BOSTON UNIVERSITY

CS 561: Data Systems Architectures

The *design space* of data structures

Prof. Manos Athanassoulis

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

data structures

b+ trees

hash tables

zonemaps

radix trees

bitmap indexes

are in the core of:

database systems

file systems

operating systems

machine learning systems

systems for data science

how to decide which one to use?

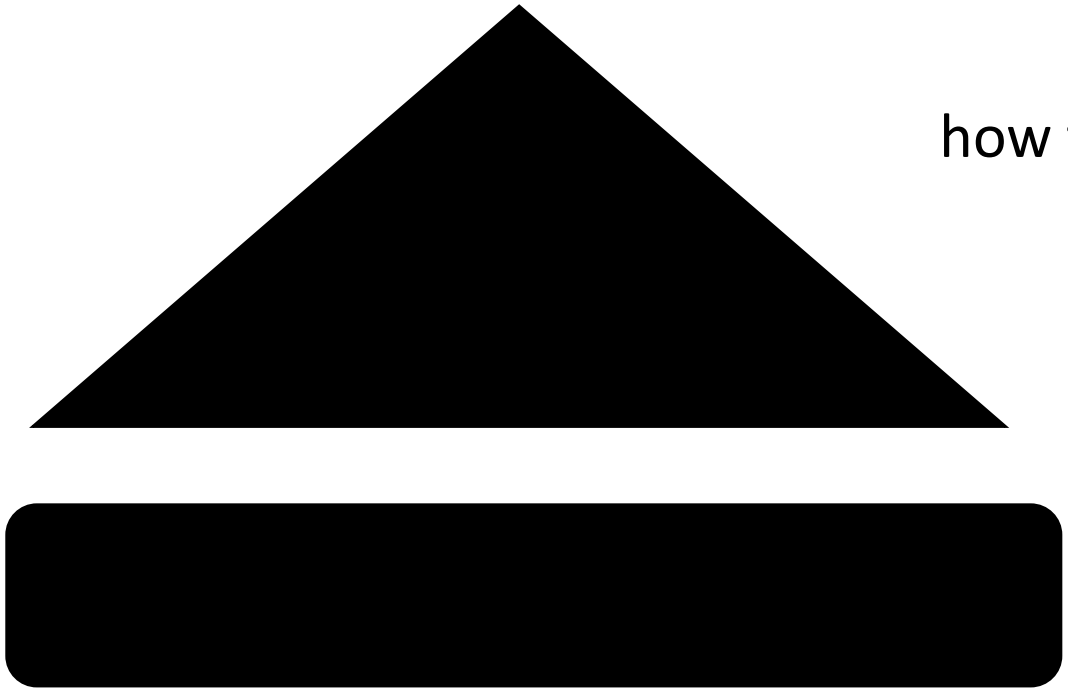
workload (access patterns) ← current focus

next → hardware (memory/storage/network/compute)

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?

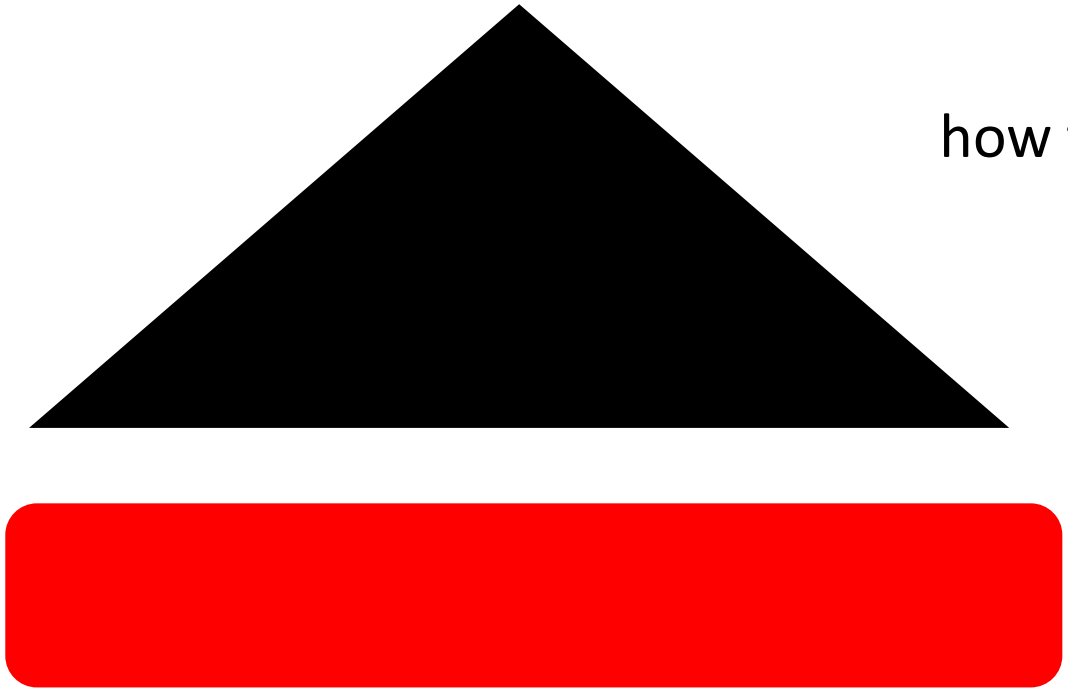
multiple levels of nested organization?

