A Fully On-disk Updatable Learned Index

Hai Lan[1], Zhifeng Bao[1], J. Shane Culpepper[2], Renata Borovica-Gajic[3], Yu Dong[4]

[1] RMIT University, [2] The University of Queensland
[3] The University of Melbourne, [4] PingCAP

Apply Learned Indexes on Disk?

FACT 1

Learned indexes in **main memory** show promising performance in **throughput** and **index size**.

FACT 2

Widely used database systems are still **on disk** due to the large dataset size, index size and so on.

The Case for Learned Index Structures
SIGMOD 2020

Are Updatable Learned Indexes Ready?
VLDB 2022

Apply Learned Indexes on Disk?

FACT 1

Learned indexes in **main memory** show promising performance in **throughput** and **index size**.

FACT 2

Widely used database systems are still **on disk** due to the large dataset size, index size and so on.

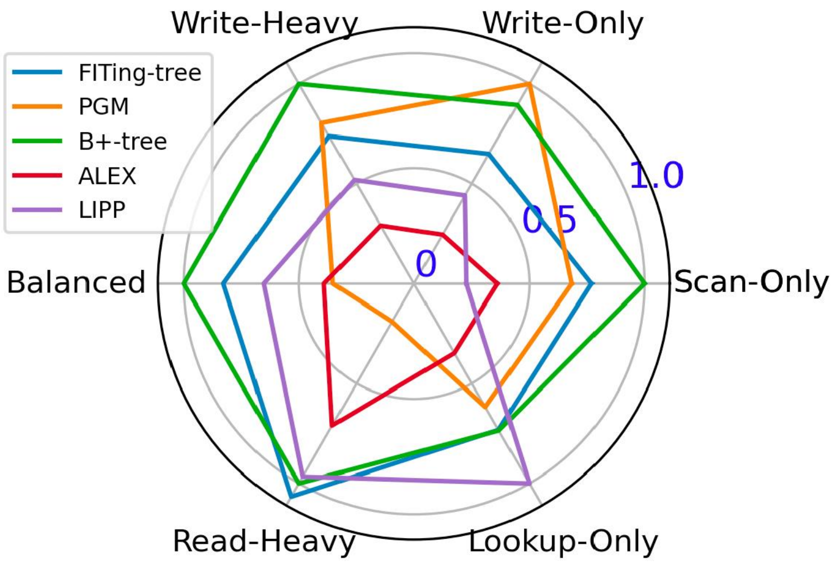
Can we apply learned indexes on the disk setting?

The Case for Learned Index Structures
SIGMOD 2020

Are Updatable Learned Indexes Ready?
VLDB 2022

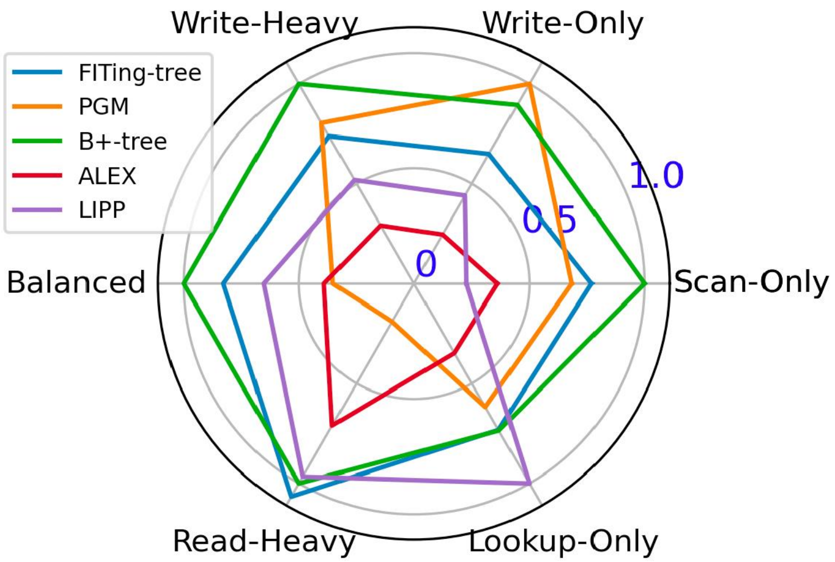
Apply Learned Indexes on Disk?

Normalized throughputs on the FB dataset



Apply Learned Indexes on Disk?

Normalized throughputs on the FB dataset



Overall, **B+-tree** is the **(second-)best**.

LIPP outperforms other indexes on **Lookup-Only** workload.

PGM outperforms other indexes on **Write-Only** workload.

Apply Learned Indexes on Disk?

Apply Learned Indexes on Disk?

#blocks/nodes fetched in Read-Only workload

Lookup

Scan

	# Inner Nodes	# Inner Blocks	# Total Blocks (L)	# Total Blocks (S)
FITing-tree	5	3	4.2	5
PGM	6	3.9	5.2	5.6
ALEX	7.7	6.5	8.1	10.6
LIPP	1.8 (18.8)	-	3	24
B+-tree	4	3	4	4.5

Challenge 1. A learned index cannot guarantee to reduce **I/O costs** when searching data on disk.

Apply Learned Indexes on Disk?

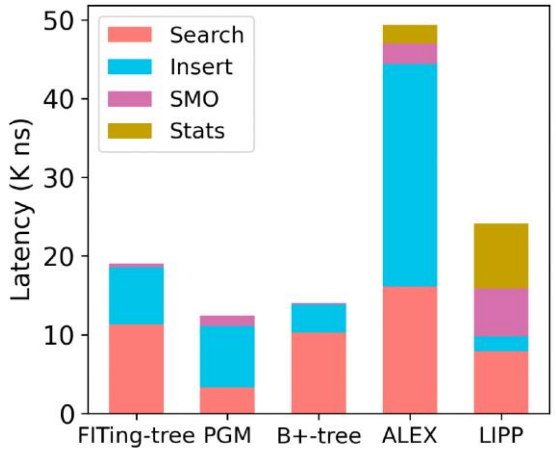
#blocks/nodes fetched in Read-Only workload

Lookup

Scan

	# Inner Nodes	# Inner Blocks	# Total Blocks (L)	# Total Blocks (S)
FITing-tree	5	3	4.2	5
PGM	6	3.9	5.2	5.6
ALEX	7.7	6.5	8.1	10.6
LIPP	1.8 (18.8)	-	3	24
B+-tree	4	3	4	4.5

Latency breakdown in Write-Only workload



Challenge 1. A learned index cannot guarantee to reduce **I/O costs** when searching data on disk.

Challenge 2. Most learned indexes suffer from large **insertion overheads**.

Design Principles

Challenge 1. A learned index cannot guarantee to reduce **I/O costs** when searching data on disk.

Challenge 2. Most learned indexes suffer from large **insertion overheads**.

Design Principles

P1. Reducing the Tree Height of the Index.

Challenge 1. A learned index cannot guarantee to reduce **I/O costs** when searching data on disk.

Challenge 2. Most learned indexes suffer from large **insertion overheads**.

Design Principles

P1. Reducing the Tree Height of the Index.

Challenge 1. A learned index cannot guarantee to reduce **I/O costs** when searching data on disk.

P2. Model-based Operations (Search and Insert).

Challenge 2. Most learned indexes suffer from large **insertion overheads**.

Design Principles

Challenge 1. A learned index cannot guarantee to reduce **I/O costs** when searching data on disk.

Challenge 2. Most learned indexes suffer from large **insertion overheads**.

P1. Reducing the Tree Height of the Index.

P2. Model-based Operations (Search and Insert).

P3. Lightweight Structure Modification Operations (SMO).

Design Principles

Challenge 1. A learned index cannot guarantee to reduce **I/O costs** when searching data on disk.

Challenge 2. Most learned indexes suffer from large **insertion overheads**.

P1. Reducing the Tree Height of the Index.

P2. Model-based Operations (Search and Insert).

P3. Lightweight Structure Modification Operations (SMO).

P4. Better Scan Performance.

Design Principles

Challenge 1. A learned index cannot guarantee to reduce **I/O costs** when searching data on disk.

Challenge 2. Most learned indexes suffer from large **insertion overheads**.

P1. Reducing the Tree Height of the Index.

P2. Model-based Operations (Search and Insert).

P3. Lightweight Structure Modification Operations (SMO).

P4. Better Scan Performance.

P5. Support Duplicate Index Keys.

Design Principles

Challenge 1. A learned index cannot guarantee to reduce **I/O costs** when searching data on disk.

Challenge 2. Most learned indexes suffer from large **insertion overheads**.

P1. Reducing the Tree Height of the Index.

P2. Model-based Operations (Search and Insert).

P3. Lightweight Structure Modification Operations (SMO).

P4. Better Scan Performance.

P5. Support Duplicate Index Keys.

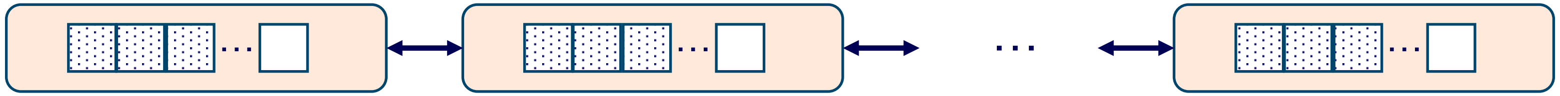
AULID, an updatable learned index on disk

