BOSTON UNIVERSITY

CS 561: Data Systems Architectures

Introduction to Indexing:

Trees, Tries, Hashing, Bitmap Indexes, Database Cracking

Prof. Manos Athanassoulis

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

Recap: Key-Value Stores

<key, value>

put(key, value)
stores value and associates with key

get(key)
returns the associated value

delete(key)
deletes the value associated with the key

get_range (key_start,key_end)
get_set(key1, key2, …)

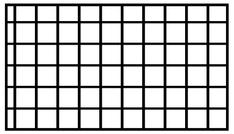
how to organize keys/values?

depends on the workload!

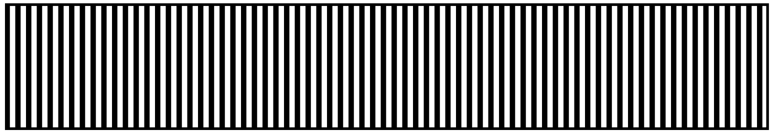
KVS

Recap: Key-Value Stores

inserts and point queries?



hash table



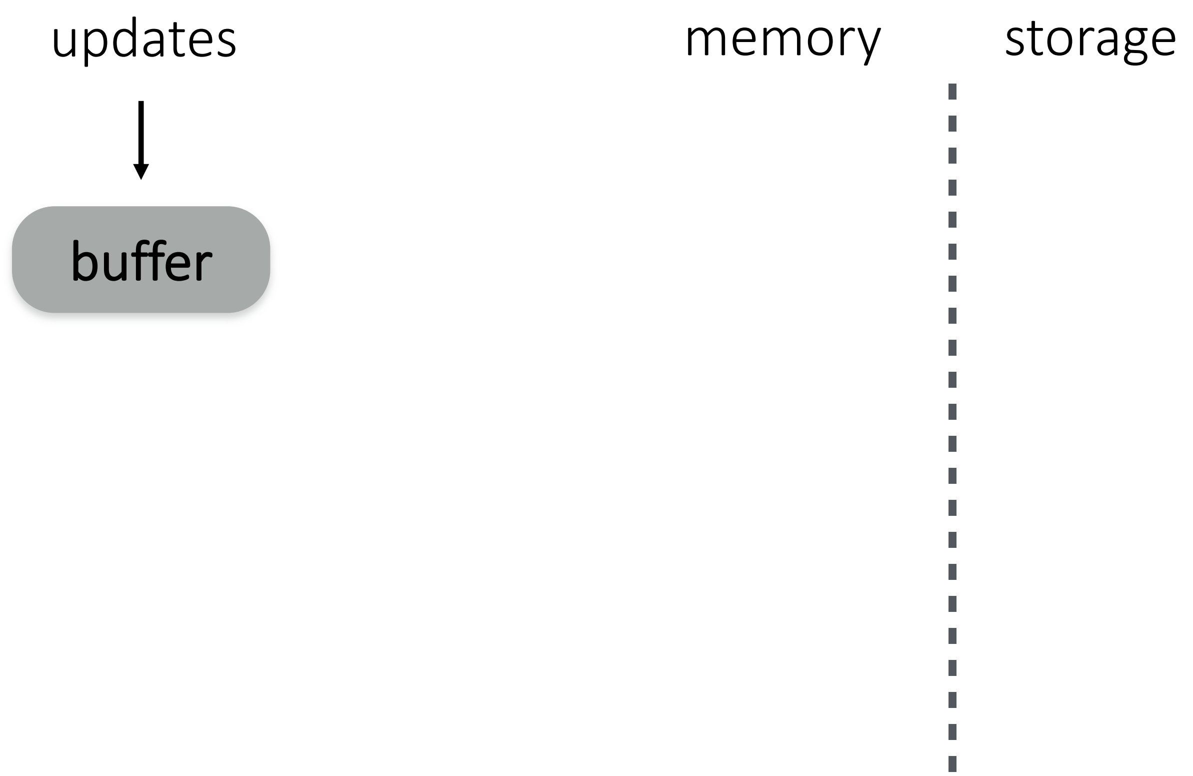
log

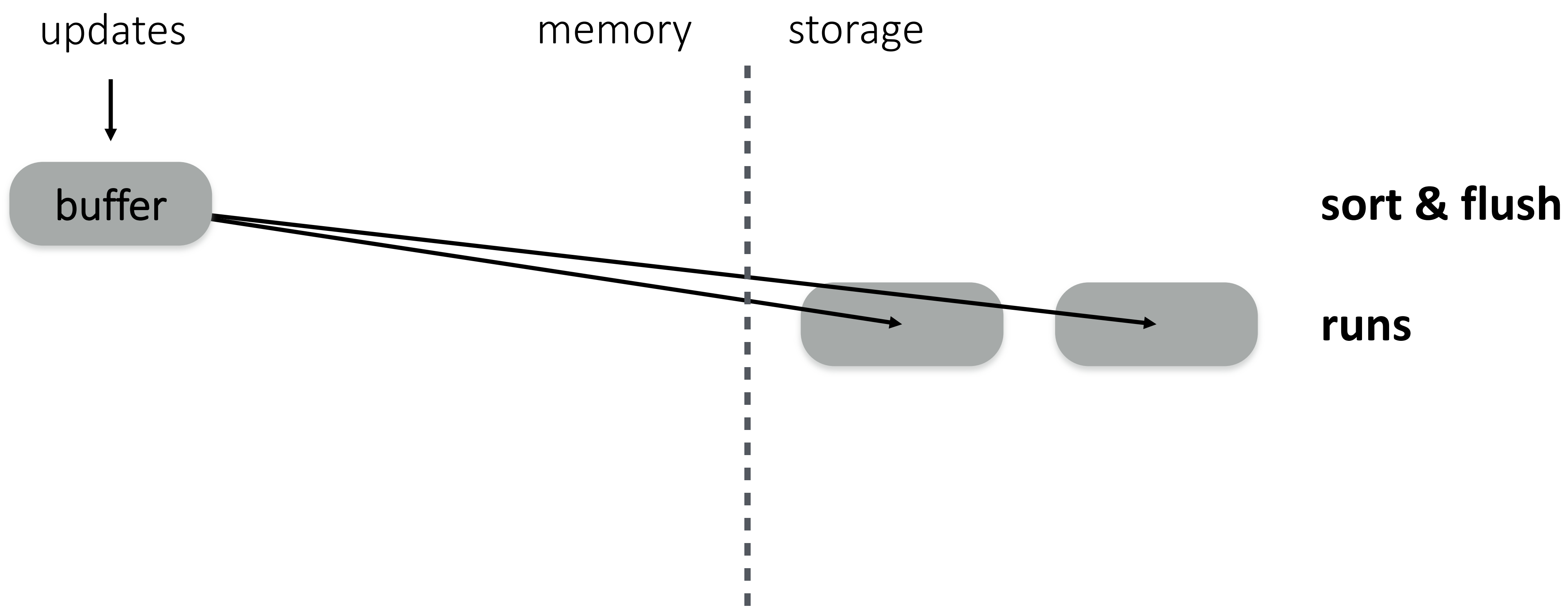
inserts, point queries, and range queries?

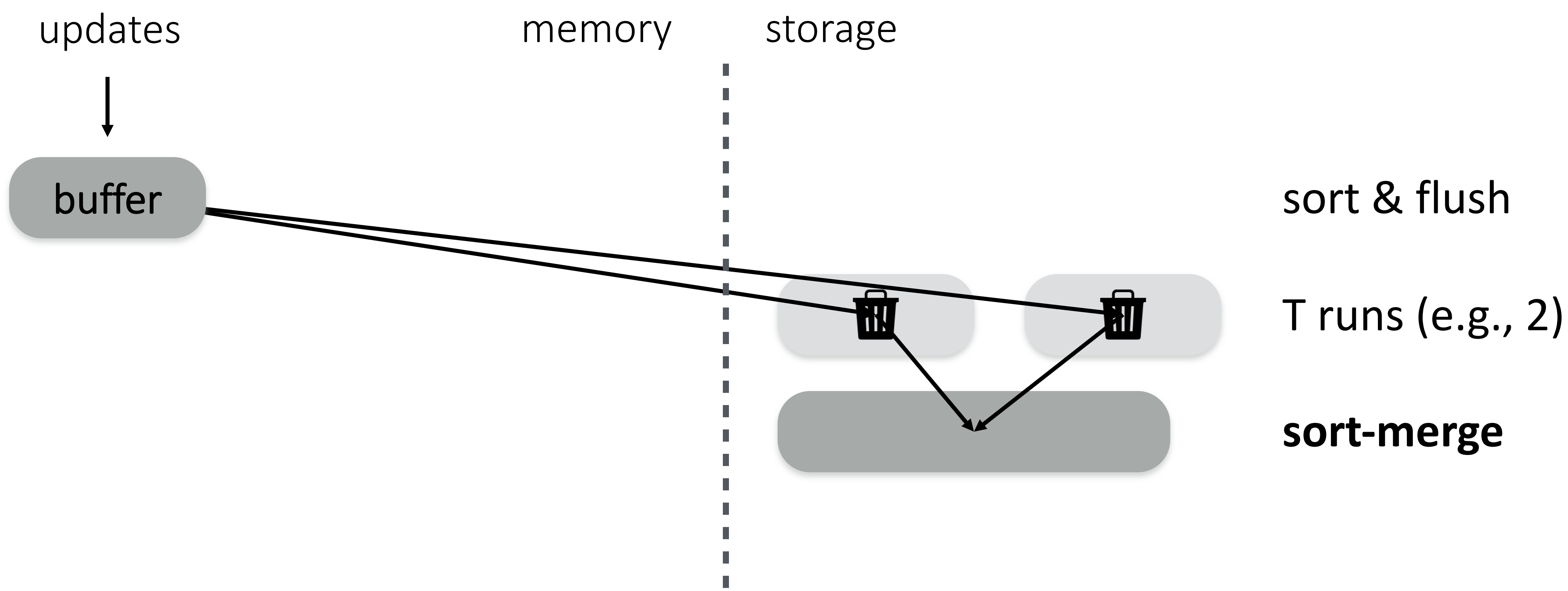
log-structured merge tree

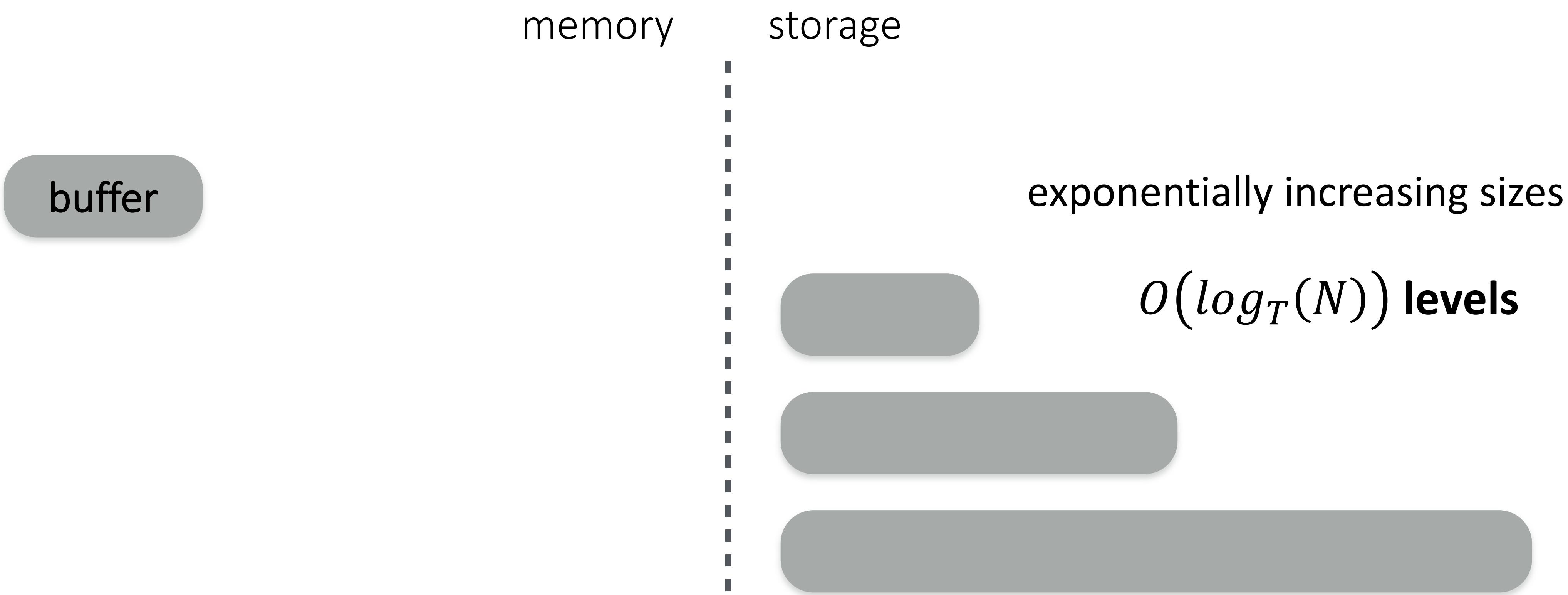
LSM-Trees

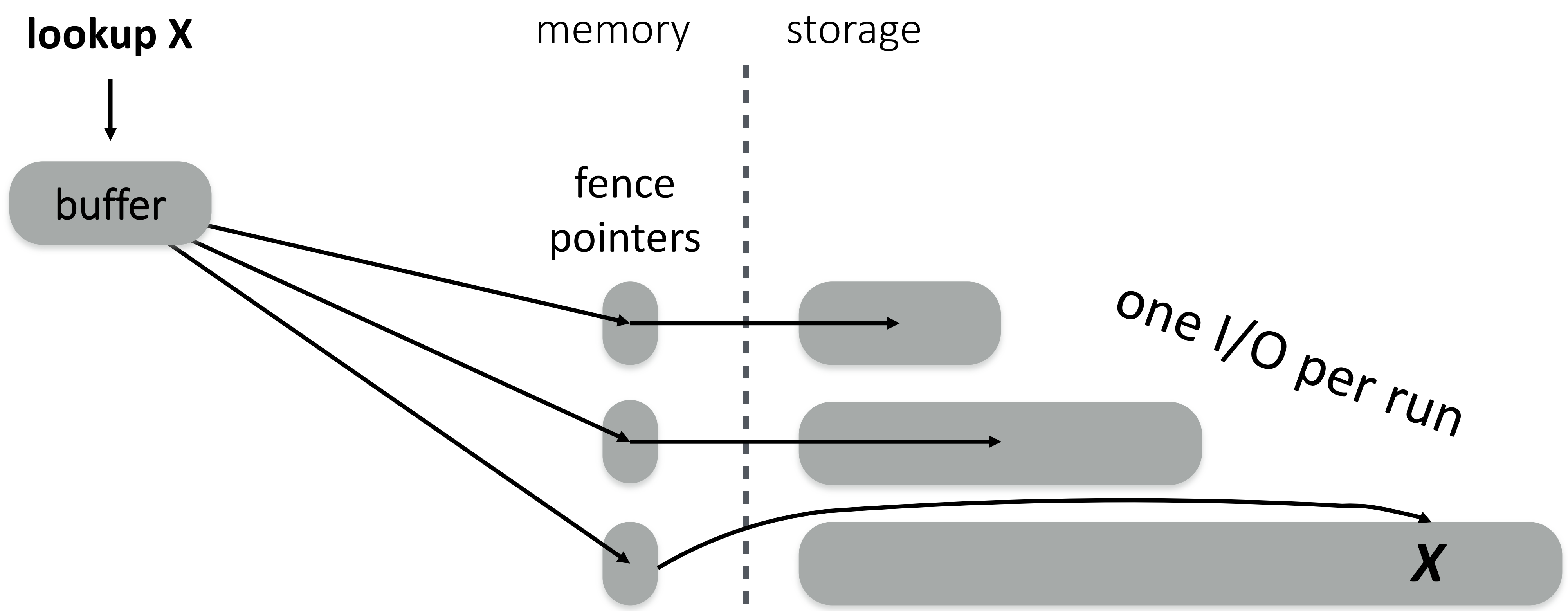
A quick review of LSM-Trees and what is expected for the systems project

