

CS 561: Data Systems Architectures

Various forms of 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)

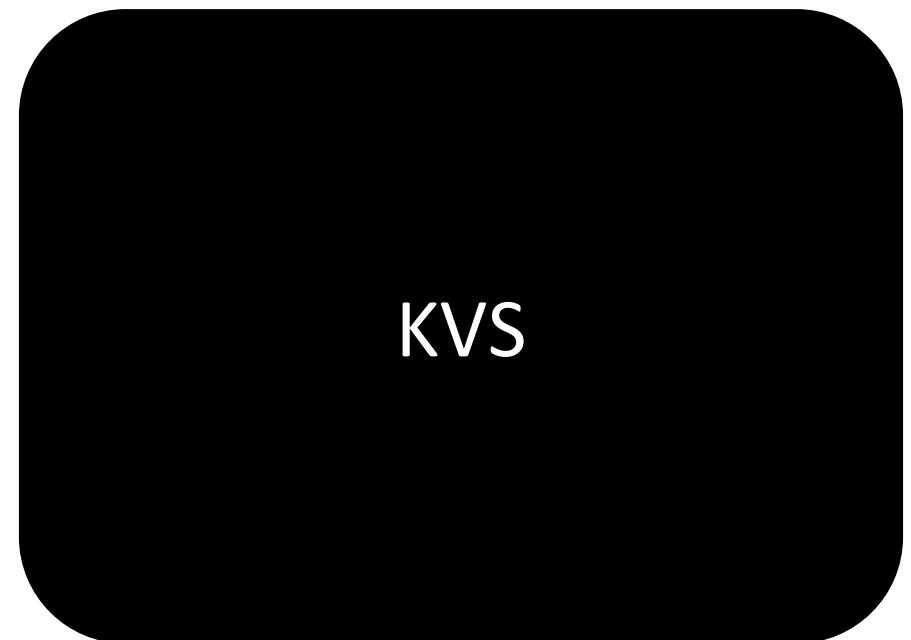
deletes the value associated with the key

get_range (key_start, key_end)

get_set(key1, key2, ...)

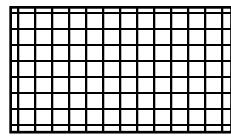
how to organize keys/values?

depends on the workload!

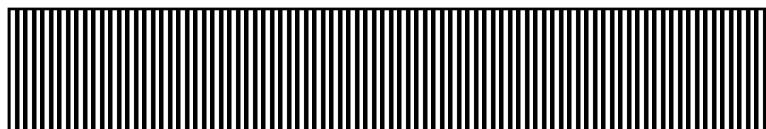


Recap: Key-Value Stores

inserts and point queries?