Simple Yet Effective

AULID Index Structure

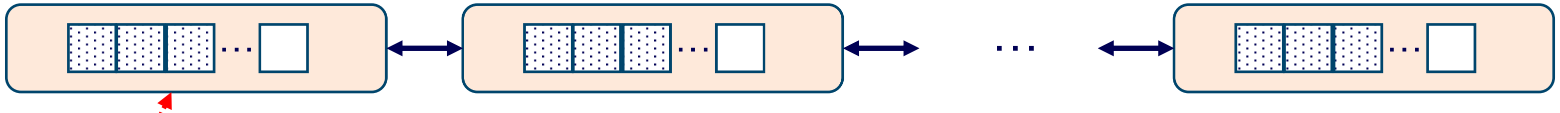
AULID Index Structure

Leaf Node Layer



AULID Index Structure

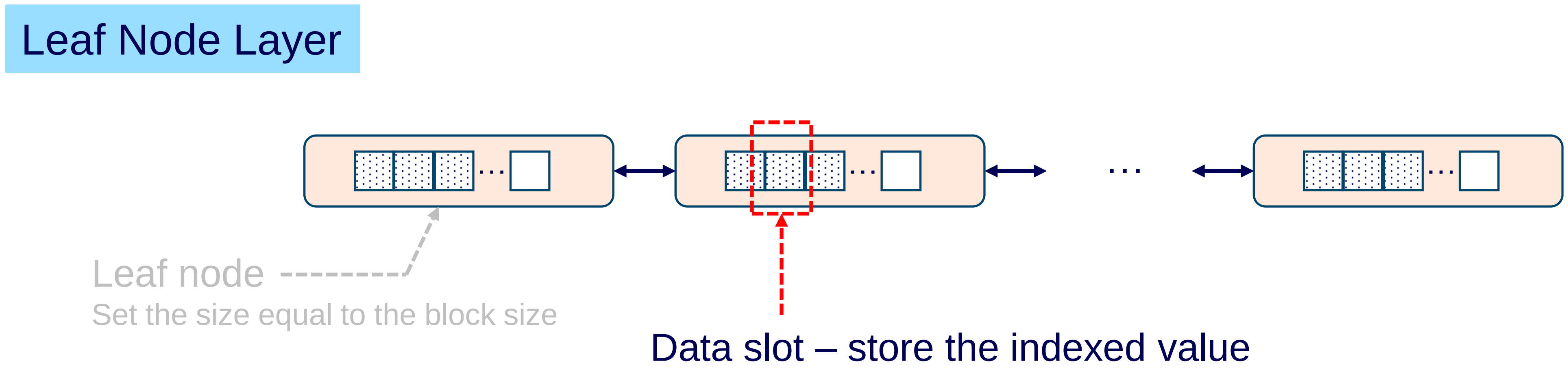
Leaf Node Layer



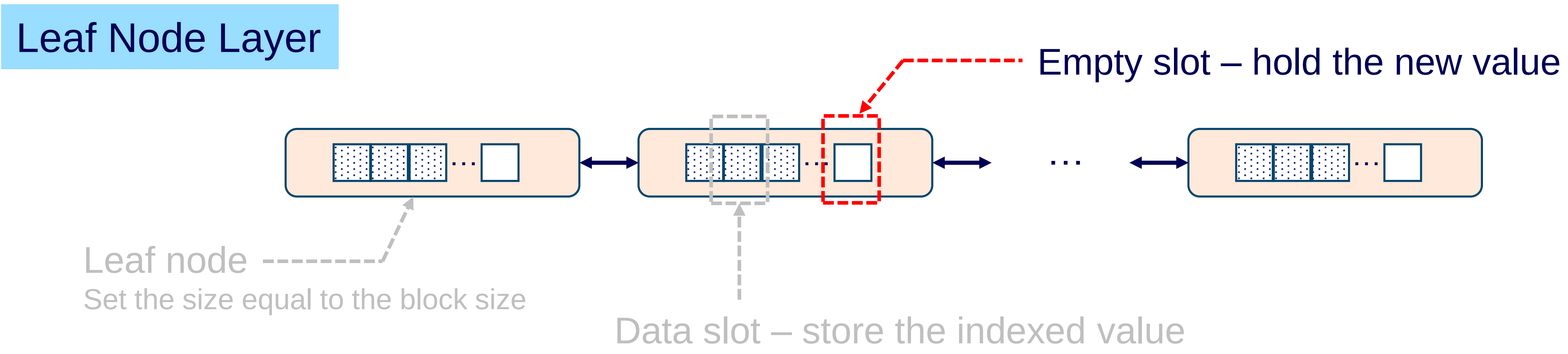
Leaf node

Set the size equal to the block size

AULID Index Structure

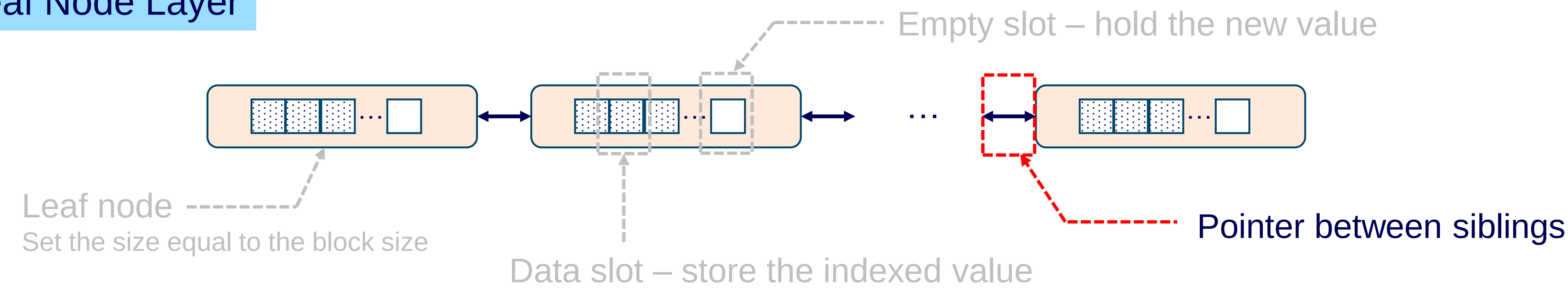


AULID Index Structure



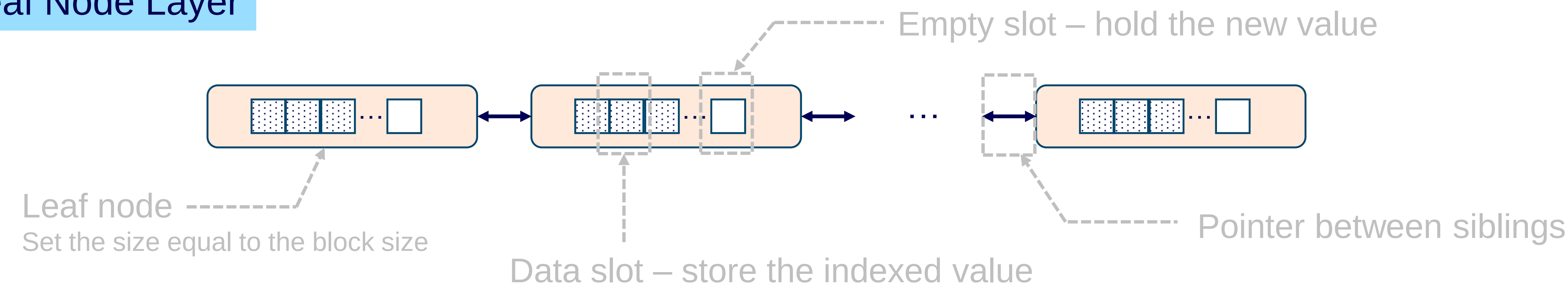
AULID Index Structure

Leaf Node Layer



AULID Index Structure

Leaf Node Layer

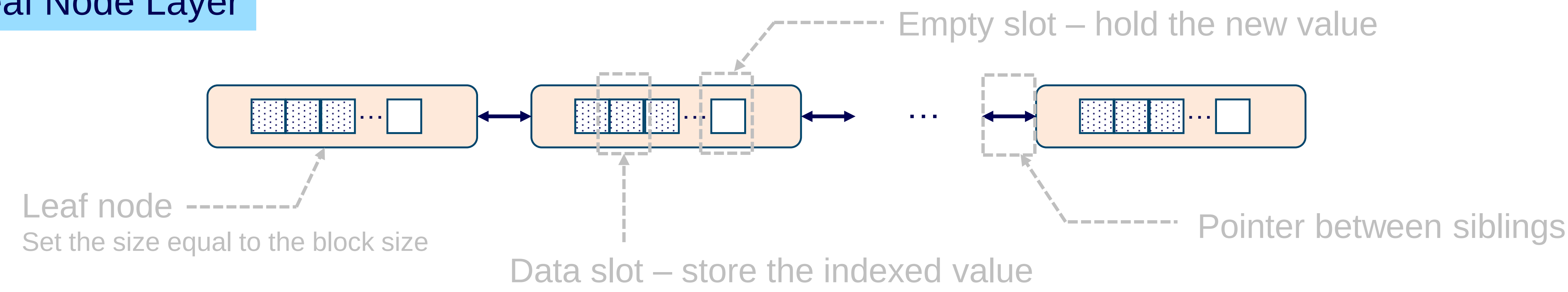


Benefits

- Low overhead for **scan** operations in fetching the *next* item (**P4**).

AULID Index Structure

Leaf Node Layer

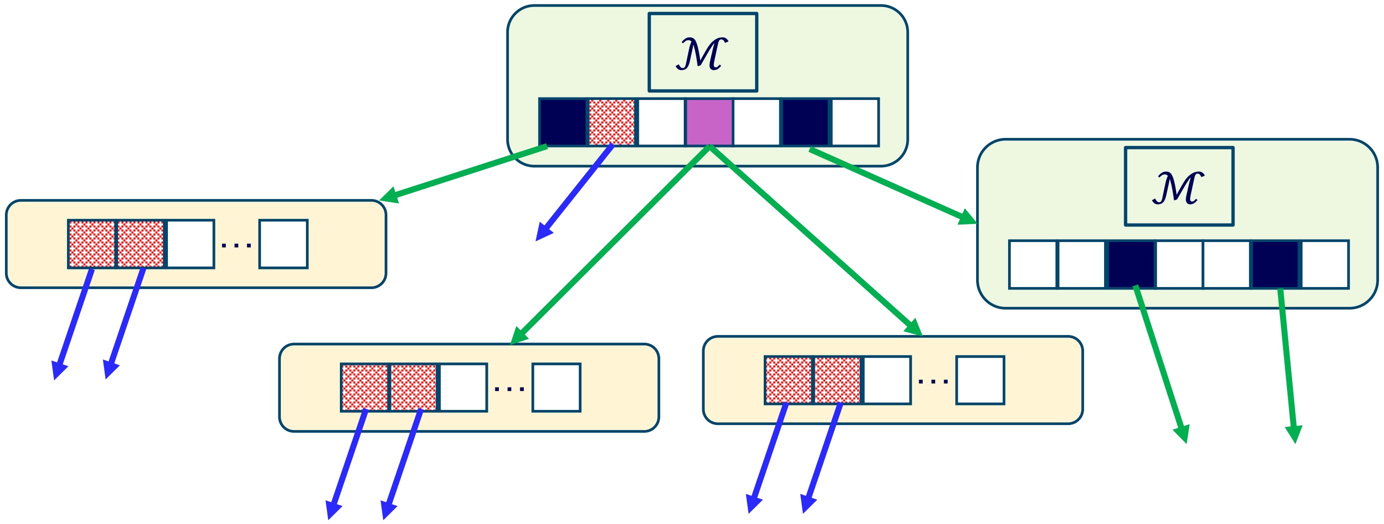


Benefits

- Low overhead for **scan** operations in fetching the *next* item (**P4**).
- Low **insertion** overhead and **SMO** overhead (**P3**).

