BOSTON UNIVERSITY

CS 561: **Data Systems Architectures**

Introduction to Indexing:

Trees, Tries, Hashing, Bitmaps: The Whole Design Space

Dr. Subhadeep Sarkar

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

A few reminders

A) Choose your **class project**. (40% of your grades)

B) Project proposals **due on 03/12** in groups of 2-3.

C) Proposal format:
   (i) Project background
   (ii) Problem statement
   (iii) Project objectives
   (iv) Project timeline

BOSTON
UNIVERSITY

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/…

BOSTON UNIVERSITY

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

	Data Organization	Comments
B+ Trees	Sorted & partitioned	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***

What are the possible ***<u>index designs</u>***?

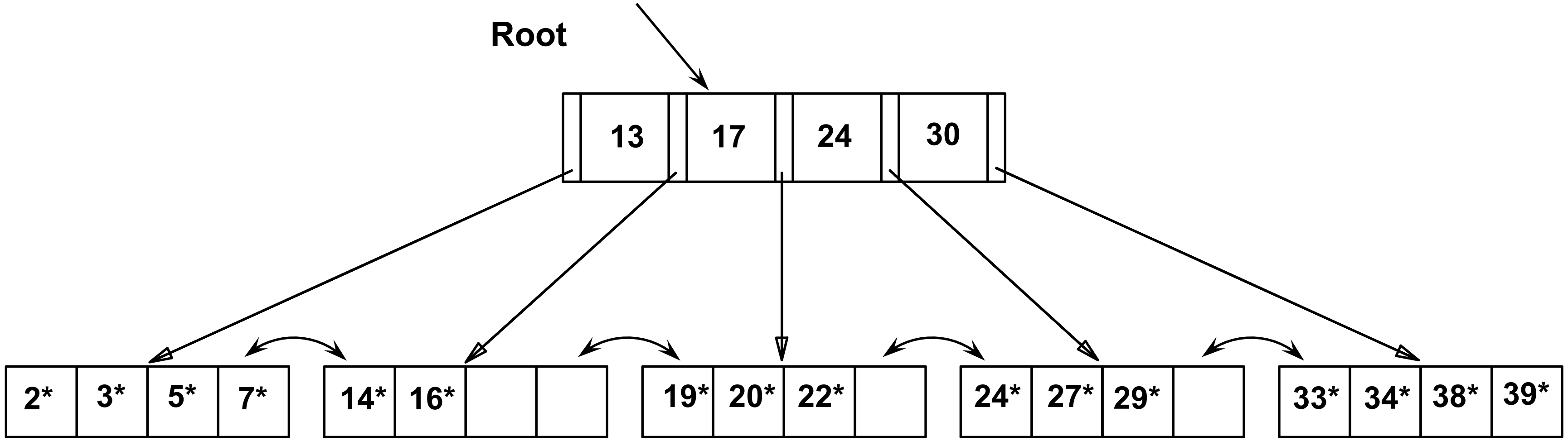
	Point Queries	**Short Range Queries**	**Long Range Queries**	**Data Skew**	**Updates**
B+ Trees					
LSM Trees					
Radix Trees					
Hash Indexes					
Bitmap Indexes					
Scan Accelerators					

BOSTON UNIVERSITY

B+ Trees

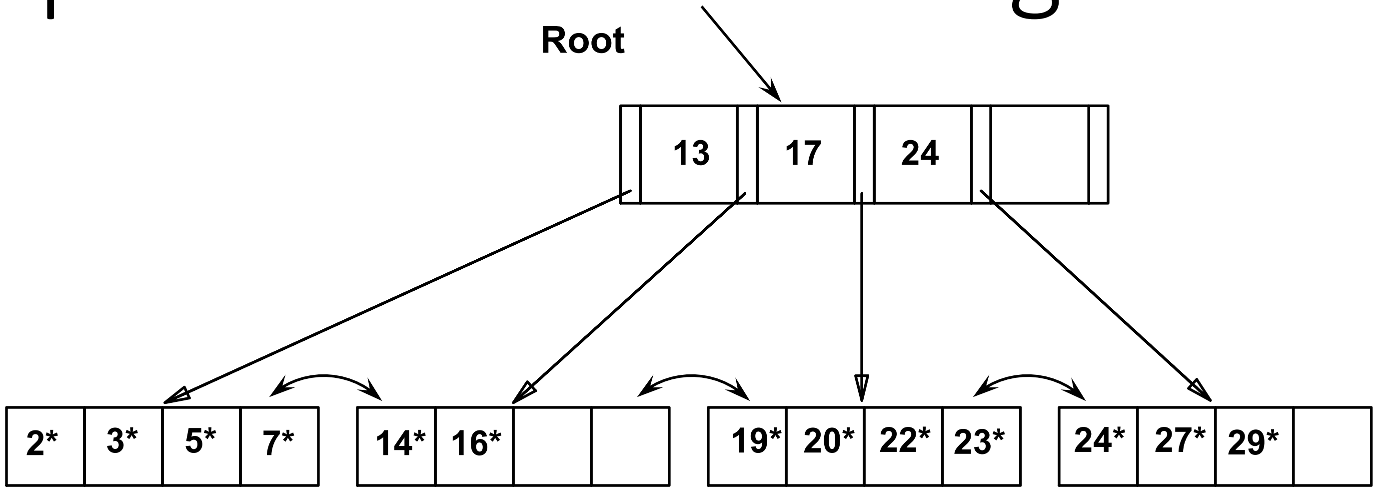
Search begins at root, and key comparisons direct it to a leaf.

