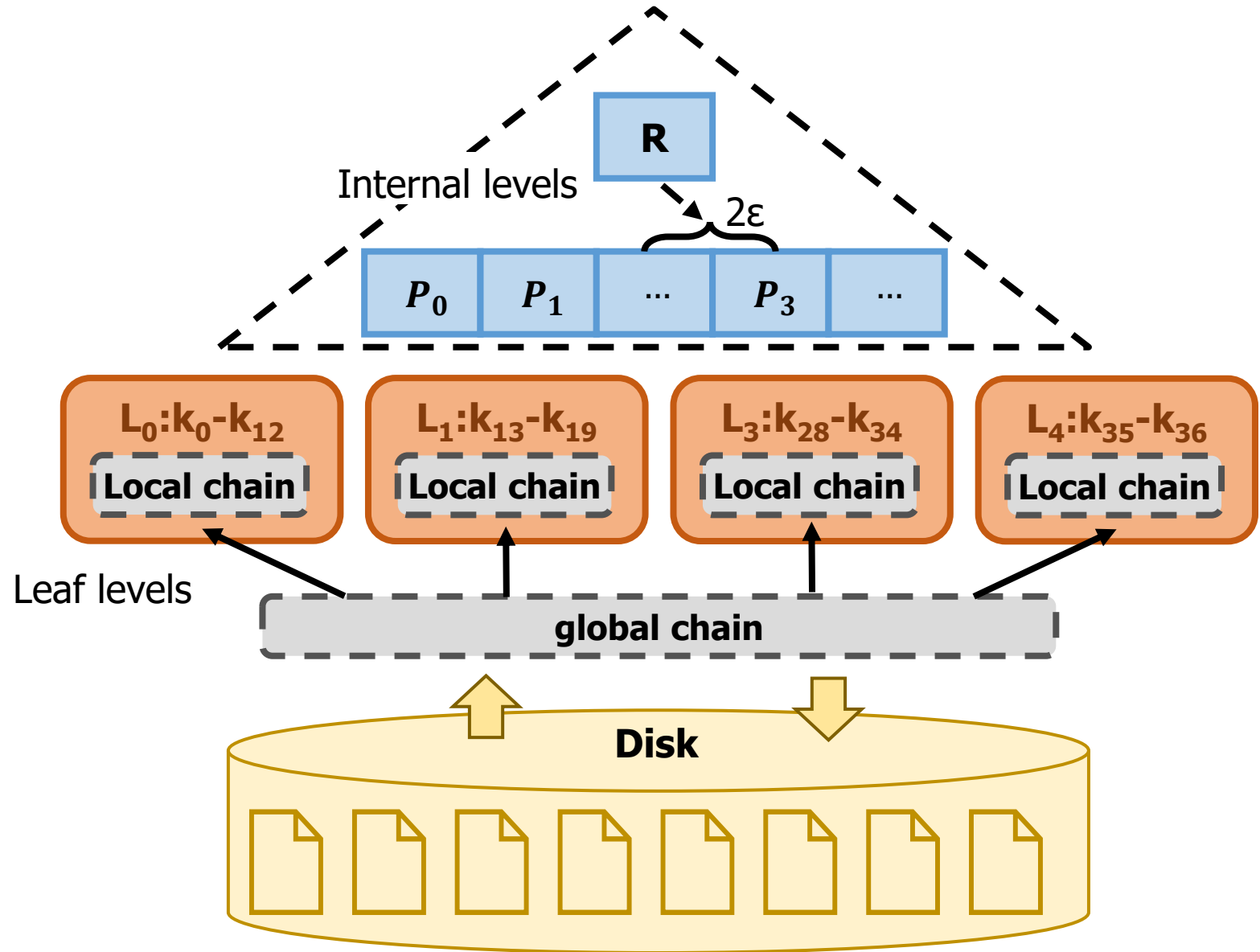
# Design

- Learned index
- **adaptive LRU**
- Unified tree structure



# Design

- Learned index
- adaptive LRU
- **Unified tree structure**



# The Design of Piece

- Piece

- Contains a sub-range of data and an approximation model fitted on the sub-range
- Use a list of pwlfes (piecewise linear functions) to partition the sub-range of keys
- The model guarantees the following formula ( $\varepsilon$ : a specified distance bound)

$$|pred\_pos - true\_pos| \leq \varepsilon$$

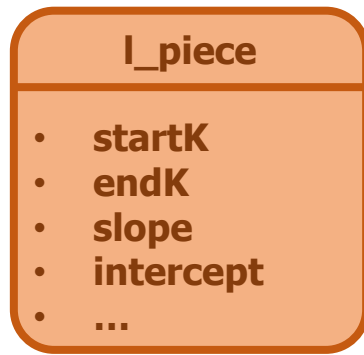
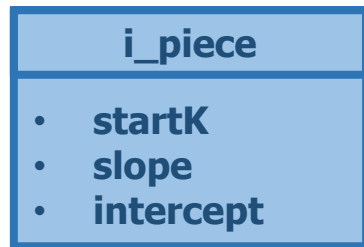
- Newly inserted key breaking constraint is inserted into new piece, this key is called 'break\_k'

# The Design of Piece

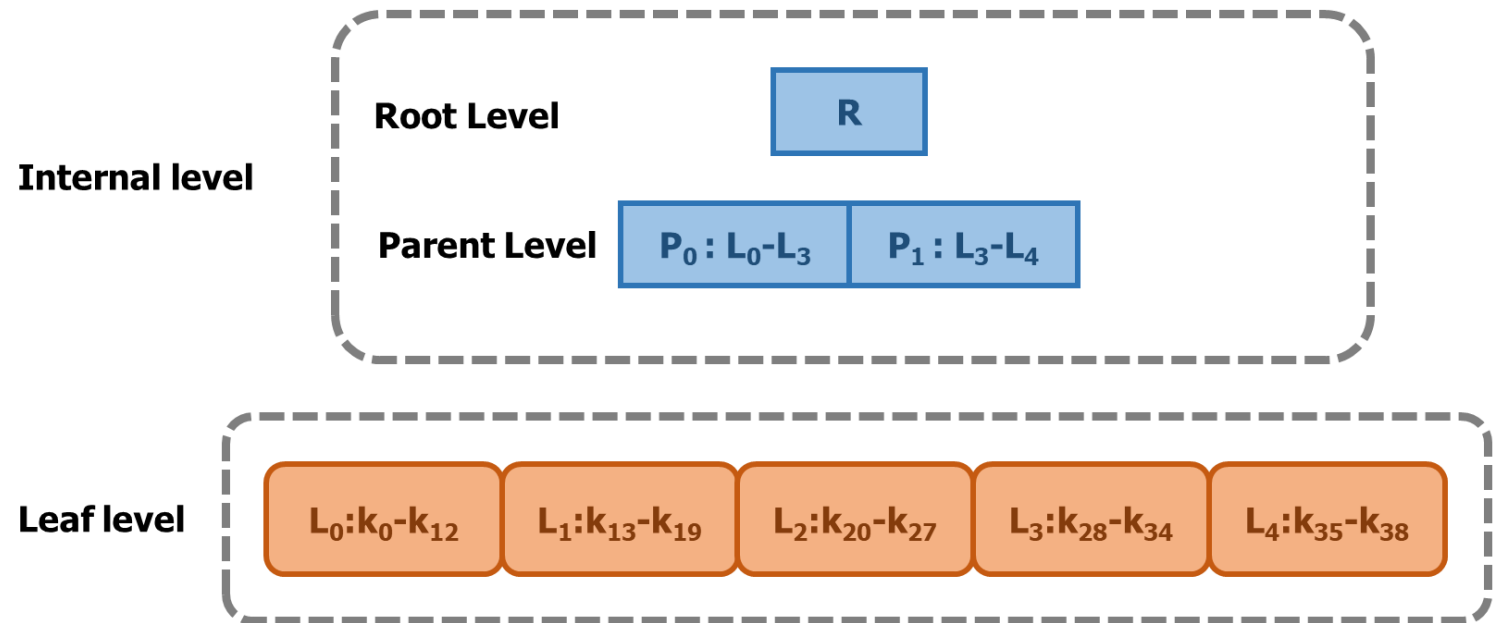
i\_piece 

l\_piece 

- Two types of piece
  - Component of pieces



- Learned model with pieces

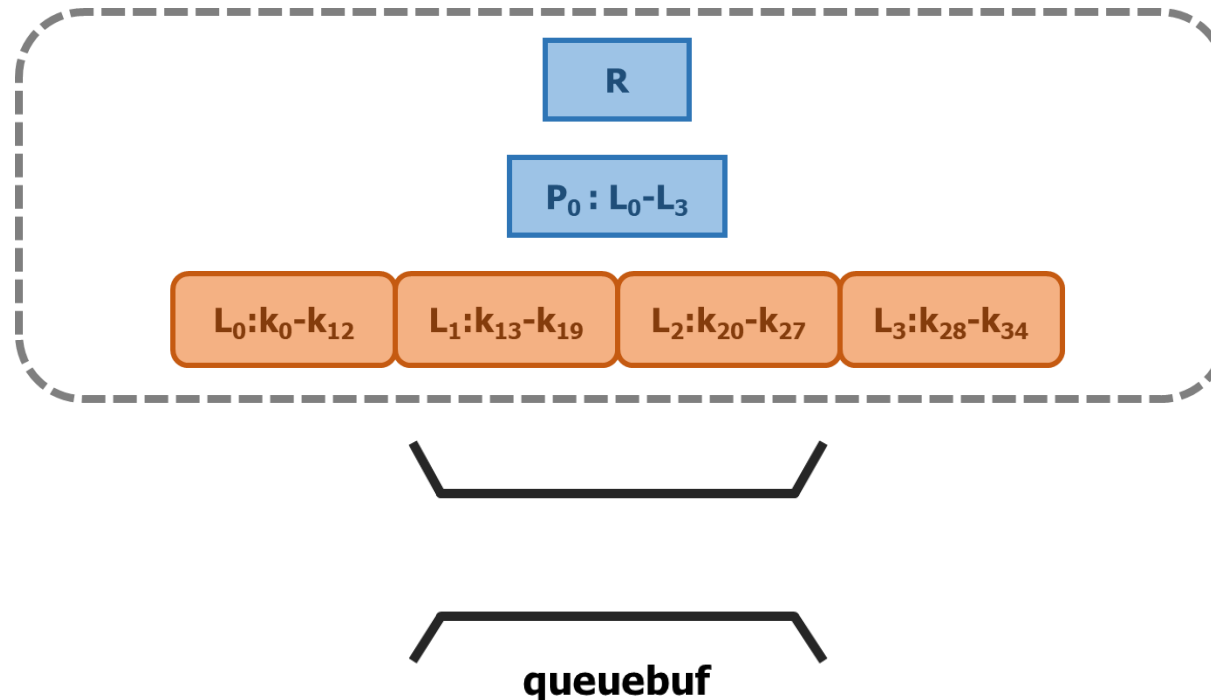


# Dynamic Learned Model

- The construction process of the learned model
  - Two break\_k are needed to create l\_piece
  - Queuebuf is designed to temporarily store the break\_k

i\_piece 

l\_piece 

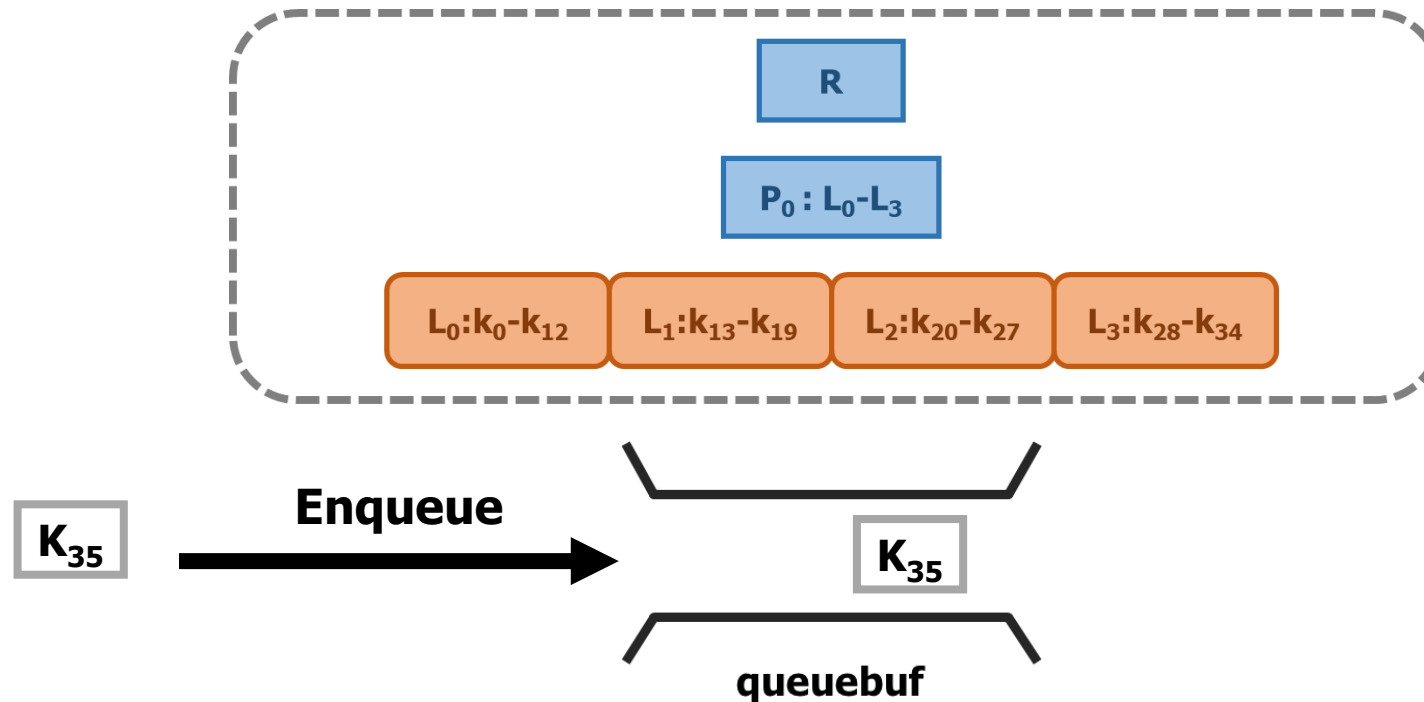


# Dynamic Learned Model

- The construction process of the learned model
  - Put newly arrived key (break\_k) into queuebuf
  - If Key is not break\_k, directly put into l\_piece