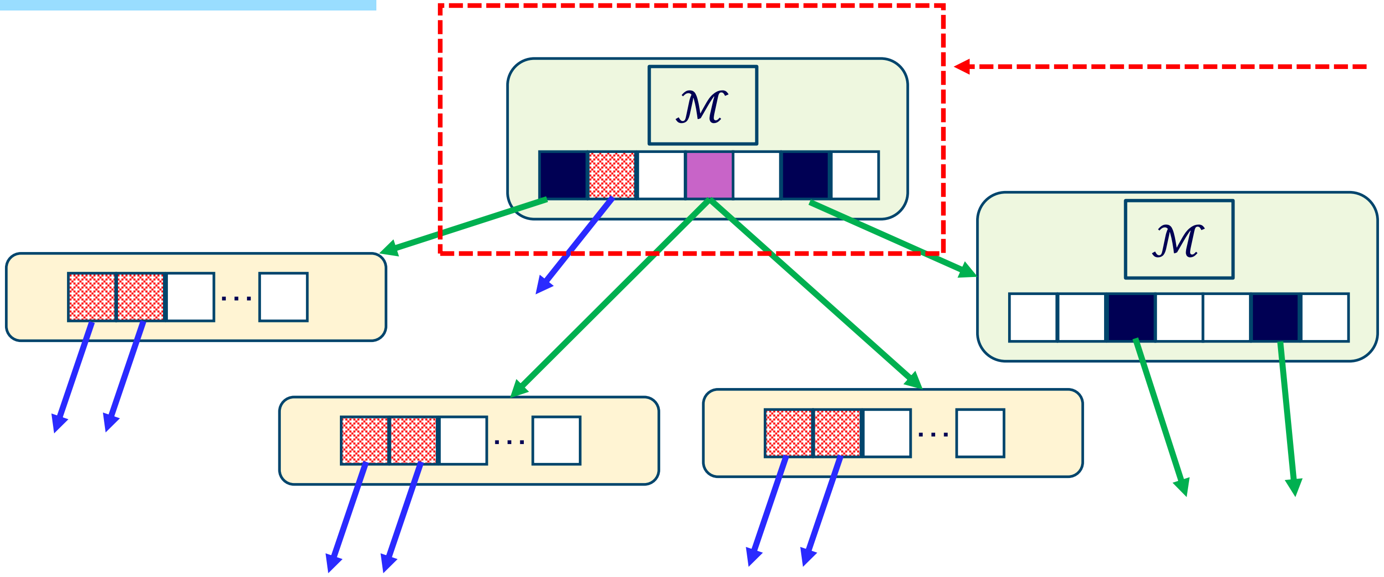
AULID Index Structure

Inner Node Layer



AULID Index Structure

Inner Node Layer

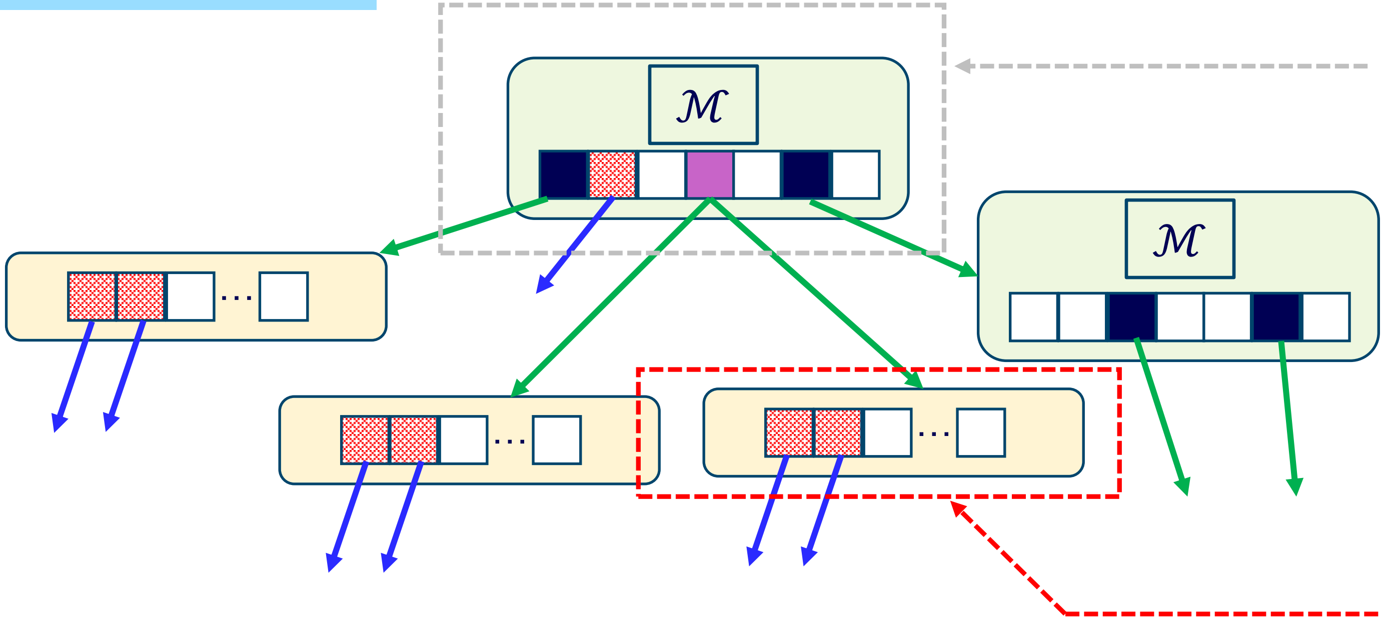


Mixed inner node

- Can hold different **slot** types
- Use a **model** to determine which slot to be accessed next

AULID Index Structure

Inner Node Layer

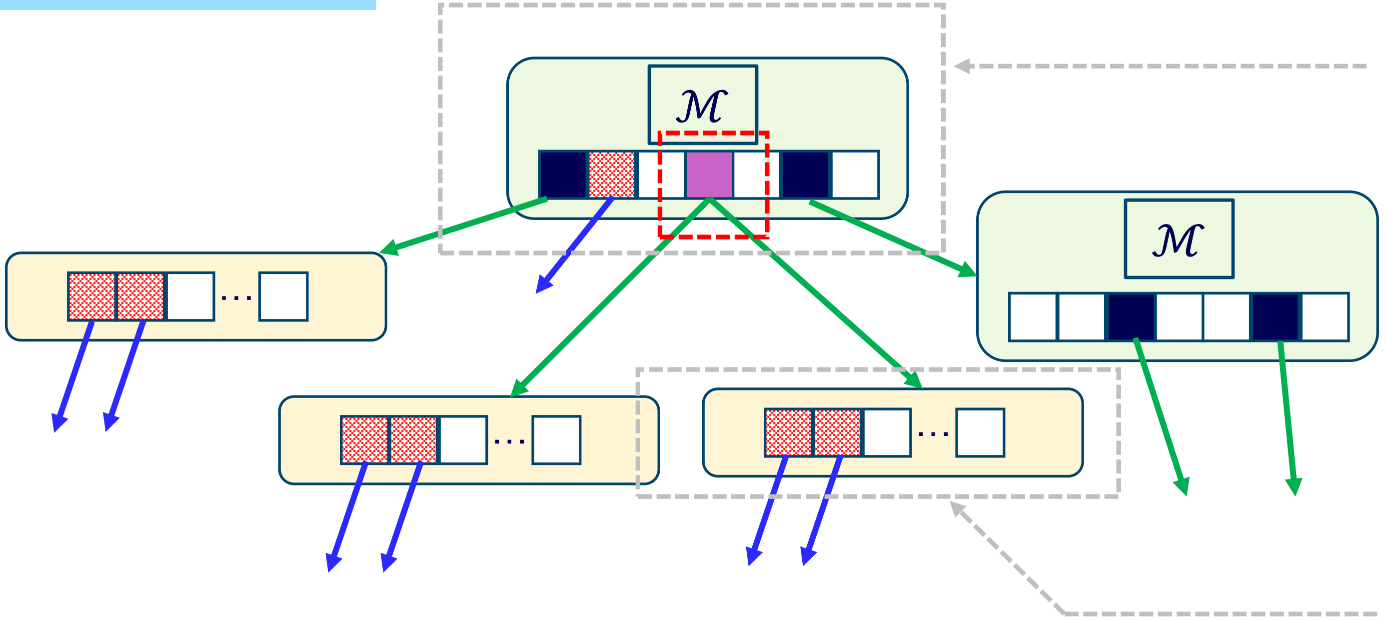


Mixed inner node
- Can hold different **slot** types
- Use a **model** to determine which slot to be accessed next

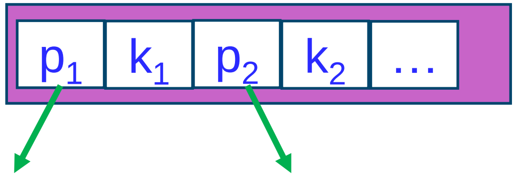
Packed inner node
- Hold the **pointer** to the leaf node and the **maximum key** in the indexed leaf node

AULID Index Structure

Inner Node Layer



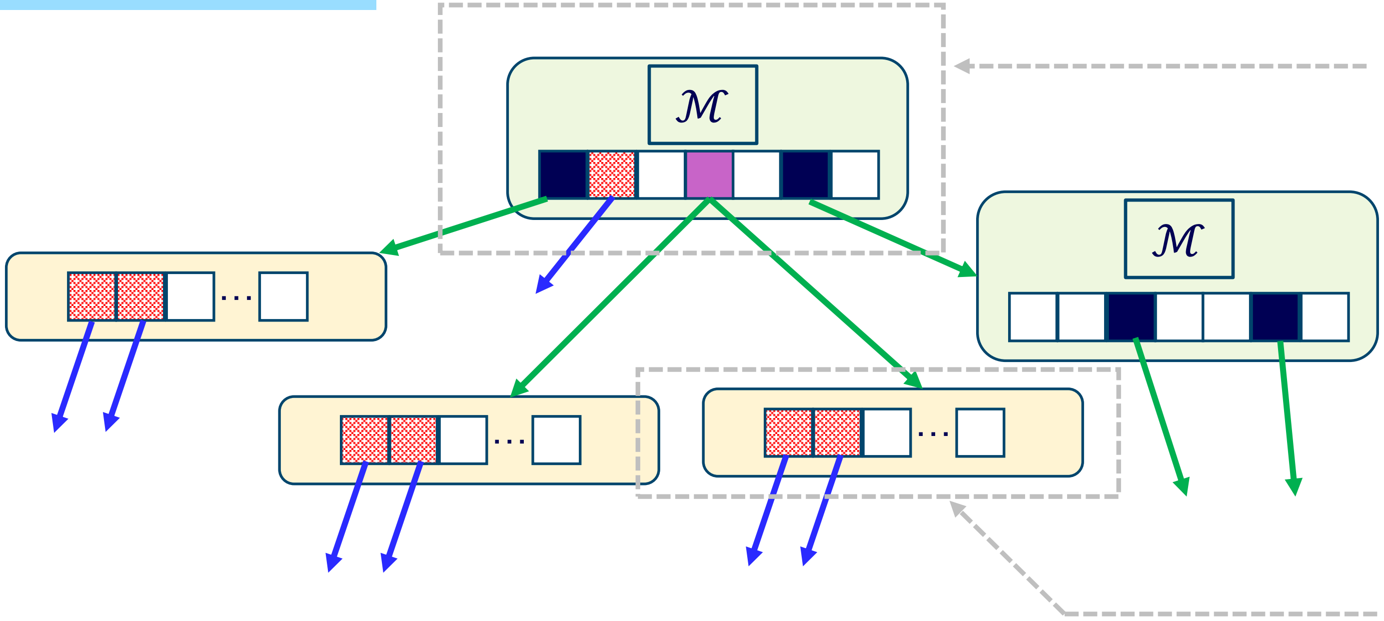
Mixed inner node
- Can hold different **slot** types
- Use a **model** to determine which slot to be accessed next