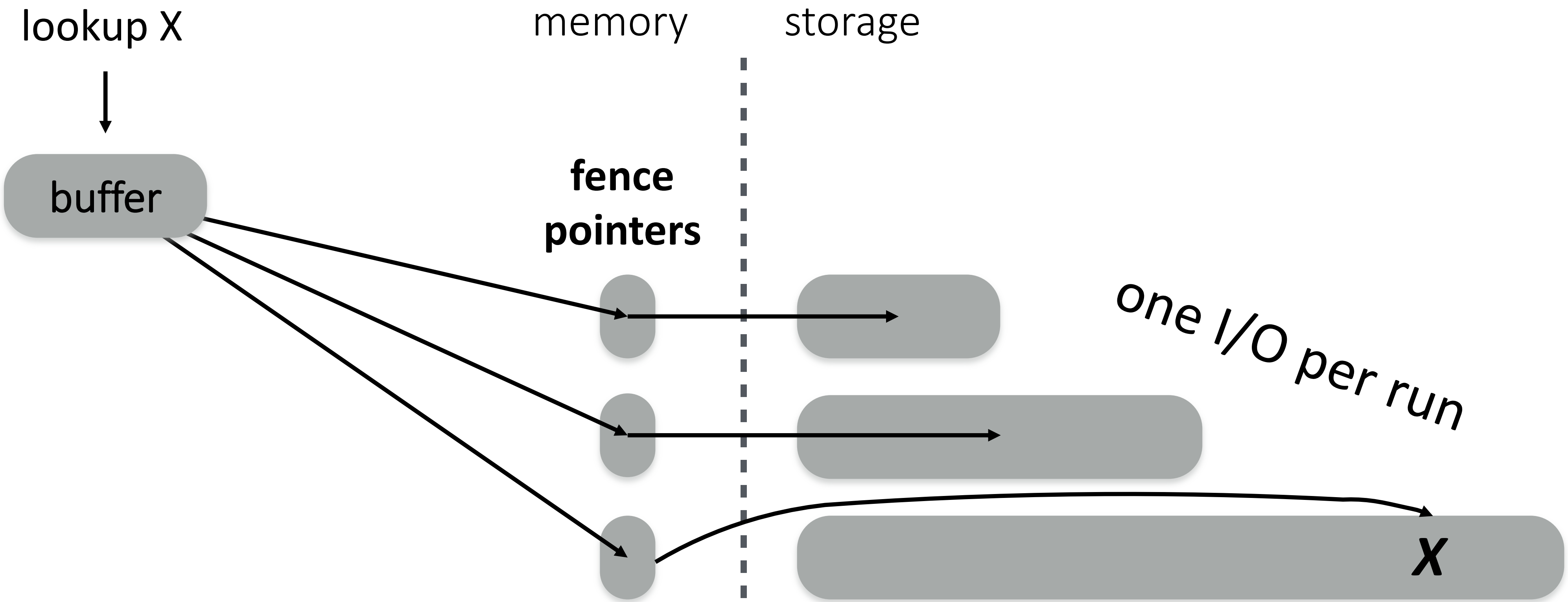

updates
buffer
memory
storage

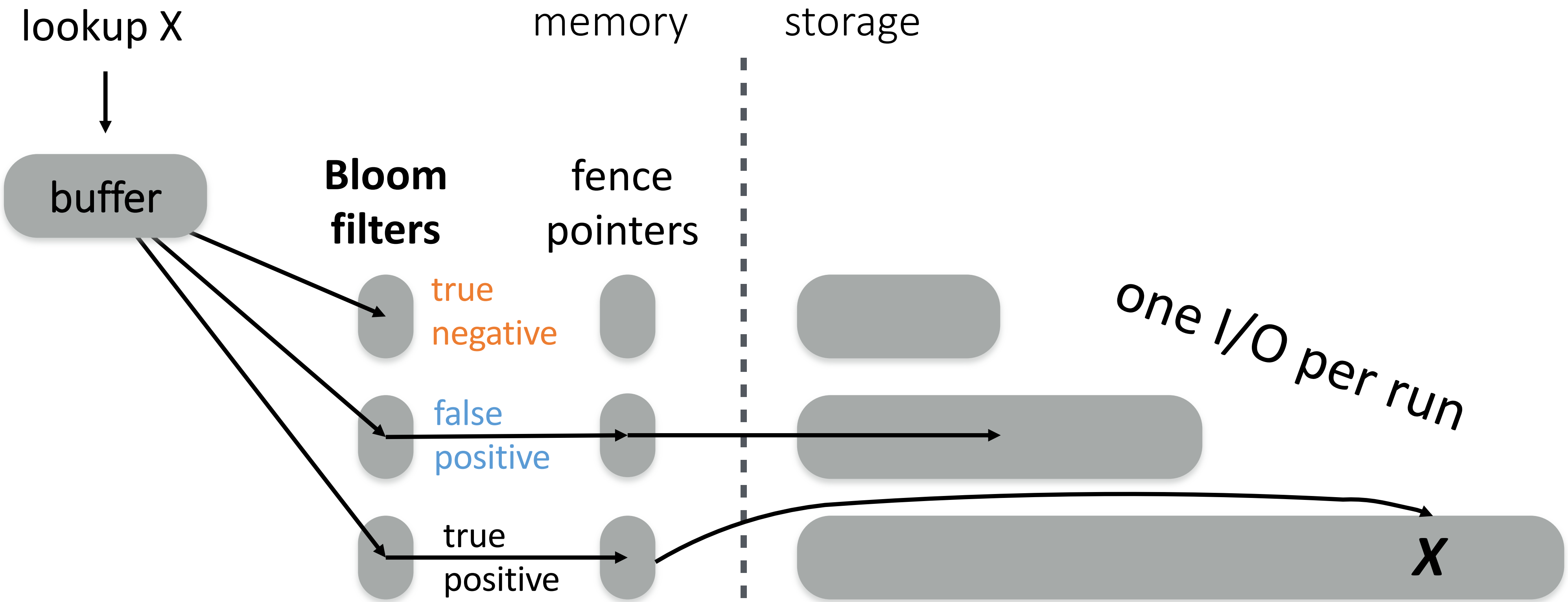

updates
memory
storage
buffer
sort & flush
runs


updates
memory
storage
buffer
sort & flush
T runs (e.g., 2)
sort-merge


memory
storage
buffer
exponentially increasing sizes
$O\left(log_T(N)\right)$ **levels**

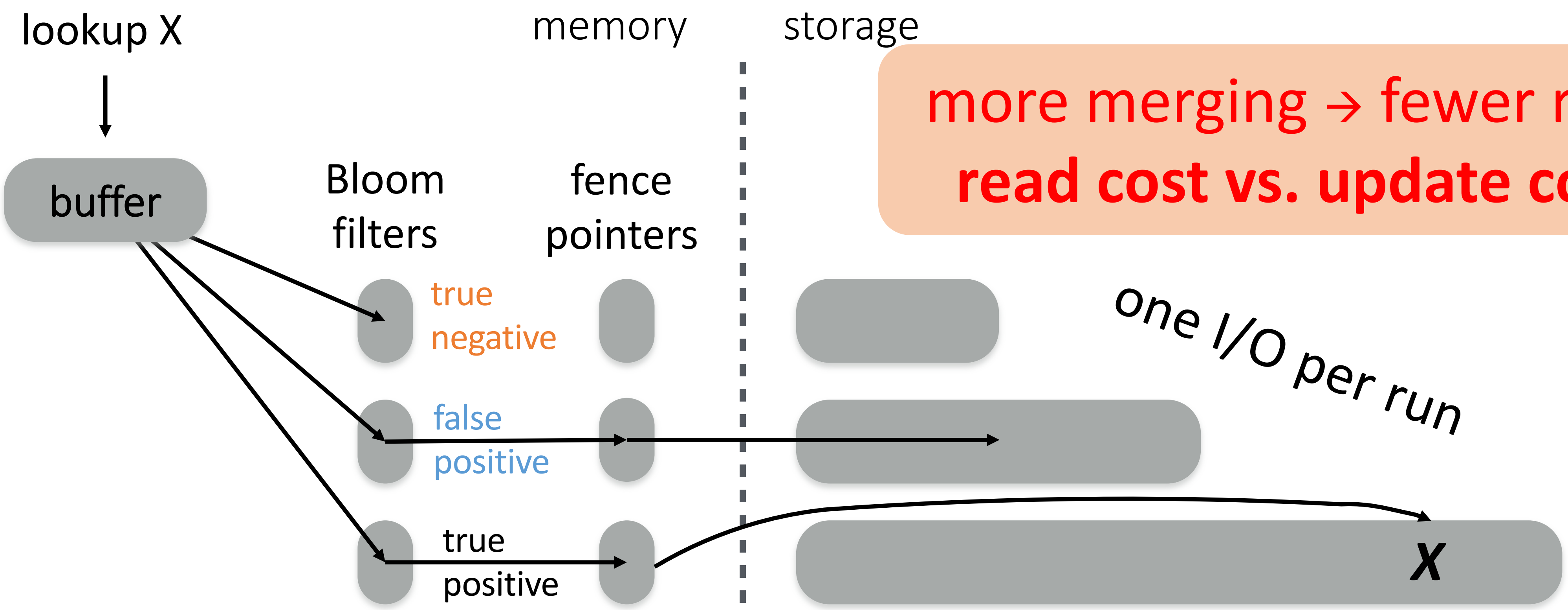

lookup X
memory
storage
buffer
fence
pointers
one I/O per run
X


lookup X
memory
storage
buffer
fence
pointers
one I/O per run
X



lookup X
memory
storage
buffer
Bloom filters
fence pointers
true negative
false positive
true positive
one I/O per run
X

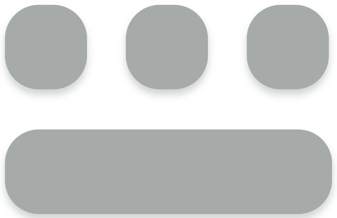
performance & cost trade-offs



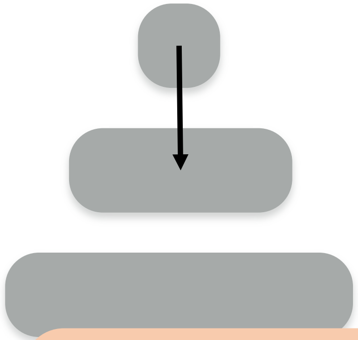
more merging → fewer runs
read cost vs. update cost