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



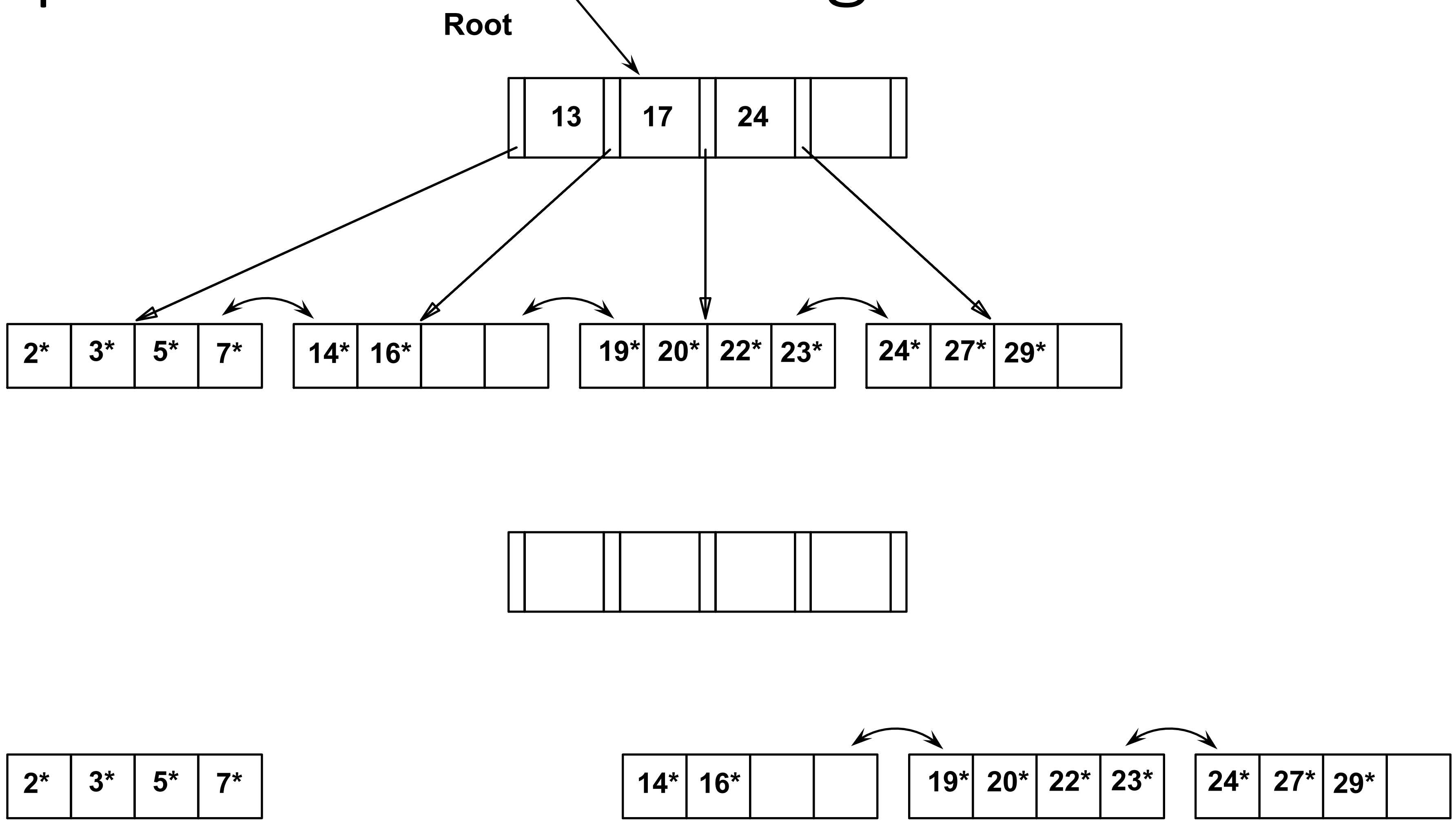
BOSTON UNIVERSITY

Example B+ Tree - Inserting 8*



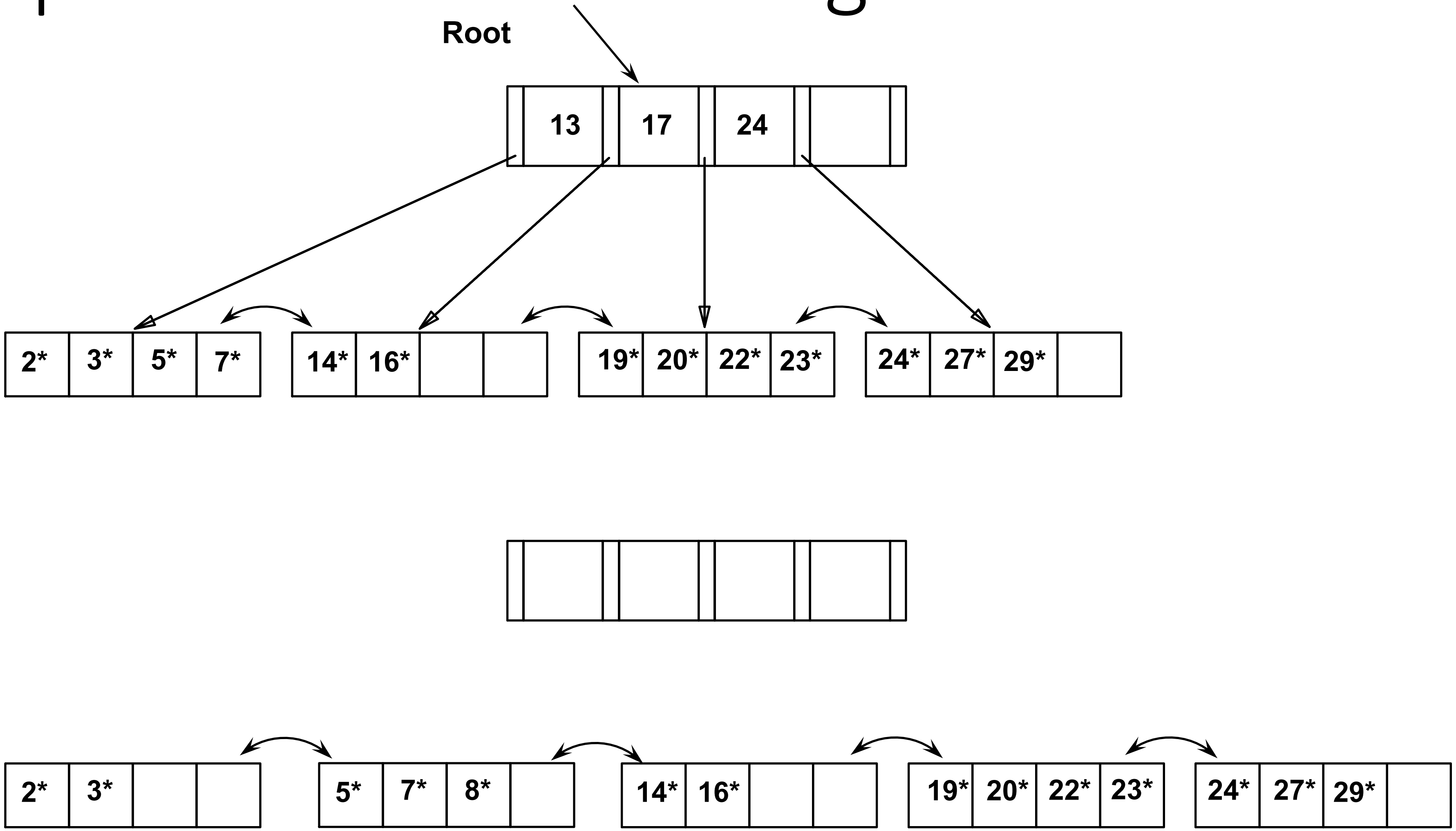
BOSTON UNIVERSITY

Example B+ Tree - Inserting 8*

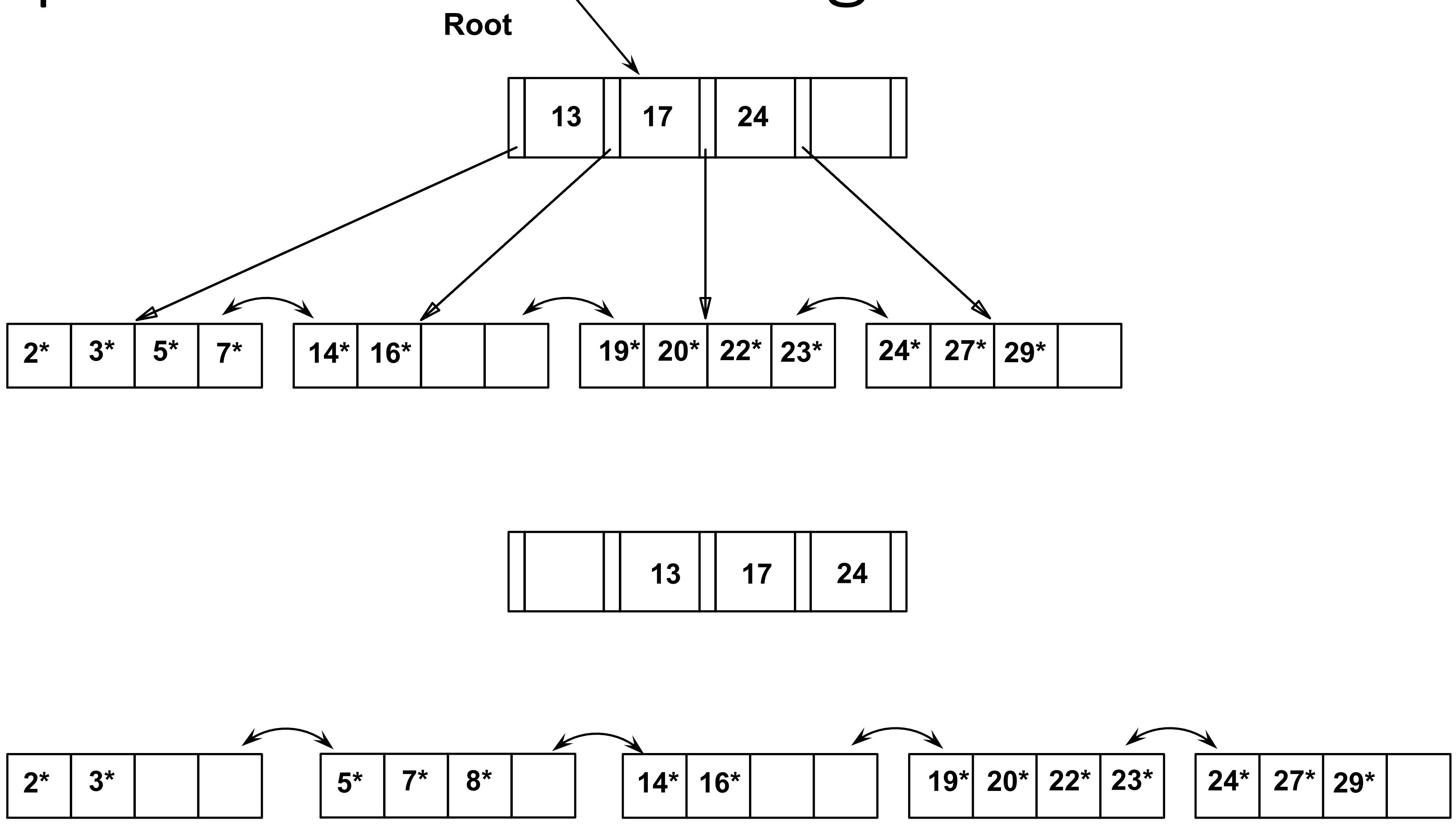


BOSTON UNIVERSITY

Example B+ Tree - Inserting 8*

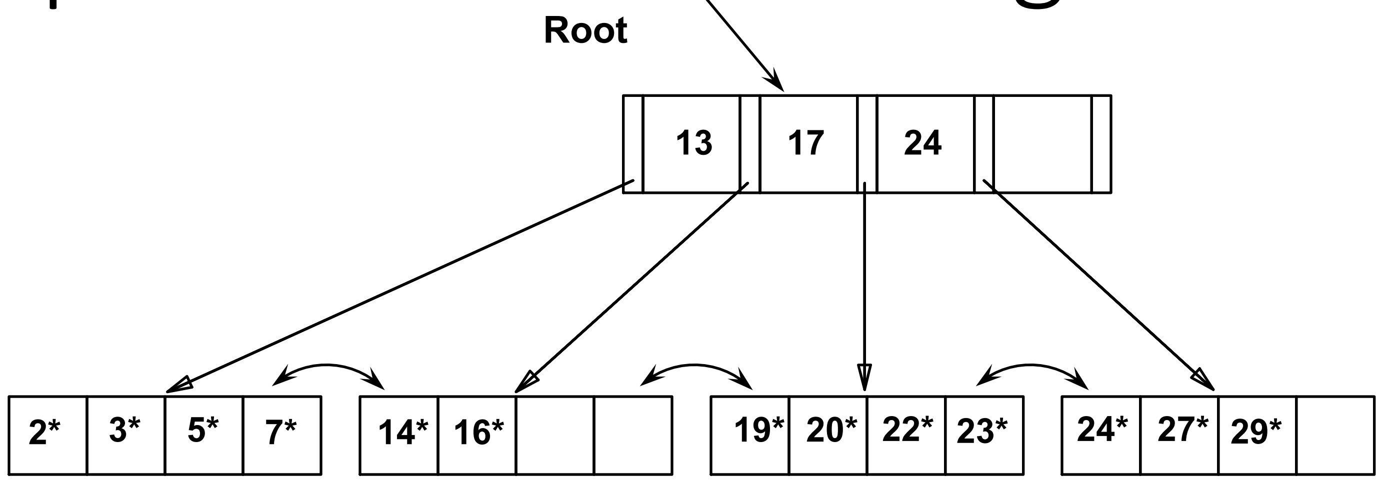


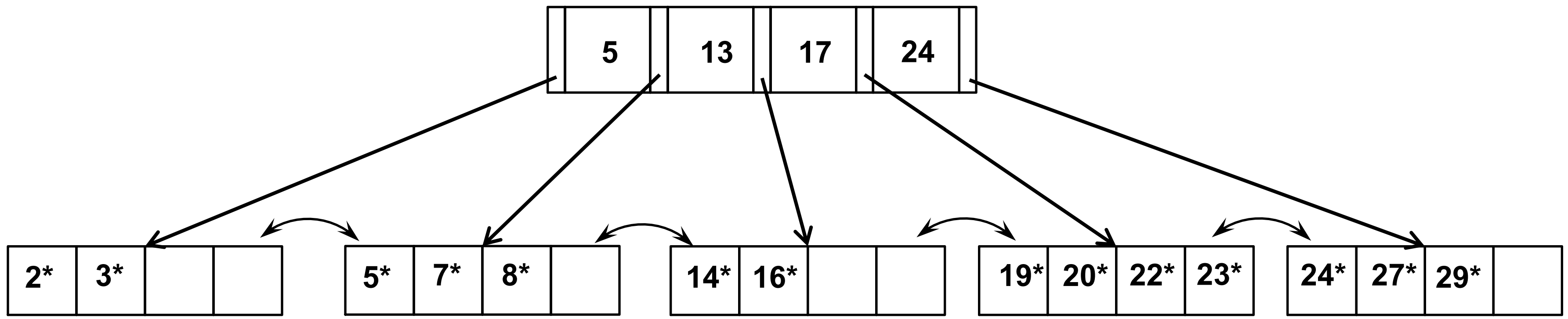
Example B+ Tree - Inserting 8*



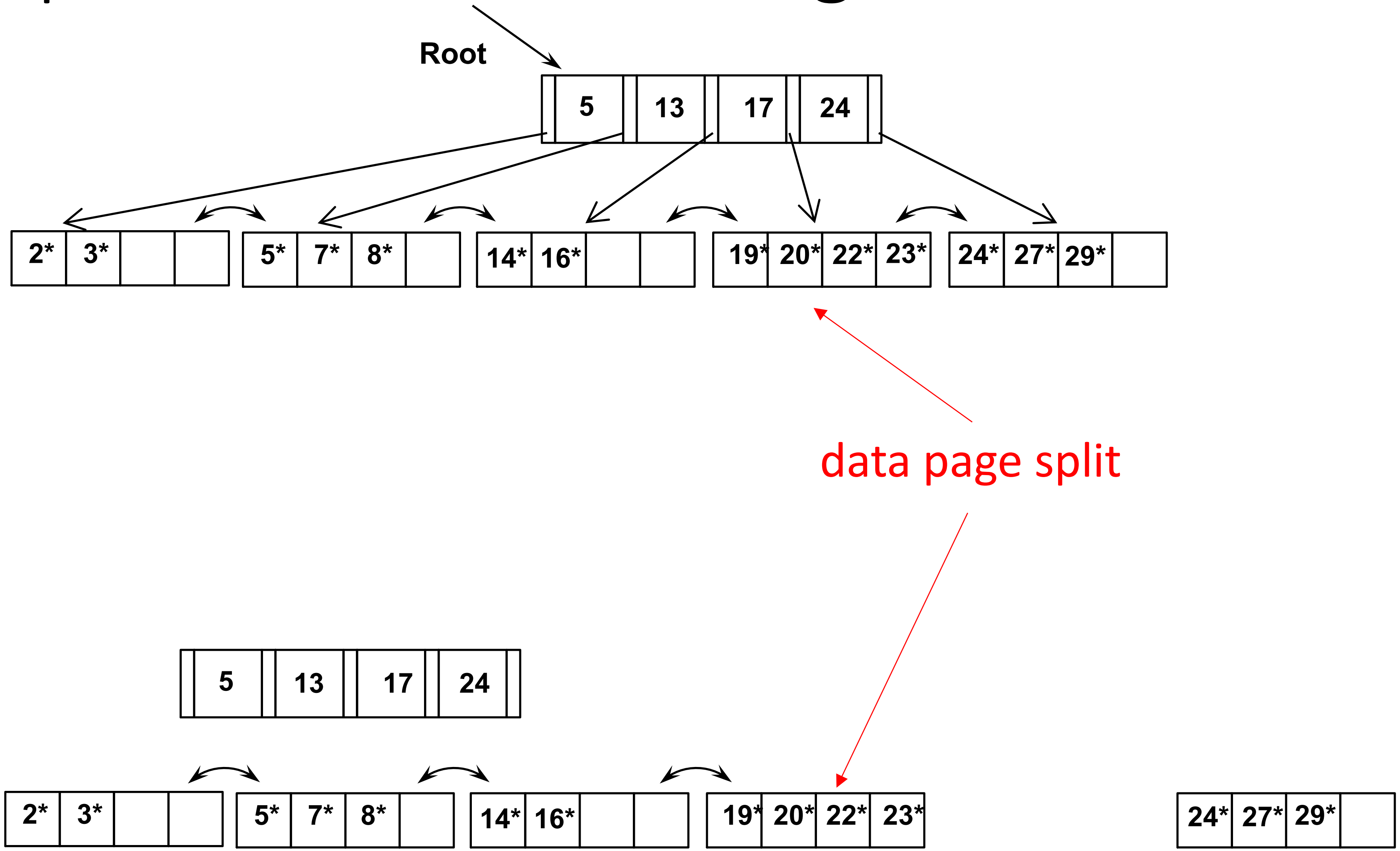
BOSTON UNIVERSITY

Example B+ Tree - Inserting 8*



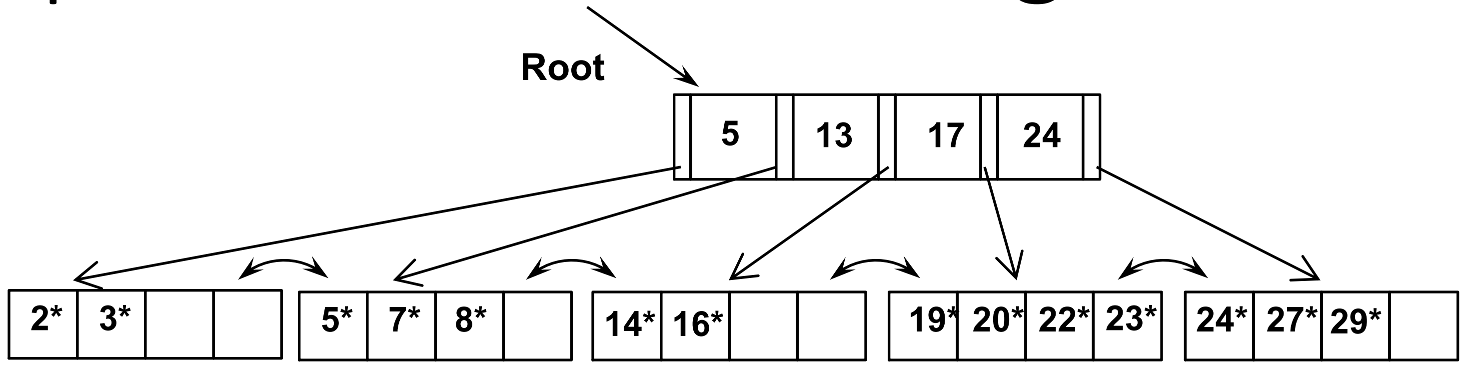


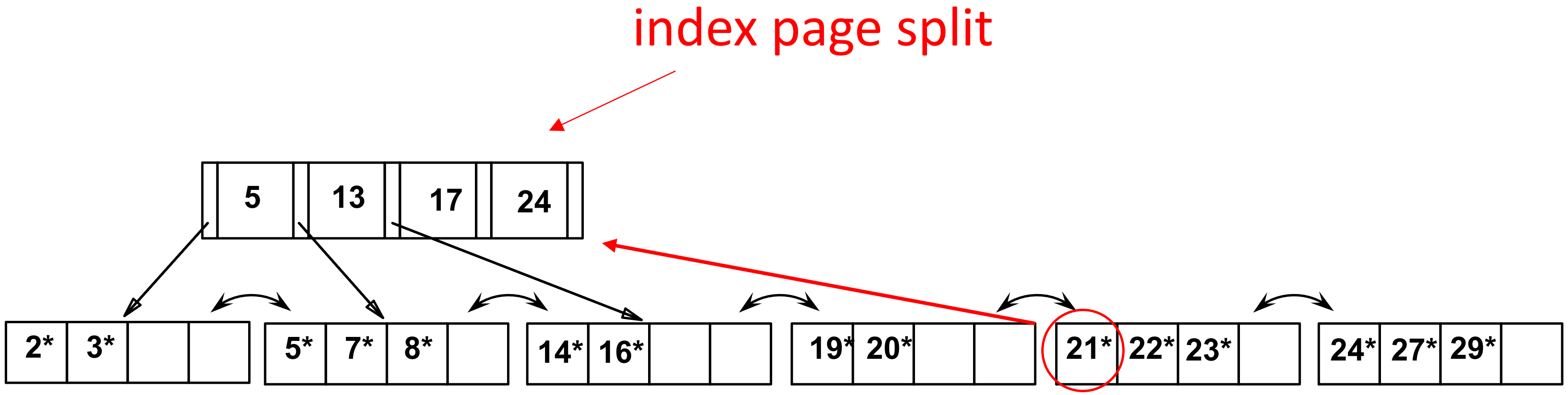