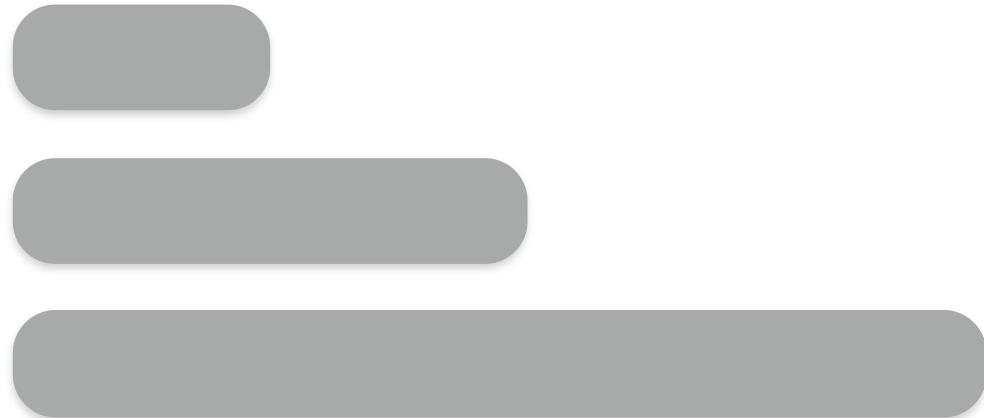


hash table



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

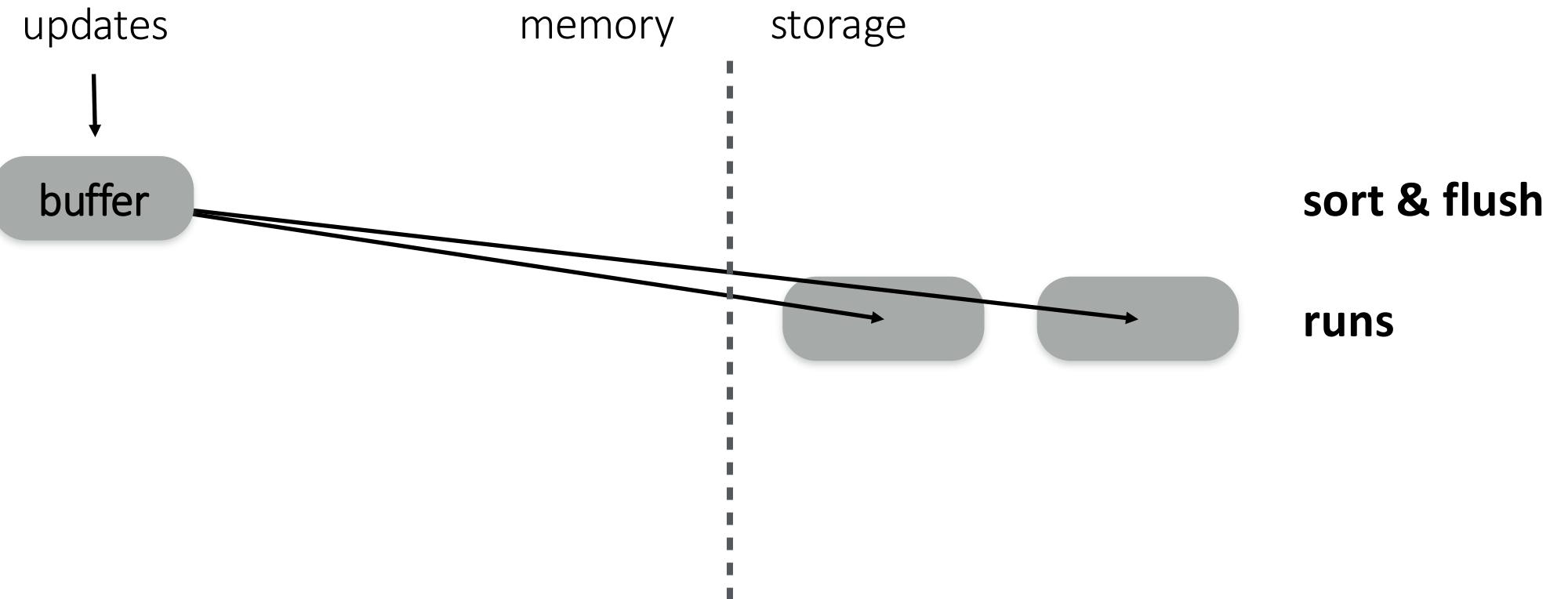


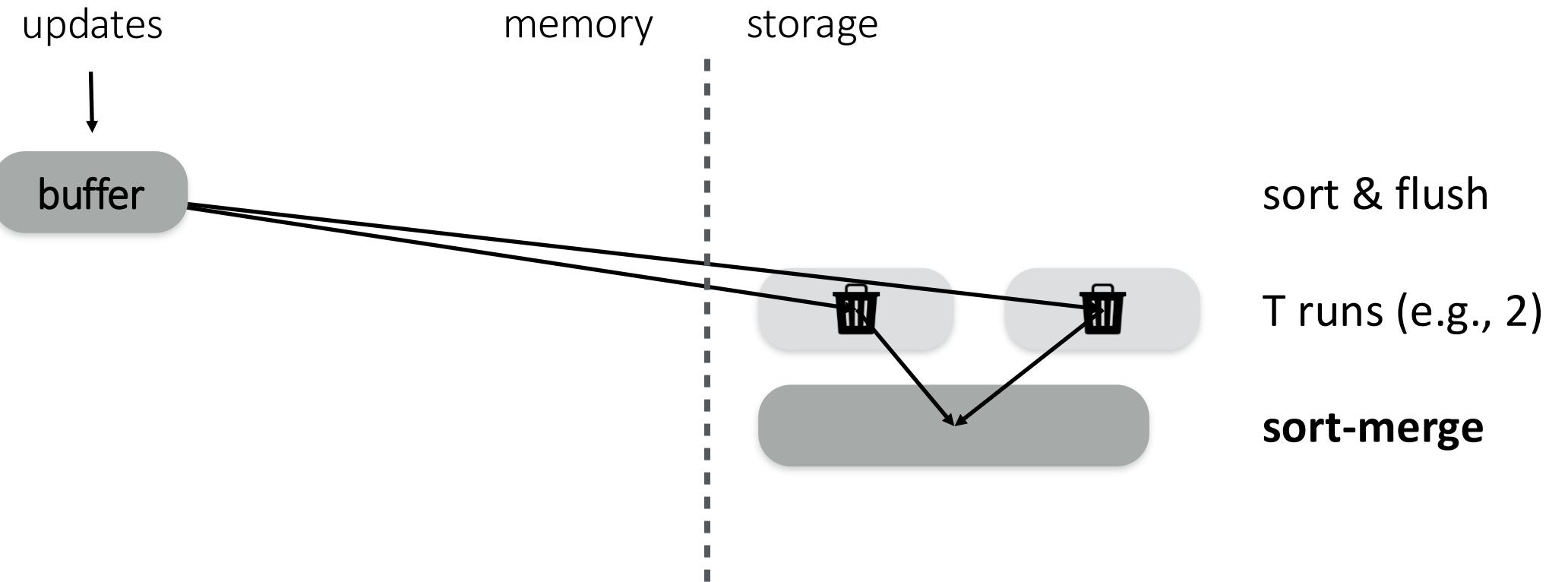
buffer

memory

storage







buffer

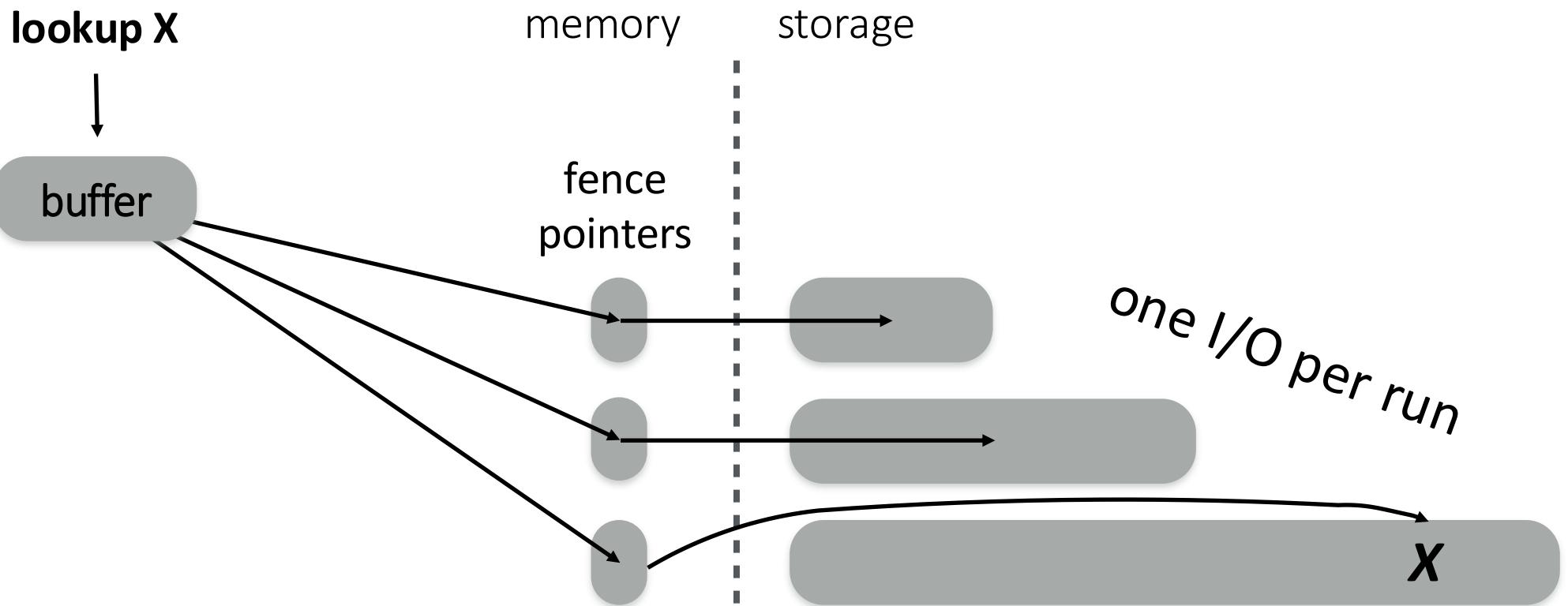
memory

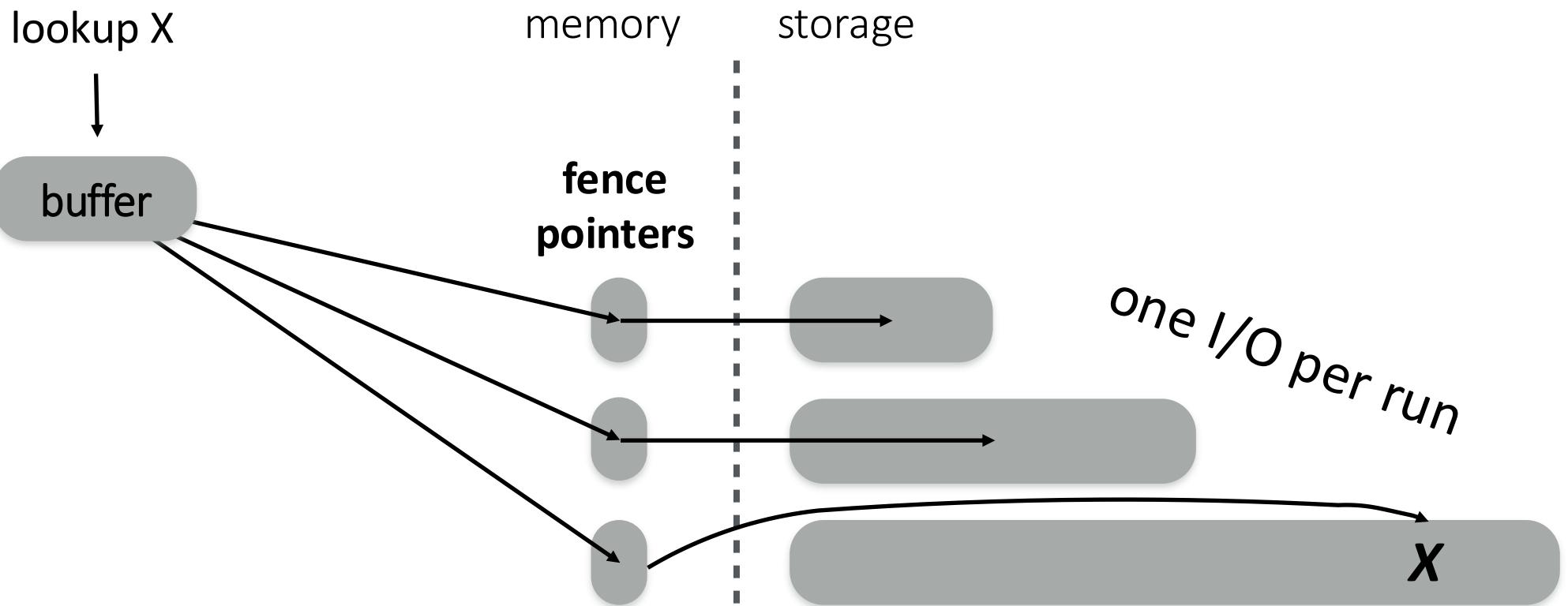
storage

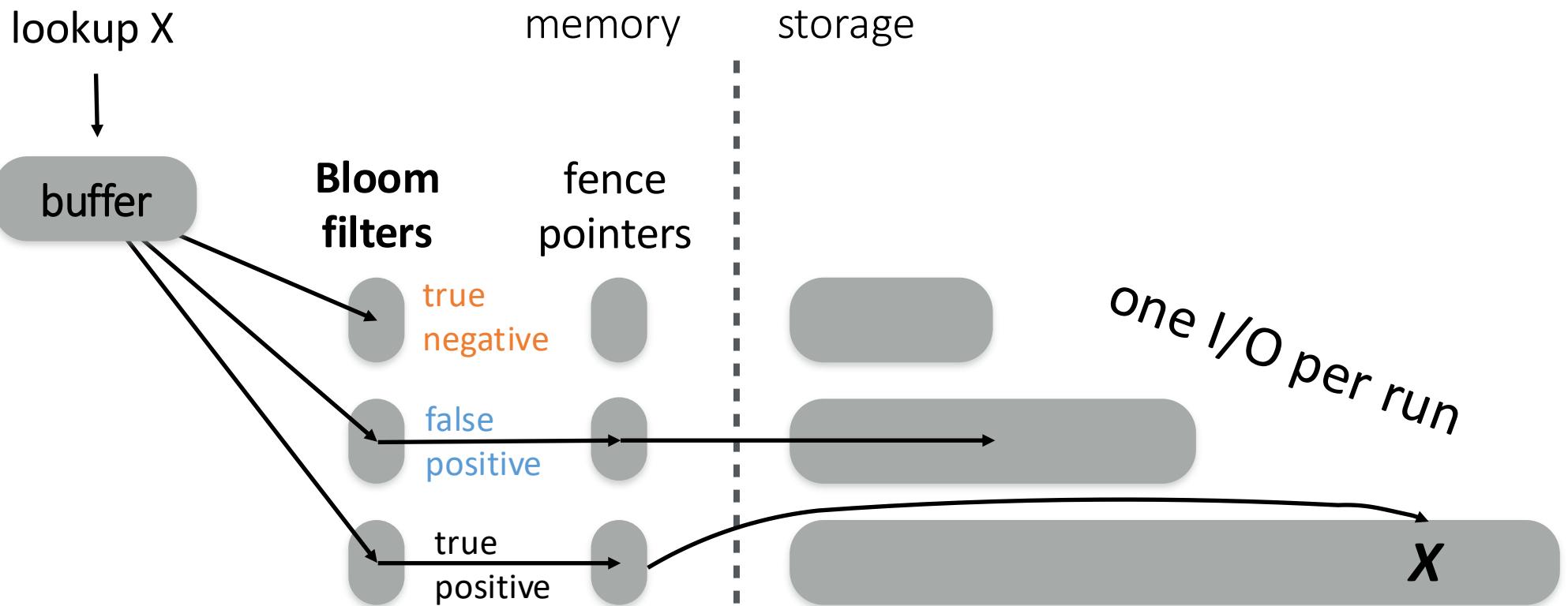
exponentially increasing sizes

$O(\log_T(N))$ levels

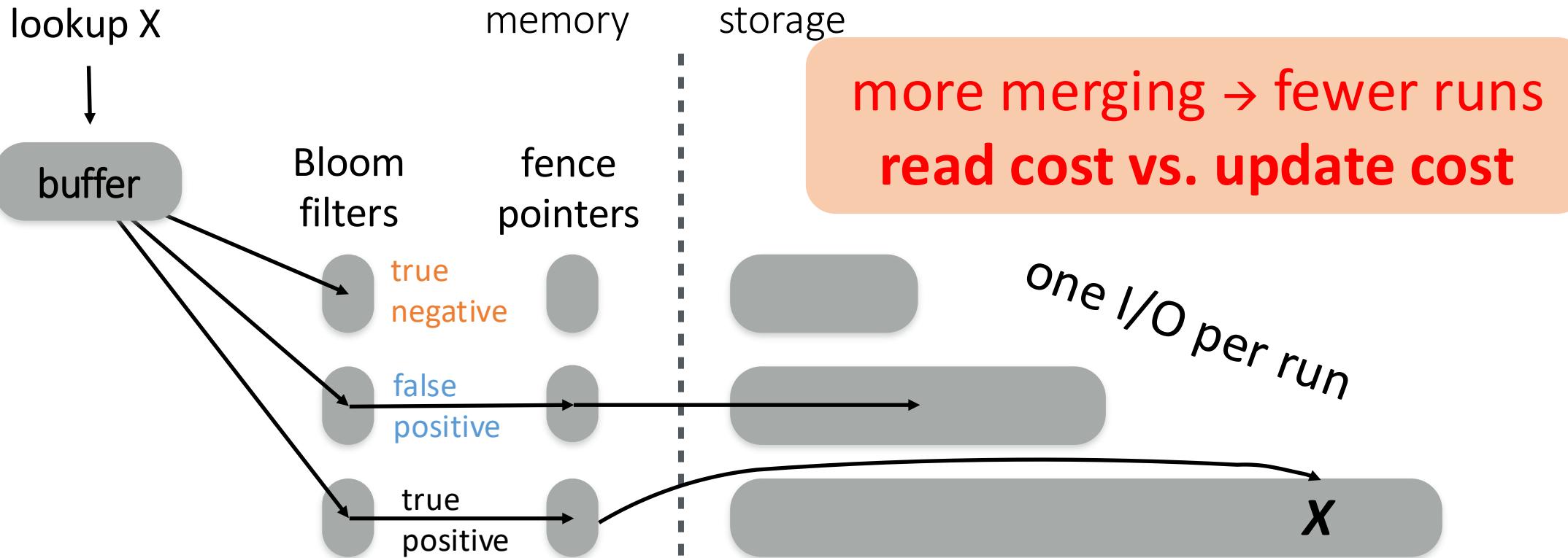




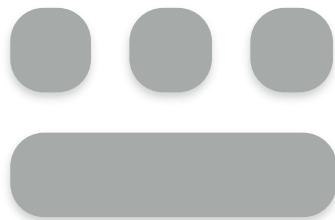




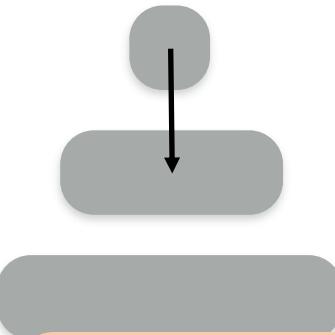
performance & cost trade-offs



tuning *reads* vs. *updates*



merge policy



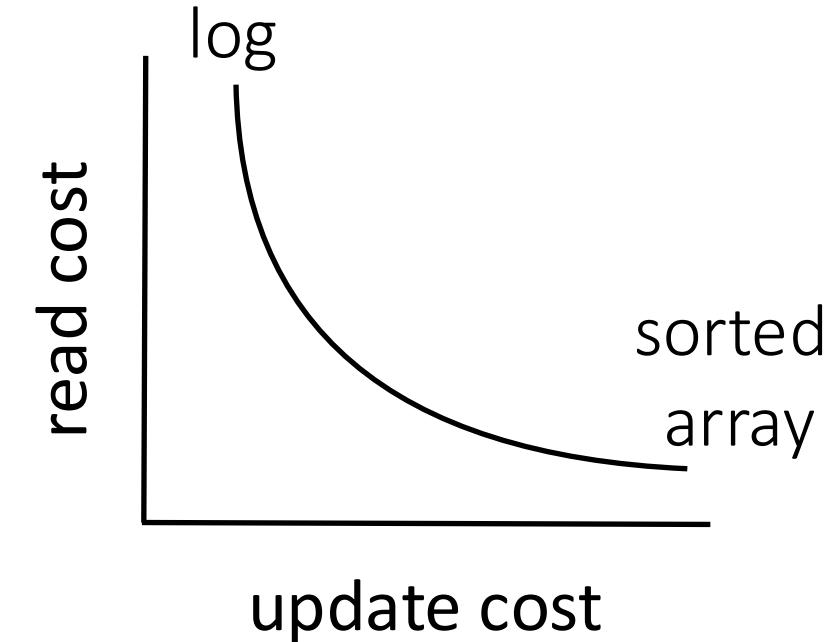
size ratio



log

LSM-Tree

sorted array



Merge Policies

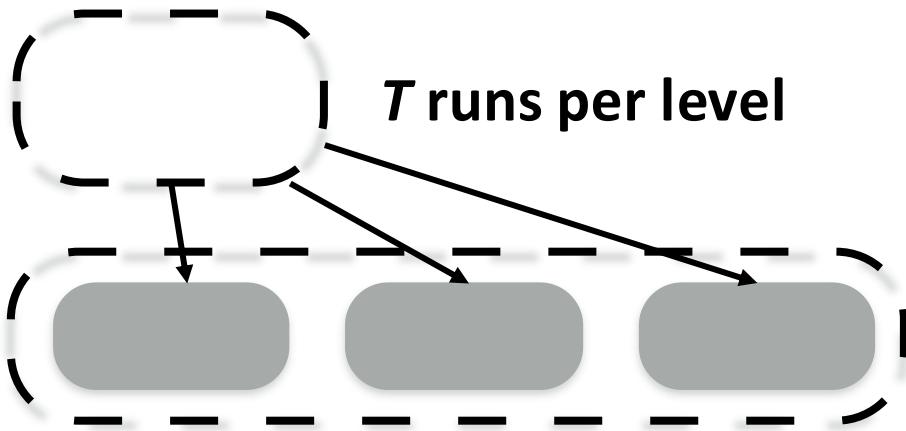
Tiering

write-optimized

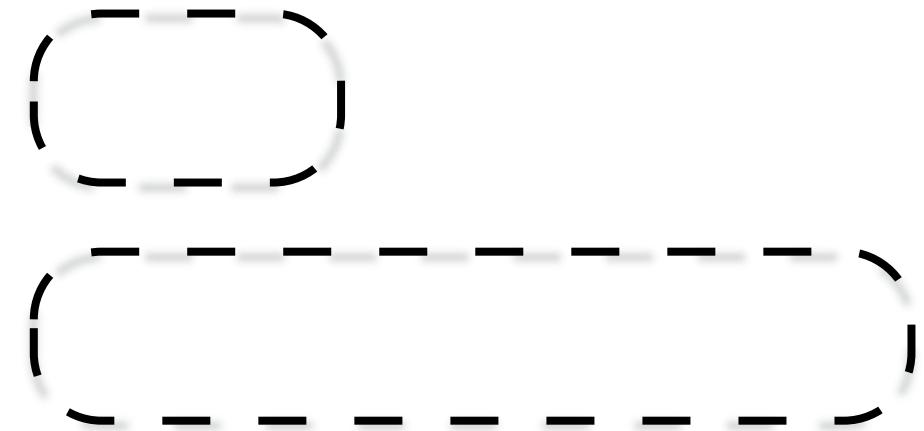
Leveling

read-optimized

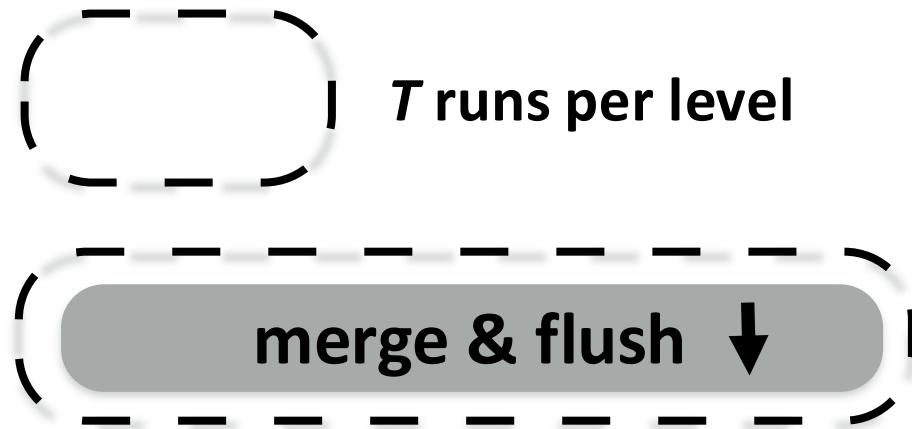
Tiering
write-optimized



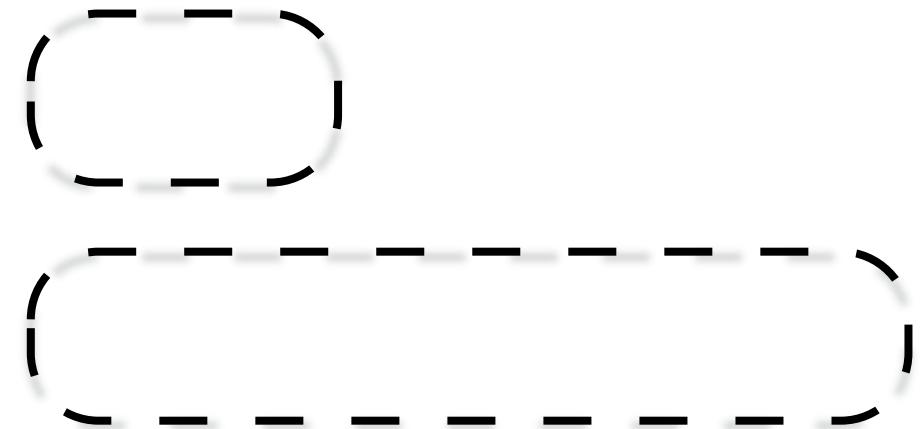
Leveling
read-optimized



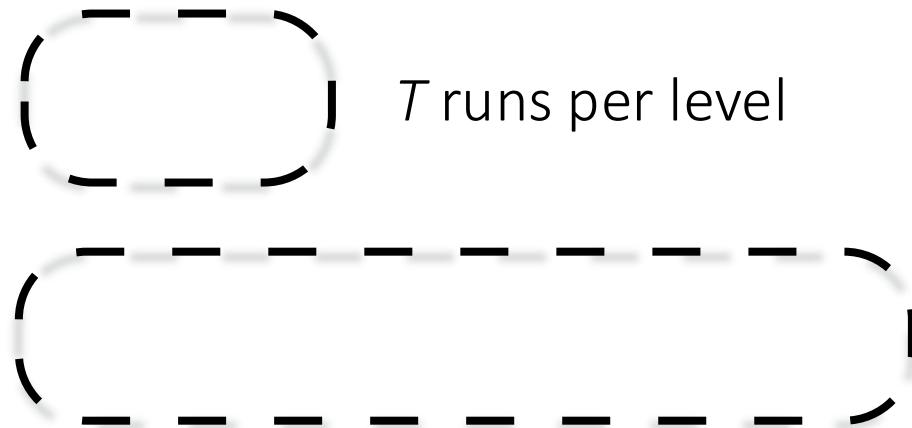
Tiering write-optimized



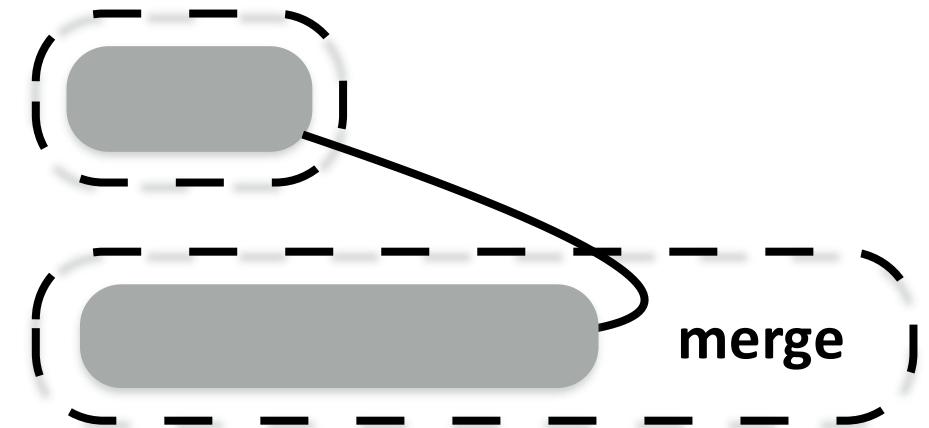
Leveling read-optimized



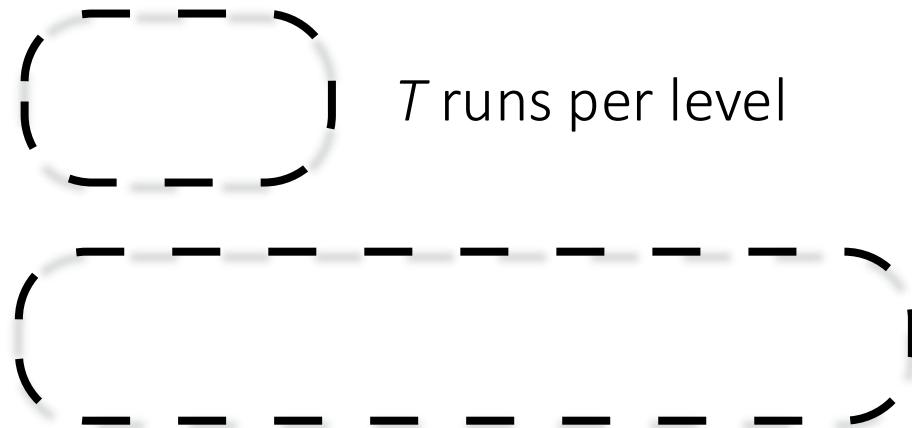
Tiering write-optimized



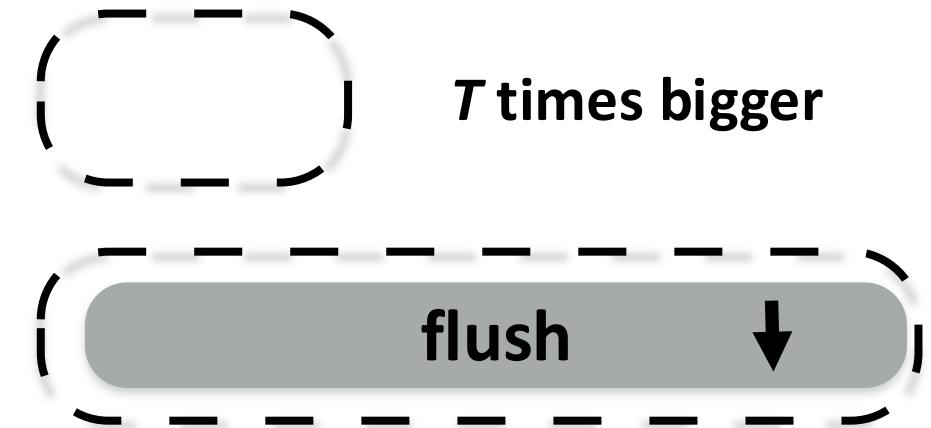
Leveling read-optimized



Tiering write-optimized



Leveling read-optimized



Tiering write-optimized



Leveling read-optimized



lookup cost:

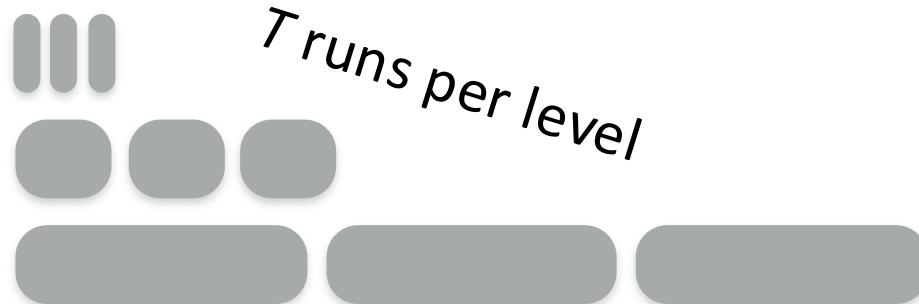
$$O(T \cdot \log_T(N) \cdot e^{-M/N})$$

runs per level levels false positive rate

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

levels false positive rate

Tiering write-optimized



Leveling read-optimized



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

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

update cost: $O(\log_T(N))$

↑
levels

$O(T \cdot \log_T(N))$

↑
merges per level ↑
levels

Tiering write-optimized



Leveling read-optimized



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

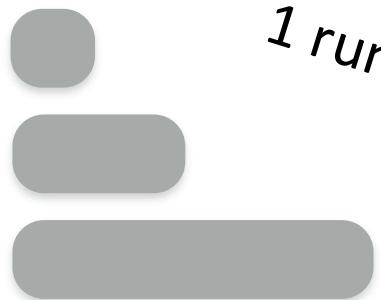
$O(\log_T(N) \cdot e^{-M/N})$

update cost: $O(\log_T(N))$

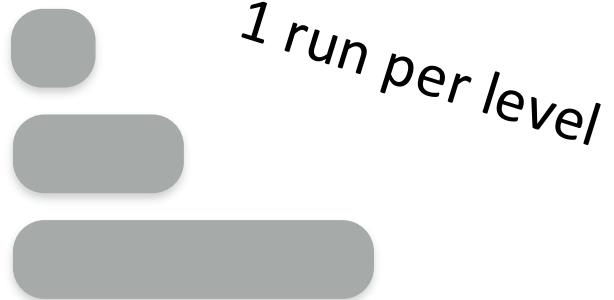
$O(T \cdot \log_T(N))$

for size ratio T

Tiering write-optimized



Leveling read-optimized



lookup cost:

$$O(\log_T(N) \cdot e^{-M/N}) = o(\log_T(N) \cdot e^{-M/N})$$

update cost:

$$O(\log_T(N)) = o(\log_T(N))$$

for size ratio T

Tiering write-optimized



Leveling read-optimized



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

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

update cost: $O(\log_T(N))$

$O(T \cdot \log_T(N))$

for size ratio T \nwarrow

Tiering
write-optimized

Leveling
read-optimized

$O(N)$ runs per level



log

1 run per level

sorted array

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

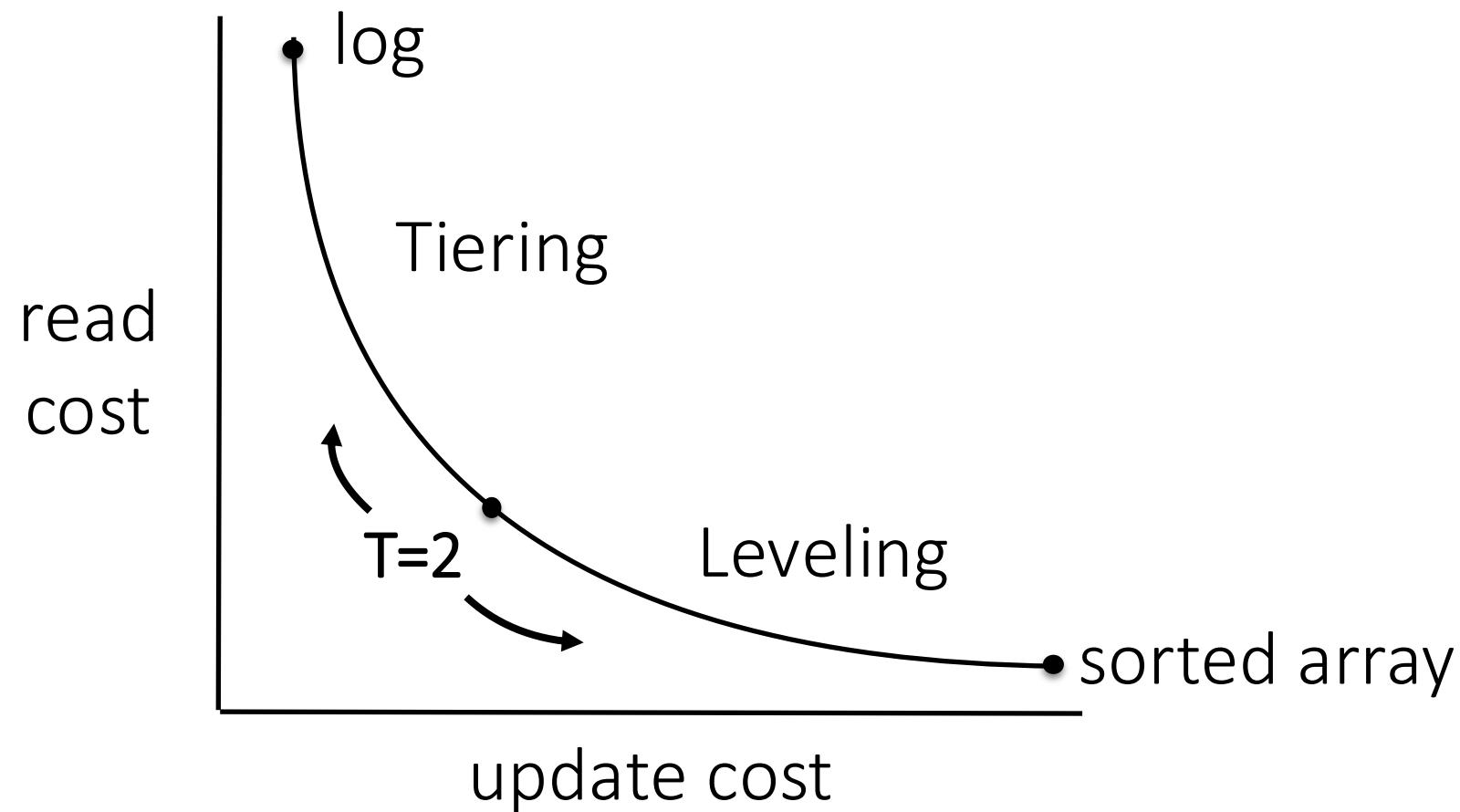
$O(e^{-M/N})$

update cost: $O(\log_N(N)) = o(1)$

$O(N \cdot \log_N(N)) = o(N)$

for size ratio T

N
↗

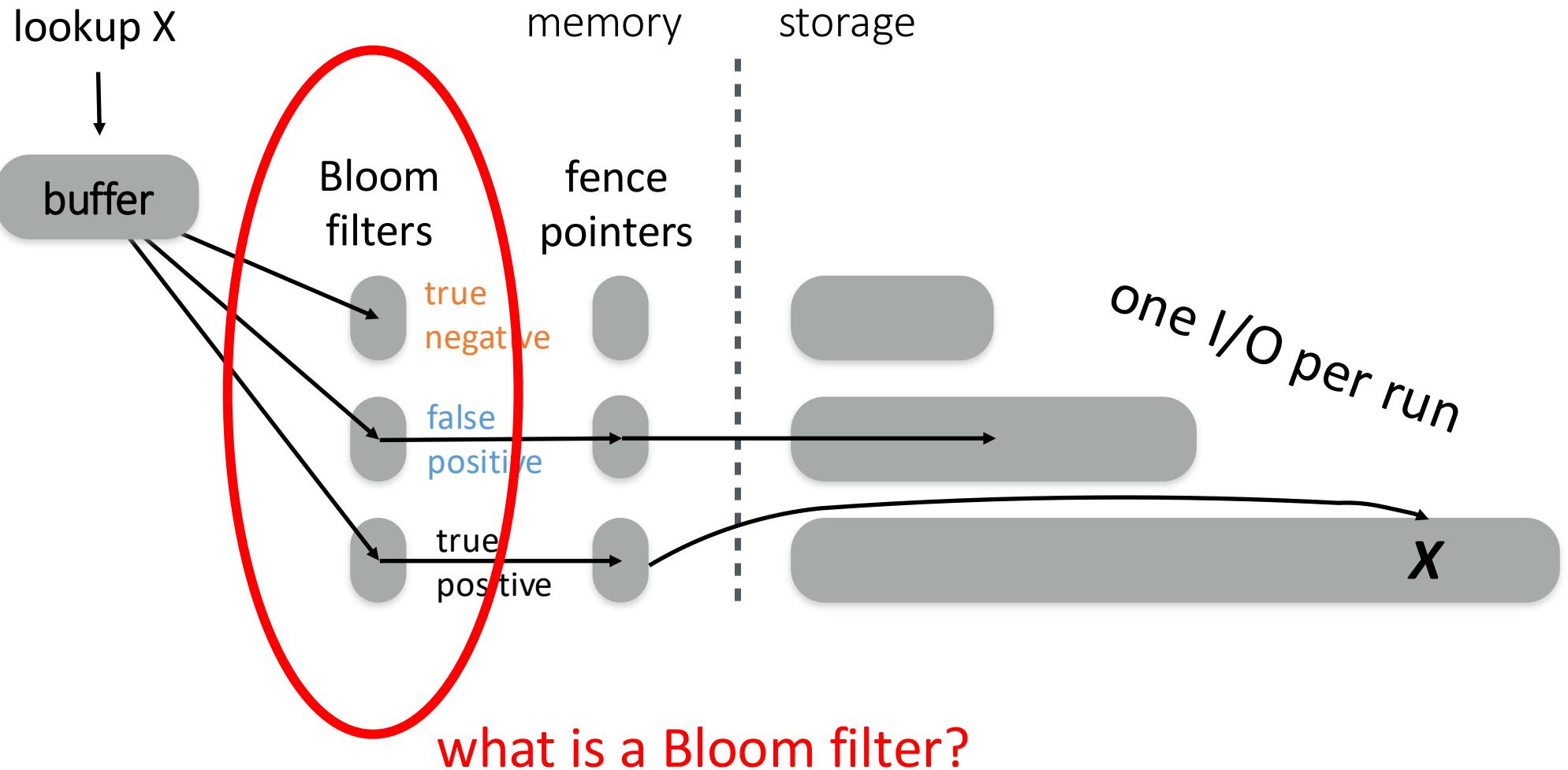


log

LSM-Tree

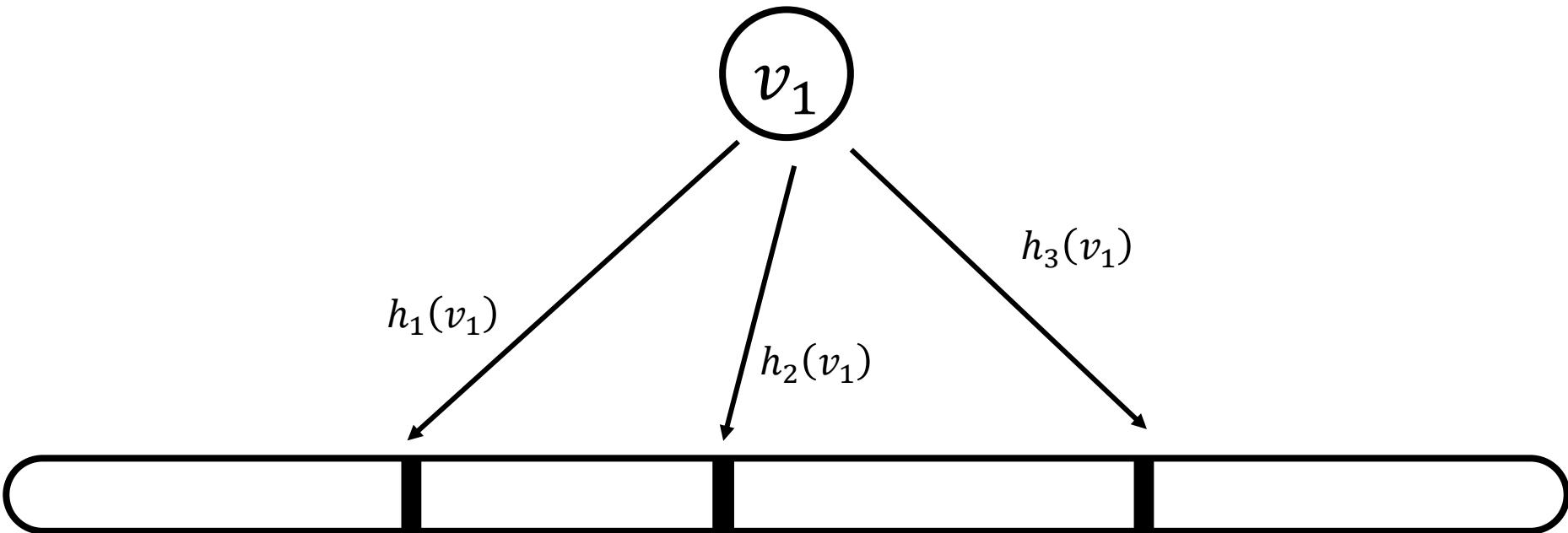
sorted array

performance & cost trade-offs

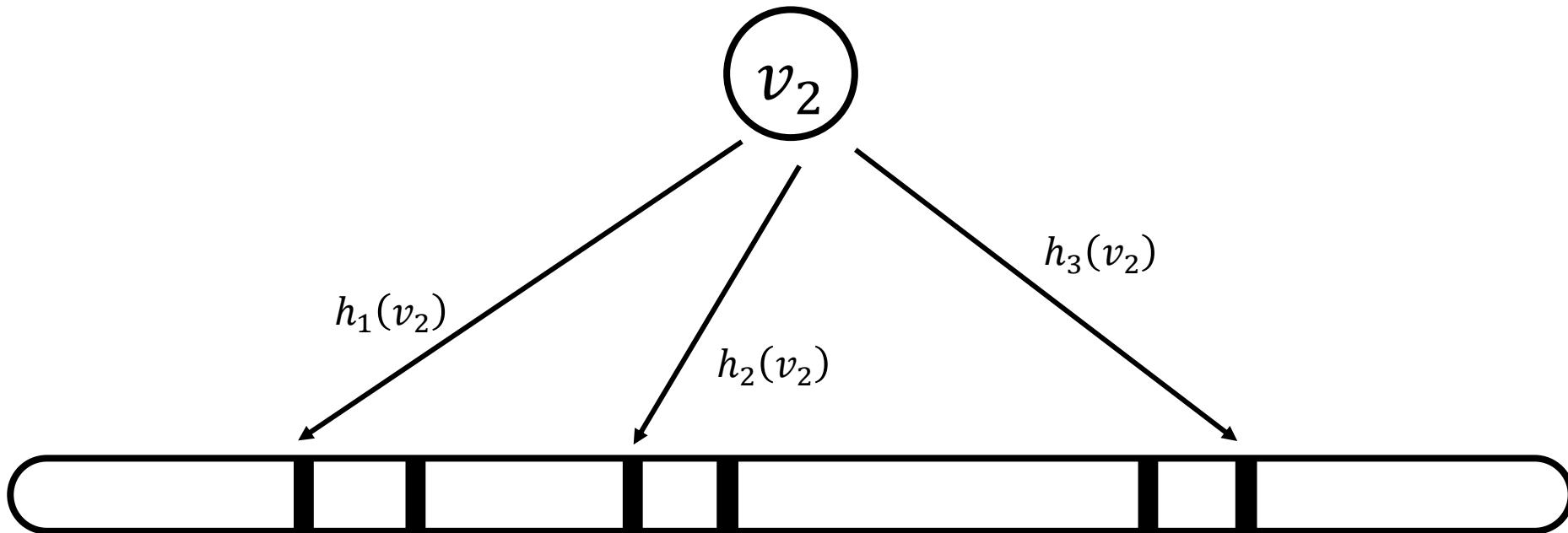


Details on Bloom filters

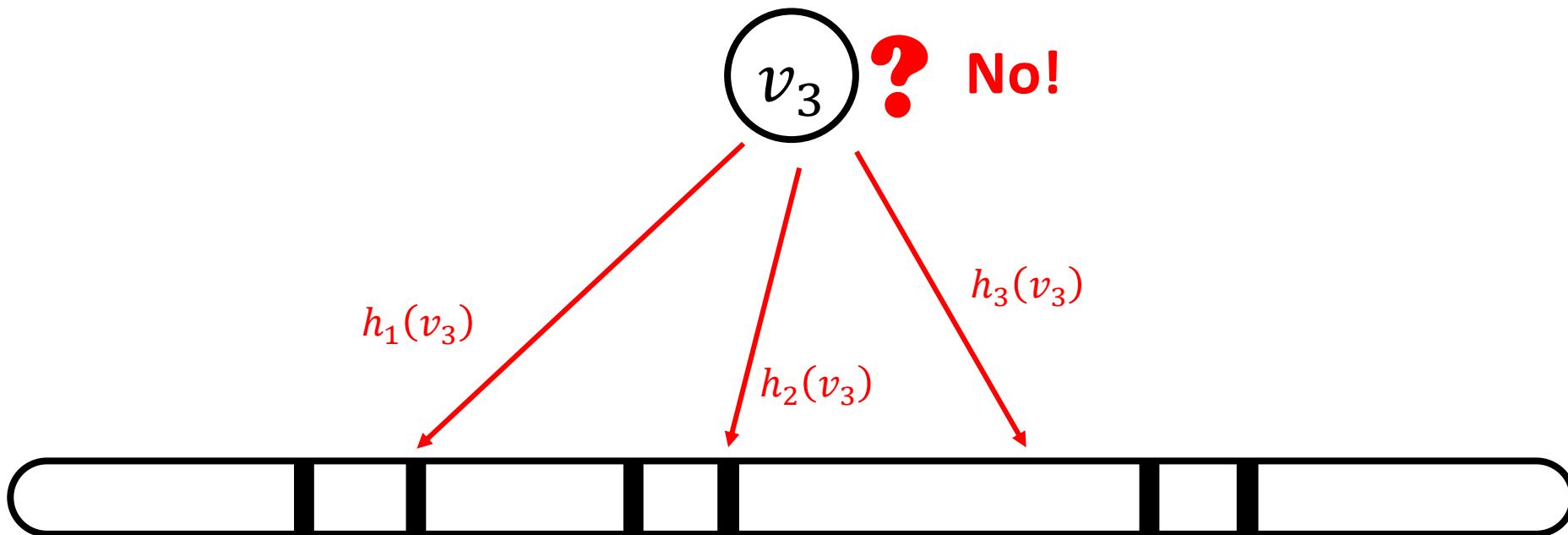
Inserting into a Bloom filter



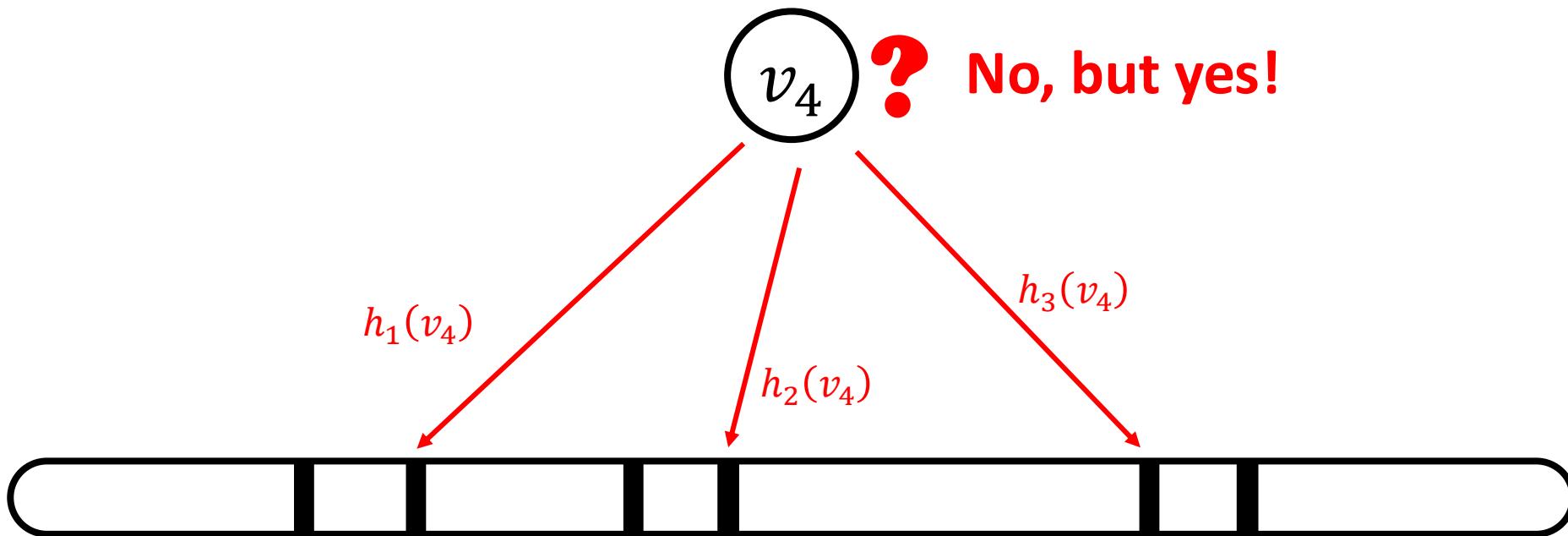
Inserting into a Bloom filter



Probing a Bloom filter (true negative)



Probing a Bloom filter (false positive)



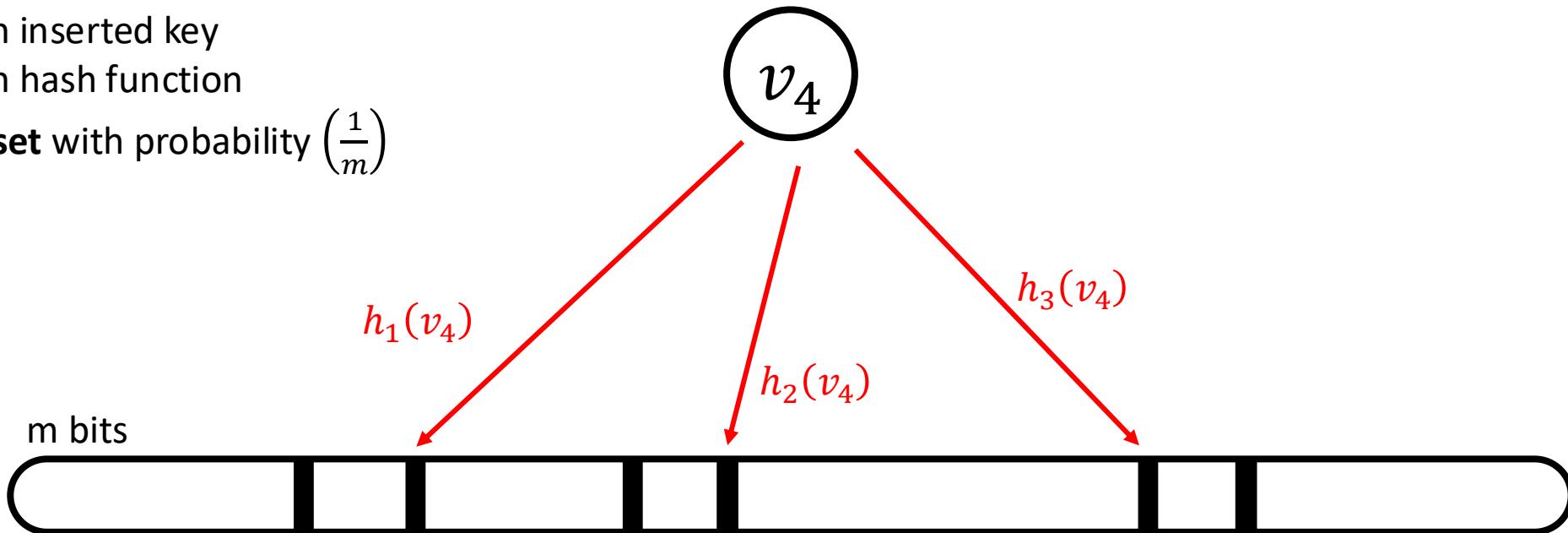
what is the probability of a false positive?

Bloom filter false positive

for each inserted key

for each hash function

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

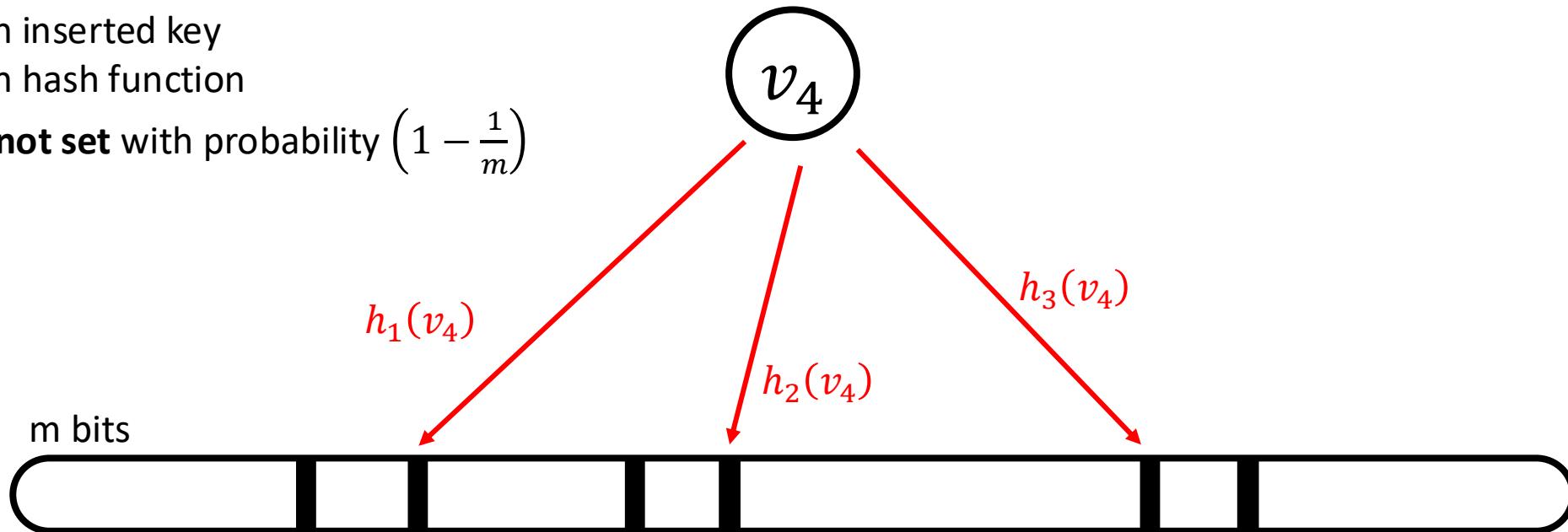


Bloom filter false positive

for each inserted key

for each hash function

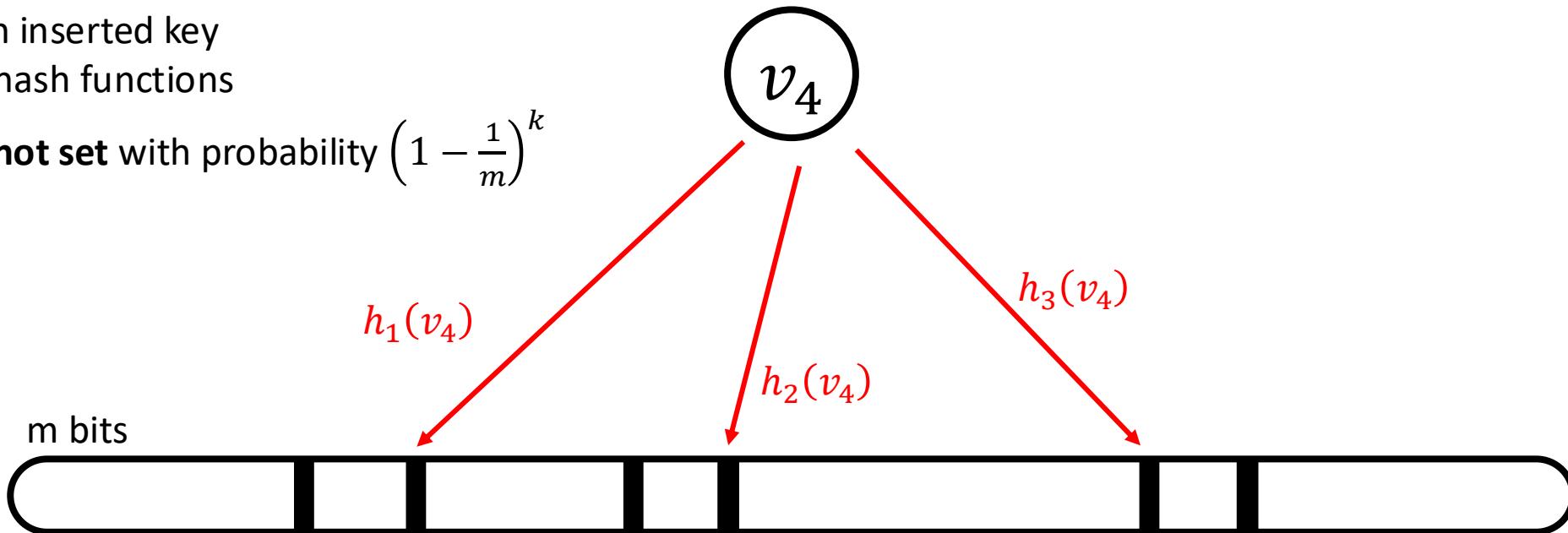
a bit is **not set** with probability $\left(1 - \frac{1}{m}\right)$