how to update or add new data?

how to exploit additional memory/storage?

should the above decisions be applied eagerly or lazily?

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



global data organization

how to search through the data?

can I accelerate search through metadata?

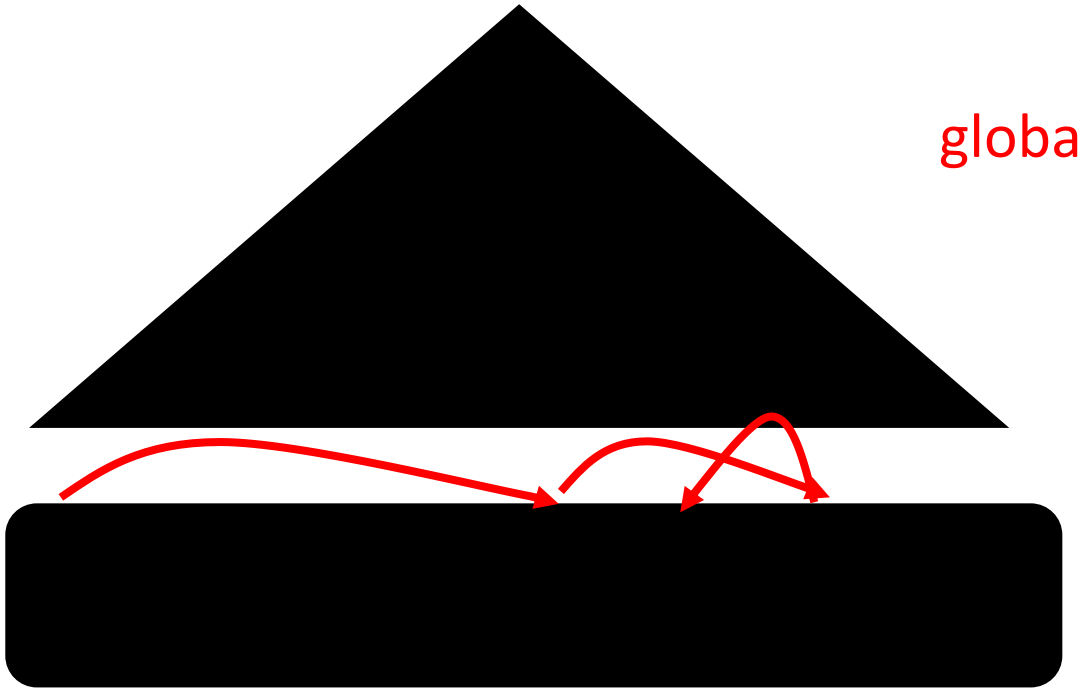
multiple levels of nested organization?

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?