Packed inner node
- Hold the **pointer** to the leaf node and the **maximum key** in the indexed leaf node

AULID Index Structure

Inner Node Layer



Mixed inner node
- Can hold different **slot** types
- Use a **model** to determine which slot to be accessed next

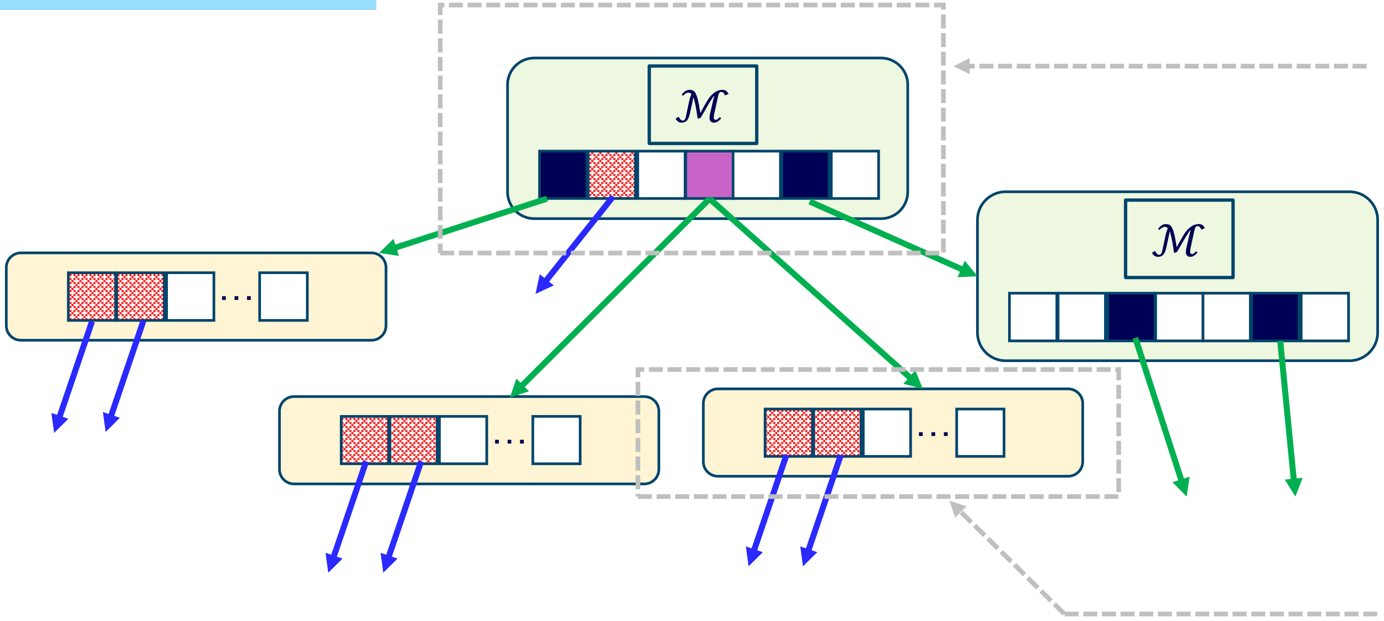
Packed inner node
- Hold the **pointer** to the leaf node and the **maximum key** in the indexed leaf node

Benefits

- Reducing the **tree height** of the index (**P1**).

AULID Index Structure

Inner Node Layer



Mixed inner node
- Can hold different **slot** types
- Use a **model** to determine which slot to be accessed next

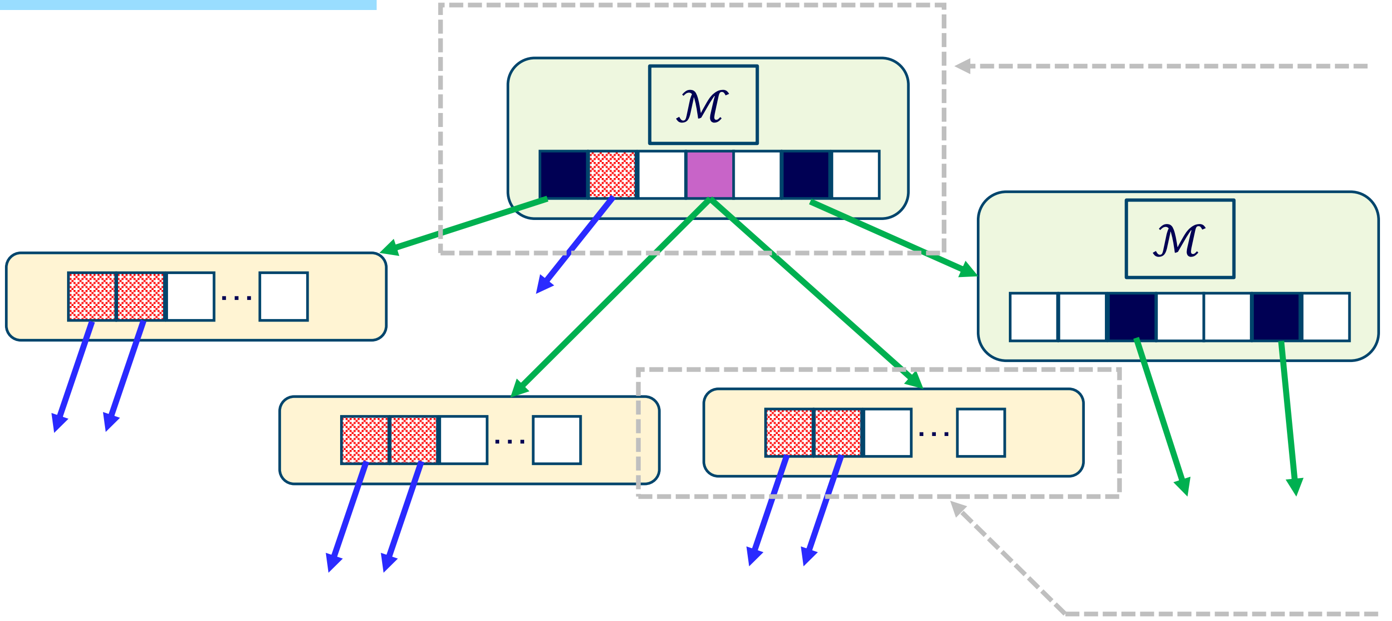
Packed inner node
- Hold the **pointer** to the leaf node and the **maximum key** in the indexed leaf node

Benefits

- Reducing the **tree height** of the index (**P1**).
- **Model-based** operations (search and insert) (**P2**).

AULID Index Structure

Inner Node Layer



Mixed inner node

- Can hold different **slot** types
- Use a **model** to determine which slot to be accessed next

Packed inner node

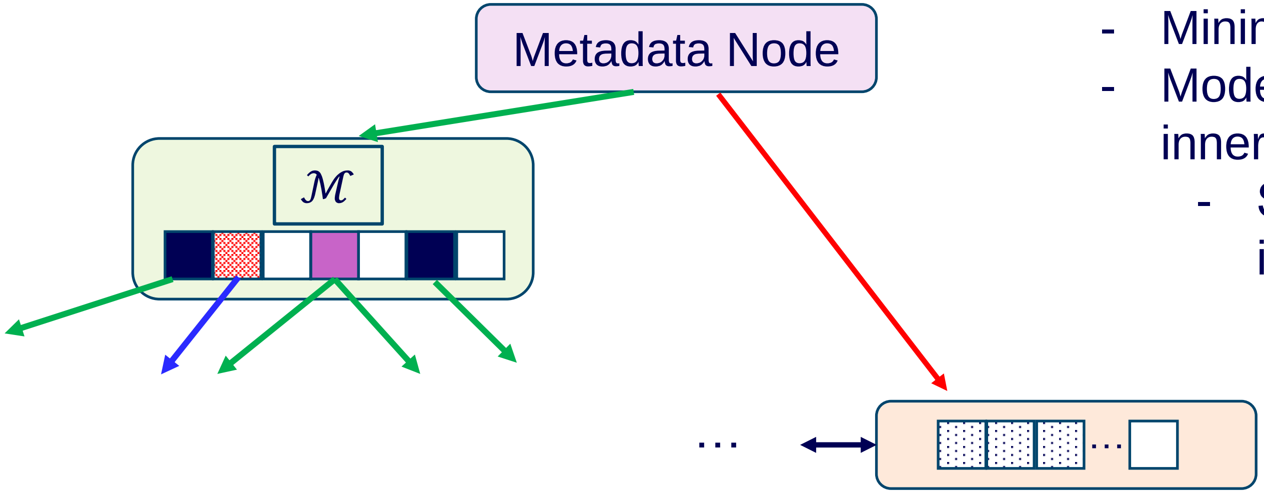
- Hold the **pointer** to the leaf node and the **maximum key** in the indexed leaf node

Benefits

- Reducing the **tree height** of the index (**P1**).
- **Model-based** operations (search and insert) (**P2**).
- Low **SMO** overhead in inner nodes (**P3**).

AULID Index Structure

Metadata Node



- Address of the root node
- Address of the last leaf node
- Minimum key in the last leaf node
- Model of the root node if it is the mixed inner node
    - Store all mixed inner node's model in its parent node

AULID Operations

AULID Operations

Bulkload

Step 1: Construct the leaf nodes and collect the maximum key and address of each leaf node.

AULID Operations

Bulkload

Step 1: Construct the leaf nodes and collect the maximum key and address of each leaf node.

Step 2: Call **FMCD**-based algorithm to construct the inner nodes.

Fastest Minimum Conflict Degree (FMCD)

AULID Operations

Bulkload

Step 1: Construct the leaf nodes and collect the maximum key and address of each leaf node.

Step 2: Call **FMCD**-based algorithm to construct the inner nodes.

Packed inner nodes to hold the keys when #keys mapped to the same slot is not greater than **64**.

A special routing slot to hold the keys when #keys mapped to the same slot is greater than **64** while not larger than **1024**.

When #keys mapped to the same slot is greater than **1024**, we build another **mixed node**.

AULID Operations

Bulkload

Step 1: Construct the leaf nodes and collect the maximum key and address of each leaf node.

Step 2: Call **FMCD**-based algorithm to construct the inner nodes.

- **Packed inner nodes** to hold the keys when #keys mapped to the same slot is not greater than **64**.
- when #keys mapped to the same slot is greater than **64** while not larger than **1024**.
- When #keys mapped to the same slot is greater than **1024**, we build another **mixed node**.