i\_piece 

l\_piece 

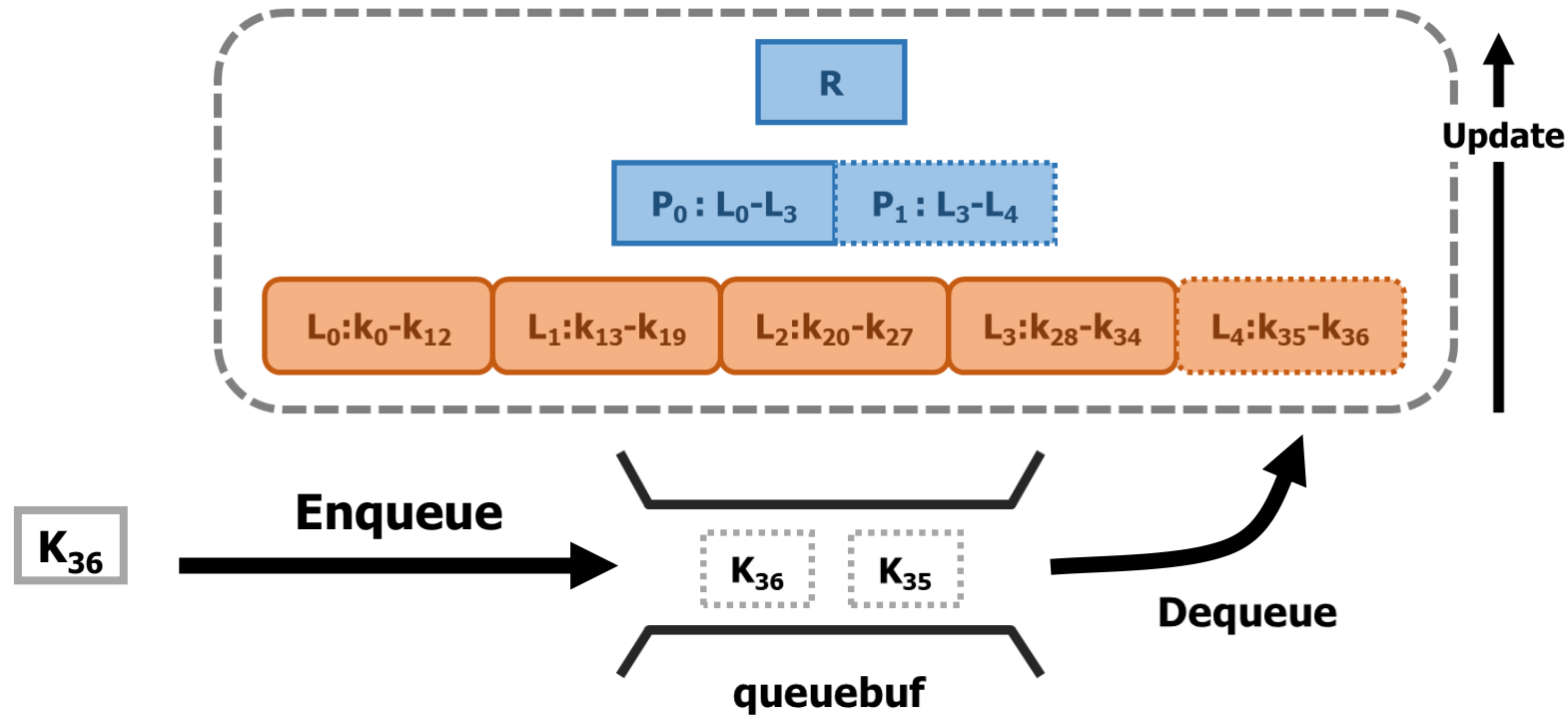


# Dynamic Learned Model

- The construction process of the learned model
  - Queuebuf is full, keys are removed and create a  $l\_piece$
  - Check if the parent level should be updated

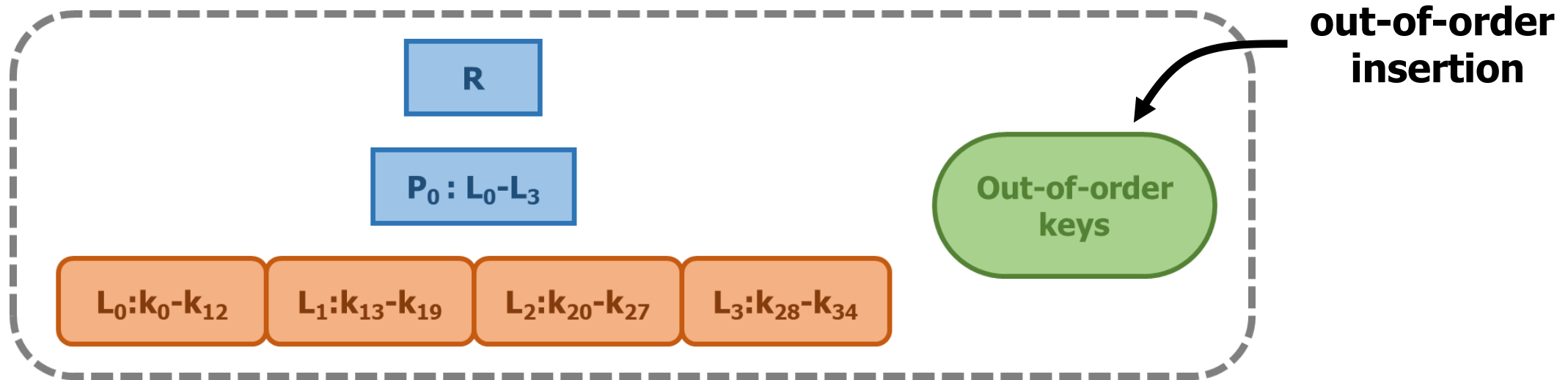
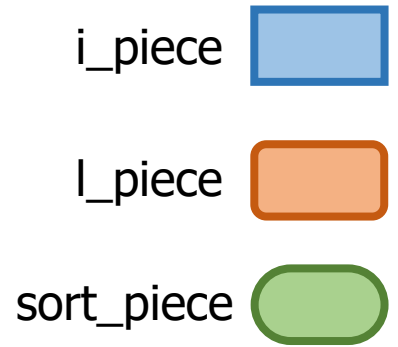
$i\_piece$  

$l\_piece$  



# Dynamic Learned Model

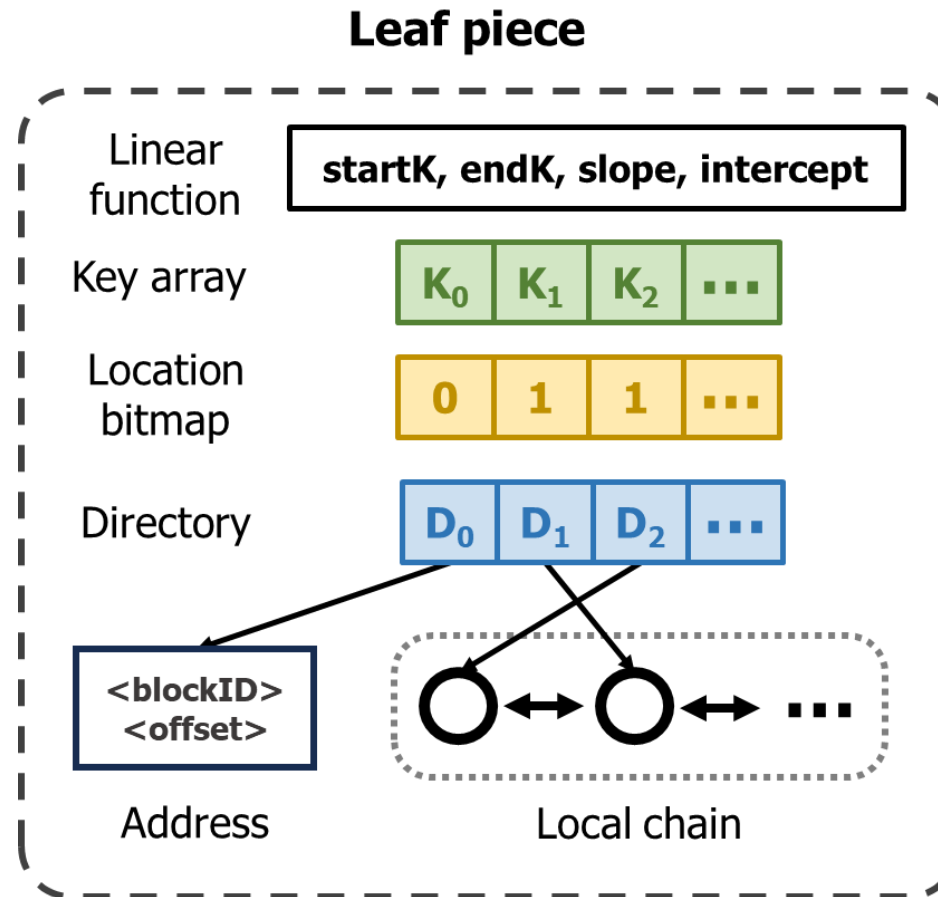
- Handling out-of-order insertions
  - Create sort\_piece, which is special type of I\_piece
  - Directly locates keys using binary search
  - Create a new I\_piece when the size of sort\_piece reaches the threshold





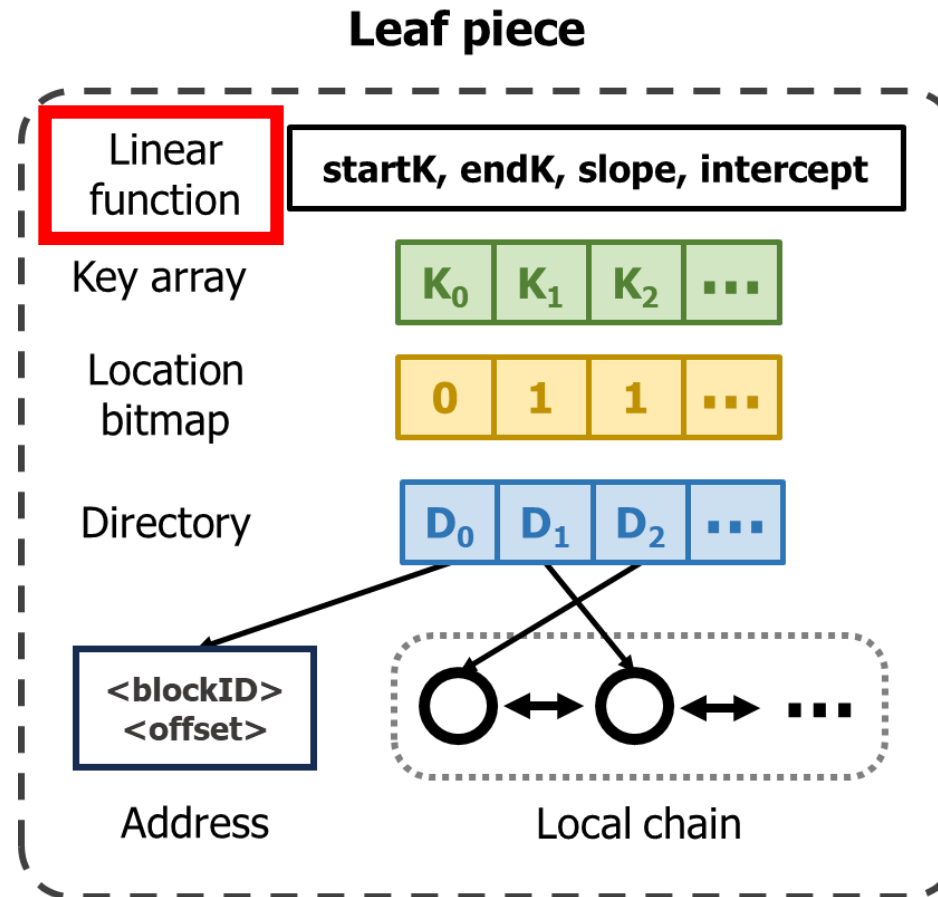
# Dynamic Learned Model

- Leaf piece of FILM
  - Linear function
  - Key array
  - Location bitmap
  - Directory
  - Local chain
  - Address



