BOSTON UNIVERSITY

CS 561: Data Systems Architectures

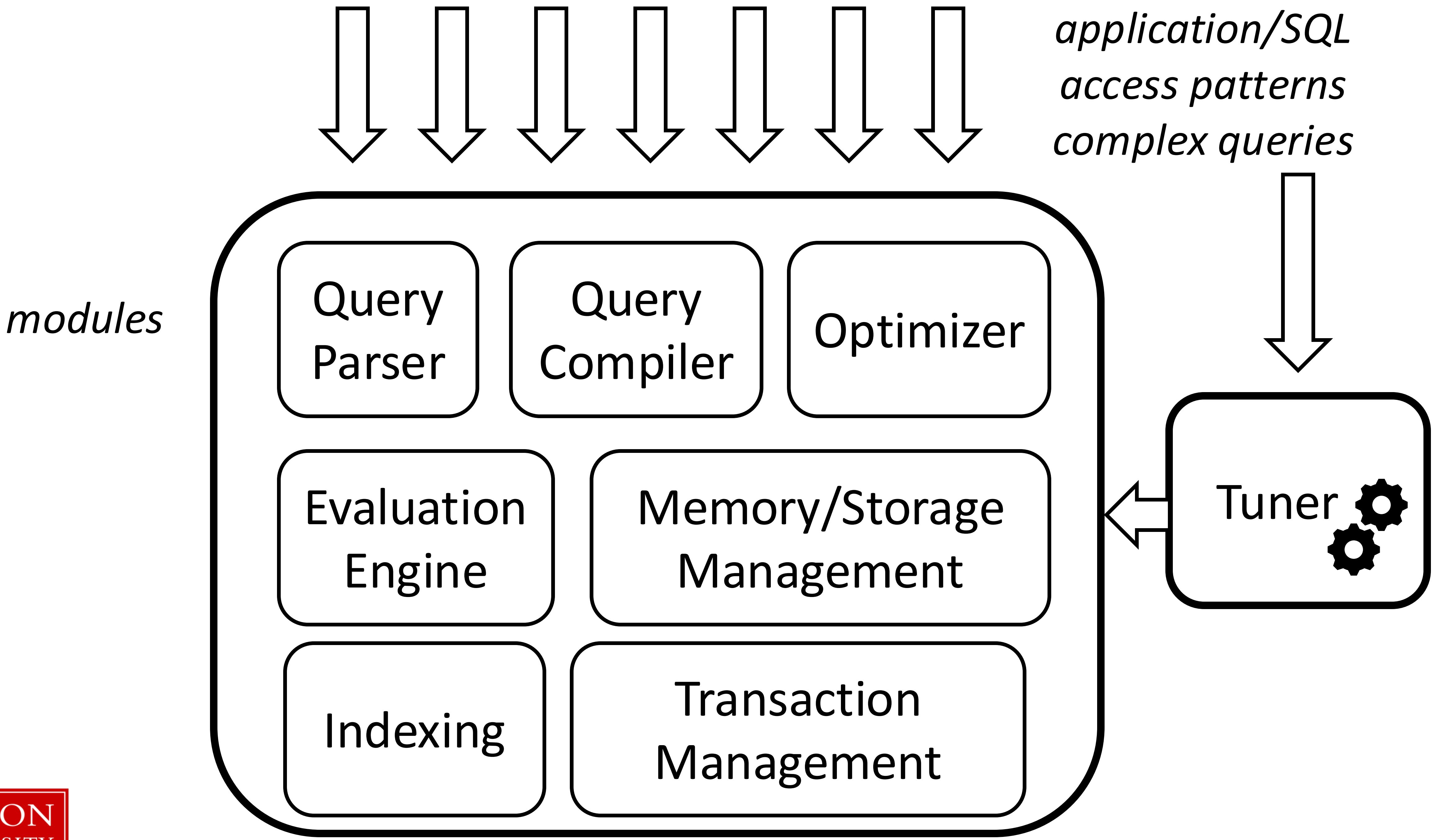
class 24

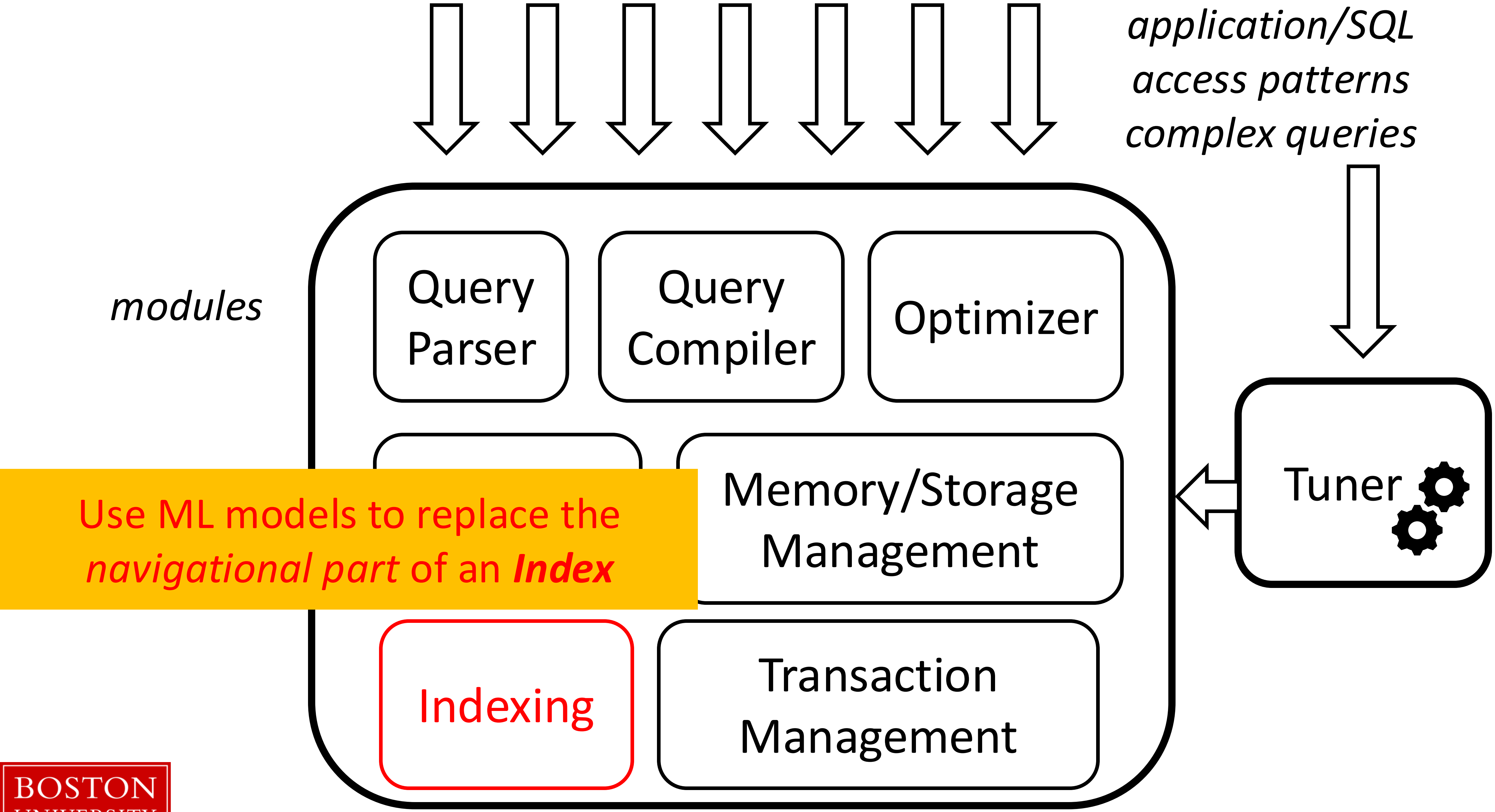
Learned Indexes

Prof. Manos Athanassoulis

https://bu-disc.github.io/CS561/

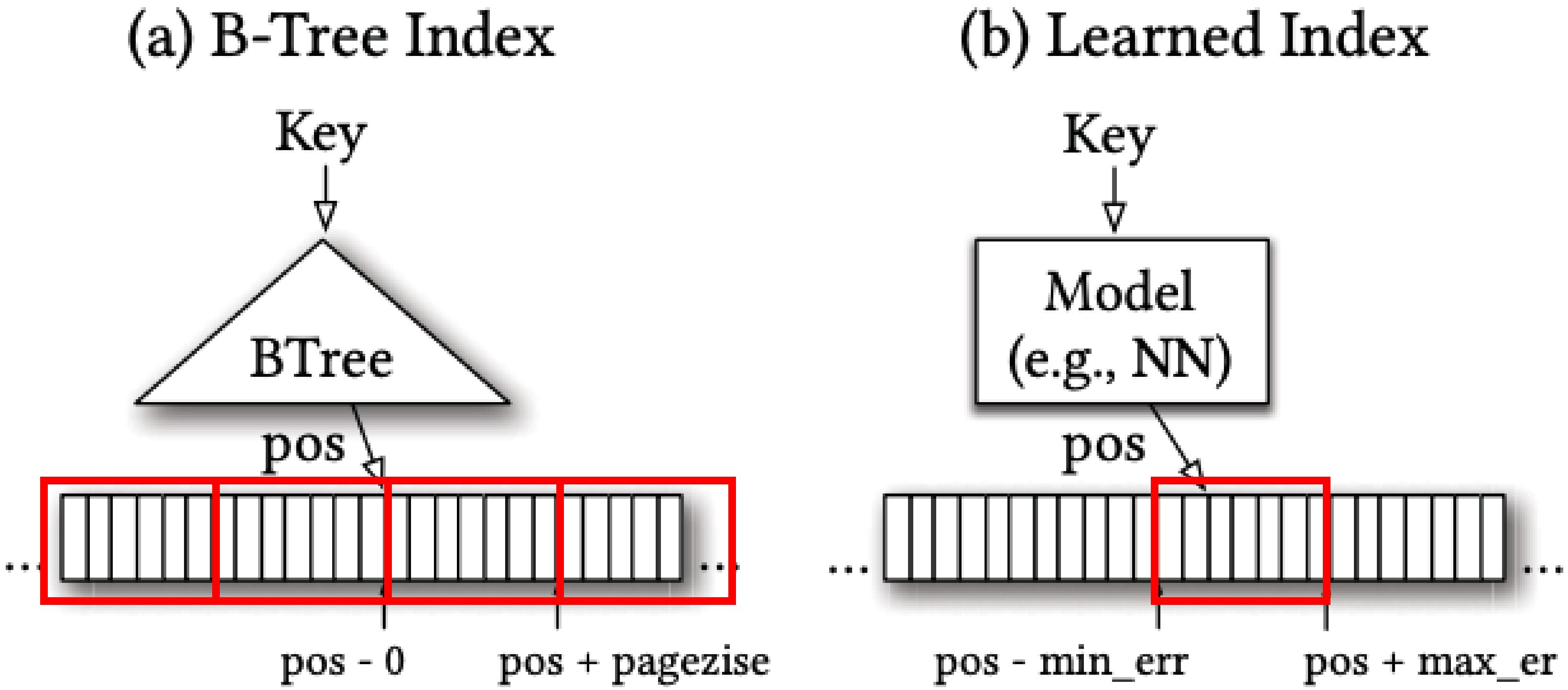
with slides from Jialin Ding


application/SQL
access patterns
complex queries
modules
Query Parser
Query Compiler
Optimizer
Evaluation Engine
Memory/Storage Management
Indexing
Transaction Management
Tuner



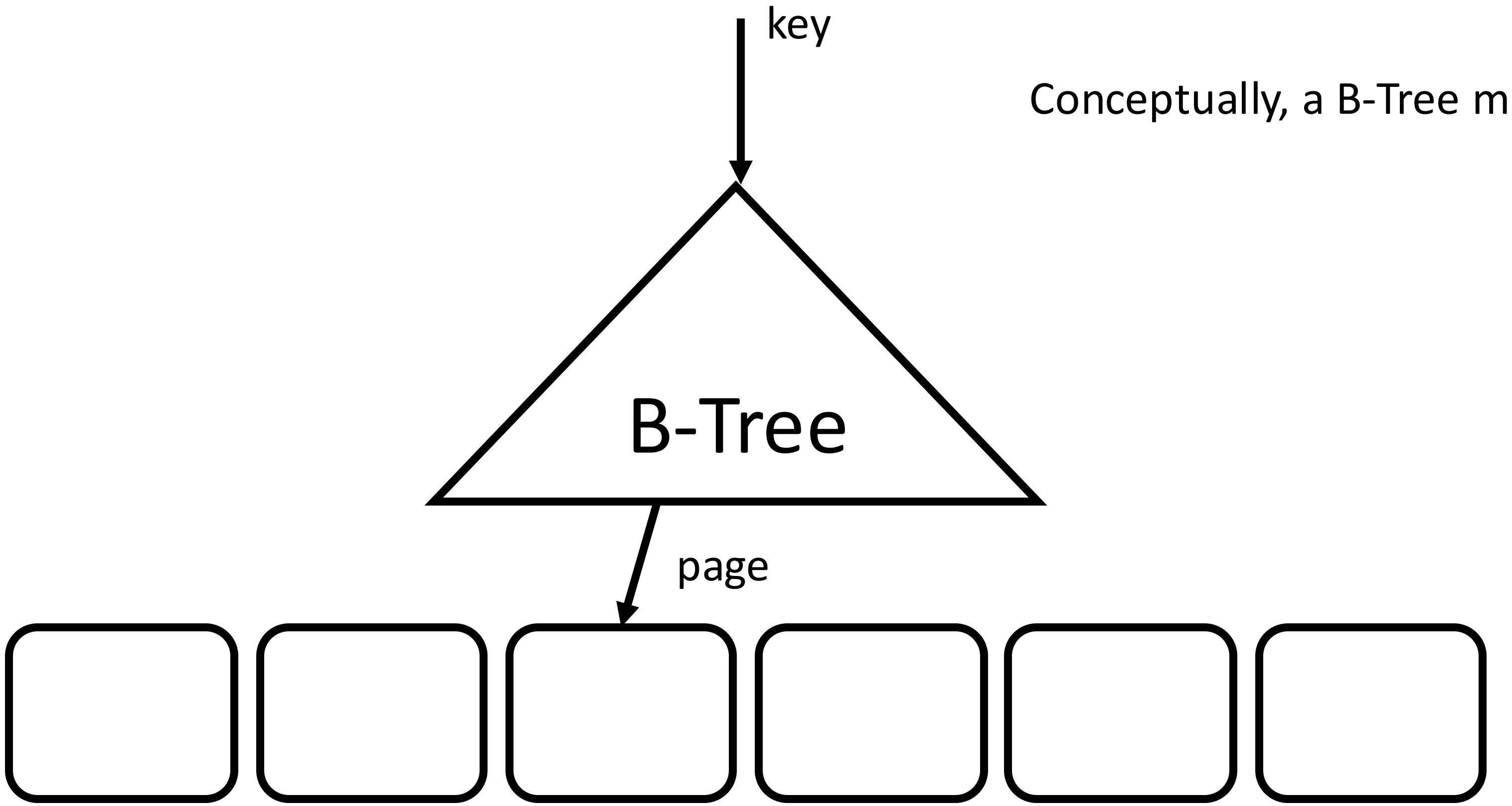
BOSTON UNIVERSITY

B-Trees vs. Learned Indexes



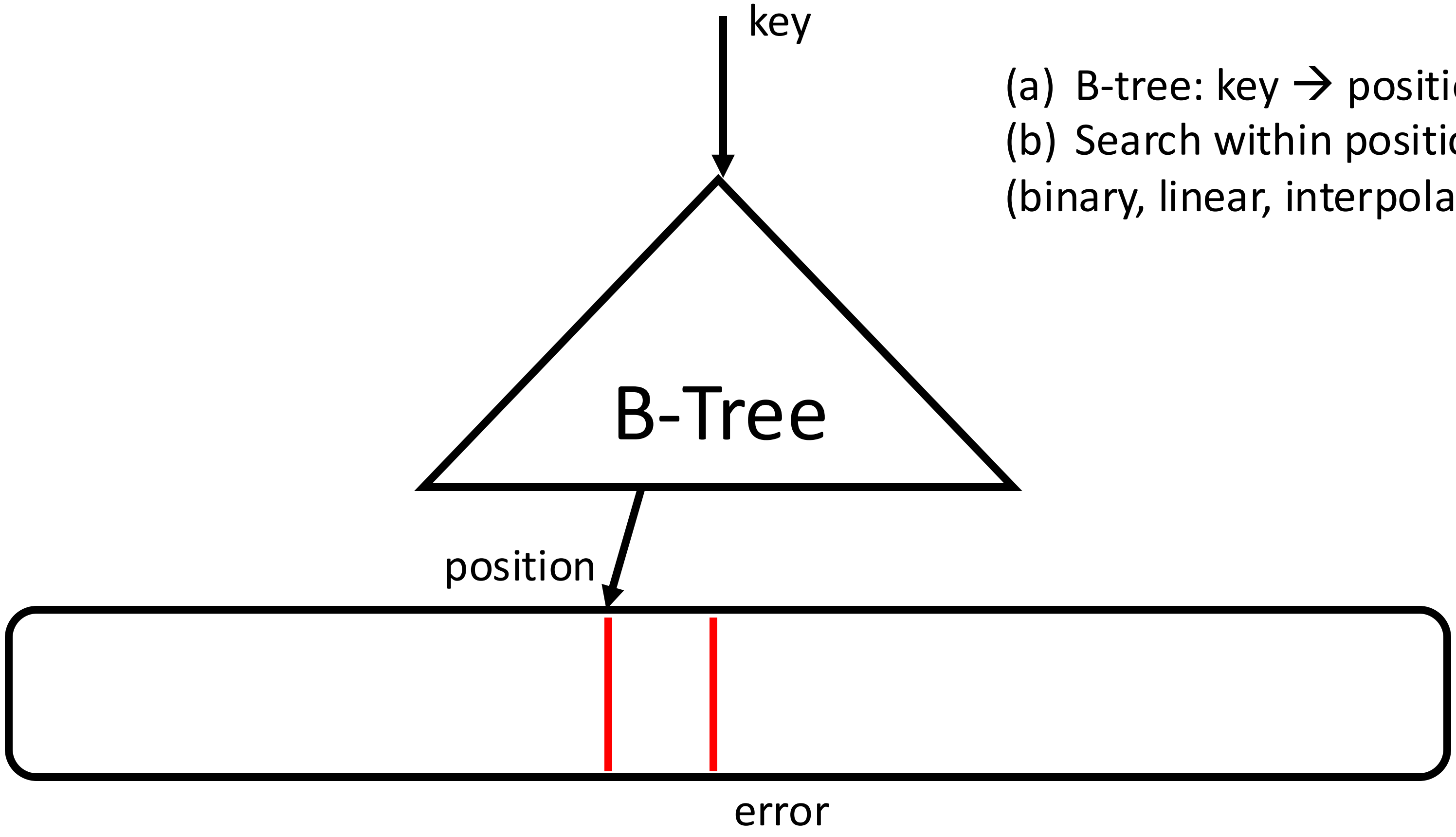
BOSTON UNIVERSITY

What is the difference?



