AirIndex: Versatile Index Tuning Through Data and Storage

Supawit Chockchowwat, Wenjie Liu, Yongjoo Park

SIGMOD '24e (Accepted 23 May 2023)

2024. 09. 11

Presentation by ZHU YONGJIE

harasho2015@dankook.ac.kr

DANKOOK UNIVERSITY

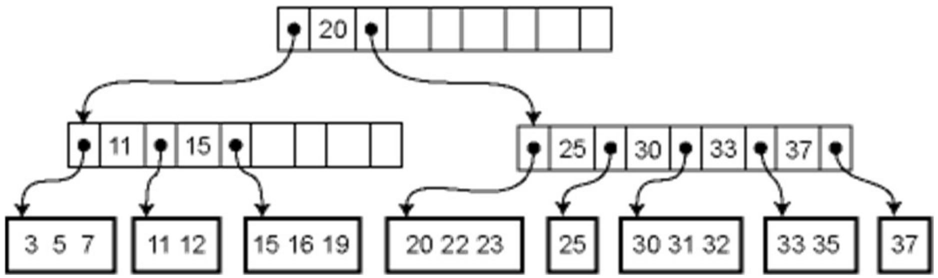
Dankook University System Software Laboratory

Contents

Introduction

- Introduction to Index Structures

Key ⟶ Position



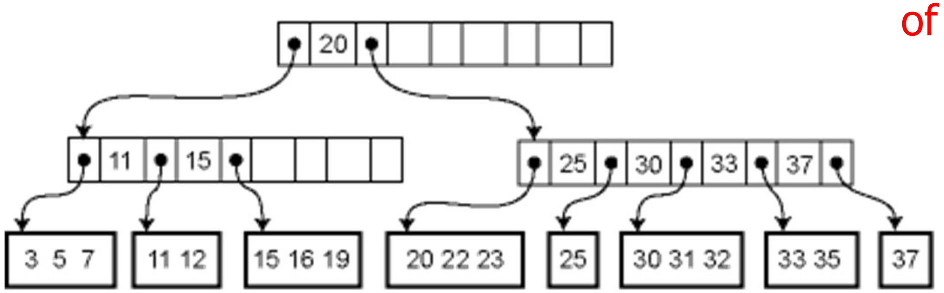
B-Tree

Introduction

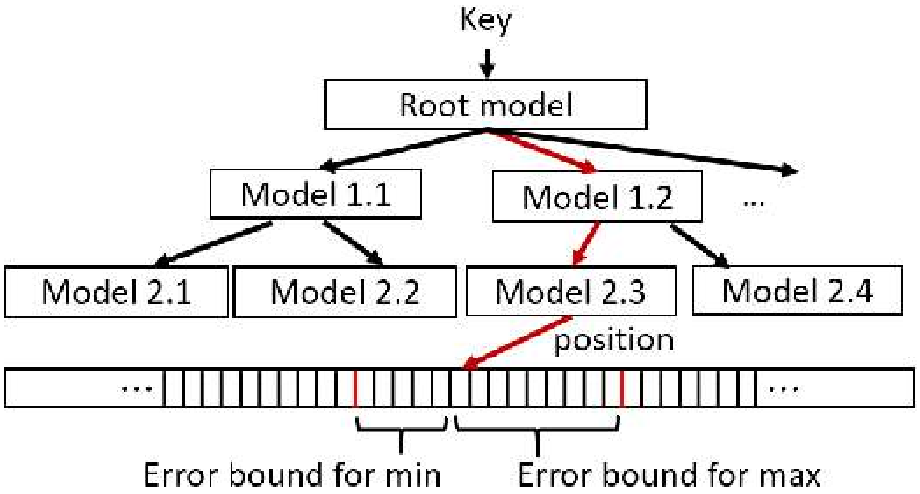
- Introduction to Index Structures

Key → Position

With the growth
of data volume



B-Tree

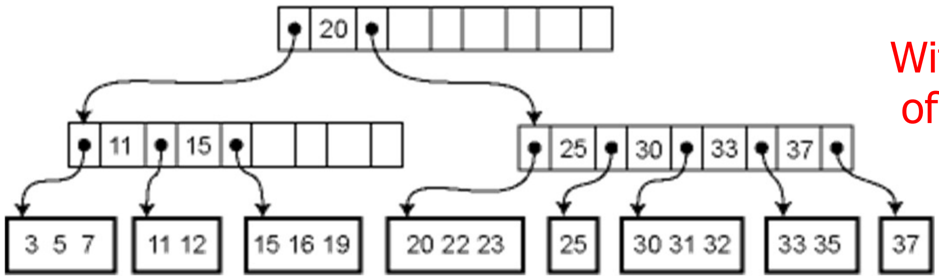


RMI (Learned index)

Introduction

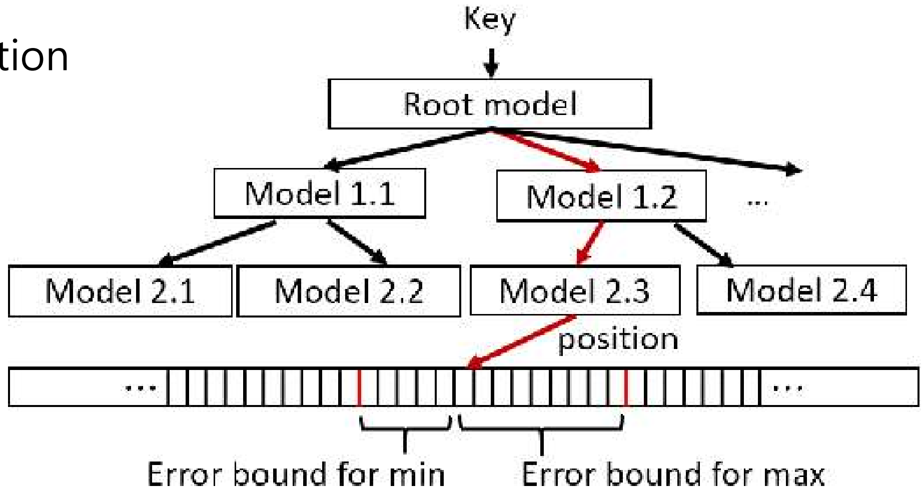
- Introduction to Index Structures

Key → Position



B-Tree

With the growth of data volume



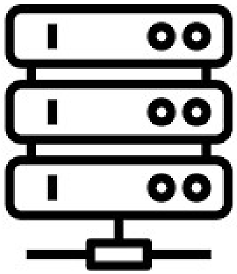
RMI (Learned index)

Different Device environments
&
Different datasets

Indexes need tuning

Introduction

- Limitations of Existing Indexes



Local SSD

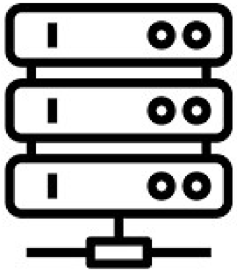
Fast I/O latency
Relatively smaller bandwidth

Cloud Server

High I/O latency
Larger bandwidth

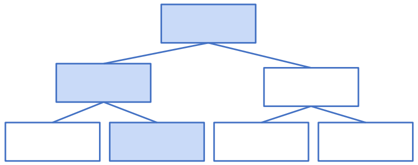
Introduction

- Limitations of Existing Indexes



Local SSD

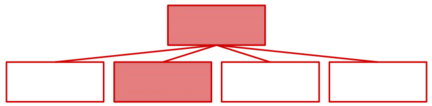
Fast I/O latency
Relatively smaller bandwidth