Bloom filter false positive

for each inserted key
after k hash functions

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

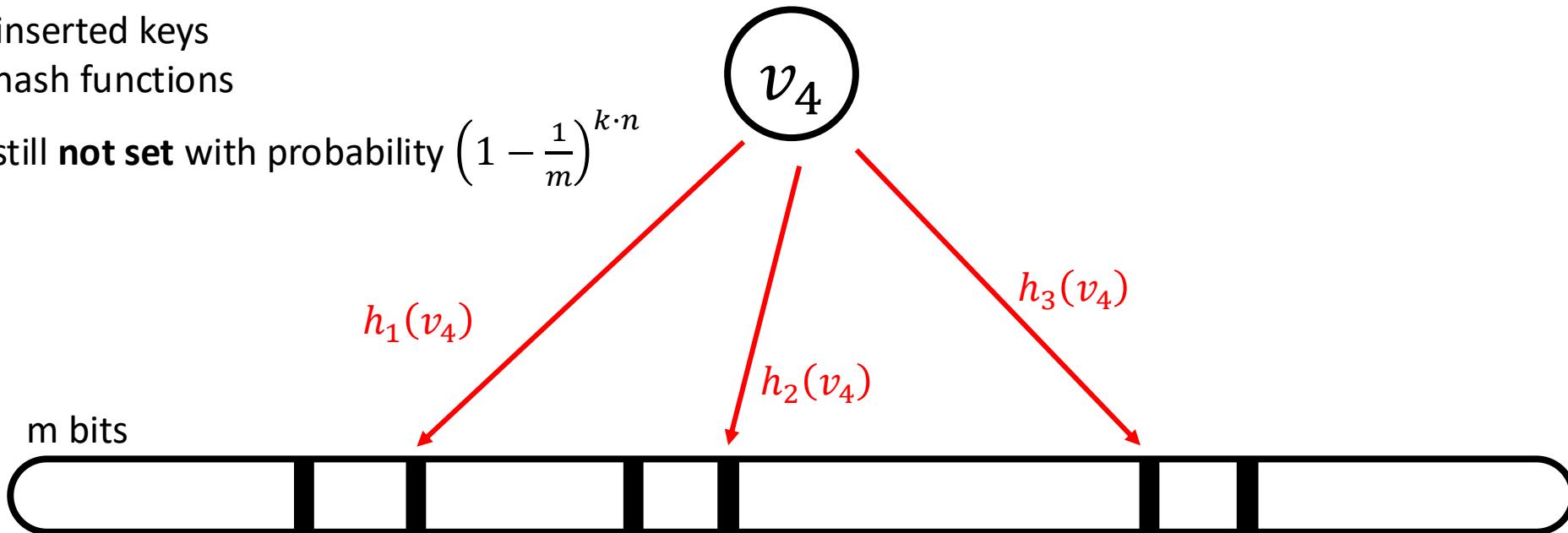


Bloom filter false positive

after n inserted keys

after k hash functions

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

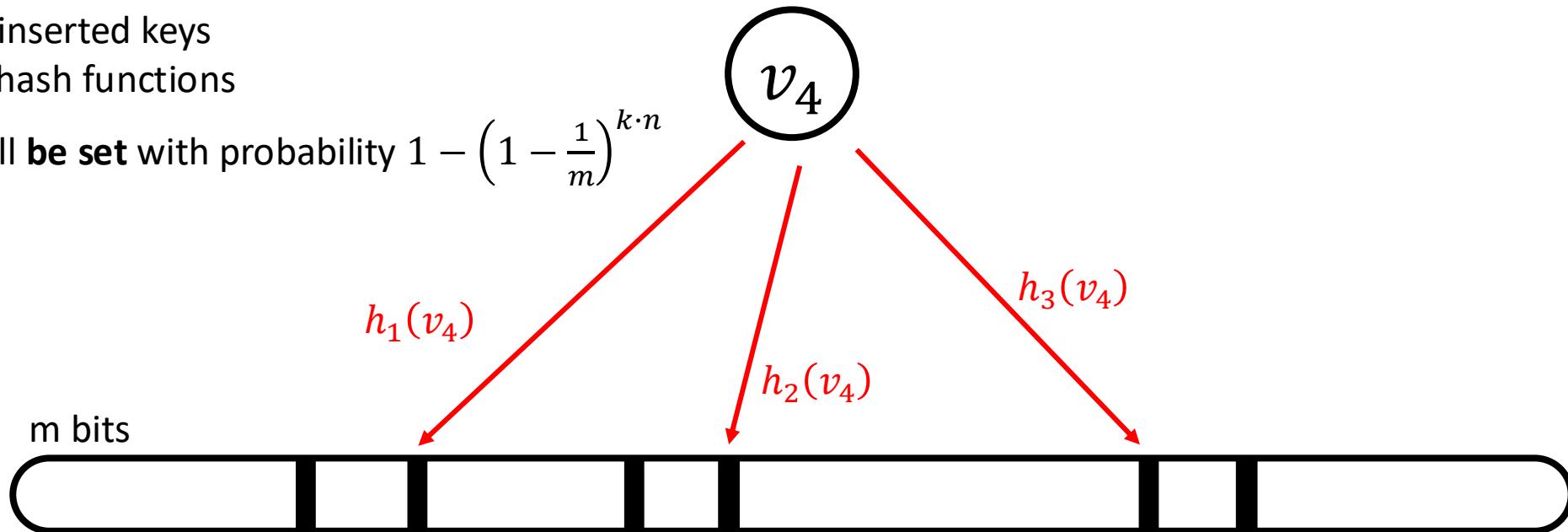


Bloom filter false positive

after n inserted keys

after k hash functions

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



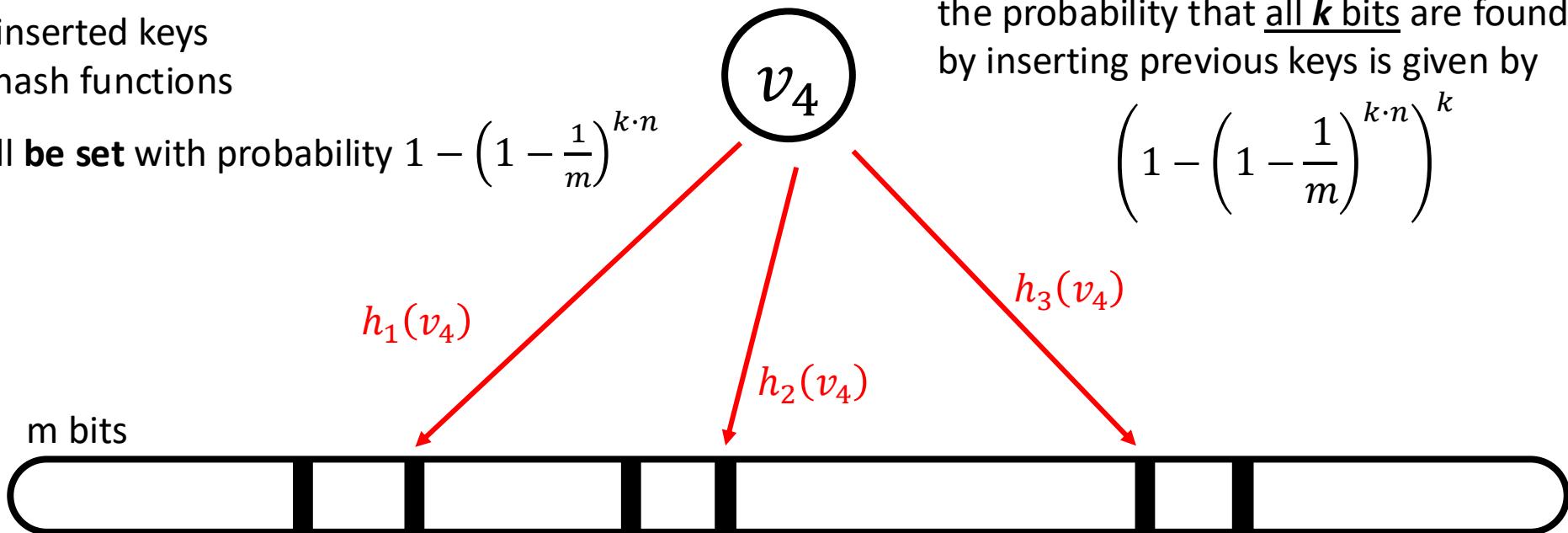
Bloom filter false positive

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 all k bits are found set by inserting previous keys is given by

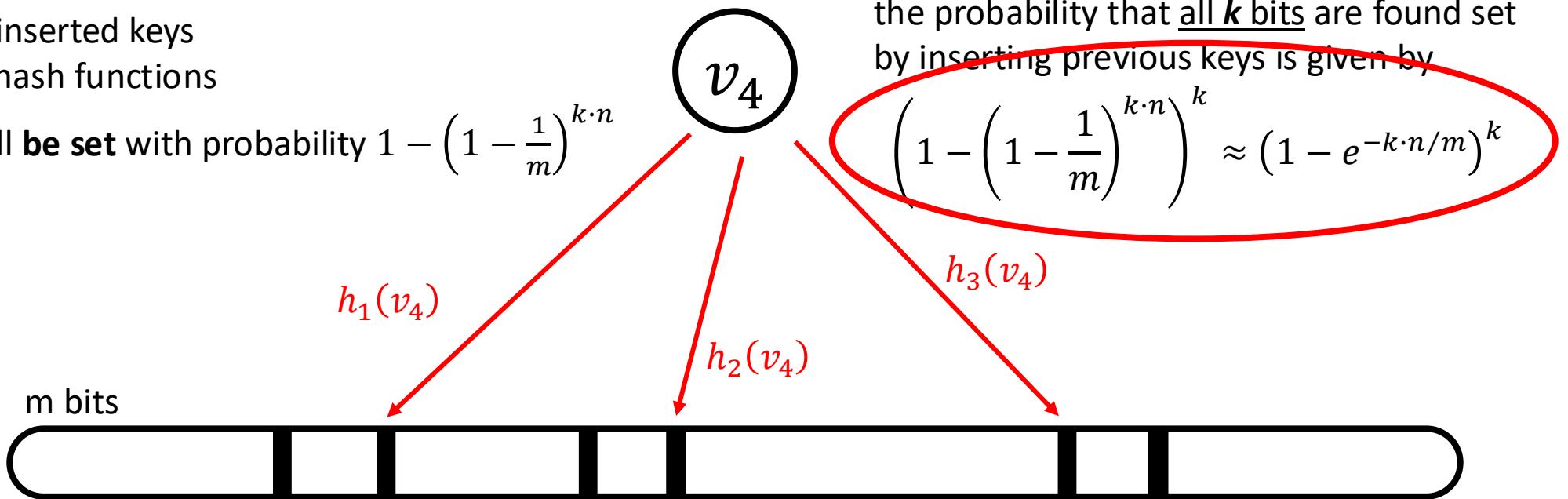
$$\left(1 - \left(1 - \frac{1}{m}\right)^{k \cdot n}\right)^k$$



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 all k bits are found set
by inserting previous keys is given by

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

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 :

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

hence, the probability that all k bits are found set by inserting previous keys is given by

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

Bloom filter false positive

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

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

we get:

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

details:

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

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

key-value stores vs. indexes

What is an index?

Auxiliary structure to quickly find rows based on arbitrary attribute

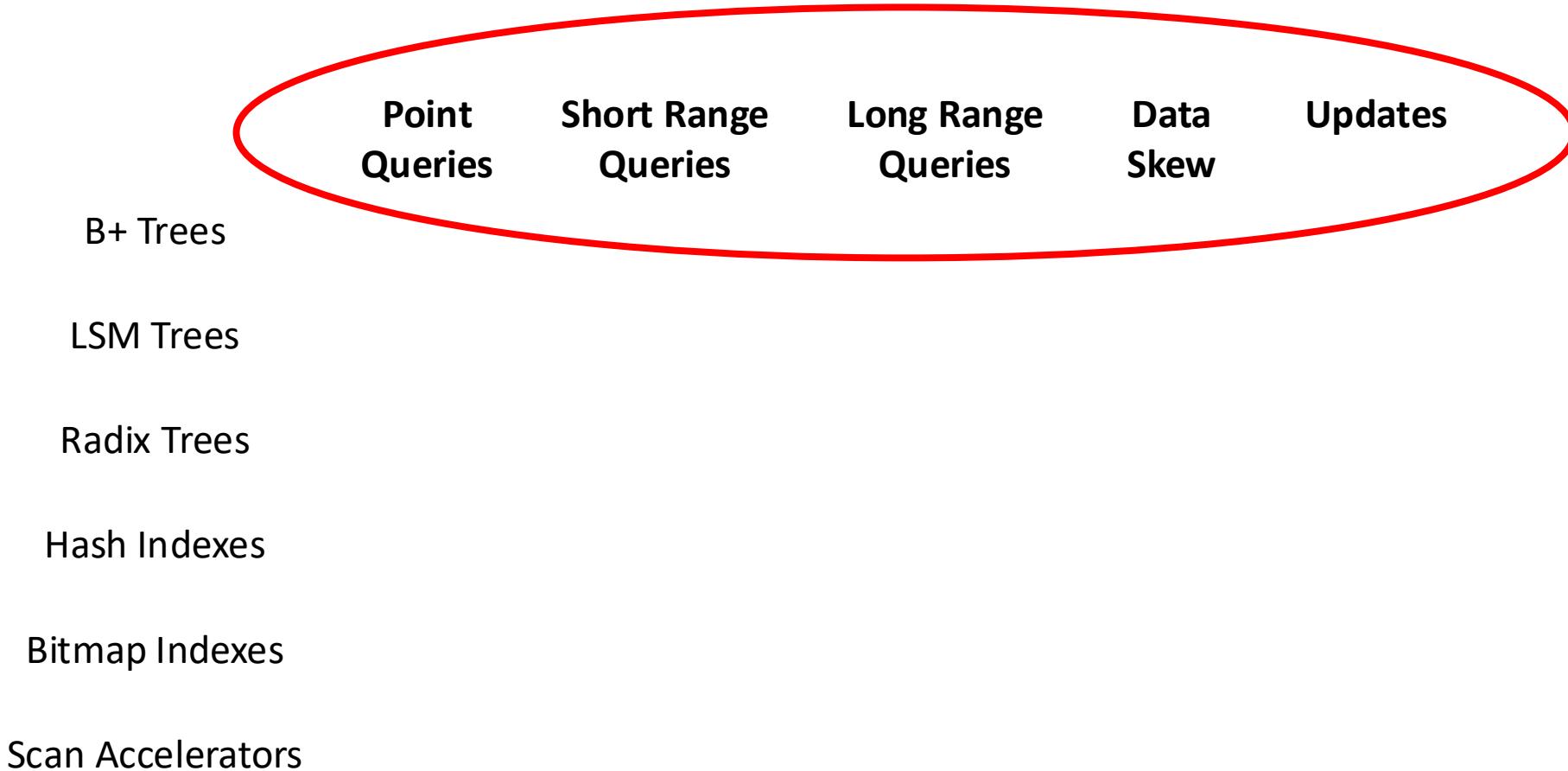
Special form of <key, value>



What are the possible *index designs*?

	Data Organization	Comments
B+ Trees	Sorted & partitioned	Partition <i>k-ways</i> recursively
LSM Trees	Log-structured & Sorted	Optimizes <i>insertion</i>
Radix Trees	Radix	Partition using the <i>key radix</i> representation
Hash Indexes	Hash	Partition by <i>hashing the key</i>
Bitmap Indexes	None	Succinctly represent <i>all rows with a key</i>
Scan Accelerators	None	Metadata to <i>skip accesses</i>

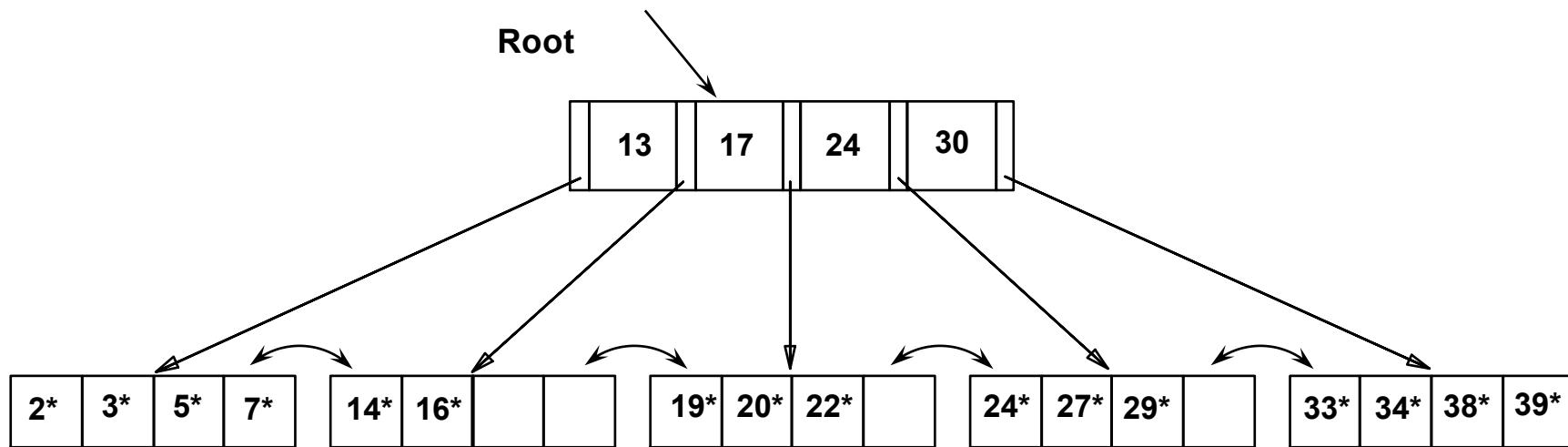
What are the possible *index designs*?



B+ Trees

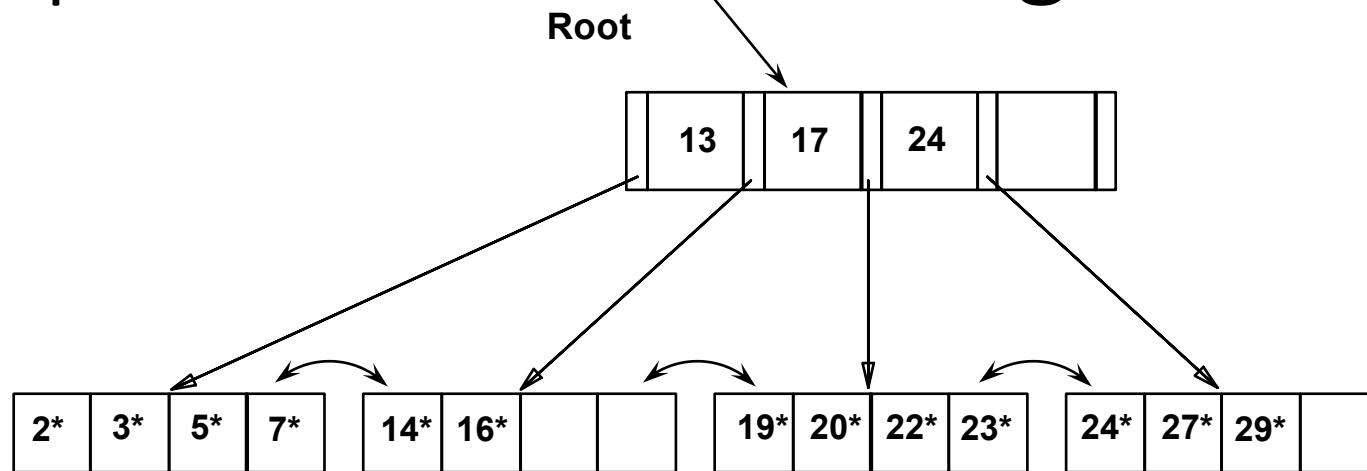
Search begins at root, and key comparisons direct it to a leaf.

Search for 5*, 15*, all data entries $\geq 24^*$...