bigger filters → fewer false positives
memory space vs. read cost

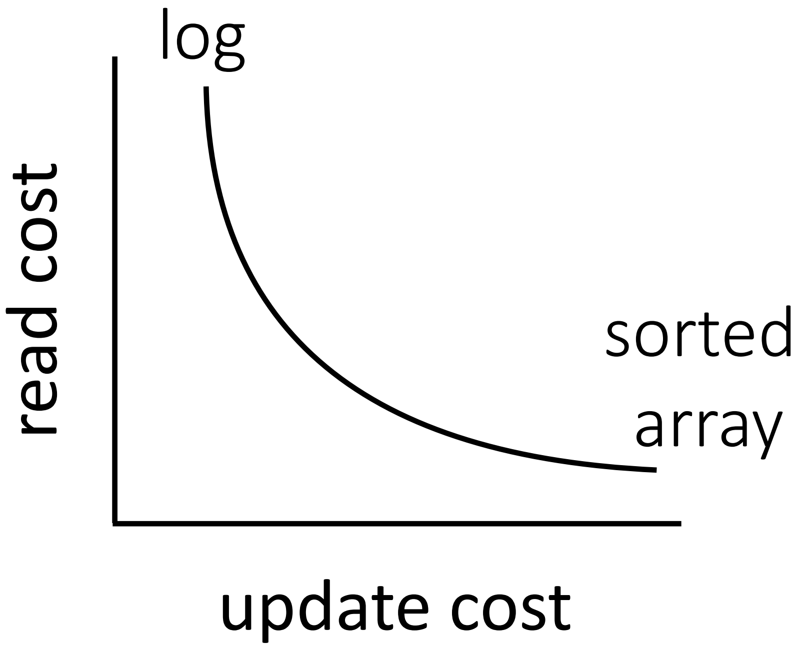
tuning *reads* vs. *updates*



merge policy



size ratio



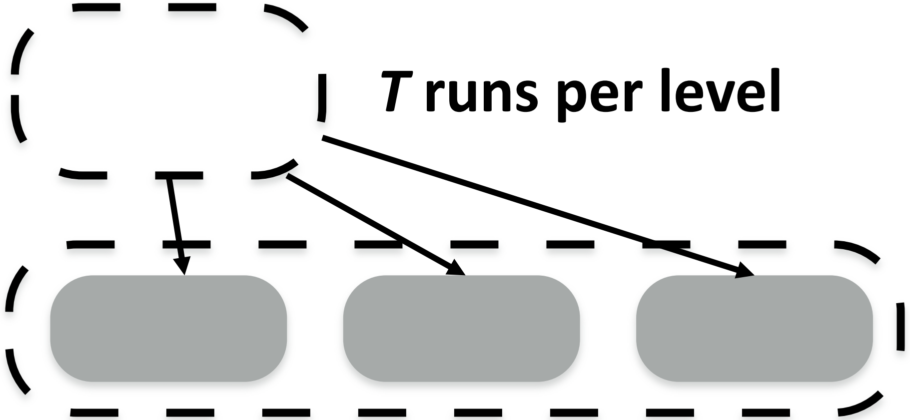
log **LSM-Tree** sorted array

Merge Policies

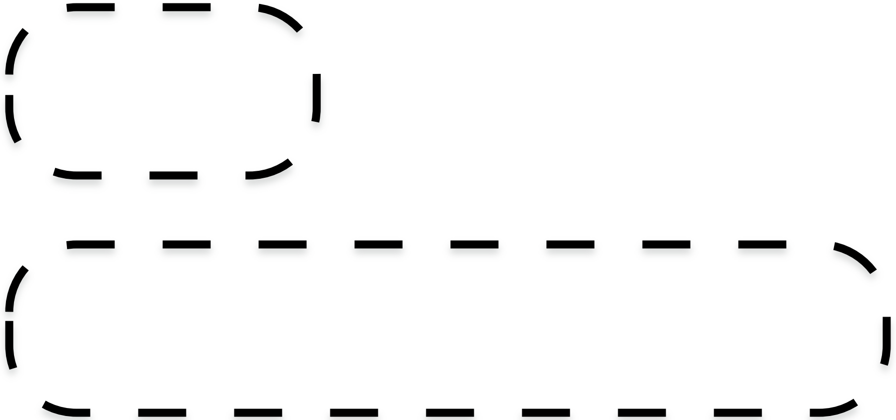
Tiering
write-optimized

Leveling
read-optimized

Tiering
write-optimized

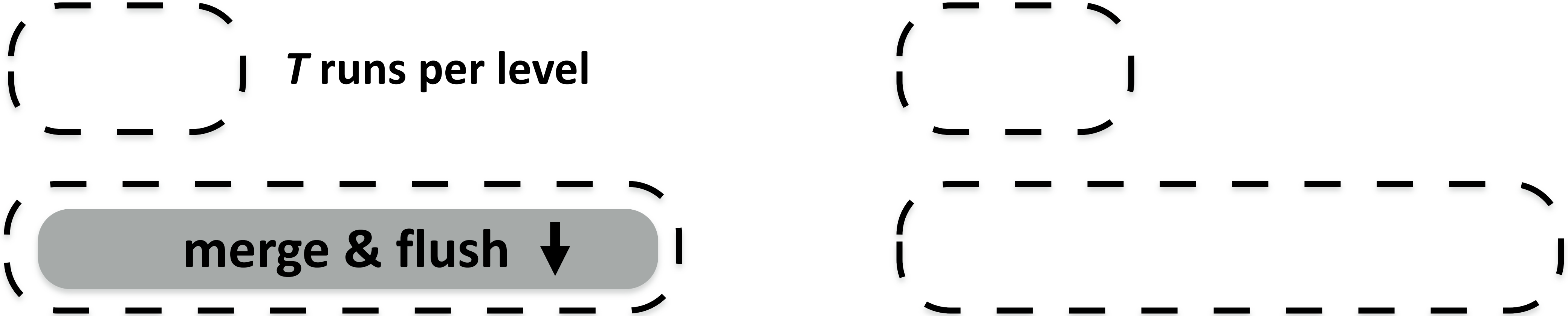


Leveling
read-optimized



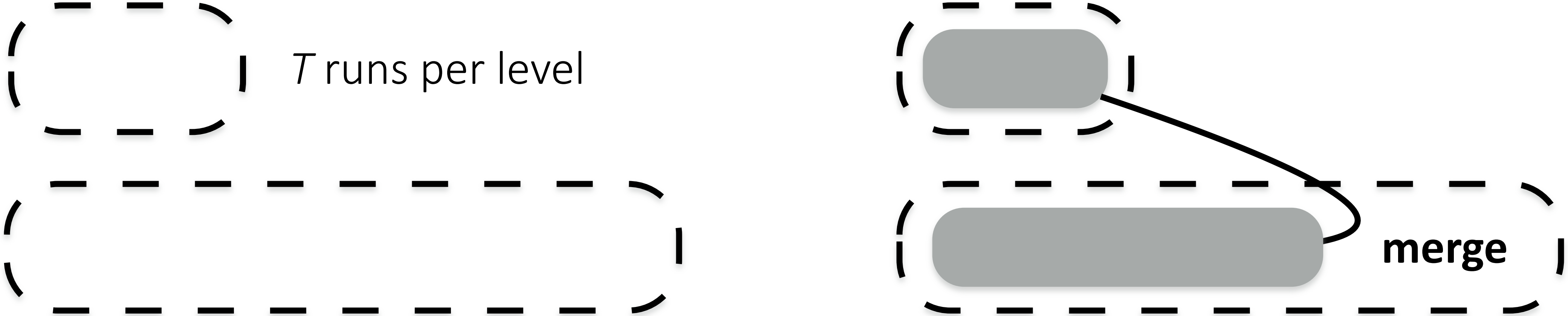
Tiering
write-optimized

Leveling
read-optimized

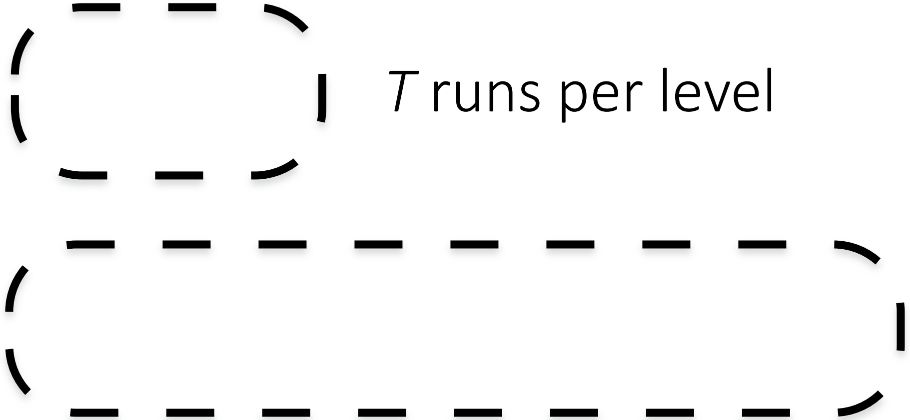


Tiering
write-optimized

Leveling
read-optimized

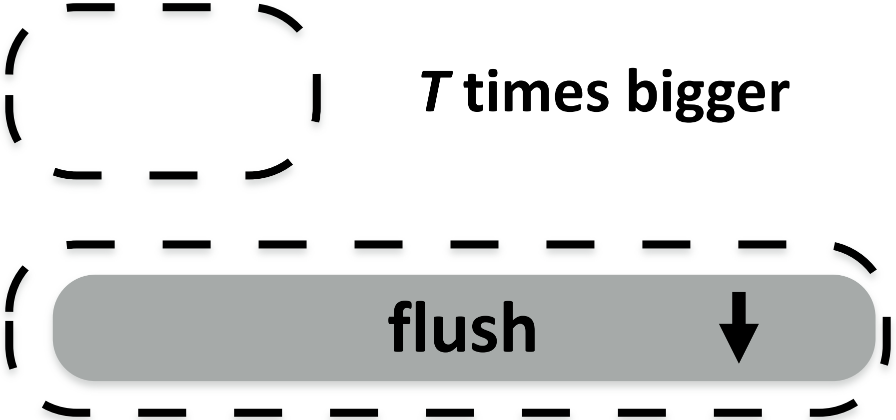


Tiering
write-optimized



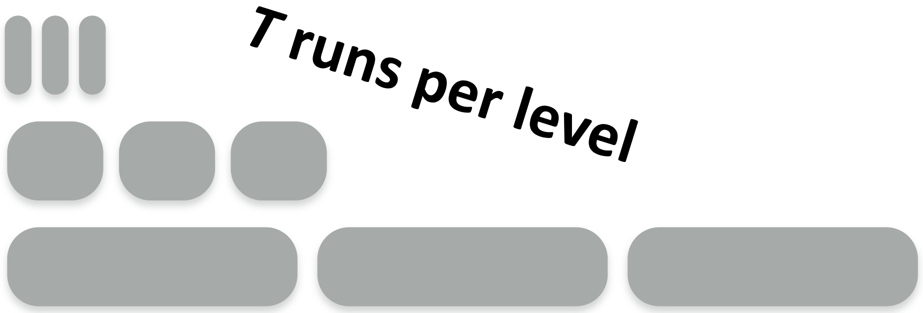
T runs per level

Leveling
read-optimized



***T* times bigger**

Tiering
write-optimized



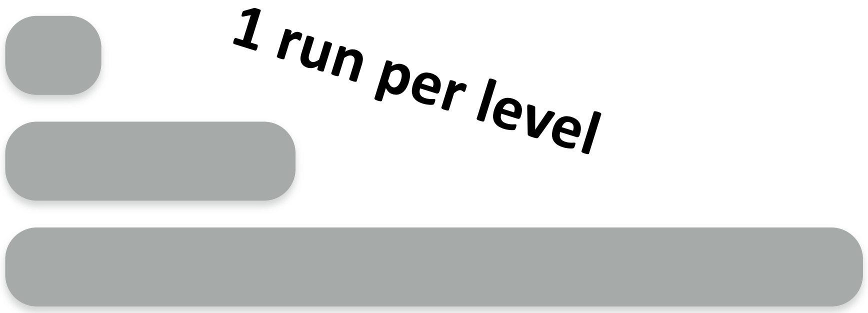
lookup cost: $O\left(T \cdot log_T(N) \cdot e^{-M/N}\right)$

runs per level

levels

false positive rate

Leveling
read-optimized

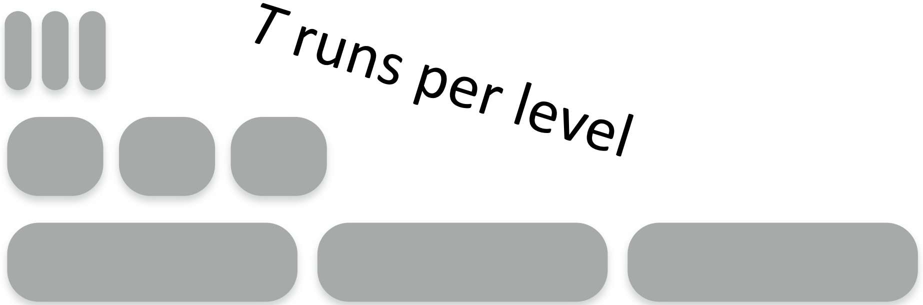


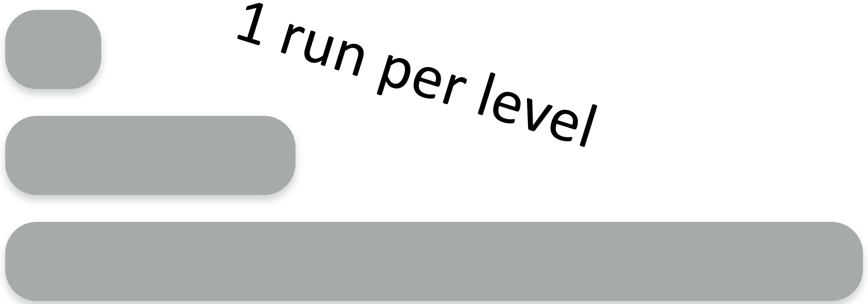
$O\left(log_T(N) \cdot e^{-M/N}\right)$

levels

false positive rate

Tiering write-optimized	Leveling read-optimized

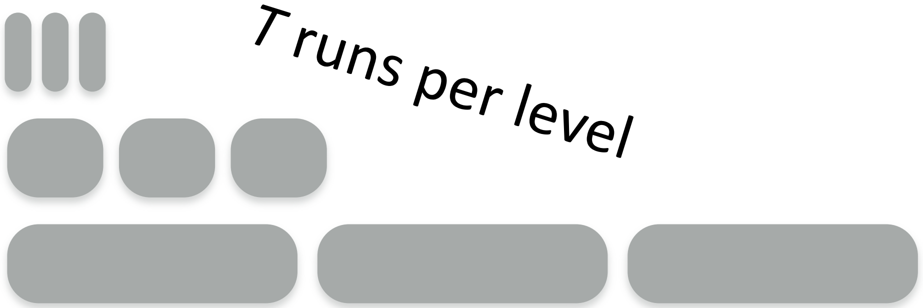


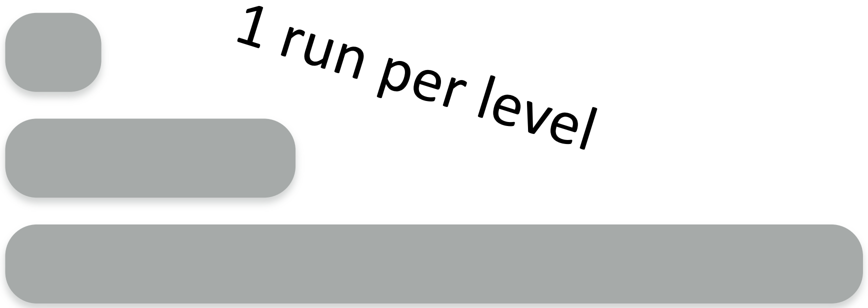


	Tiering	Leveling
lookup cost:	$O\left(T \cdot log_T(N) \cdot e^{-M/N}\right)$	$O\left(log_T(N) \cdot e^{-M/N}\right)$
update cost:	$\boldsymbol{O\left(log_T(N)\right)}$	$\boldsymbol{O\left(T \cdot log_T(N)\right)}$
	levels	merges per level levels

Tiering
write-optimized

Leveling
read-optimized

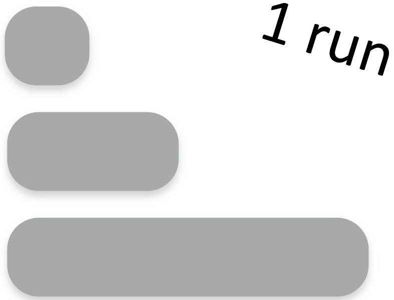




	Tiering	Leveling
lookup cost:	$O\left(T \cdot log_T(N) \cdot e^{-M/N}\right)$	$O\left(log_T(N) \cdot e^{-M/N}\right)$
update cost:	$O\left(log_T(N)\right)$	$O\left(T \cdot log_T(N)\right)$

for size ratio T

Tiering
write-optimized



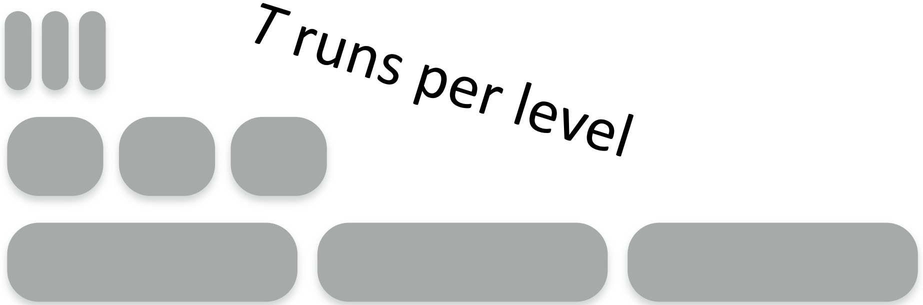
Leveling
read-optimized

lookup cost: $O\left(log_T(N) \cdot e^{-M/N}\right) = O\left(log_T(N) \cdot e^{-M/N}\right)$

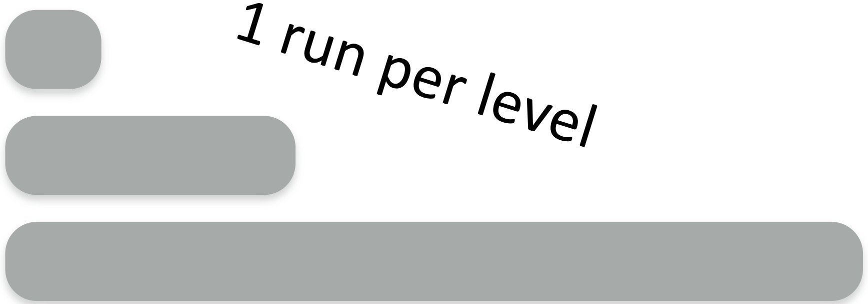
update cost: $O\left(log_T(N)\right) = O\left(log_T(N)\right)$

for size ratio T

Tiering
write-optimized

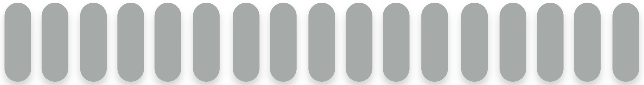


Leveling
read-optimized



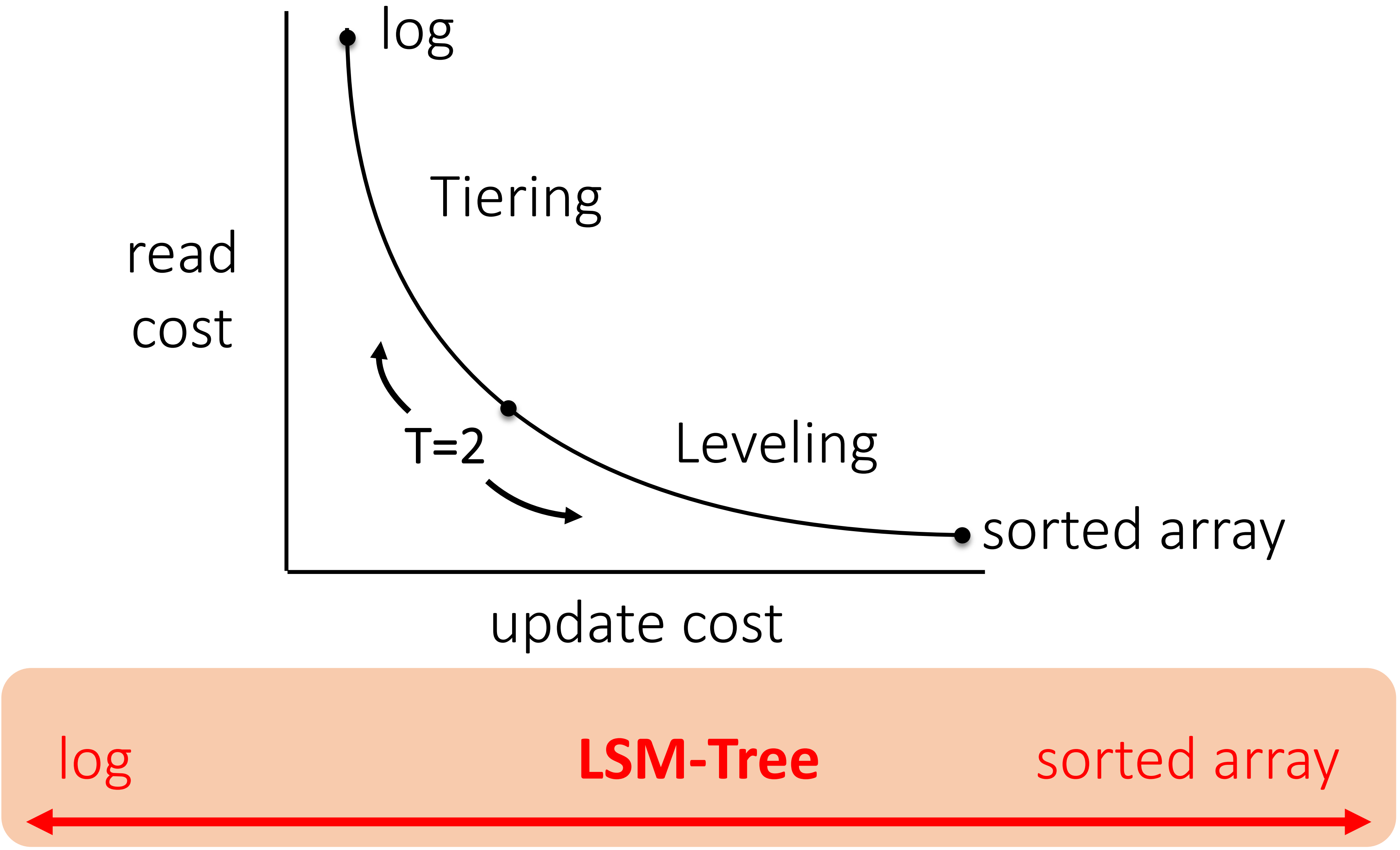
	Tiering	Leveling
lookup cost:	$O\left(T \cdot log_T(N) \cdot e^{-M/N}\right)$	$O\left(log_T(N) \cdot e^{-M/N}\right)$
update cost:	$O\left(log_T(N)\right)$	$O\left(T \cdot log_T(N)\right)$

for size ratio T ⇑

	Tiering write-optimized	Leveling read-optimized
	$O(N)$ runs per level	1 run per level
		
	log	**sorted array**
lookup cost:	$O\left(N \cdot e^{-M/N}\right)$	$O\left(e^{-M/N}\right)$
update cost:	$O\left(log_N(N)\right) = \boldsymbol{O(1)}$	$O\left(N \cdot log_N(N)\right) = \boldsymbol{O(N)}$