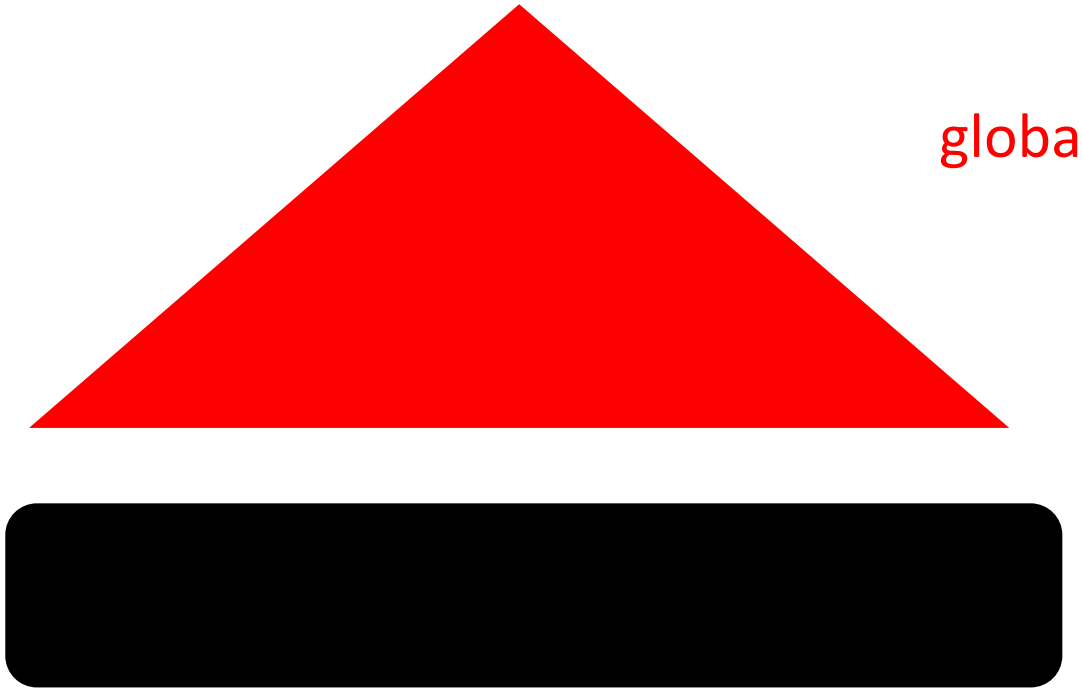
multiple levels of nested organization?

how to update or add new data?

how to exploit additional memory/storage?

should the above decisions be applied eagerly or lazily?

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



global data organization

global search algorithm

metadata for searching

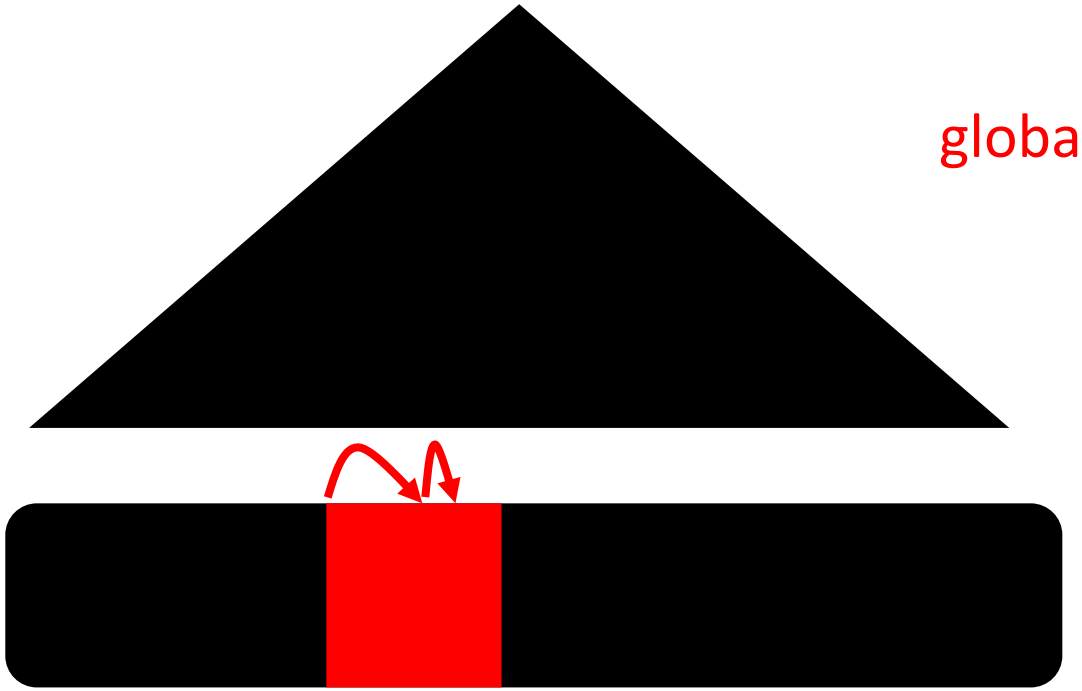
multiple levels of nested organization?