B200: B-tree with 200 child pointers

Cloud Server

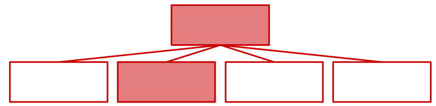
High I/O latency
Larger bandwidth



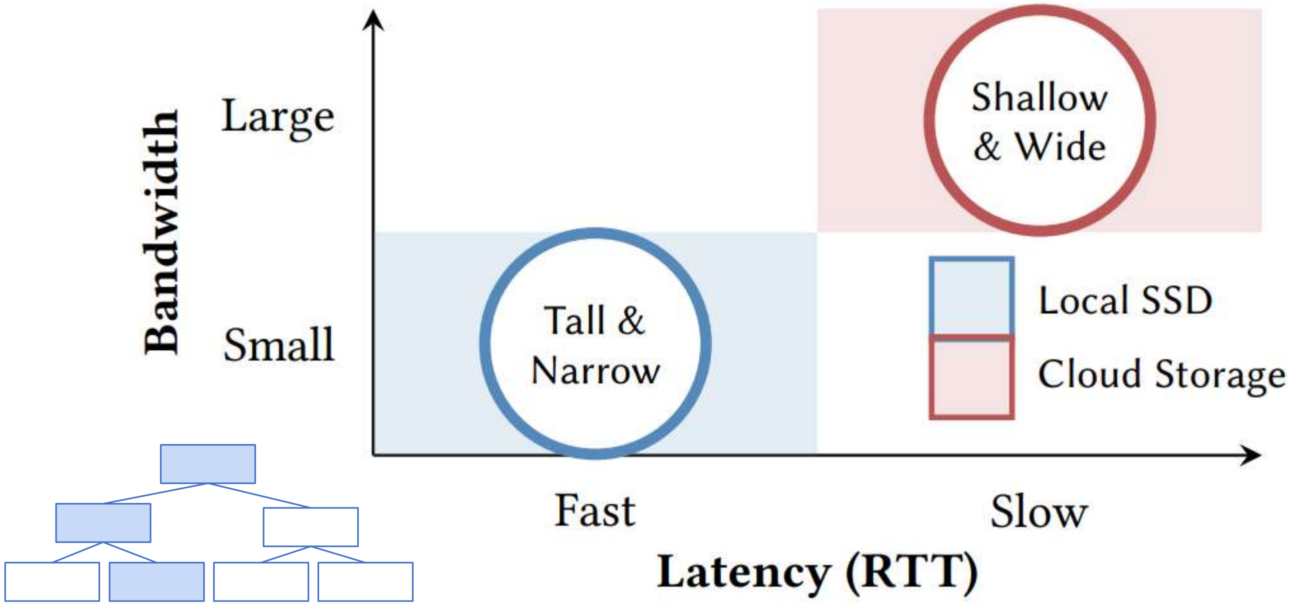
B5000: B-tree with 5000 child pointers

Introduction

- Limitations of Existing Indexes



B5000: B-tree with 5000 child pointers

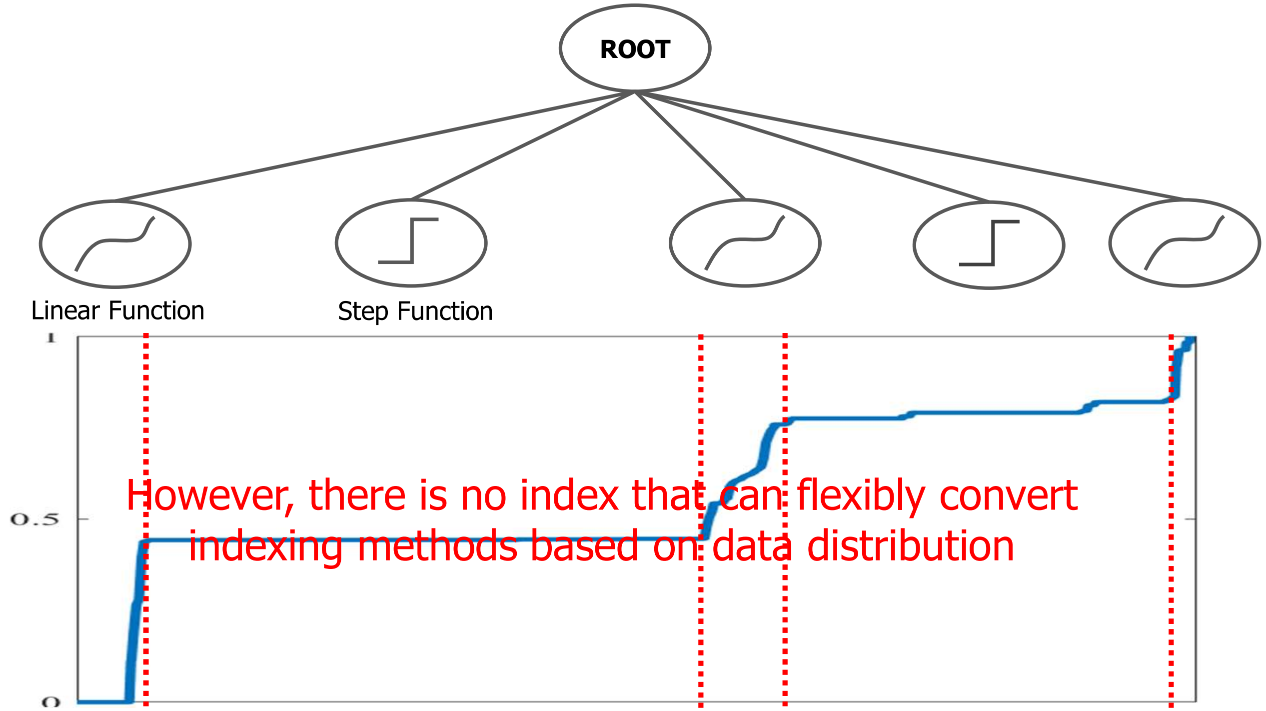


B200: B-tree with 200 child pointers

But there is no index that can adapt to the storage profile

Introduction

- Limitations of Existing Indexes



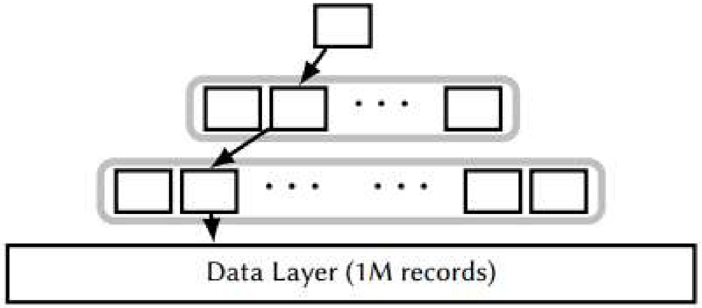
Keys CDF

Contents

Motivation

- Why does the index need to adapt to different storage profiles

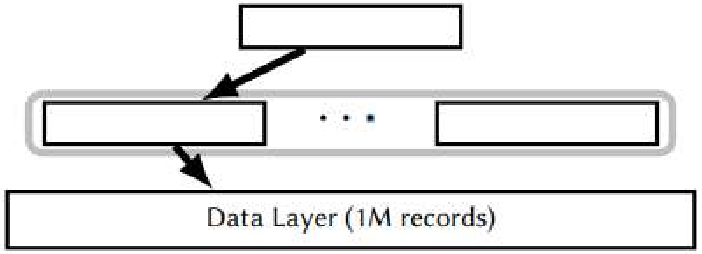
Node Size = 4KB



(a) B200: **B-tree with 200 child pointers**

200*200*200 = 8M > 1M

Node Size = 100KB



(b) B5000: **B-tree with 5,000 child pointers**

5000*5000 = 25M > 1M

(**data transfer time**) = (latency) + (data size) / (bandwidth)