Example B+ Tree - Inserting 21*



BOSTON UNIVERSITY

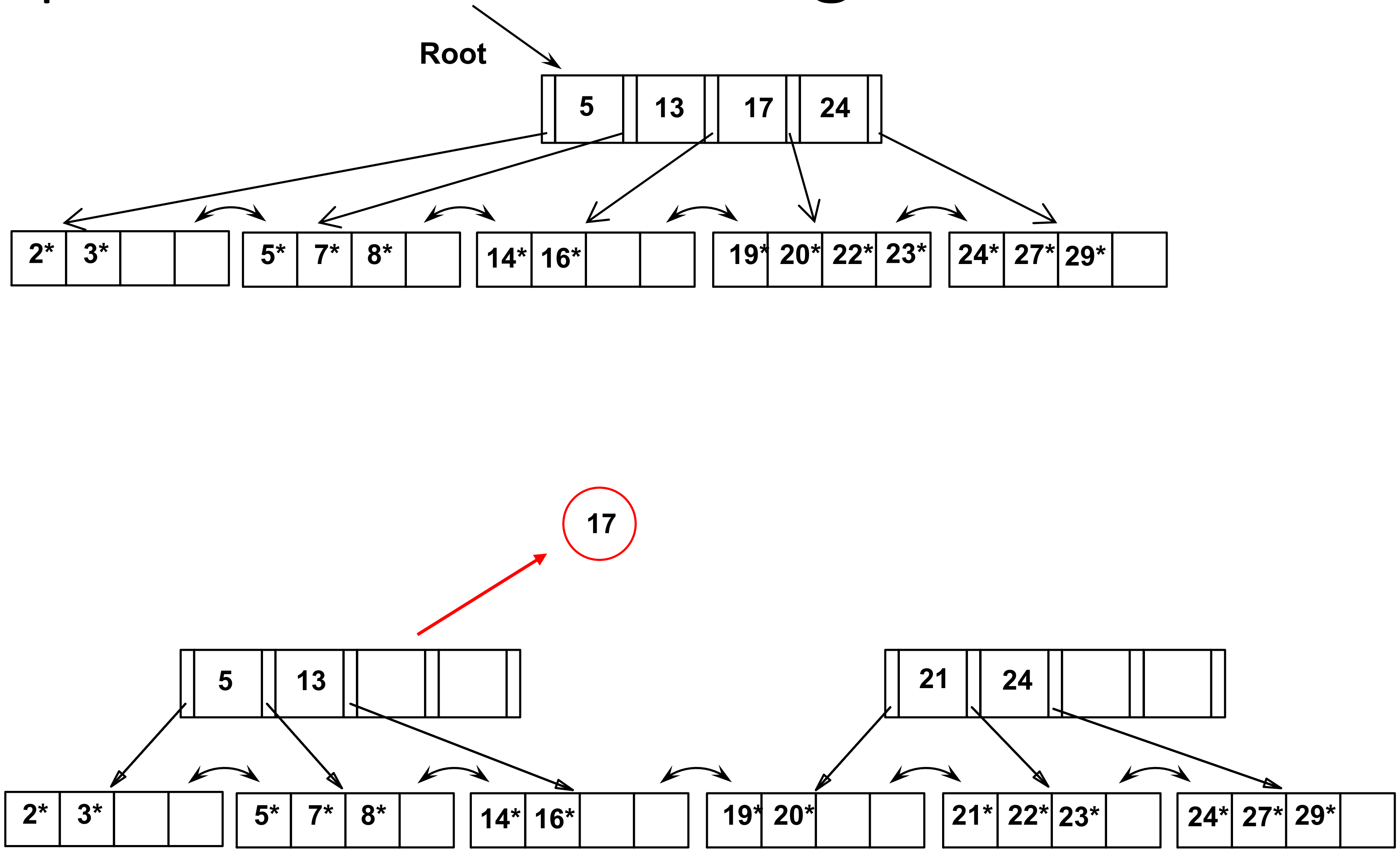
Example B+ Tree - Inserting 21*





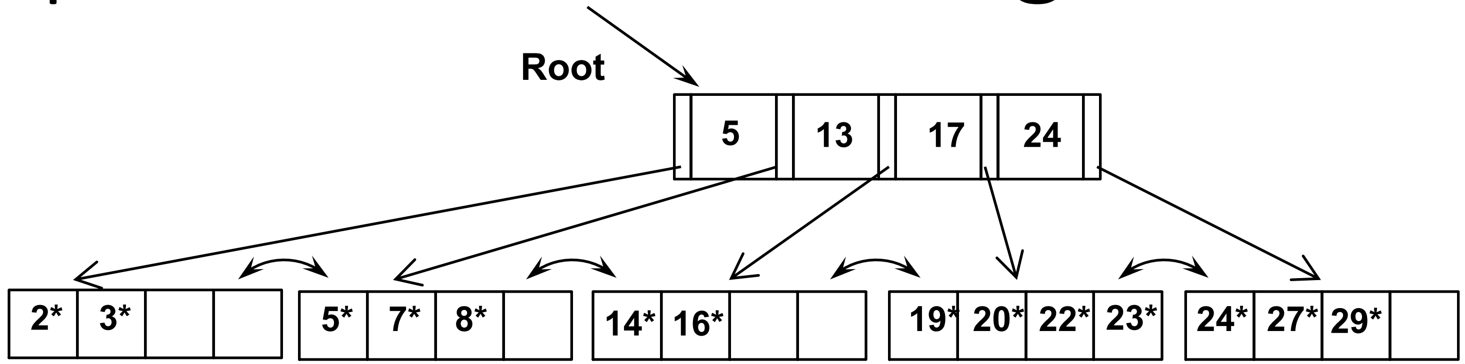
BOSTON UNIVERSITY

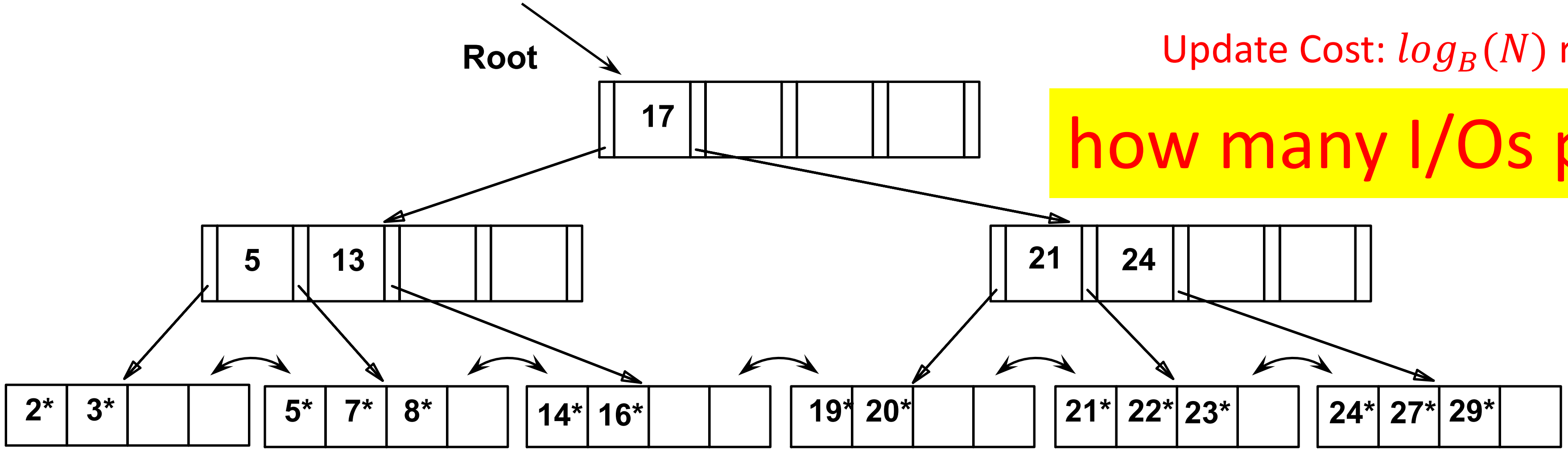
Example B+ Tree - Inserting 21*



BOSTON UNIVERSITY

Example B+ Tree - Inserting 21*





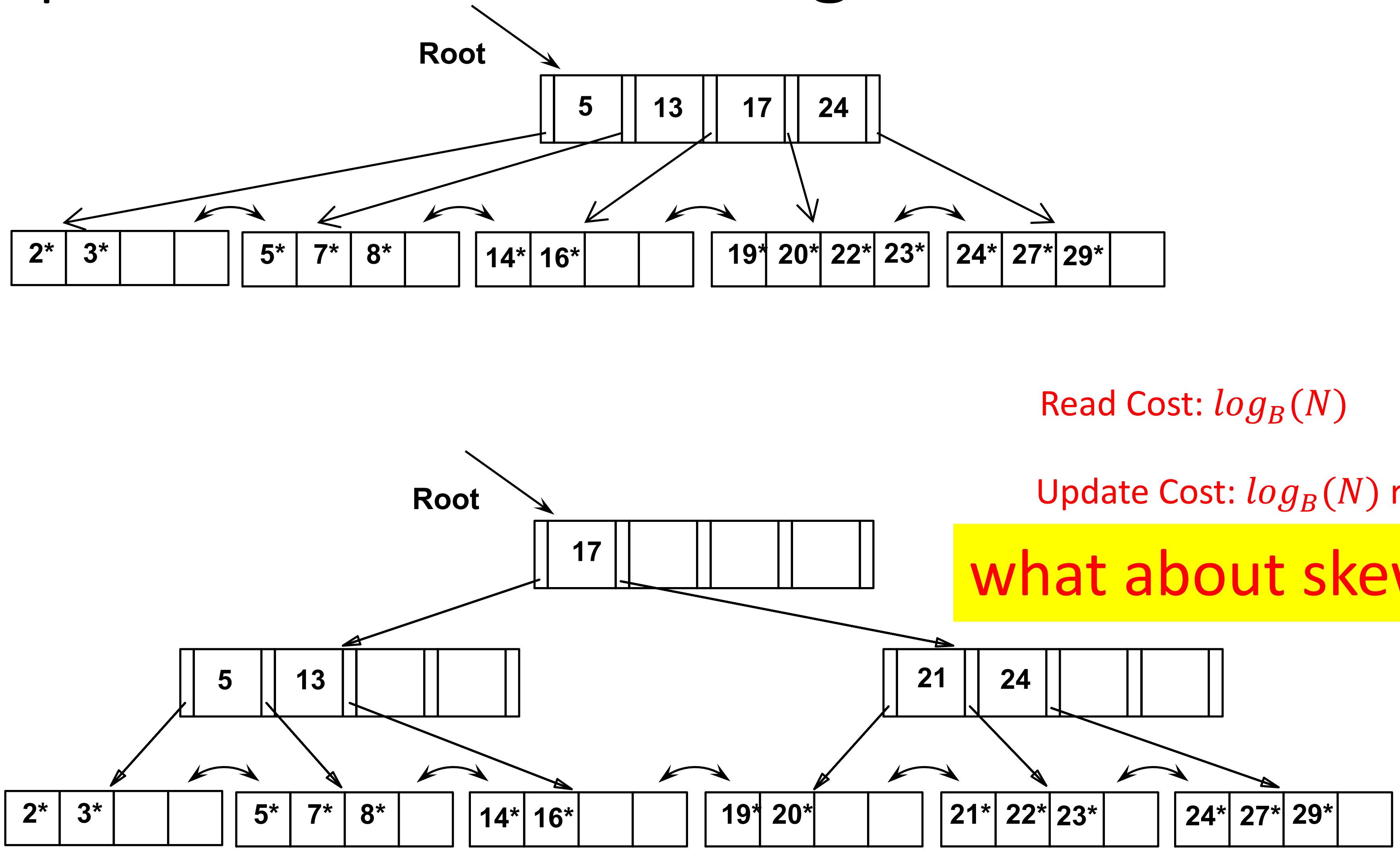
Read Cost: $log_B(N)$

Update Cost: $log_B(N)$ reads

how many I/Os per insert?

BOSTON UNIVERSITY

Example B+ Tree - Inserting 21*



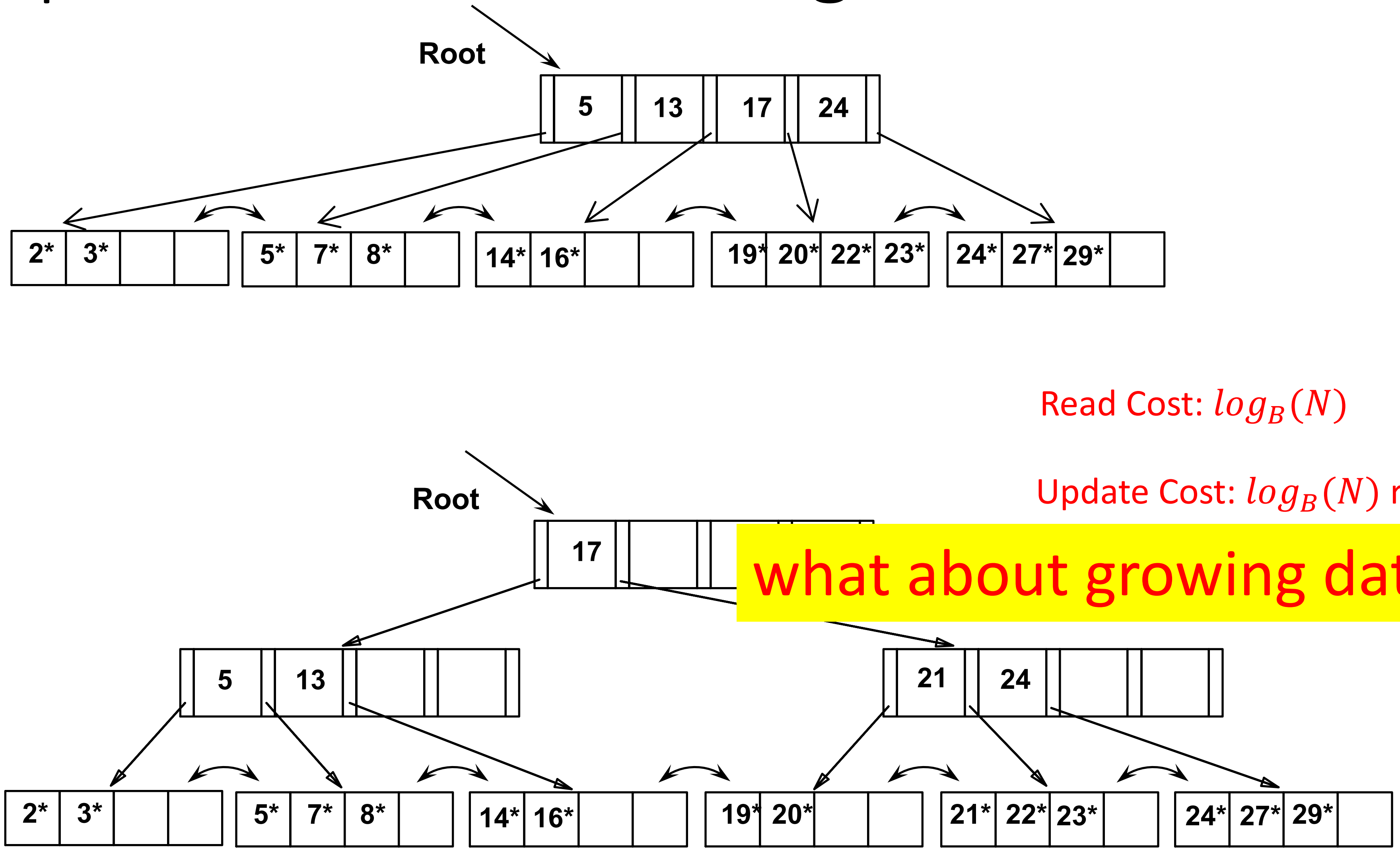
Read Cost: $log_B(N)$

Update Cost: $log_B(N)$ reads

what about skewed data?