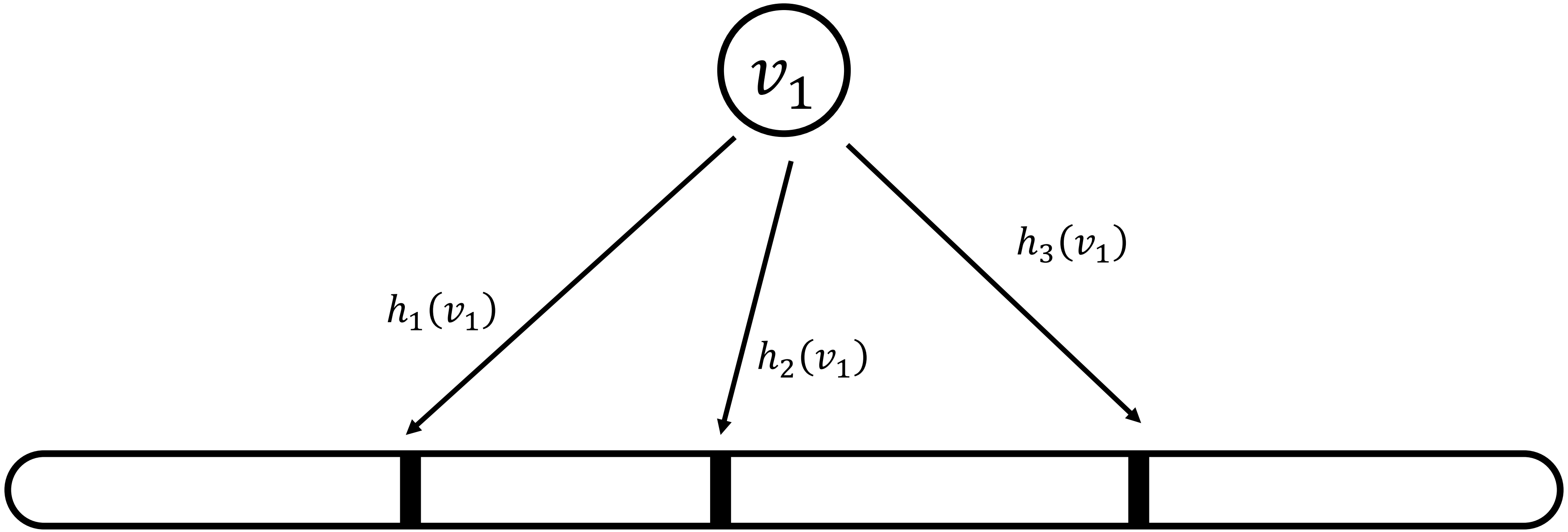
N

for size ratio T


log
Tiering
read
cost
T=2
Leveling
sorted array
update cost
log
LSM-Tree
sorted array

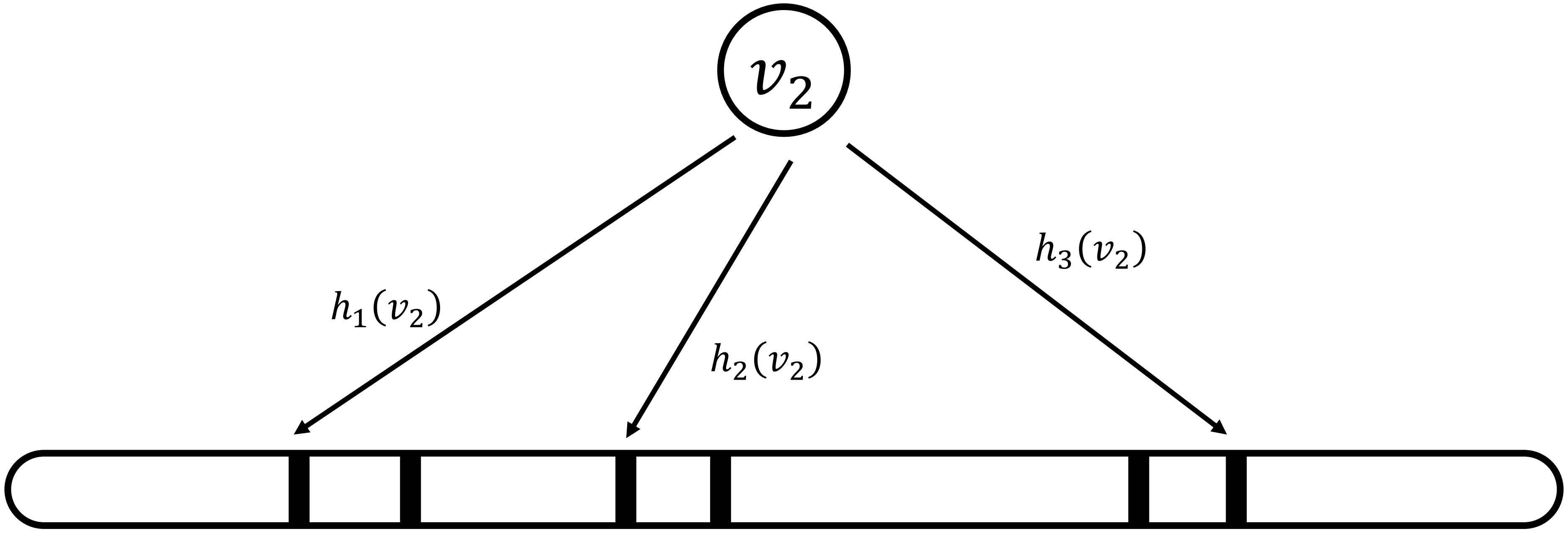
Details on Bloom filters

Inserting into a Bloom filter



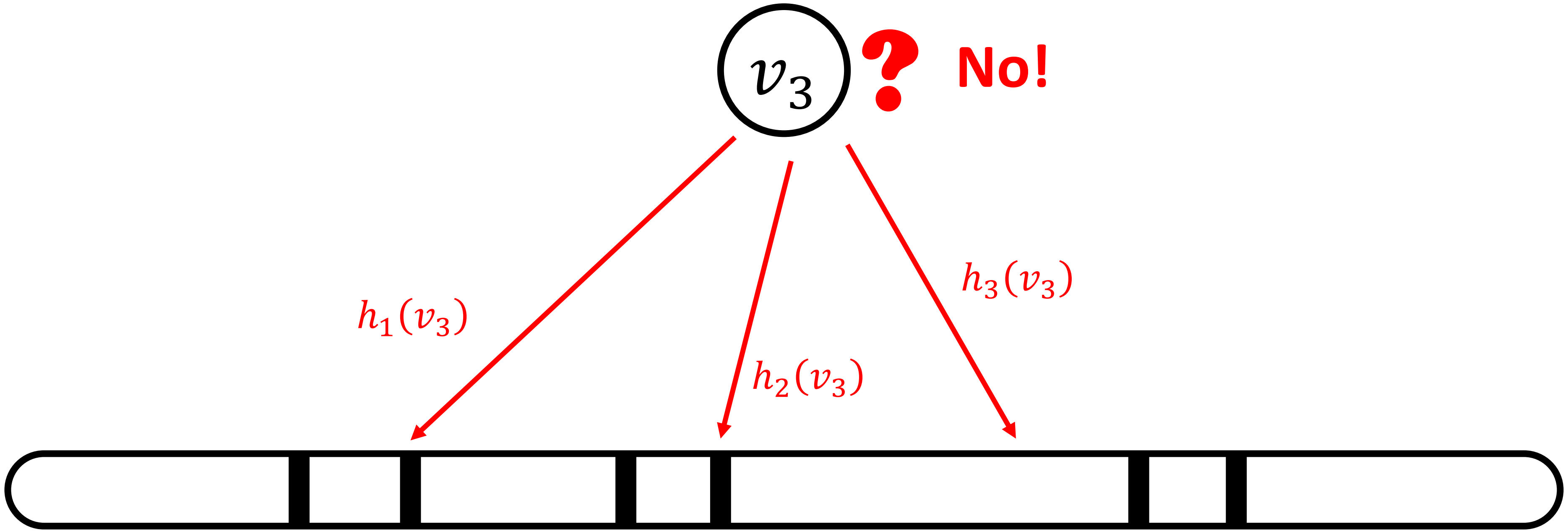
BOSTON UNIVERSITY

Inserting into a Bloom filter

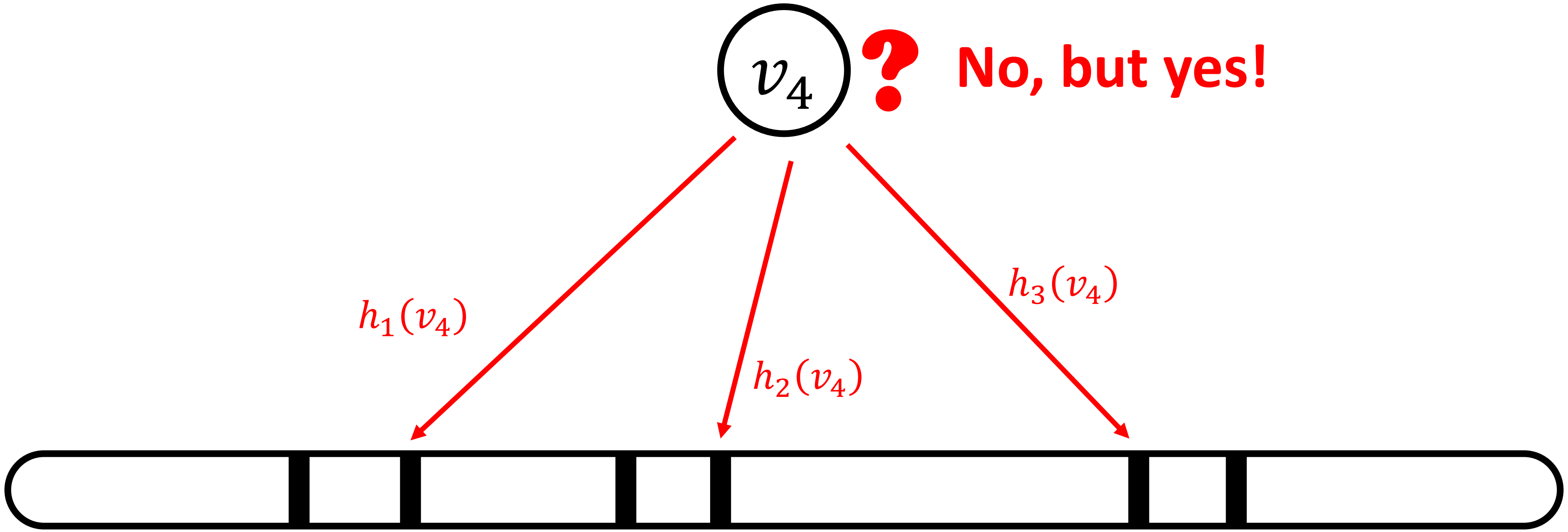


BOSTON UNIVERSITY

Probing a Bloom filter (true negative)



Probing a Bloom filter (false positive)

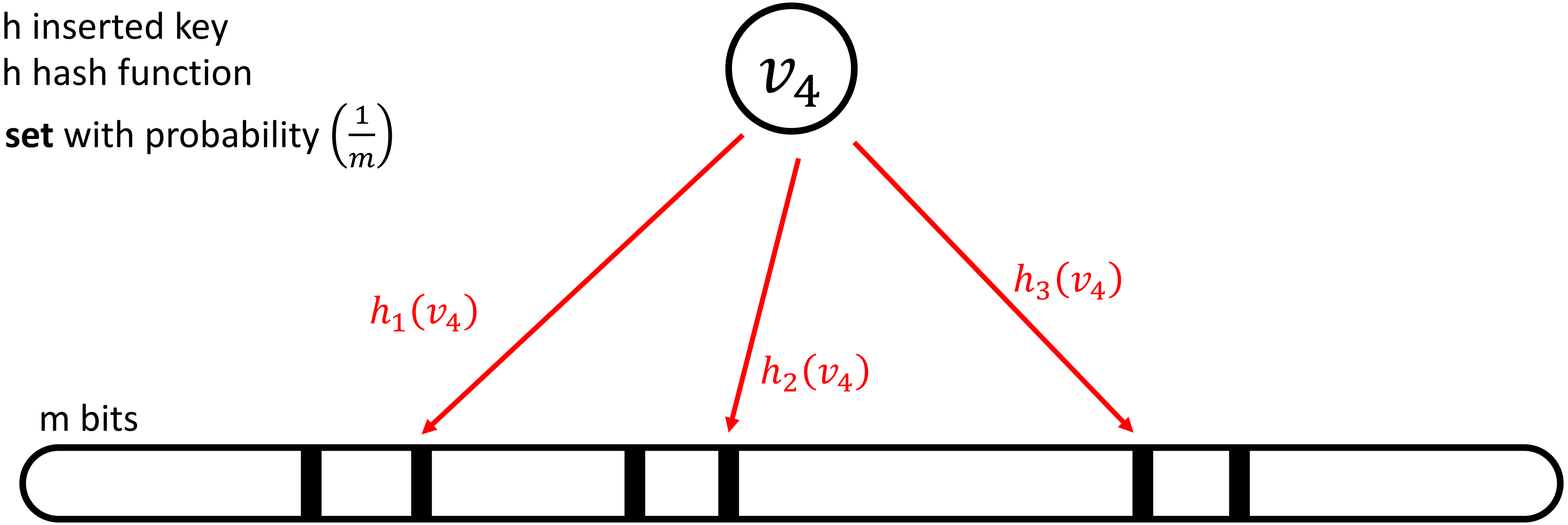


what is the probability of a false positive?

Bloom filter false positive

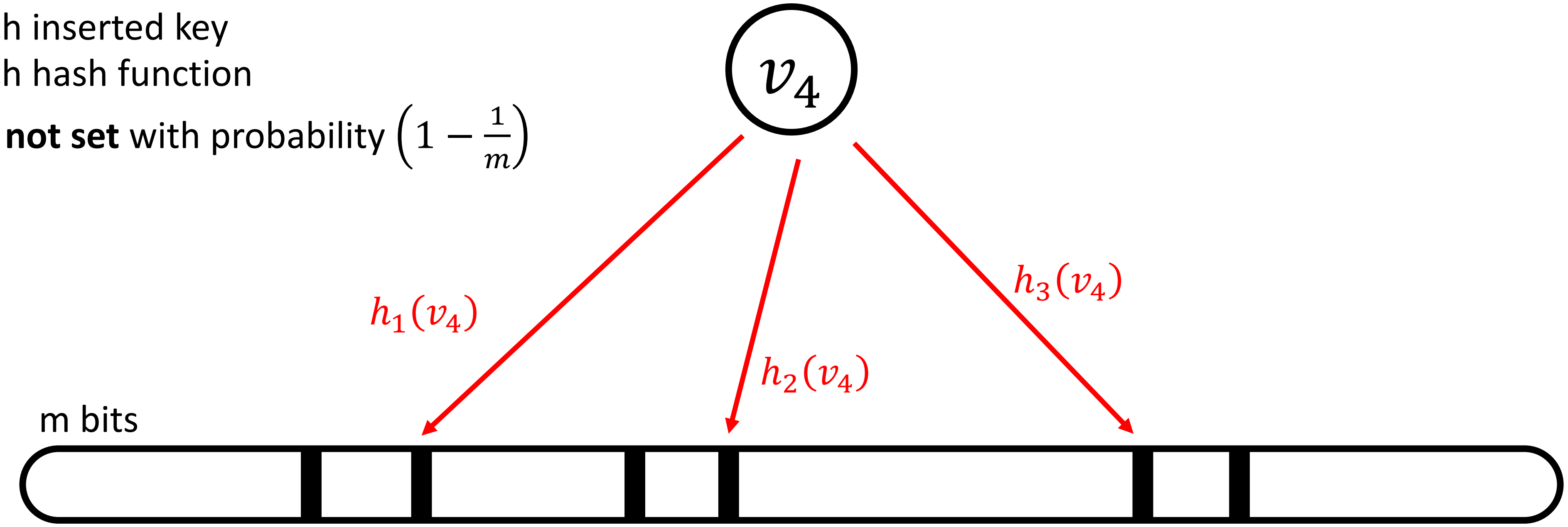
for each inserted key
for each hash function

a bit is **set** with probability $\left(\frac{1}{m}\right)$



Bloom filter false positive

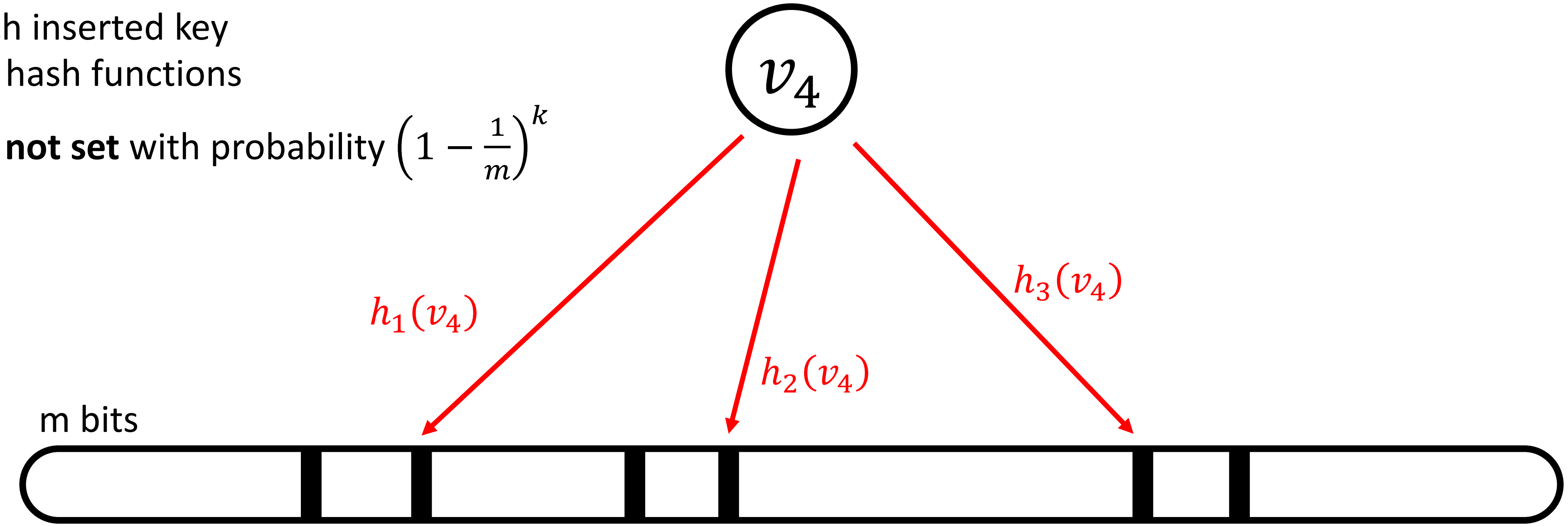
for each inserted key
for each hash function
a bit is **not set** with probability $\left(1 - \frac{1}{m}\right)$