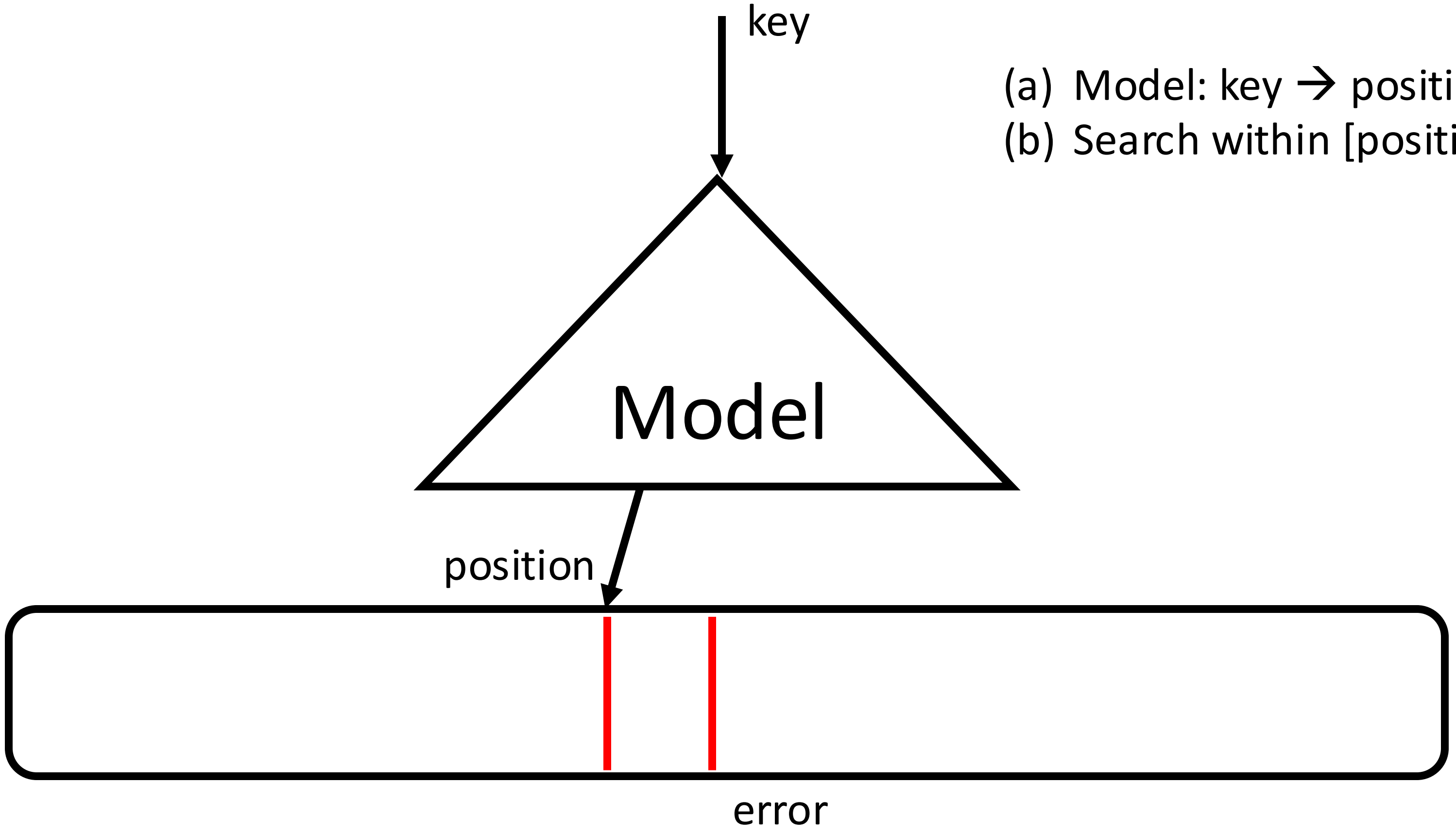
Conceptually, a B-Tree maps a key to a location (page)

Alternative view: data is sorted



(a) B-tree: key → position
(b) Search within position, position+error
(binary, linear, interpolation, exponential search)

A B-Tree is a Model

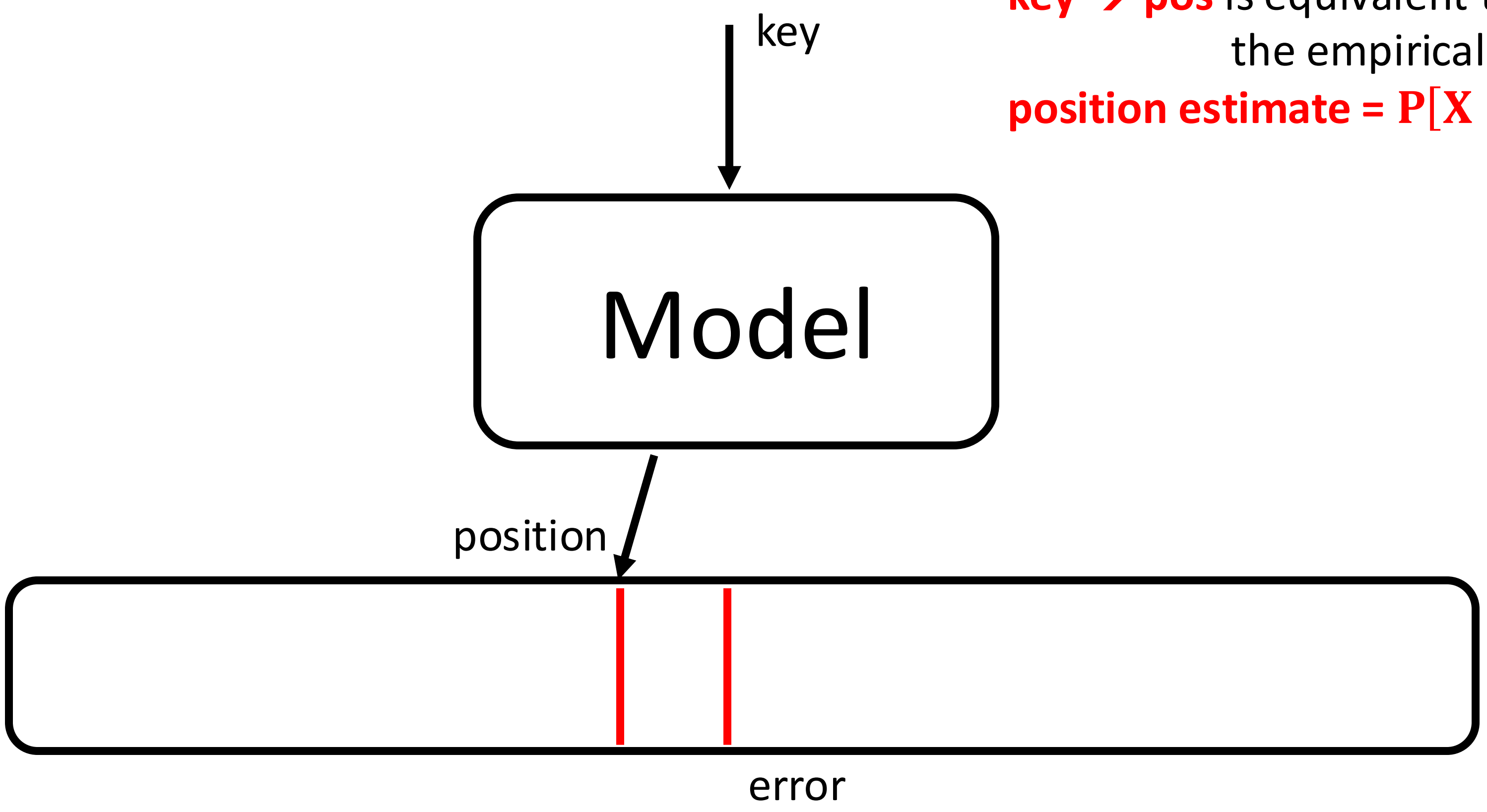


(a) Model: key → position estimate
(b) Search within [position-error, position+error]

BOSTON UNIVERSITY

A B-Tree is already a model!

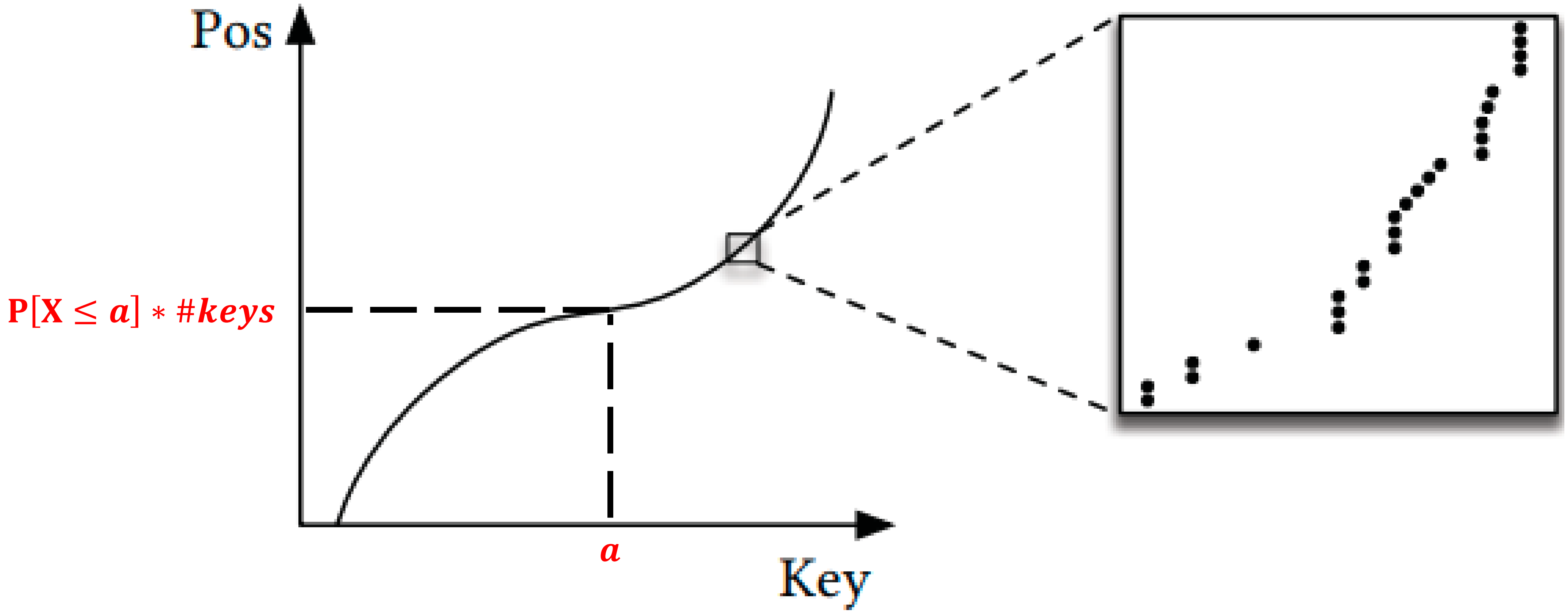
A B-Tree is a Model



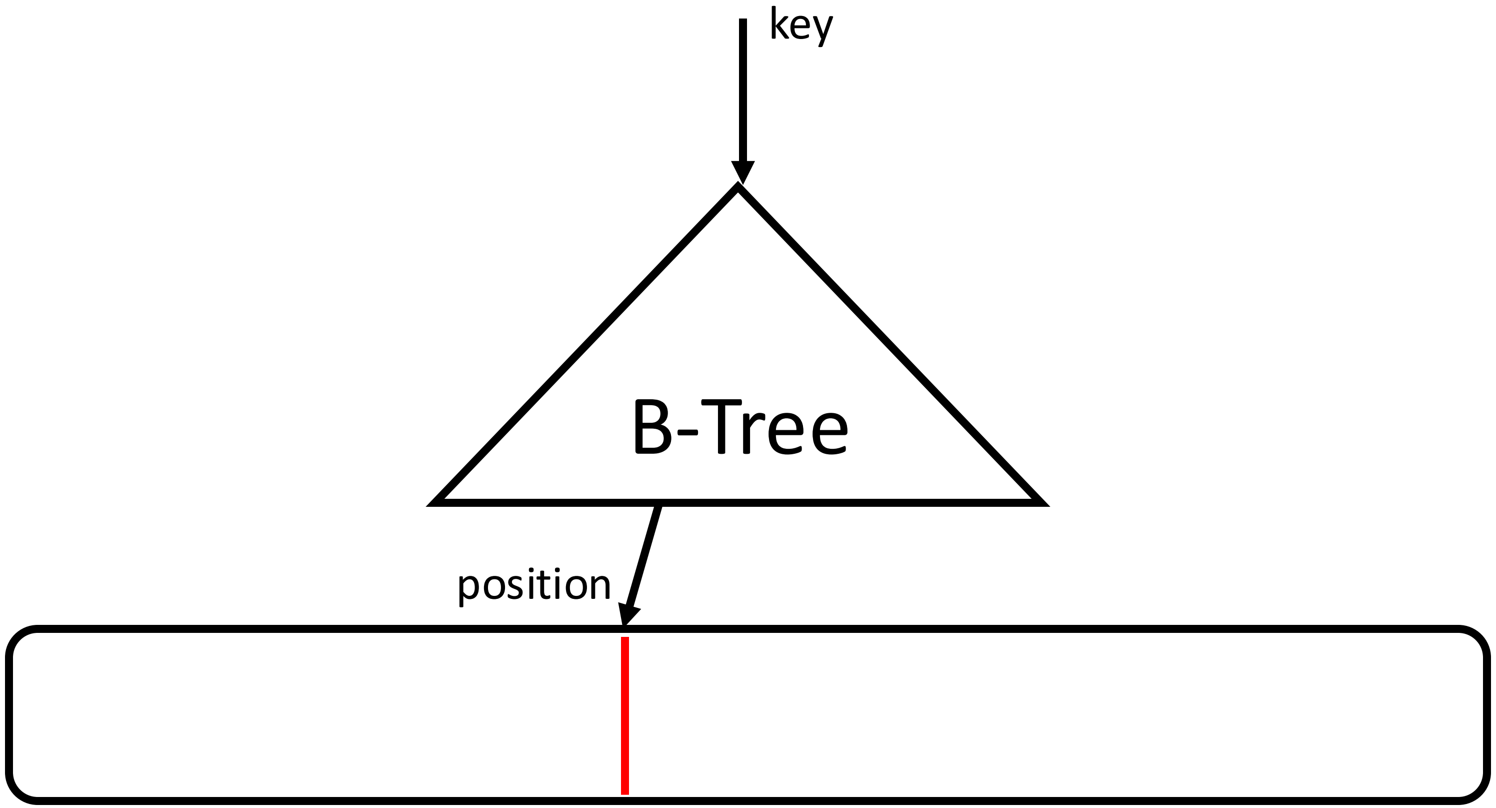
A form of regression model

key → pos is equivalent to modeling the empirical CDF of the data

position estimate = $\mathrm{P}[\mathrm{X} \leq \boldsymbol{key}] * \#\boldsymbol{keys}$


Pos
$P[X \leq a] * \#keys$
a
Key

B-Trees are regression trees



BOSTON UNIVERSITY

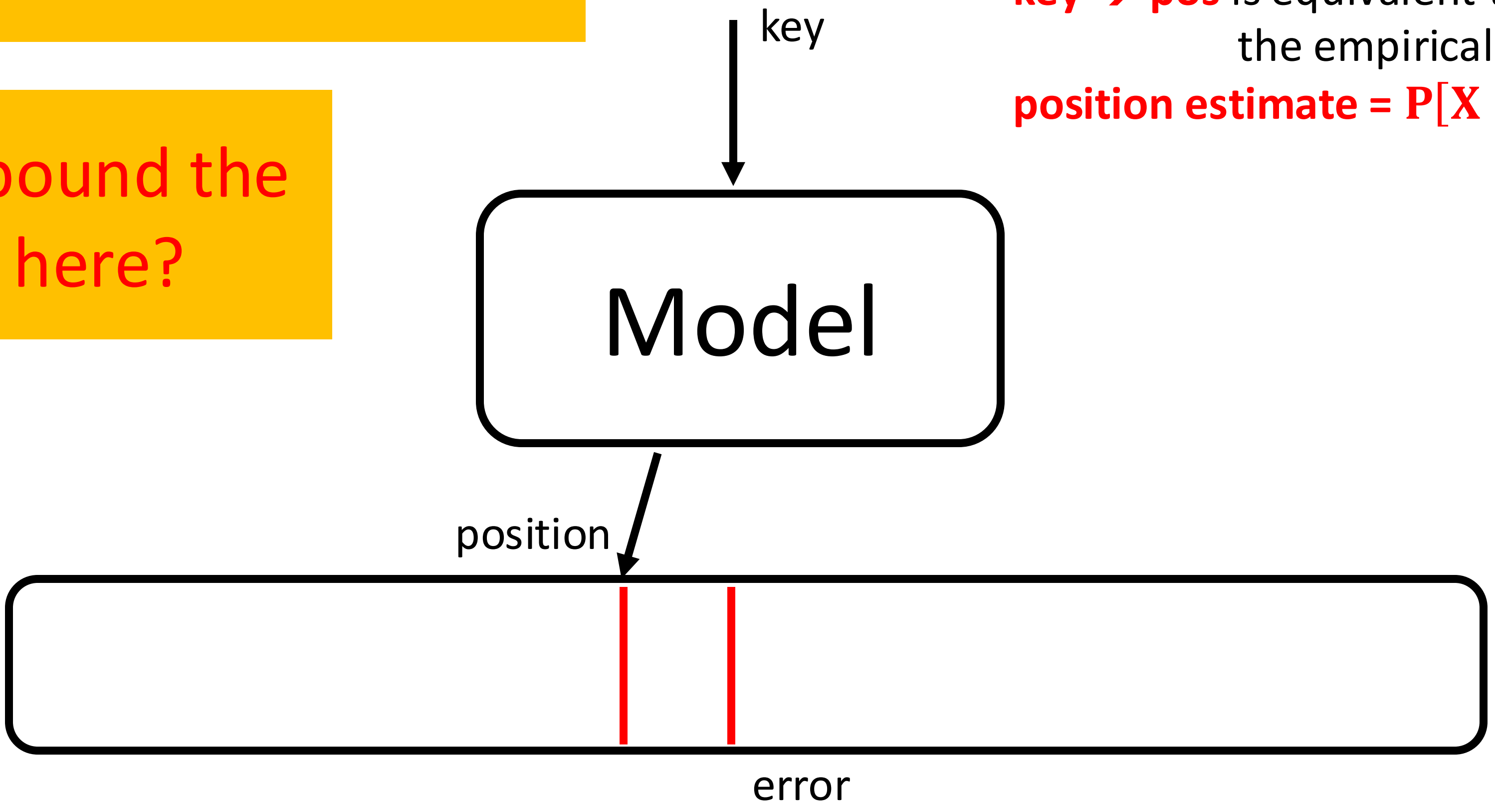
B-Trees is ***already*** a form of a learned index

What does this mean?