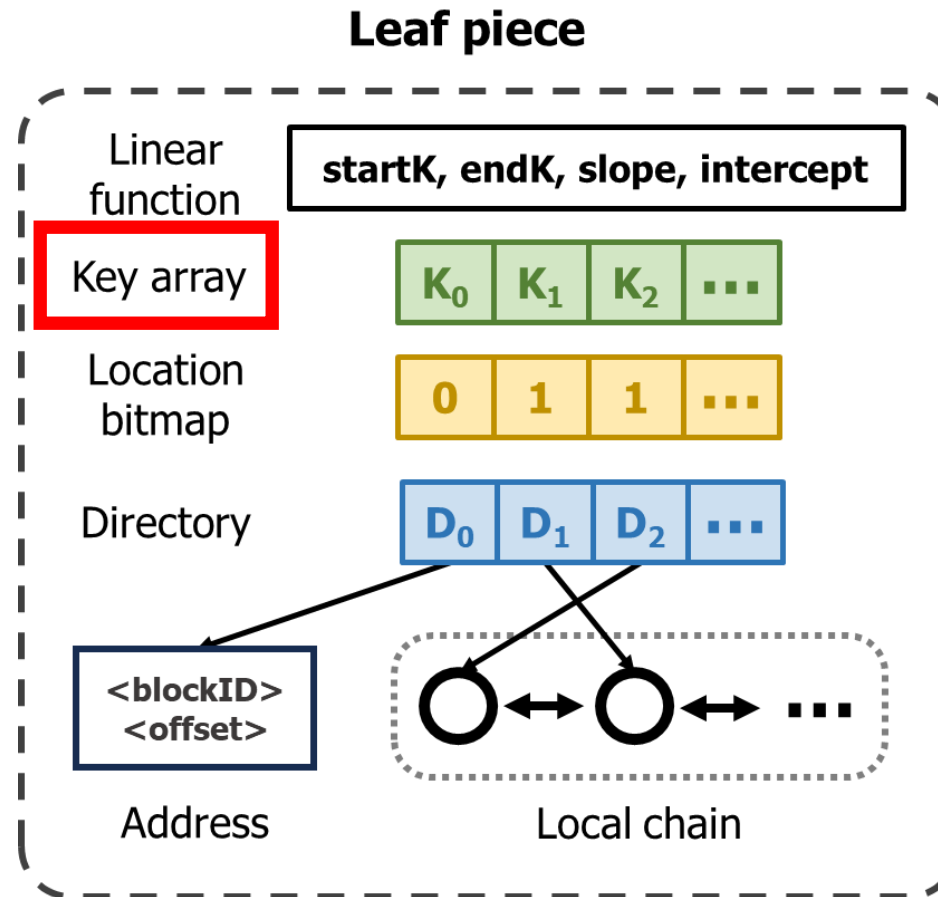
# Dynamic Learned Model

- Leaf piece of FILM
  - Linear function
    - Predicting position of key
  - Key array
  - Location bitmap
  - Directory
  - Local chain
  - Address



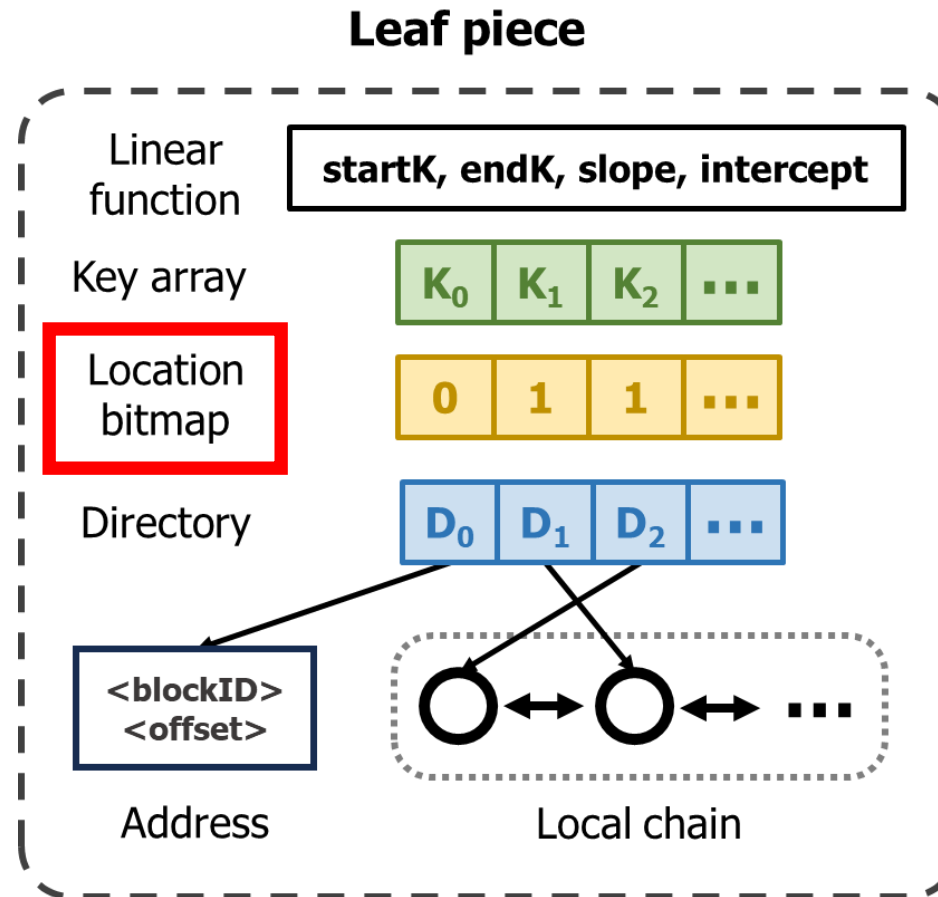
# Dynamic Learned Model

- Leaf piece of FILM
  - Linear function
  - Key array
    - Recording the keys belonging to leaf piece
  - Location bitmap
  - Directory
  - Local chain
  - Address



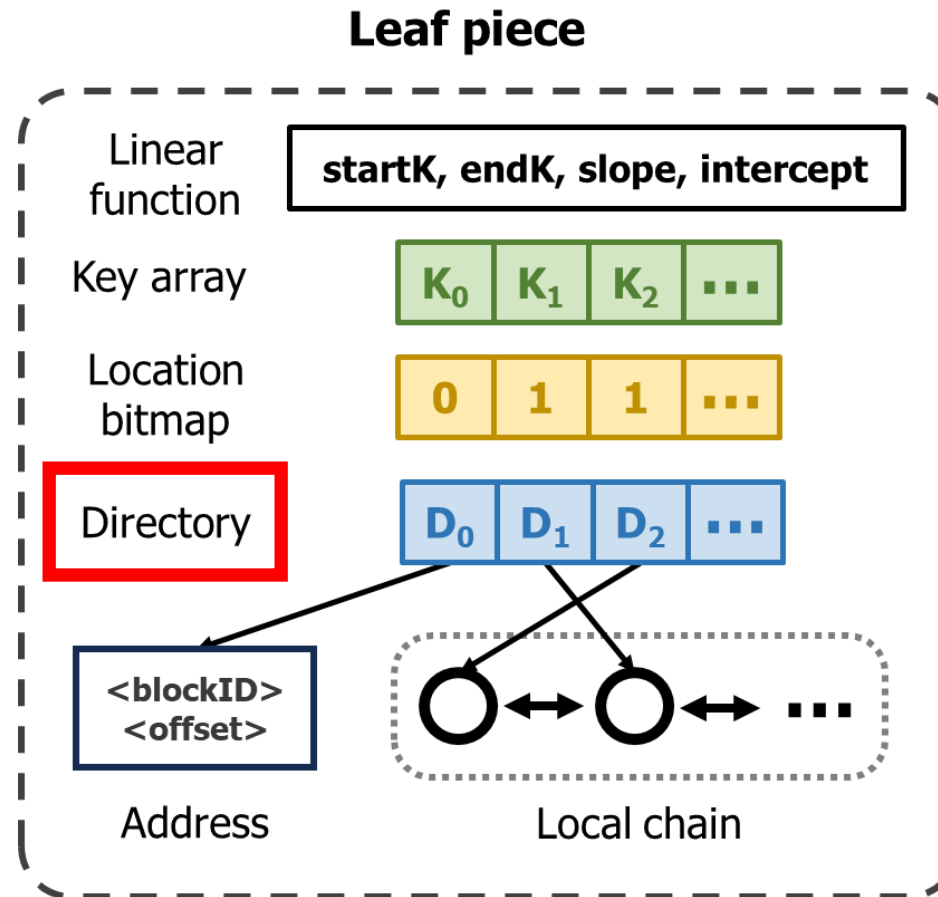
# Dynamic Learned Model

- Leaf piece of FILM
  - Linear function
  - Key array
  - Location bitmap
    - Tracking whether corresponding data record is in memory or not
  - Directory
  - Local chain
  - Address



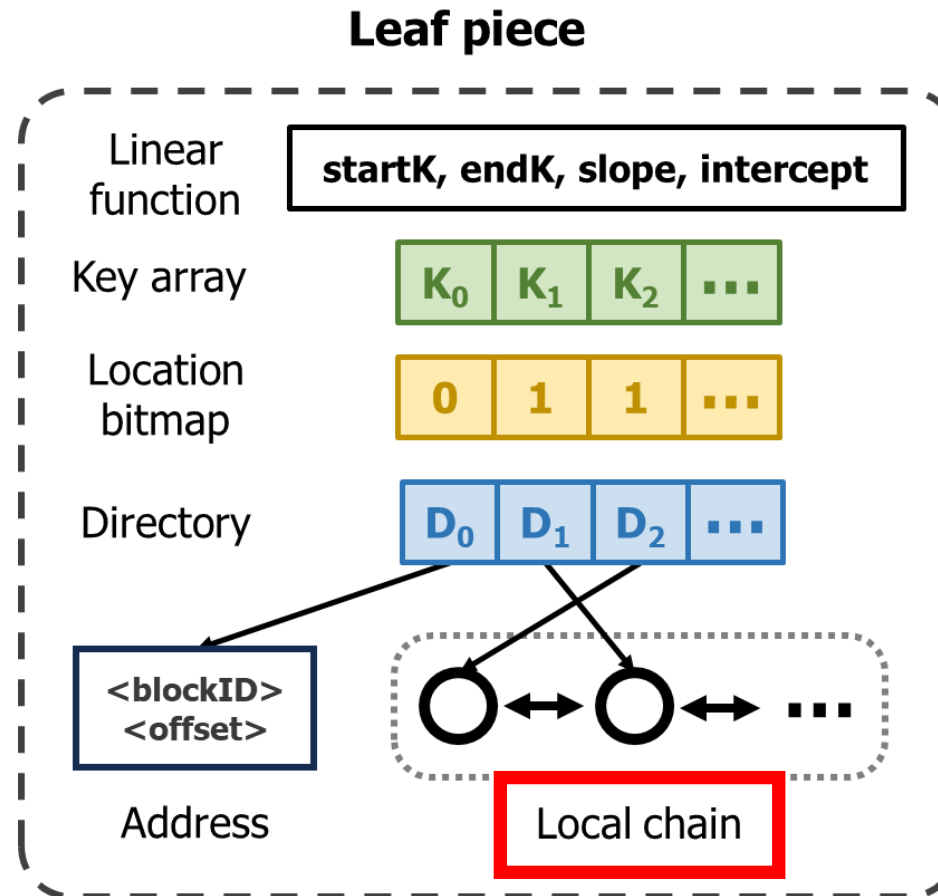
# Dynamic Learned Model

- Leaf piece of FILM
  - Linear function
  - Key array
  - Location bitmap
  - Directory
    - Tracking the real position of all keys
  - Local chain
  - Address



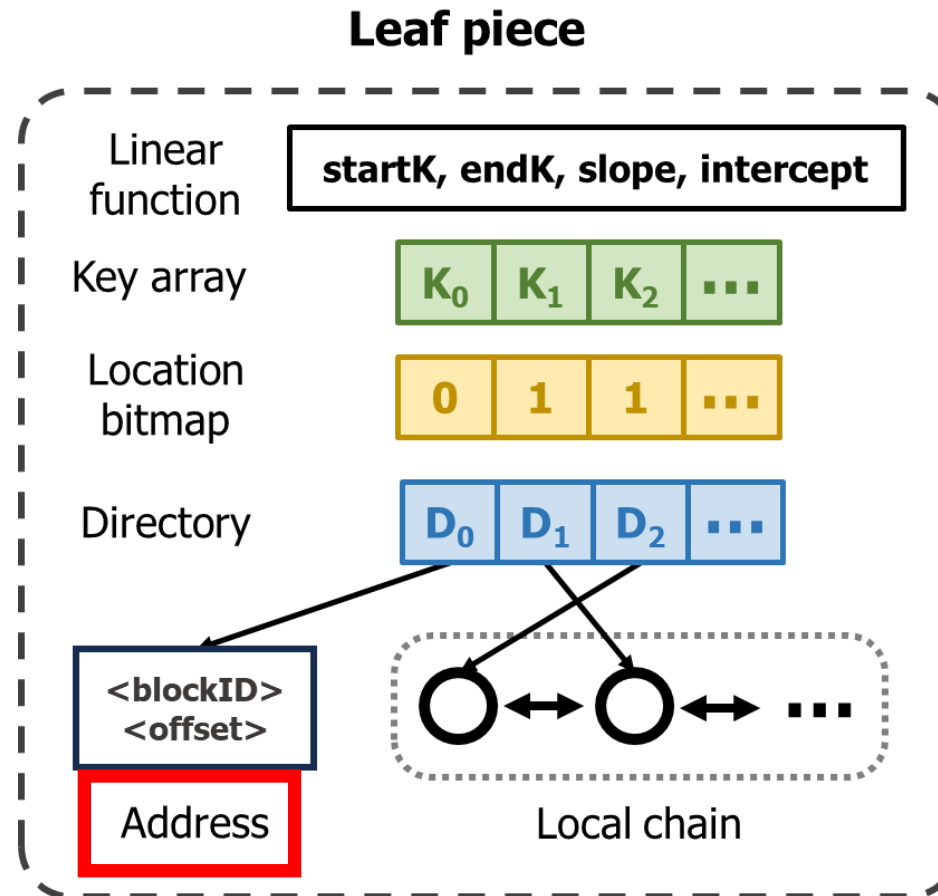
# Dynamic Learned Model

- Leaf piece of FILM
  - Linear function
  - Key array
  - Location bitmap
  - Directory
  - Local chain
    - Maintain LRU order of keys
    - More details in next section
  - Address