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

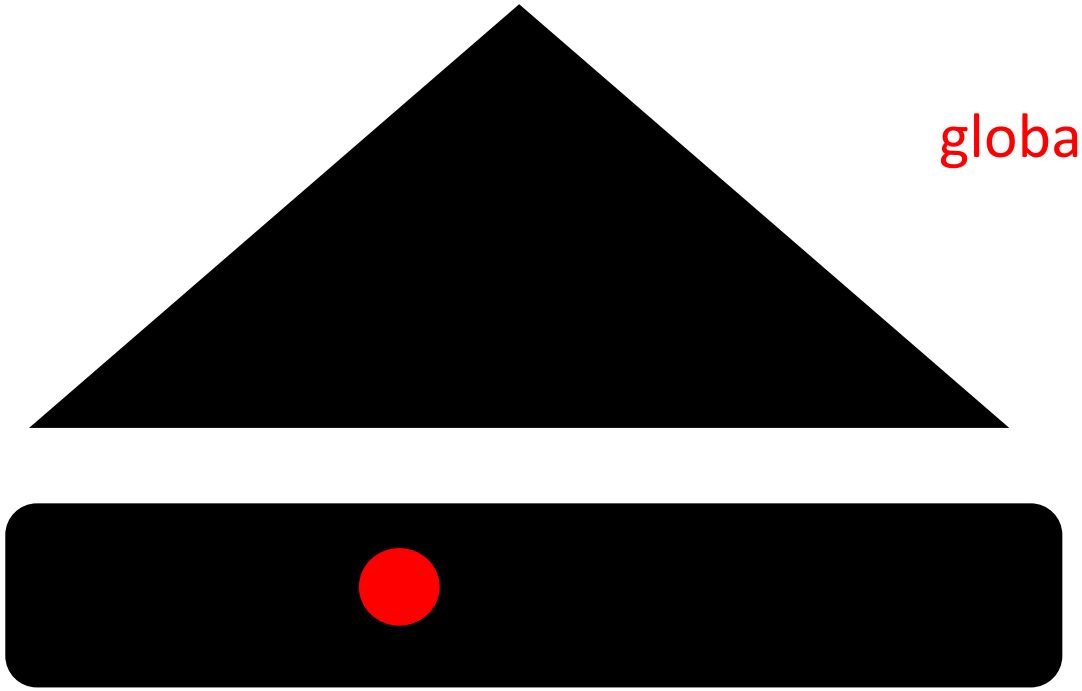
local data organization & search algorithm

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

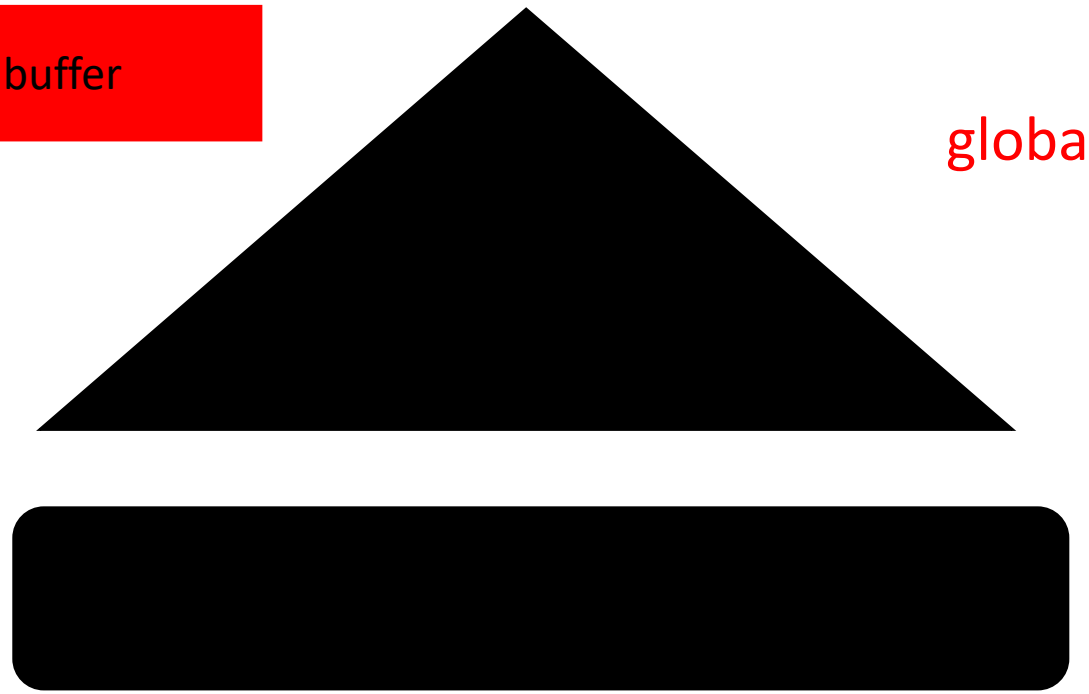
local data organization & search algorithm