Learned Indexes

B-Trees have bounded error

Can we bound the error here?



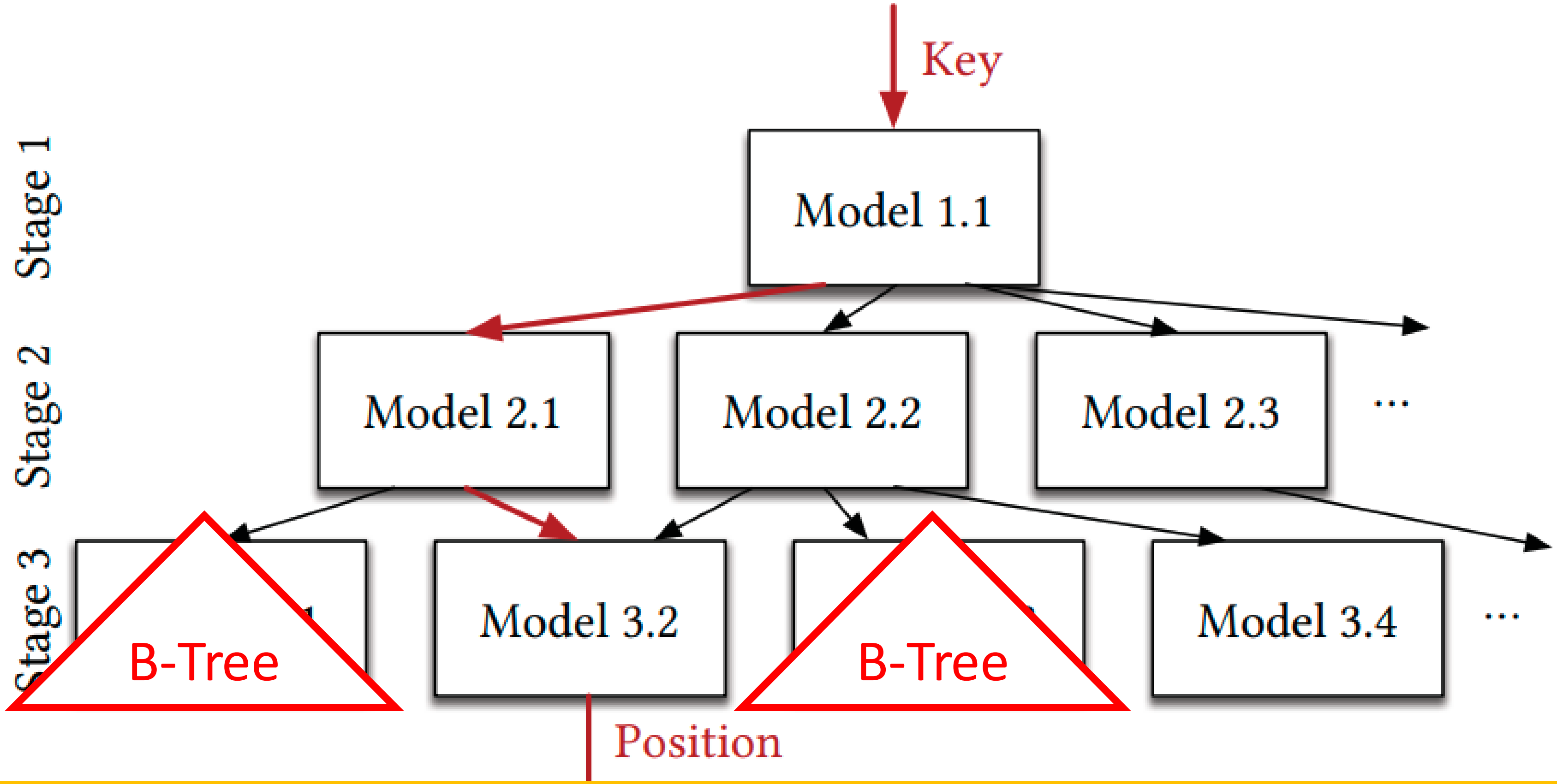
A form of regression model

key → pos is equivalent to modeling the empirical CDF of the data

position estimate = $\mathbf{P[X \leq \mathit{key}] * \#\mathit{keys}}$

What is the problem if we use an arbitrary model?

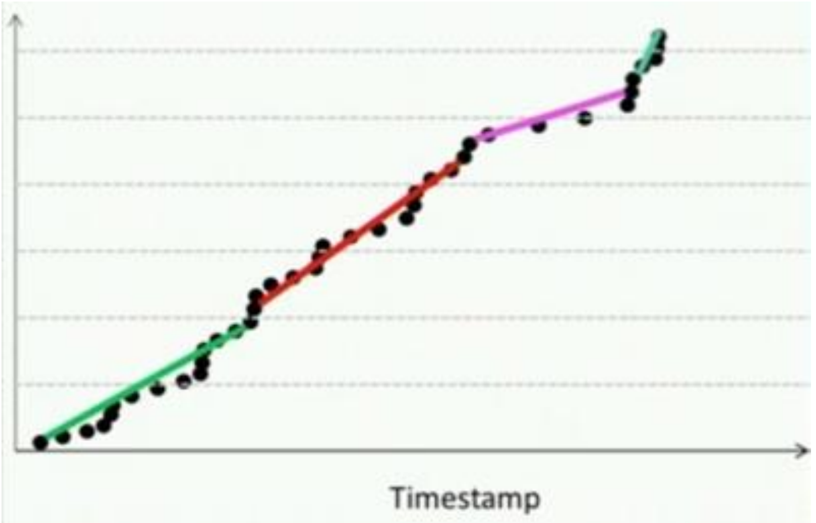
Last-mile indexing



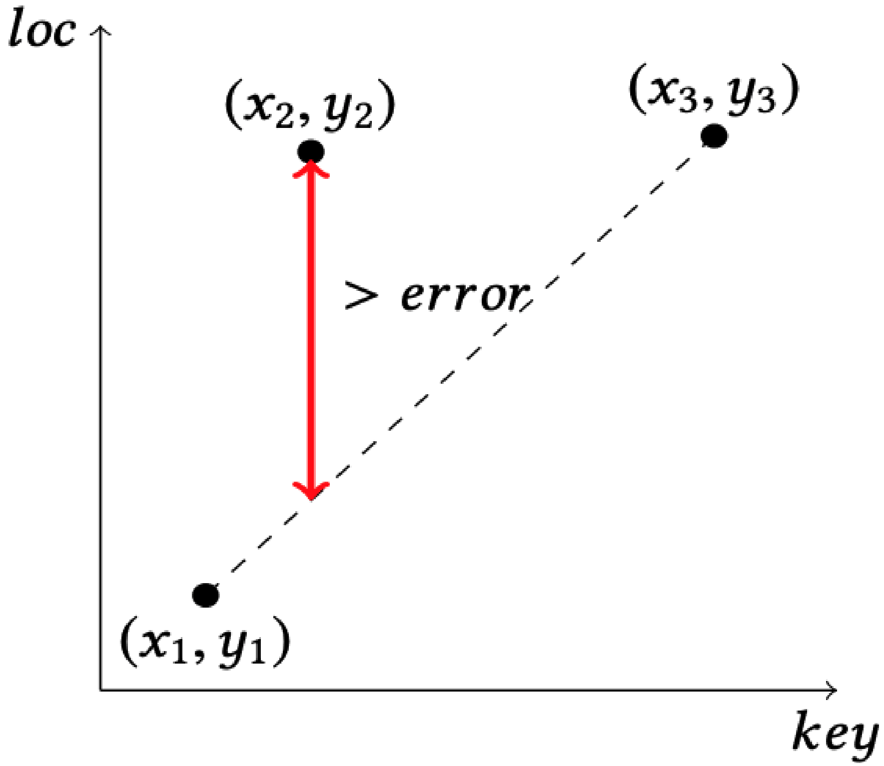
Some models can be replaced sub-B-Trees

Every level provides gain in accuracy

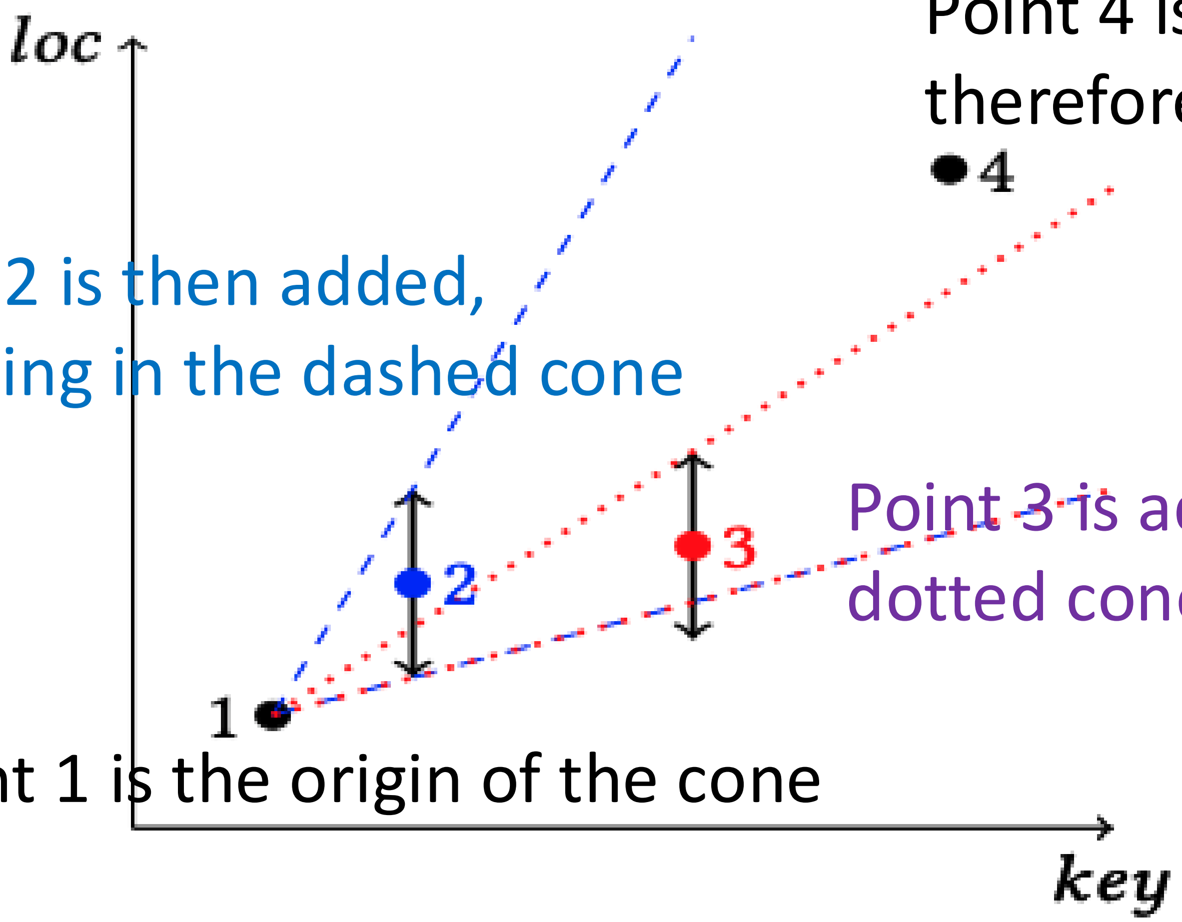
Use case: FITing-Tree



Piece-wise linear approximation



A segment from (x_1,y_1) to (x_3,y_3) is **not valid** if (x_2,y_2) is further than ***error*** from the interpolated line.



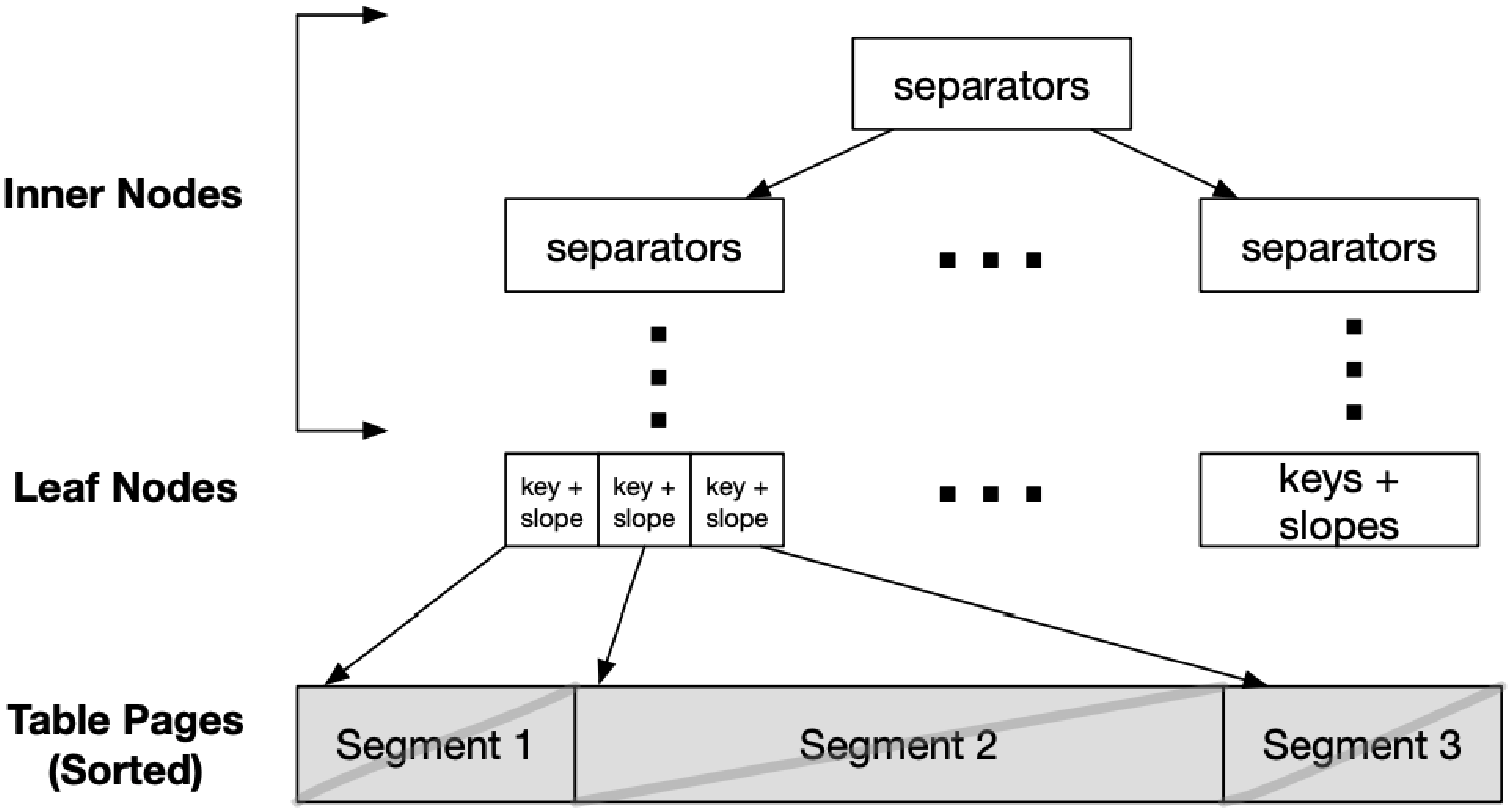
Point 4 is outside the dotted cone and therefore starts a new segment