Why we choose FMCD and extend it?

AULID Operations

Bulkload

Step 1: Construct the leaf nodes and collect the maximum key and address of each leaf node.

Packed inner nodes to hold the keys when #keys mapped to the same slot is not greater than **64**.

Step 2: Call **FM** the inner nodes.

hen #keys
4 while

When #keys mapped to the same slot is greater than **1024**, we build another **mixed node**.

Why we choose FMCD and extend it?

- ✓ Compared to other model construction strategies, it can achieve the lowest average tree height most of time.

AULID Operations

Bulkload

Step 1: Construct the leaf nodes and collect the maximum key and address of each leaf node.

Packed inner nodes to hold the keys when #keys mapped to the same slot is not greater than **64**.

Step 2: Call **FM**

the inner nodes.

when #keys

64 while

greater

.

Why we choose FMCD and extend it?

- ✓ Compared to other model construction strategies, it can achieve the lowest average tree height most of time.
- ✓ Each conflict in FMCD will build a new node, which may lead to a high tree height in some parts of the dataset.

AULID Operations

Bulkload

Step 1: Construct the leaf nodes and collect the maximum key and address of each leaf node.

Step 2: Call **FM**
the inner nodes.

Packed inner nodes to hold the keys when #keys mapped to the same slot is not greater than **64.**

vhen #keys
54 while

greater

Why we choose FMCD and extend it?

- ✓ Compared to other model construction strategies, it can achieve the lowest average tree height most of time.
- ✓ Each conflict in FMCD will build a new node, which may lead to a high tree height in some parts of the dataset.
- ✓ Packed inner nodes reduce memory consumption and overhead in SMO.

AULID Operations

Bulkload

Step 1: Construct the leaf nodes and collect the maximum key and address of each leaf node.

Step 2: Call **FMCD**-based algorithm to construct the inner nodes.

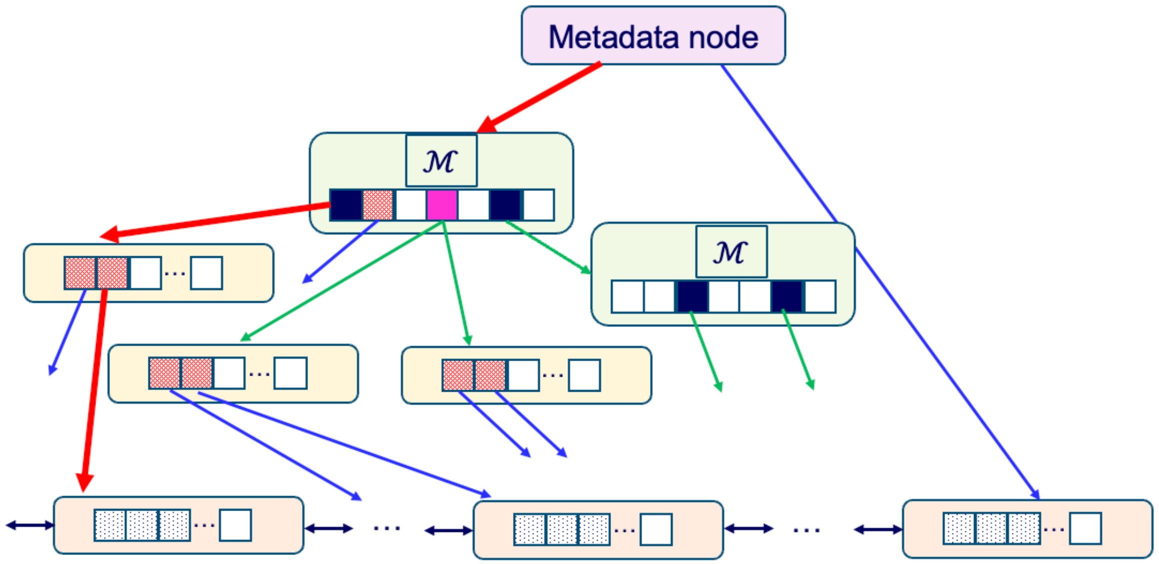
- **Packed inner nodes** to hold the keys when #keys mapped to the same slot is not greater than **64**.
- **A special routing slot** to hold the keys when #keys mapped to the same slot is greater than **64** while not larger than **1024**.
- When #keys mapped to the same slot is greater **1024**, we build another **mixed node**.

Step 3: Build the metadata node.

AULID Operations

Search

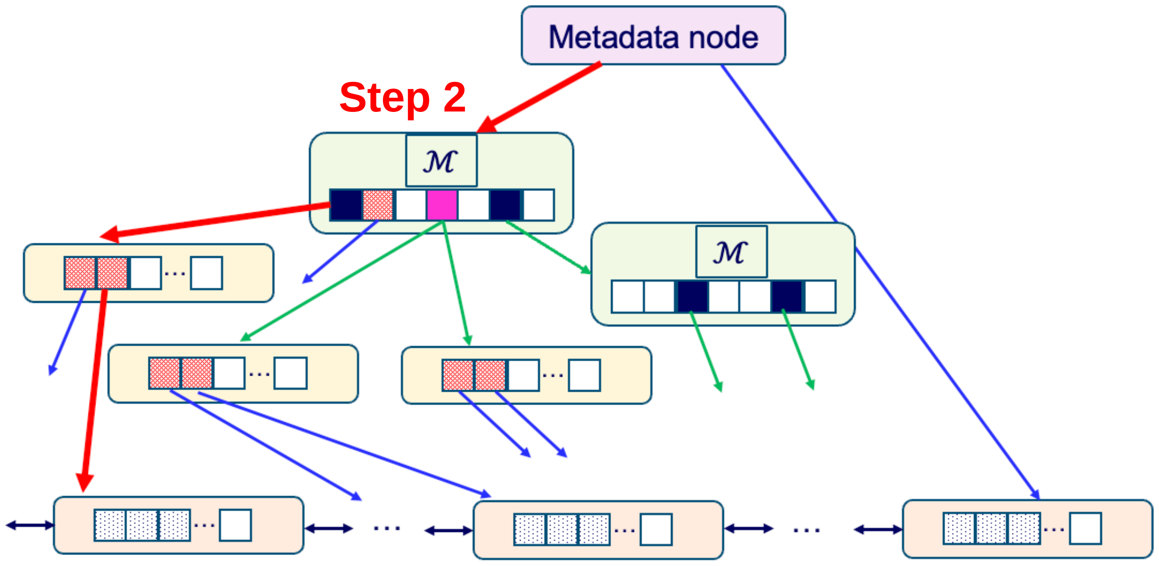
Step 1



Step 1: Visit metadata node and check key_{lk}? key_{min}

AULID Operations

Search

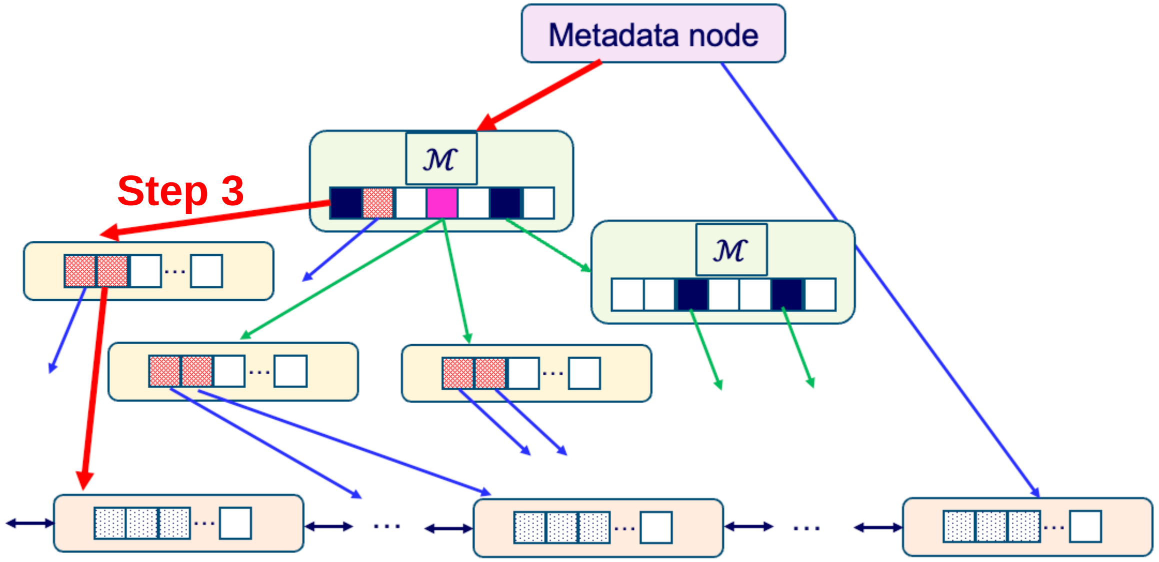


Step 1: Visit metadata node and check key_{lk}? key_{min}

Step 2: Compute which slot to access next in the root node, load related block and read the slot.

AULID Operations

Search



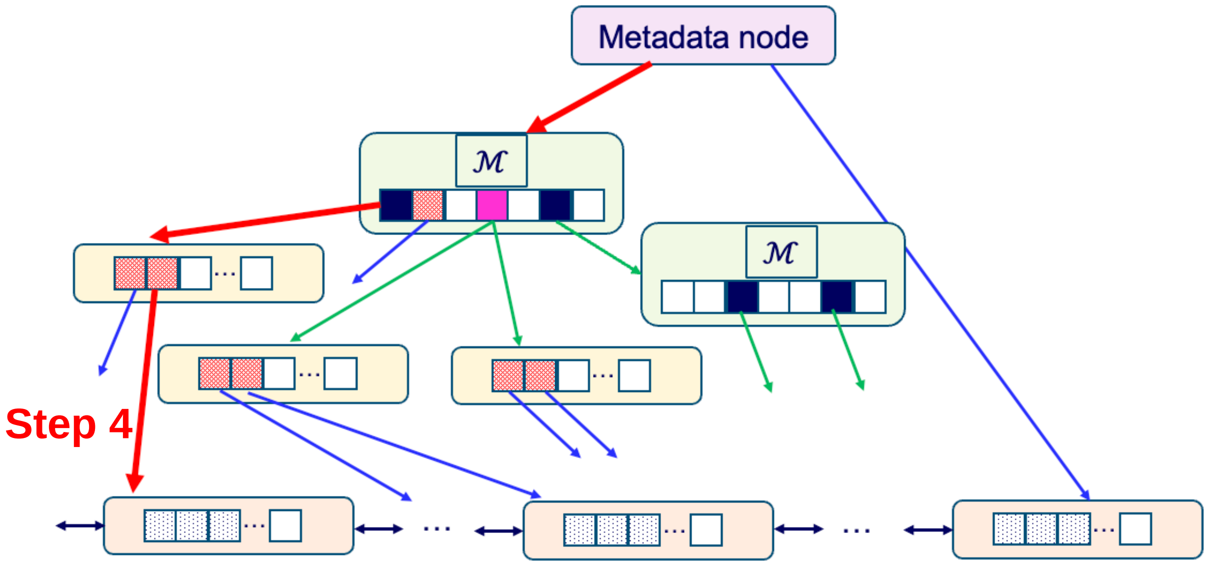
Step 1: Visit metadata node and check key_{lk}? key_{min}

Step 2: Compute which slot to access next in the root node, load related block and read the slot.