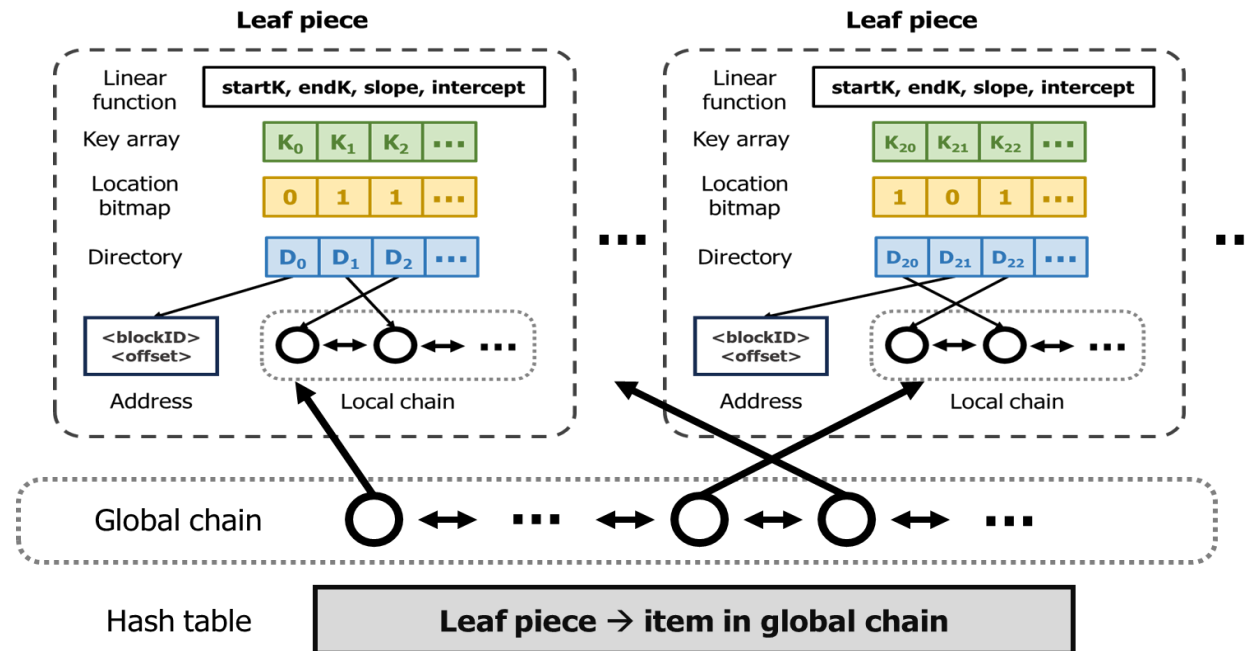
# Dynamic Learned Model

- Leaf piece of FILM
  - Linear function
  - Key array
  - Location bitmap
  - Directory
  - Local chain
  - Address
    - Stores the addresses of all evicted records



# Dynamic Learned Model

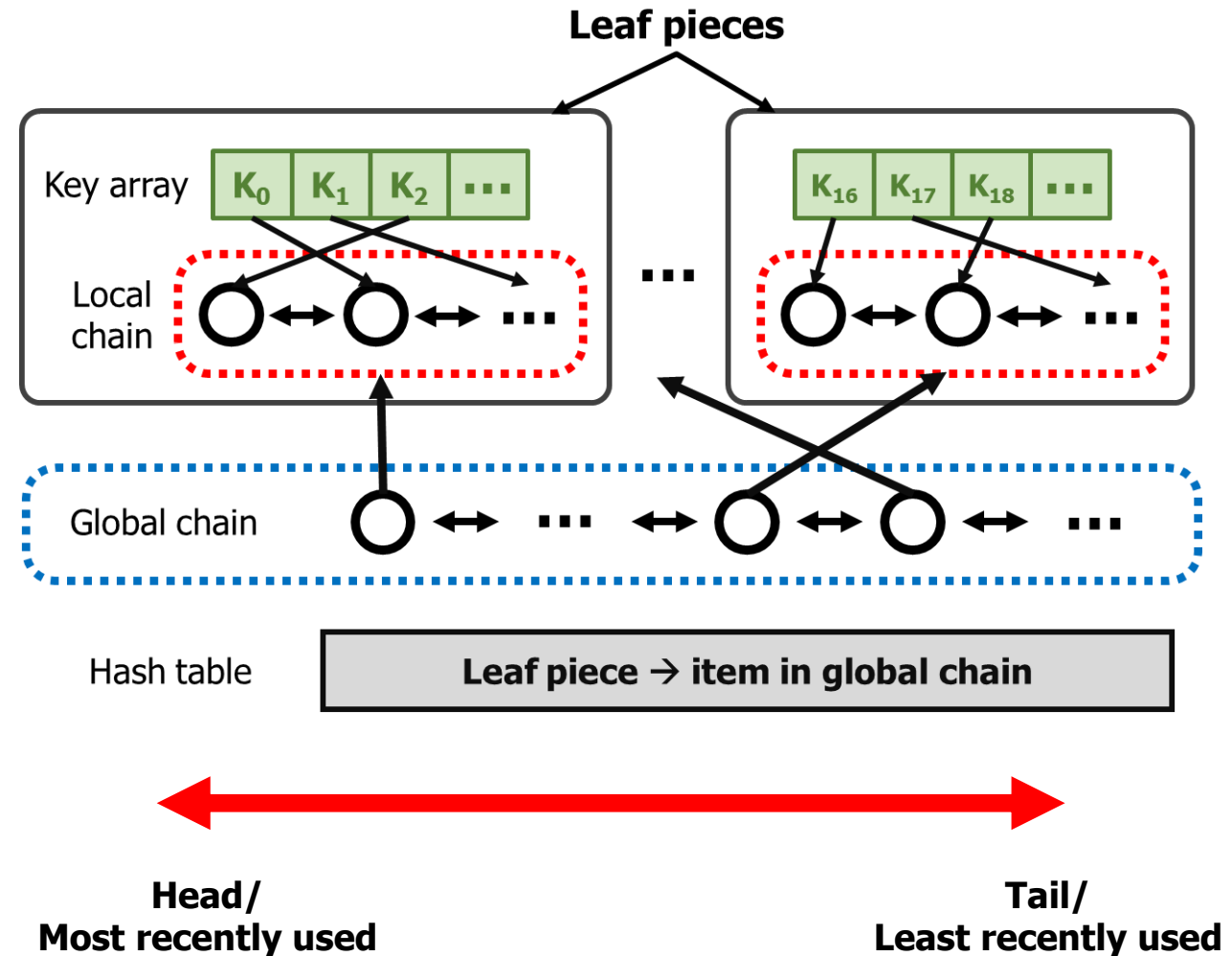
- Leaf piece and adaptive LRU of FILM
  - FILM consists of local, global chain to support adaptive LRU
  - Reduce the overhead of maintaining the LRU chain by **piggybacking** the maintenance of the LRU order on the index lookup





# Adaptive LRU

- Local chain
  - A component in a leaf piece
  - Maintain LRU order of the keys
  - Stores the payloads of the keys
- Global chain
  - Points to a leaf piece
  - Maintain LRU order of the l\_pieces
  - Track the global data access order across pieces
- Hash table
  - For fast search of nodes in global chain



# Adaptive LRU

## ■ Local chain

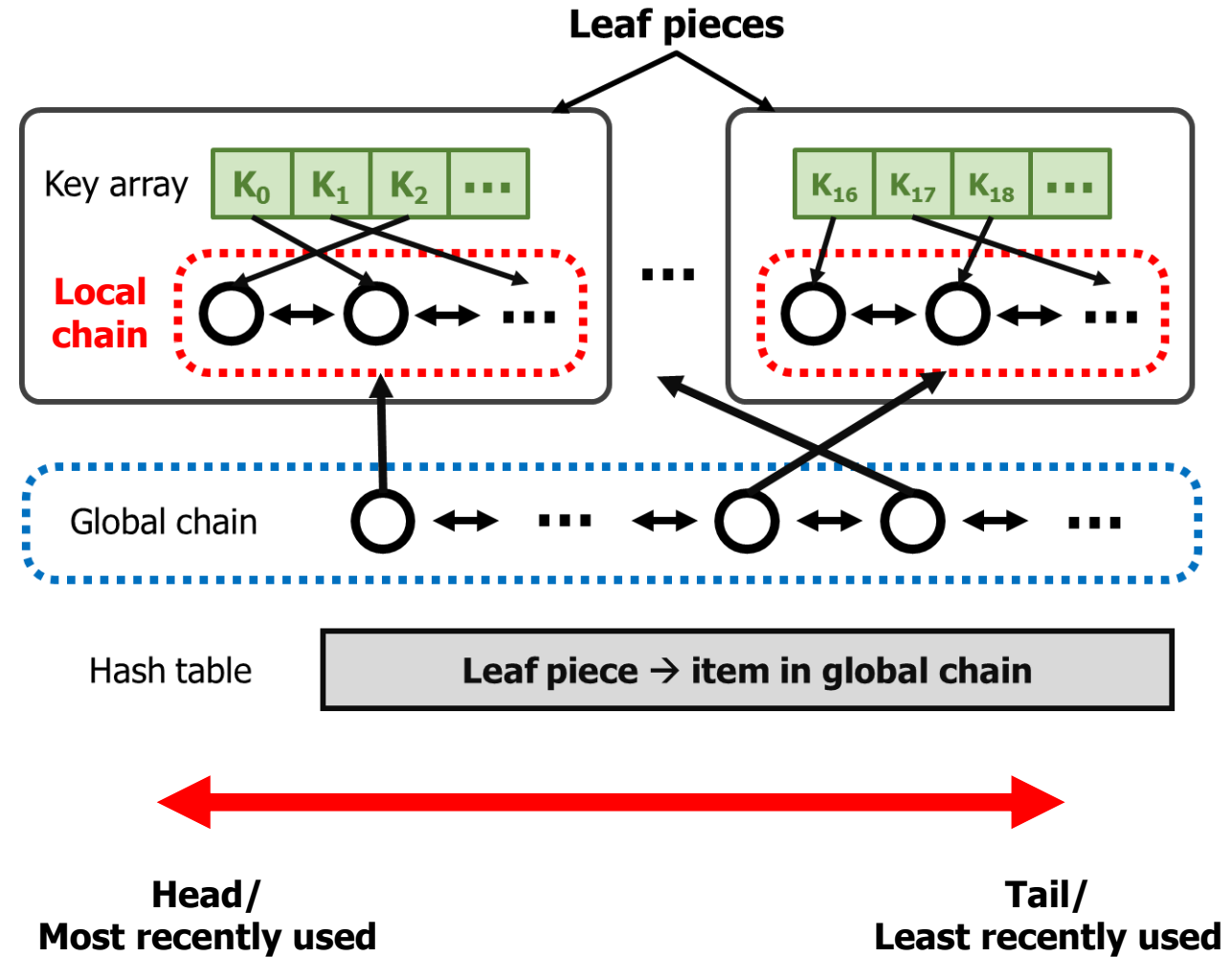
- A component in a leaf piece
- Maintain LRU order of the keys
- Stores the payloads of the keys

## ■ Global chain

- Points to a leaf piece
- Maintain LRU order of the l\_pieces
- Track the global data access order across pieces