Point 2 is then added, resulting in the dashed cone

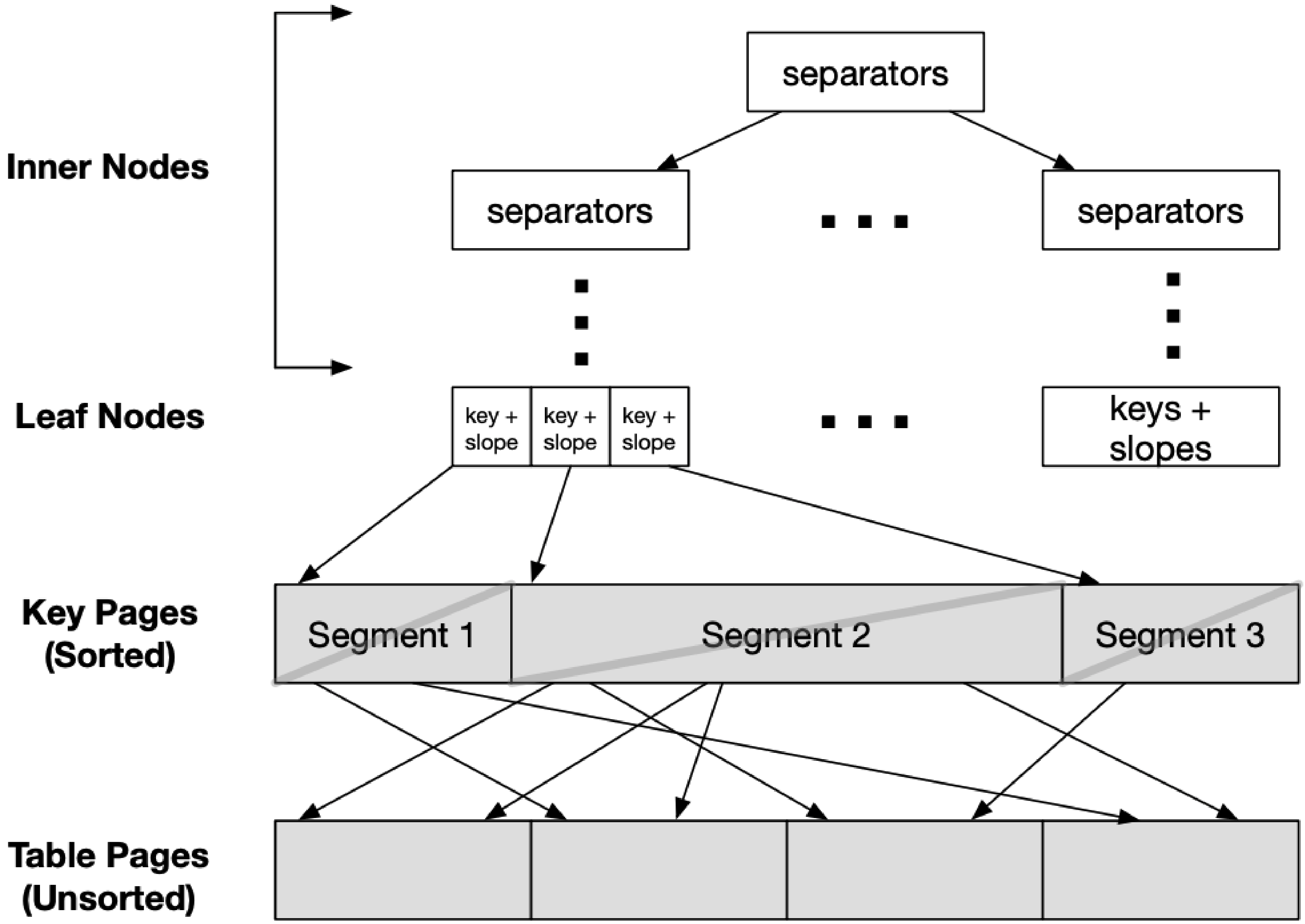
Point 3 is added next, yielding in the dotted cone

Point 1 is the origin of the cone

BOSTON UNIVERSITY



What if base data is not sorted?

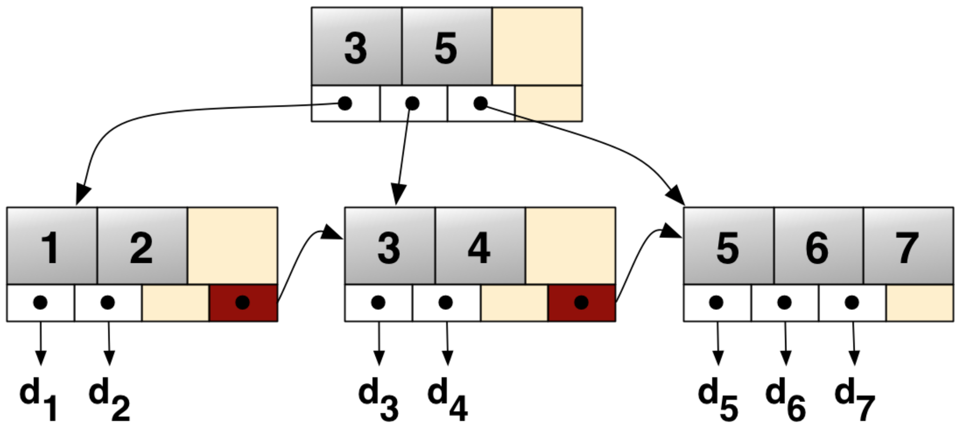


Need to materialize sorted data

What about updates and learned indexes?

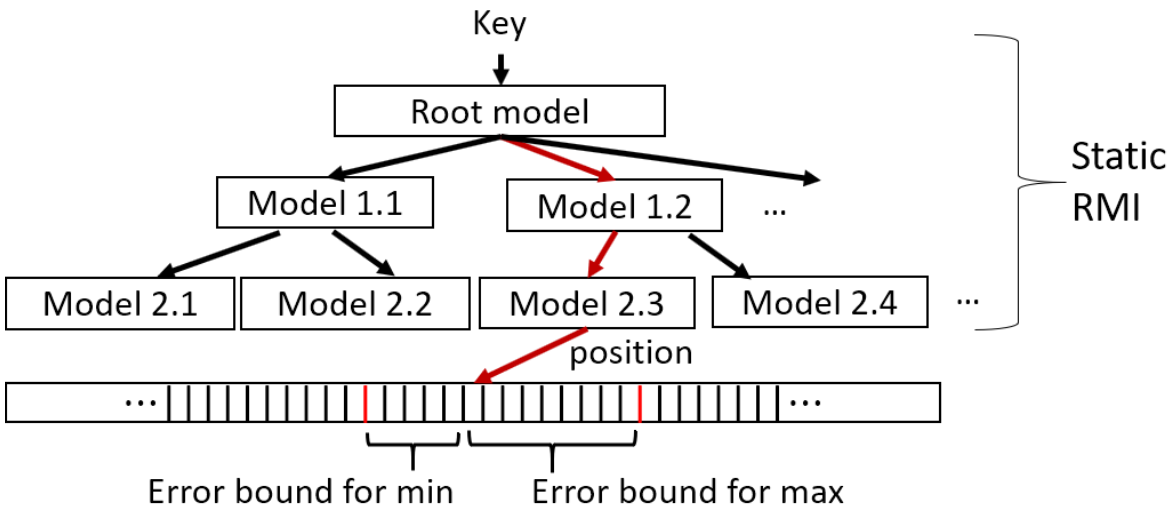
B+ Tree

- Traverses tree using comparisons
- Supports OLTP-style mixed workloads
  - Point lookups, range queries
  - Inserts, updates, deletes



Learned Index (Kraska et al., 2018)

- Traverses tree using computations (models)
- Supports point lookups and range queries
- Advantages: 3X faster reads, 10X smaller size
- Limitation: does not support writes



ALEX goals

	B+ Tree	Learned Index	ALEX
Lookup time	Slow	Fast	Faster
Insert time	Fast	Not Supported	Fast
Space usage	High	Low	Low

(every row should be read independently)

ALEX goals

	B+ Tree	Learned Index	ALEX
Lookup time	Slow	Fast	Faster
Insert time	Fast	Not Supported	Fast
Space usage	High	Low	Low

BOSTON UNIVERSITY

(every row should be read independently)

ALEX goals

	B+ Tree	Learned Index	ALEX
Lookup time	Slow	Fast	Faster
Insert time	Fast	Not Supported	Fast
Space usage	High	Low	Low

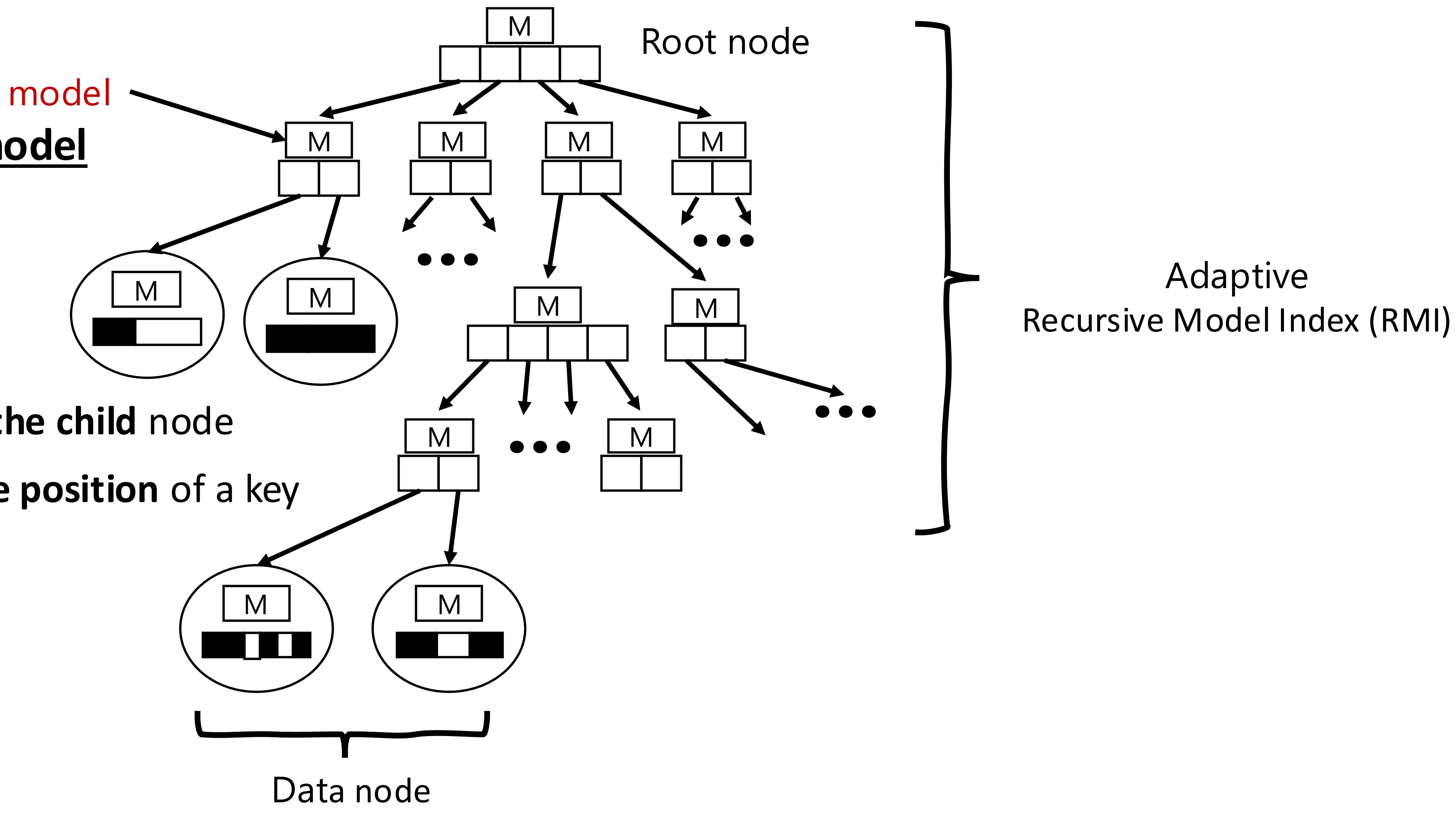
BOSTON UNIVERSITY

(every row should be read independently)

ALEX: An Updatable Learned Index

Dynamic tree structure

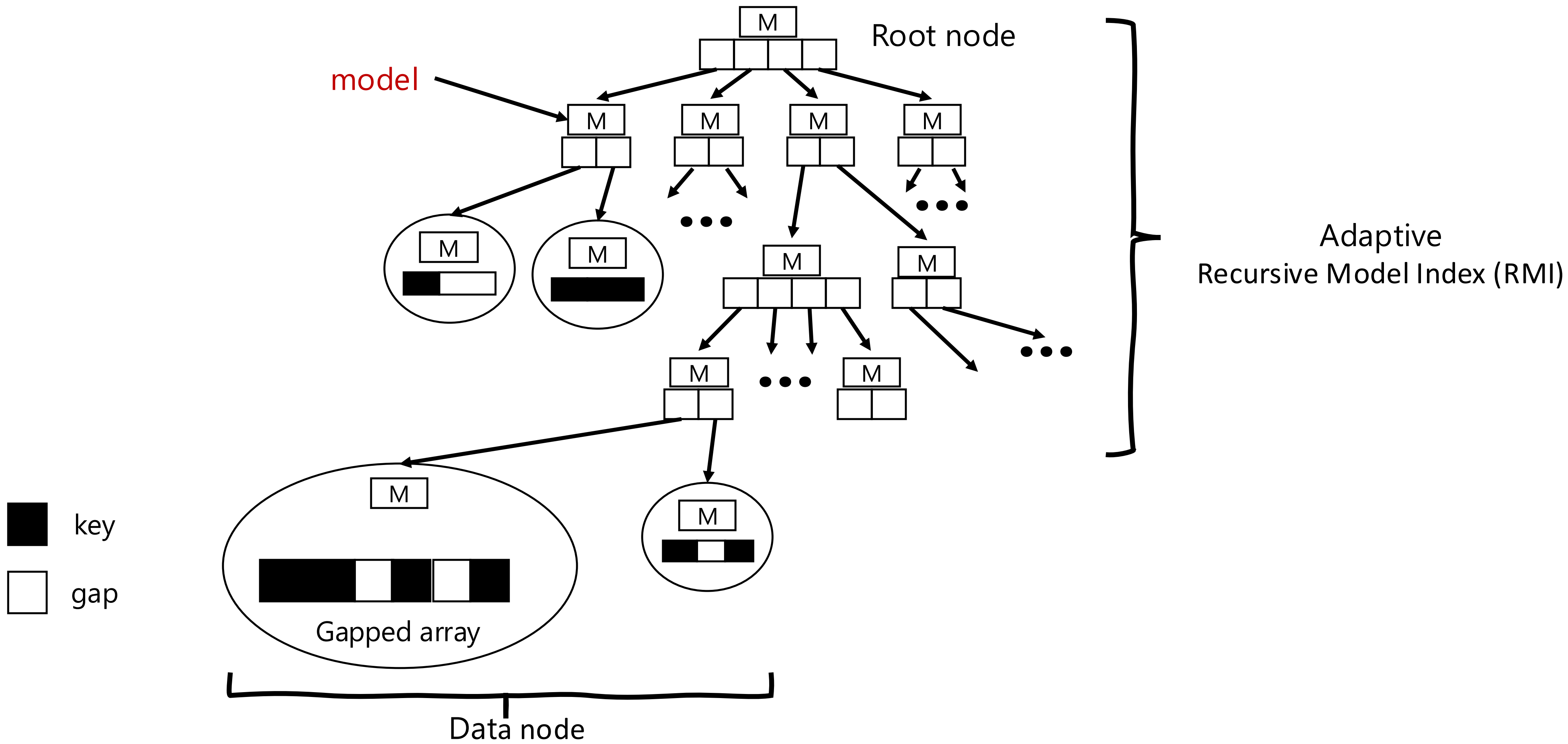
Each **node** contains a **<u>linear model</u>**



