Spring25 CS598YP

13.2 AirIndex

Yongjoo Park

University of Illinois at Urbana-Champaign

Outline Today

- Learned Index, SIGMOD’18
- AirIndex, SIGMOD’24

Learned Index

Choice of indexes

Storage	Index
Memory	binary search tree
	original learned index (RMI)
Secondary storage (disk)	B+ tree
	AirIndex (SIGMOD'24)

Why do we need indexes?

```
struct KeyValue {
  int key;
  char value[MAX_VALUE_LENGTH];
};

struct KeyValue data[LIST_SIZE]
```

1	2	4	11	25	55	106	172	231	267	286	294	299	302	310	341	440	622	760	820	856	898	955	1025	1099	1166	1216	1250	1271	1284	1291	1295	1297	1298	1299

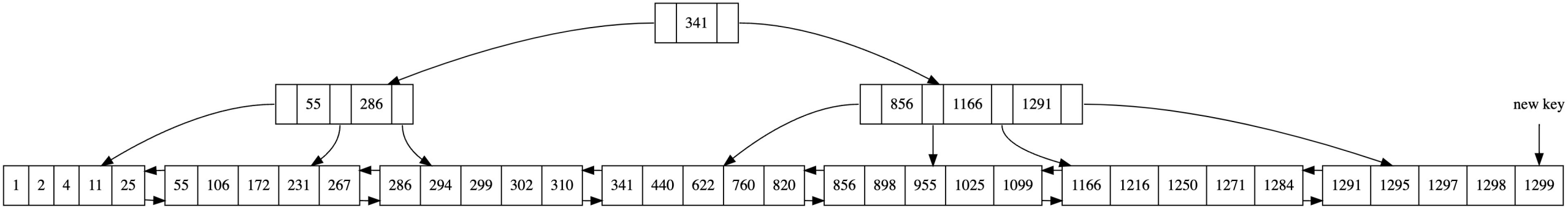
data layer (values are hidden)

What is the (index) position of my key = 172?

B+ tree example (max degree 5)

keys (35 items):

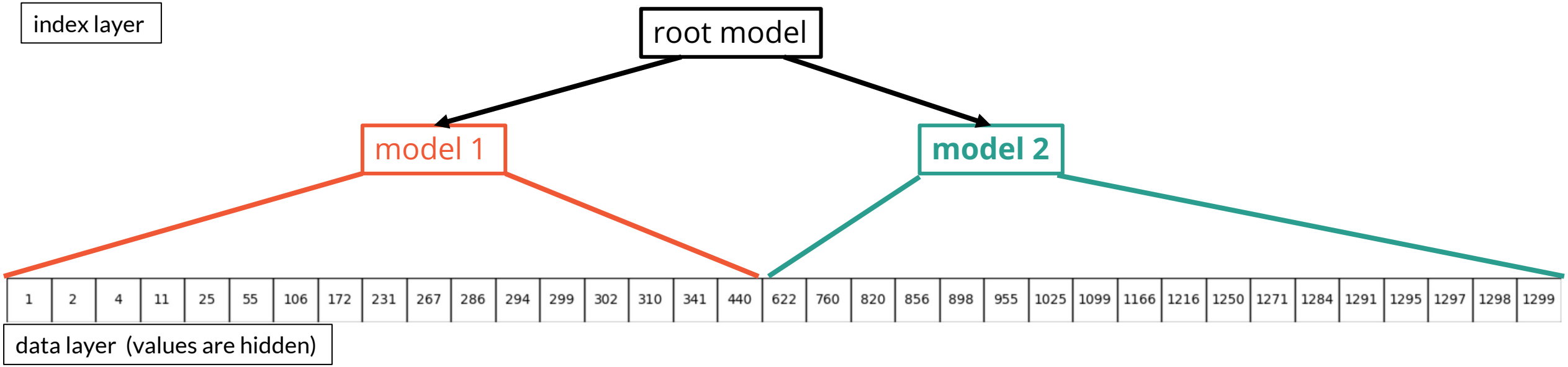
```
1 2 4 11 25 55 106 172 231 267 286 294 299 302 310 341 440 622 760 820 856 898 955 1025 1099
1166 1216 1250 1271 1284 1291 1295 1297 1298 1299
```



Good when we must access data through *pages*

Compact index through *learning*

B+ tree use exact pointers for each “model”

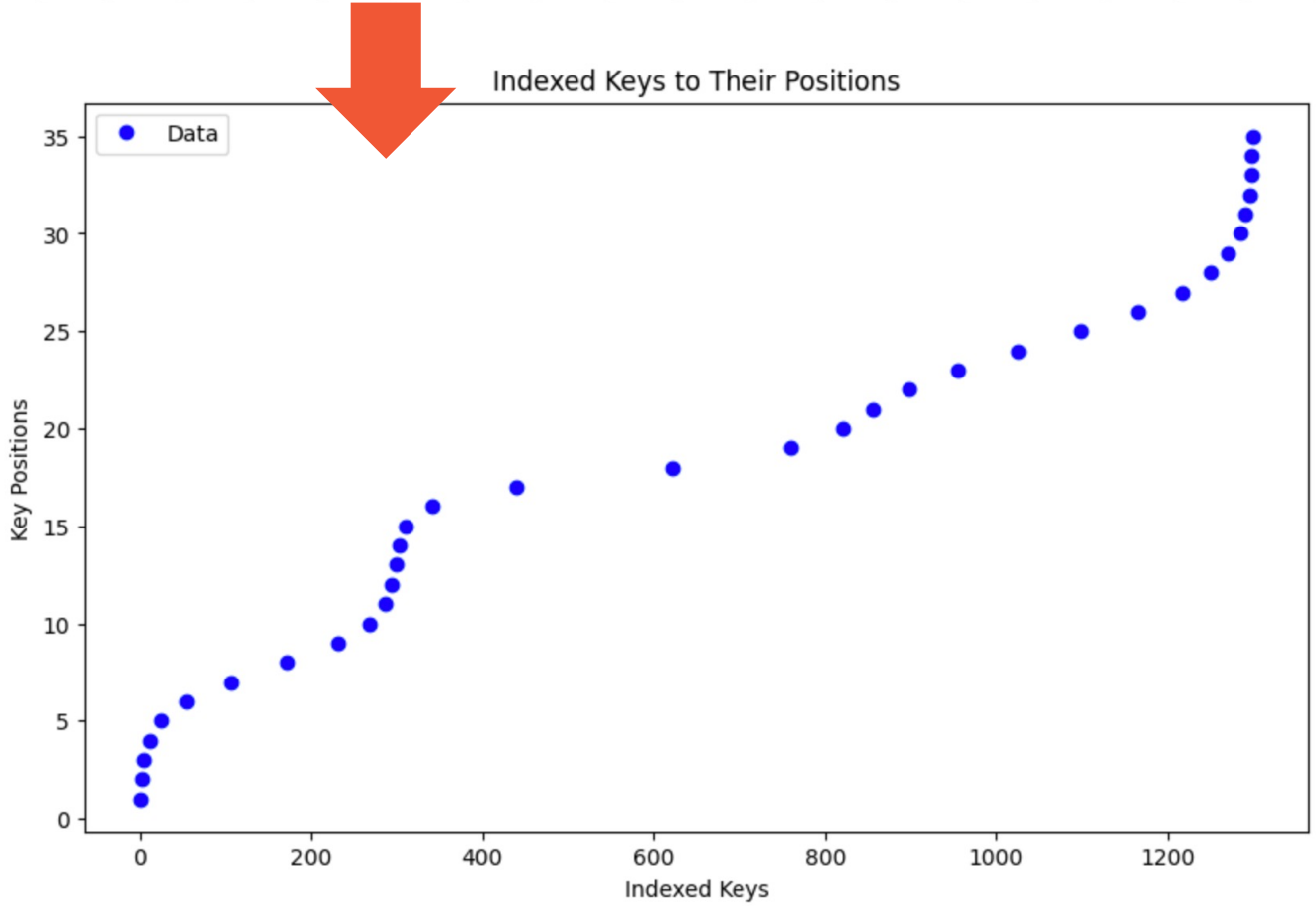


f: key -> pos (integer)