Step 3: If the slot type is ■ , read the pointer and load the corresponding block

AULID Operations

Search



Step 1: Visit metadata node and check key_{lk}? key_{min}

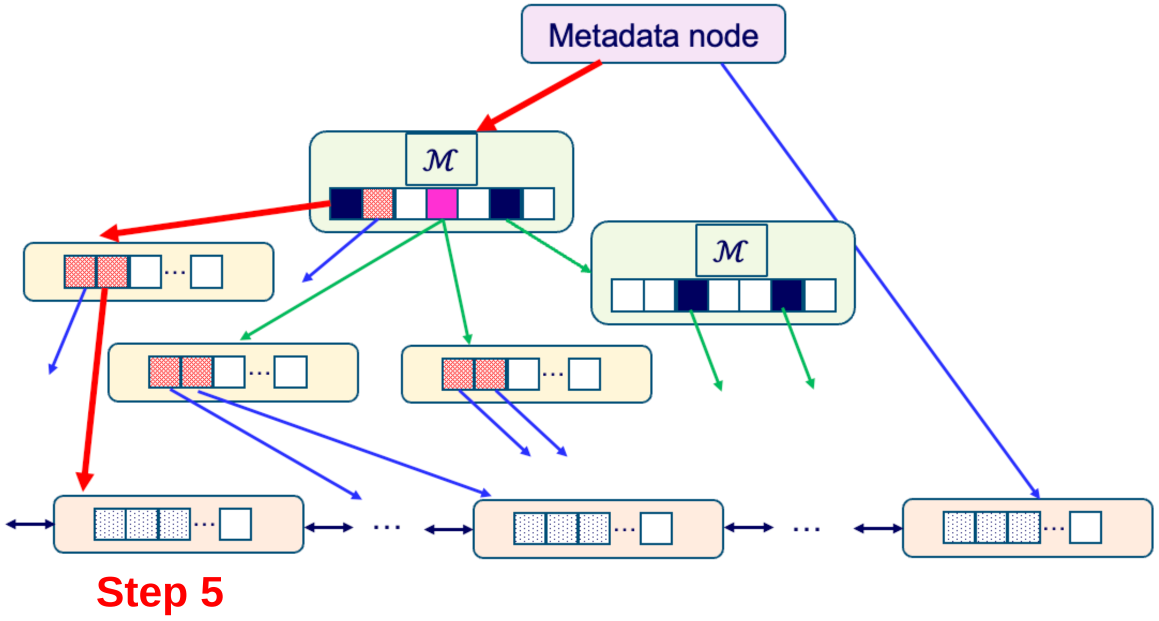
Step 2: Compute which slot to access next in the root node, load related block and read the slot.

Step 3: If the slot type is ▇ , read the pointer and load the corresponding block.

Step 4: If the node is packed inner node, do a binary search to find the slot to access next and load the corresponding leaf node.

AULID Operations

Search



Step 1: Visit metadata node and check key_{lk}? key_{min}

Step 2: Compute which slot to access next in the root node, load related block and read the slot.

Step 3: If the slot type is ▇ , read the pointer and load the corresponding block.

Step 4: If the node is packed inner node, do a binary search to find the slot to access next and load the corresponding leaf node.

Step 5: Do a binary search on the leaf node.

AULID Operations

Insert

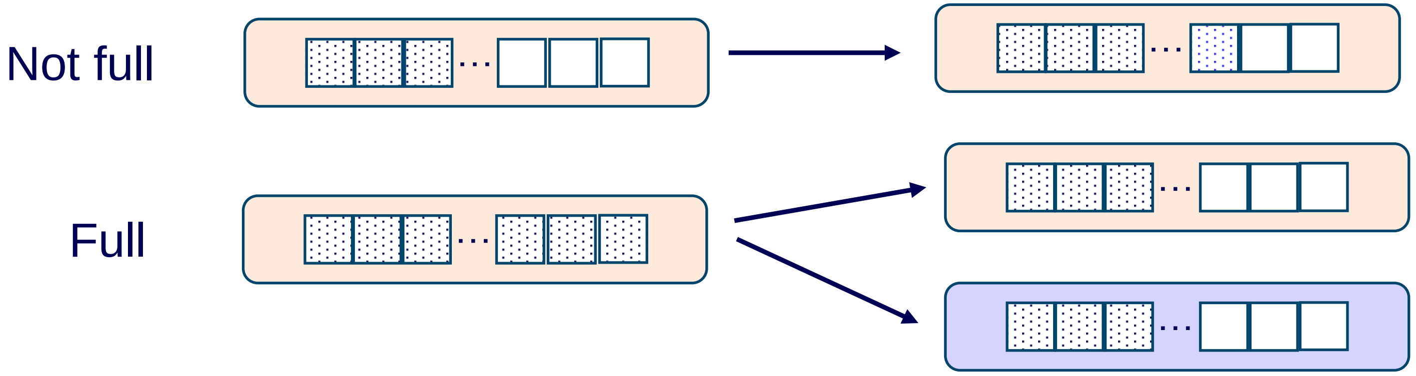
Step 1: Insert into leaf node

Not full

AULID Operations

Insert

Step 1: Insert into leaf node



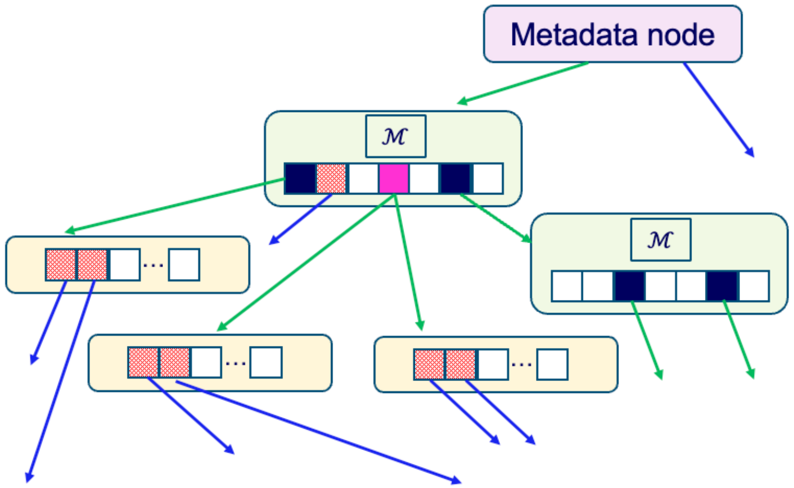
Store the large half values in the **original** block.

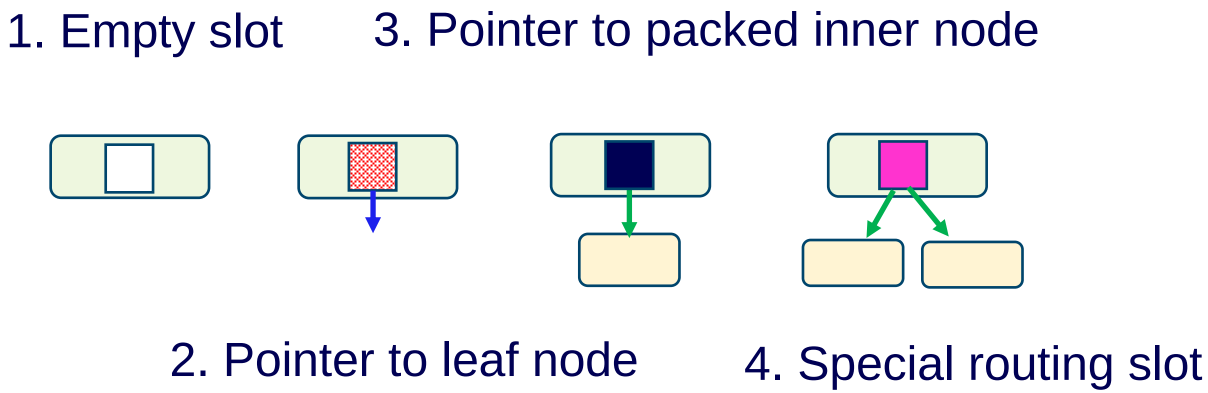
Collect address and key_{max} of the new leaf node.

AULID Operations

Insert

Step 2: Insert (key_{max}, addr) into the inner nodes.

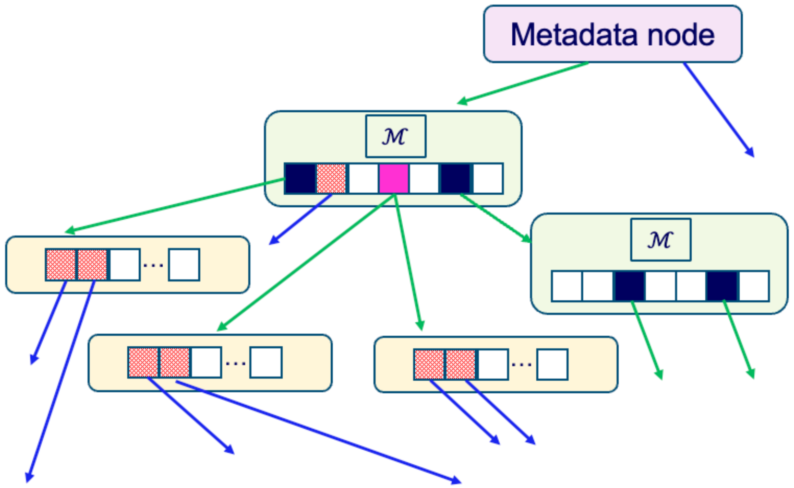




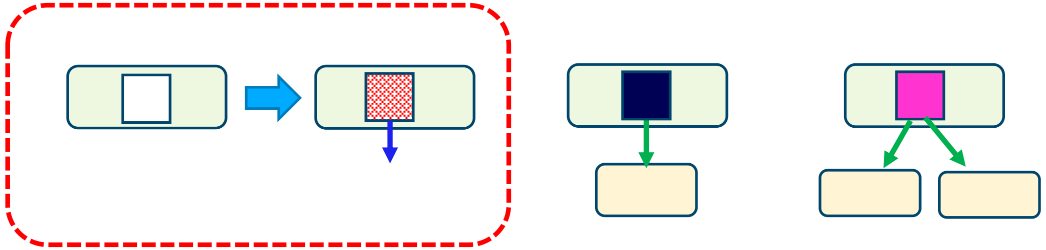
AULID Operations

Insert

Step 2: Insert (key_{max}, addr) into the inner nodes.