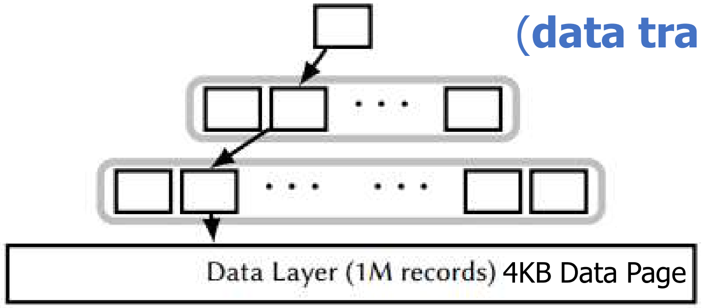
Motivation

- Why does the index need to adapt to different storage profiles

Node Size = 4KB

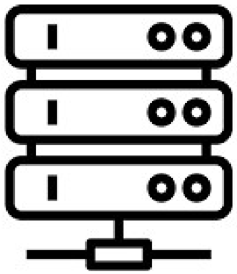
Node Size = 100KB

(**data transfer time**) = (latency) + (data size) / (bandwidth)



(a) B200: **B-tree with 200 child pointers**

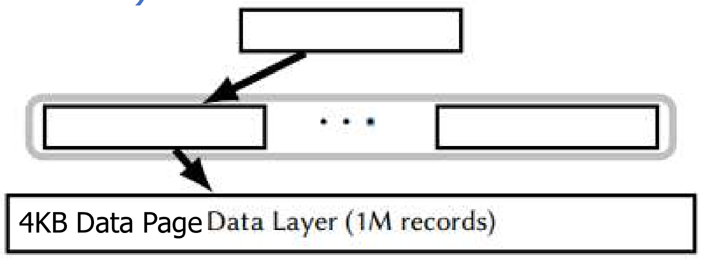
200*200*200 = 8M > 1M



Local SSD

Latency = 100μs

Bandwidth = 1GB/s



(b) B5000: **B-tree with 5,000 child pointers**

5000*5000 = 25M > 1M

3 × (100μs + 4KB / (1GB/s)) + (100μs + 4KB / (1GB/s))

=**416μs**

2 × (100μs + 100KB / (1GB/s)) + (100μs + 4KB / (1GB/s))

=**504μs**

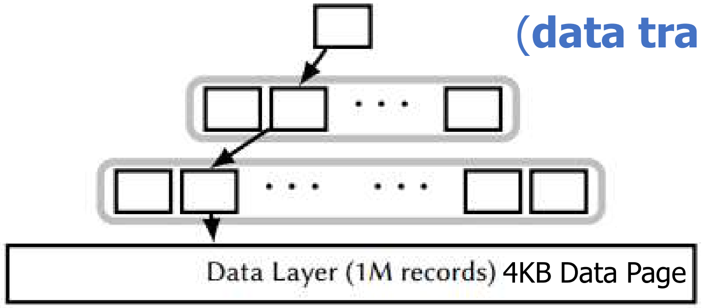
Motivation

- Why does the index need to adapt to different storage profiles

Node Size = 4KB

Node Size = 100KB

(**data transfer time**) = (latency) + (data size) / (bandwidth)



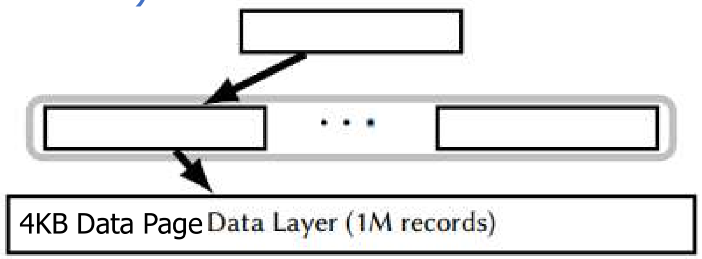
(a) B200: **B-tree with 200 child pointers**

200*200*200 = 8M > 1M

Cloud Server

Latency = 100ms

Bandwidth = 100MB/s



(b) B5000: **B-tree with 5,000 child pointers**

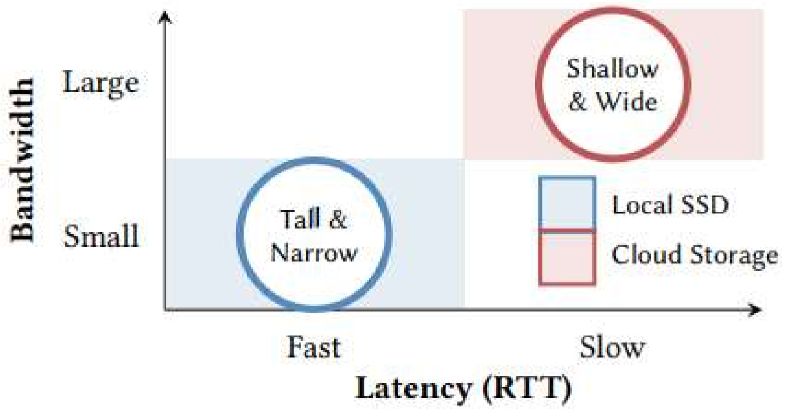
5000*5000 = 25M > 1M

3 × (100ms + 4KB / (100MB/s)) + (100ms + 4KB / (100MB/s))
=**400.16ms**

2 × (100ms + 100KB / (100MB/s)) + (100ms + 4KB / (100MB/s))
=**302.04ms**

Motivation

- Why does the index need to adapt to different storage profiles



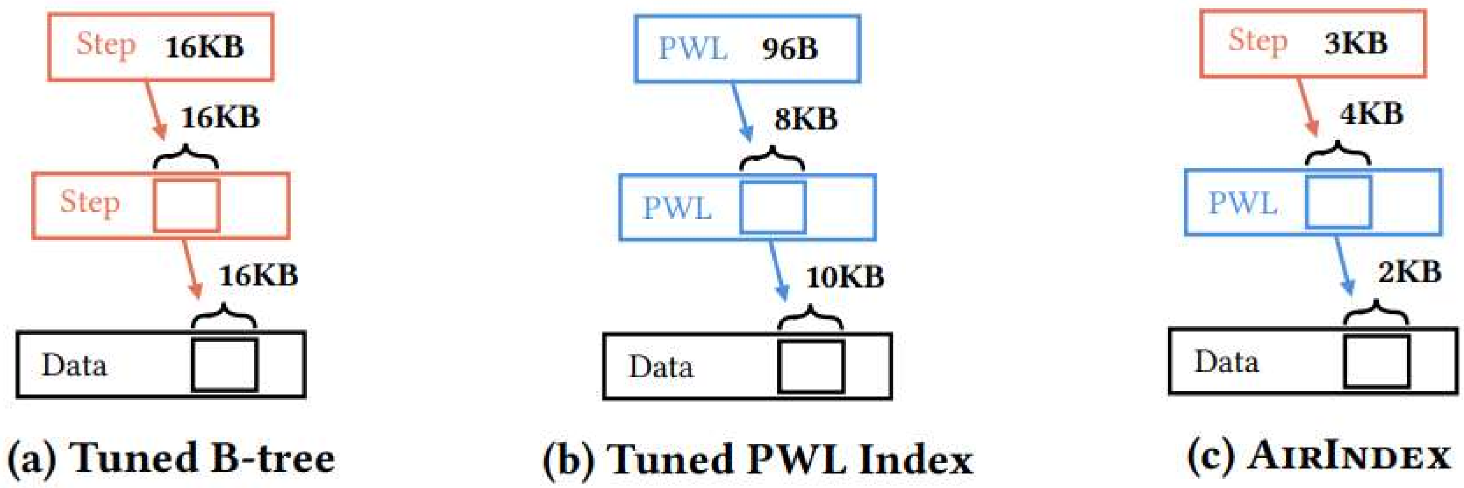
(c) Lookup Speed Comparison

In **local SSDs**, the impact of index depth is not as significant as index width,
but the opposite is true in **cloud storage**