Regression problem

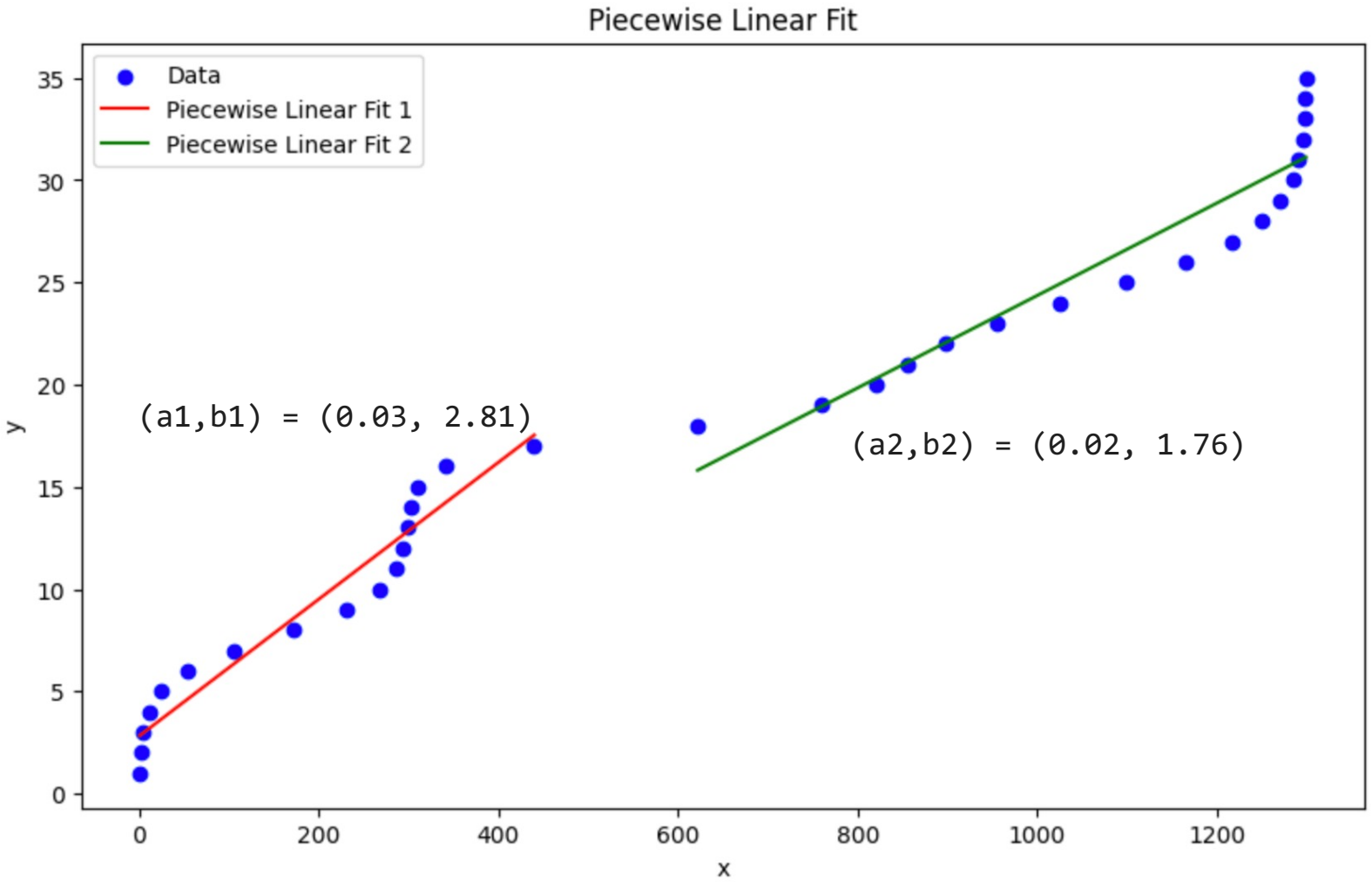
Goal: Fit regression functions

1	2	4	11	25	55	106	172	231	267	286	294	299	302	310	341	440	622	760	820	856	898	955	1025	1099	1166	1216	1250	1271	1284	1291	1295	1297	1298	1299



RMI uses piece-wise linear functions

Two fitted functions



AirIndex

AirIndex: Versatile Index Tuning Through Data and Storage

SUPAWIT CHOCKCHOWWAT, University of Illinois at Urbana-Champaign, USA
WENJIE LIU, University of Illinois at Urbana-Champaign, USA
YONGJOO PARK, University of Illinois at Urbana-Champaign, USA

The end-to-end lookup latency of a hierarchical index—such as a B-tree or a learned index—is determined by its structure such as the number of layers, the kinds of branching functions appearing in each layer, the amount of data we must fetch from layers, etc. Our primary observation is that by optimizing those structural parameters (or *designs*) specifically to a target system's I/O characteristics (e.g., latency, bandwidth), we can offer a faster lookup compared to the ones that are not optimized. Can we develop a systematic method for finding those optimal design parameters? Ideally, the method must have the potential to generate almost any existing index or a novel combination of them for the fastest possible lookup.

In this work, we present a new data-and-I/O-aware index builder (called AirIndex) that can find high-speed hierarchical index designs in a principled way. Specifically, AirIndex minimizes an objective expressing the end-to-end latency in terms of various *designs*—the number of layers, types of layers, and more—for given data and a *storage profile*, using a graph-based optimization method purpose-built to address the computational challenges rising from the inter-dependencies among index layers and the exponentially many candidate parameters in a large search space. Our evaluations confirm that AirIndex can find optimal index designs, build them within the times comparable to existing methods, and deliver up to 4.1× faster lookup than a lightweight B-tree library (LMDB), 3.3×–46.3× faster than state-of-the-art learned indexes (RMI/CDFShop, PGM-index, ALEX/APEX, PLEX), and 2.0× faster than Data Calculator's suggestion on various dataset and storage settings.

CCS Concepts: • **Information systems → Data structures**; **Autonomous database administration**.

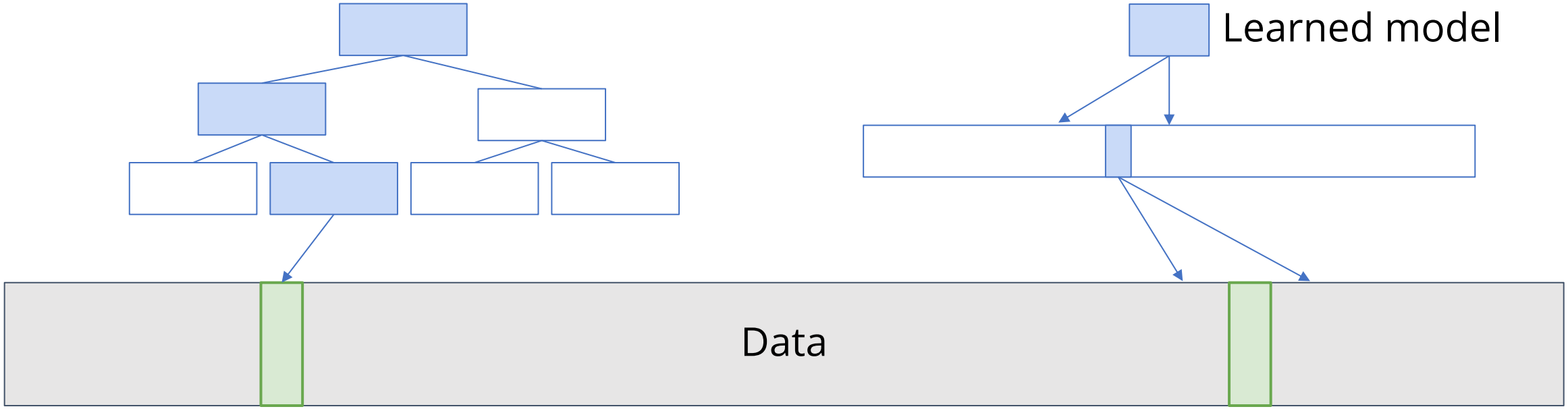
Additional Key Words and Phrases: tuning, indexing, learned indexes, hierarchical indexes, storage, external memory, constrained optimization, instance optimization, graph search, index complexity, parallel tuning

ACM Reference Format:
Supawit Chockchowwat, Wenjie Liu, and Yongjoo Park. 2023. AirIndex: Versatile Index Tuning Through Data and Storage. *Proc. ACM Manag. Data* 1, 3 (SIGMOD), Article 204 (September 2023), 26 pages. https://doi.org/10.1145/3617308

Indexes accelerates data retrieval.