BOSTON UNIVERSITY

Example B+ Tree - Inserting 21*



Read Cost: $log_B(N)$

Update Cost: $log_B(N)$ reads

what about growing dataset size?

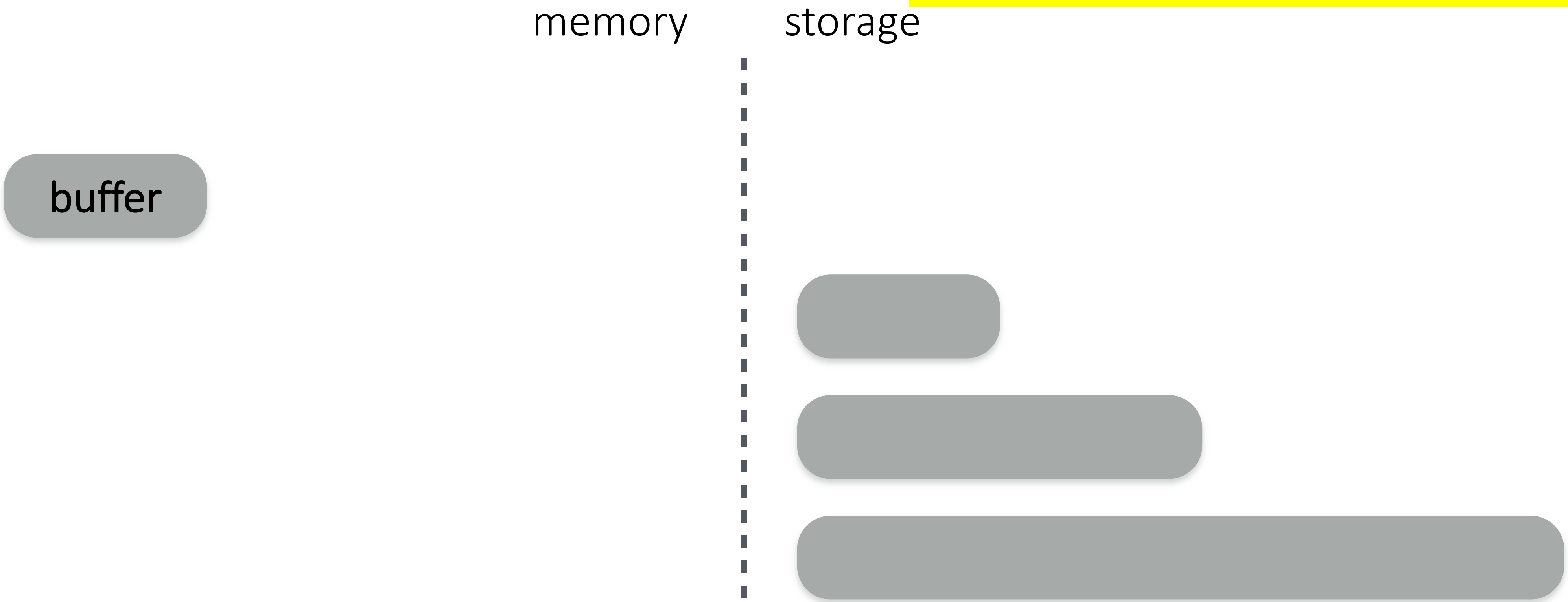
What are the possible ***index designs***?

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

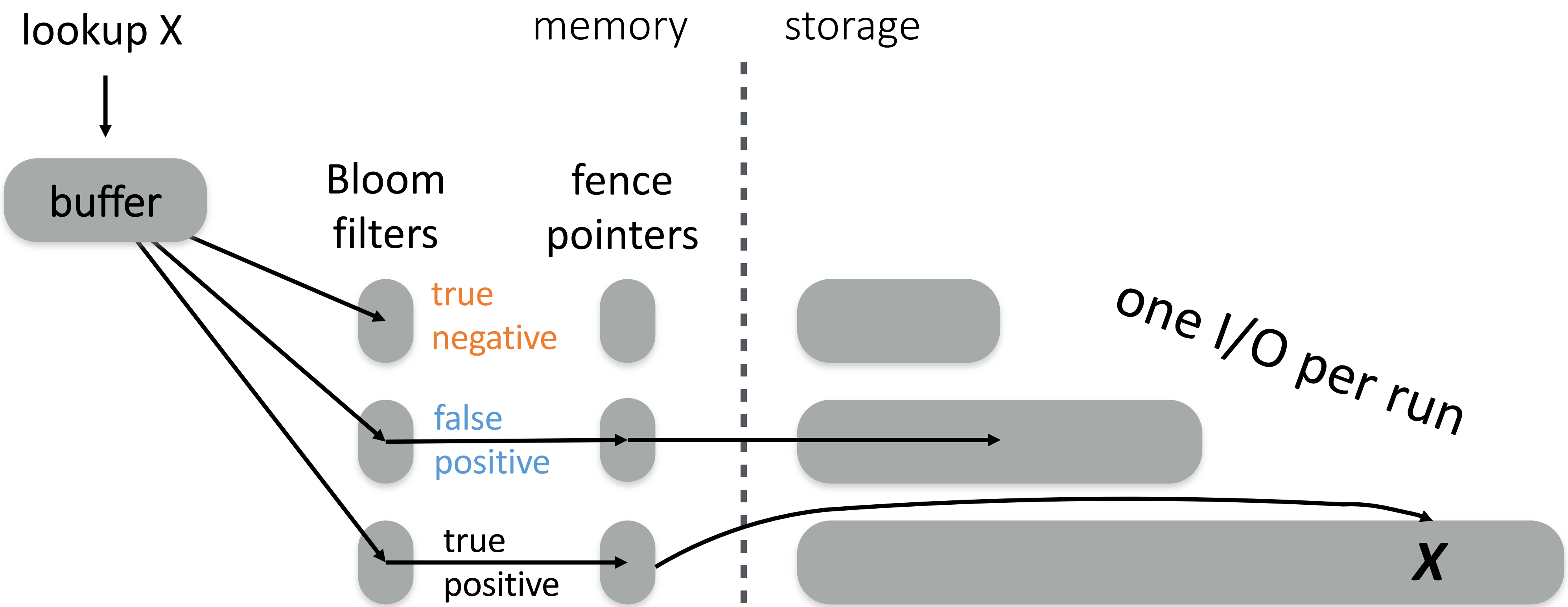
BOSTON UNIVERSITY

LSM-trees

Is LSM-tree an index at all?



LSM-trees


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

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					

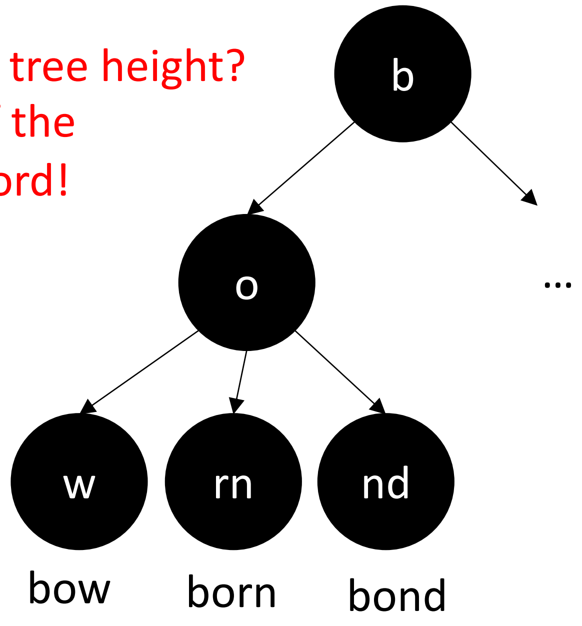
BOSTON UNIVERSITY

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!



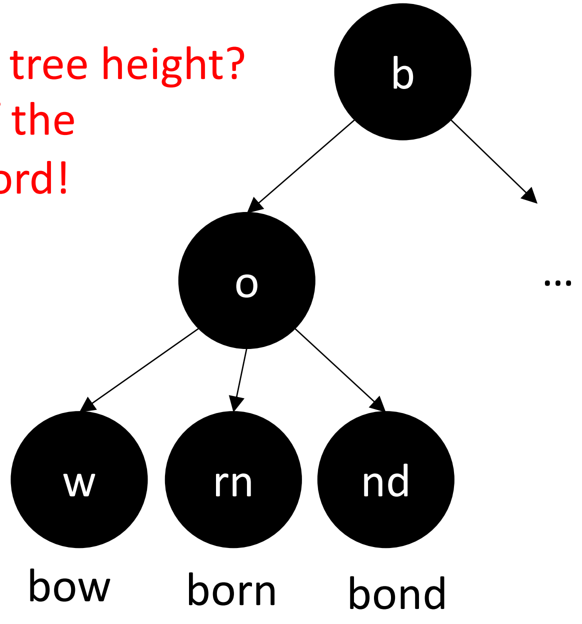
BOSTON UNIVERSITY

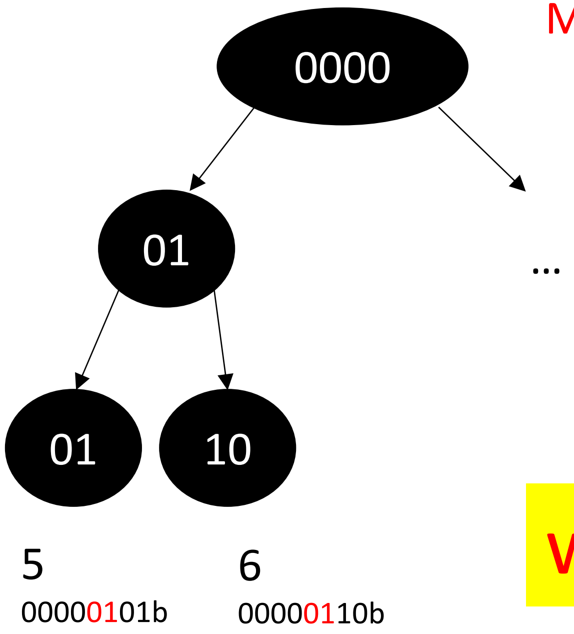
Radix Trees (special case of tries and prefix B-Trees)

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?

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

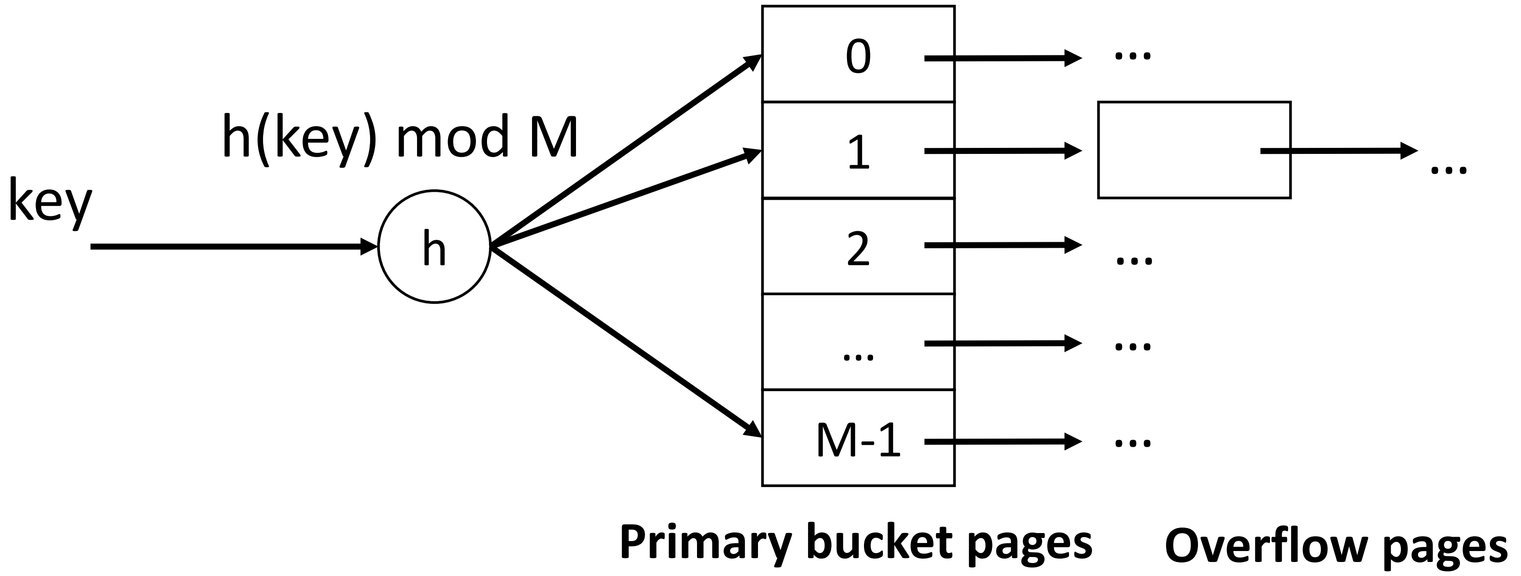
	Point Queries	Short Range Queries	Long Range Queries	Data Skew	Updates
B+ Trees	✅	✅	✅	✅	➰
LSM Trees	✅	❌	➰	✅	✅
Radix Trees	✅	✅	✅	❌	➰
Hash Indexes					
Bitmap Indexes					
Scan Accelerators					

BOSTON UNIVERSITY

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)