Motivation

- Why does the index need to adapt to different data distributions

Gmm dataset, SSD(250μs latency, 175MB/s bandwidth)

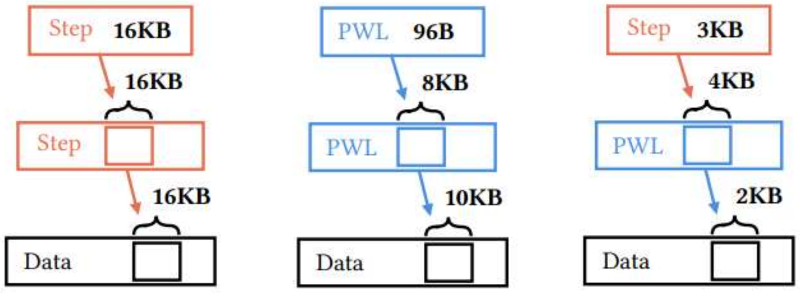


48KB | 18.096KB | 9KB

Motivation

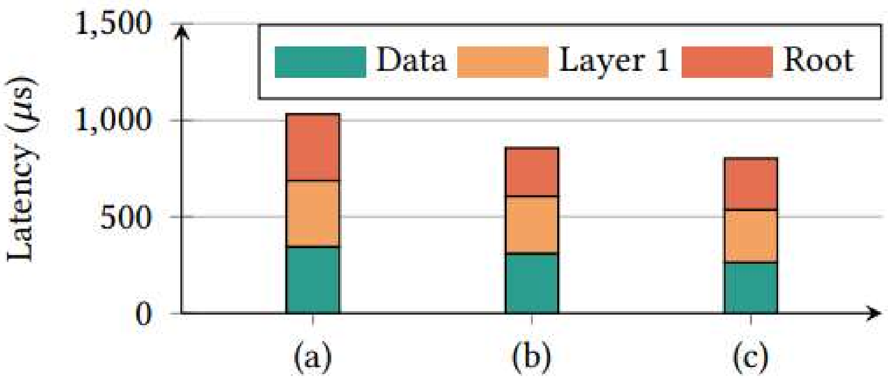
- Why does the index need to adapt to different data distributions

Gmm dataset, SSD(250μs latency, 175MB/s bandwidth)



(a) Tuned B-tree (b) Tuned PWL Index (c) AIRINDEX

48KB 18.096KB 9KB

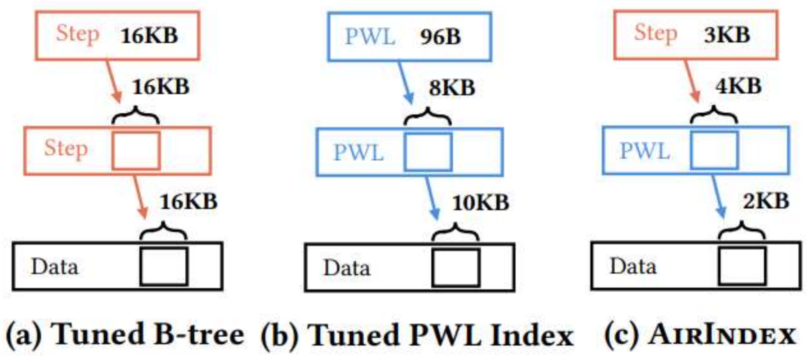


(d) Lookup Speed Comparison

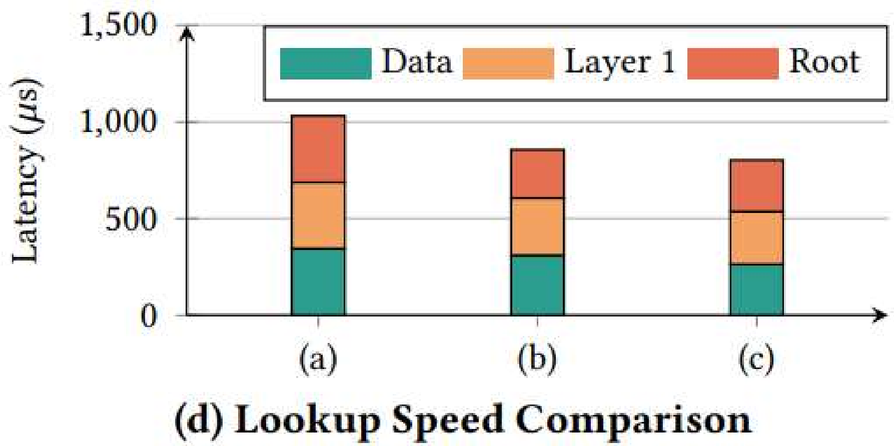
Motivation

- Why does the index need to adapt to different data distributions

Gmm dataset, SSD(250μs latency, 175MB/s bandwidth)