Traditional Indexes
B-Tree [1,2], Skiplist [3], Bloom filter [4]

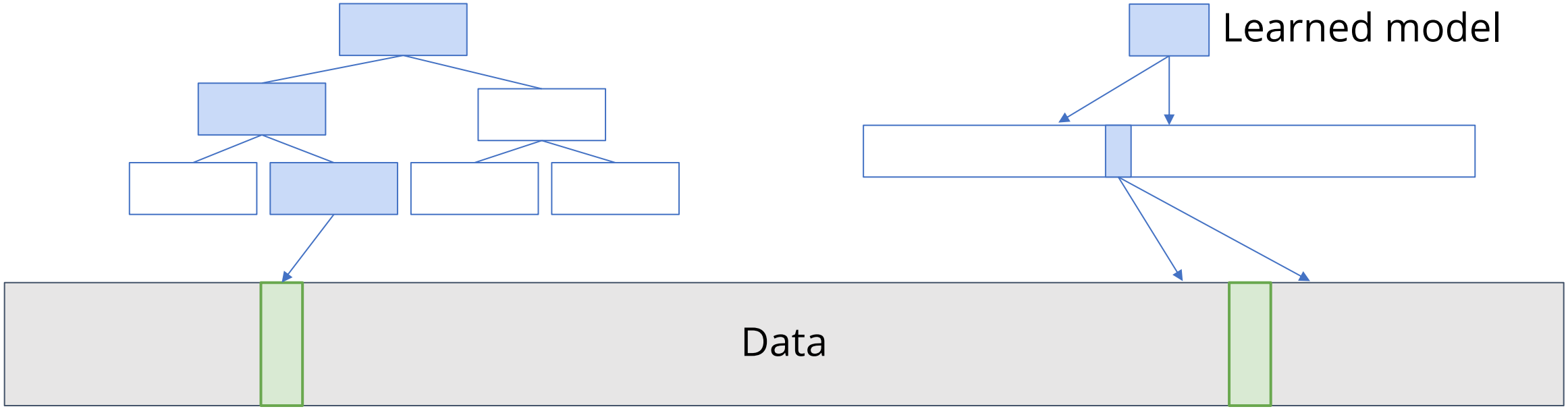
Learned Indexes
RMI [5], PGM-index [6], ALEX [7]



Indexes accelerates data retrieval… however, they are often suboptimal.

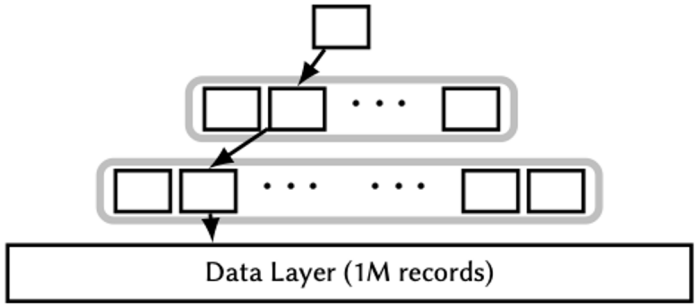
Traditional Indexes
B-Tree [1,2], Skiplist [3], Bloom filter [4]

Learned Indexes
RMI [5], PGM-index [6], ALEX [7]

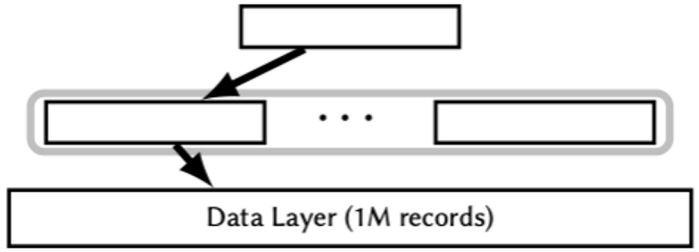


However, these indexes **need to be tuned**.

Need for **I/O-aware index tuning**.



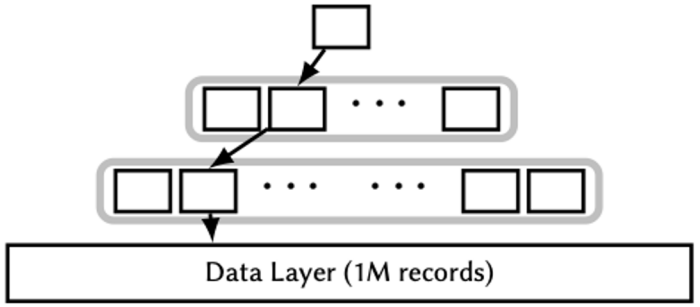
B200: B-tree with 200 child pointers



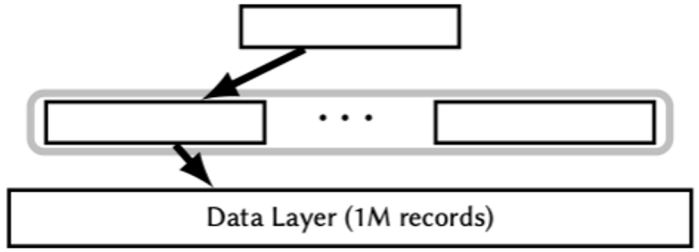
B5000: B-tree with 5000 child pointers

Which B-tree is **faster**?

Need for **I/O-aware index tuning**.



B200: B-tree with 4KB page size



B5000: B-tree with 100KB page size

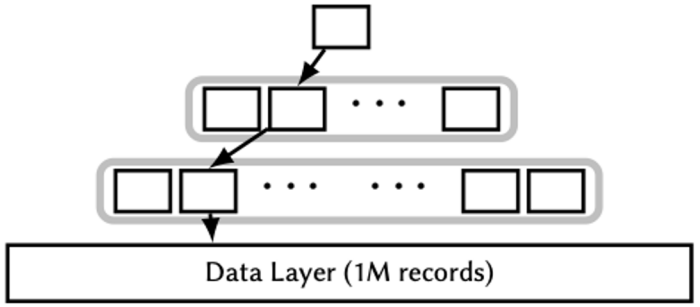
Which B-tree is **faster**?

Case 1: data on SSD (100 us latency, 1GB/s bandwidth)

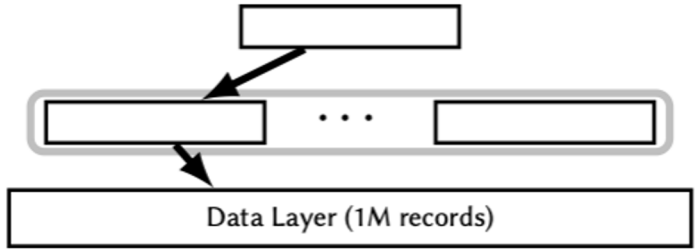
Case 2: data in cloud (100 ms latency, 100MB/s bandwidth)

Data layer uses 4 KB pages

Need for **I/O-aware index tuning**.