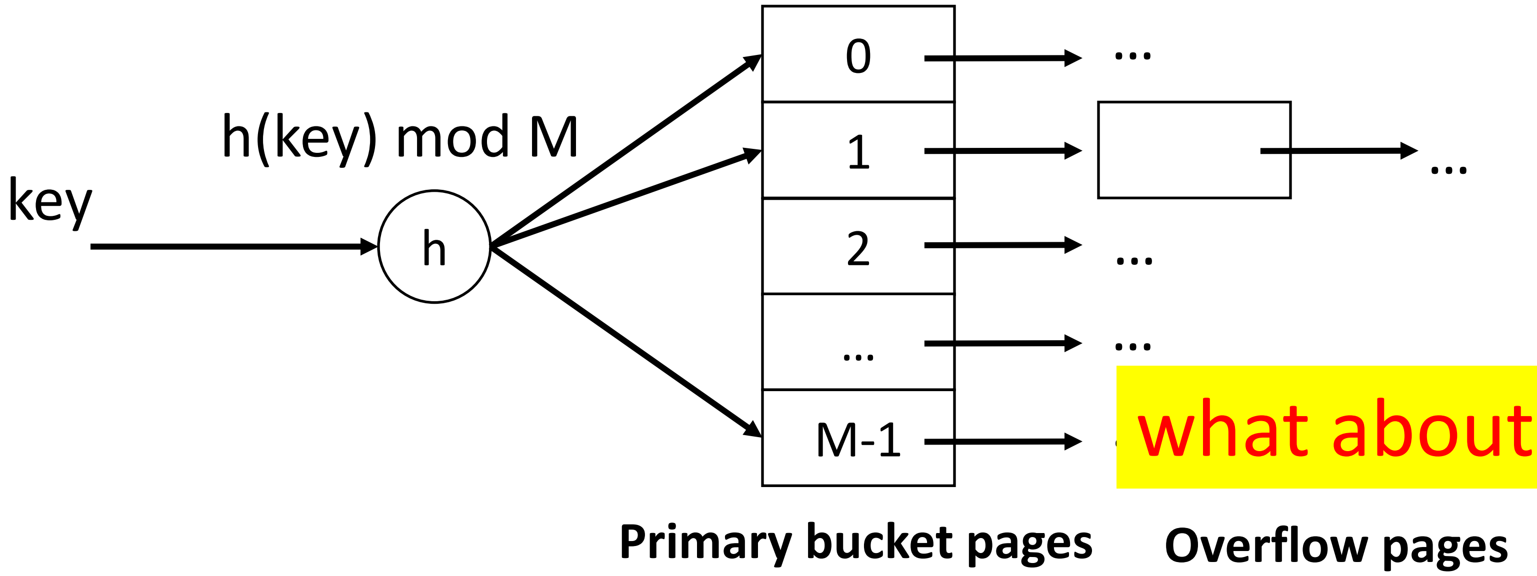
BOSTON UNIVERSITY

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



what about range queries?

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					

BOSTON UNIVERSITY

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

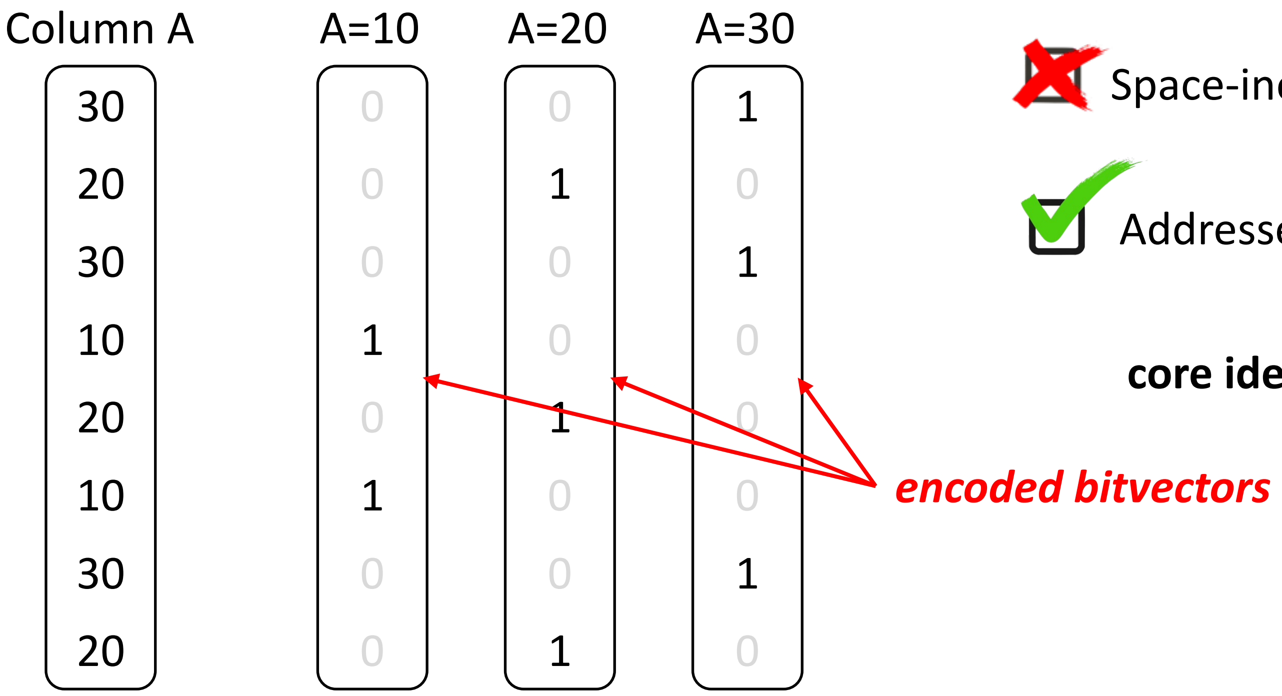
Speed & Size

- 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

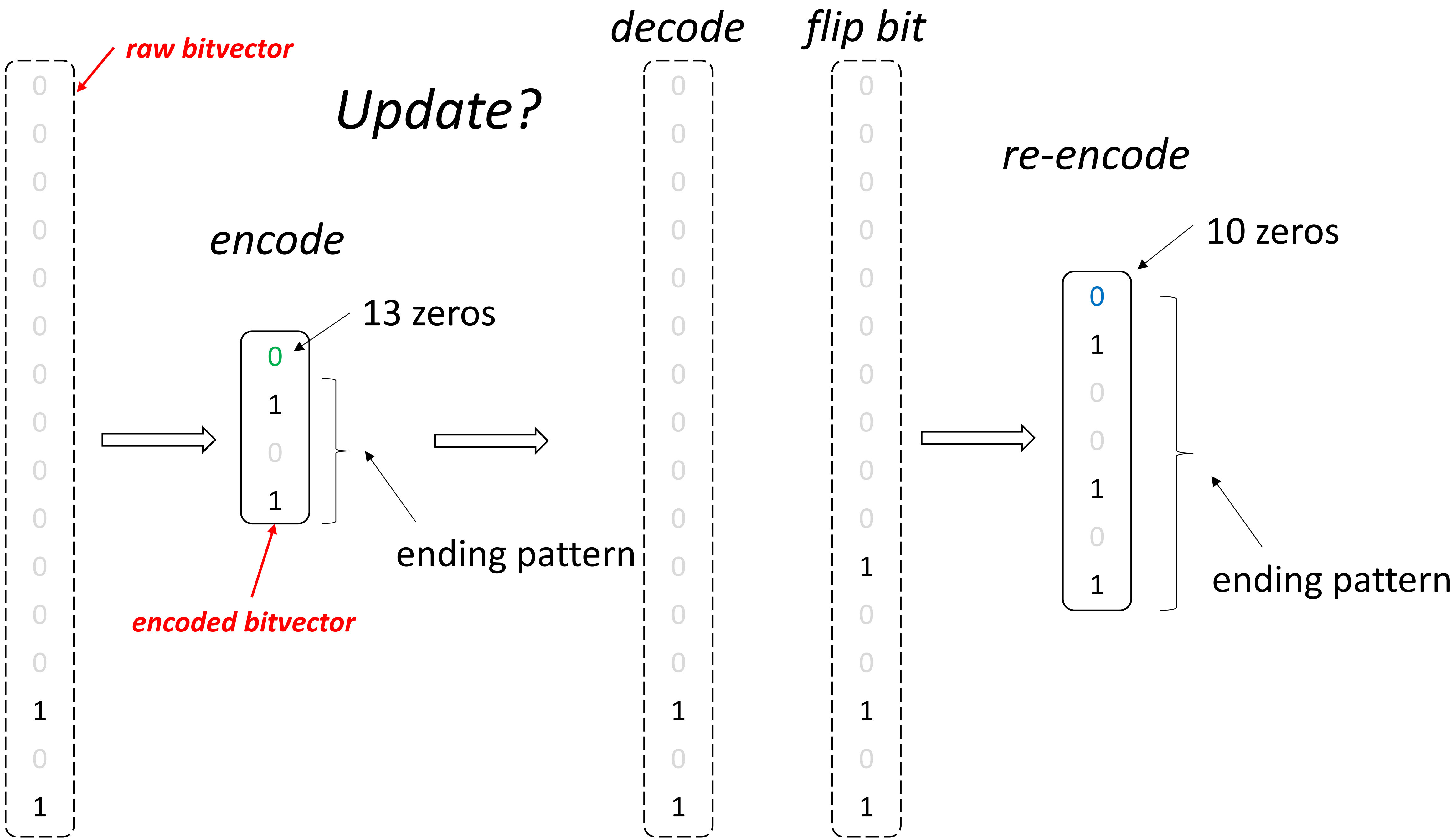


Index Size

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

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

what about updates?



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					

BOSTON UNIVERSITY

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]

BOSTON UNIVERSITY

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?

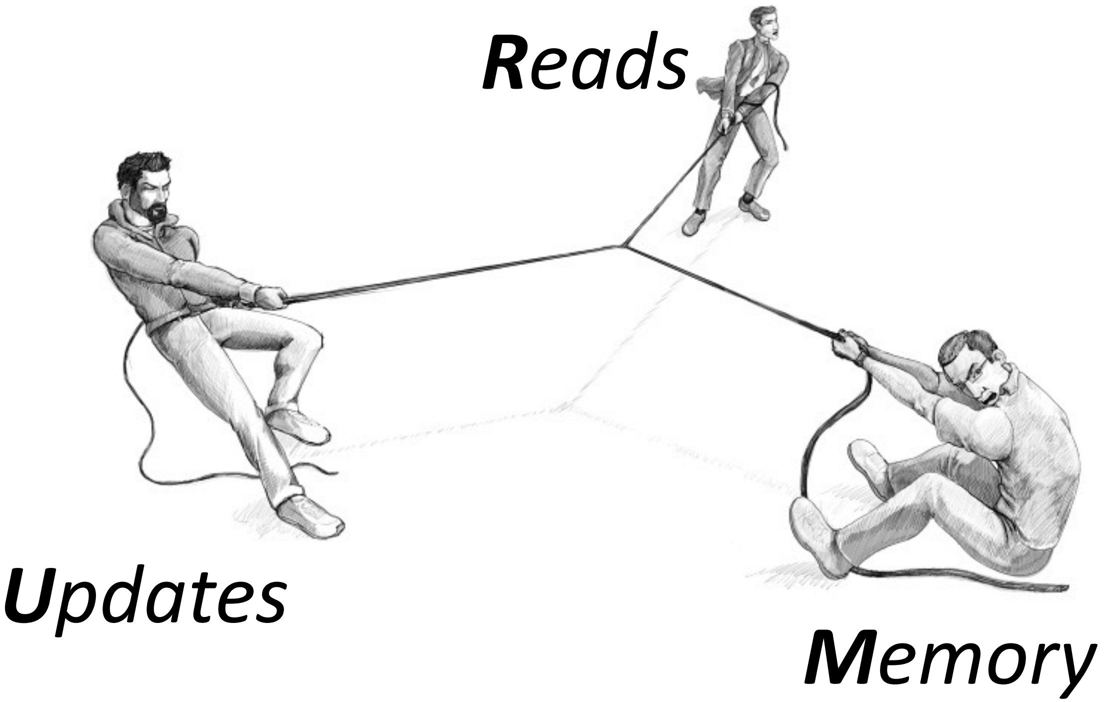
Z1: [1,99]	Z2: [2,95]	Z3: [1,100]	Z4: [2,100]	Z5: [3,97]	Z6: [2,99]

BOSTON UNIVERSITY

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	❌	~	✅	✅	~

data structure designs navigate a three-way tradeoff

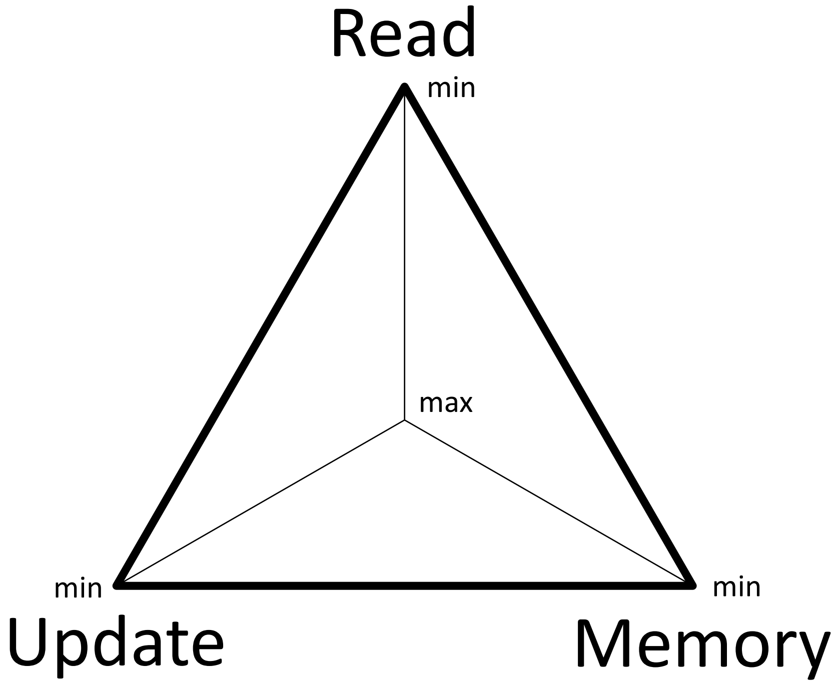


The RUM Conjecture

every access method has a (quantifiable)

- read overhead
- update overhead
- memory overhead

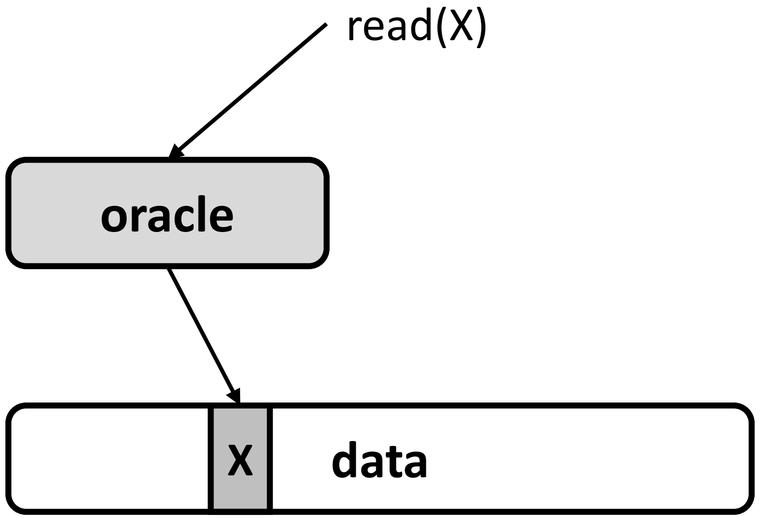
the three of which form a competing triangle



we can optimize for two of the overheads at the expense of the third

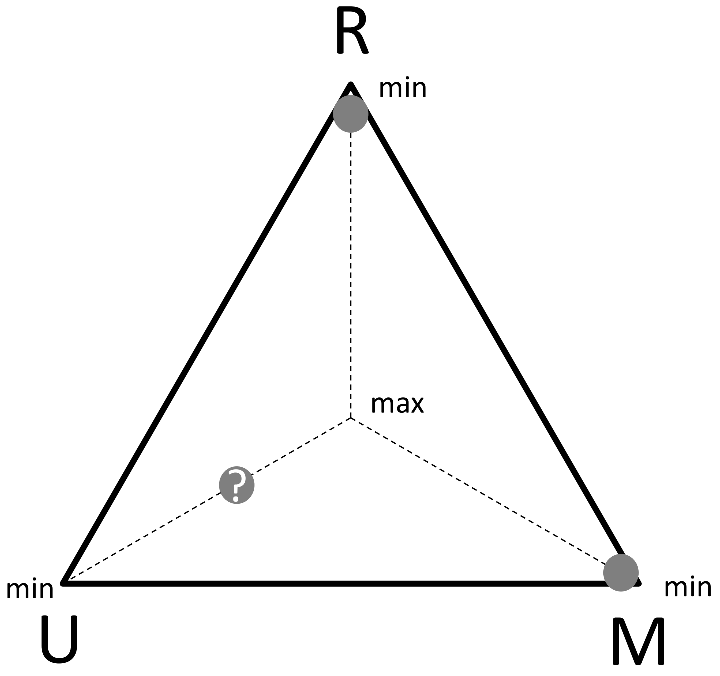
BOSTON UNIVERSITY

what would be an **optimal read** behavior?