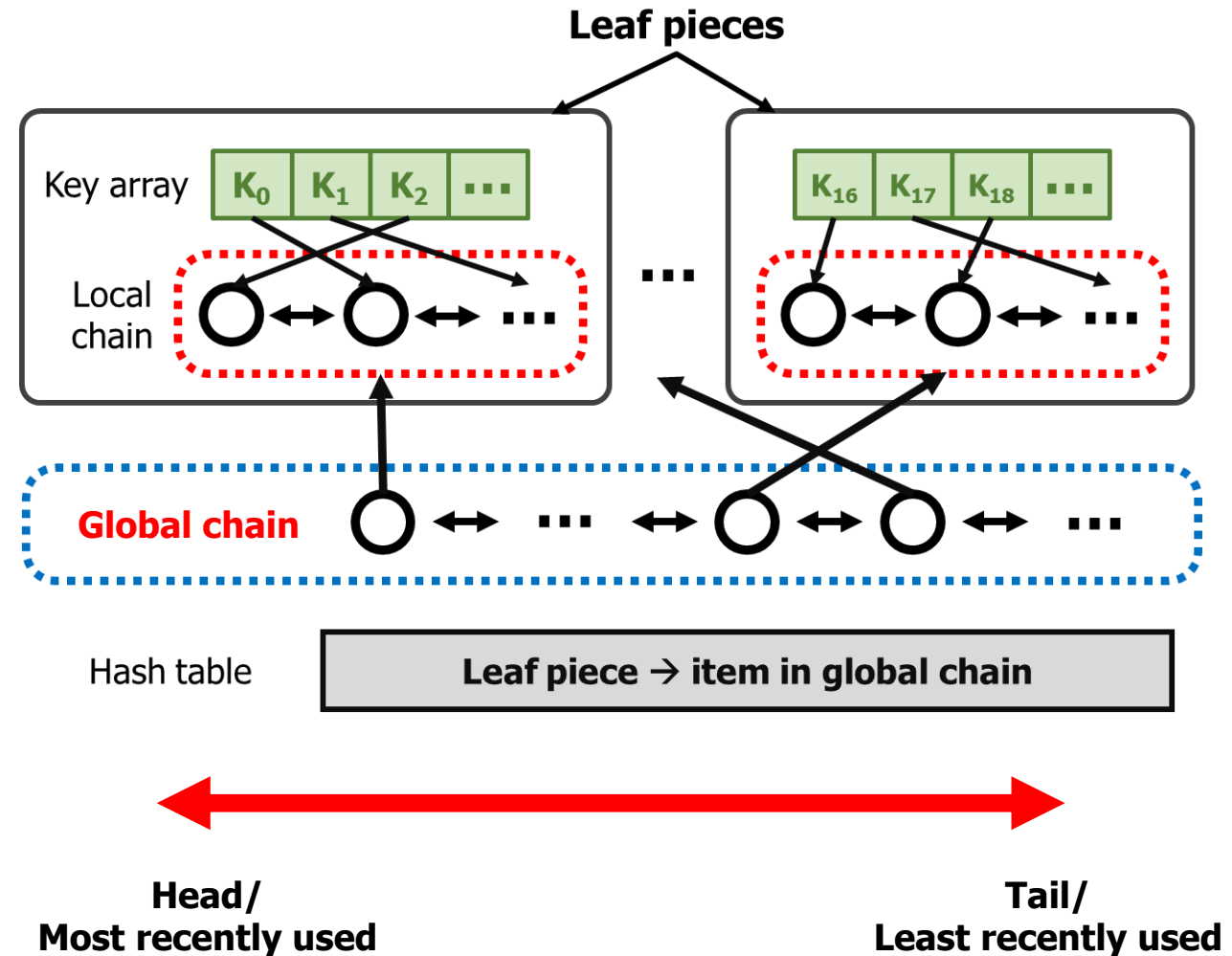
## ■ Hash table

- For fast search of nodes in global chain



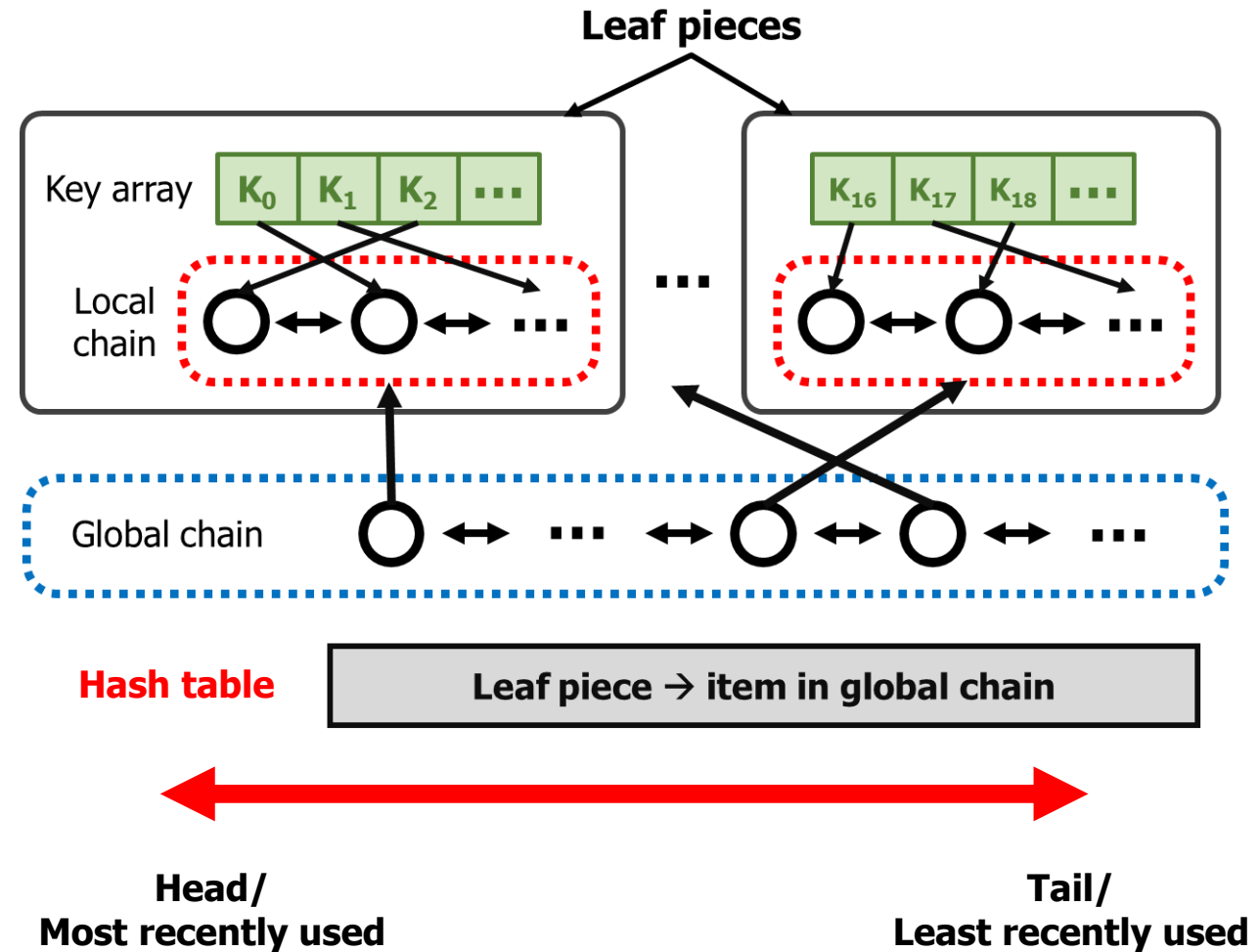
# Adaptive LRU

- Local chain
  - A component in a leaf piece
  - Maintain LRU order of the keys
  - Stores the payloads of the keys
- Global chain
  - Points to a leaf piece
  - Maintain LRU order of the l\_pieces
  - Track the global data access order across pieces
- Hash table
  - For fast search of nodes in global chain



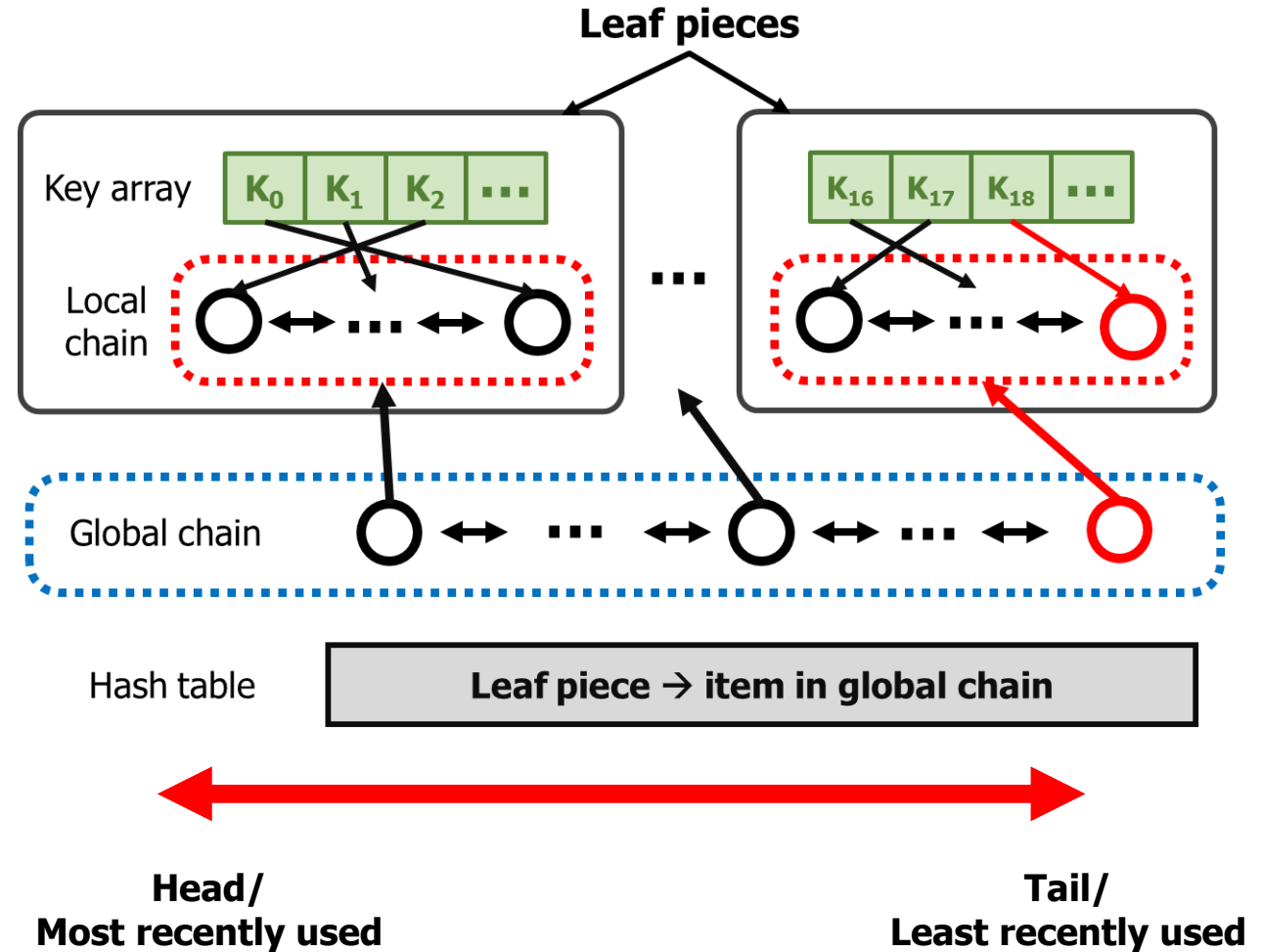
# Adaptive LRU

- Local chain
  - A component in a leaf piece
  - Maintain LRU order of the keys
  - Stores the payloads of the keys
- Global chain
  - Points to a leaf piece
  - Maintain LRU order of the l\_pieces
  - Track the global data access order across pieces
- Hash table
  - For fast search of nodes in global chain



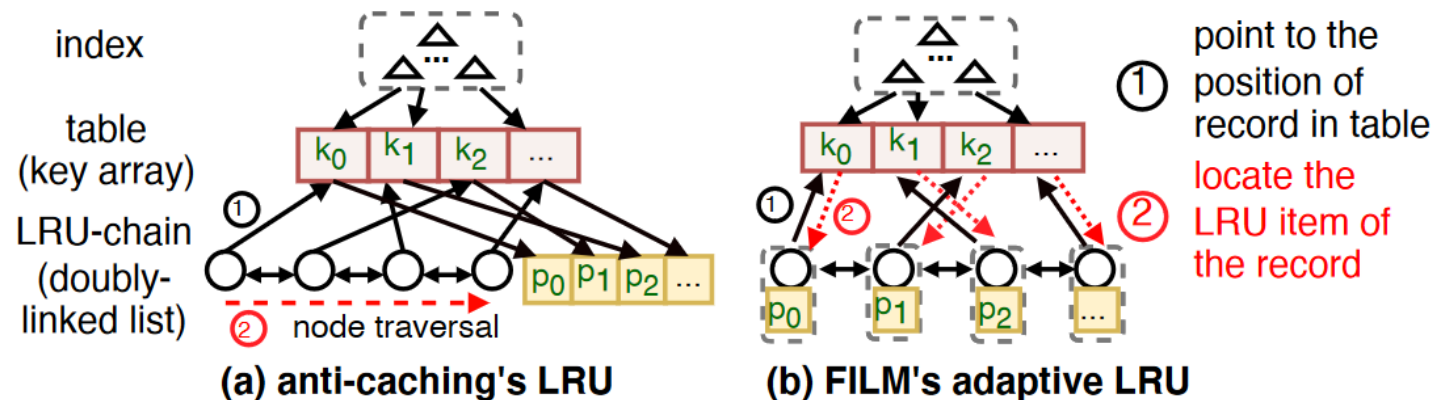
# Adaptive LRU

- Cold Data identification
  - When available memory runs out
    - Global chain identifies cold leaf piece to eviction
    - Evicts records from the coldest leaf to a block until block is full
  - Continue until database size reaches a user-specified threshold



# Adaptive LRU

- Difference between FILM's adaptive LRU and anti-caching's LRU
  - FILM avoids the node traversal by piggybacking the locating of the LRU item onto the index lookup
  - An LRU item in FILM's adaptive LRU stores the payload of the corresponding record