B200: B-tree with 4KB page size



B5000: B-tree with 100KB page size

Case 1: data on SSD (100 us latency, 1GB/s bandwidth)

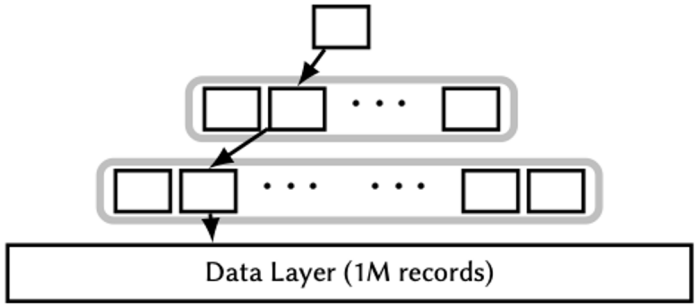
B200: Need to retrieve 3 index pages and 1 data page

(total time) = 3 * (100 us + 4 KB/(1GB/s)) + (100 us + 4 KB/(1GB/s)) = 416 us

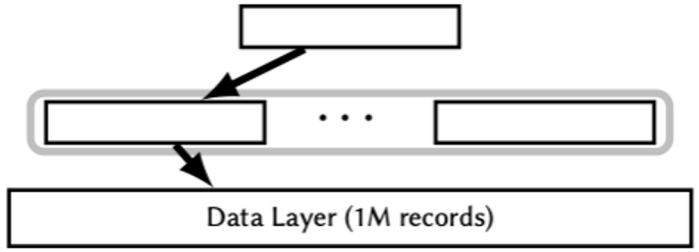
B5000: Need to retrieve 2 index pages and 1 data page

(total time) = 2 * (100 us + 100 KB/(1GB/s)) + (100 us + 100 KB/(1GB/s)) = 504 us

Need for **I/O-aware index tuning**.



B200: B-tree with 4KB page size



B5000: B-tree with 100KB page size

Case 2: data in Cloud (100 ms latency, 100MB/s bandwidth)

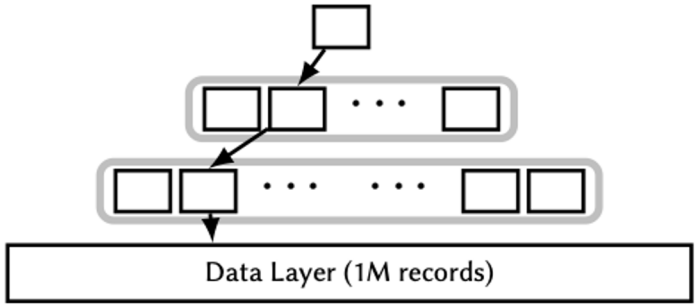
B5000: Need to retrieve 3 index pages and 1 data page

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

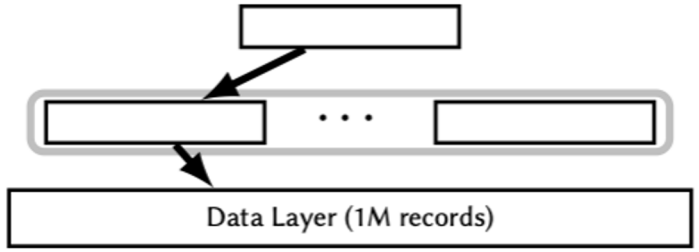
B5000: Need to retrieve 2 index pages and 1 data page

(total time) = 2 * (100 ms + 100 KB/(100MB/s)) + (100 ms + 100 KB/(100MB/s)) = 302.04 ms

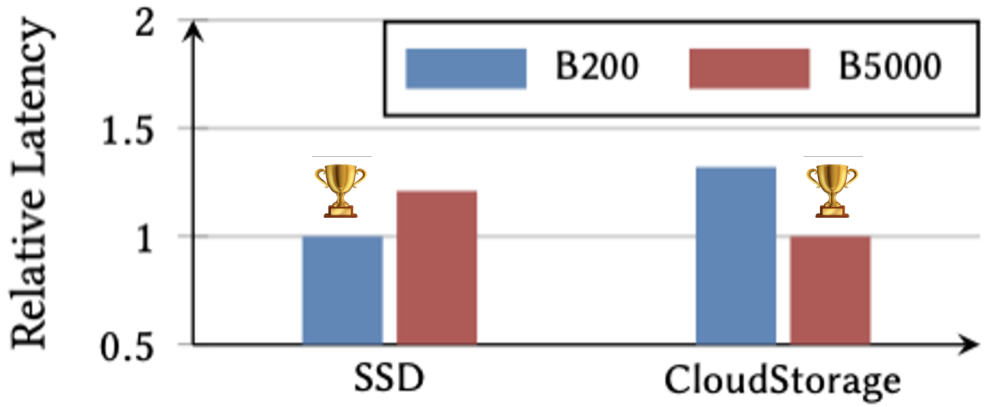
Need for **I/O-aware index tuning**.



B200: B-tree with 200 child pointers

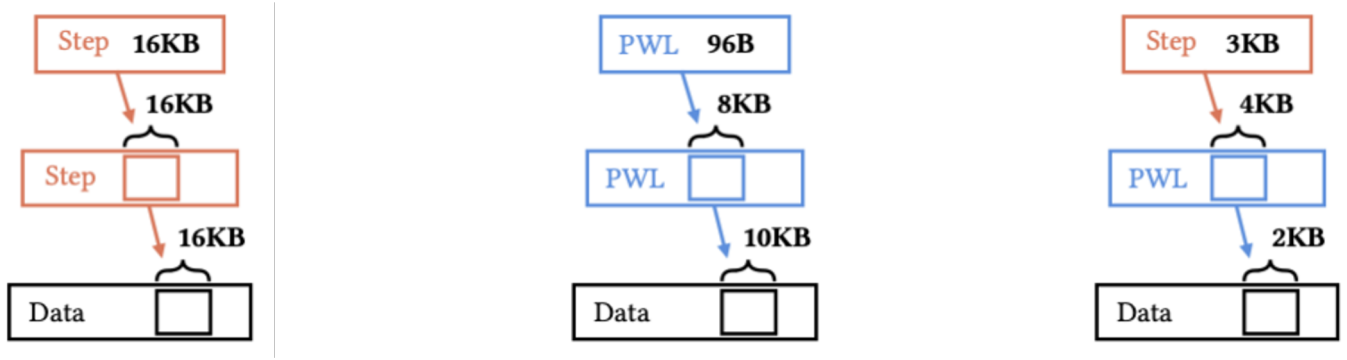


B5000: B-tree with 5000 child pointers

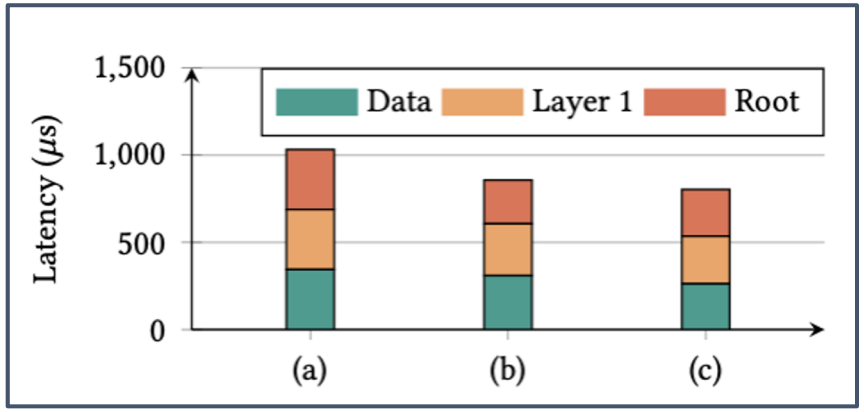


The optimal index depends on storage performance (SSD, CloudStorage).

Need for **layer-wise index tuning**.