48KB 18.096KB 9KB

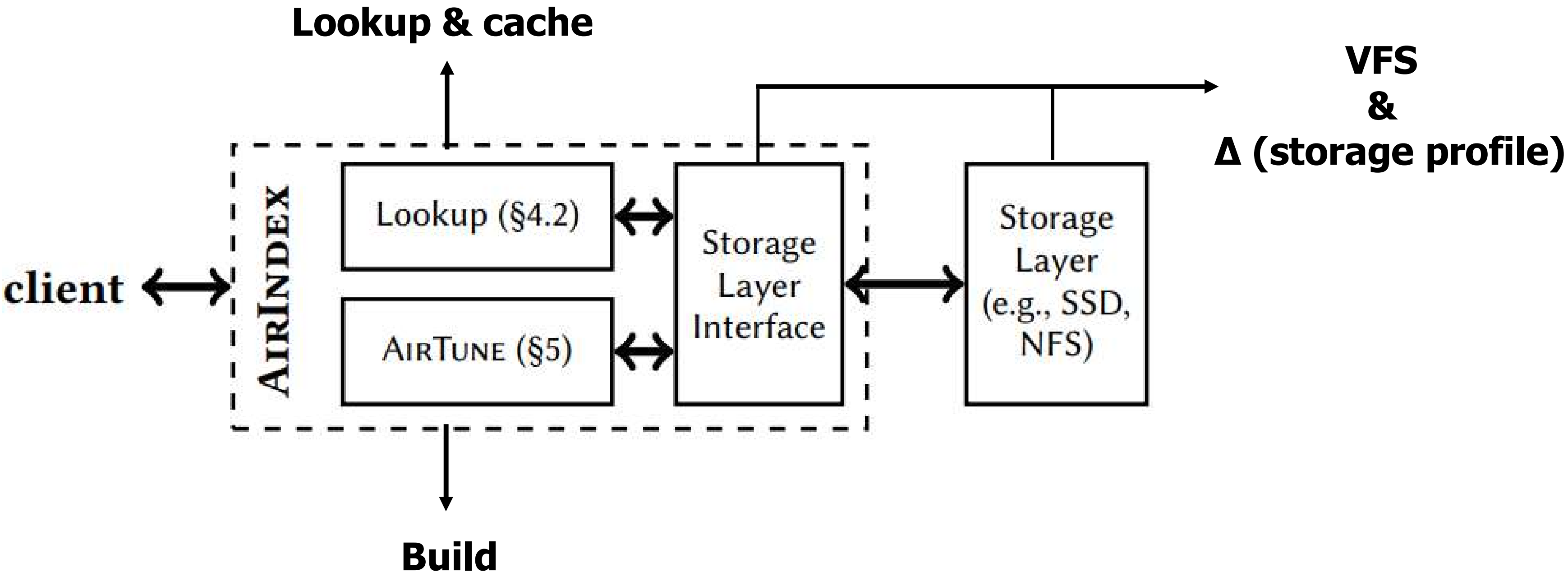


Index needs to be optimized layer by layer based on data distribution

Contents

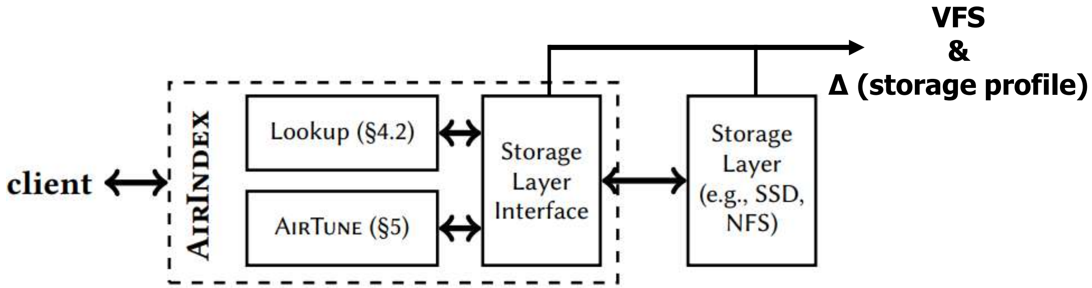
AirIndex Overview

- Architecture



AirIndex Overview

- Storage Model



$$T_{aff}(\Delta) = l + \frac{\Delta}{B}$$

l **:** latency ; **B** **:** bandwidth ;
Δ **:** Storage layer reads

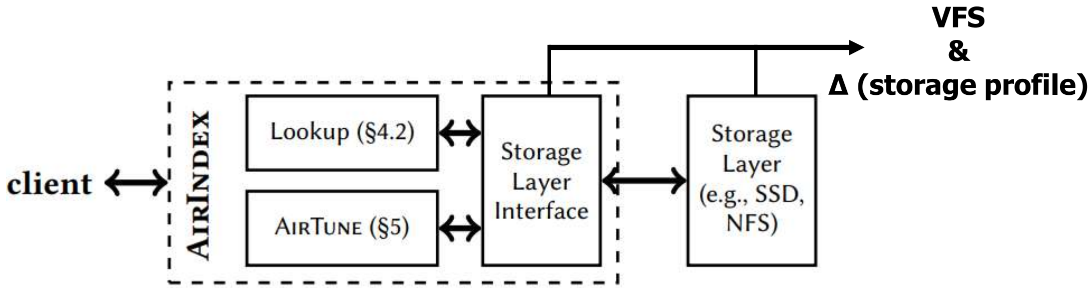
Using latency to represent **storage profile**

$$T_{\text{aff-uniform}}(\Delta) = \frac{\ell_1 + \ell_0}{2} + \Delta \underline{\frac{\ln B_1 - \ln B_0}{B_1 - B_0}}$$

The calculation of the expected value of **1/B** involves logarithms

AirIndex Overview

- Storage Model



$$T_{aff}(\Delta) = l + \frac{\Delta}{B}$$

l : latency ; **B** : bandwidth ;
Δ : Storage layer reads

Using latency to represent **storage profile**

$$T_{\text{aff-uniform}}(\Delta) = \frac{\ell_1 + \ell_0}{2} + \Delta \frac{\ln B_1 - \ln B_0}{B_1 - B_0}$$

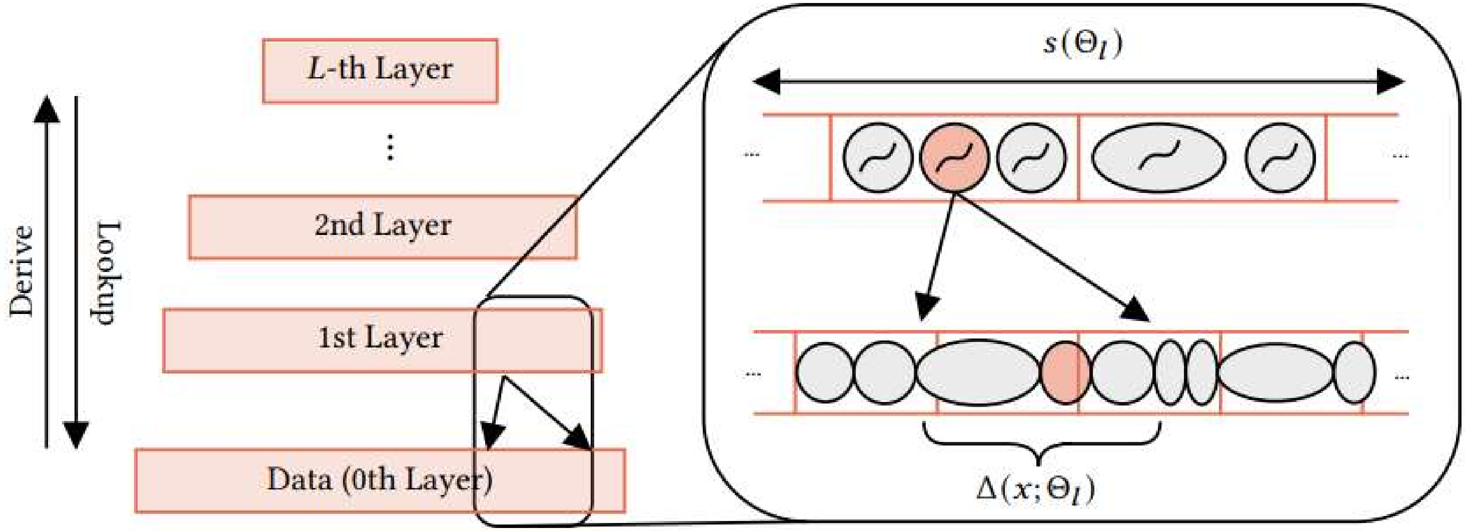
The calculation of the expected value of **1/B** involves logarithms

It is more convenient for AirIndex to self tune without considering too many low relational variables

Contents

AirIndex-Model

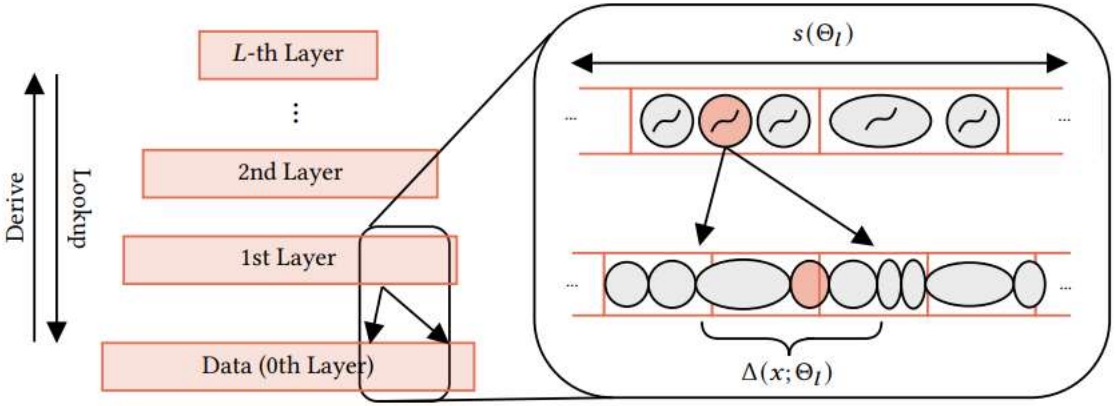
- Hierarchical Indexes



Multi layer & Single layer multiple nodes with multi type nodes

AirIndex-Model

- Query Process



Algorithm 1: AirIndex Query Process, $\text{Lookup}(x;\ \hat{y}_L, L)$

Input: Query key x, root position $\hat{y}_L$, number of layers L
Output: Relevant key-value (x, v)

for l *from* L *to* 0 **do**
    $\{(x[i], v[i])\}_i \leftarrow \text{Read}(\hat{y}_l(x))$ // Storage access
    $v_l \leftarrow \text{Search}(x, \{(x[i], v[i])\}_i)$
    if *In index layer,* $l \geq 1$ **then**
        $\hat{y}_{l-1} \leftarrow \text{ReconstructNode}(v_l)$
return $(x, v) = (x, v_0)$