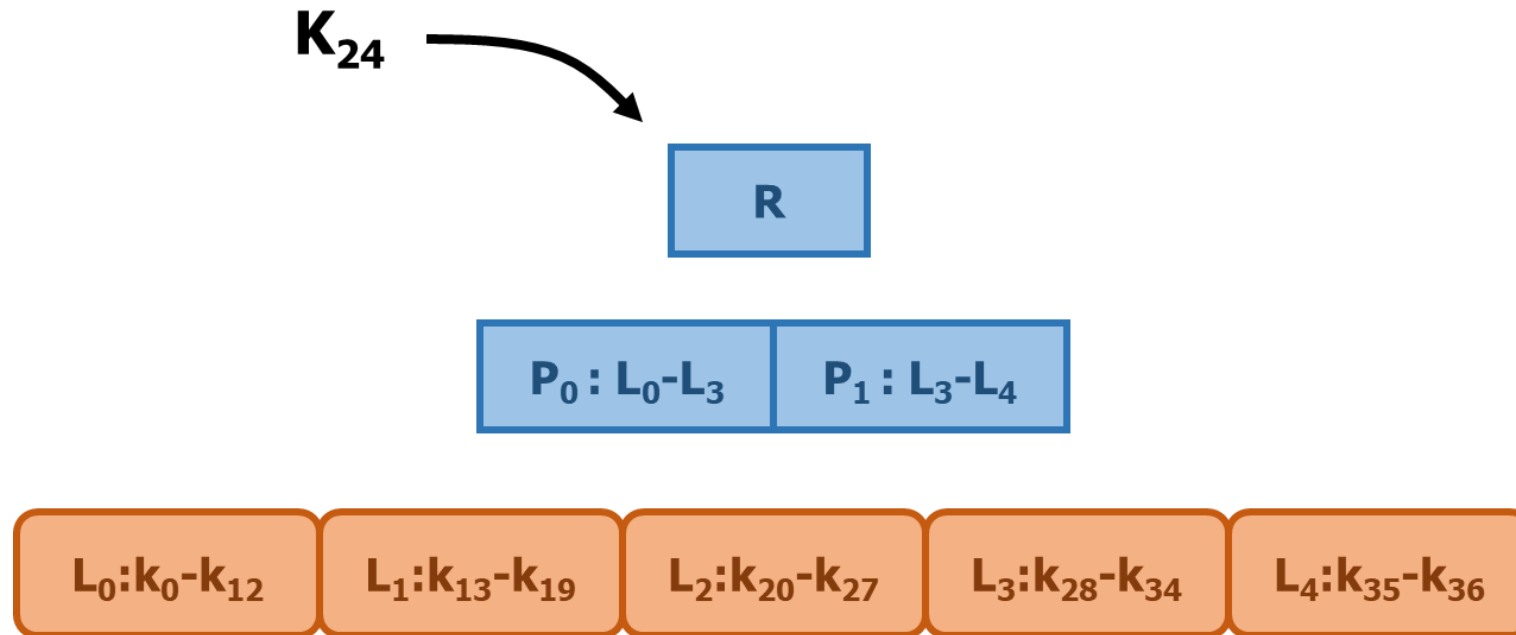
Note: In anti-caching, the pointers of LRU chain are embeds in the records' headers. To show the difference when locating LRU items more intuitive, we separate them in this toy example.

**Fig. 4: LRU in anti-caching vs. adaptive LRU in FILM**

# Query Processing

- Point Query

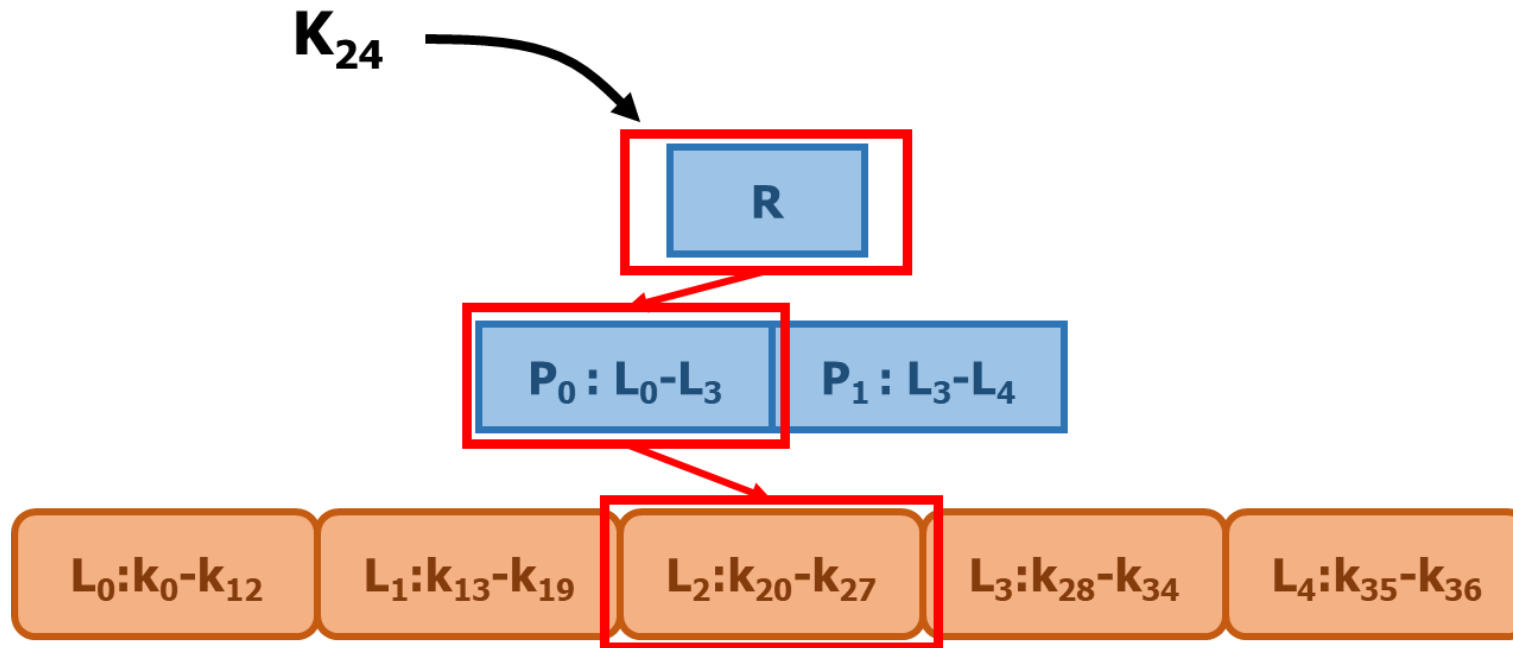
- Find the leaf piece that **K** belongs to  
Recursively predict the candidate pieces



# Query Processing

- Point Query

- Find the leaf piece that **K** belongs to  
Recursively predict the candidate pieces



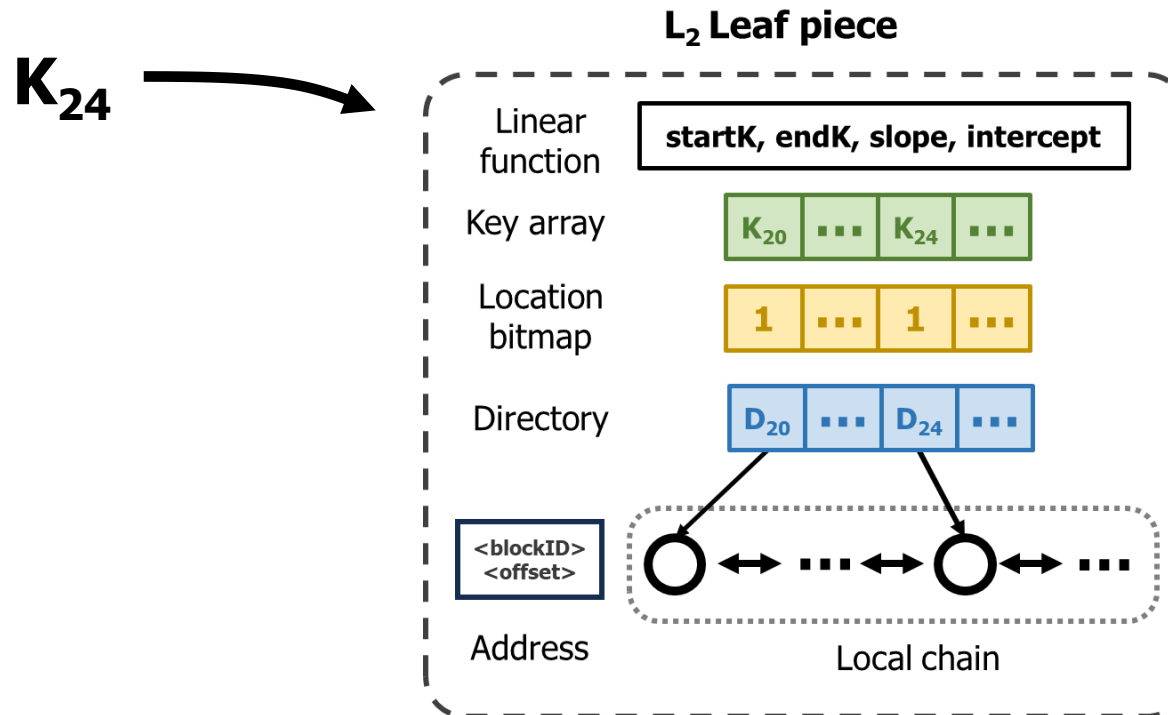


# Query Processing

- Point Query

- 2. Locate **K** in the piece

- Predict the position of **K**

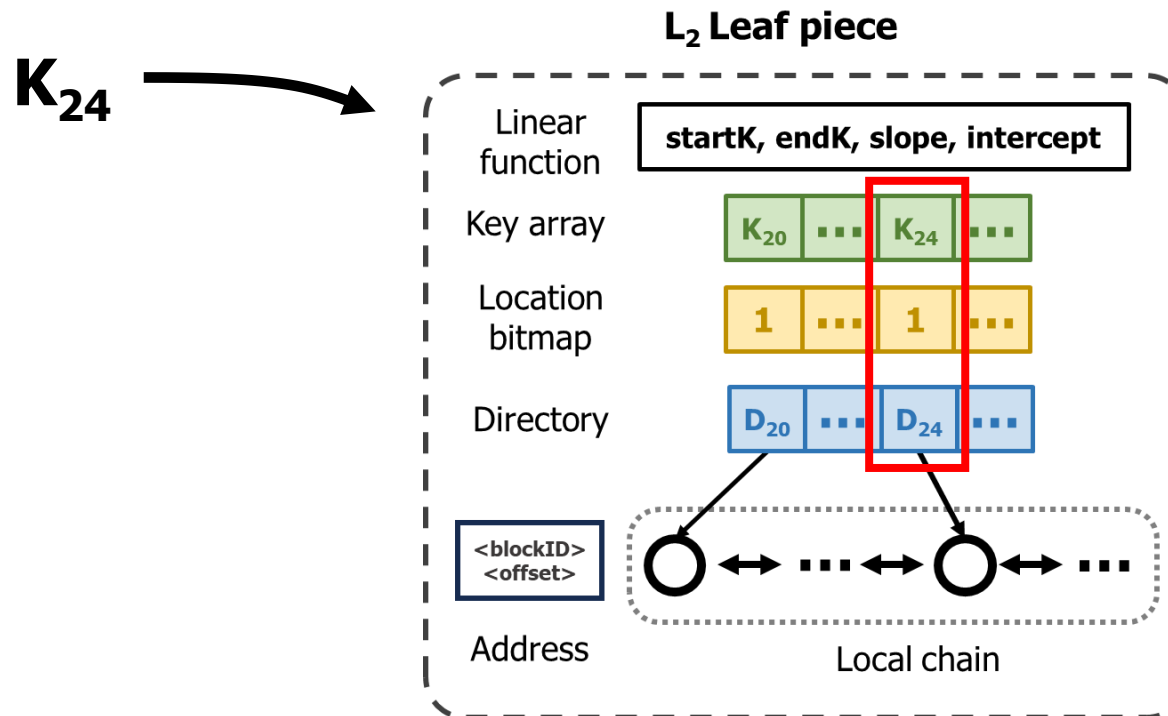


# Query Processing

- Point Query

- 2. Locate **K** in the piece

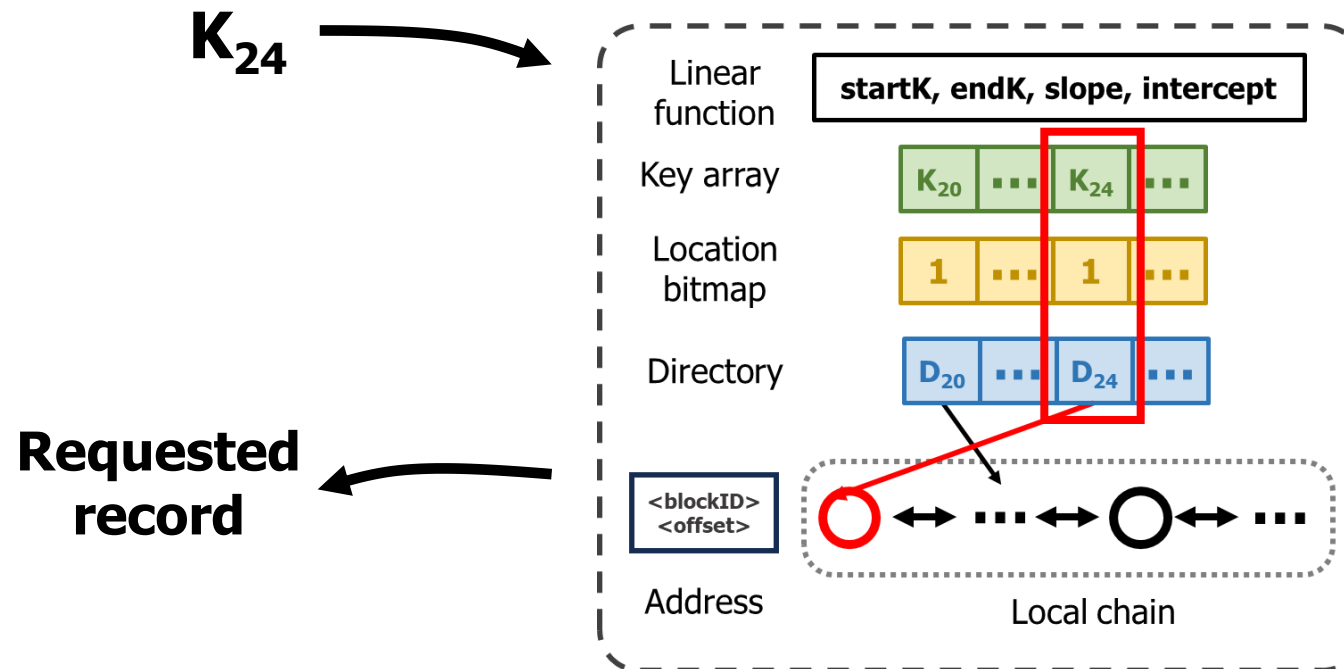
FILM uses the location bitmap to know if k's payload is in memory or on disk