(a) Tuned B-tree (b) Tuned PWL (c) Tuned mixed



Layer-wise tuning unlocks **better** solutions.

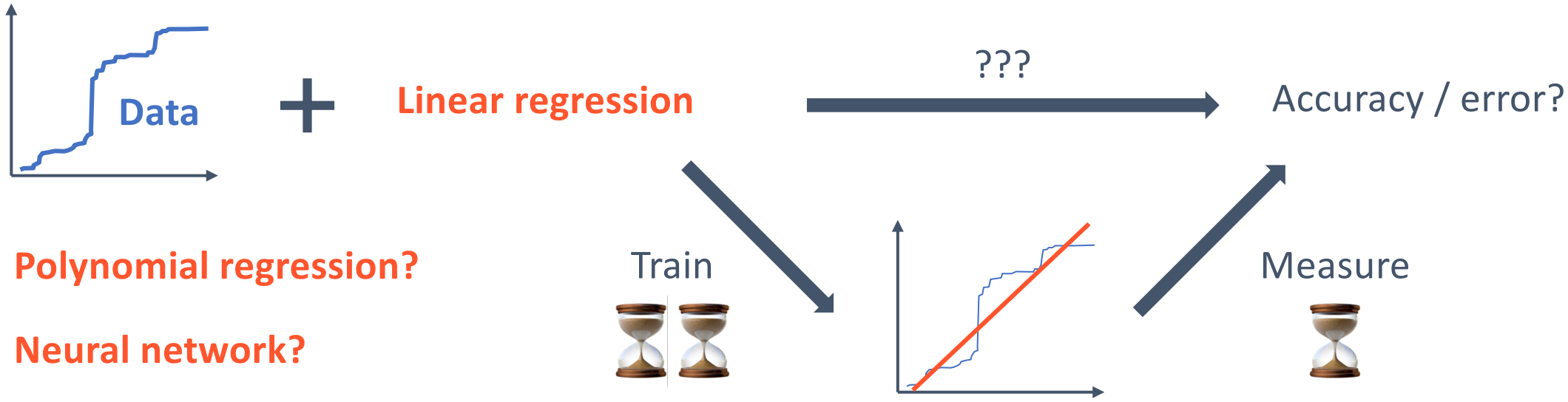
PWL: Piecewise linear model

Need for **learning-aware index tuning**.

Expanded search space: Not just *fanout* and *occupancy ratio*, but including:

Model type, training algorithm, hyperparameters, error correction…

Lack of perf. predictability: Performance is unknown before training.



AirIndex's road to **minimize index's search latency**.

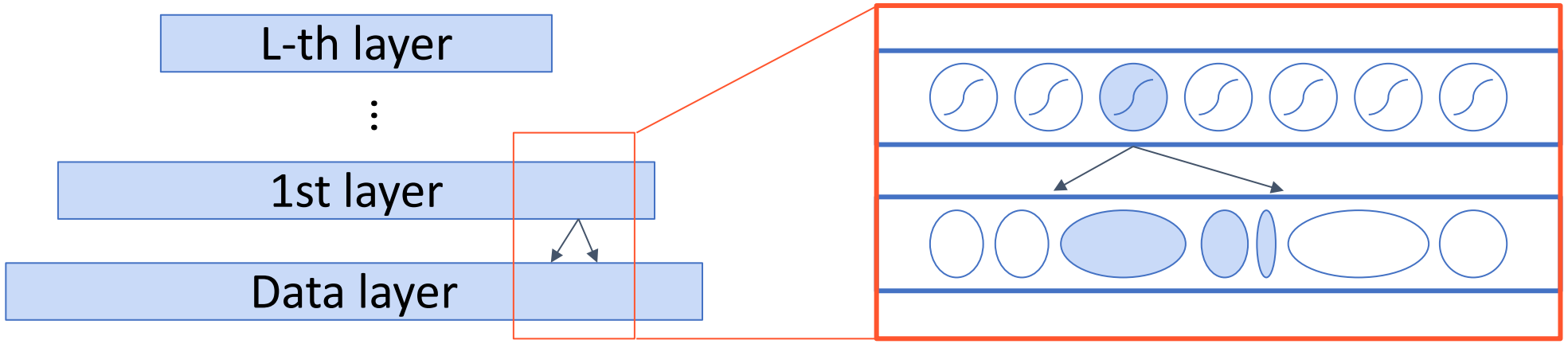
1. Unified index model (Parameters)

2. Cost model based on storage profile (Objective function)

3. Guided graph search (Optimization algorithm)

AirIndex unifies all hierarchical indexes.

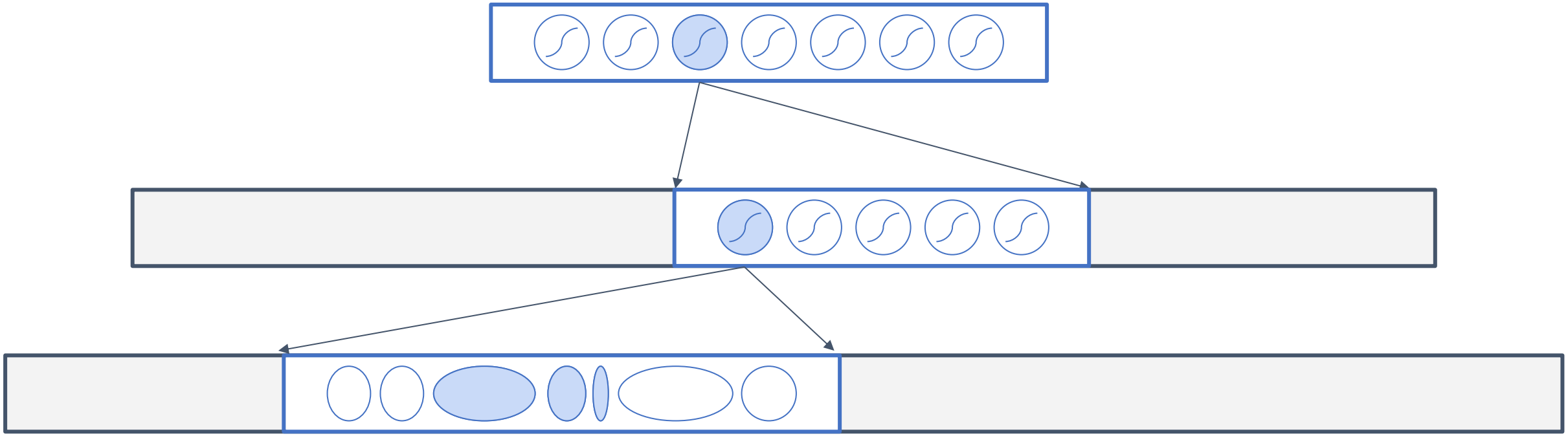
A hierarchical index has L layers, each contains **index node(s)**.



Given a key, an **index node** predicts the **range of positions** in the next layer.

Hierarchical indexes lookup a key by following nodes' direction.