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

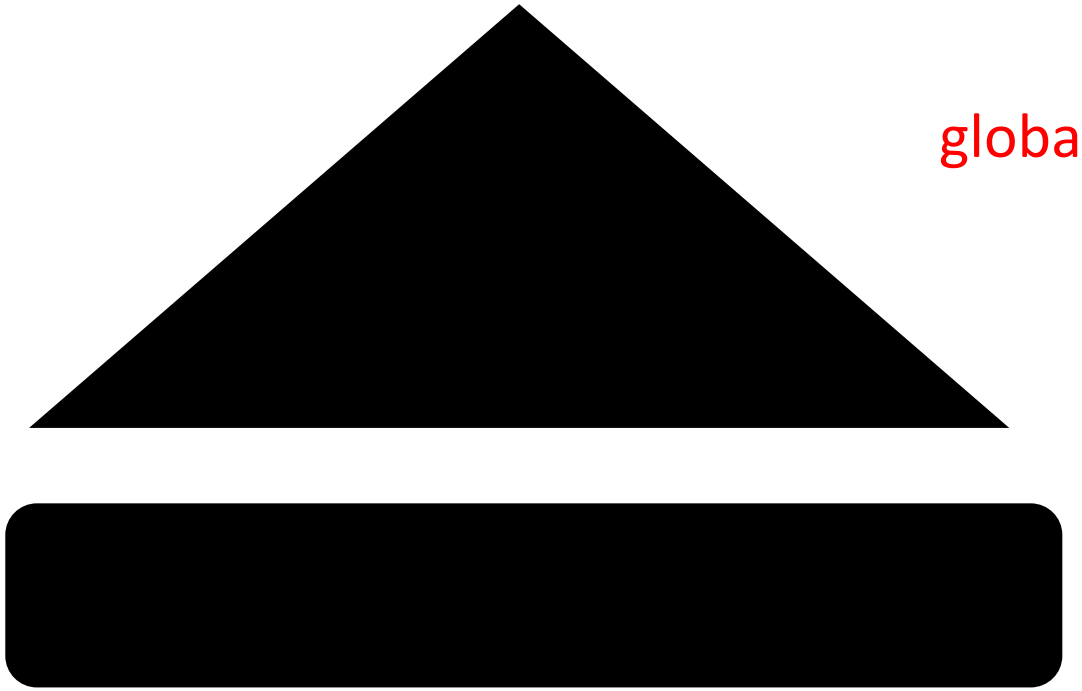
local data organization & search algorithm