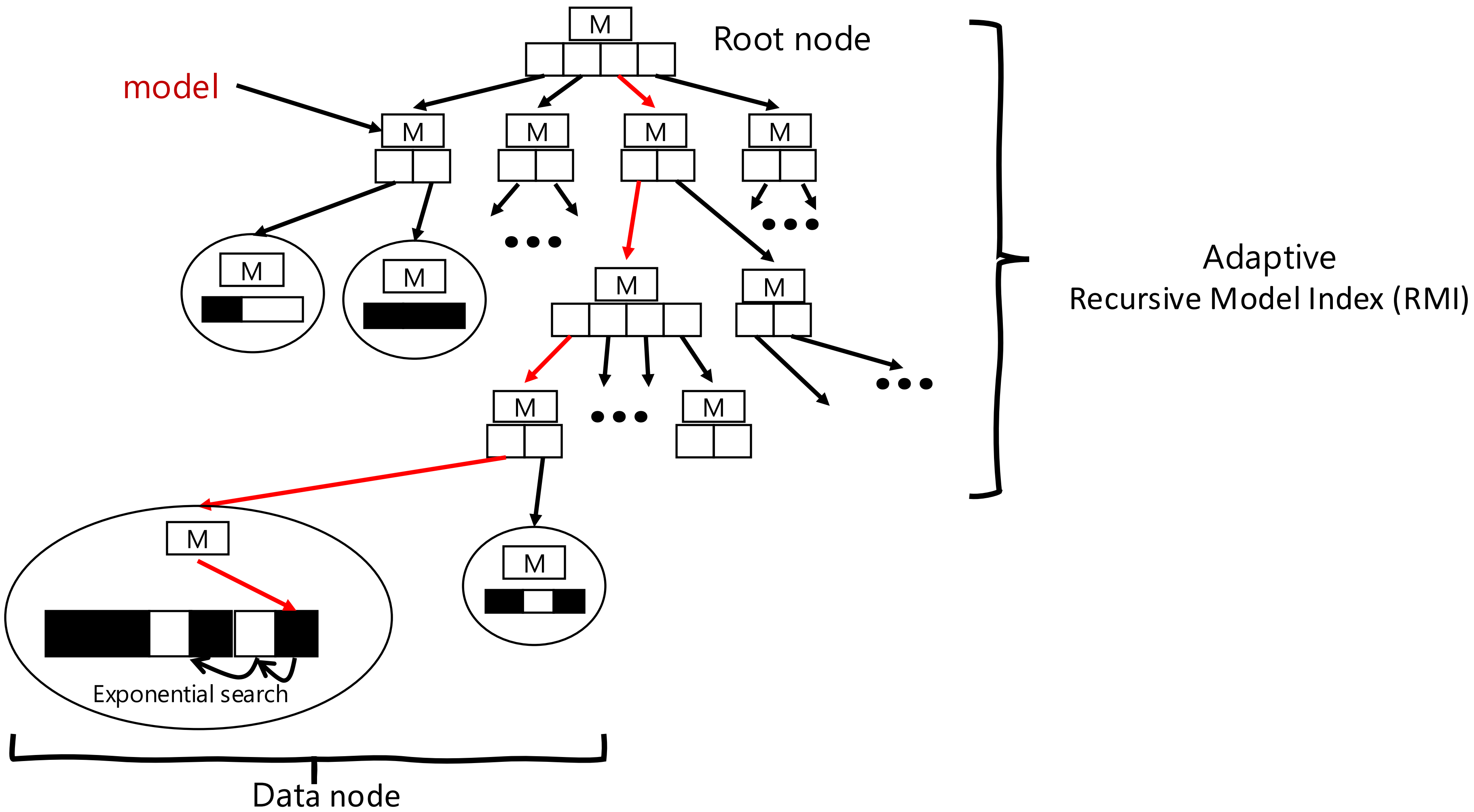
internal nodes → **models select the child** node

data nodes → **models predict the position** of a key

ALEX: An Updatable Learned Index

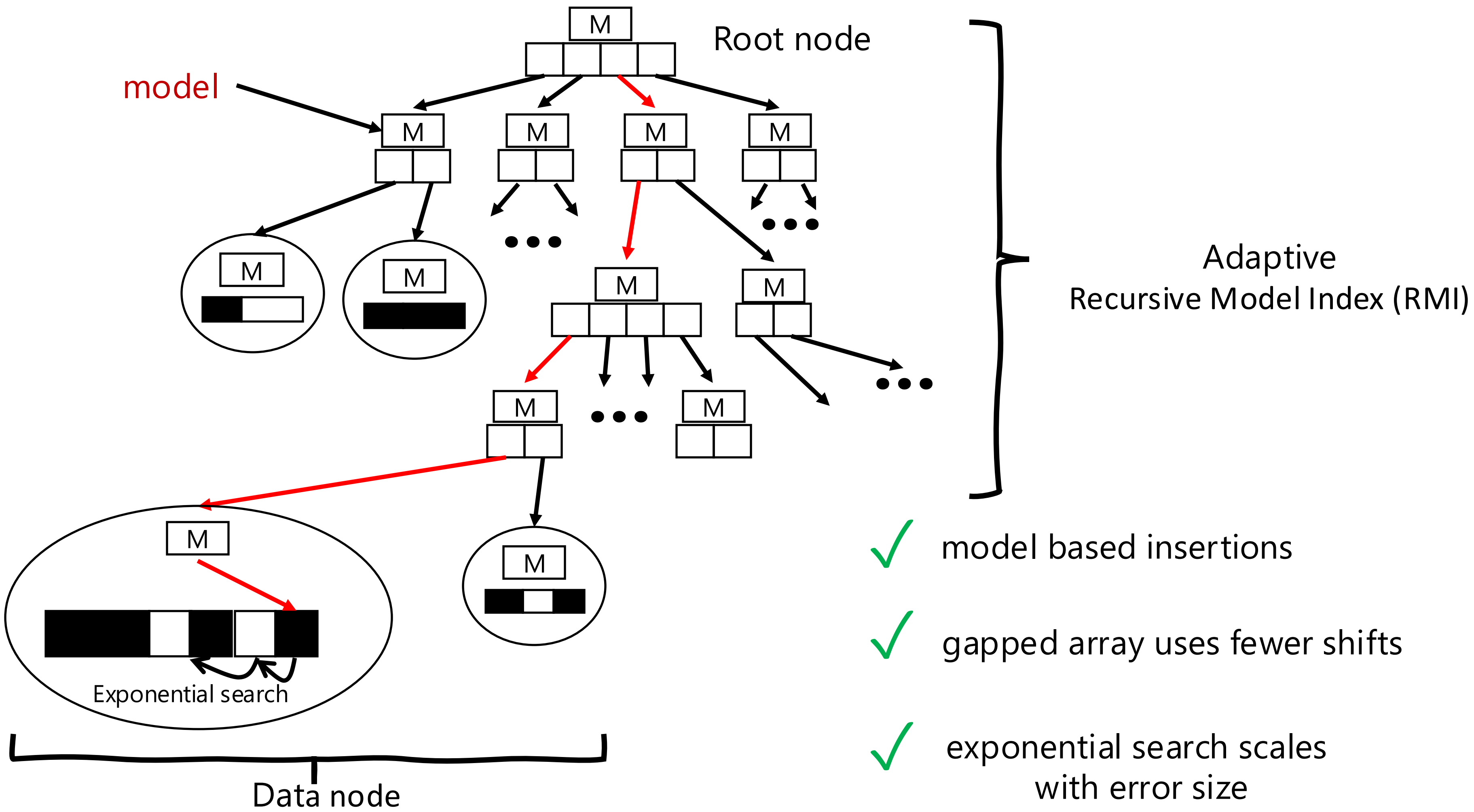


Lookups in ALEX



BOSTON UNIVERSITY

Insertions in ALEX

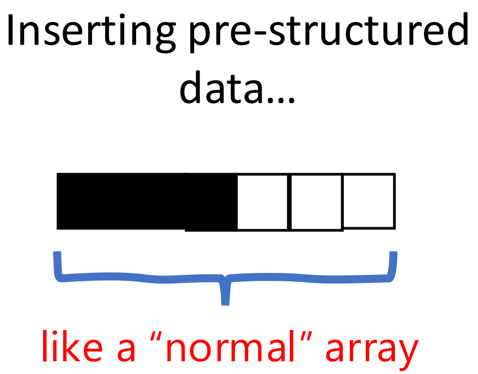


✓ model based insertions

✓ gapped array uses fewer shifts

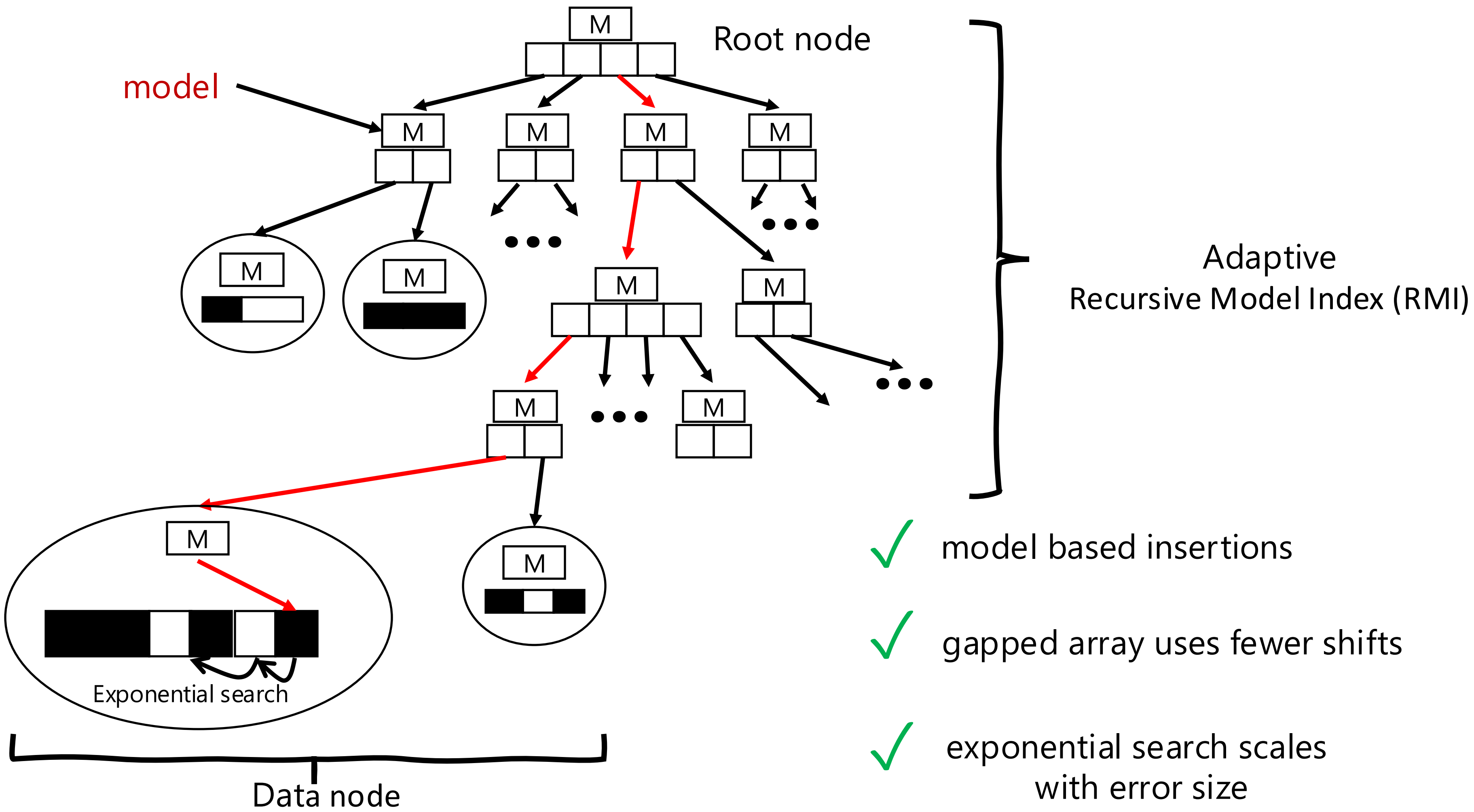
✓ exponential search scales with error size

Insertions in ALEX



gaps are infrequent

faster exp. Search!



✓ model based insertions

✓ gapped array uses fewer shifts

✓ exponential search scales with error size

ALEX Core Ideas

	Faster Reads	Faster Writes	Adaptiveness
1. Gapped Array		✓	
2. Model-based Inserts	✓		
3. Exponential Search	✓		
4. Adaptive Tree Structure	✓	✓	✓

BOSTON UNIVERSITY

1. Gapped Array

How should data be stored in data nodes?

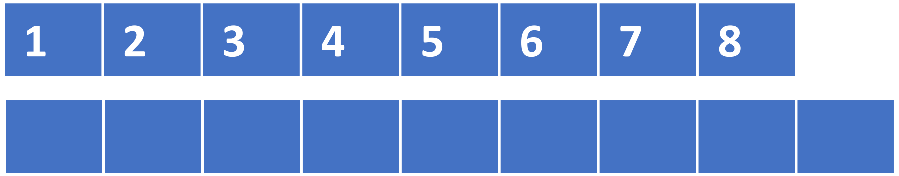
1. Gapped Array

Dense Array

1	2	3	4	5	6	7	8

1. Gapped Array

Dense Array



1. Gapped Array

Dense Array

1	2	3	4	5	6	7	8	
	1	2	3	4	5	6	7	8

1. Gapped Array

Dense Array

1	2	3	4	5	6	7	8	
0	1	2	3	4	5	6	7	8

1. Gapped Array

Dense Array	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

BOSTON UNIVERSITY

1. Gapped Array

Insertion Time

Dense Array	0	1	2	3	4	5	6	7	8	$O(n)$

BOSTON UNIVERSITY

1. Gapped Array

													Insertion Time
Dense Array	0	1	2	3	4	5	6	7	8				$O(n)$
B+ Tree Node	1	2	3	4	5	6	7	8					

BOSTON UNIVERSITY

1. Gapped Array

													Insertion Time
Dense Array	0	1	2	3	4	5	6	7	8				$O(n)$
B+ Tree Node		1	2	3	4	5	6	7	8				

1. Gapped Array

Insertion Time

Dense Array

0	1	2	3	4	5	6	7	8

$O(n)$

B+ Tree Node

0	1	2	3	4	5	6	7	8			

1. Gapped Array

													Insertion Time
Dense Array	0	1	2	3	4	5	6	7	8				$O(n)$
B+ Tree Node	0	1	2	3	4	5	6	7	8				$O(n)$

1. Gapped Array

													Insertion Time
Dense Array	0	1	2	3	4	5	6	7	8				$O(n)$
B+ Tree Node	0	1	2	3	4	5	6	7	8				$O(n)$
Gapped Array	1	2		3	4		5	6		7	8		

BOSTON UNIVERSITY

1. Gapped Array

													Insertion Time
Dense Array	0	1	2	3	4	5	6	7	8				$O(n)$
B+ Tree Node	0	1	2	3	4	5	6	7	8				$O(n)$
Gapped Array		1	2	3	4		5	6		7	8		

1. Gapped Array

													Insertion Time
Dense Array	0	1	2	3	4	5	6	7	8				$O(n)$
B+ Tree Node	0	1	2	3	4	5	6	7	8				$O(n)$
Gapped Array	0	1	2	3	4		5	6		7	8		

1. Gapped Array

													Insertion Time
Dense Array	0	1	2	3	4	5	6	7	8				$O(n)$
B+ Tree Node	0	1	2	3	4	5	6	7	8				$O(n)$
Gapped Array	0	1	2	3	4		5	6		7	8		$O(\log n)$

1. Gapped Array

													Insertion Time
Dense Array	0	1	2	3	4	5	6	7	8				$O(n)$
B+ Tree Node	0	1	2	3	4	5	6	7	8				$O(n)$
Gapped Array	0	1	2	3	4		5	6		7	8		$O(\log n)$

Storing data in Gapped Arrays achieves inserts using fewer shifts, leading to faster writes

BOSTON UNIVERSITY

1. Gapped Array

													Insertion Time
Dense Array	0	1	2	3	4	5	6	7	8				$O(n)$
B+ Tree Node	0	1	2	3	4	5	6	7	8				$O(n)$
Gapped Array	0	1	2	3	4	5	5	6	7	7	8	S	$O(\log n)$

Storing data in Gapped Arrays achieves inserts using fewer shifts, leading to faster writes

BOSTON UNIVERSITY

2. Model-based Inserts

Where do we put gaps in the Gapped Array?