Bloom filter false positive

for each inserted key
after ***k*** hash functions

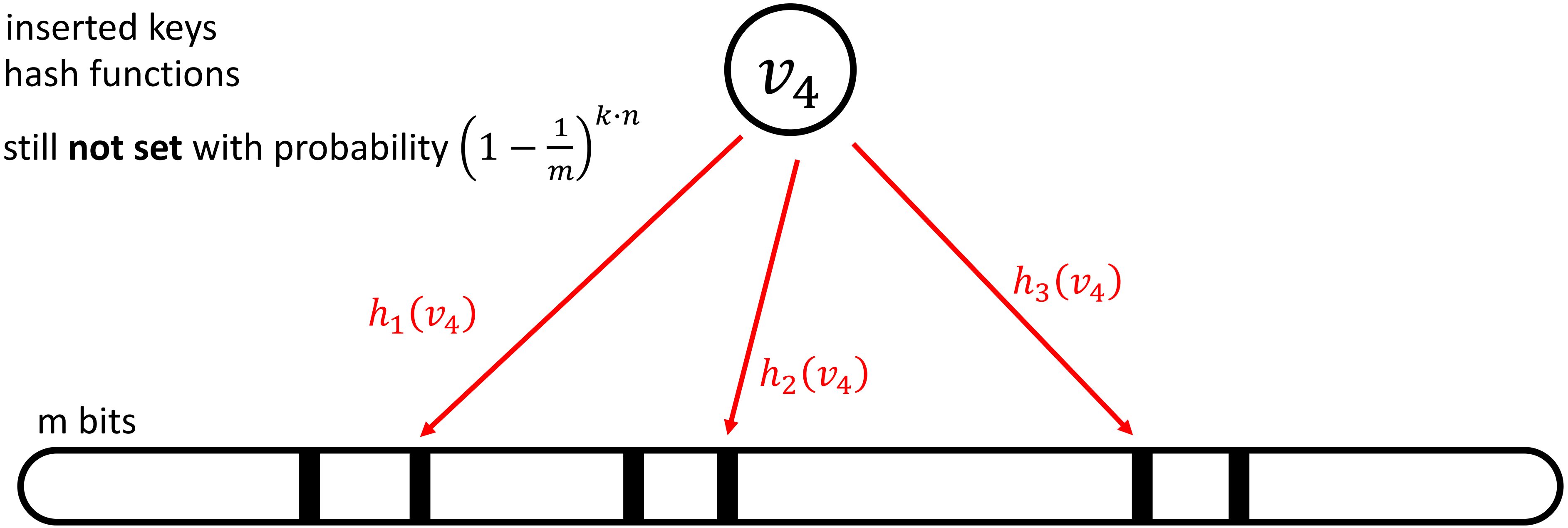
a bit is **not set** with probability $\left(1-\frac{1}{m}\right)^k$



Bloom filter false positive

after ***n*** inserted keys
after ***k*** hash functions

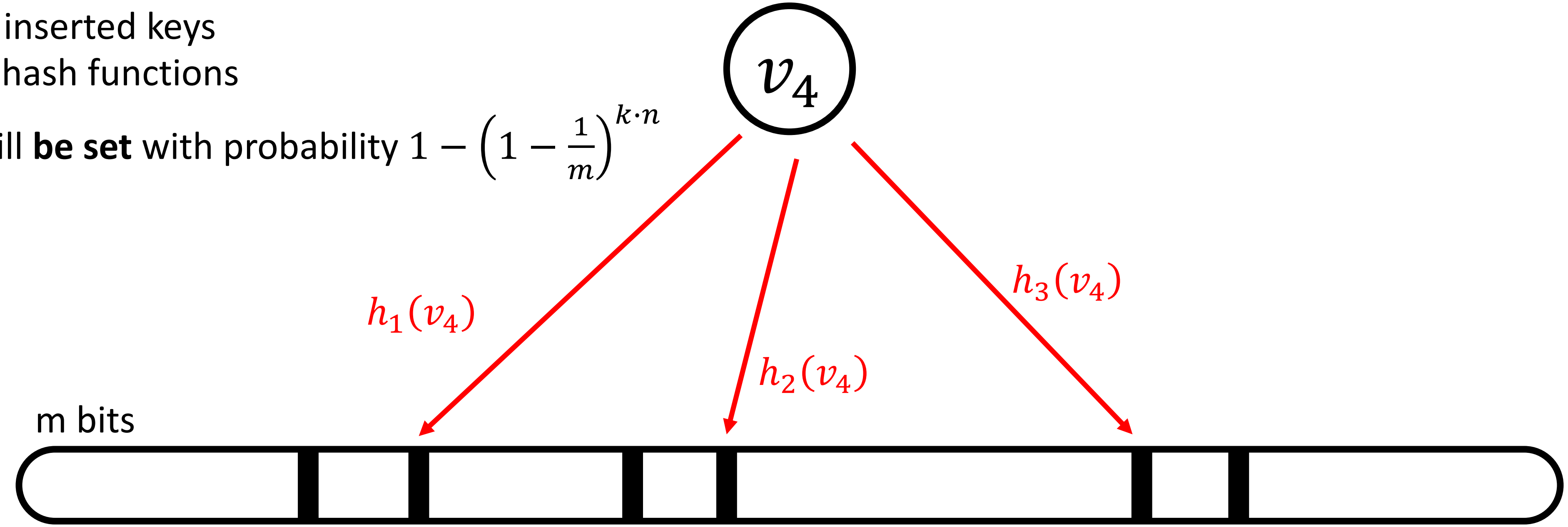
a bit is still **not set** with probability $\left(1-\frac{1}{m}\right)^{k\cdot n}$



Bloom filter false positive

after ***n*** inserted keys
after ***k*** hash functions

a bit will **be set** with probability $1 - \left(1 - \frac{1}{m}\right)^{k \cdot n}$

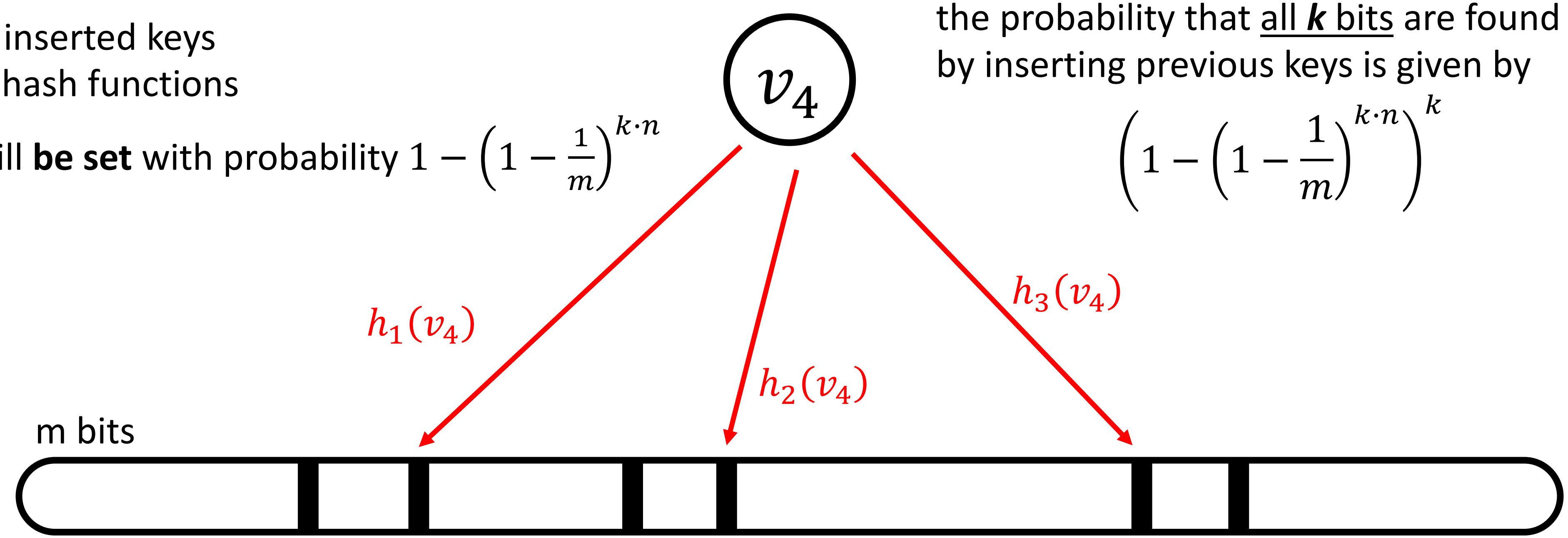


BOSTON UNIVERSITY

Bloom filter false positive

after ***n*** inserted keys
after ***k*** hash functions

a bit will **be set** with probability $1-\left(1-\frac{1}{m}\right)^{k\cdot n}$



the probability that <u>all ***k*** bits</u> are found set by inserting previous keys is given by

$$\left(1-\left(1-\frac{1}{m}\right)^{k\cdot n}\right)^{k}$$

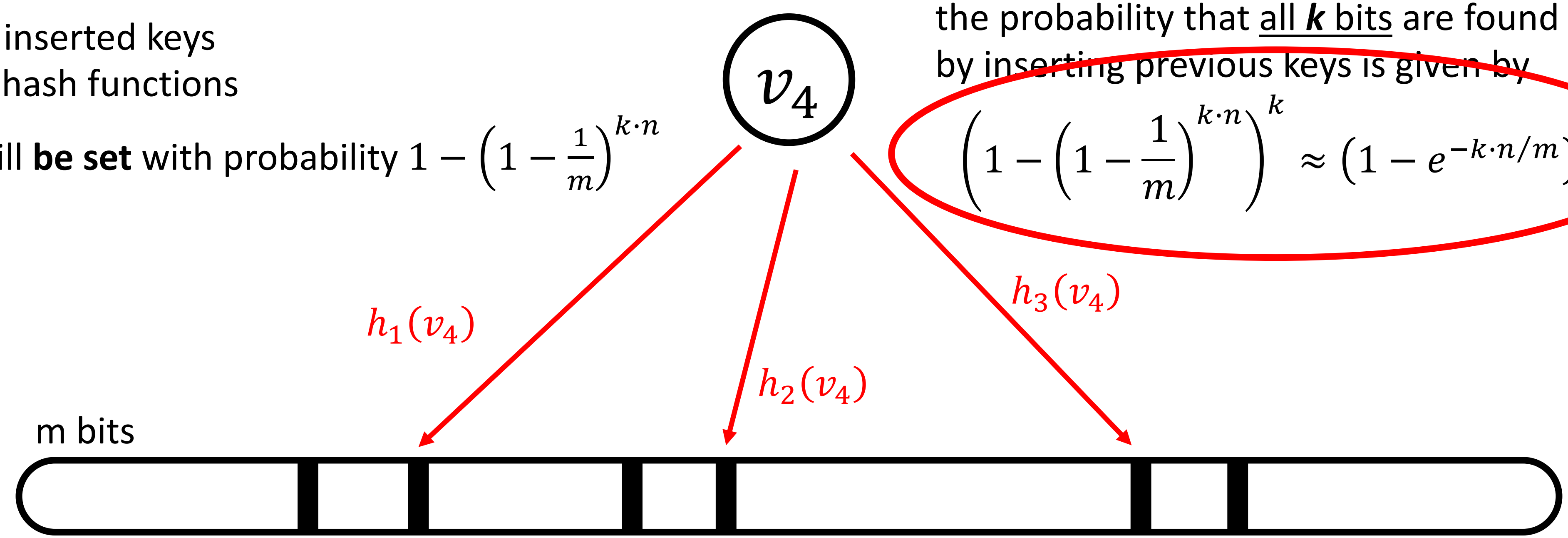
Bloom filter false positive

after ***n*** inserted keys
after ***k*** hash functions

a bit will **be set** with probability $1-\left(1-\frac{1}{m}\right)^{k\cdot n}$

the probability that <u>all ***k*** bits</u> are found set by inserting previous keys is given by

$$\left(1-\left(1-\frac{1}{m}\right)^{k\cdot n}\right)^{k} \approx \left(1-e^{-k\cdot n/m}\right)^{k}$$



Bloom filter false positive (derivation details)

let's focus on the term: $\left(1-\frac{1}{m}\right)^n$

assuming $\alpha = \frac{m}{n}$, and for large m, n:

$\left(1-\frac{1}{m}\right)^n = \left(1-\frac{1}{\alpha \cdot n}\right)^n = \left(1+\frac{-1/\alpha}{n}\right)^n \approx e^{-1/\alpha} = e^{-n/m}$, because $\lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n = e^x$

hence, the probability that <u>all ***k*** bits</u> are found set by inserting previous keys is given by

$$\left(1-\left(1-\frac{1}{m}\right)^{n\cdot k}\right)^k = \left(1-\left(\left(1-\frac{1}{m}\right)^n\right)^k\right)^k = \left(1-\left(e^{-n/m}\right)^k\right)^k = \left(1-\left(e^{-k\cdot n/m}\right)\right)^k$$

Bloom filter false positive

false positive $p = \left(1 - e^{-k \cdot n/m}\right)^k$

how to minimize?

it can be shown (not easy):
the optimal number of hash functions k, that minimize the false positive is:

$$k = \frac{m}{n} \cdot ln(2)$$

Rule of thumb: k is a number, often between 2 and 10.

Bloom filter false positive

Combining $p = \left(1 - e^{-k \cdot n/m}\right)^k$ and $k = \frac{m}{n} \cdot ln(2)$

we get: $e^{-\frac{m}{n} \cdot (ln(2))^2}$

details:

$$p = \left(1 - e^{-\frac{m}{n} \cdot ln(2) \cdot \frac{n}{m}}\right)^{\frac{m}{n} \cdot ln(2)} = \left(1 - e^{-ln(2)}\right)^{\frac{m}{n} \cdot ln(2)} = \left(1 - \frac{1}{2}\right)^{\frac{m}{n} \cdot ln(2)} = \left({}^{1}/_{2}\right)^{\frac{m}{n} \cdot ln(2)}$$

using twice that ${}^{1}/_{2} = e^{-ln(2)}$, $p = \left(e^{-ln(2)}\right)^{\frac{m}{n} \cdot ln(2)} = e^{-\frac{m}{n} \cdot ln(2) \cdot ln(2)} = e^{-\frac{m}{n} \cdot (ln(2))^2}$

BOSTON UNIVERSITY

key-value stores vs. indexes

What is an index?

Auxiliary structure to quickly find rows based on arbitrary attribute

Special form of <key, value>

indexed attribute

position/location/rowID/primary key/…

What are the possible ***index designs***?