AirIndex-Model

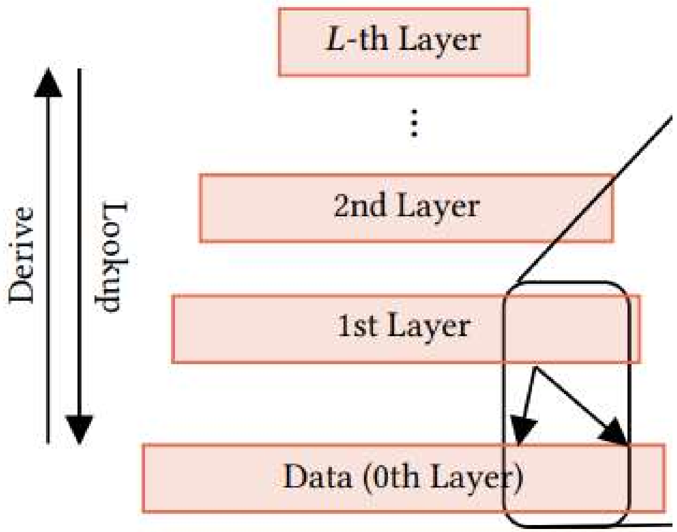
- Latency Under Storage Model

Design variables Θ:

L	Number of layers
$\mathtt{NodeType}_l$	Node type in layer $l \in \{1, \ldots, L\}$
n_l	Number of nodes in layer $l \in \{1, \ldots, L\}$
$\theta_{l,i}$	Parameters of the i-th node in layer l, $i \in \{1, \ldots, n_l\}$

Fixed variables:

T	Storage profile



$$\mathcal{L}_{SM}(x; \Theta, T) = T(s(\Theta_L)) + \sum_{l=1}^{L} T(\Delta(x; \Theta_l))$$

AirIndex-Model

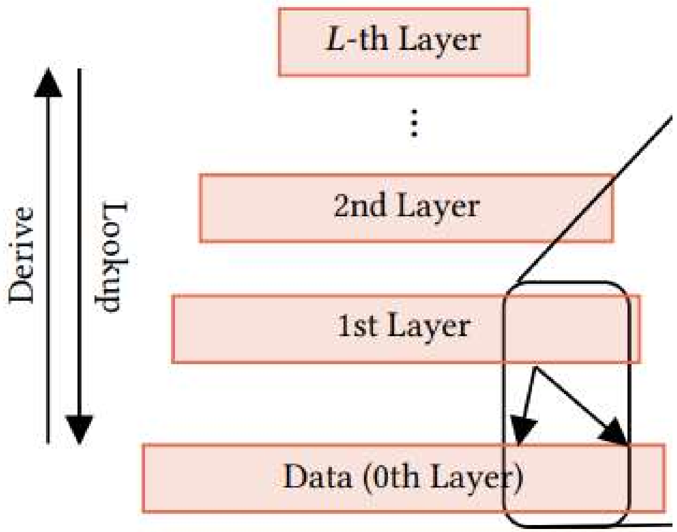
- Latency Under Storage Model

Design variables Θ:

L	Number of layers
$\mathsf{NodeType}_l$	Node type in layer $l \in \{1, \ldots, L\}$
n_l	Number of nodes in layer $l \in \{1, \ldots, L\}$
$\theta_{l,i}$	Parameters of the i-th node in layer l, $i \in \{1, \ldots, n_l\}$

Fixed variables:

T	Storage profile
$\mathcal{X}$	Query key distribution
D	Key-position collection $D = \{(x_i, y_i)\}_{i=1}^{n}$



$$\mathcal{L}_{SM}(x; \Theta, T) = \mathop{\mathrm{E}}_{x \sim \mathcal{X}} \left[T(s(\Theta_L)) + \sum_{l=1}^{L} T(\Delta(x; \Theta_l)) \right]$$

AirIndex-Model

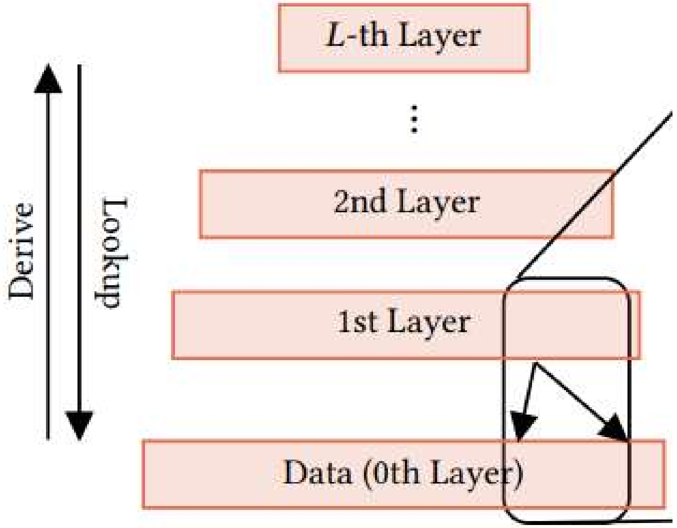
- Latency Under Storage Model

Design variables Θ:

L	Number of layers
$\mathsf{NodeType}_l$	Node type in layer $l \in \{1, \ldots, L\}$
n_l	Number of nodes in layer $l \in \{1, \ldots, L\}$
$\theta_{l,i}$	Parameters of the i-th node in layer l, $i \in \{1, \ldots, n_l\}$

Fixed variables:

T	Storage profile
$\mathcal{X}$	Query key distribution
D	Key-position collection $D = \{(x_i, y_i)\}_{i=1}^{n}$