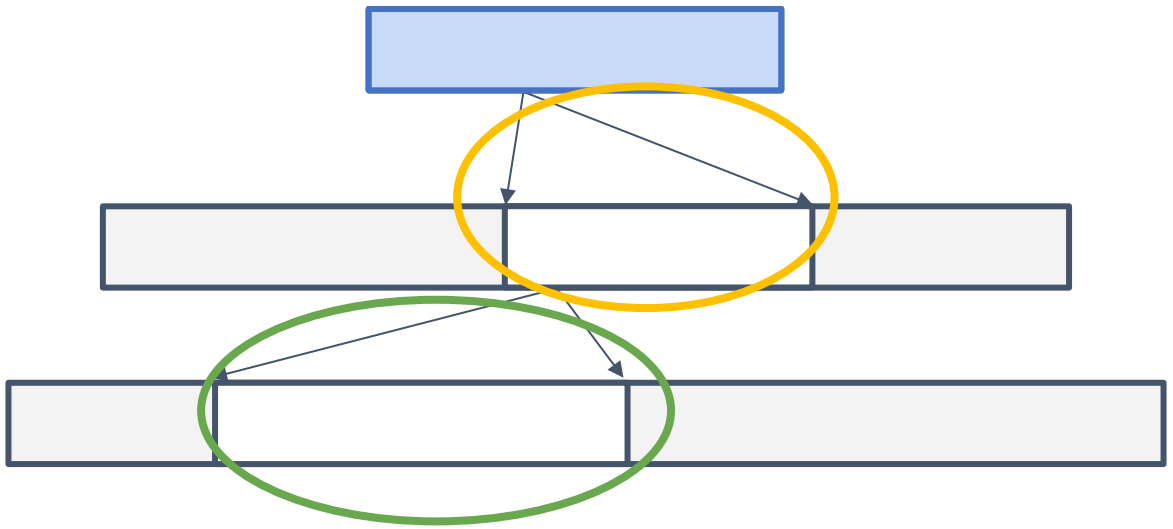
Given a key, an **index node** predicts the **range of positions** in the next layer.



Found!

Hierarchical indexes reveal an I/O-aware cost model.

Let T(N) be the I/O cost to read N bytes from storage:



T(root size)

+ T(root precision)

+ T(next layer precision)

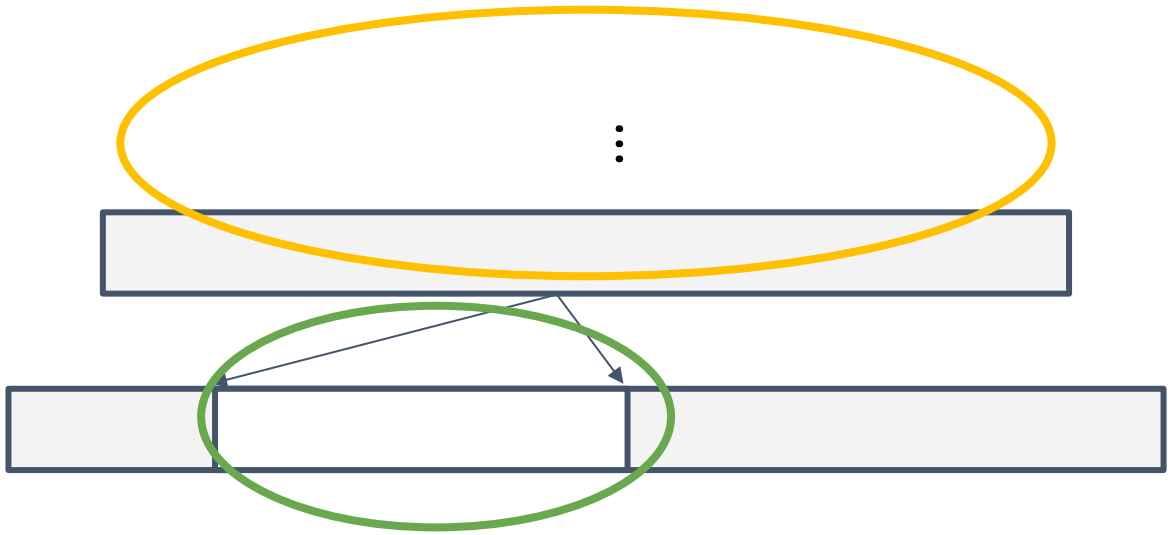
+ …

+ T(last layer precision)

AirIndex minimizes the *expectation* of this cost model.

From cost model, arises the guided graph search.

Key observation: cost model is recursive.



T(root size)

+ T(root precision)

+ T(next layer precision)

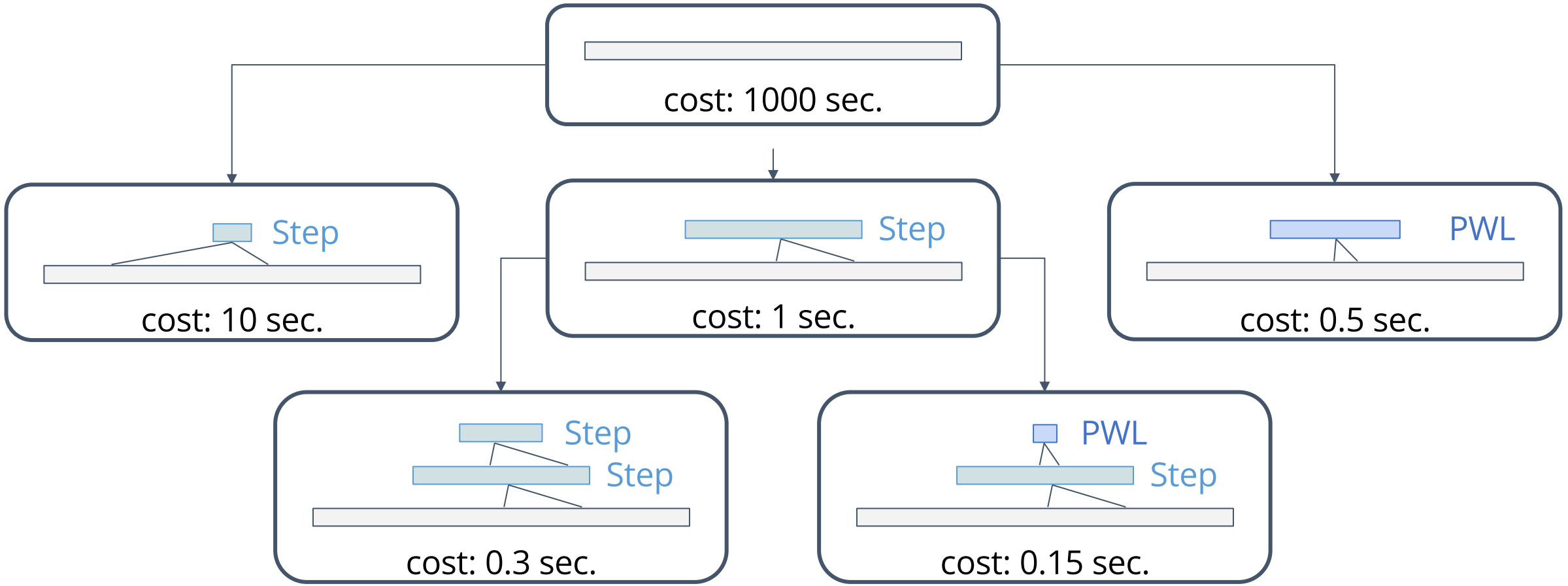
+ …

+ T(last layer precision)

Consider ***an index*** as a ***node***, and ***stacking an index layer*** as an ***edge***.

AirIndex recursively searches the graph by stacking one index layer at a time.

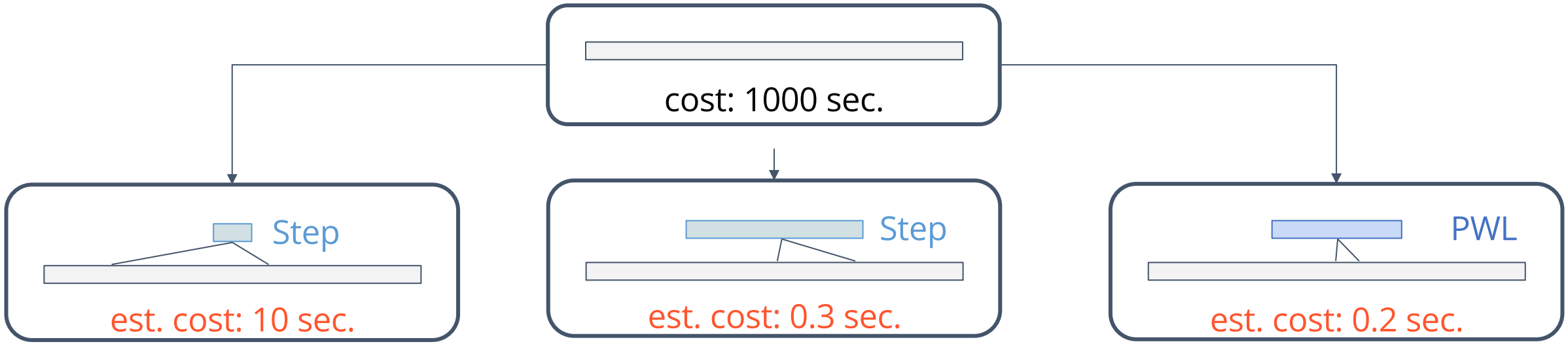
PWL: Piecewise linear model



Too many indexes to traverse!