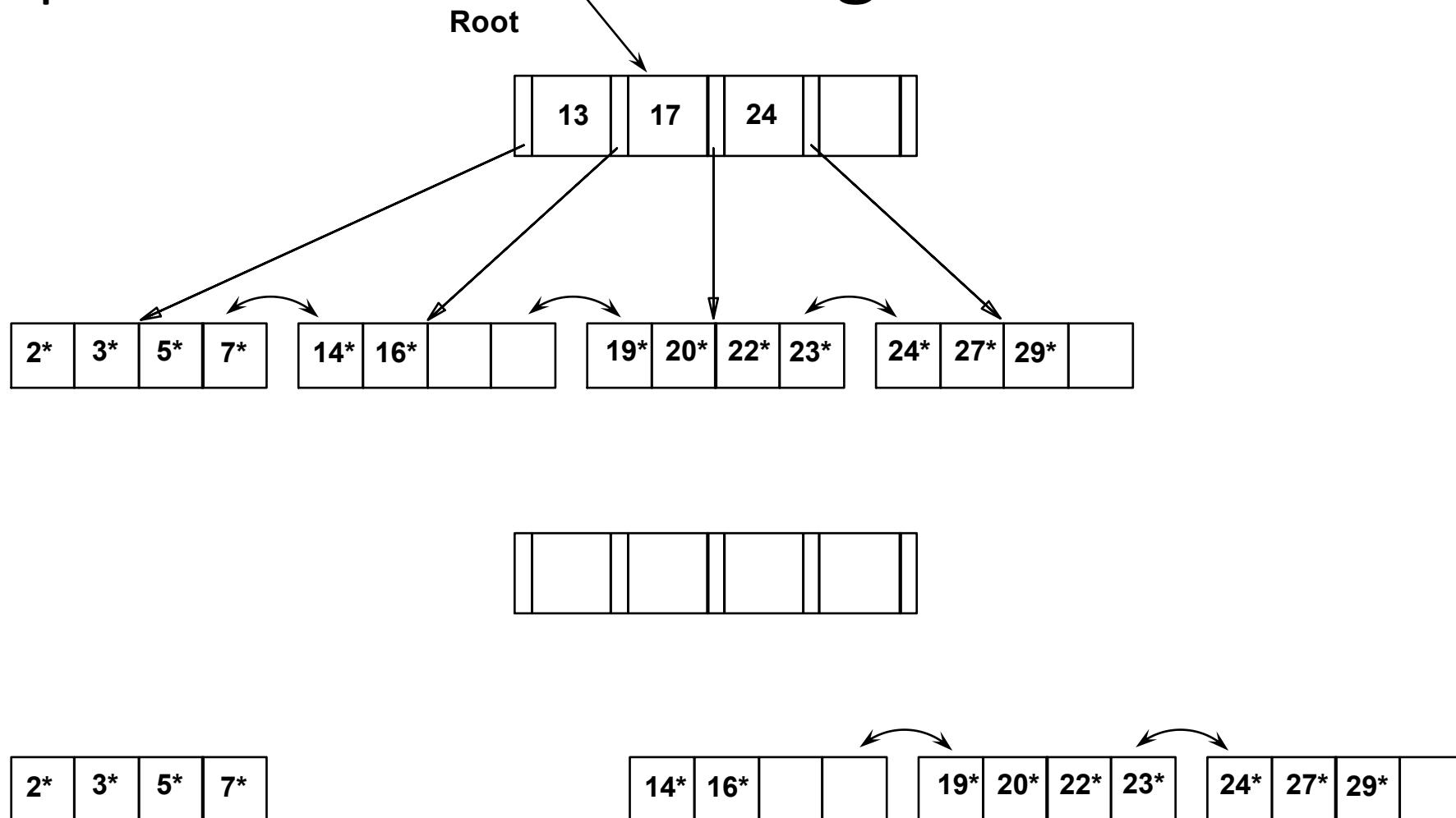


Based on the search for 15, we know it is not in the tree!*

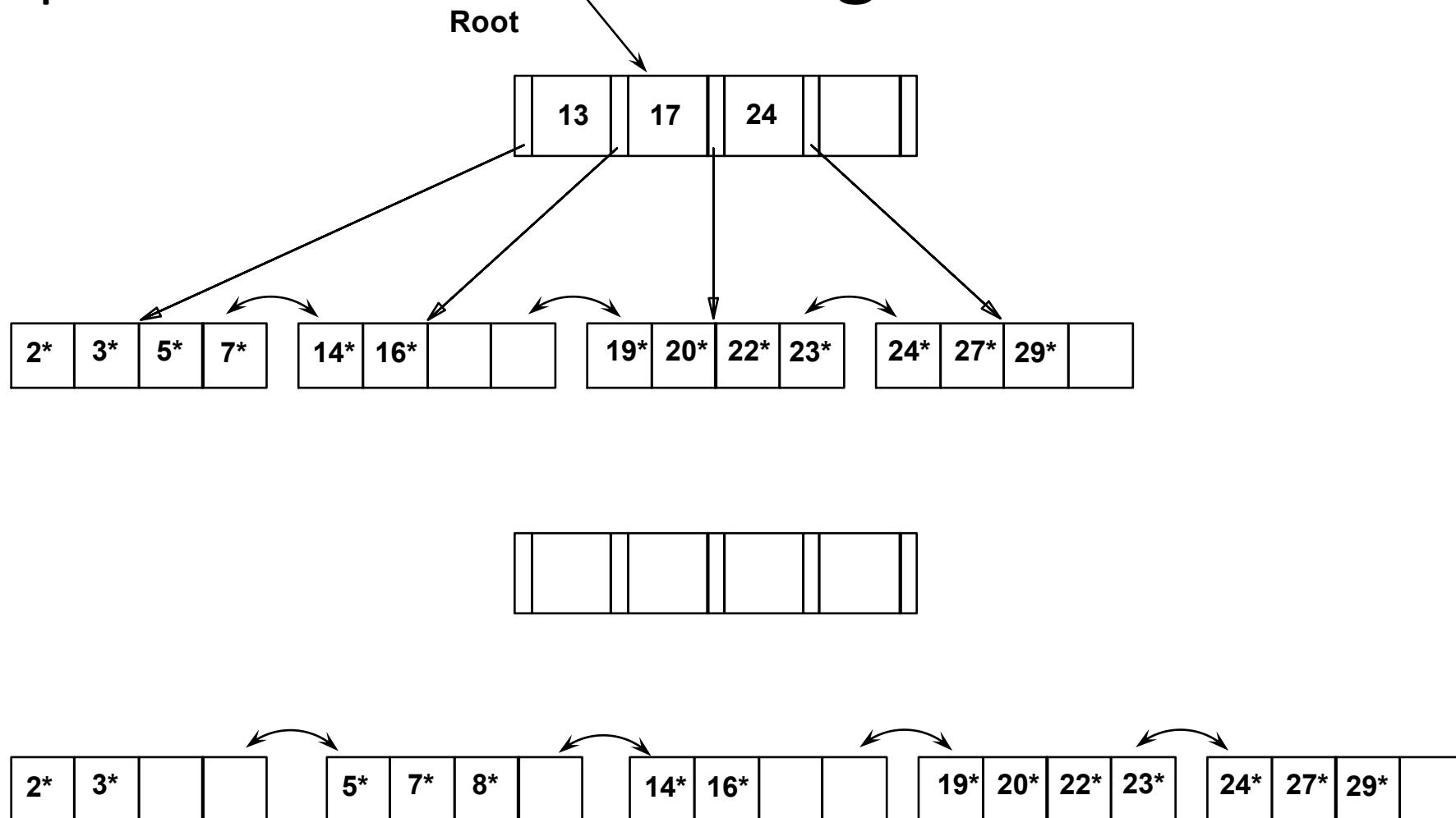
Example B+ Tree - Inserting 8*



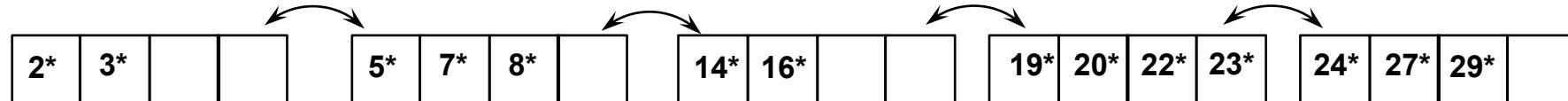
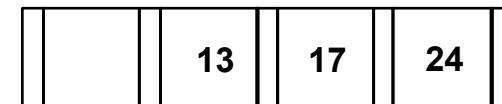
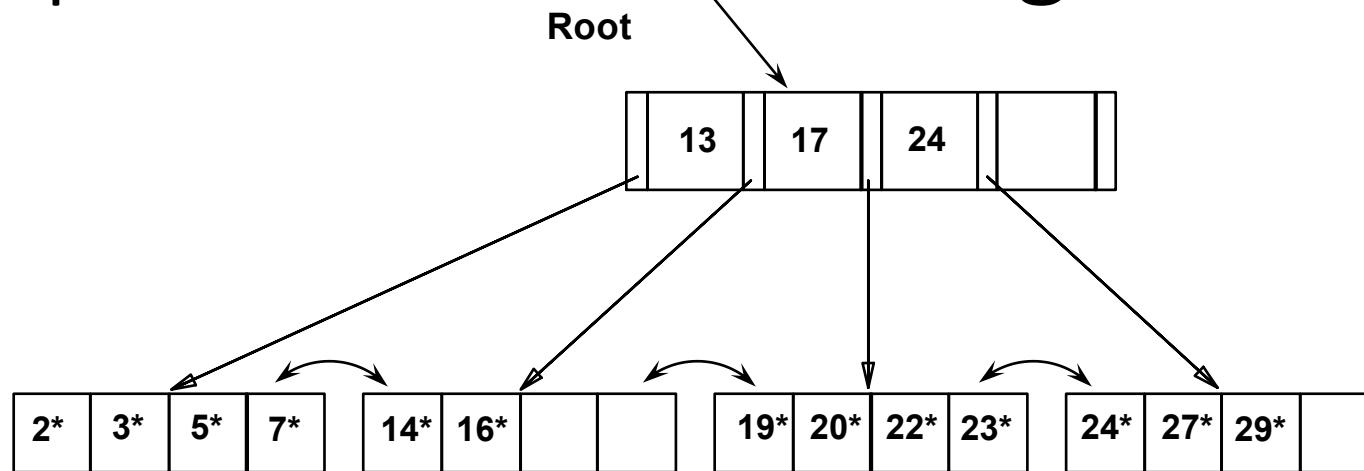
Example B+ Tree - Inserting 8*



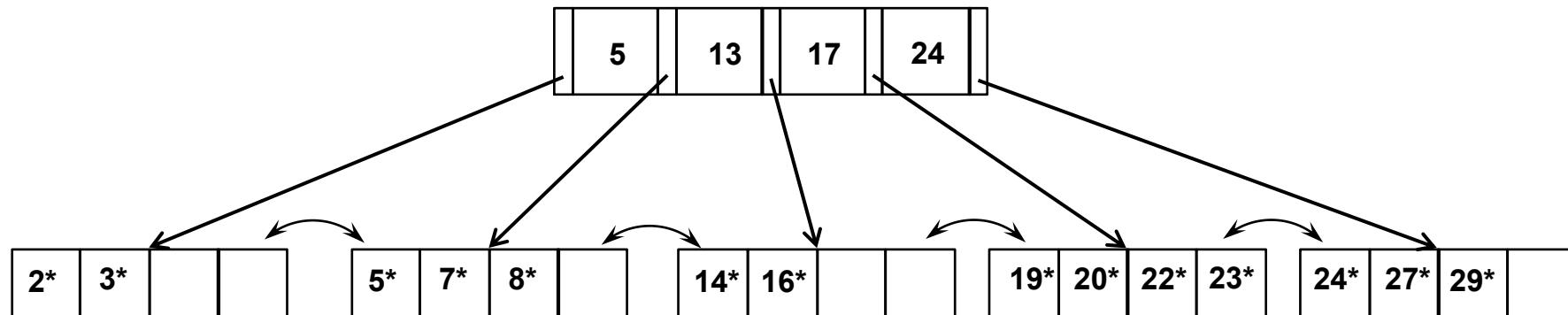
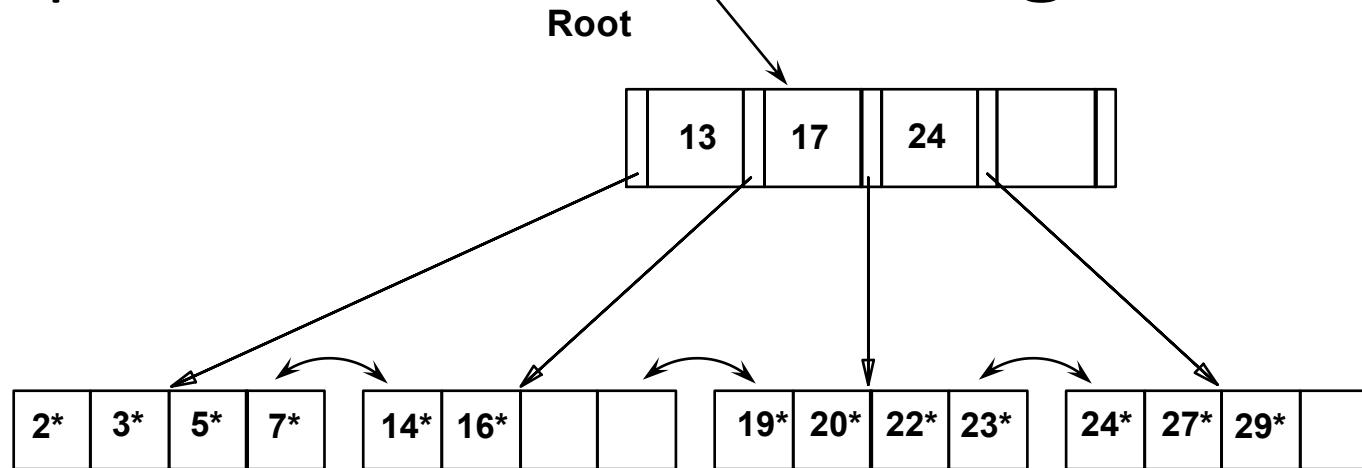
Example B+ Tree - Inserting 8*