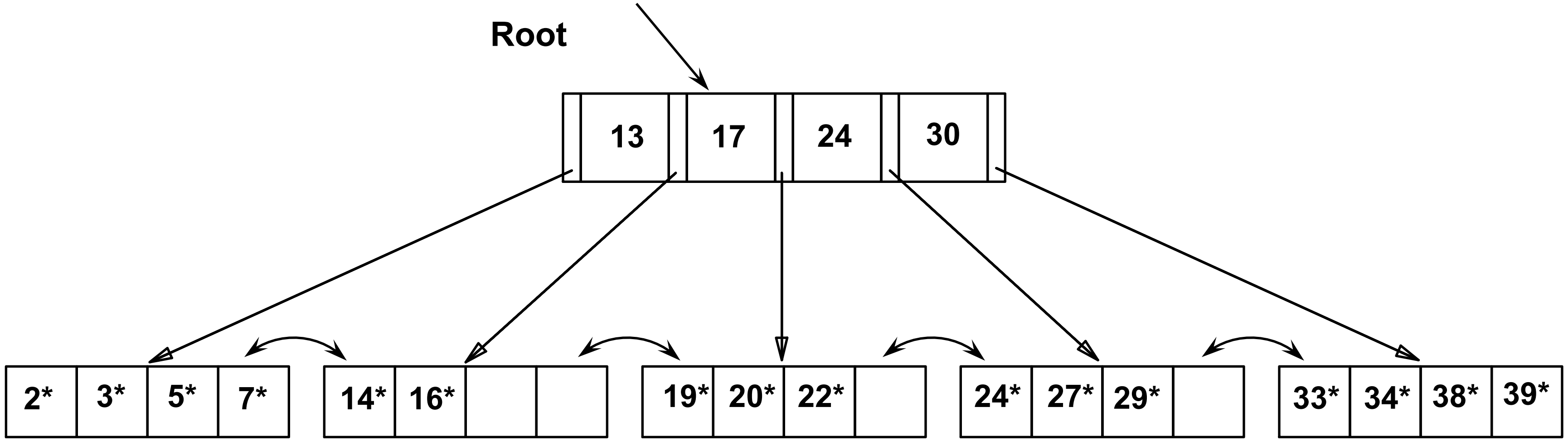
	Data Organization	Point Queries	Short Range Queries	Long Range Queries	Comments
B+ Trees	Range	✅	✅	✅	Partition ***k-ways*** recursively
LSM Trees	Insertion & Sorted	✅	❌	✅	Optimizes ***insertion***
Radix Trees	Radix	✅	✅	✅	Partition using the ***key radix*** representation
Hash Indexes	Hash	✅	–	❌	Partition by ***hashing the key***
Bitmap Indexes	None	✅	–	❌	Succinctly represent ***all rows with a key***
Scan Accelerators	None	❌	–	✅	Metadata to ***skip accesses***

BOSTON UNIVERSITY

B+ Trees

Search begins at root, and key comparisons direct it to a leaf.

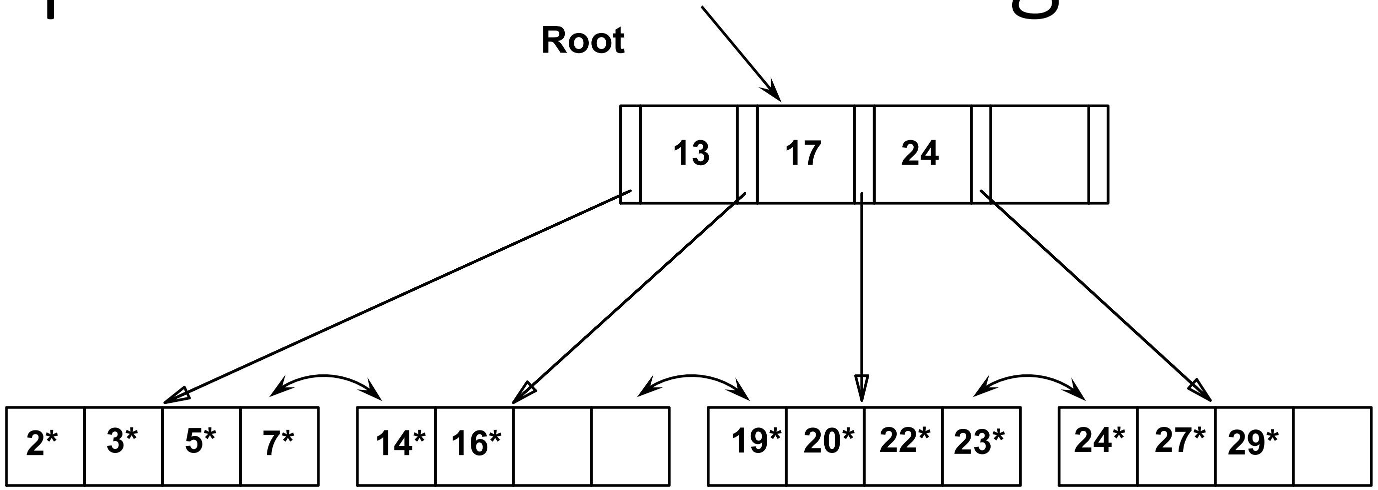
Search for 5*, 15*, all data entries >= 24* ...



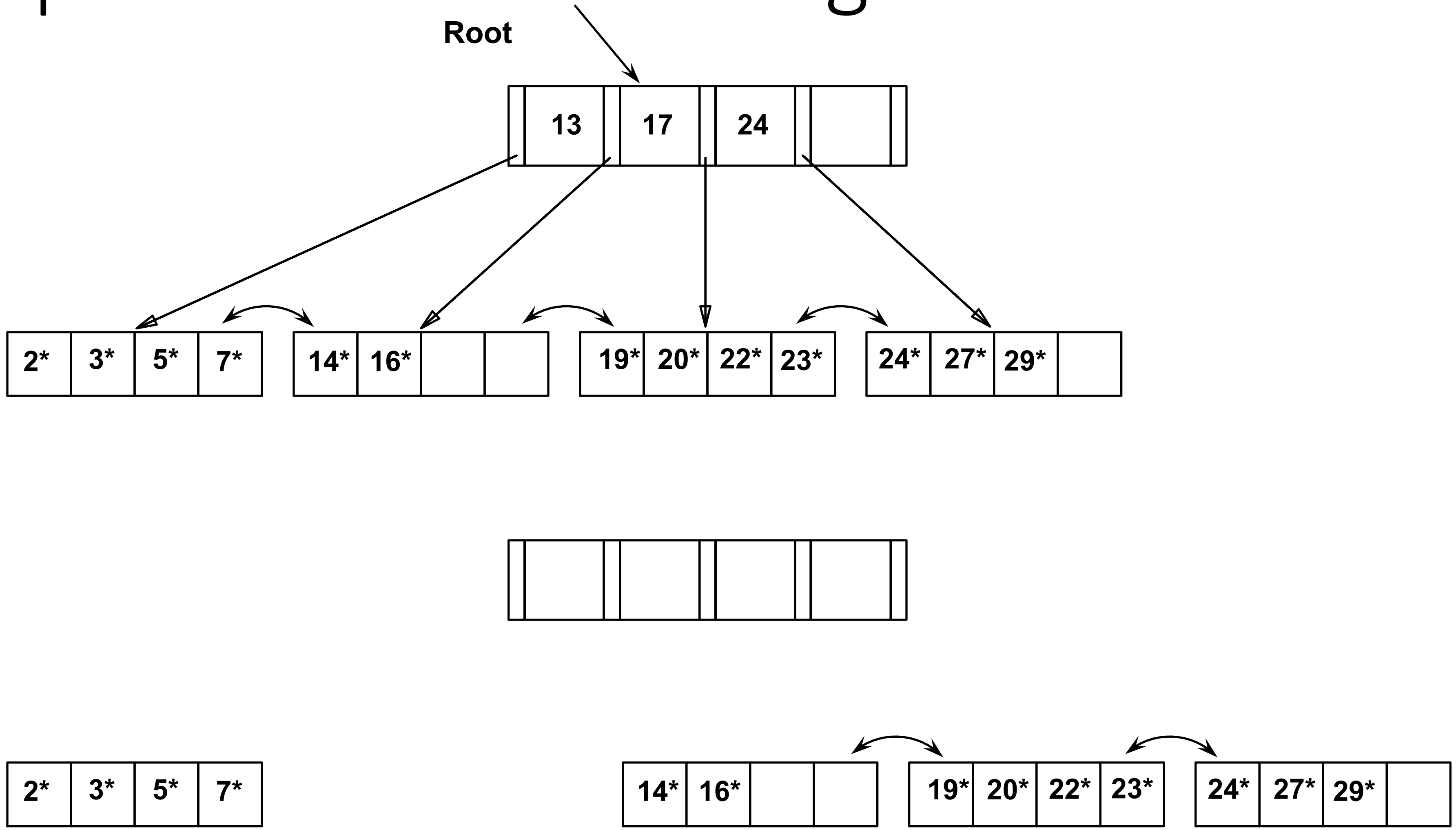
Based on the search for 15, we <u>know</u> it is not in the tree!*