# Query Processing

- Point Query

- Retrieve the data record and update the adaptive LRU  
Record can be directly accessed using local chain or address

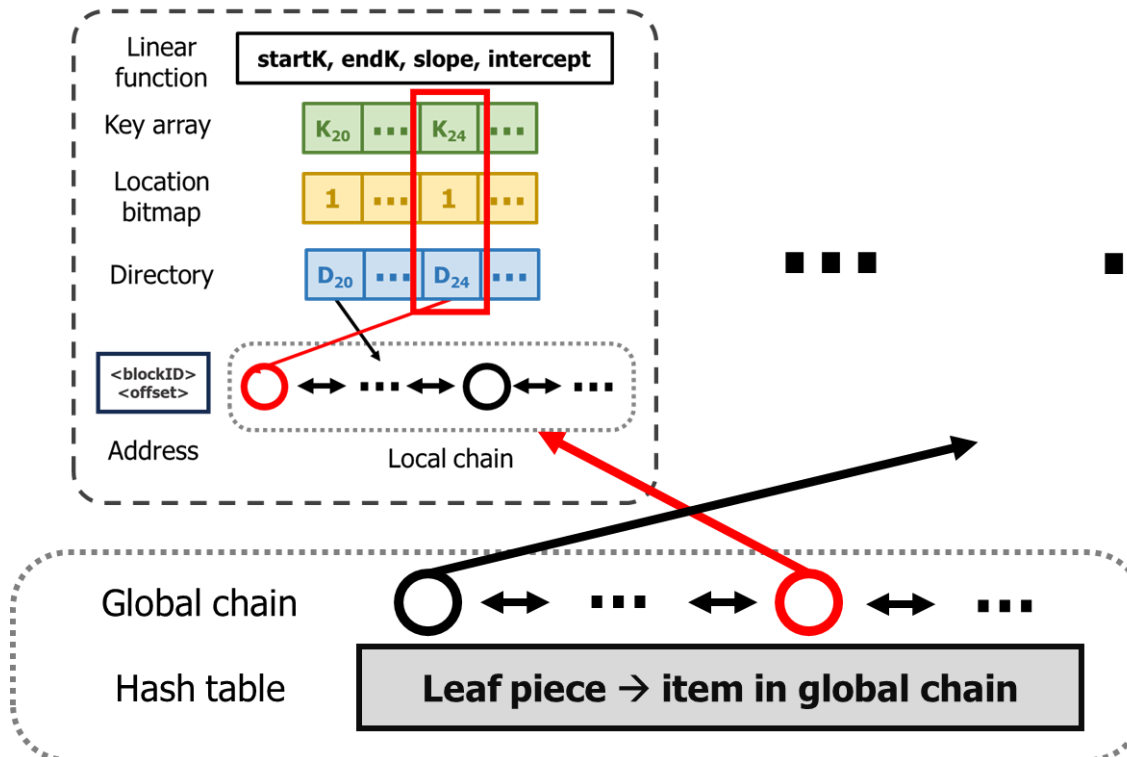


# Query Processing

- Point Query

3. Retrieve the data record and update the adaptive LRU

The accessed leaf will be moved to the head of the global chain

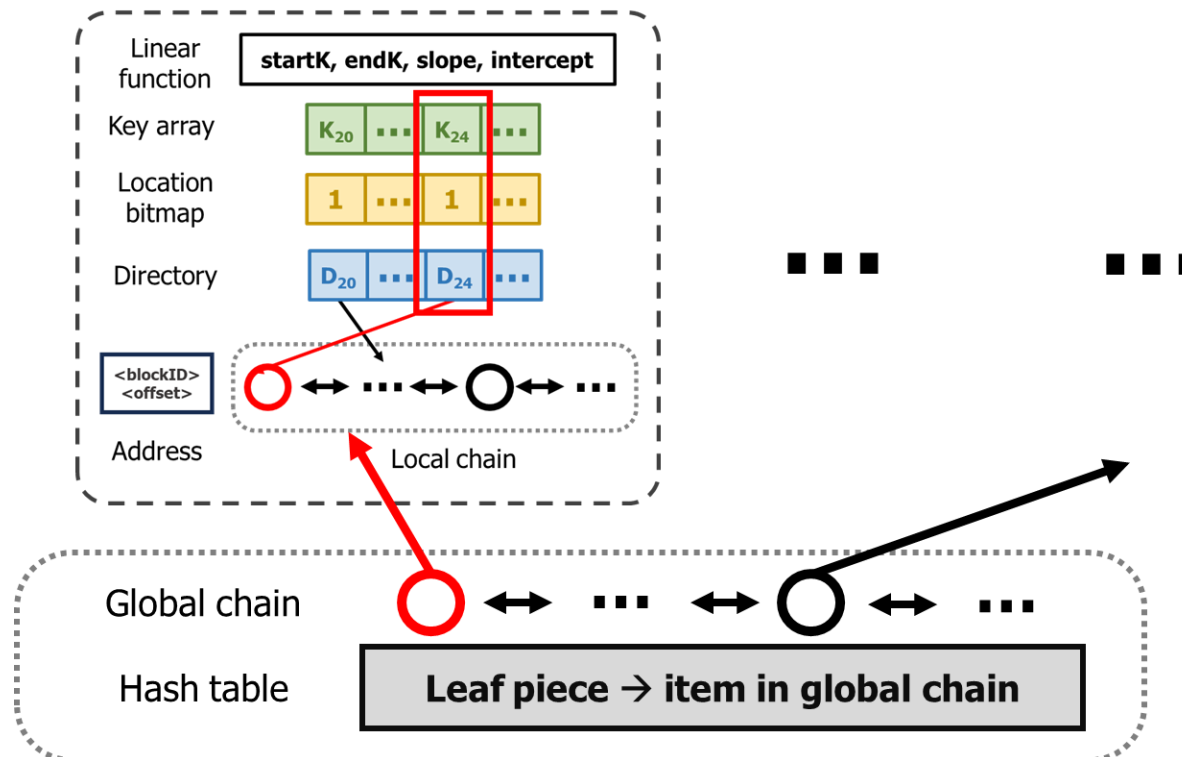


# Query Processing

- Point Query

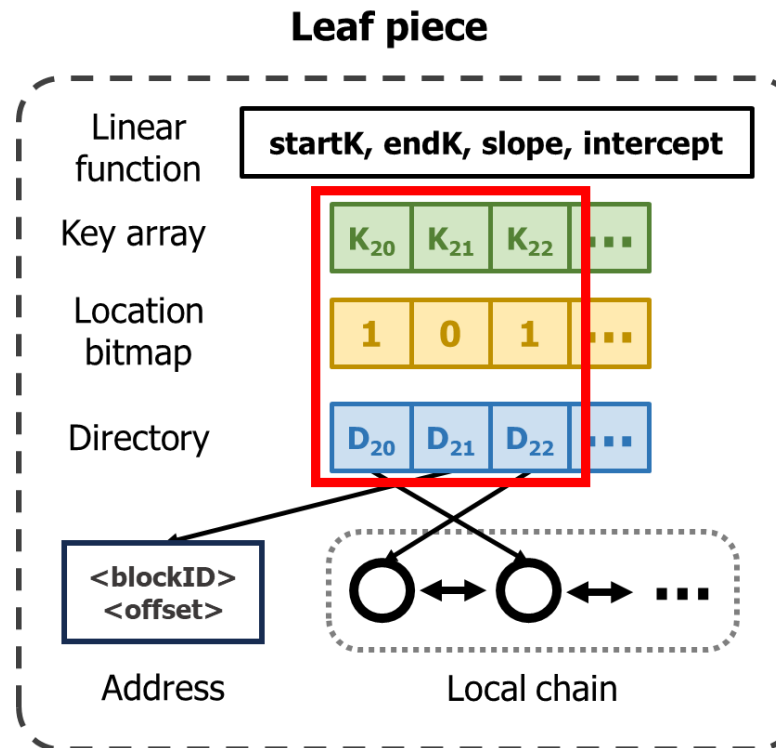
3. Retrieve the data record and update the adaptive LRU

The accessed leaf will be moved to the head of the global chain



# Query Processing

- Range Query
  - FILM handles requested data records similarly whether they reside in a single type or multiple types of storage devices.



# Experiments

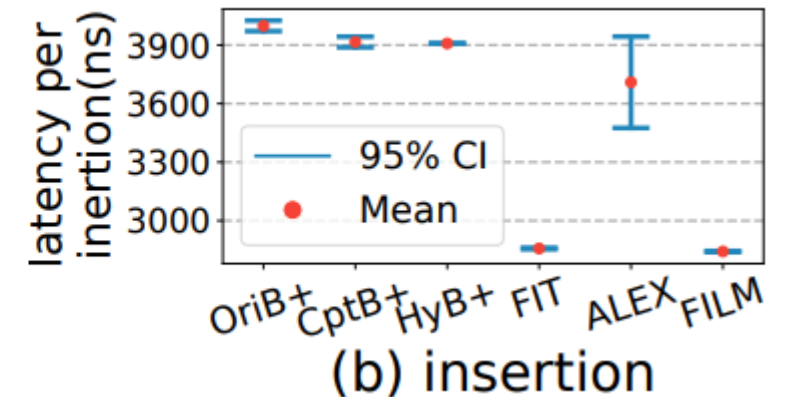
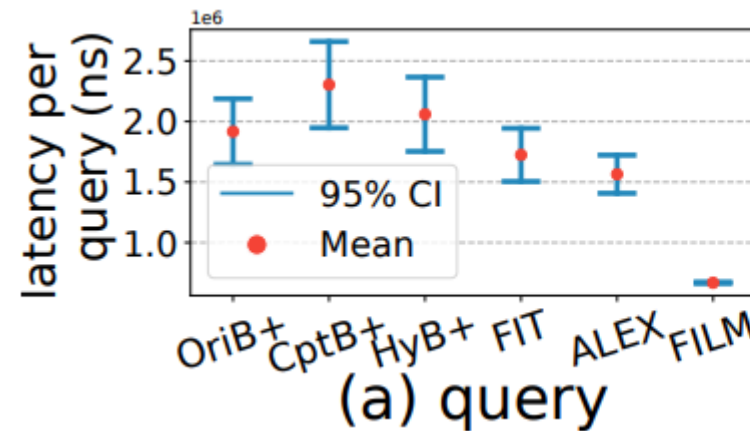
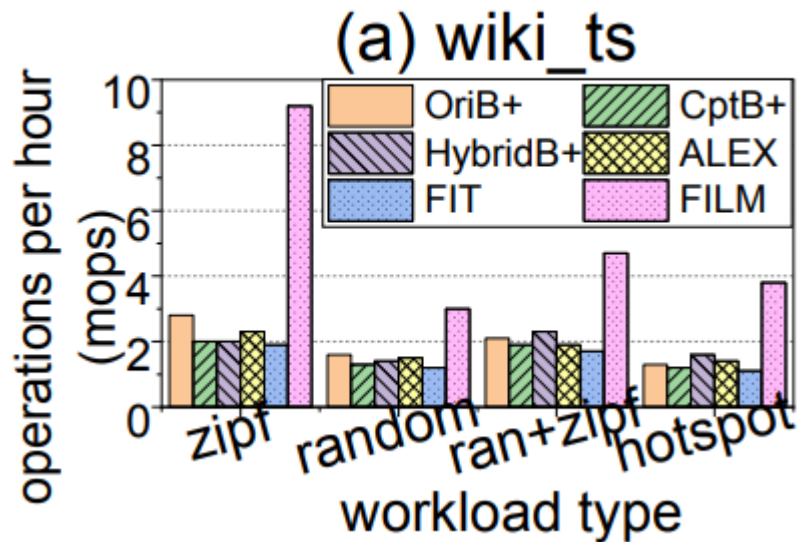
- Baselines and Experiment Setup

- Comparison(Base Line) : B+tree, HybridB+tree, CptB+tree, Alex, Fitting-Tree
- Datasets : wiki\_ts, books, astro\_a, synthetic, YCSB (64bit Key, 128Byte Value)
- Workload : Zipfian, Hotspot, Random, Zipfian+Random
- Spec :
  - 3.6GHz Intel CPU with 256KB L1 cache
  - 128GB memory (4 × 32GB)
  - 557GB disk
  - Single thread
  - Direct I/O

# Experiments

## ■ Comparison with Baselines

- Insertion and query performance(Avg. 4GB All dataset, 1:1 , 1 hour)



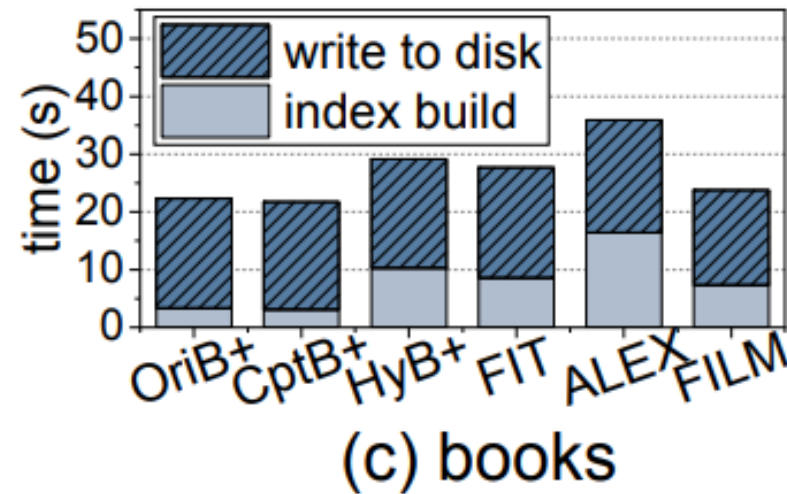
FILM has a **2-5 times advantage** over base line in terms of insertion and query performance.