AirIndex anticipates the **best total cost** by current cost & est. remaining cost.

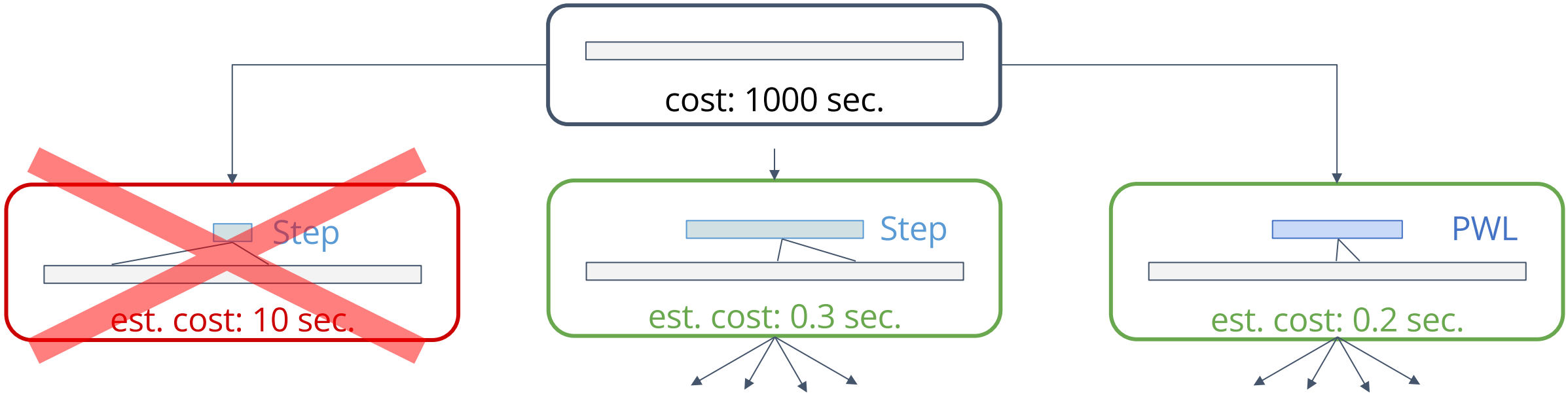
PWL: Piecewise linear model



Calculate estimated remaining cost (**index complexity**)
based on **topmost index layer** and **storage profile**

AirIndex then selects top-k candidates to traverse: **Guided Graph Search**

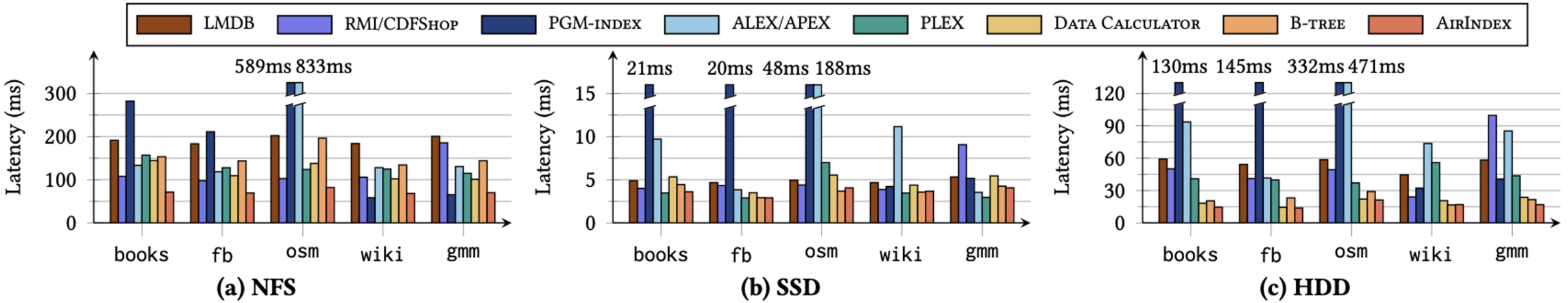
PWL: Piecewise linear model



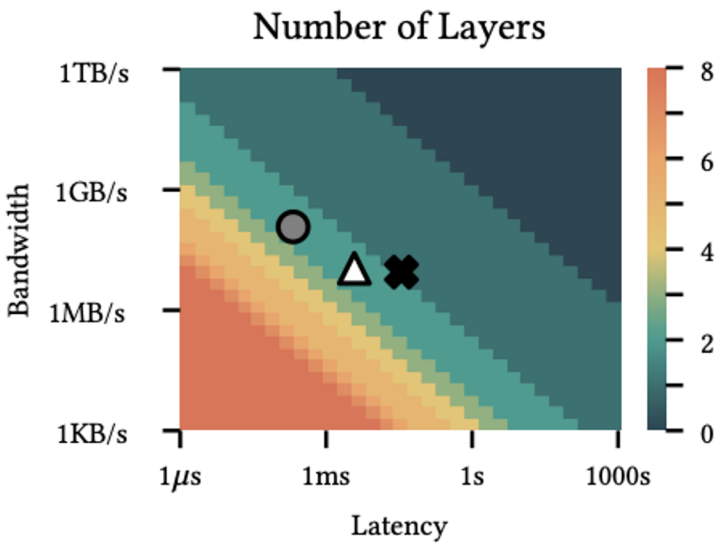
Thanks to this technique, worst case complexity reduces to $O(|\mathcal{F}|(L+1)n)$

Set of index configurations

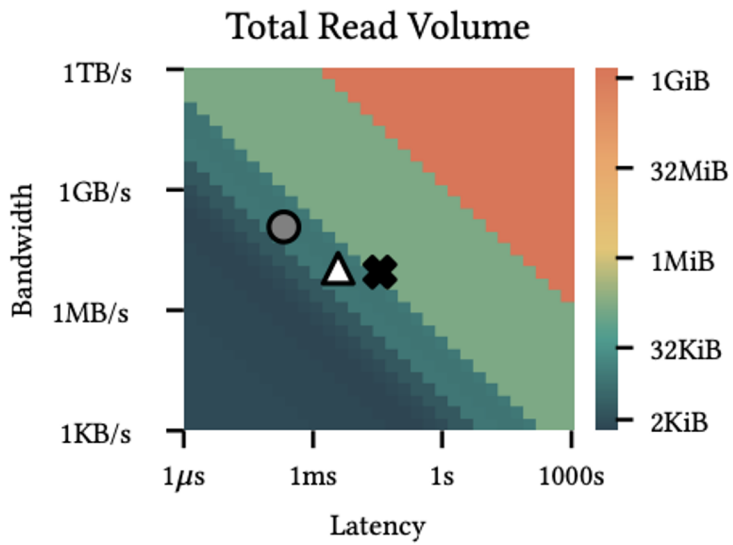
AirIndex finds the fastest indexes across datasets and storages.