BOSTON UNIVERSITY

2. Model-based Inserts

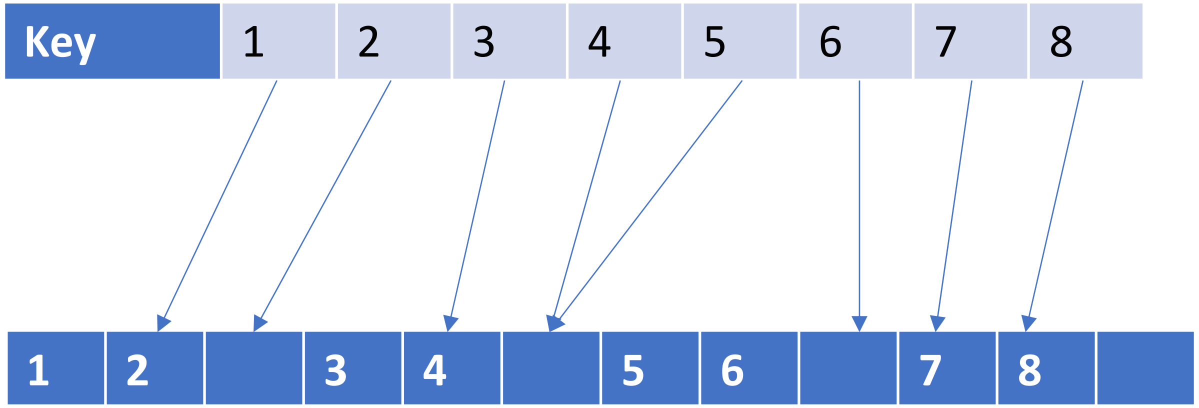
Gapped Array

1	2		3	4		5	6		7	8	

2. Model-based Inserts

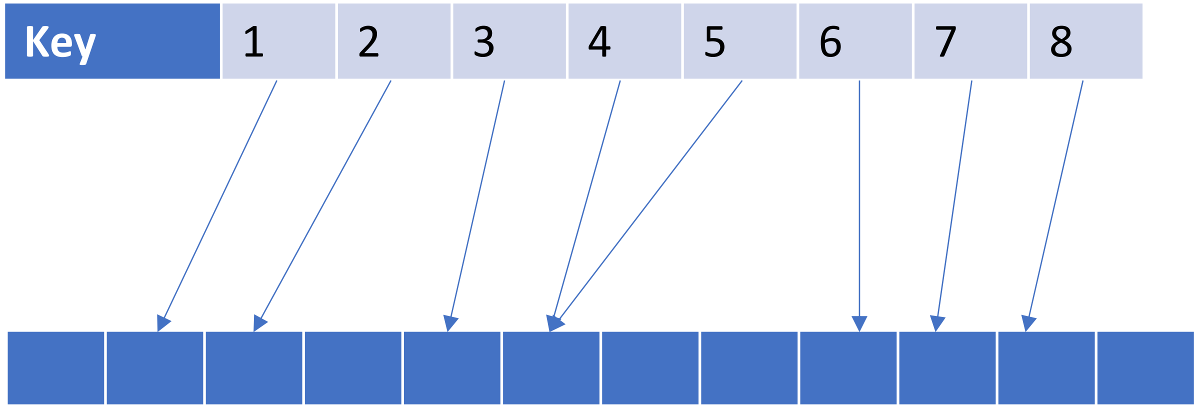
Model

Gapped Array



2. Model-based Inserts

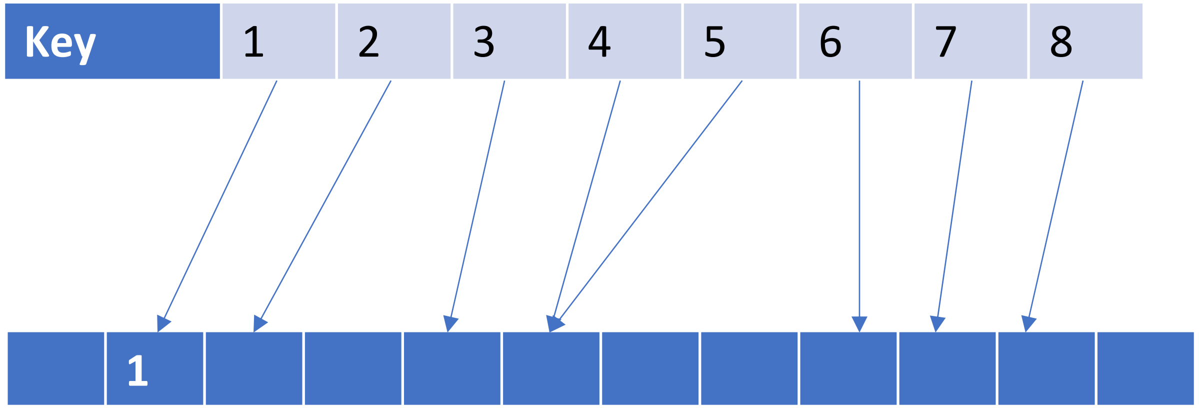
Model



Gapped Array

2. Model-based Inserts

Model

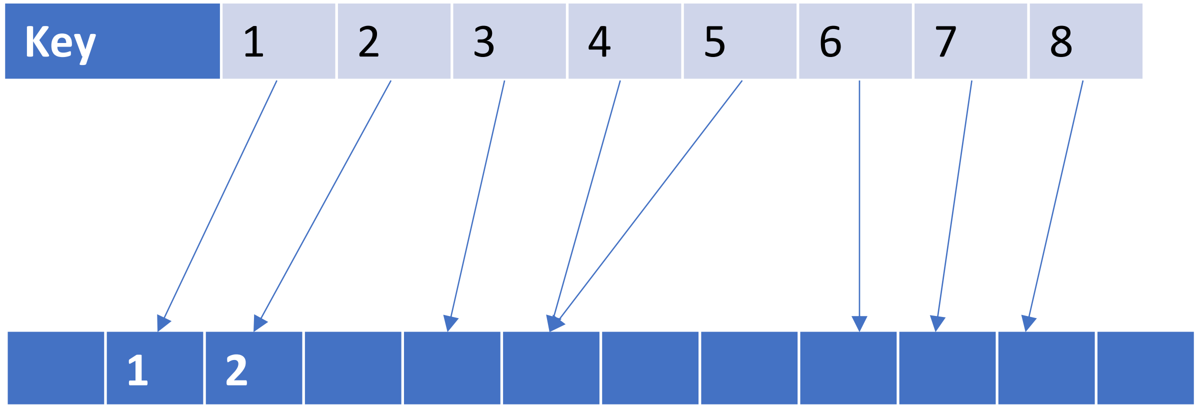


Gapped Array

2. Model-based Inserts

Model

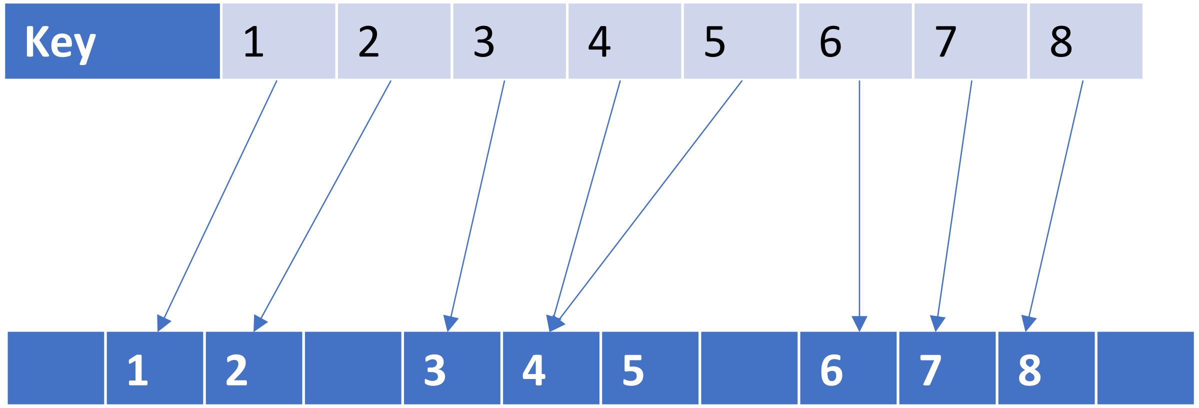
Gapped Array



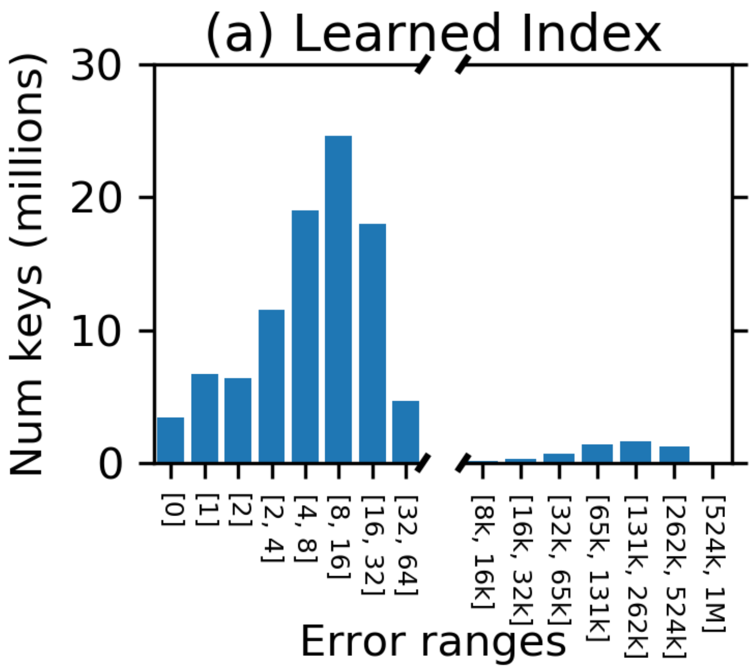
2. Model-based Inserts

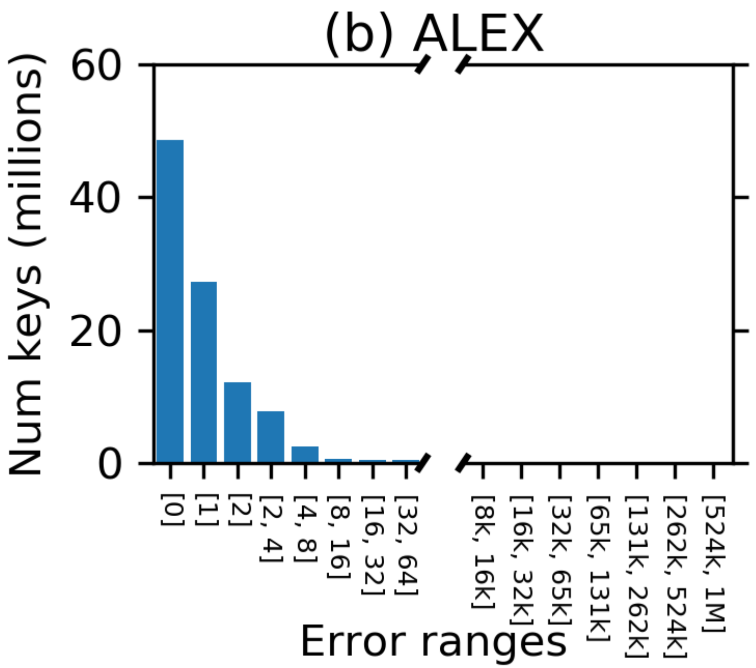
Model

Gapped Array



2. Model-based Inserts

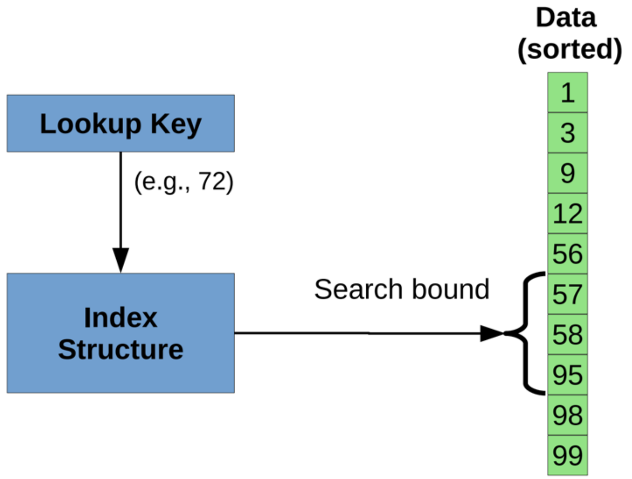




Model-based inserts achieve lower prediction error, leading to faster reads

BOSTON UNIVERSITY

3. Exponential Search



Can we do better than binary search?

BOSTON UNIVERSITY

Explanation: Exponential Search

1	2	3	5	6	7	8	13	15	17	18	21	22	23

```
int exponential_search(T arr[], int size, T key)
{
    if (size == 0) {
        return NOT_FOUND;
    }

    int bound = 1;
    while (bound < size && arr[bound] < key) {
        bound *= 2;
    }

    return binary_search(arr, key, bound/2, min(bound + 1, size));
}
```

BOSTON UNIVERSITY

Core algorithm: binary search!

Key difference: exp. increasing search bound

Explanation: Exponential Search

1	2	3	5	6	7	8	13	15	17	18	21	22	23

search for 3

```
int exponential_search(T arr[], int size, T key)
{
    if (size == 0) {
        return NOT_FOUND;
    }

    int bound = 1;
    while (bound < size && arr[bound] < key) {
        bound *= 2;
    }

    return binary_search(arr, key, bound/2, min(bound + 1, size));
}
```

BOSTON UNIVERSITY

Explanation: Exponential Search

1	2	3	5	6	7	8	13	15	17	18	21	22	23

search for 3

```
int exponential_search(T arr[], int size, T key)
{
    if (size == 0) {
        return NOT_FOUND;
    }

    int bound = 1;
    while (bound < size && arr[bound] < key) {
        bound *= 2;
    }

    return binary_search(arr, key, bound/2, min(bound + 1, size));
}
```

Explanation: Exponential Search

1	2	3	5	6	7	8	13	15	17	18	21	22	23

search for 3

```
int exponential_search(T arr[], int size, T key)
{
    if (size == 0) {
        return NOT_FOUND;
    }

    int bound = 1;
    while (bound < size && arr[bound] < key) {
        bound *= 2;
    }

    return binary_search(arr, key, bound/2, min(bound + 1, size));
}
```

BOSTON UNIVERSITY

Explanation: Exponential Search

1	2	3	5	6	7	8	13	15	17	18	21	22	23

search for 3

```
int exponential_search(T arr[], int size, T key)
{
    if (size == 0) {
        return NOT_FOUND;
    }

    int bound = 1;
    while (bound < size && arr[bound] < key) {
        bound *= 2;
    }

    return binary_search(arr, key, bound/2, min(bound + 1, size));
}
```

BOSTON UNIVERSITY

Explanation: Exponential Search

1	2	3	5	6	7	8	13	15	17	18	21	22	23

search for 22

```
int exponential_search(T arr[], int size, T key)
{
    if (size == 0) {
        return NOT_FOUND;
    }

    int bound = 1;
    while (bound < size && arr[bound] < key) {
        bound *= 2;
    }

    return binary_search(arr, key, bound/2, min(bound + 1, size));
}
```

BOSTON UNIVERSITY

Explanation: Exponential Search

1	2	3	5	6	7	8	13	15	17	18	21	22	23

search for 22

```
int exponential_search(T arr[], int size, T key)
{
    if (size == 0) {
        return NOT_FOUND;
    }

    int bound = 1;
    while (bound < size && arr[bound] < key) {
        bound *= 2;
    }

    return binary_search(arr, key, bound/2, min(bound + 1, size));
}
```

BOSTON UNIVERSITY

Explanation: Exponential Search

1	2	3	5	6	7	8	13	15	17	18	21	22	23

search for 22

```
int exponential_search(T arr[], int size, T key)
{
    if (size == 0) {
        return NOT_FOUND;
    }

    int bound = 1;
    while (bound < size && arr[bound] < key) {
        bound *= 2;
    }

    return binary_search(arr, key, bound/2, min(bound + 1, size));
}
```

Explanation: Exponential Search

1	2	3	5	6	7	8	13	15	17	18	21	22	23

search for 22

```
int exponential_search(T arr[], int size, T key)
{
    if (size == 0) {
        return NOT_FOUND;
    }

    int bound = 1;
    while (bound < size && arr[bound] < key) {
        bound *= 2;
    }

    return binary_search(arr, key, bound/2, min(bound + 1, size));
}
```

Explanation: Exponential Search

1	2	3	5	6	7	8	13	15	17	18	21	22	23

search for 22

```
int exponential_search(T arr[], int size, T key)
{
    if (size == 0) {
        return NOT_FOUND;
    }

    int bound = 1;
    while (bound < size && arr[bound] < key) {
        bound *= 2;
    }

    return binary_search(arr, key, bound/2, min(bound + 1, size));
}
```

Explanation: Exponential Search

1	2	3	5	6	7	8	13	15	17	18	21	22	23

search for 22

```
int exponential_search(T arr[], int size, T key)
{
    if (size == 0) {
        return NOT_FOUND;
    }

    int bound = 1;
    while (bound < size && arr[bound] < key) {
        bound *= 2;
    }

    return binary_search(arr, key, bound/2, min(bound + 1, size));
}
```

BOSTON UNIVERSITY

Explanation: Exponential Search

1	2	3	5	6	7	8	13	15	17	18	21	22	23

```
int exponential_search(T arr[], int size, T key)
{
    if (size == 0) {
        return NOT_FOUND;
    }
```

We begin our search from the "predicted" location, ***low error expected***!

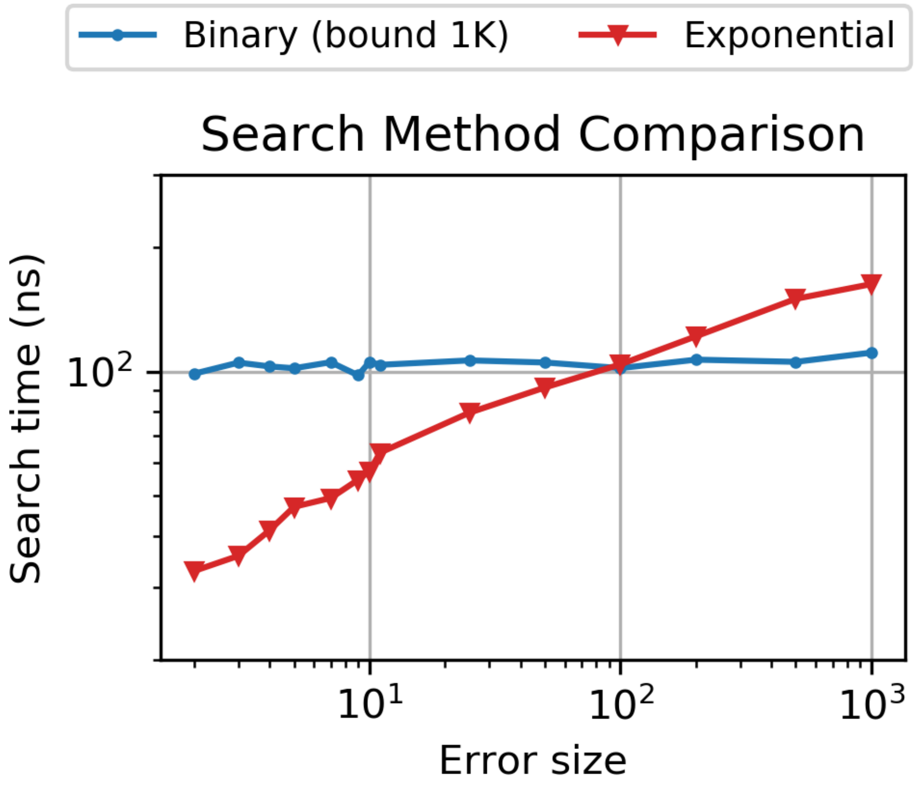
Why is this helpful in our case?

```
}
```

Exp. Search is ***ideal*** for a **search key at the beginning** of the array!

BOSTON UNIVERSITY

3. Exponential Search



Model errors are low, so exponential search is faster than binary search

BOSTON UNIVERSITY

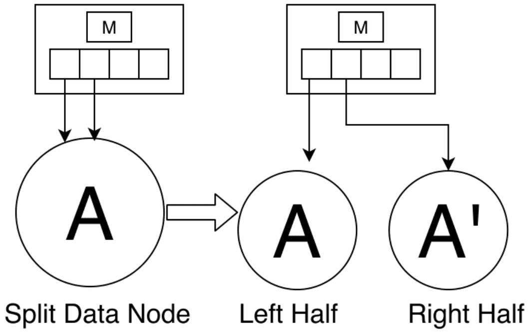
4. Adaptive Structure

What happens if data nodes become full?