1. Empty slot

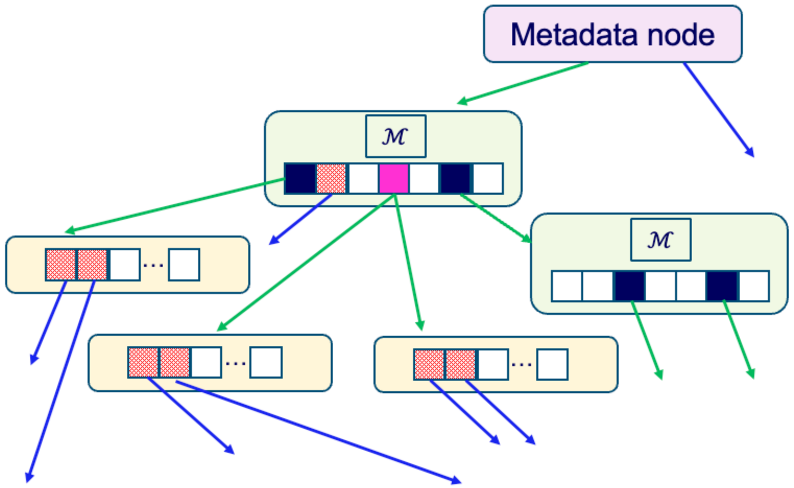


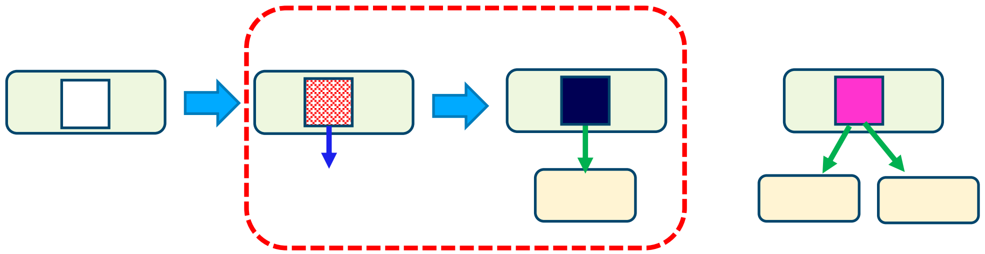
2. Pointer to leaf node

AULID Operations

Insert

Step 2: Insert (key_{max}, addr) into the inner nodes.

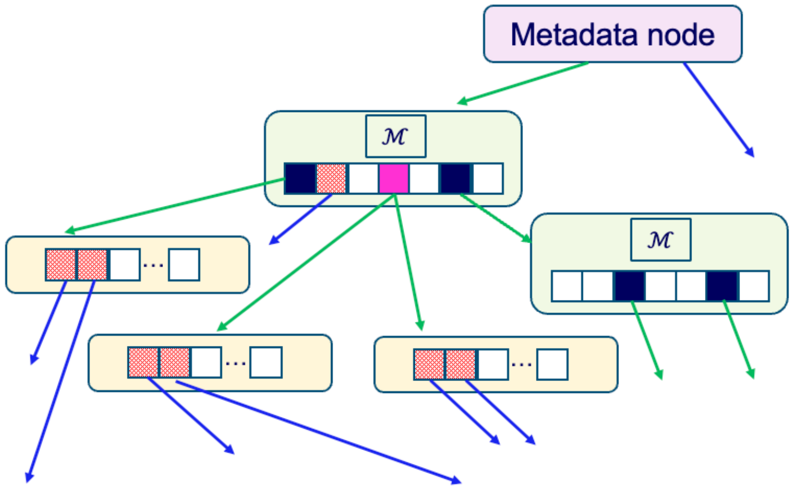


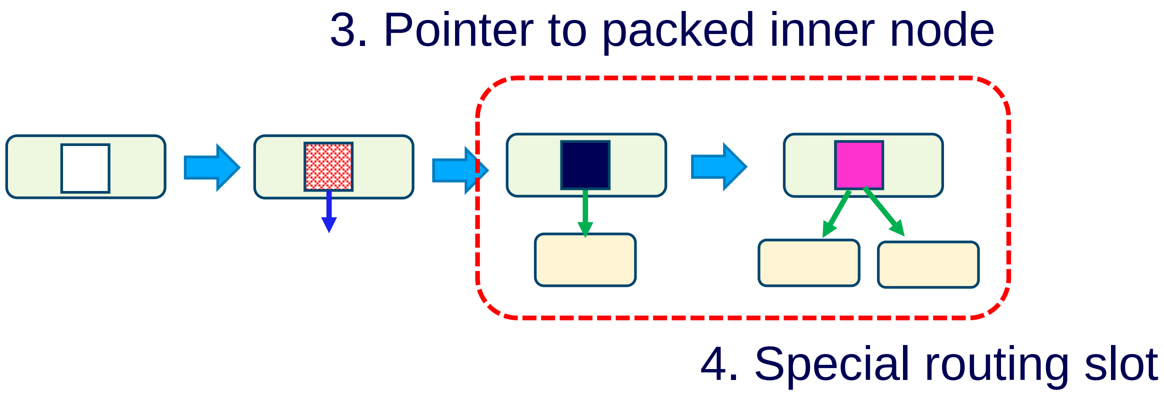


AULID Operations

Insert

Step 2: Insert (key_{max}, addr) into the inner nodes.

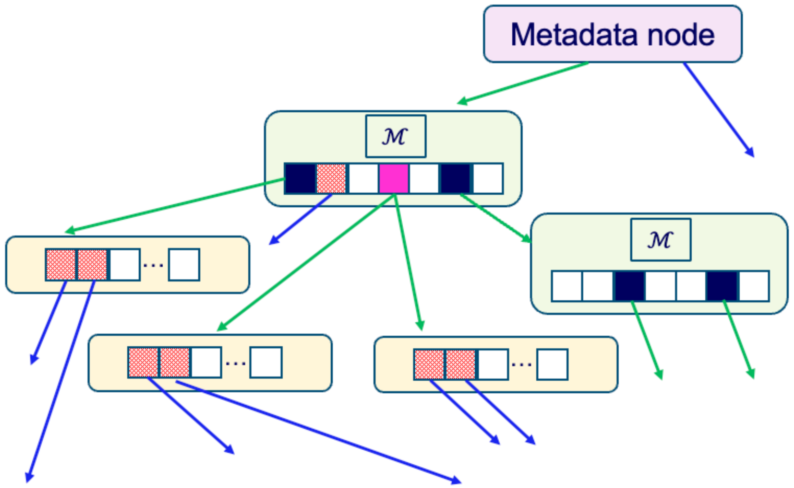


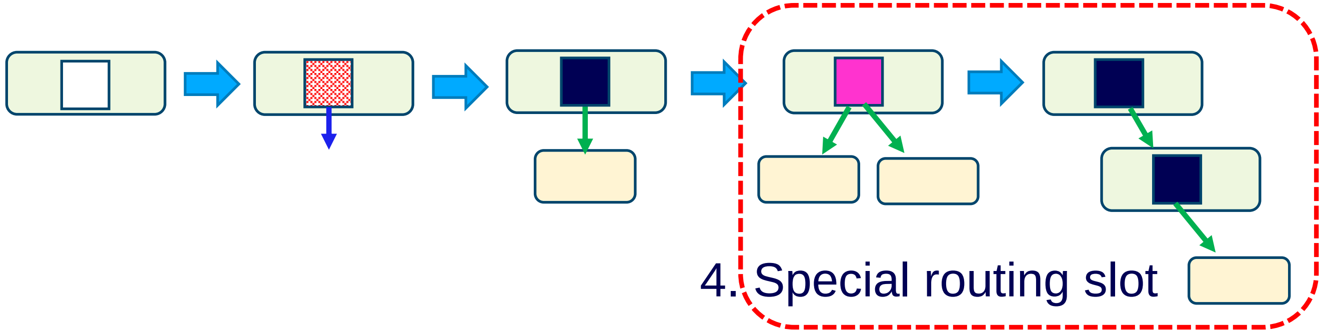


AULID Operations

Insert

Step 2: Insert (key$_{max}$, addr) into the inner nodes.

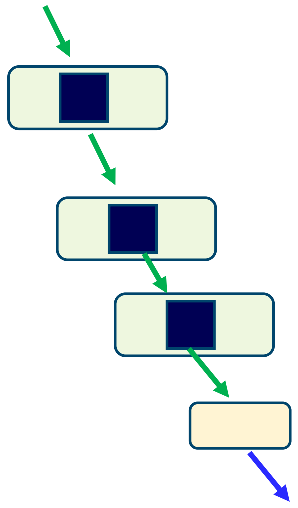




AULID Operations

Tree Adjustment

Why – with more data inserted, some parts of the index may have a large height.



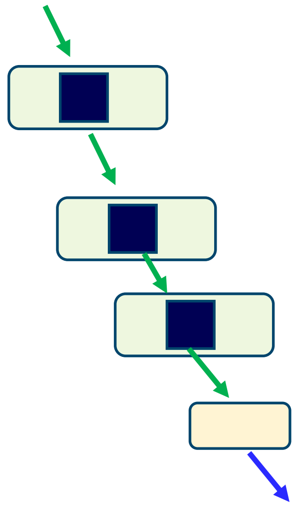
AULID Operations

Tree Adjustment

Why – with more data inserted, some parts of the index may have a large height.

When – two criteria met at the same time:

- Percentage of the items in a subtree rooted at node n in the third layer or a deeper layer is larger than α.
- Number of current items rooted at node n is larger than β times of the initial size.



AULID Operations

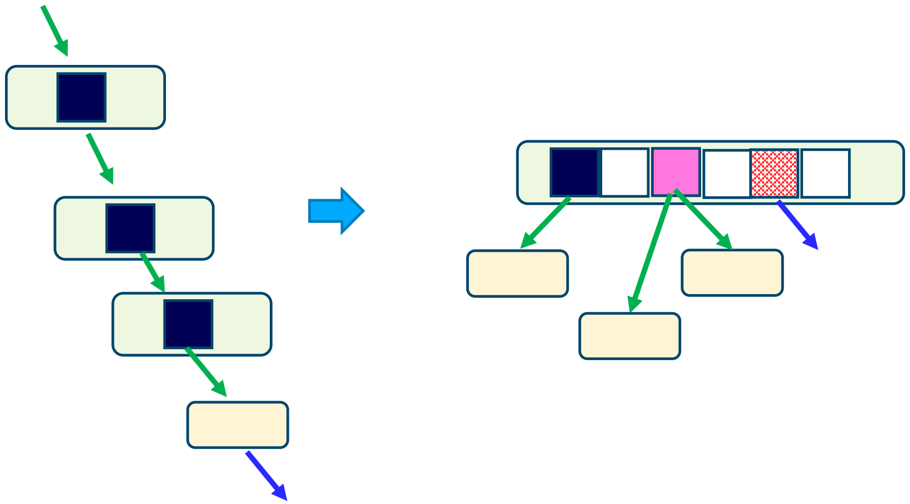
Tree Adjustment

Why – with more data inserted, some parts of the index may have a large height.