Film also has the **lowest latency and more stable performance**.



# Experiments

- Comparison with Baselines
  - Index construction



The learned index has **advantages** over traditional indexes **in memory usage**, but correspondingly, it has **disadvantages in build time** (about 2.2~2.5 times).

Also, due to occupying less memory, it has more **advantages in writing to disk**.

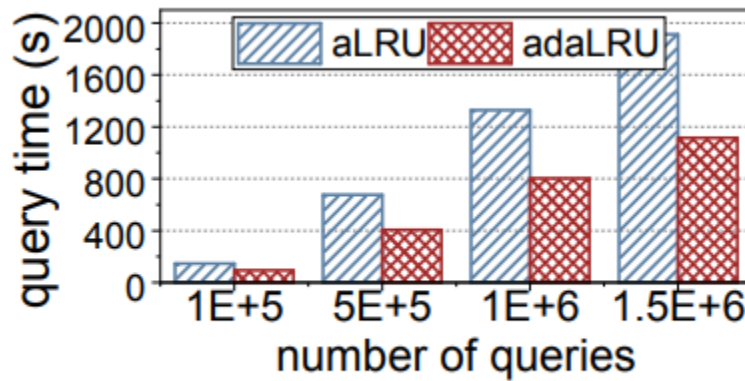
# Experiments

## ■ Comparison with Baselines

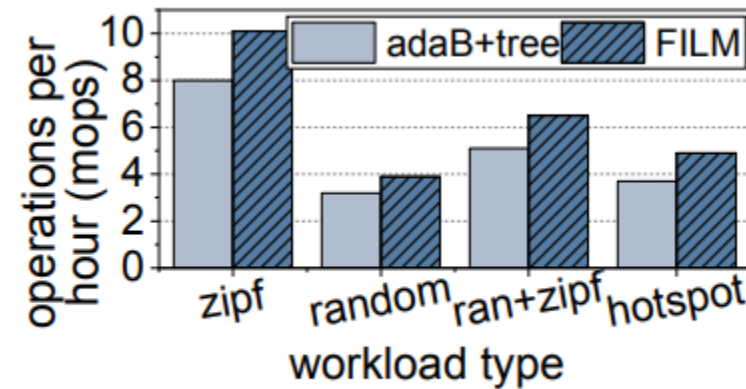
- LRU overhead(books, 4GB dataset, 100000 random queries)

aLRU : LRU with a sampling rate of 0.01

adaLRU : Adaptive LRU



(a) aLRU vs. adaptive LRU

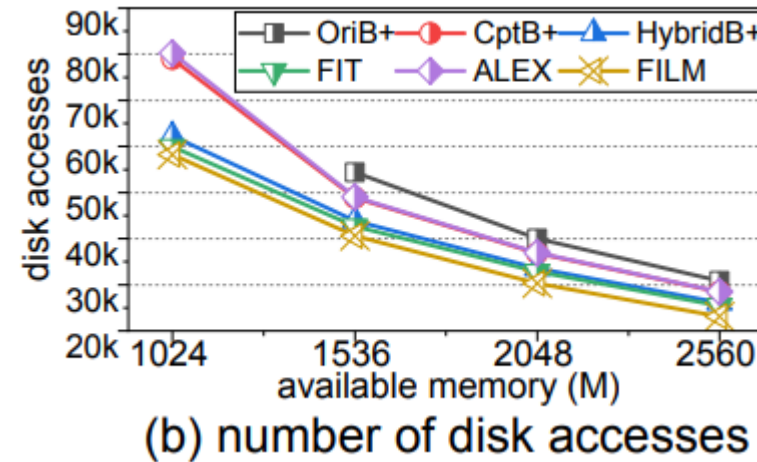
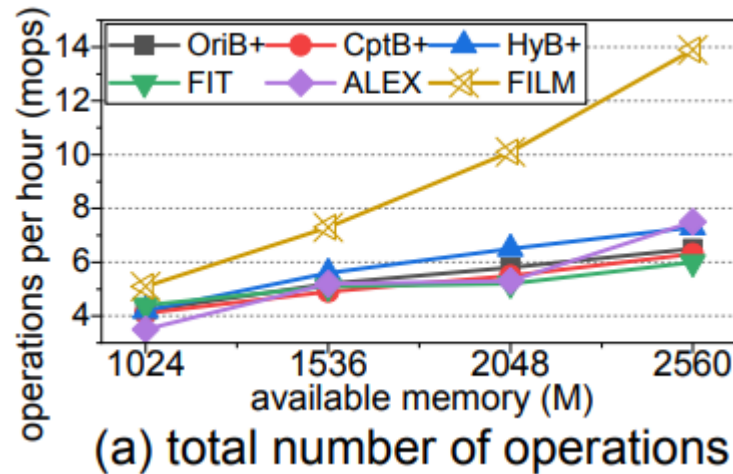


(b) adaB+tree vs. FILM

On LRU, adaLRU is superior to aLRU.  
On the index, FILM is superior to traditional indexes.

# Experiments

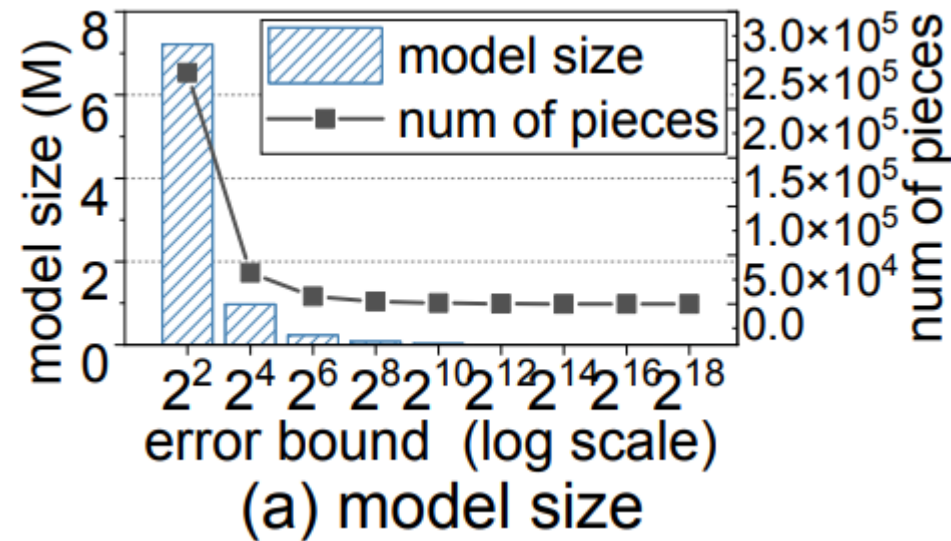
- Study of Environment Parameters
  - Available memory(4GB wiki\_ts dataset, Zipfian workload)



Increasing memory can increase the number of operations per unit time and reduce disk io, with FILM gaining more significant benefits in this regard.

# Experiments

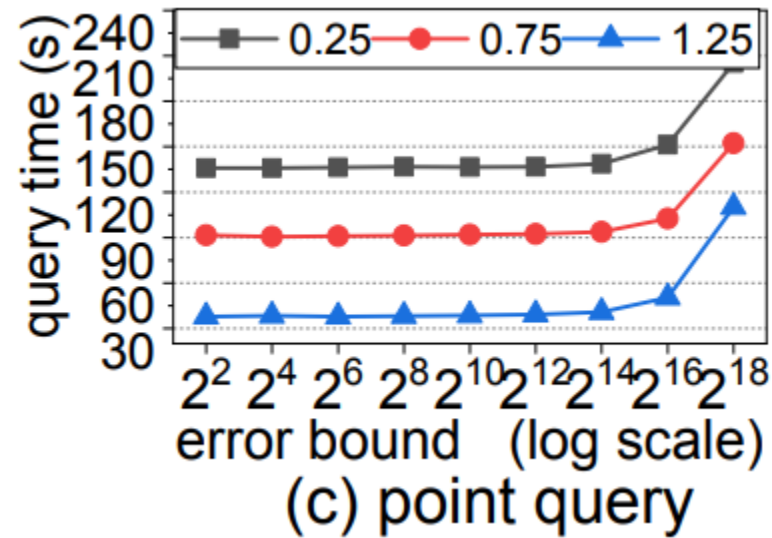
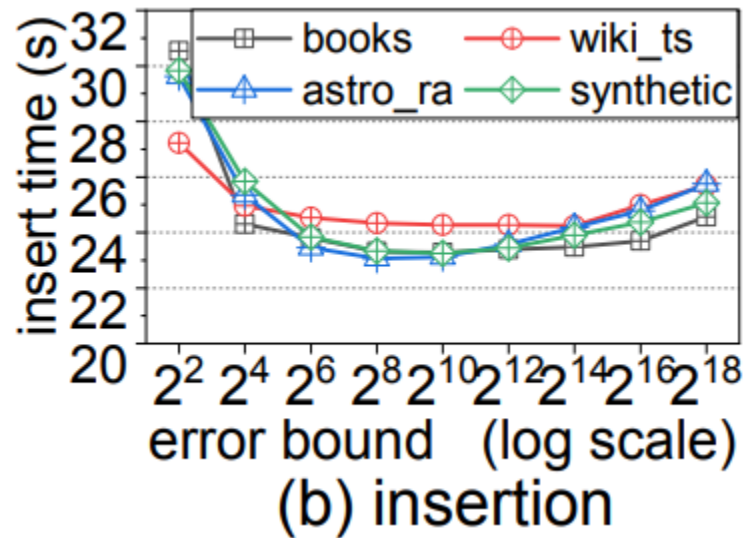
- Sensitivity to Data and Model Parameters
  - Error bound  $\varepsilon$



When the error bound increases from 4 to 16, the sharp decrease in the number of pieces fitted by FILM leads to a decrease in the memory occupied by FILM, which then stabilizes.

# Experiments

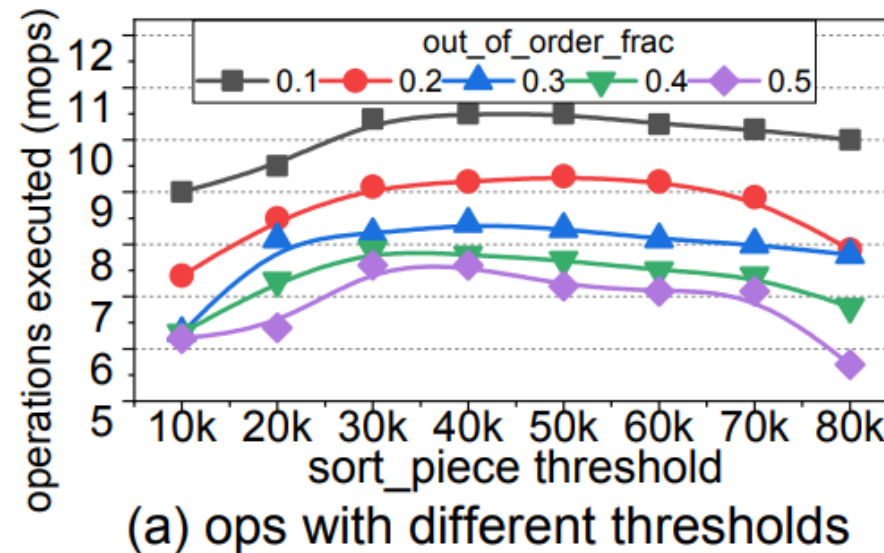
- Sensitivity to Data and Model Parameters
  - Error bound  $\epsilon$



FILM within the range of 16 to  $2^{12}$   $\epsilon$  having the best performance.

# Experiments

- Sensitivity to Data and Model Parameters
  - Handling Out-of-Order Insertions(sort\_piece)



Both too small and too large sort pieces can lead to performance loss.  
30000~70000 is the optimal range.

# Thank you