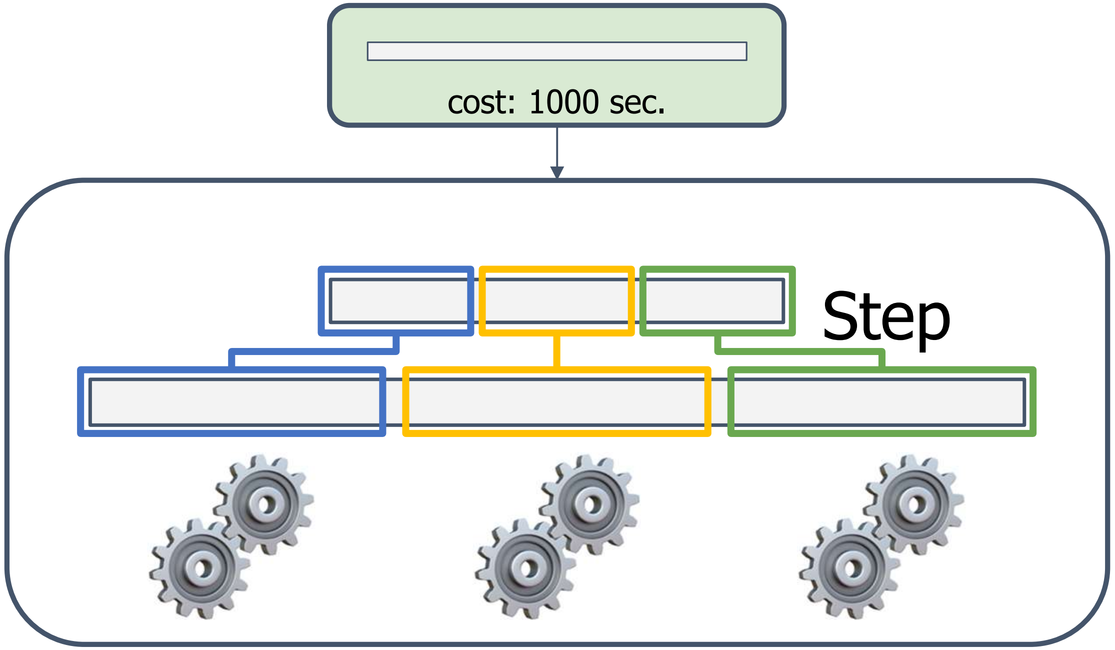


$$\Theta^* = \arg\min_{\Theta} \mathop{\mathbb{E}}_{x \sim \mathcal{X}} \left[T(s(\Theta_L)) + \sum_{l=1}^{L} T(\Delta(x; \Theta_l)) \right]$$

Contents

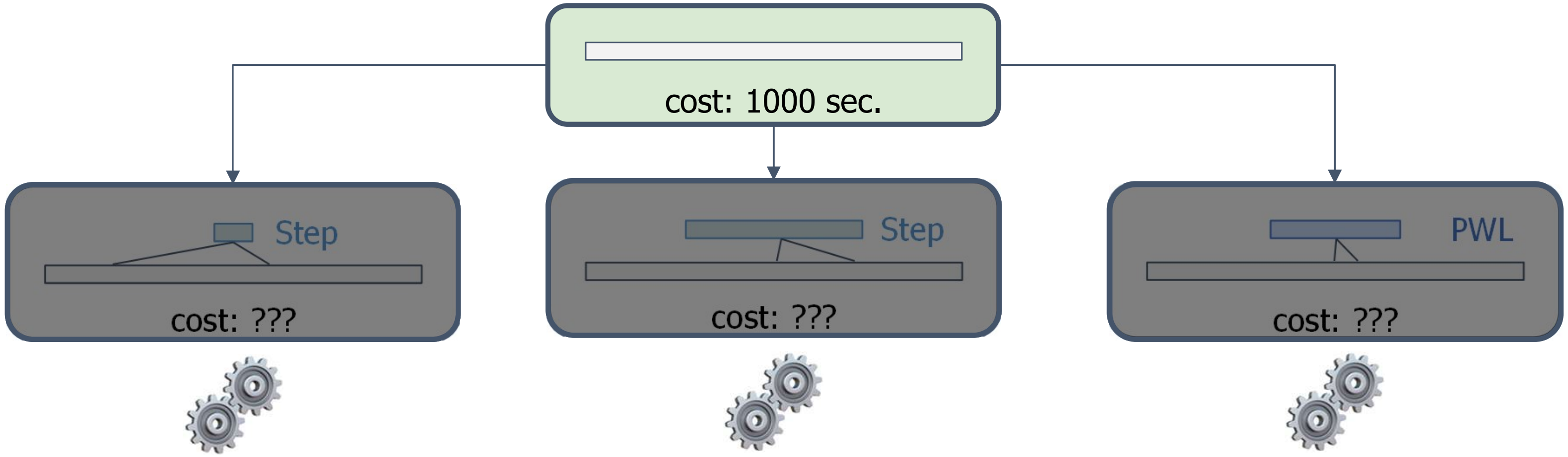
AirTune

- Guided Graph Search



AirTune

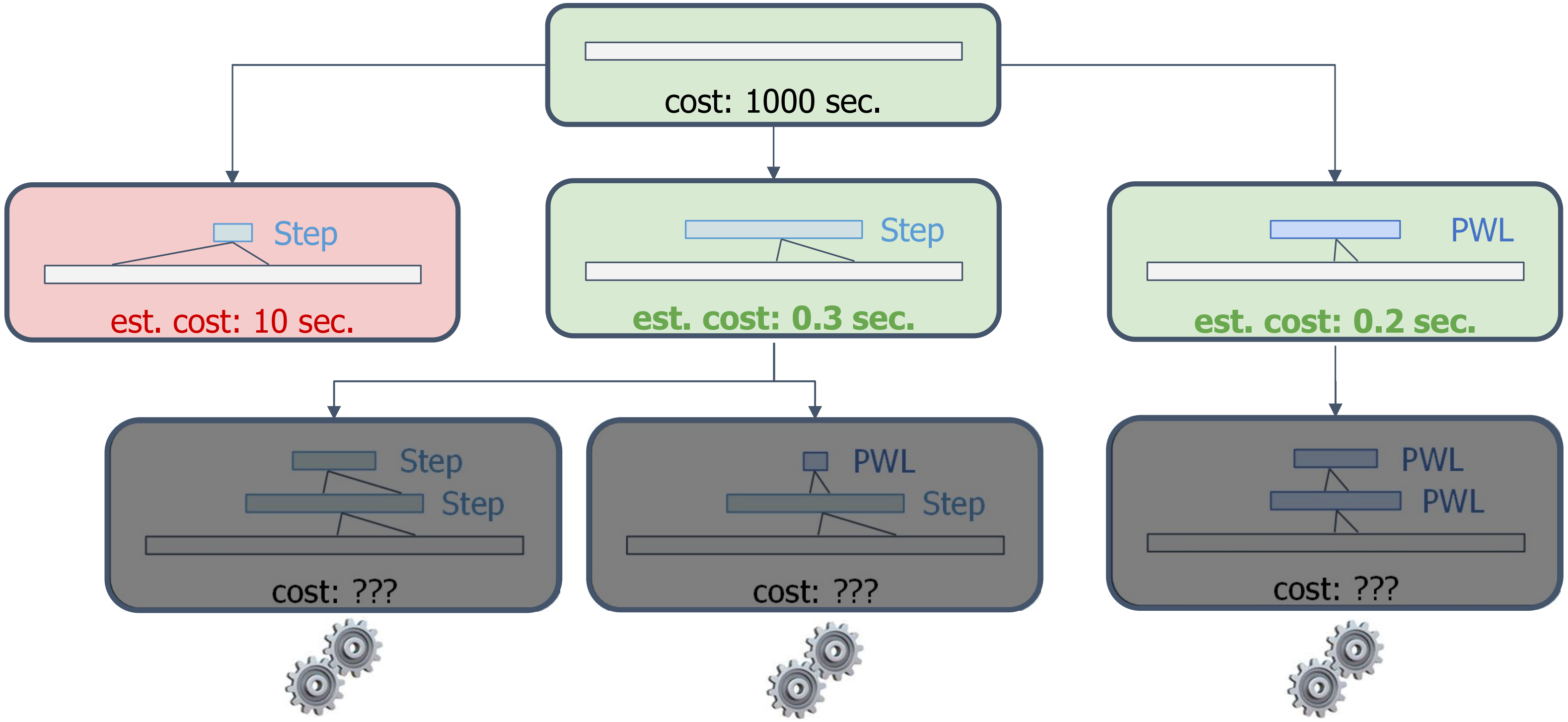
- Guided Graph Search



$$\Theta^* = \arg\min_{\Theta} \mathop{\mathrm{E}}_{x \sim \mathcal{X}} \left[T(s(\Theta_L)) + \sum_{l=1}^{L} T(\Delta(x; \Theta_l)) \right]$$