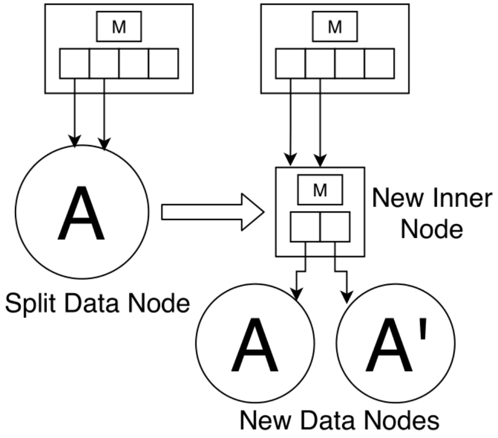
What happens if models become inaccurate?

4. Adaptive Structure

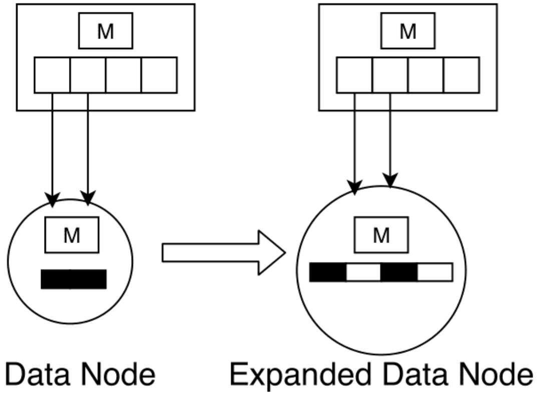
- Flexible tree structure
  - Split nodes sideways
  - Split nodes downwards
  - Expand nodes
  - Merge nodes, contract nodes
- Key idea: all decisions are made to maximize performance
  - Use cost model of query runtime
  - No hand-tuning
  - Robust to data and workload shifts



(a)

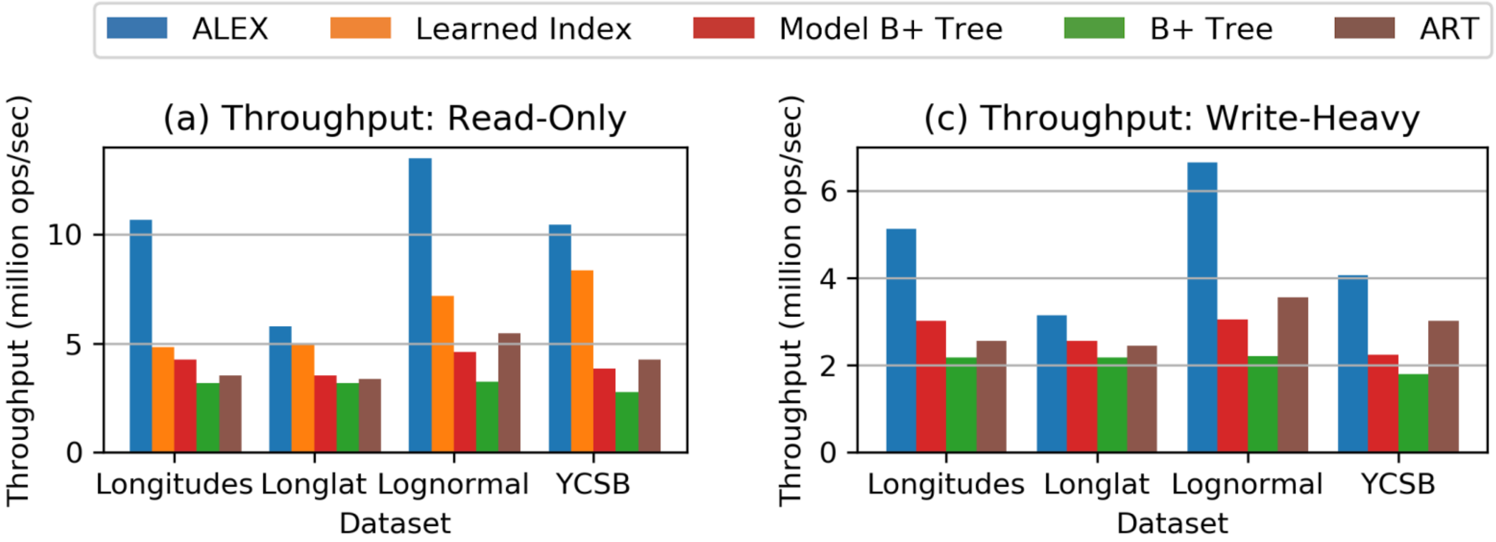


(c)



Results

- High-level results
  - Fast reads
  - Fast writes

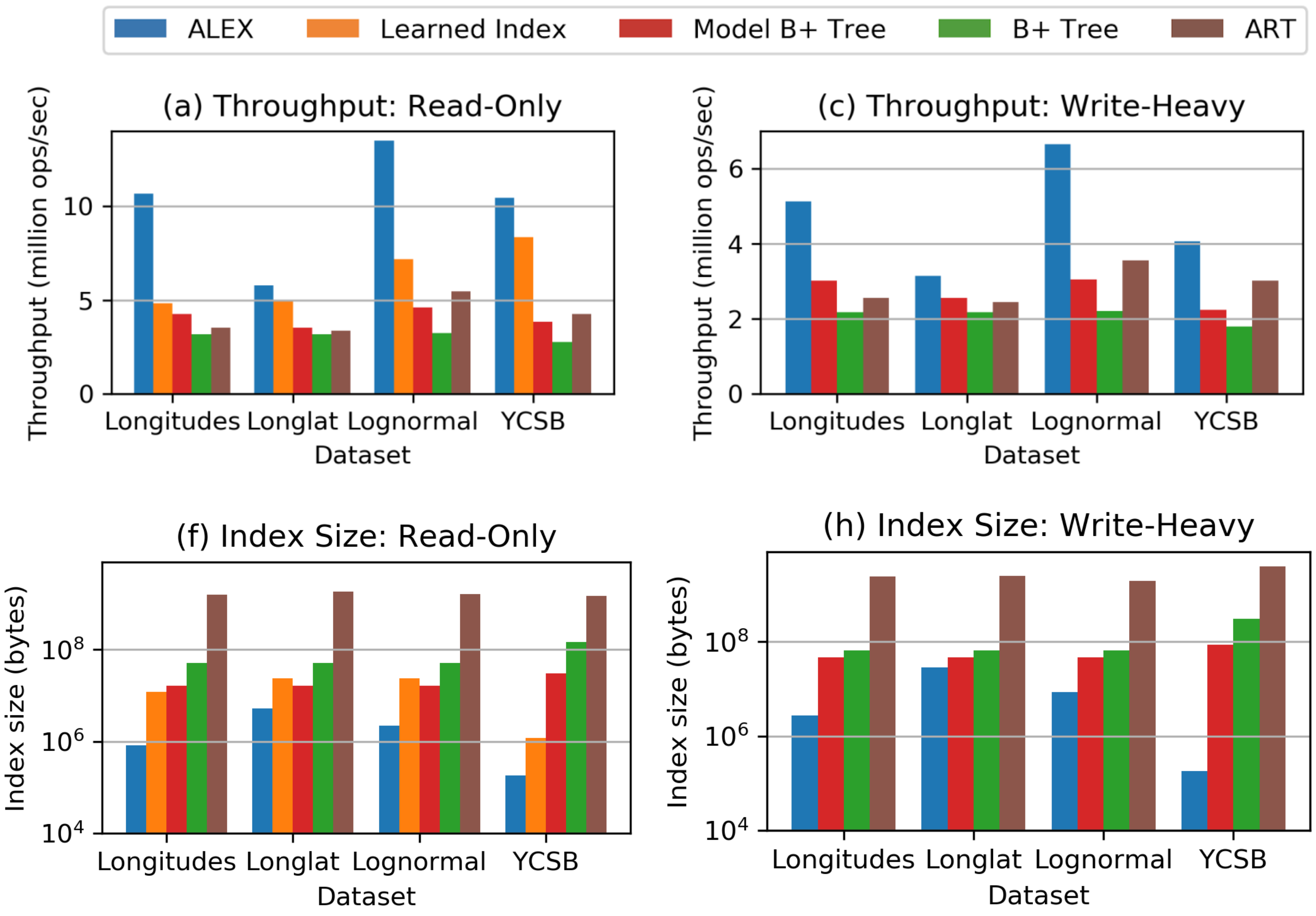


~4x faster than B+ Tree
~2x faster than Learned Index

~2-3x faster than B+ Tree

Results

- High-level results
  - Fast reads
  - Fast writes
  - Smaller index size
- Other results
  - Efficient bulk loading
  - Scales
  - Robust to data and workload shift



~3 orders of magnitude less space for index

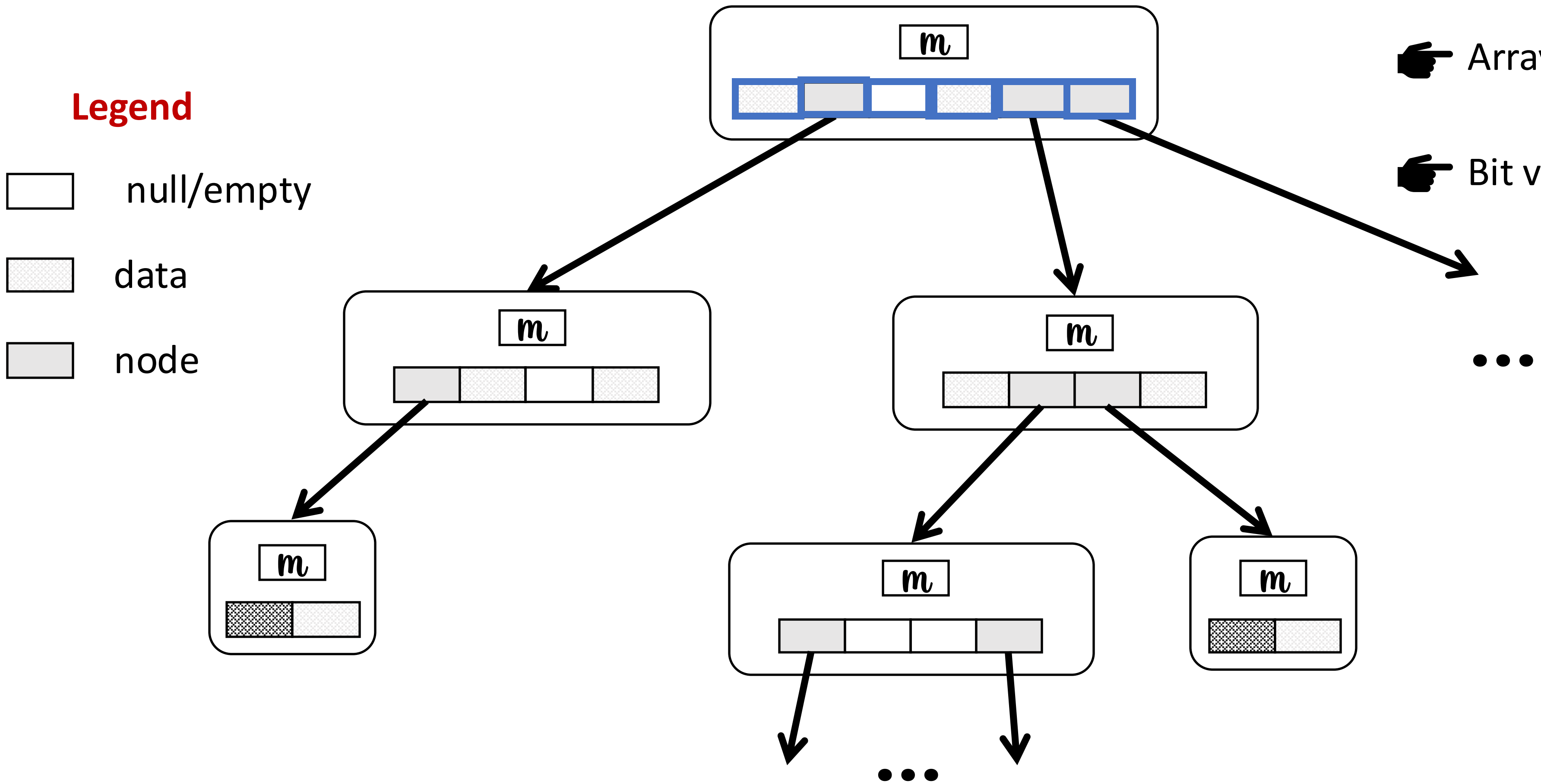
ALEX Summary

- Combines the best of B+ Tree and Learned Indexes
  - Supports OLTP-style mixed workloads
    - Point lookups, range queries
    - Inserts, updates, deletes
  - Up to 4X faster, 2000X smaller than B+ Tree
- Current research
  - String keys
  - Concurrency
  - Persistence

	Faster Reads	Faster Writes	Adaptiveness
Gapped Array		✓	
Model-based Inserts	✓		
Exponential Search	✓		
Adaptive Tree Structure	✓	✓	✓

github.com/microsoft/ALEX

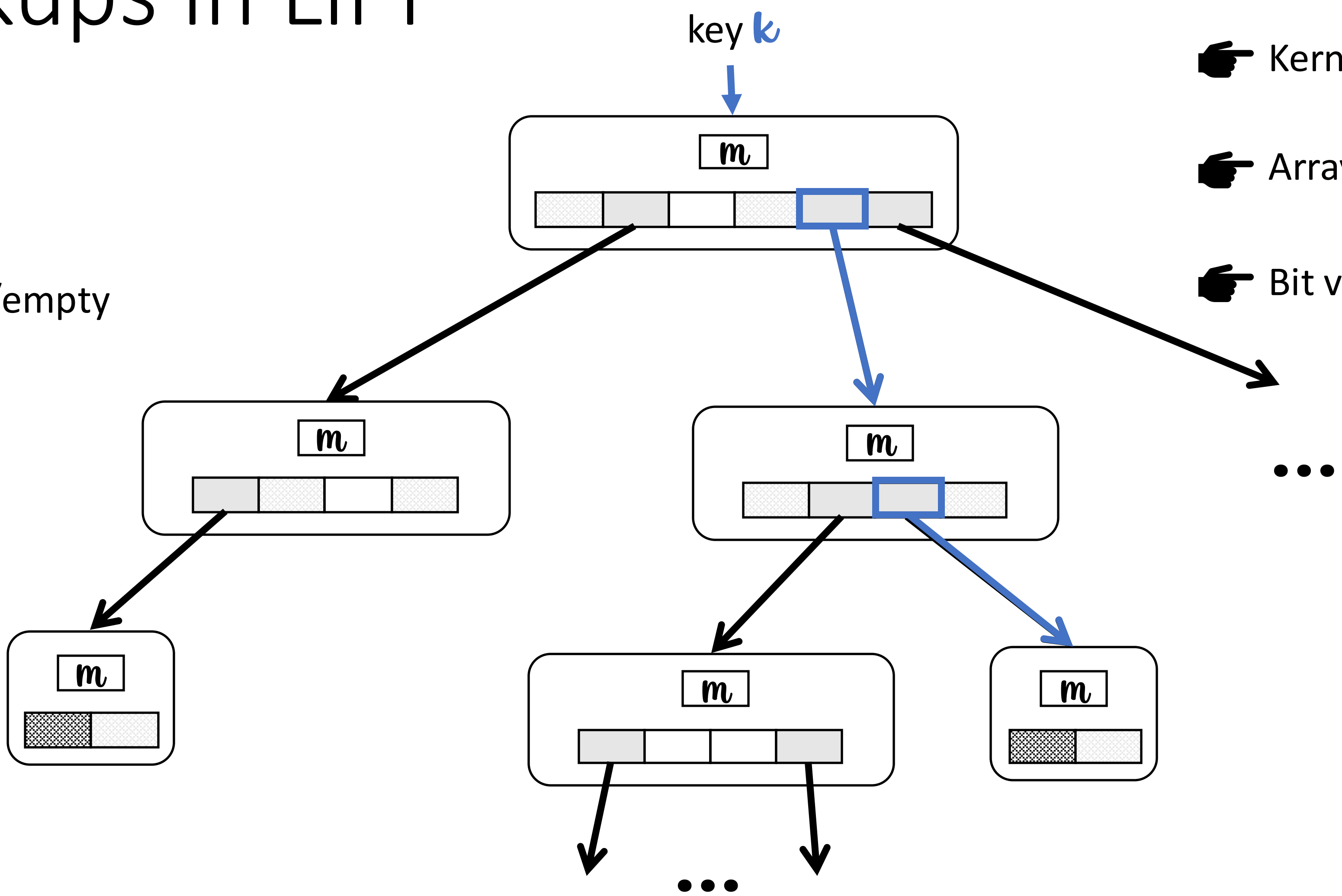
LIPP: An Updatable Learned Index with Precise Positions