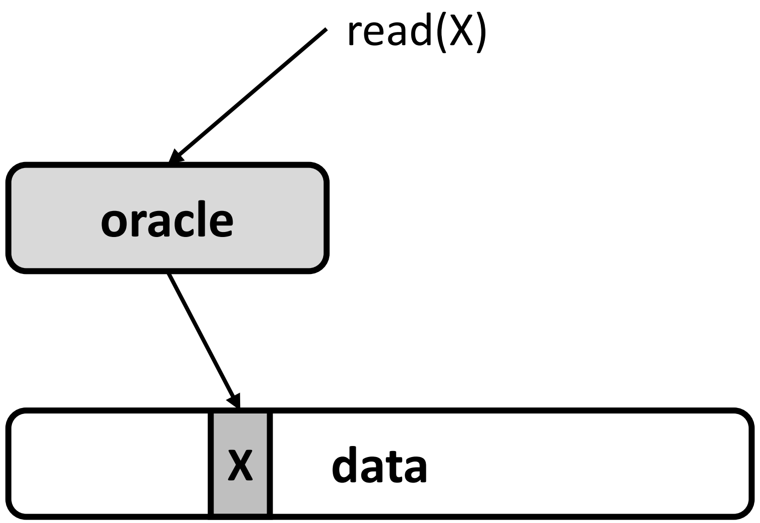
read(x) accesses only the bytes of object X

how *free* can an oracle be?



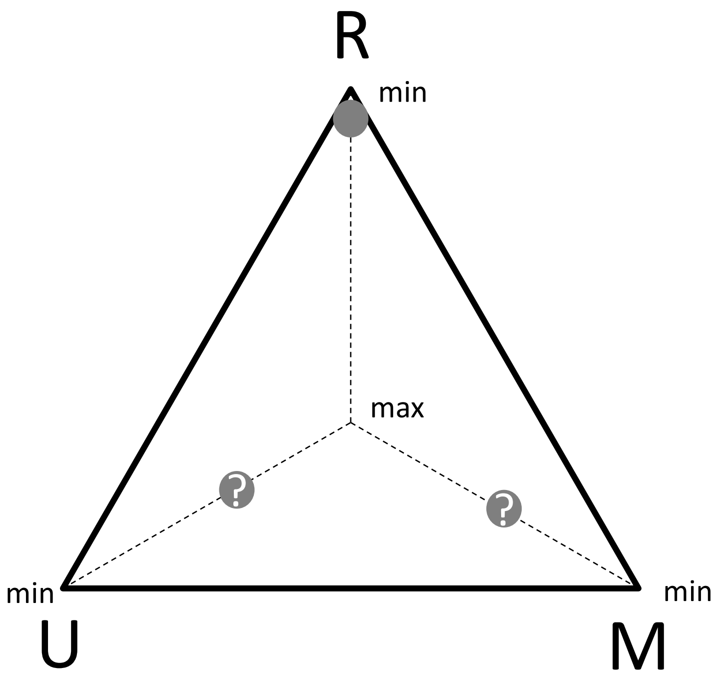
BOSTON UNIVERSITY

what would be an **optimal read** behavior?

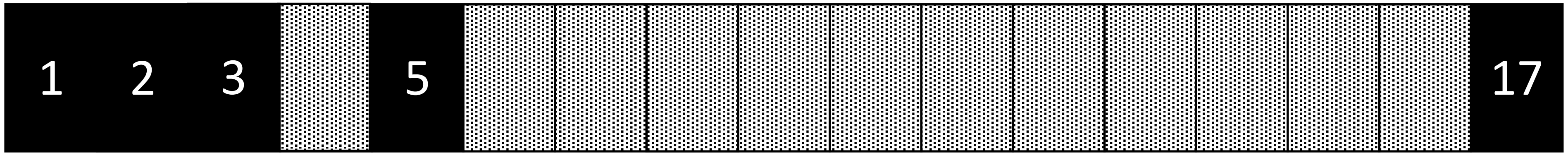


read(x) accesses only the bytes of object X

how *free* can an oracle be?



what would be an **optimal read** behavior?



update 27-> 3

minimum read overhead

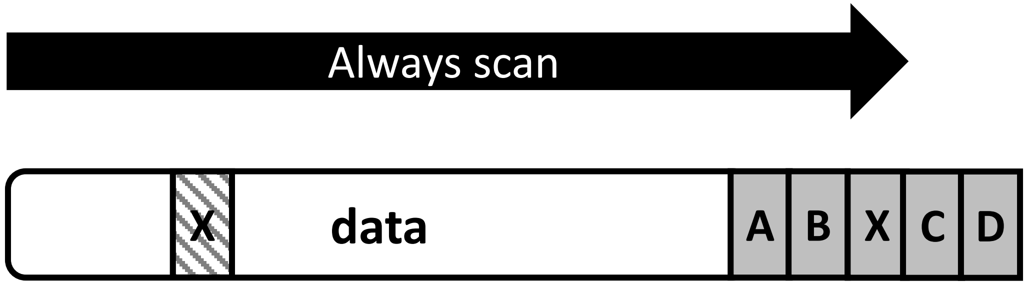
bound update overhead

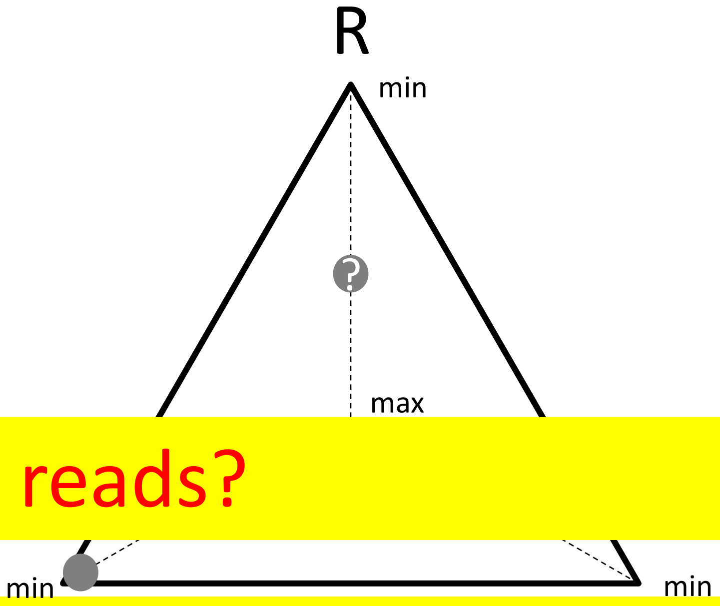
unbounded memory overhead

what would be an **optimal update** behavior?

always *append*, and on update *invalidate*

update (X) changes the minimal number of bytes





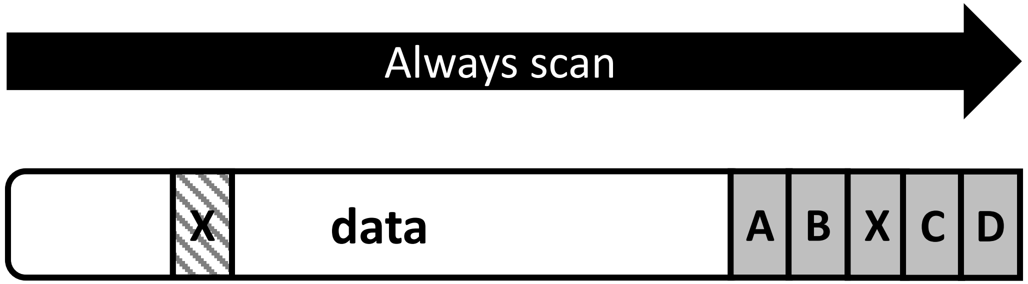
what about reads?

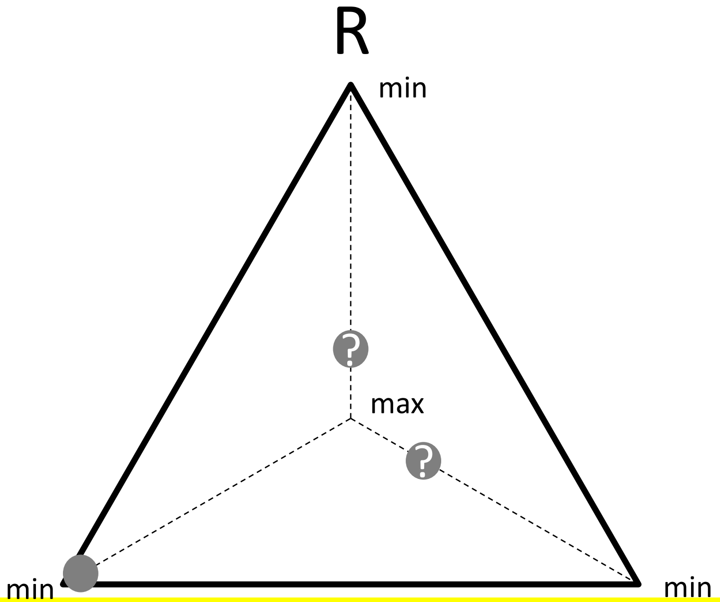
more data?

what would be an **optimal update** behavior?

always *append*, and *invalidate* on update

update (X) changes the minimal number of bytes



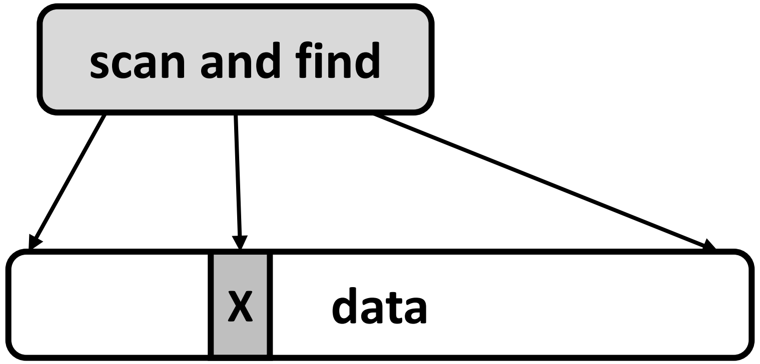


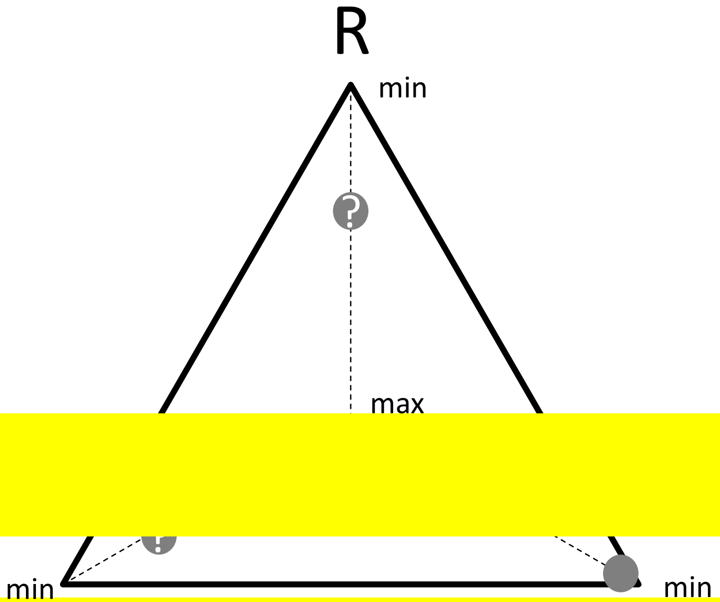
higher read and memory overhead

what would be an **optimal memory** overhead?

no metadata whatsoever, would result in the smallest memory footprint

scan and in-place updates

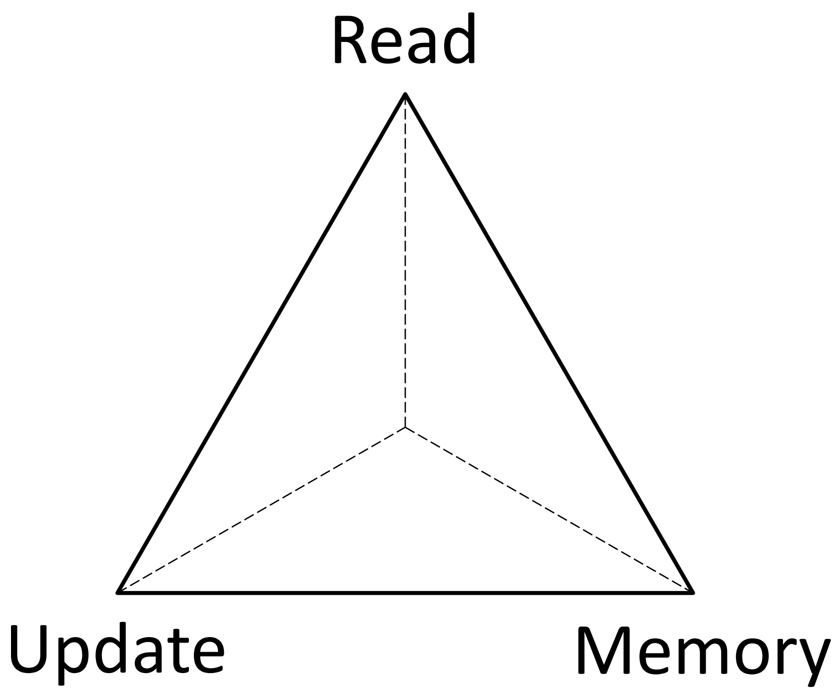




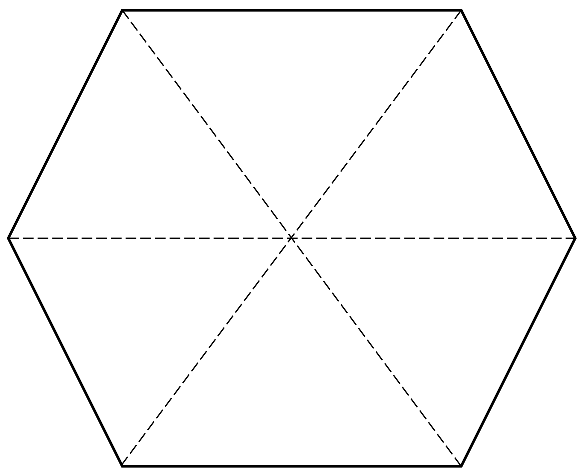
No!

do we need to reach the optimal(s)?

are there only three overheads?



are there only three overheads?