BOSTON UNIVERSITY

Example B+ Tree - Inserting 8*

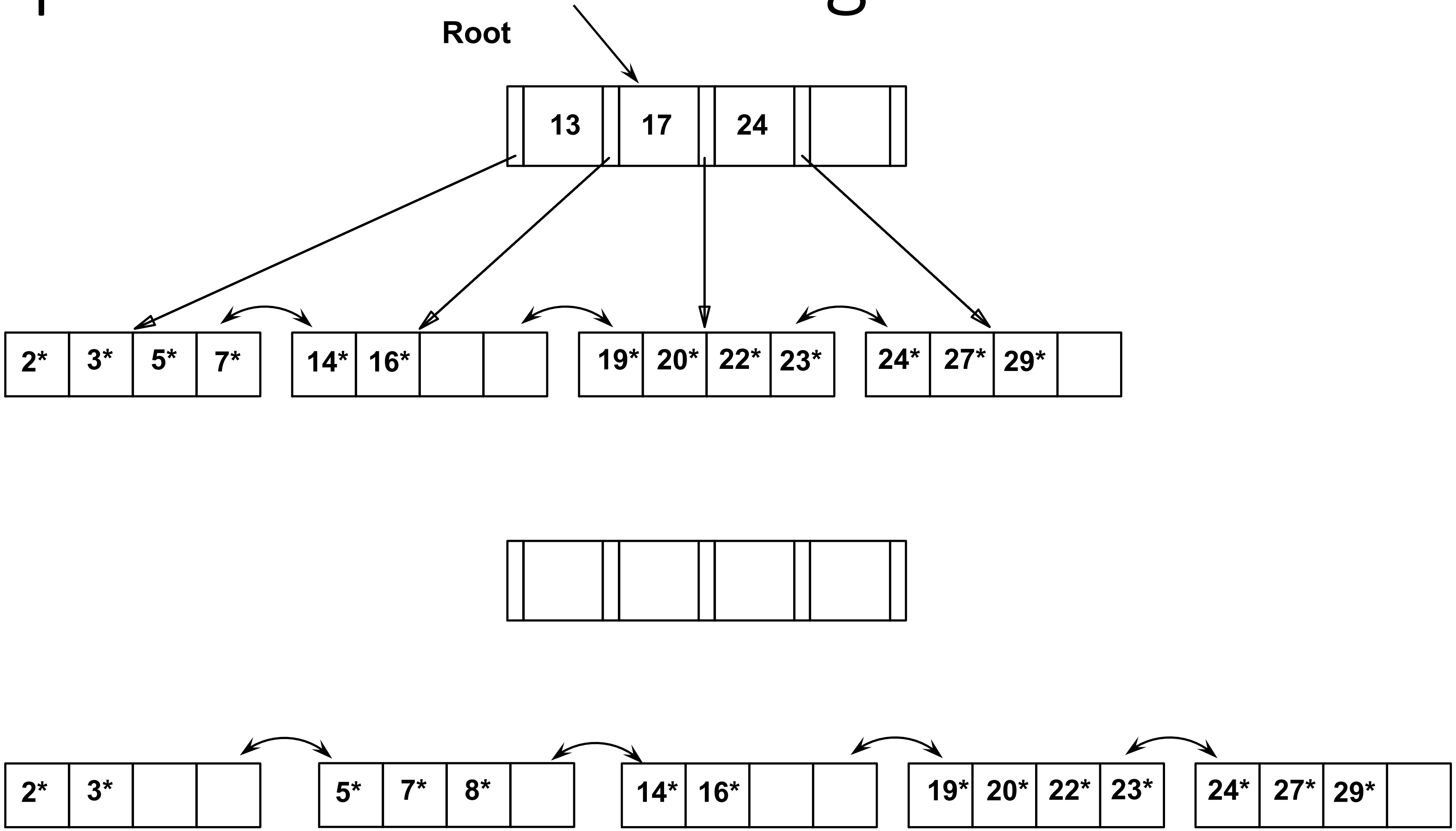


Example B+ Tree - Inserting 8*



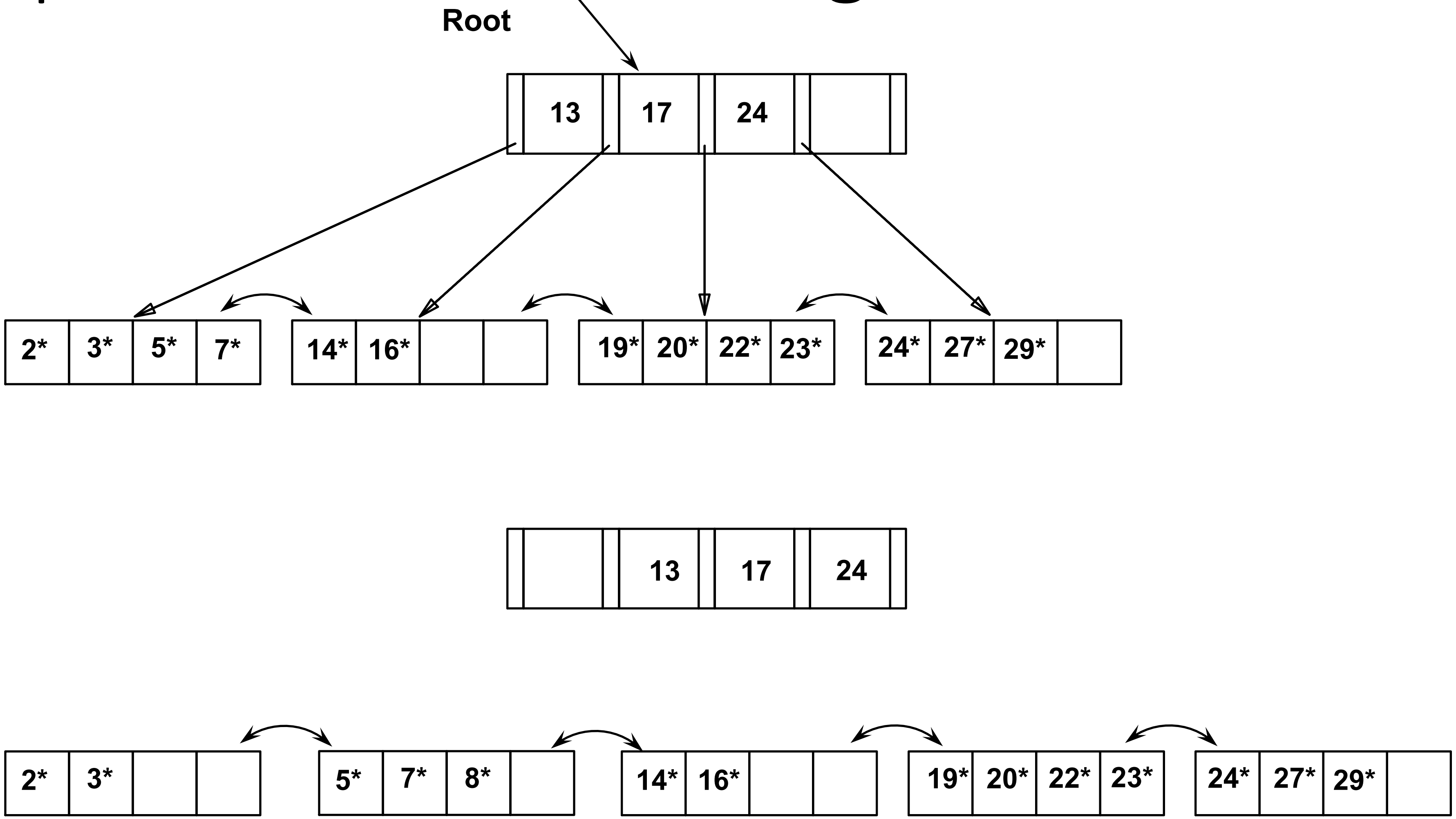
BOSTON UNIVERSITY

Example B+ Tree - Inserting 8*



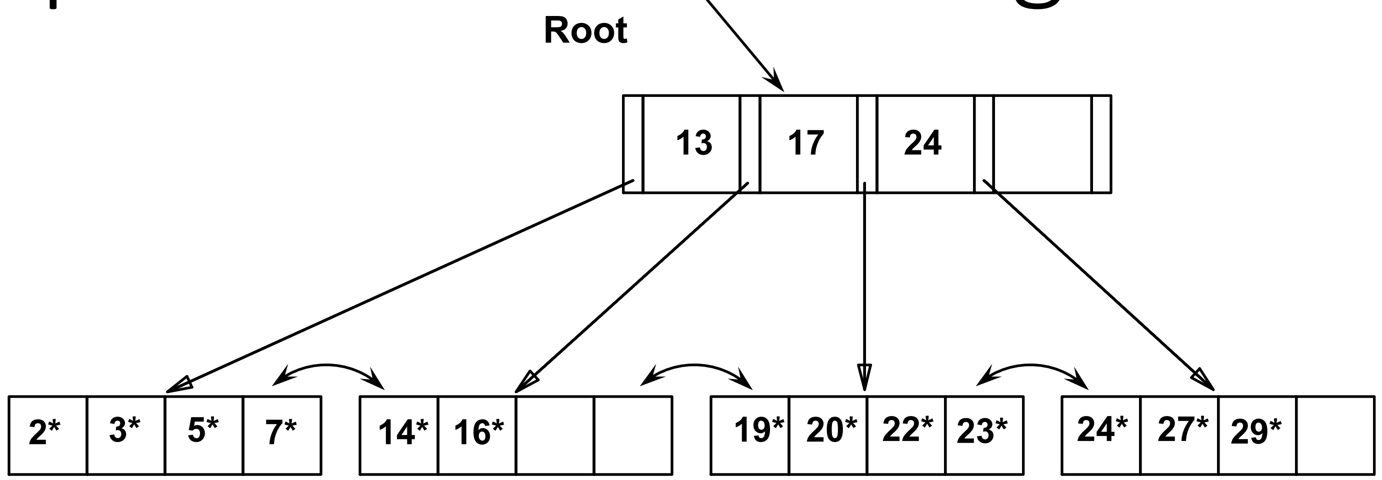
BOSTON UNIVERSITY

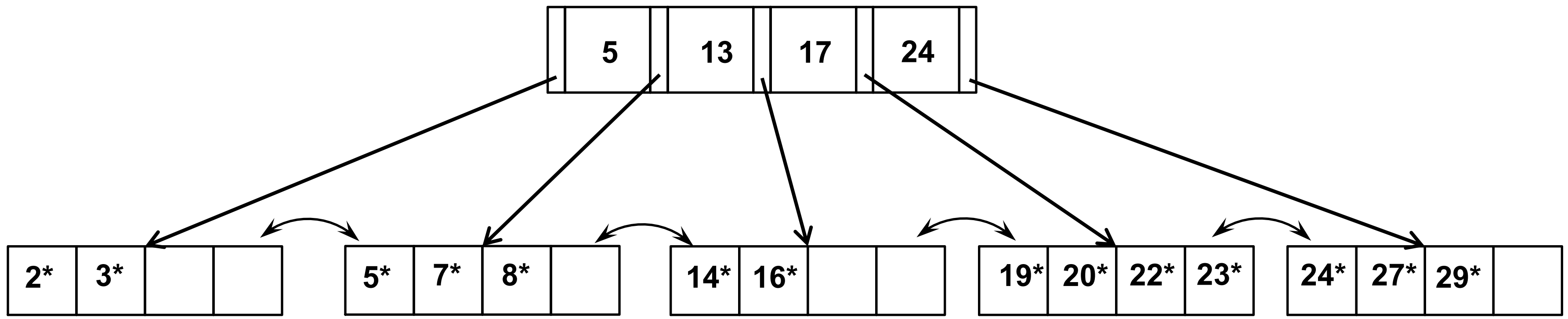
Example B+ Tree - Inserting 8*



BOSTON UNIVERSITY

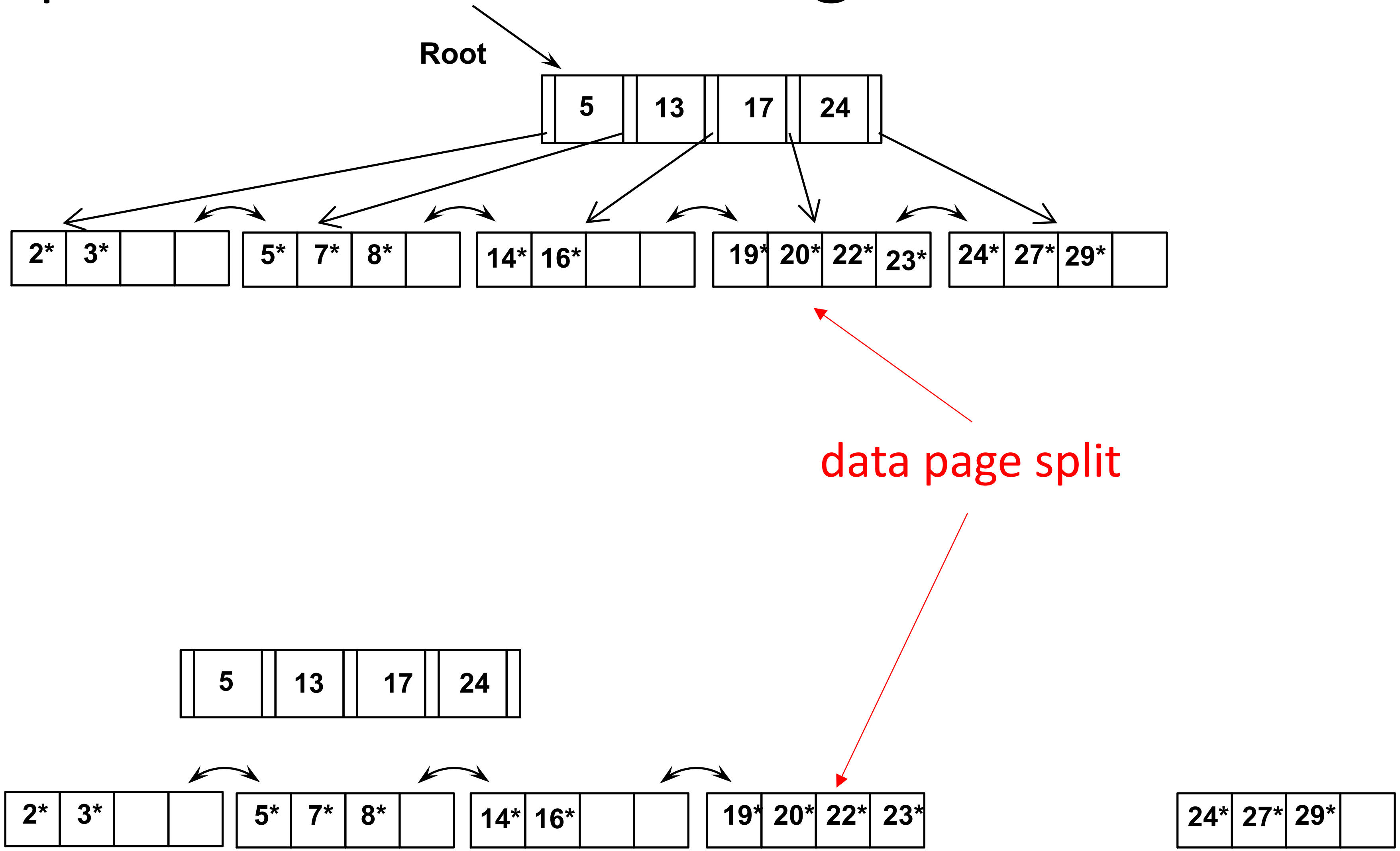
Example B+ Tree - Inserting 8*



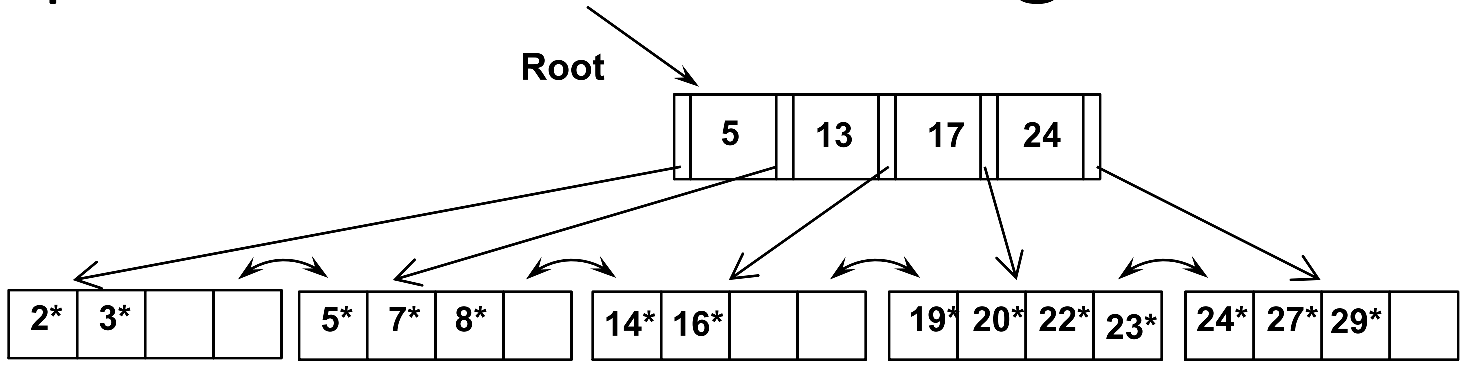


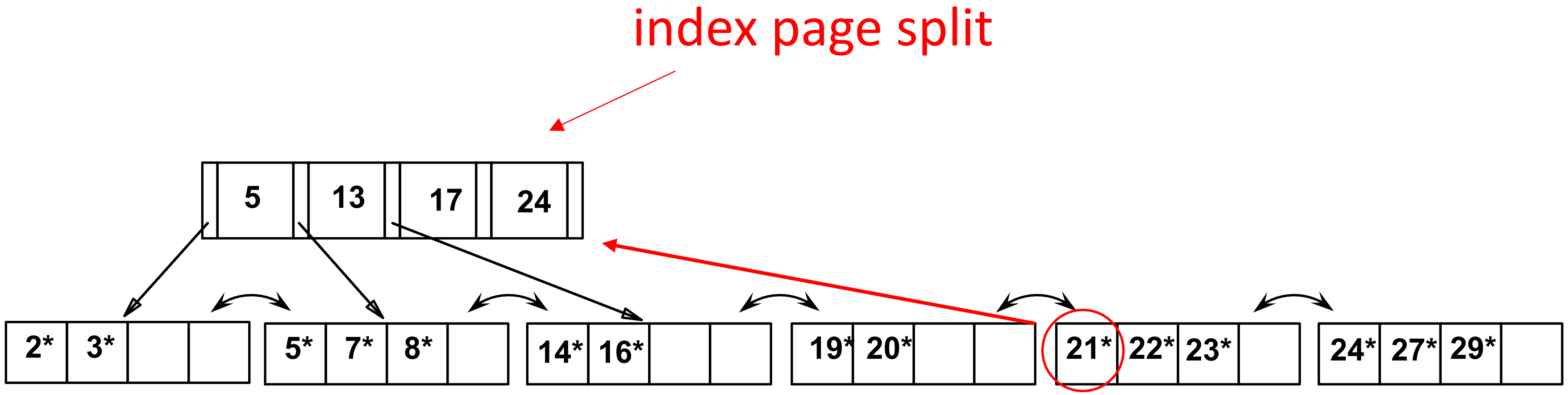
BOSTON UNIVERSITY

Example B+ Tree - Inserting 21*

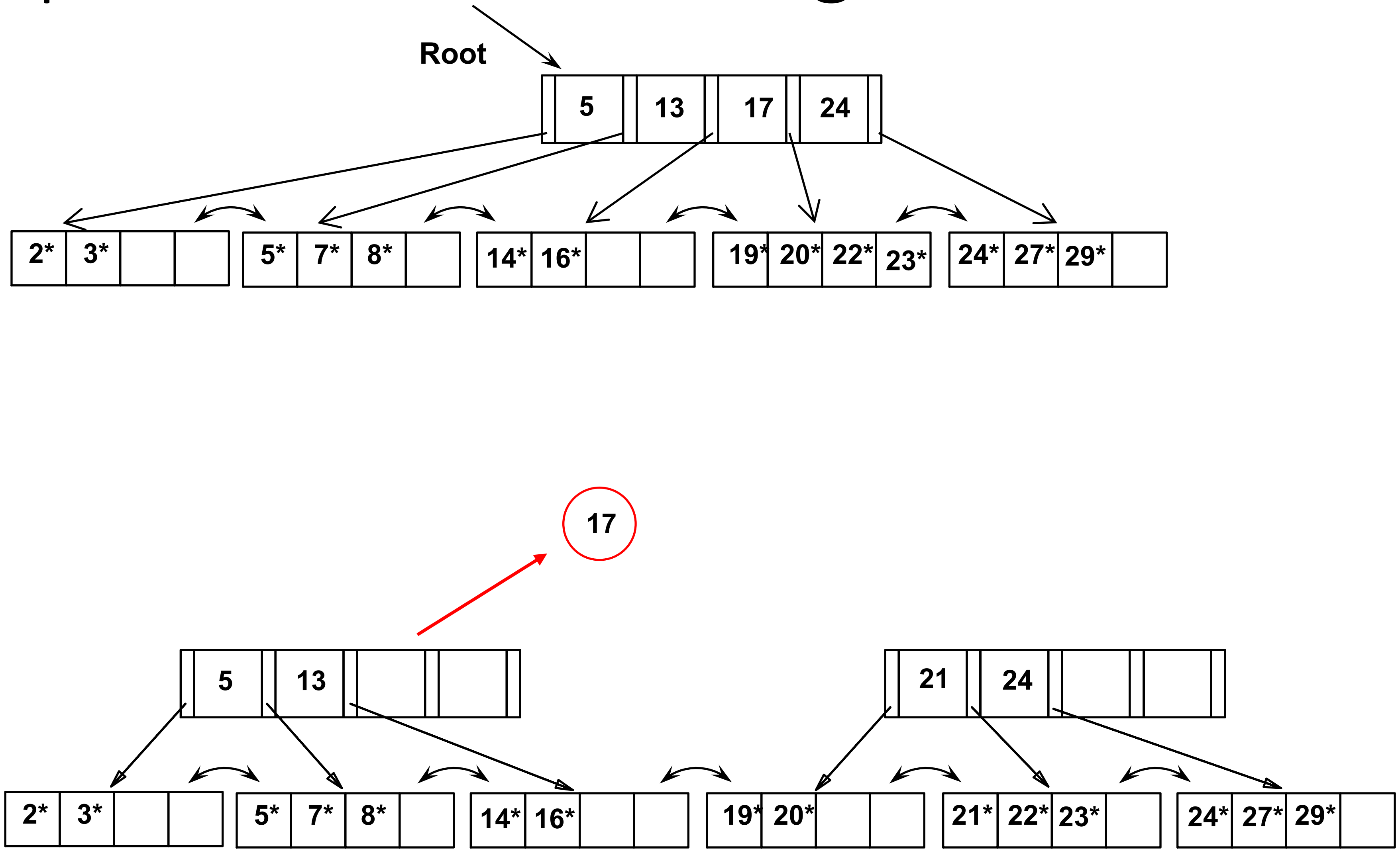


Example B+ Tree - Inserting 21*



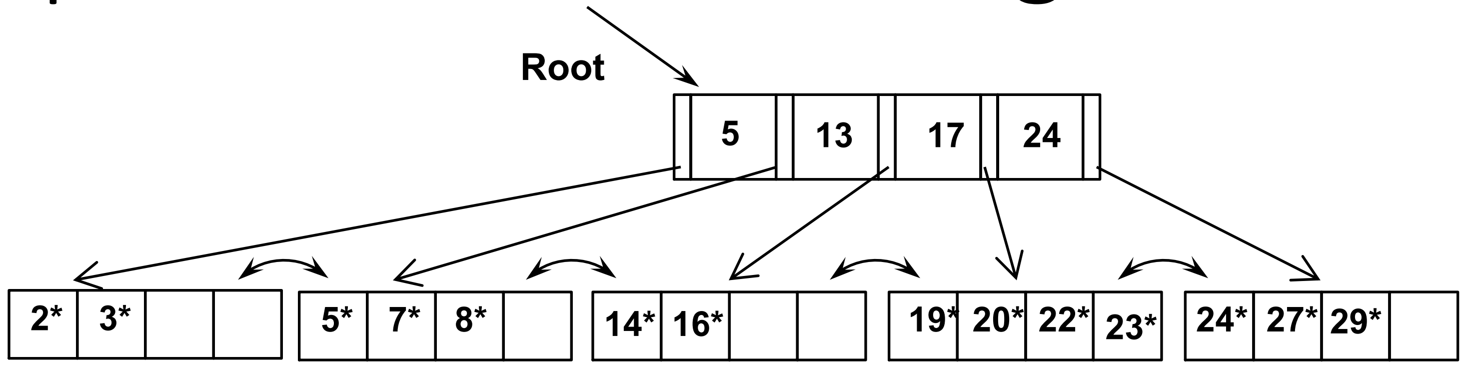


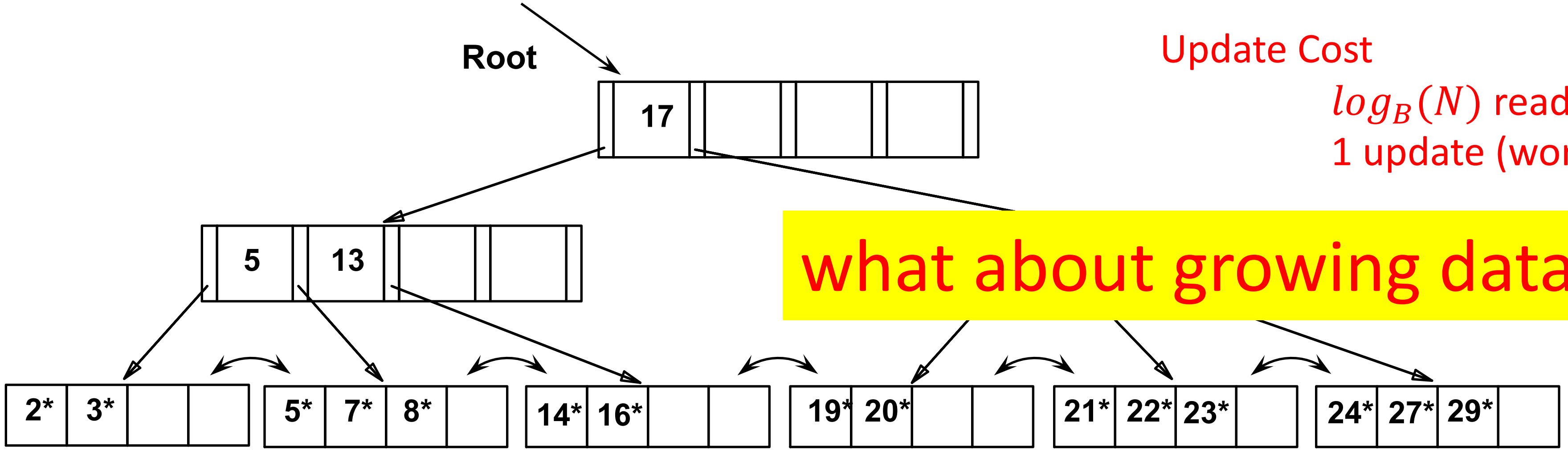
Example B+ Tree - Inserting 21*



BOSTON UNIVERSITY

Example B+ Tree - Inserting 21*





Read Cost: $log_B(N)$

Update Cost

$log_B(N)$ reads

1 update (worse case $log_B(N)$)