- Kernalized linear model m
- Array of entries ε
- Bit vector of entry types

BOSTON UNIVERSITY

Lookups in LIPP



Legend

null/empty

data

node

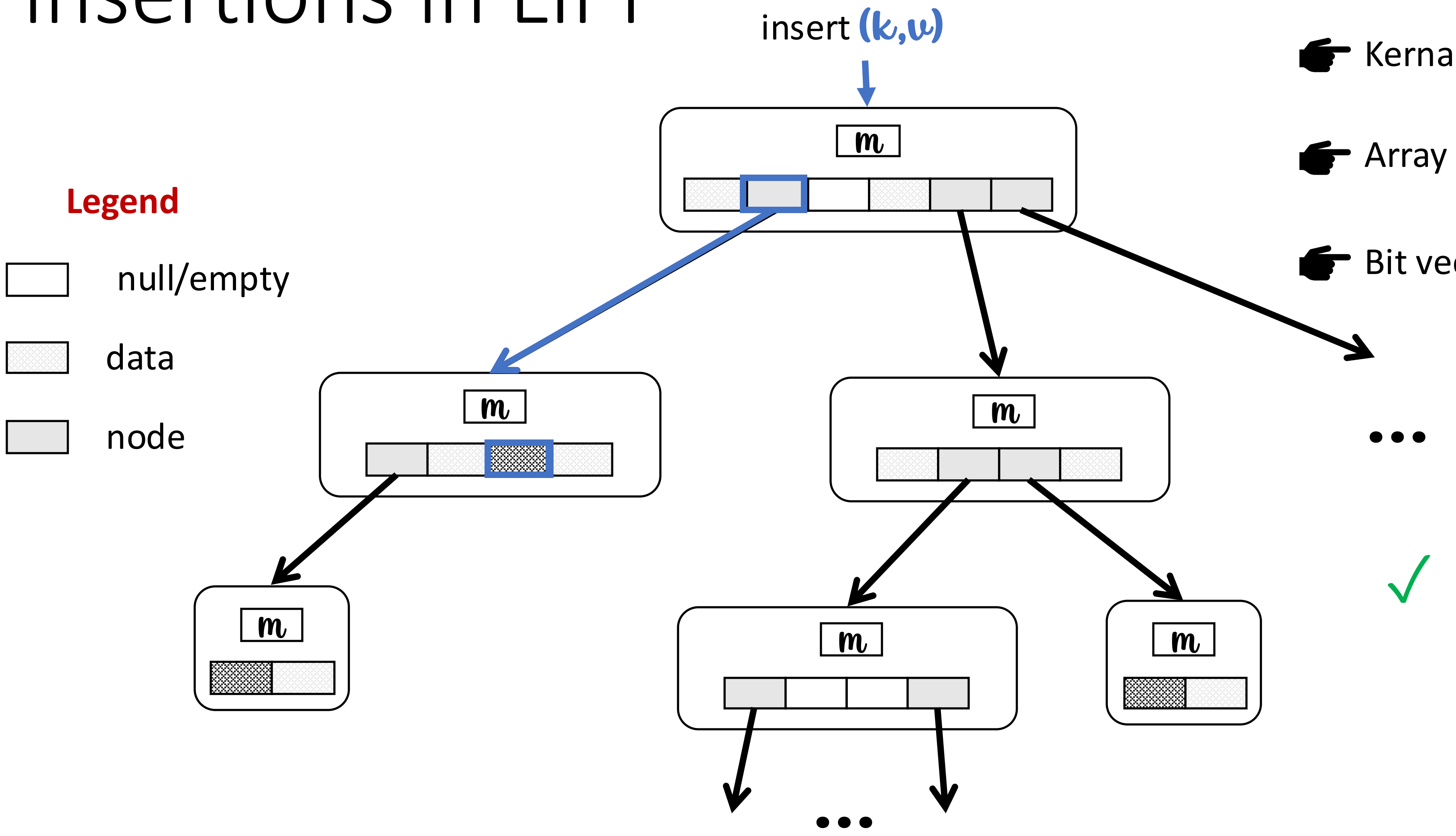
Kernalized linear model m

Array of entries ε

Bit vector of entry types

BOSTON UNIVERSITY

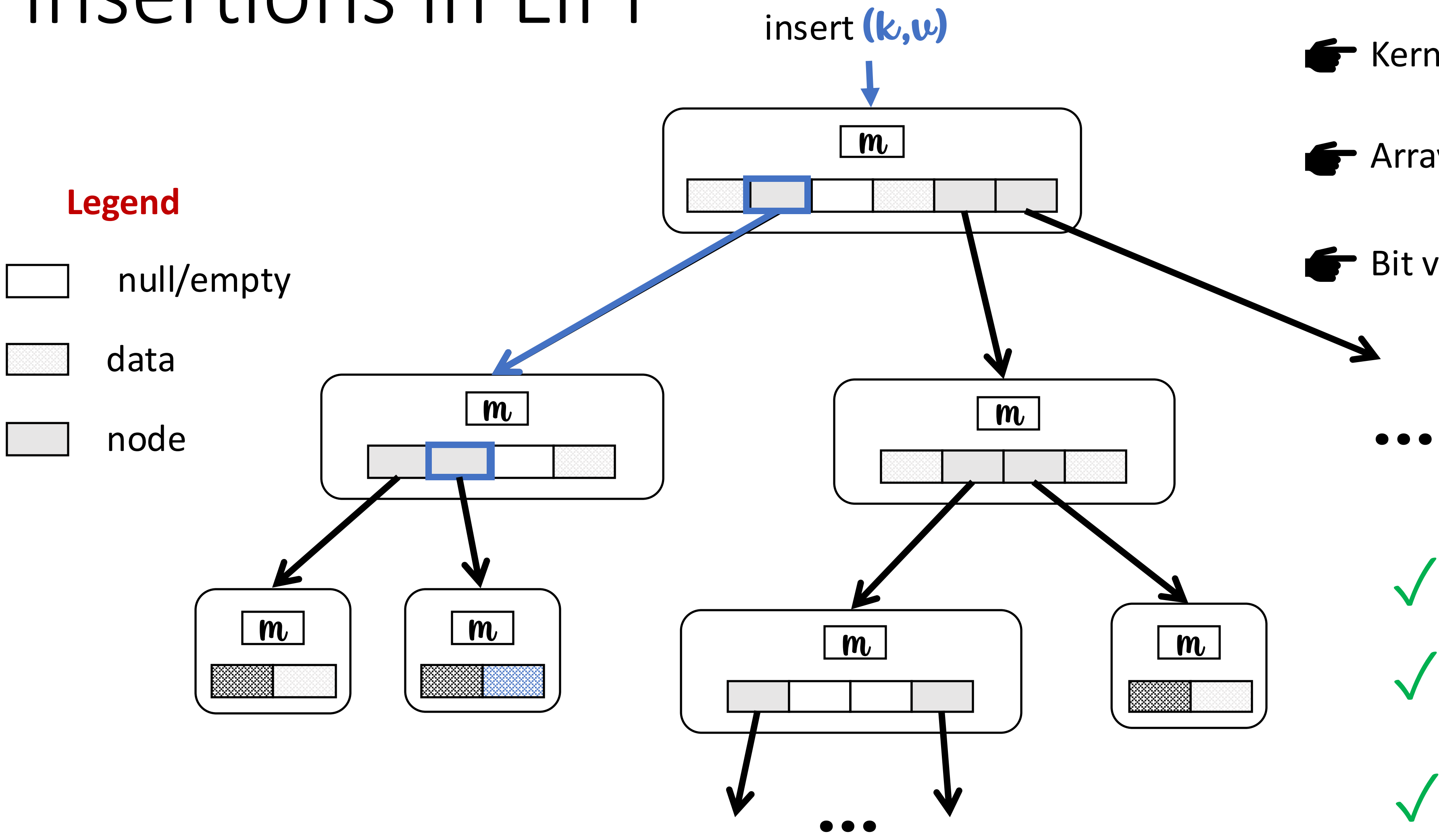
Insertions in LIPP



- Kernalized linear model m
- Array of entries ε
- Bit vector of entry types

✓ model based insertions

Insertions in LIPP



- Kernalized linear model m
- Array of entries ε
- Bit vector of entry types

✓ model based insertions

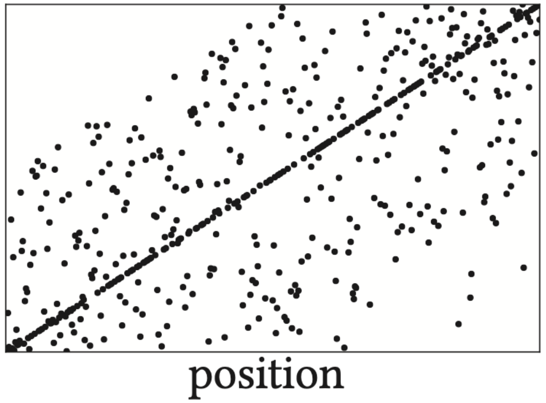
✓ create smaller nodes for conflicts

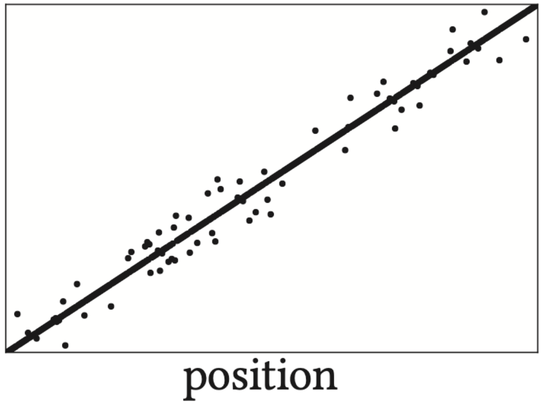
✓ no shifting necessary

Learned Indexes are meant to exploit data properties!

How about data sortedness?

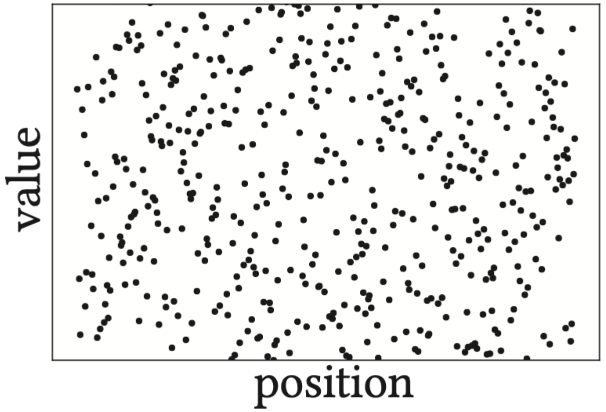
Reminder! Classical Indexes …





Near-sorted data

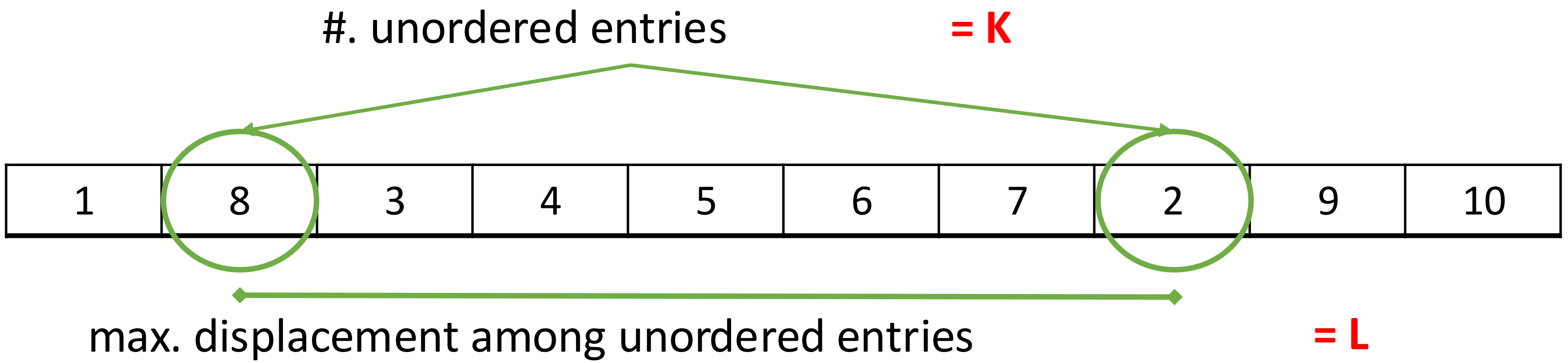
$\approxeq$



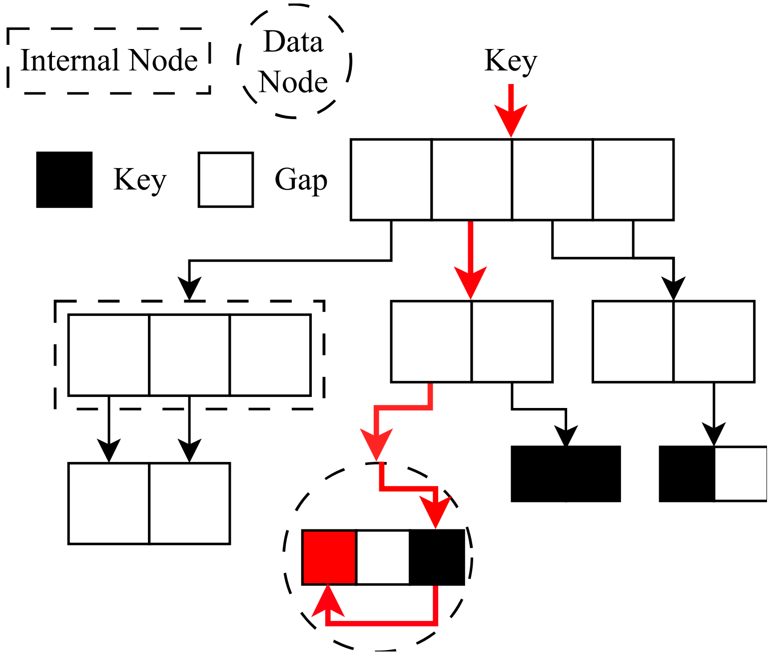
treated same as unstructured data!

Sortedness Metric?

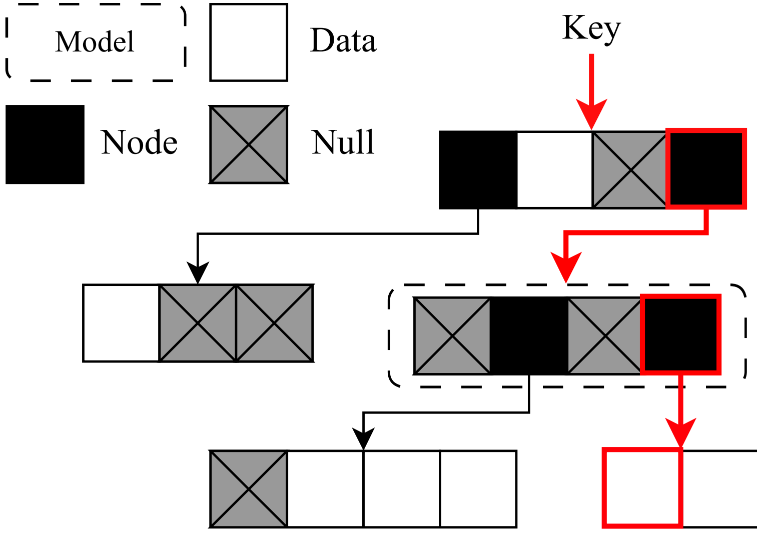
[BenMoshe, ICDT 2011]



Can Learned Indexes Capture Sortedness?

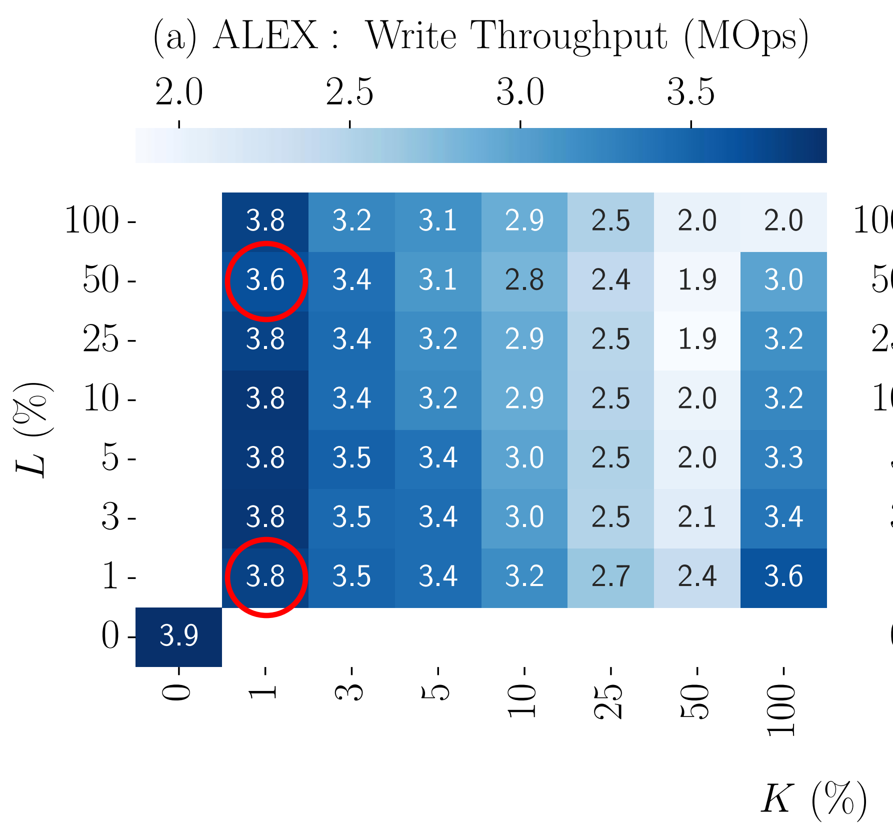


ALEX



LIPP

Learned Indexes are Unpredictable!



LIPP fails to sequentially write fully sorted data!

ALEX v/s LIPP:
LIPP can be anywhere between 4.4x faster t

Learned Indexes

Replace data structure with **learned models**

- ✓ Simple approaches like linear approximation work well
- ✓ Empty space for updates
- ✓ Error bounds to split model nodes
- ✓ Exponential search for last-mile searching

- ➢ A very fertile area of research!
- ➢ A comprehensive list of papers: http://dsg.csail.mit.edu/mlforsystems/papers/#learned-range-indexes

BOSTON UNIVERSITY

CS 561: Data Systems Architectures

class 24

Learned Indexes

Prof. Manos Athanassoulis

https://bu-disc.github.io/CS561/