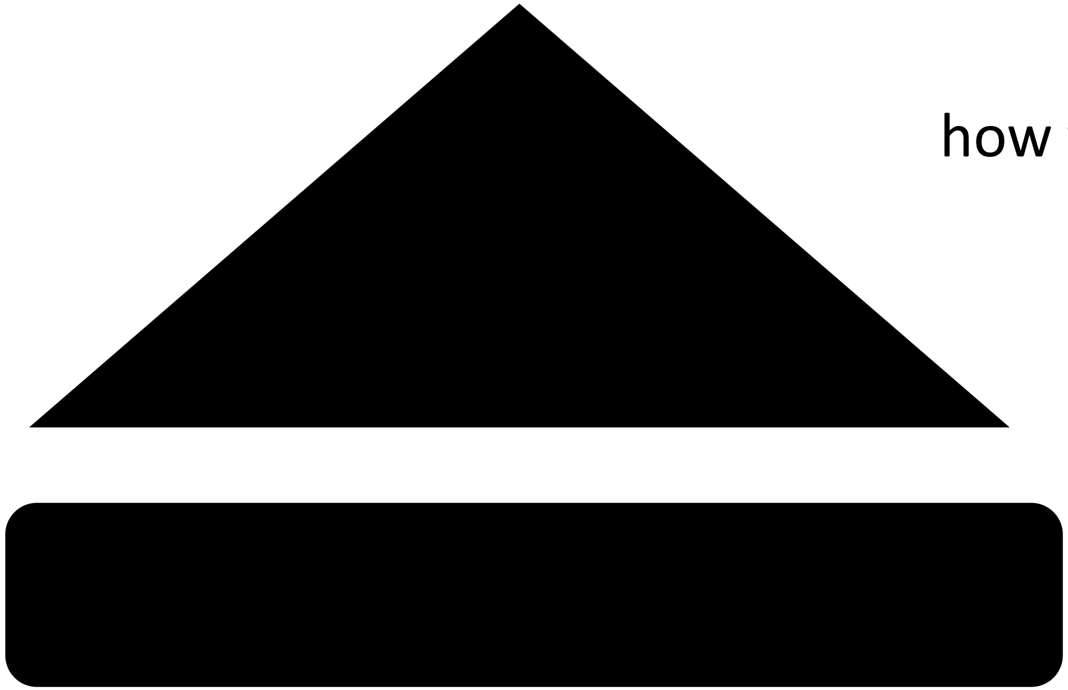
PyRUMID overheads

BOSTON UNIVERSITY

how to decide how to ***design*** a data structure?

break it down to ***design dimensions***

how to break down the *design* in independent *dimensions*?



how to physically organize the data?

how to search through the data?

can I accelerate search through metadata?

multiple levels of nested organization?

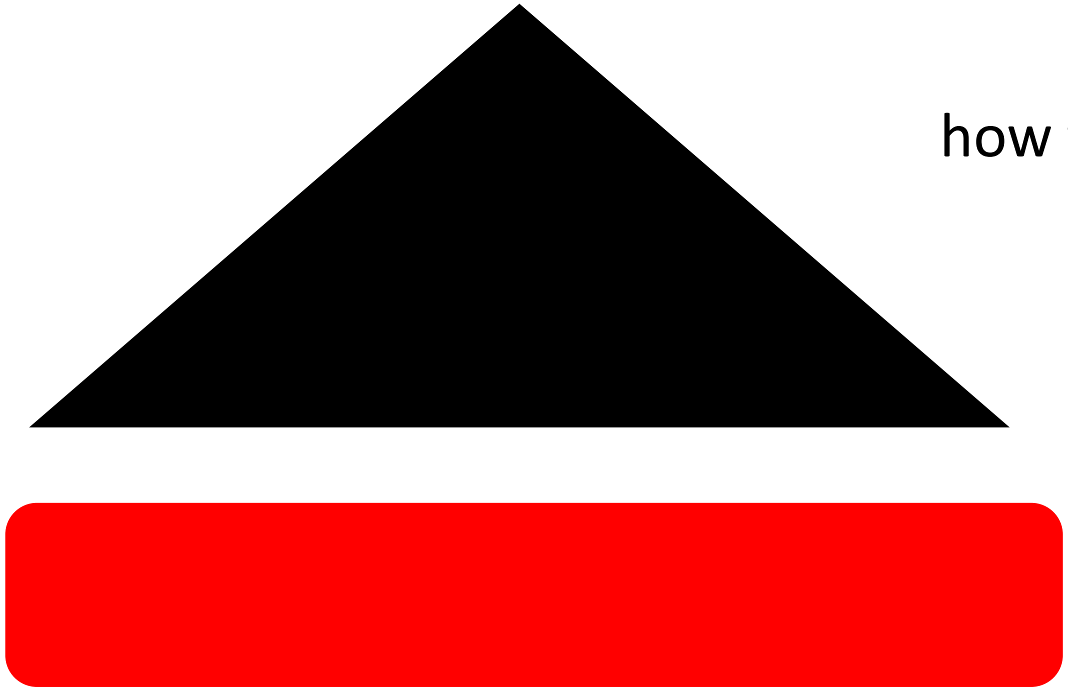
how to update or add new data?

how to exploit additional memory/storage?

should the above decisions be applied eagerly or lazily?

BOSTON UNIVERSITY

how to break down the *design* in independent *dimensions*?



global data organization

how to search through the data?

can I accelerate search through metadata?

multiple levels of nested organization?

how to update or add new data?

how to exploit additional memory/storage?

should the above decisions be applied eagerly or lazily?

how to break down the *design* in independent *dimensions*?

global data organization

global search algorithm

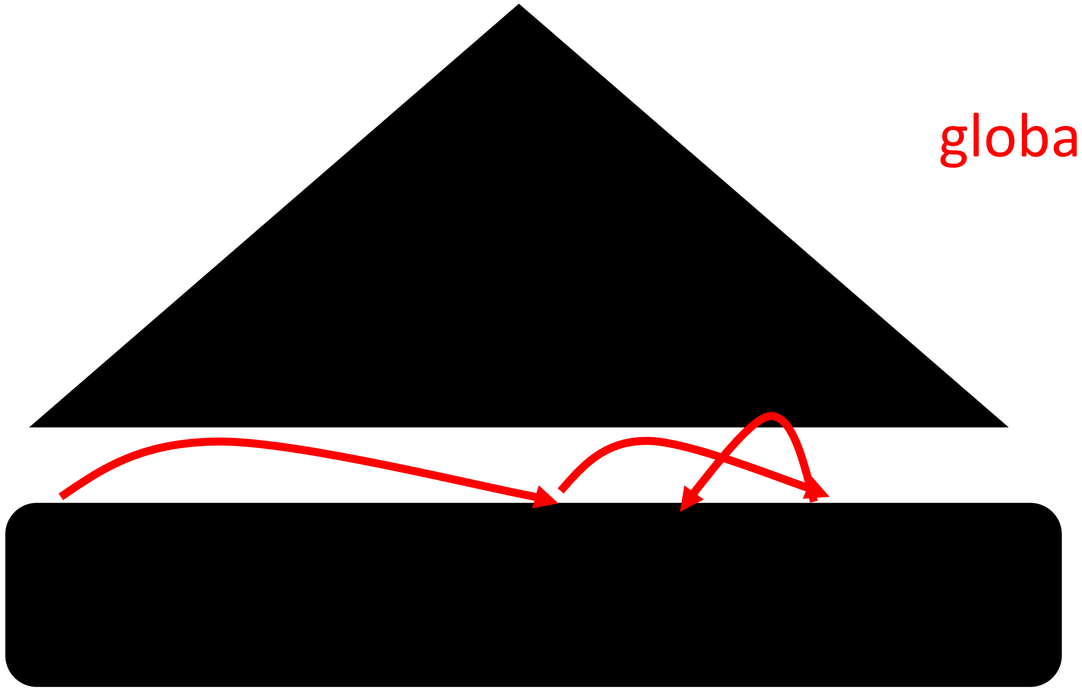
can I accelerate search through metadata?

multiple levels of nested organization?

how to update or add new data?

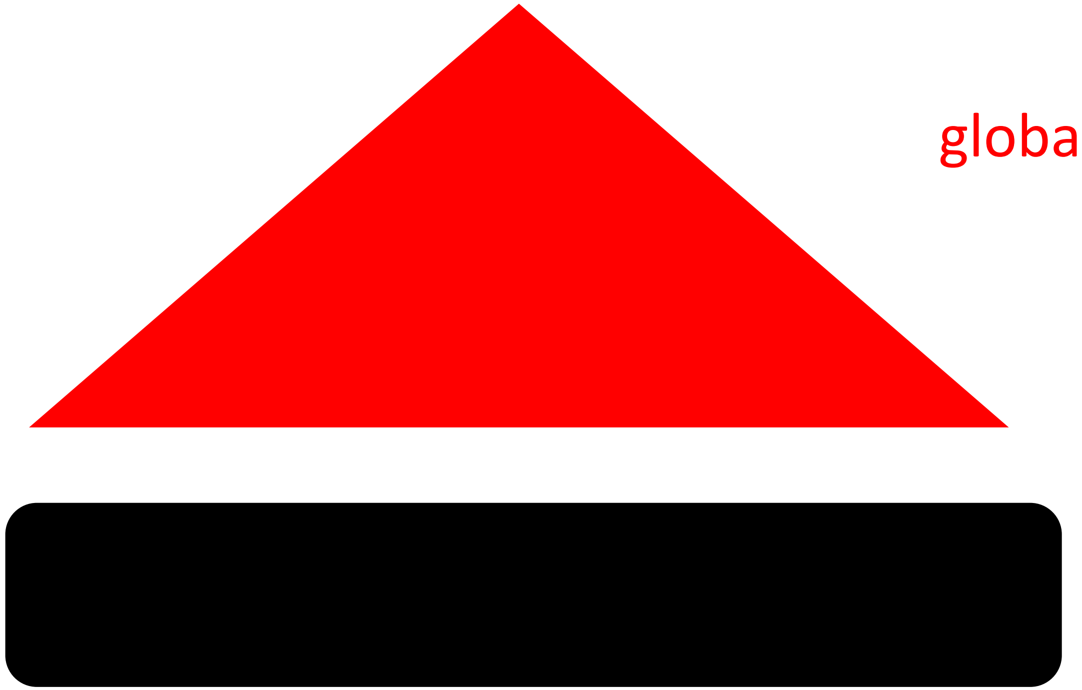
how to exploit additional memory/storage?

should the above decisions be applied eagerly or lazily?



BOSTON UNIVERSITY

how to break down the *design* in independent *dimensions*?



global data organization

global search algorithm

metadata for searching

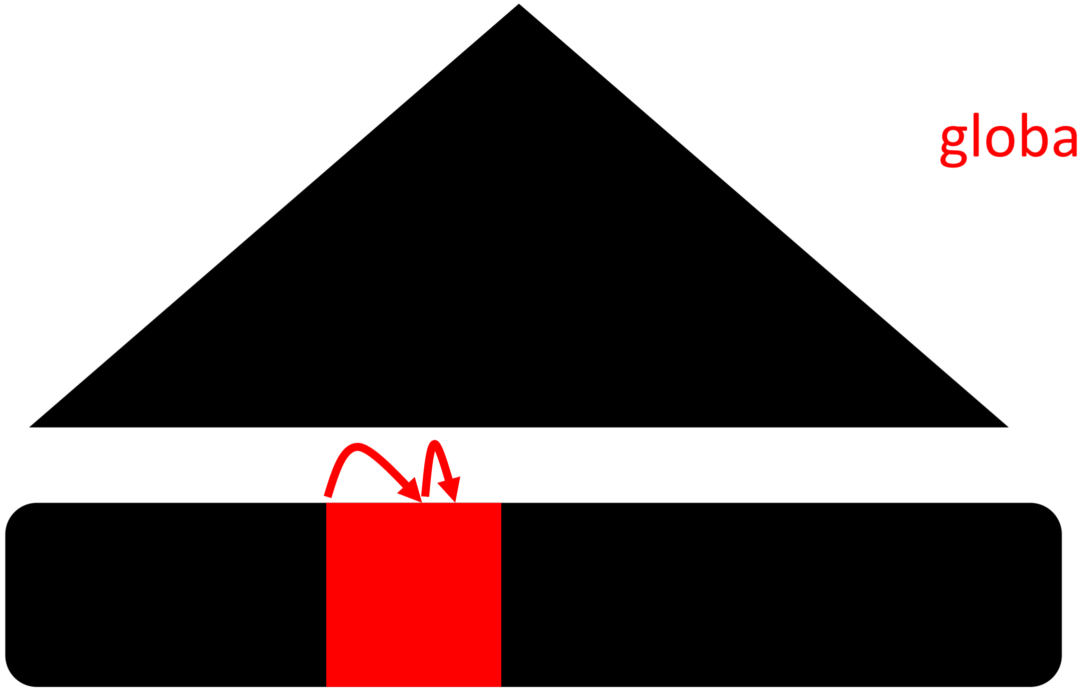
multiple levels of nested organization?