When – two criteria met at the same time:

- Percentage of the items in a subtree rooted at node n in the third layer or a deeper layer is larger than α.
- Number of current items rooted at node n is larger than β times of the initial size.

How – reload the inner node items rooted at node n and call our revised FMCD algorithm.



Experiment – Goal

Q1: How good is AULID as compared to other learned indexes and a B+-tree when disk-resident?

Q2: How well does AULID scale to large datasets?

Q3: Do the proposed index structure design and structural modification operation help improve the performance?

Q4: What are the impacts of different parameter settings on AULID performance?

Experiment – Goal

Q1: How good is AULID as compared to other learned indexes and a B+-tree when disk-resident?

Q2: How well does AULID scale to large datasets?

Q3: Do the proposed index structure design and structural modification operation help improve the performance?

Q4: What are the impacts of different parameter settings on AULID performance?

Experiment – Setup

- Datasets

Hardness		Global Hardness		
		Easy	Normal	Hard
Local Hardness	Easy	***C1***		
	Normal		***C2***	***C4***
	Hard		***C3***	

C1: *COVID* (200M / **800M**)

C2: *PLANET* (200M / 800M)

C3: *GENOME* (200M / 800M)

C4: *OSM* (200M / 800M)

- Baselines
  - ALEX, PGM, FITing-tree, LIPP, B+-tree

Are Updatable Learned Indexes Ready? VLDB 2022

Updatable Learned Indexes Meet Disk-Resident DBMS - From Evaluations to Design Choices SIGMOD 2023

Experiment – Setup

Workload

Lookup-Only	Scan-Only	Write-Only	Write-Heavy	Balanced	Read-Heavy
100% lookups	100% scans	100% inserts	90% inserts 10% lookups	50% inserts 50% lookups	10% inserts 90% lookups

Metric

Throughput (number of operations per second) — Larger is better

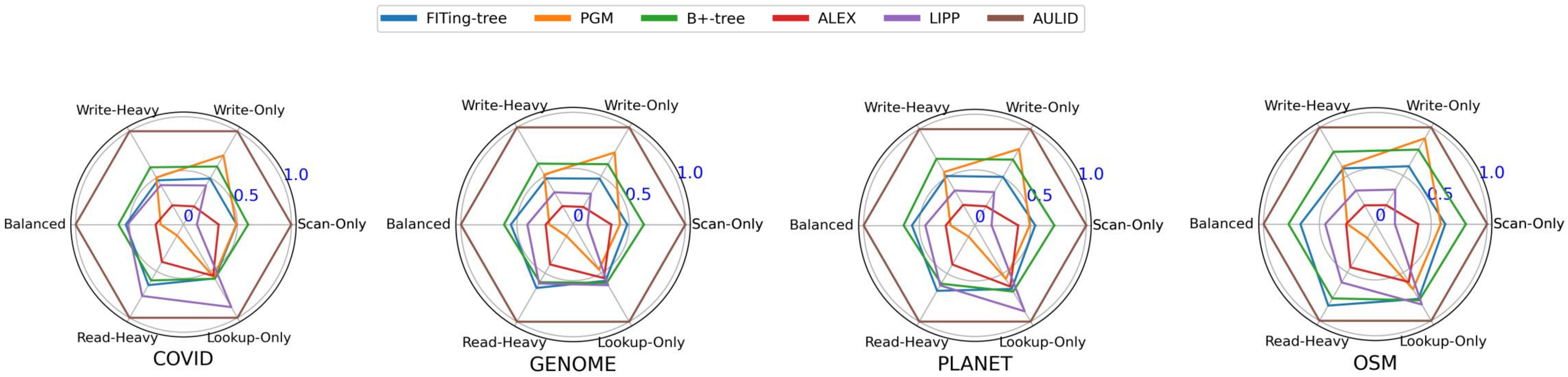
Fetched block count from disk

Storage size

P99 latency & standard deviation

Smaller is better

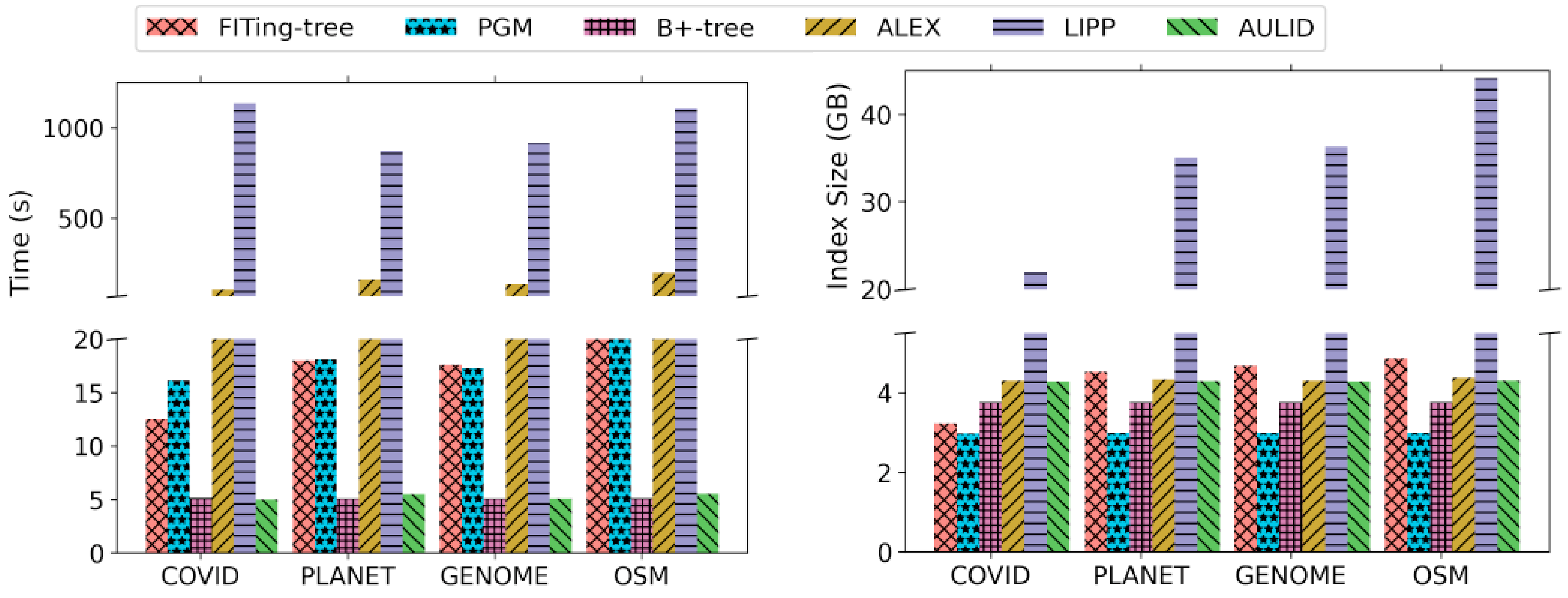
Experiment – Throughput Comparison



✓ AULID **significantly** beats other indexes in **all** datasets and workloads.

✓ B+-tree is the **second best** in **most** workloads and datasets.

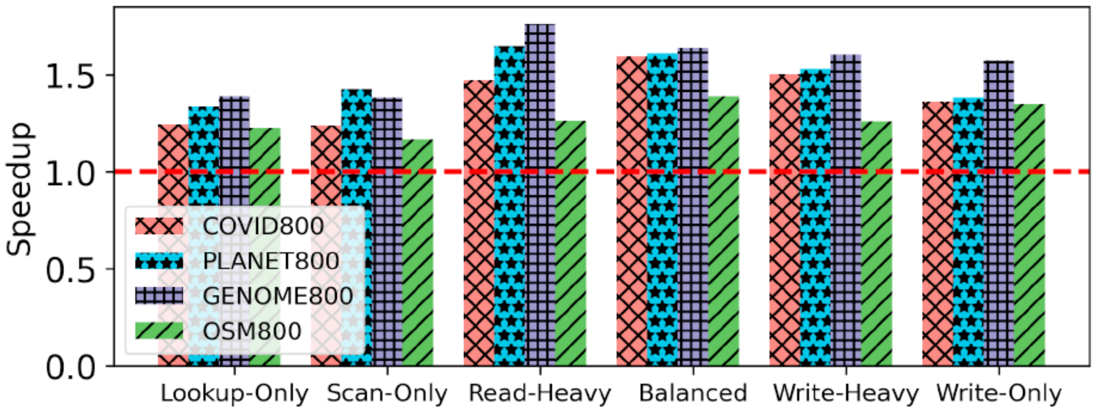
Experiment – Bulkload & Storage



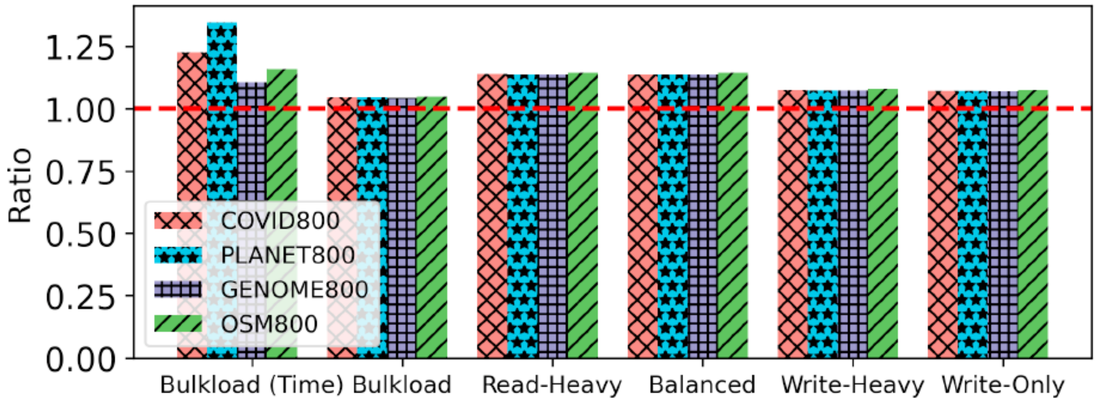
✓ AULID has **similar** bulkload time to B+-tree and is **faster** than other indexes.

✓ AULID has a **stable** index size among different dataset and is **competitive** to B+-tree.

Experiment – Large Scale Data



Throughput Speedup



Storage/time ratio

✓ The **superiority** of AULID also holds on large scale datasets.

Conclusion

- We reveal the **challenges** when applying the learned indexes on disk and propose our design **principles**.
- We propose **AULID** to meet the principles with the carefully designed index layout and operations.
- Our experiments show AULID **significantly beats** our baselines in all workloads and testing datasets.

Thanks!