AirTune

- Guided Graph Search



Contents

Experiment

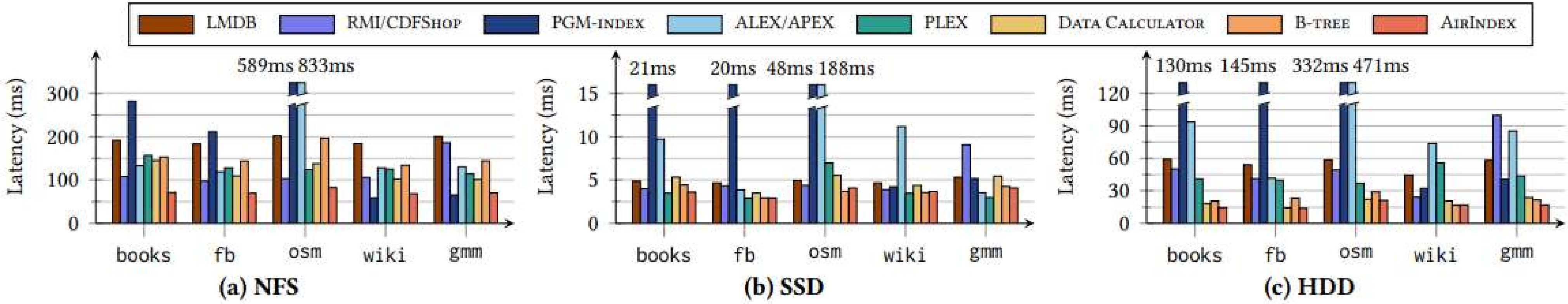
- Setup
  - System Environment :
    - Azure cloud platform (8 vCPUs, 32 GiB RAM)
      - NFS: Azure network file system
      - SSD: Azure Premium SSD (256 GiB, 2300 IOPS, 150 MBps, read/write host caching).
      - HDD: Azure Standard HDD (1024 GiB, 500 IOPS, 60 MBps, no host caching)
  - Baselines :
    - LMDB: B-tree database
    - RMI, PGM, ALEX/APEX, PLEX: Learned indexes
    - Data Calculator: Index tuner
    - B-TREE: AirIndex's tuned B-Tree

Experiment

- Setup
  - Dataset
    - BOOKS(800M)
    - FB(200M)
    - OSM(800M)
    - WIKI(200M)
    - GMM(Gaussian mixture model of 100 normal distribution clusters over 800M keys)

Experiment

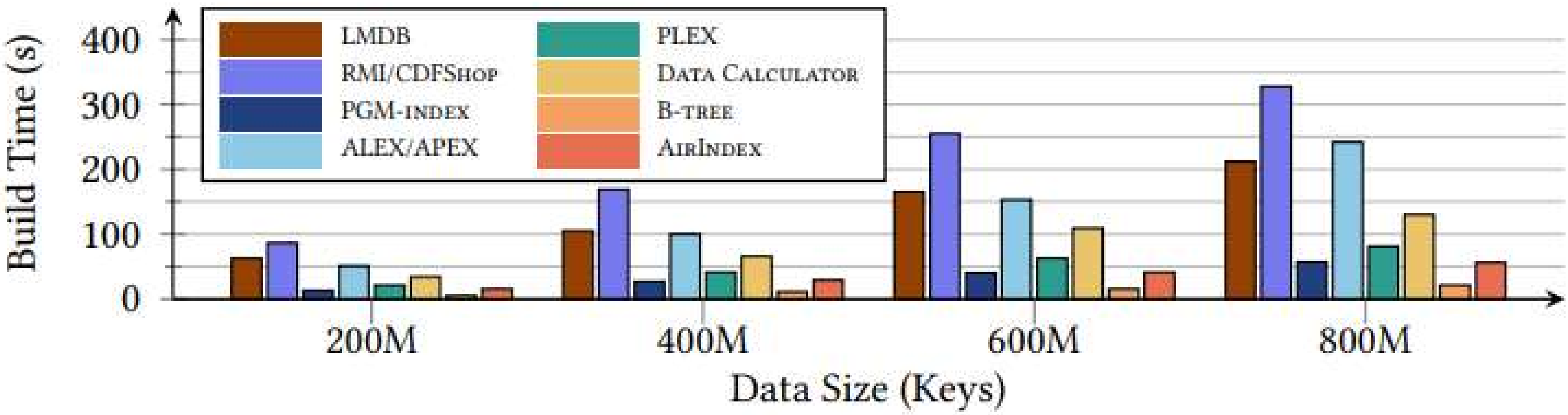
- Faster End-to-end Lookup Speed – Cold-state Latency



AirIndex is consistently one of the fastest methods at searching the first query,
across datasets and storage

Experiment

- Competitive Build Time

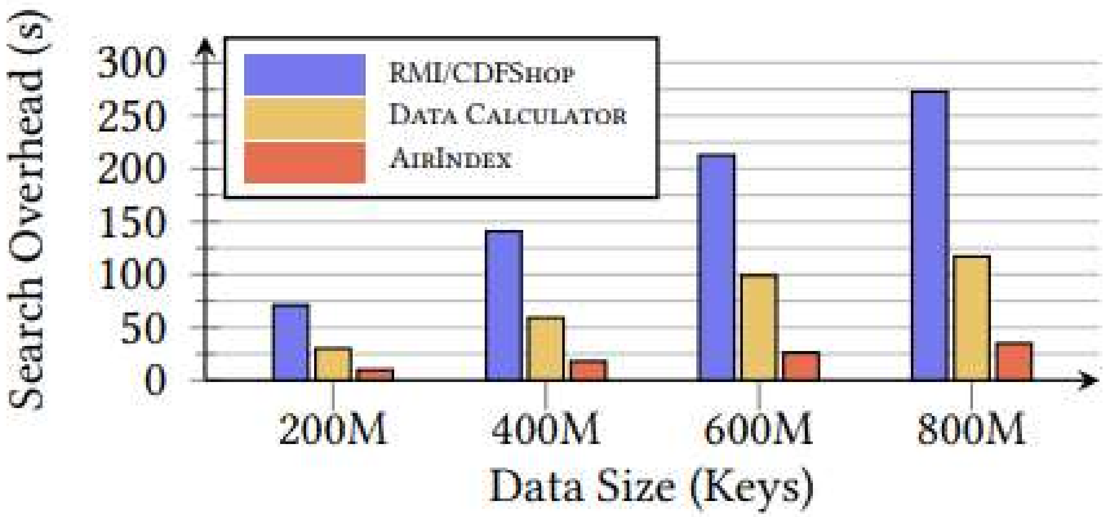


(a) Total Time to Build Index

Despite the default **tuning overhead** of AirIndex's **Airturn**, it still has excellent build time performance

Experiment

- Competitive Build Time



(b) Search Overhead Time

The **tuning cost** of **Airindex** is the **lowest** among existing tuning methods

Contents

Conclusion

- AirIndex

  - AirIndex is the **first** to build high-speed hierarchical indexes by learning data and I/O characteristics
  - AirIndex uses a specially constructed graph search method (AirTune) to explore the search optimization space
  - Compared to indexes that have not undergone specific optimization, AirIndex's data and I/O aware **optimization** can achieve significantly faster search speeds.

Thank you