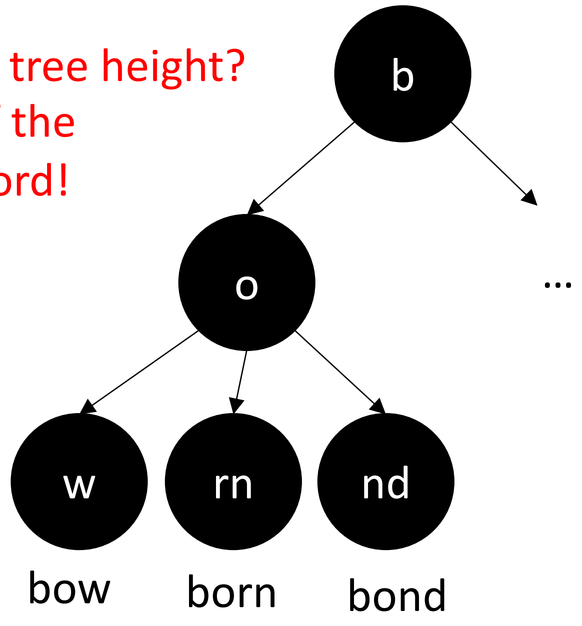
what about growing dataset size?

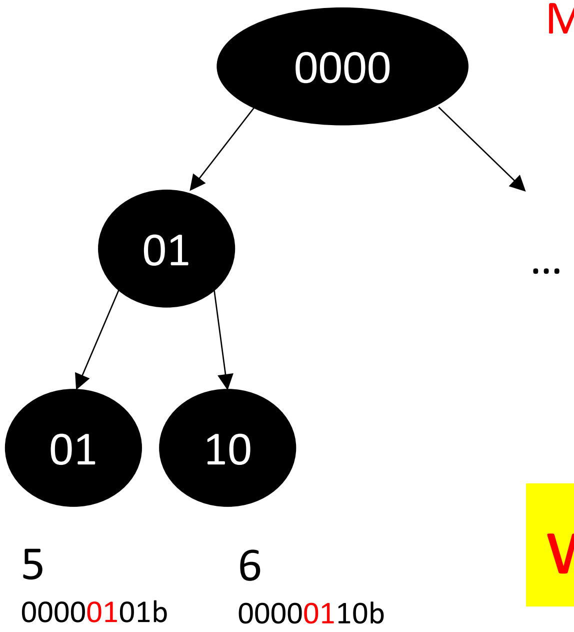
Radix Trees (special case of tries and prefix B-Trees)

Idea: use common prefixes for internal nodes to reduce size/height!

Binary representation of any domain can be used

Maximum tree height?
the size of the
longest word!





Maximum tree height?

8, that is, $log_2(\max\_domain\_value)$
fixed worst case!

what about data skew?

Bitmap Indexes

Column A	A=10	A=20	A=30
30	0	0	1
20	0	1	0
30	0	0	1
10	1	0	0
20	0	1	0
10	1	0	0
30	0	0	1
20	0	1	0

Speed & Size

- Compact representation of query result
- Query result is readily available

Bitvectors

- Can leverage fast Boolean operators
- Bitwise AND/OR/NOT faster than looping over meta data

Bitmap Indexes

Column A	A=10	A=20	A=30
30	0	0	1
20	0	1	0
30	0	0	1
10	1	0	0
20	0	1	0
10	1	0	0
30	0	0	1
20	0	1	0

encoded bitvectors

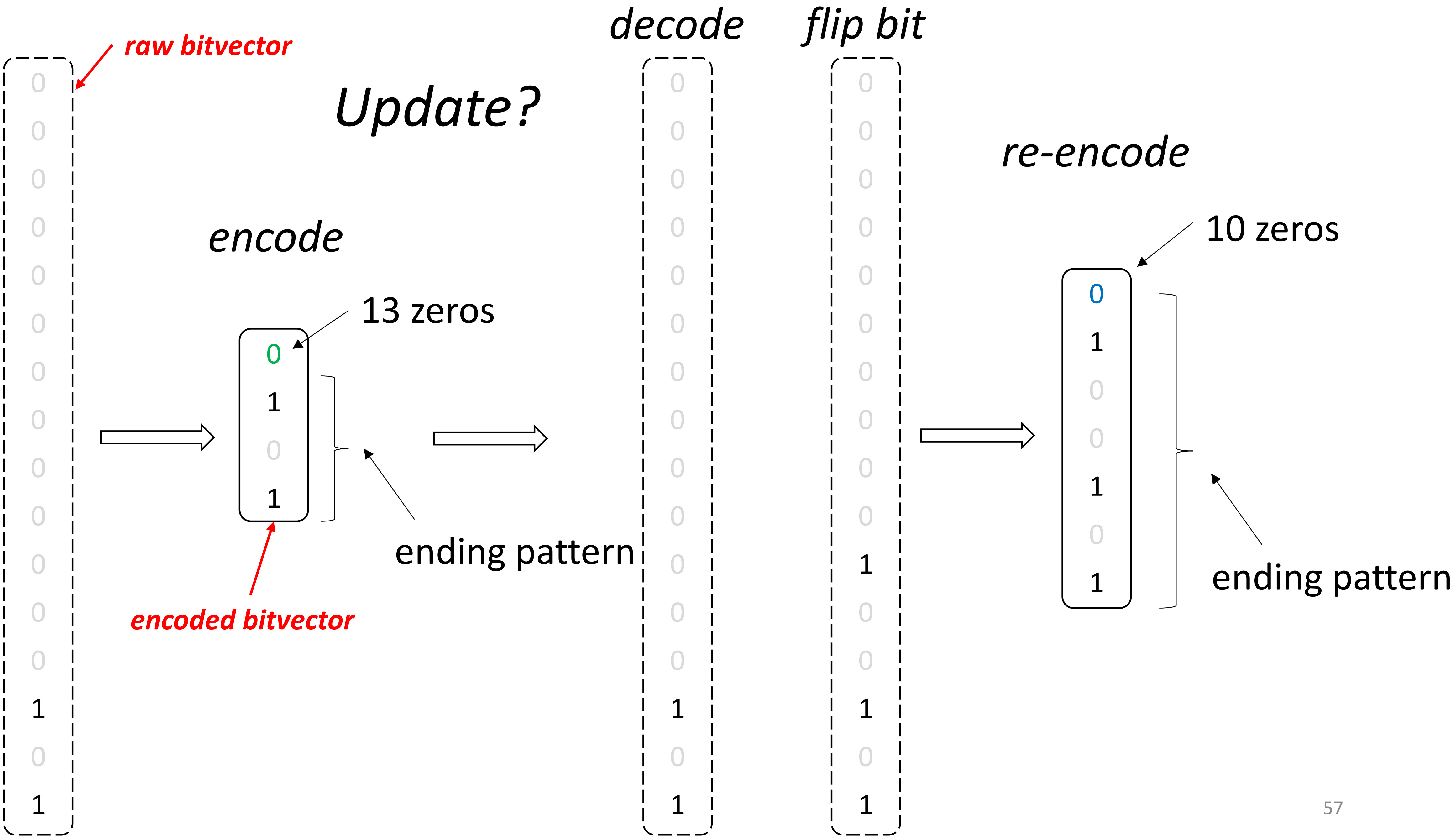
Index Size

- Space-inefficient for domains with large cardinality
- Addressed by bitvector encoding/compression

core idea: *run-length encoding* in prior work

what about updates?

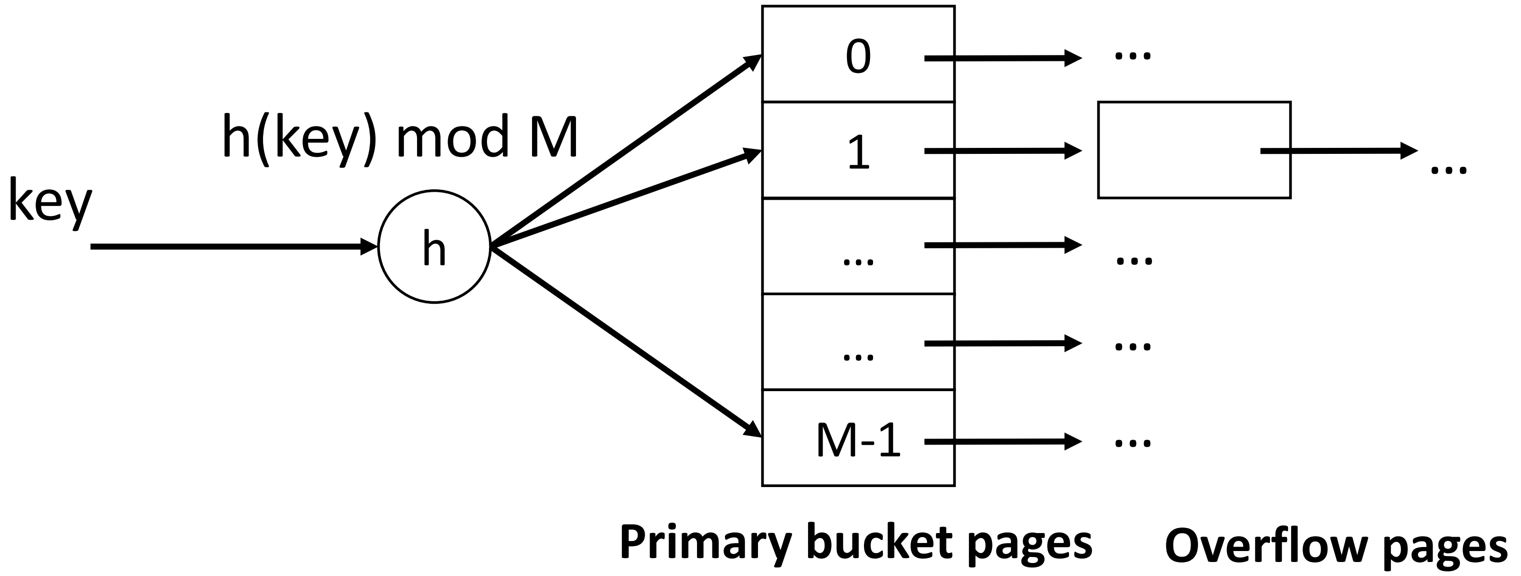
Update?


raw bitvector
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
encode
13 zeros
0 1 0 1
ending pattern
encoded bitvector
decode
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
flip bit
0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1
re-encode
10 zeros
0 1 0 0 1 0 1
ending pattern

Hash Indexes (static hashing)

#primary bucket pages fixed, allocated sequentially, never de-allocated; overflow pages if needed

h(k) mod M = bucket to insert data entry with key *k* (M: #buckets)



what if I have skew in the data set (or a bad hash function)?

Scan Accelerators

Zonemaps

Search for 25

Z1: [32,72]	Z2: [13,45]	Z3: [1,10]	Z4: [21,100]	Z5: [28,35]	Z6: [5,12]

BOSTON UNIVERSITY

Scan Accelerators

Zonemaps

Search for 25

Search for [5,11]

Z1: [32,72]	Z2: [13,45]	Z3: [1,10]	Z4: [21,100]	Z5: [28,35]	Z6: [5,12]

Scan Accelerators

Zonemaps

Z1: [32,72]	Z2: [13,45]	Z3: [1,10]	Z4: [21,100]	Z5: [28,35]	Z6: [5,12]

Search for 25

Search for [5,11]

Search for [31,46]

Scan Accelerators

Zonemaps

Z1: [32,72]	Z2: [13,45]	Z3: [1,10]	Z4: [21,100]	Z5: [28,35]	Z6: [5,12]

Search for 25

Search for [5,11]

Search for [31,46]

Scan Accelerators

Zonemaps

Z1: [32,72]	Z2: [13,45]	Z3: [1,10]	Z4: [21,100]	Z5: [28,35]	Z6: [5,12]

Search for 25

Search for [5,11]

Search for [31,46]

if data were sorted:

Z1: [1,15]	Z2: [16,30]	Z3: [31,50]	Z4: [50,67]	Z5: [68,85]	Z6: [85,100]

Search for 25

Search for [5,11]

Search for [31,46]

Scan Accelerators

Zonemaps

Z1: [32,72]	Z2: [13,45]	Z3: [1,10]	Z4: [21,100]	Z5: [28,35]	Z6: [5,12]

Search for 25

Search for [5,11]

Search for [31,46]

if data were sorted:

Z1: [1,15]	Z2: [16,30]	Z3: [31,50]	Z4: [50,67]	Z5: [68,85]	Z6: [85,100]

Search for 25

Search for [5,11]

Search for [31,46]

what if data is perfectly uniformly distributed?

Z1: [1,99]	Z2: [2,95]	Z3: [1,100]	Z4: [2,100]	Z5: [3,97]	Z6: [2,99]

What are the possible ***index designs***?

	Data Organization	**Point Queries**	**Short Range Queries**	**Long Range Queries**	**Data Skew**	**Updates**	**Affected by Physical Order**
B+ Trees	Range	✅	✅	✅	✅	✅	—
LSM Trees	Insertion & Sorted	✅	❌	✅	✅	✅	—
Radix Trees	Radix	✅	✅	✅	❌	—	—
Hash Indexes	Hash	✅	—	❌	❌	✅	—
Bitmap Indexes	None	✅	—	❌	—	❌	***no***
Scan Accelerators	None	❌	—	✅	✅	—	***yes***

BOSTON UNIVERSITY

Adaptive Data Organization: Database Cracking

idea: there is an ***ideal*** data organization

what is it (for a column of integers)?

sorted!

we can reach it ***eventually*** if we use the ***workload as a hint***

Adaptive Data Organization: Database Cracking

search < 15

32		32
19		19
11		11
	< 15	
6		6
123		123
55		55
12		12
78		78

Adaptive Data Organization: Database Cracking

	search < 15		search < 90	
32		11		11
19		6		6
11		12		12
	< 15		< 15	
6		32		32
123		19		19
55		123		123
12		55		55
			> 90	
78		78		78

Adaptive Data Organization: Database Cracking

		search < 15		search < 90		> 10 & < 30
32		11		11		11
					> 10	
19		6		6		6
11		12		12		12
	< 15		< 15		< 15	
6		32		32		32
					< 30	
123		19		19		19
55		123		55		55
12		55		78		78
			> 90		> 90	
78		78		123		123

Adaptive Data Organization: Database Cracking

		search < 15		search < 90		> 10 & < 30
32		11		11		6
					> 10	
19		6		6		11
11		12		12		12
	< 15		< 15		< 15	
6		32		32		19
					< 30	
123		19		19		32
55		123		55		55
12		55		78		78
			> 90		> 90	
78		78		123		123

what about updates/inserts?

Project Implementation

What to plan for the implementation (1/3)

Durable Database (open/close without losing state)

Components:

- Memory buffer (array, hashtable, B+ tree)
- Files (sorted levels/tiers)
- Fence pointers (**Zonemaps**)
- **Bloom filters**

What to plan for the implementation (2/3)

Durable Database (open/close without losing state)

Components:

- Memory buffer (search, read, write, unpin)
- Priority data structure
- Eviction policy

What to plan for the implementation (3/3)

API + basic testing and benchmarking available at:

LSM Implementation:

https://github.com/BU-DiSC/cs561_templatedb

with a Reference Bloom filter implementation

Bufferpool Implementation:

https://github.com/BU-DiSC/cs561_templatebufferpool

BOSTON UNIVERSITY

CS 561: Data Systems Architectures

Introduction to Indexing:

Trees, Tries, Hashing, Bitmap Indexes, Database Cracking

Prof. Manos Athanassoulis

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