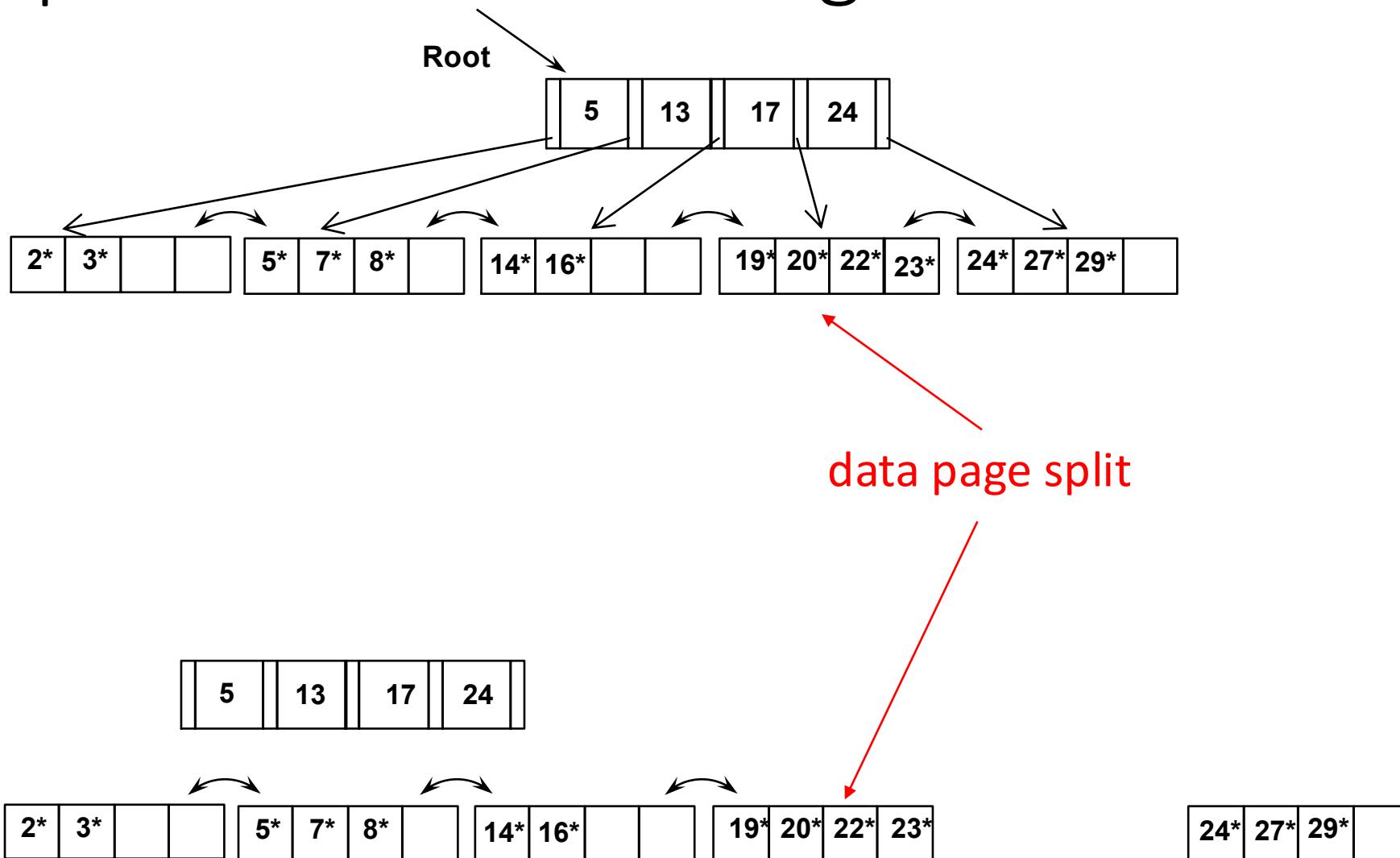
Example B+ Tree - Inserting 8*



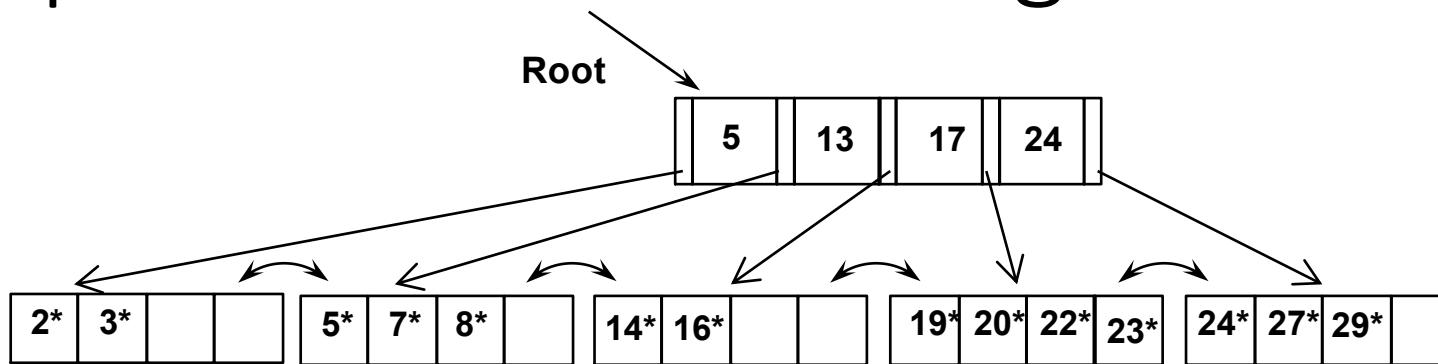
Example B+ Tree - Inserting 8*



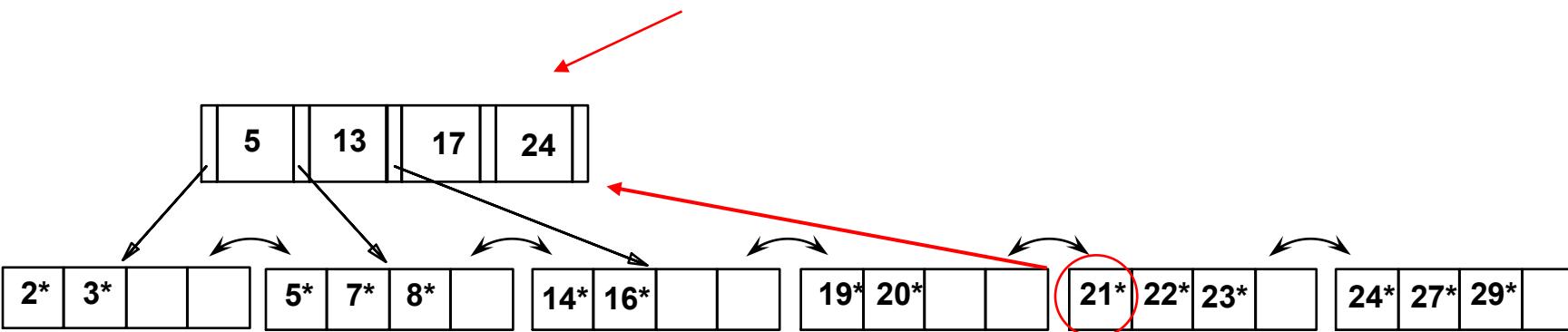
Example B+ Tree - Inserting 21*



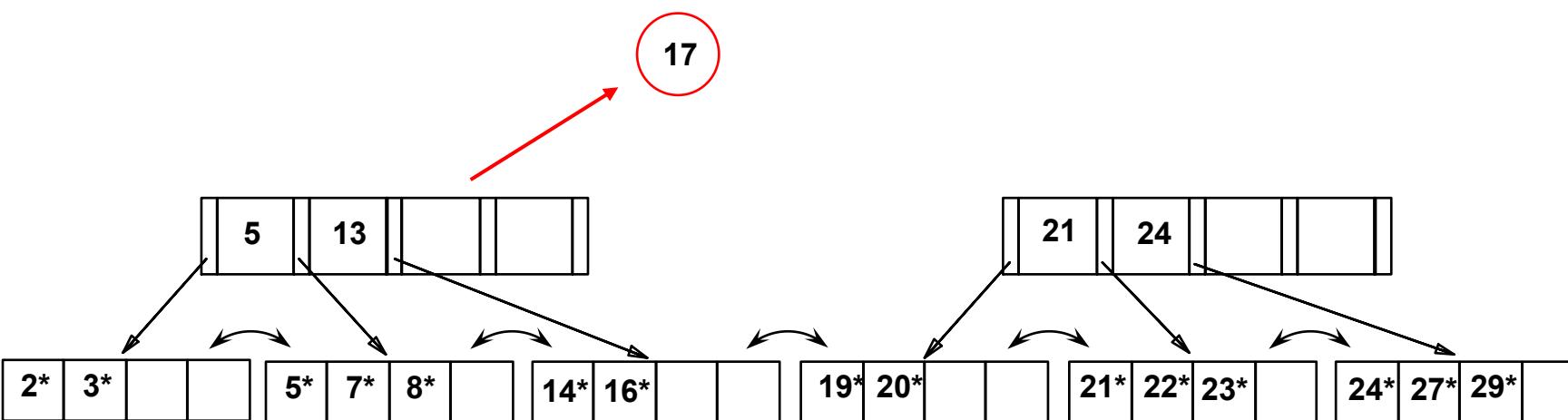
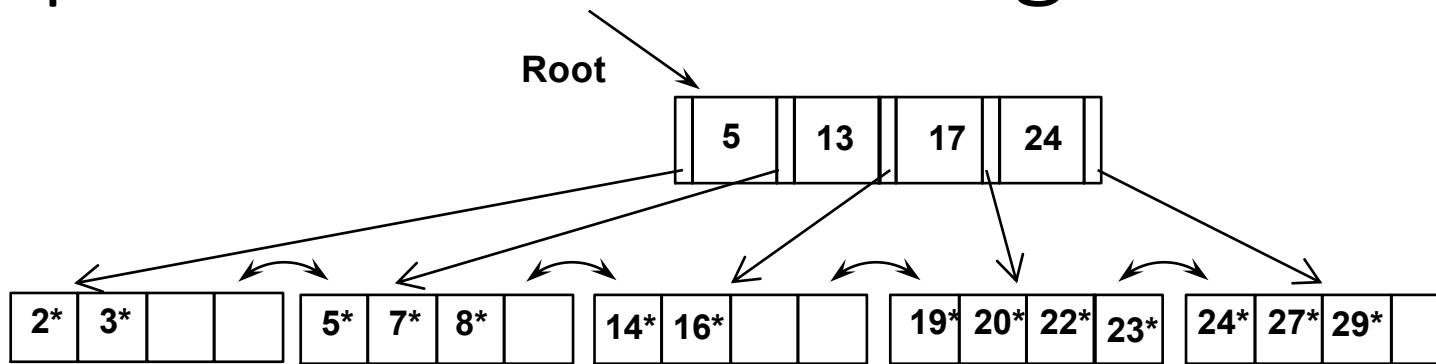
Example B+ Tree - Inserting 21*



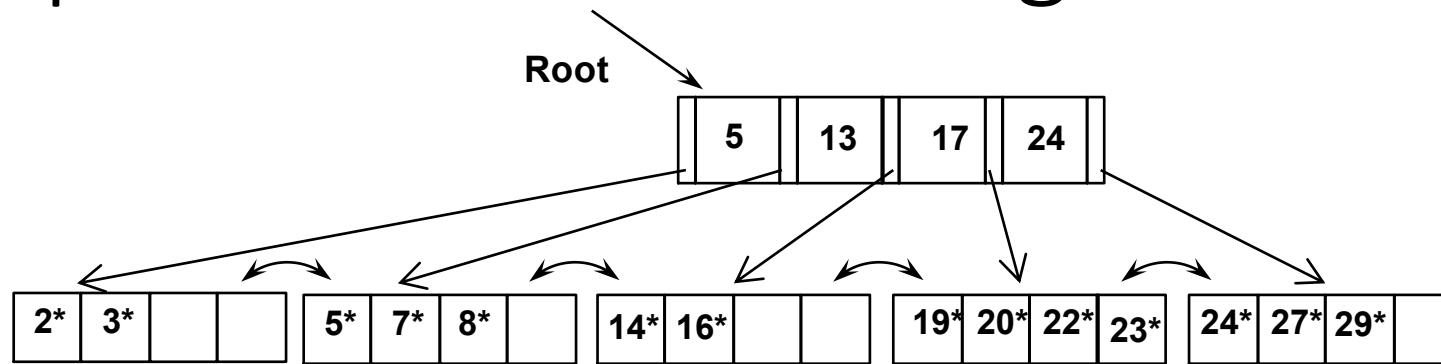
index page split



Example B+ Tree - Inserting 21*

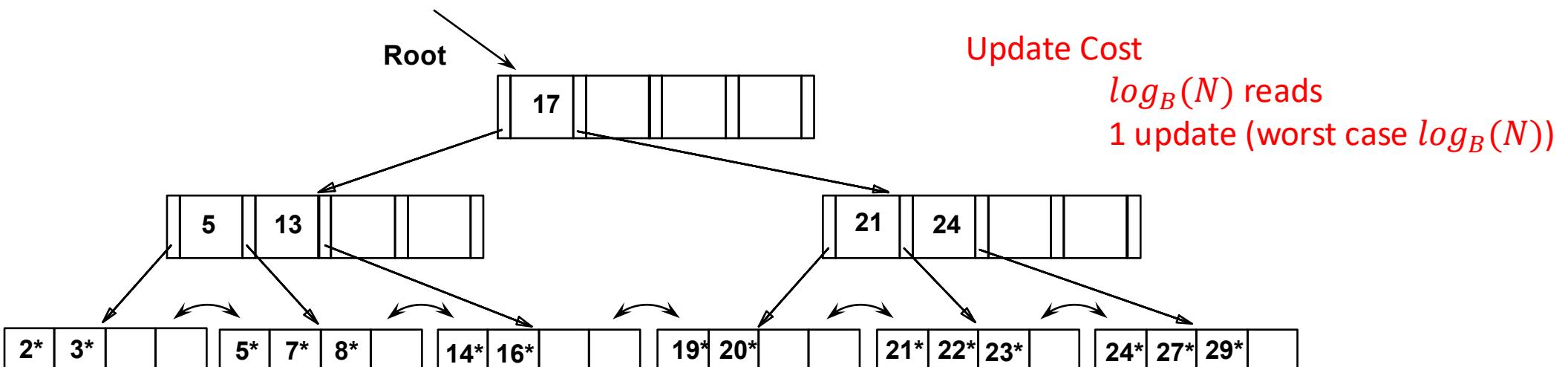


Example B+ Tree - Inserting 21*



what about growing dataset size?

Read Cost: $\log_B(N)$



Update Cost

$\log_B(N)$ reads
1 update (worst case $\log_B(N)$)

what about skewed data?

how many I/Os per insert?

What are the possible *index designs*?

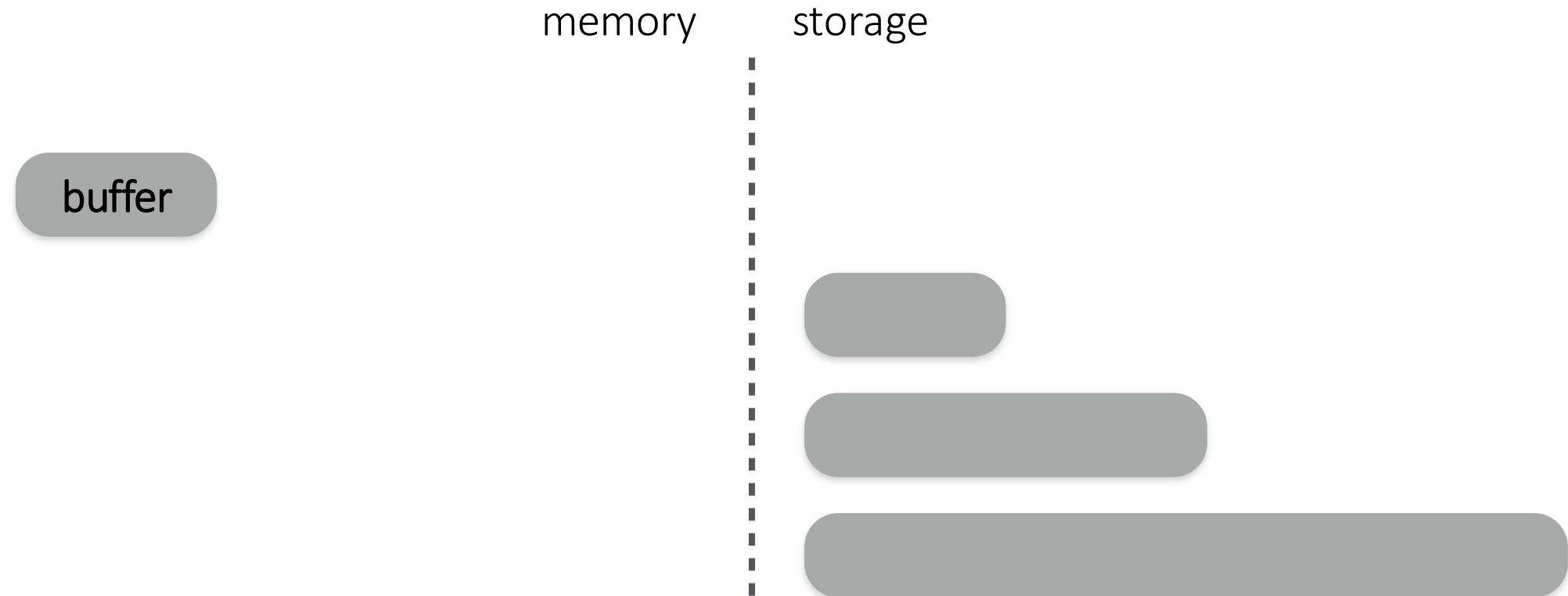
	Point Queries	Short Range Queries	Long Range Queries	Data Skew	Updates
B+ Trees	✓	✓	✓	✓	〰
LSM Trees					
Radix Trees					
Hash Indexes					
Bitmap Indexes					
Scan Accelerators					

LSM-trees

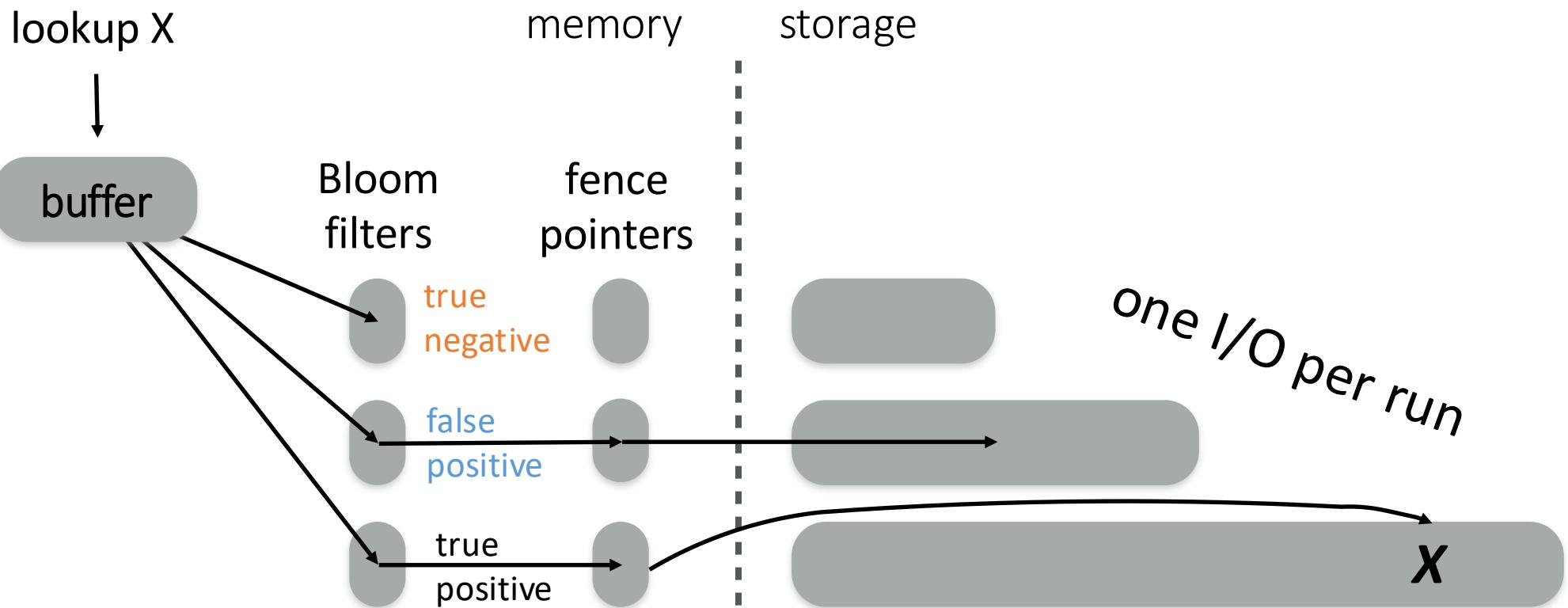
Is LSM-tree an index?

LSM-trees

Is LSM-tree an index?



LSM-trees



What are the possible *index designs*?

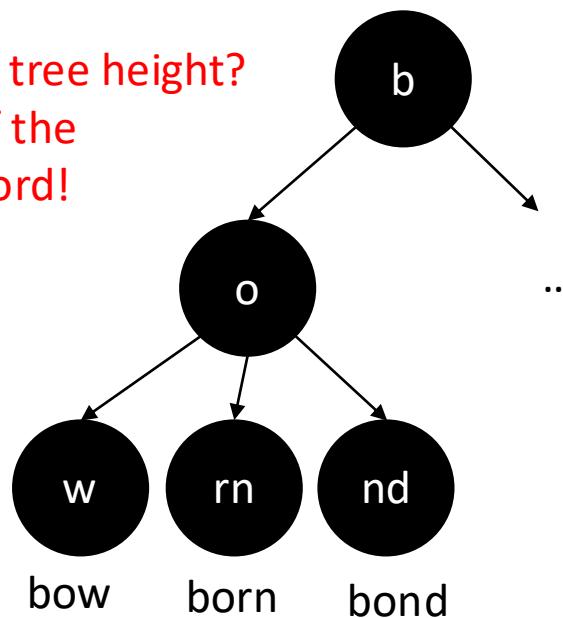
	Point Queries	Short Range Queries	Long Range Queries	Data Skew	Updates
B+ Trees	✓	✓	✓	✓	〰
LSM Trees	✓	✗	〰	✓	✓
Radix Trees					
Hash Indexes					
Bitmap Indexes					
Scan Accelerators					

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

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

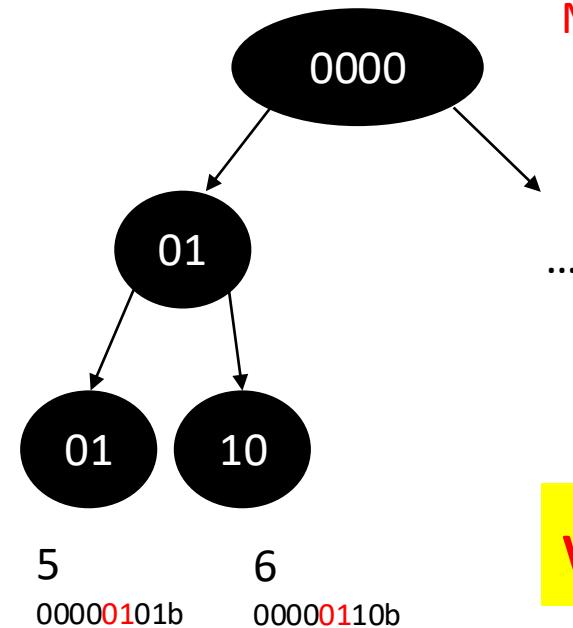
what about integer keys?