how to update or add new data?

how to exploit additional memory/storage?

should the above decisions be applied eagerly or lazily?

how to break down the *design* in independent *dimensions*?



global data organization

global search algorithm

metadata for searching

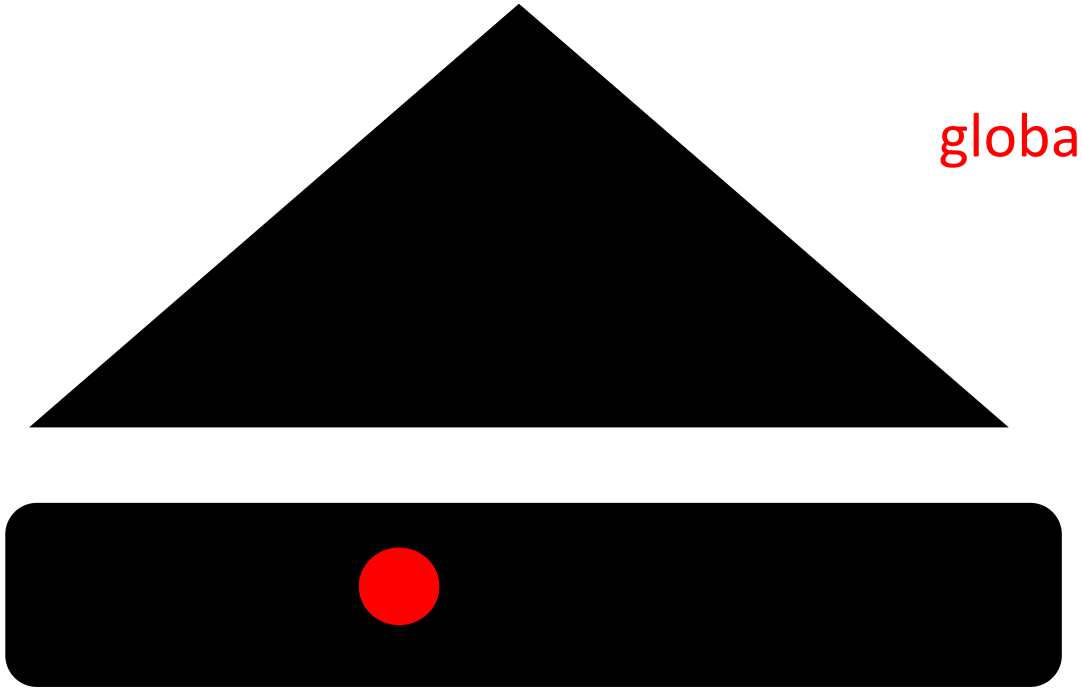
local data organization & search algorithm

how to update or add new data?

how to exploit additional memory/storage?

should the above decisions be applied eagerly or lazily?

how to break down the *design* in independent *dimensions*?



global data organization

global search algorithm

metadata for searching

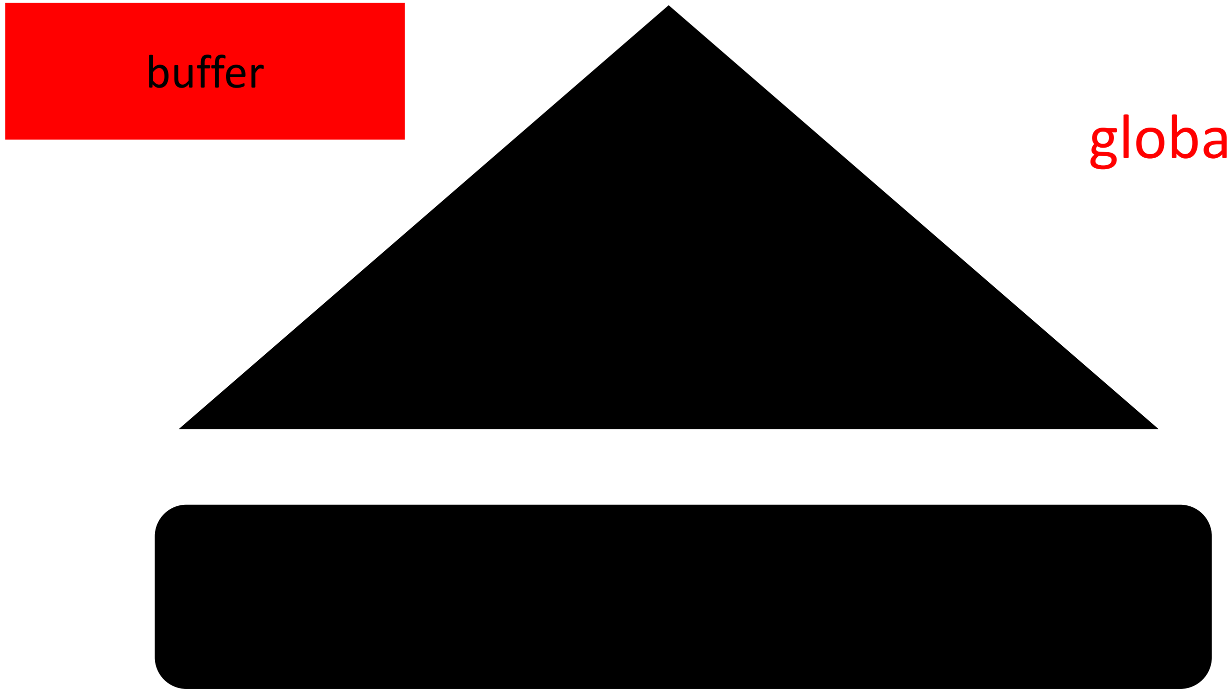
local data organization & search algorithm

modification policy

how to exploit additional memory/storage?

should the above decisions be applied eagerly or lazily?

how to break down the *design* in independent *dimensions*?



global data organization

global search algorithm

metadata for searching

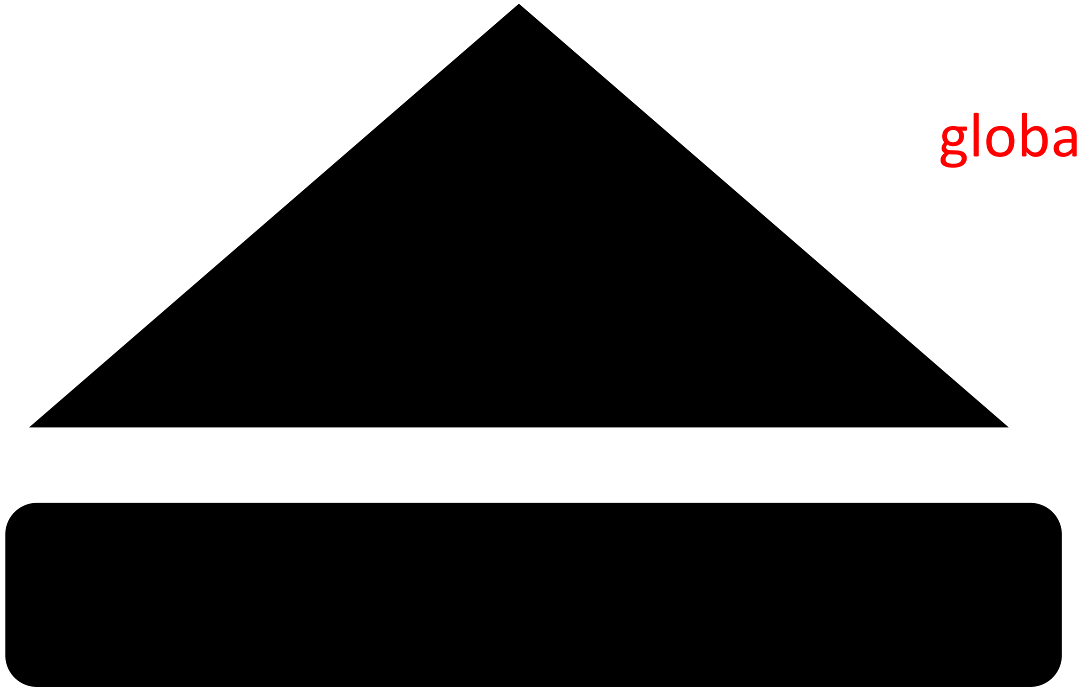
local data organization & search algorithm

modification policy

batching via buffering

should the above decisions be applied eagerly or lazily?

how to break down the *design* in independent *dimensions*?



global data organization

global search algorithm

metadata for searching

local data organization & search algorithm

modification policy

batching via buffering

adaptivity

data structures ***design dimensions and their values***

global data organization

global search algorithm

metadata for searching

local data organization & search algorithm

modification policy

batching via buffering

adaptivity

BOSTON UNIVERSITY

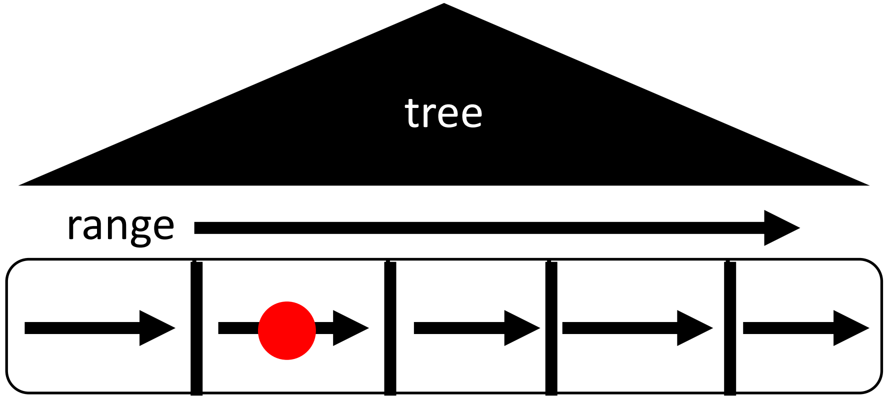
how to break down ***popular designs***
to those design decisions?

b+ trees

global data organization	**range partitioning**
global searching (algorithm or index)	**search tree**
local data organization	**sorted**
local search algorithm	**binary search / scan**
modification policy	**in-place**

Workload? 

point and range queries and some modifications



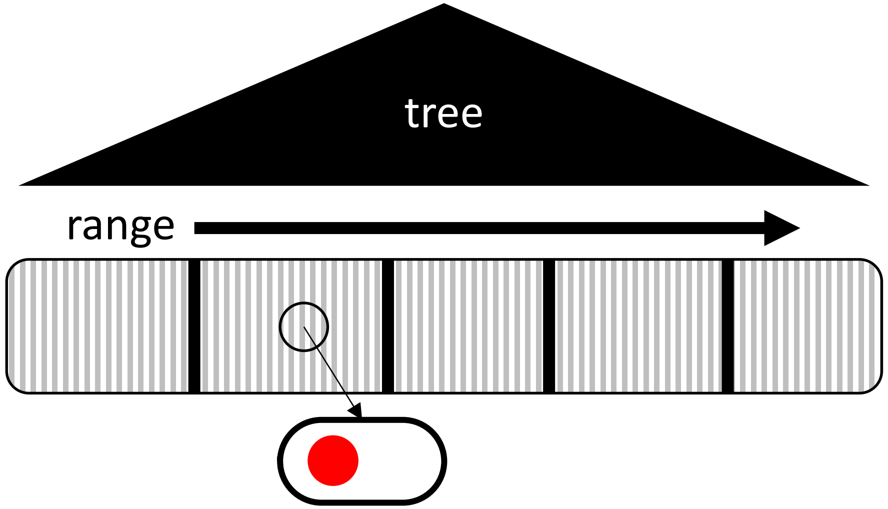
BOSTON UNIVERSITY

insert optimized b+ trees

global data organization	**range partitioning**
global searching (algorithm or index)	**search tree**
local data organization	**logging**
local search algorithm	**binary search / scan**
modification policy	**deferred in-place**

Workload? 

increased number of modifications



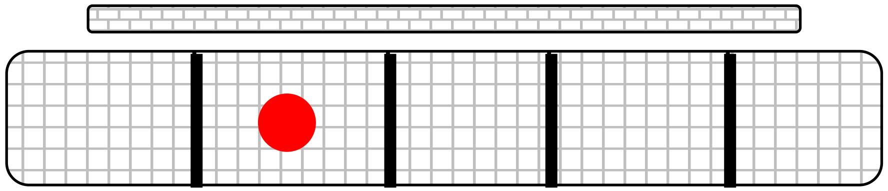
BOSTON UNIVERSITY

static hashing

global data organization	**hash partitioning**
global searching (algorithm or index)	**direct addressing (hashing)**
local data organization	**logging**
local search algorithm	**scan**
modification policy	**in-place**

Workload? 

point queries and modifications

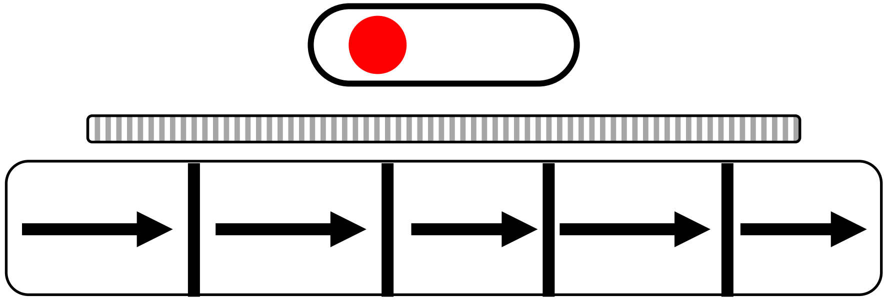


lsm-trees

global data organization	**partitioned logging**
global searching (algorithm or index)	**filter indexing**
local data organization	**sorted**
local search algorithm	**binary / data-driven search**
modification policy	**out-of-place**

Workload? 

modification-heavy with point and range queries



Projects

A) LSM-tree implementation

B) Bufferpool implementation

C) WA quantification for LSMs on SSDs
D) Range deletes in LSMs
E) Query-driven compactions in LSMs
F) Finding the optimal compaction strategy in LSMs to reduce WA
G) Finding the optimal indexing granularity for LSMs
H) Evaluating sorting algorithms for varied data sortedness
I) Measuring robustness in SplinterDB
J) Boosting join performance in Postgres for skewed correlation
K) Benchmarking large-scale graph processing systems

BOSTON UNIVERSITY

CS 561: **Data Systems Architectures**

Introduction to Indexing:

Trees, Tries, Hashing, Bitmaps: The Whole Design Space

Dr. Subhadeep Sarkar

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