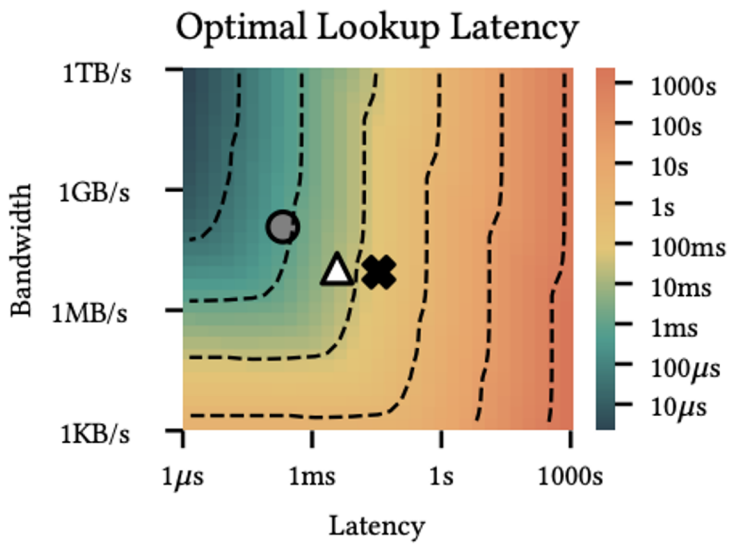
AirIndex adapts to different storage profiles.



(a) Number of layers



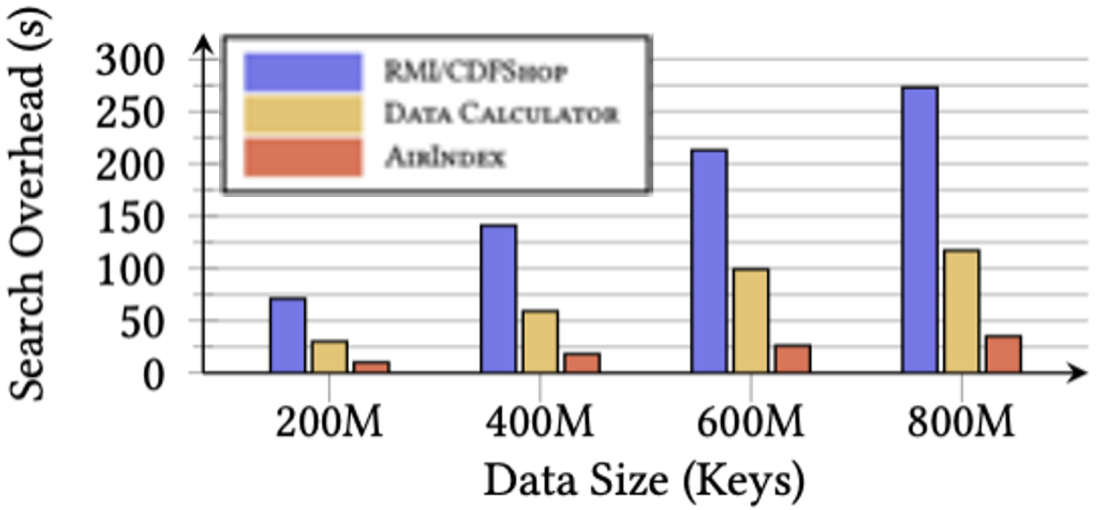
(b) Total Read volume



(c) Optimal cost

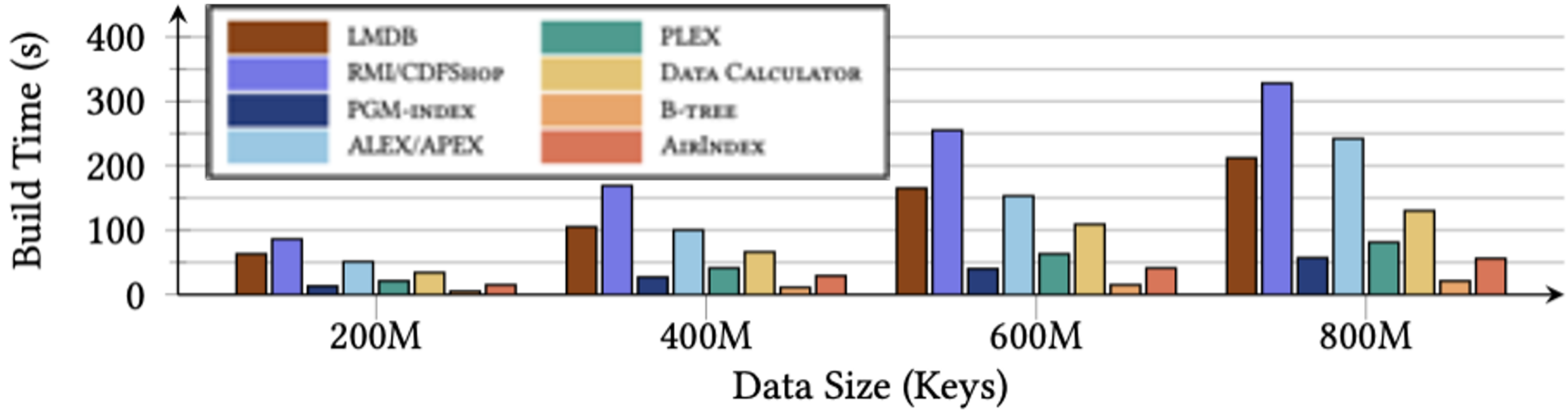
3 storages from benchmark:

As a result, AirIndex simultaneously tunes and builds indexes efficiently.



Faster search for optimal indexes.

Faster index building.



AirIndex: Versatile Index Tuning Through Data and Storage

Index performance needs **learning-aware**, **I/O-aware**, **layer-wise tuning**.

AirIndex unifies **hierarchical indexes**, derives a **storage-aware cost function**, and tunes with a **highly parallelizable guided graph search**.

AirIndex **quickly** discovers the **best index** for each **dataset-storage** setting.

Code repository: https://github.com/illinoisdata/airindex-public

References

[1] R. Bayer and E. M. McCreight. 1972. Organization and maintenance of large ordered indexes. Acta Informatica 1 (1972), 173–189. Issue 3. https://doi.org/10.1007/BF00288683

[2] Douglas Comer. 1979. UBIQUITOUS B-TREE. Comput Surv 11 (1979). Issue 2.

[3] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. Commun. ACM 33, 6 (1990), 668–676.

[4] Bloom, Burton H. 1970, Space/Time Trade-offs in Hash Coding with Allowable Errors, Commun. ACM 13, 7: 422–426.

[5] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. Proceedings of the ACM SIGMOD International Conference on Management of Data. https://doi.org/10.1145/3183713.3196909

[6] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. Proc. VLDB Endow. 13, 8 (2020), 1162–1175.

[7] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. Proceedings of the ACM SIGMOD International Conference on Management of Data. https://doi.org/10.1145/3318464.3389711

[8] Abdullah Al-Mamun, Hao Wu, and Walid G. Aref. 2020. A Tutorial on Learned Multi-dimensional Indexes. In Proceedings of the 28th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '20). Association for Computing Machinery, New York, NY, USA, 1–4. https://doi.org/10.1145/3397536.3426358

References

[9] Jim Gray and Goetz Graefe. 1997. The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. SIGMOD Rec. 26, 4 (1997), 63–68.

[10] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In SIGMOD Conference. ACM, 2789–2792.

[11] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In SIGMOD Conference. ACM, 535–550.

[12] Supawit Chockchowwat, Wenjie Liu, and Yongjoo Park. 2023. AirIndex: Versatile Index Tuning Through Data and Storage. Proc. ACM Manag. Data 1, 3, Article 204 (September 2023), 26 pages. https://doi.org/10.1145/3617308

[13] LMDB.[n.d.]. Lightning Memory-Mapped Database Manager. http://www.lmdb.tech/doc/ Online; accessed Jul-17-2022.

[14] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A High-Performance Learned Index on Persistent Memory. arXiv preprint arXiv:2105.00683 (2021).

[15] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. PLEX: Towards Practical Learned Indexing. CoRR abs/2108.05117 (2021).

Questions?