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?

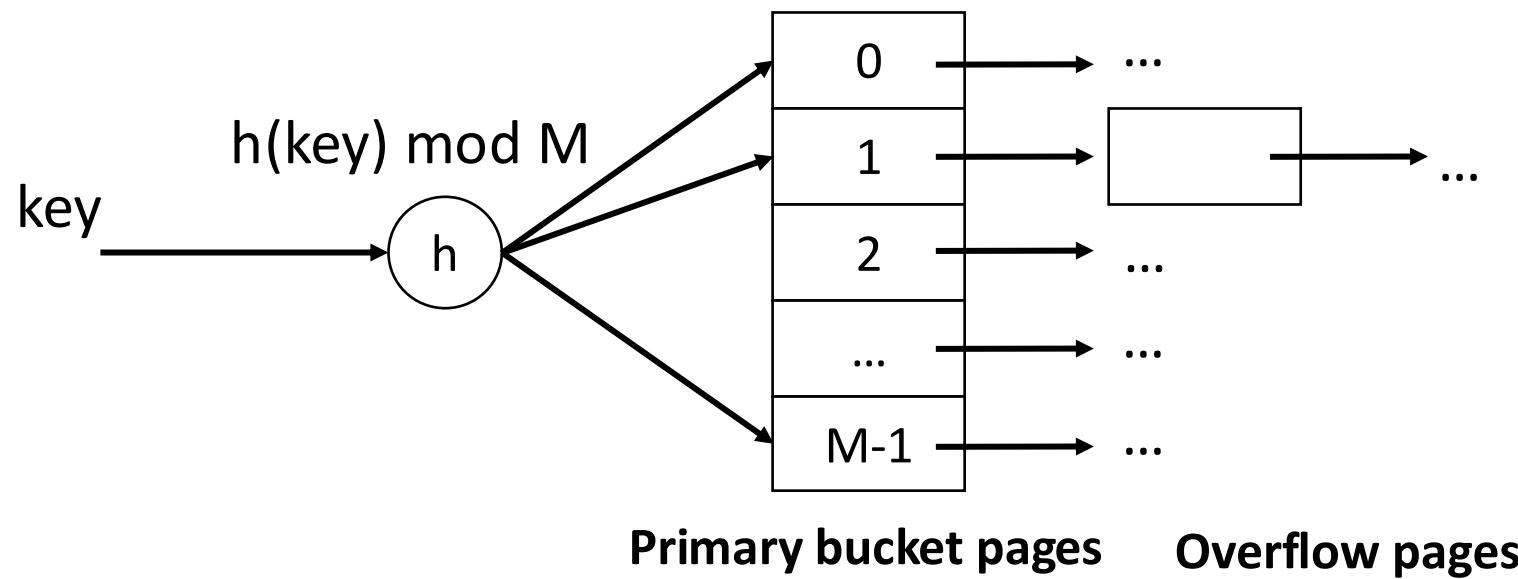
What are the possible *index designs*?

	Point Queries	Short Range Queries	Long Range Queries	Data Skew	Updates
B+ Trees	✓	✓	✓	✓	〰
LSM Trees	✓	✗	〰	✓	✓
Radix Trees	✓	✓	✓	✗	〰
Hash Indexes					
Bitmap Indexes					
Scan Accelerators					

Hash Indexes (static hashing)

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

$h(k) \bmod M$ = bucket to insert data entry with key k (M : #buckets)

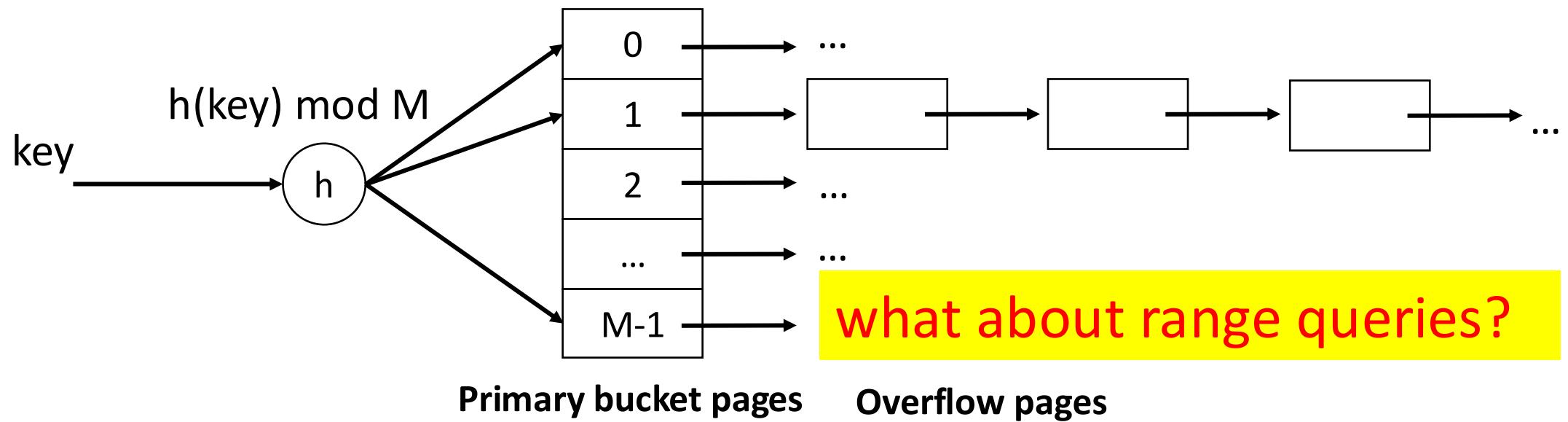


what if I have skew in the data set (or a bad hash function)?

Hash Indexes (static hashing)

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

$h(k) \bmod M$ = bucket to insert data entry with key k (M : #buckets)



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

What are the possible *index designs*?

	Point Queries	Short Range Queries	Long Range Queries	Data Skew	Updates
B+ Trees	✓	✓	✓	✓	〰
LSM Trees	✓	✗	〰	✓	✓
Radix Trees	✓	✓	✓	✗	〰
Hash Indexes	✓	〰	✗	✗	✓
Bitmap Indexes					
Scan Accelerators					

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

30
20
30
10
20
10
30
20

A=10

0
0
0
1
0

A=20

0
1
0
0

A=30

1
0
1
0
0
0
1
0

Index Size



Space-inefficient for domains with large cardinality



Addressed by bitvector encoding/compression

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

encoded bitvectors

what about updates?

raw bitvector

Update?

encode

13 zeros

ending pattern

encoded bitvector

decode

flip bit

re-encode

10 zeros

ending pattern

What are the possible *index designs*?

	Point Queries	Short Range Queries	Long Range Queries	Data Skew	Updates
B+ Trees	✓	✓	✓	✓	〰
LSM Trees	✓	✗	〰	✓	✓
Radix Trees	✓	✓	✓	✗	〰
Hash Indexes	✓	〰	✗	✗	✓
Bitmap Indexes	✓	〰	〰	〰	✗
Scan Accelerators					

Scan Accelerators

Zonemaps

Search for 25

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]

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*?

	Point Queries	Short Range Queries	Long Range Queries	Data Skew	Updates	Affected by Physical Order
B+ Trees	✓	✓	✓	✓	〰	—
LSM Trees	✓	✗	〰	✓	✓	—
Radix Trees	✓	✓	✓	✗	〰	—
Hash Indexes	✓	〰	✗	✗	✓	—
Bitmap Indexes	✓	〰	〰	〰	✗	<i>no</i>
Scan Accelerators	✗	〰	✓	✓	〰	<i>yes</i>

Adaptive Data Organization: Database Cracking

idea: there is an *ideal* data organization

what is it (for a collection of keys – e.g., a column)?
sorted!

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

Adaptive Data Organization: Database Cracking

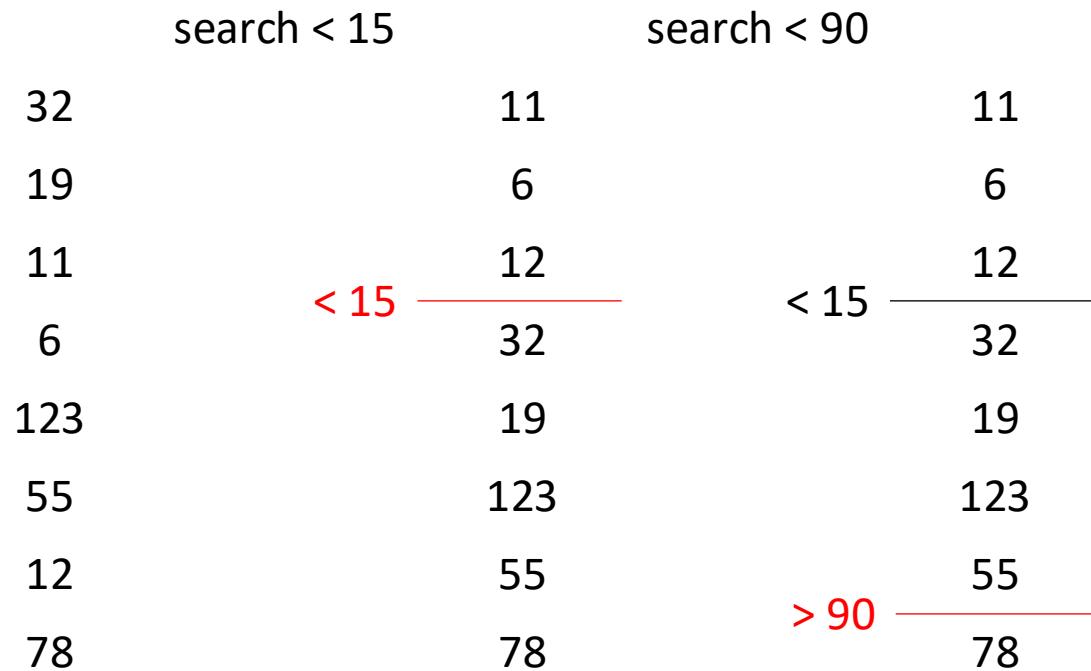
search < 15

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

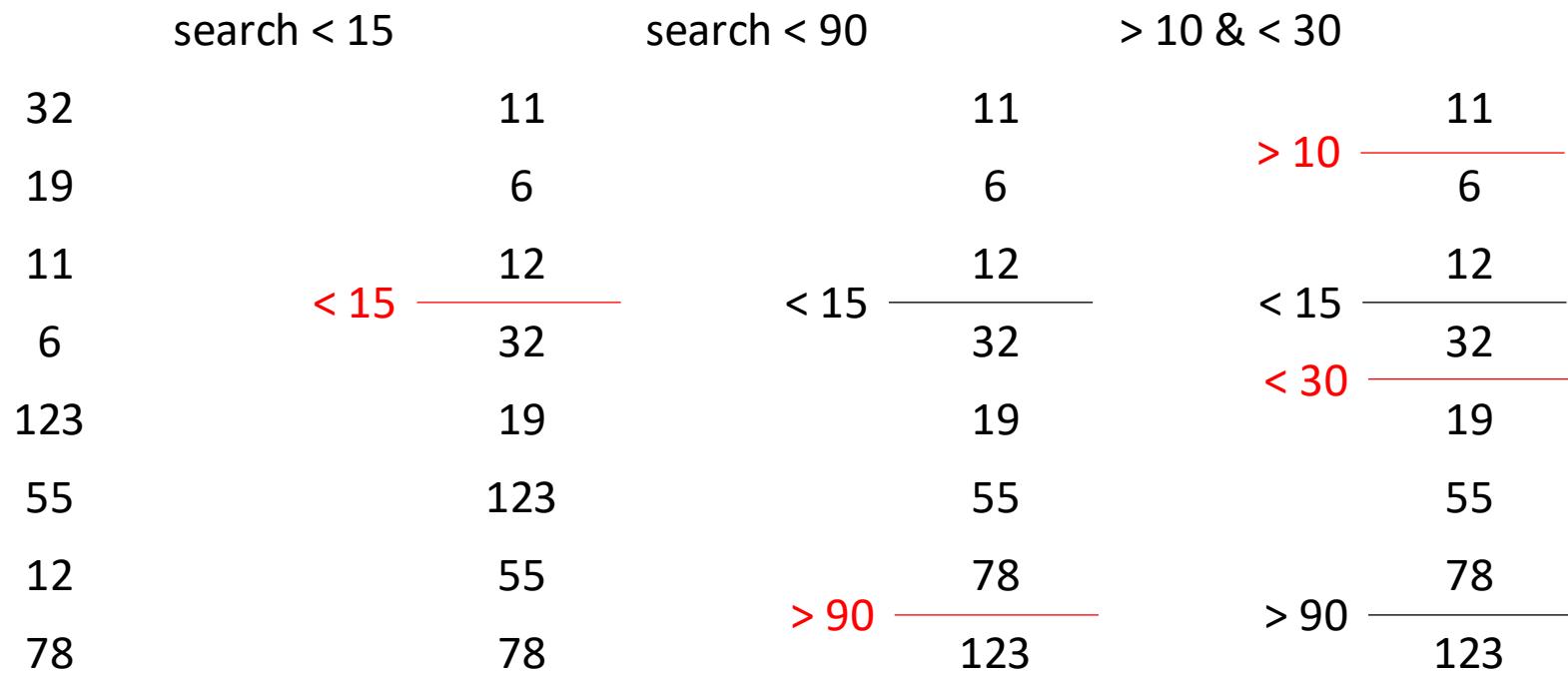
< 15

—

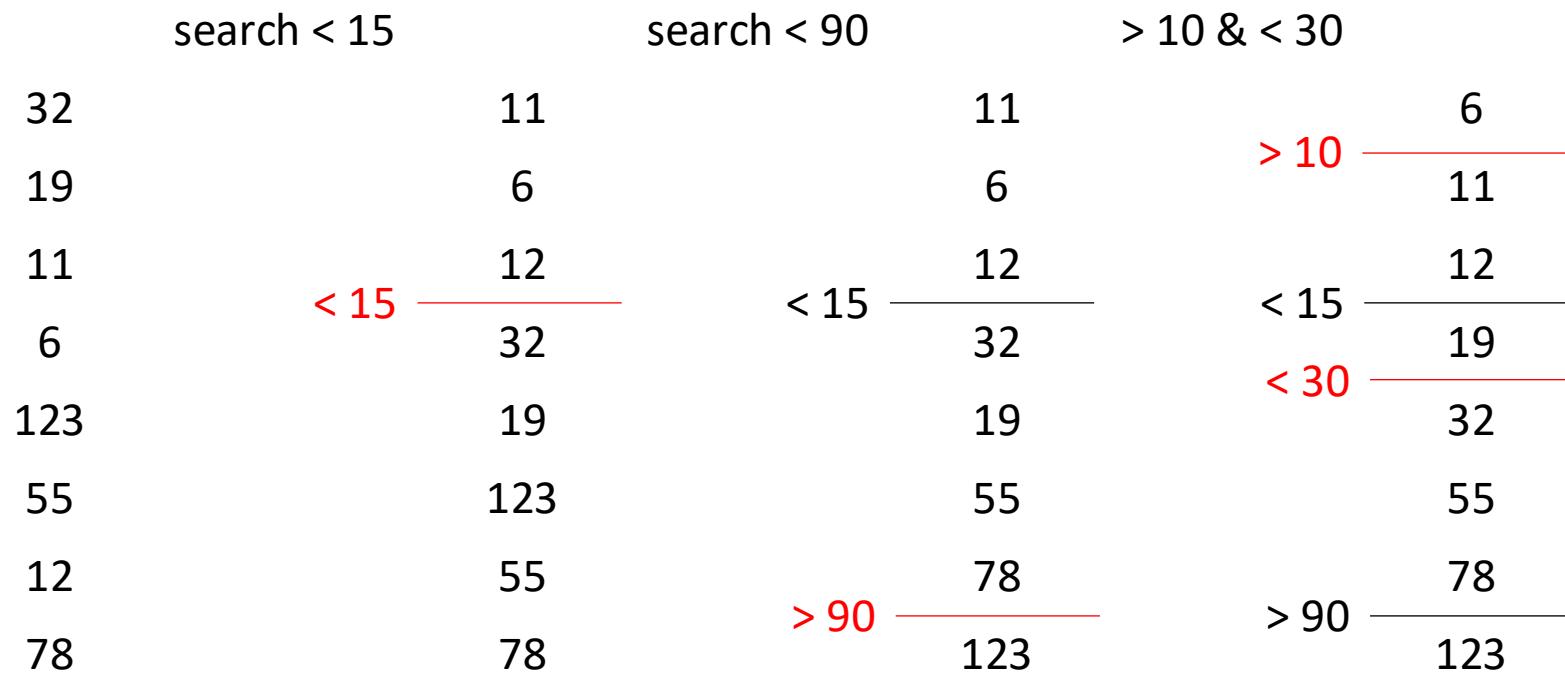
Adaptive Data Organization: Database Cracking



Adaptive Data Organization: Database Cracking



Adaptive Data Organization: Database Cracking



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

CS 561: Data Systems Architectures

Various forms of Indexing:
Trees, Tries, Hashing, Bitmap Indexes, Database Cracking

Prof. Manos Athanassoulis

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