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

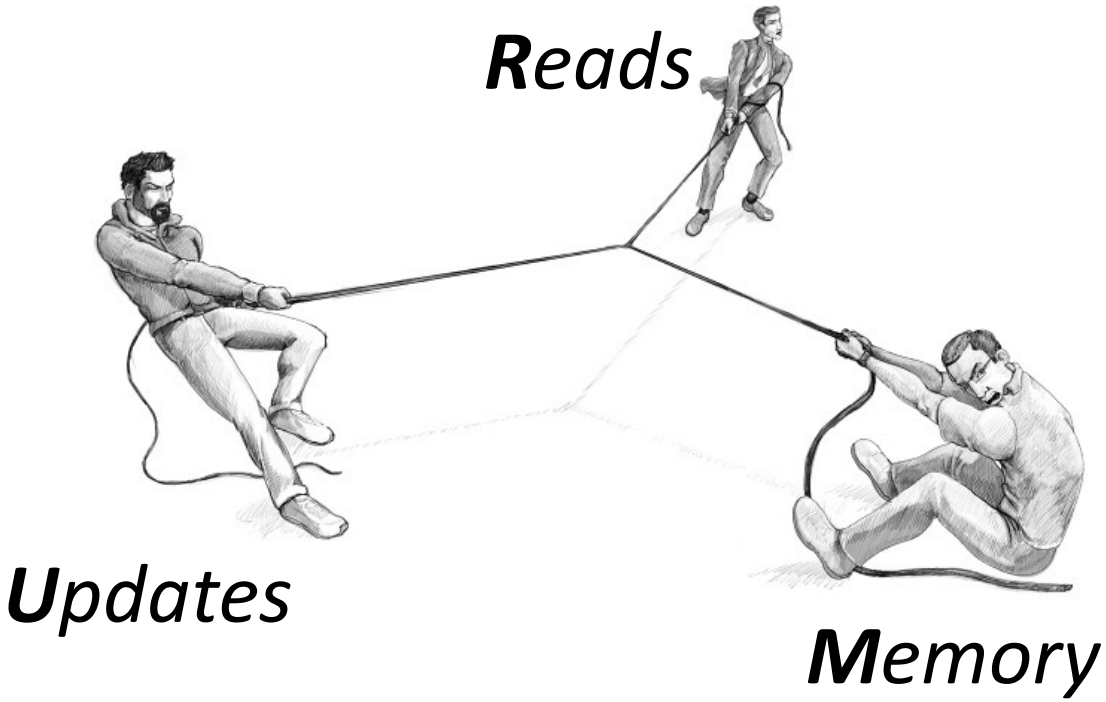
local data organization & search algorithm

modification policy

batching via buffering

adaptivity

data structure designs navigate a three-way tradeoff

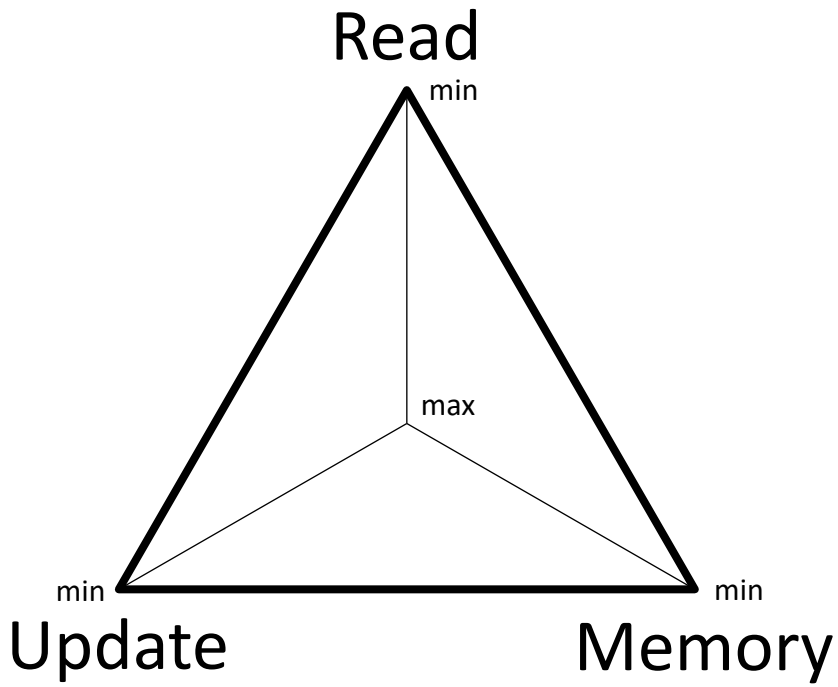


The RUM Conjecture

every access method has a (quantifiable)

- read overhead
- update overhead
- memory overhead

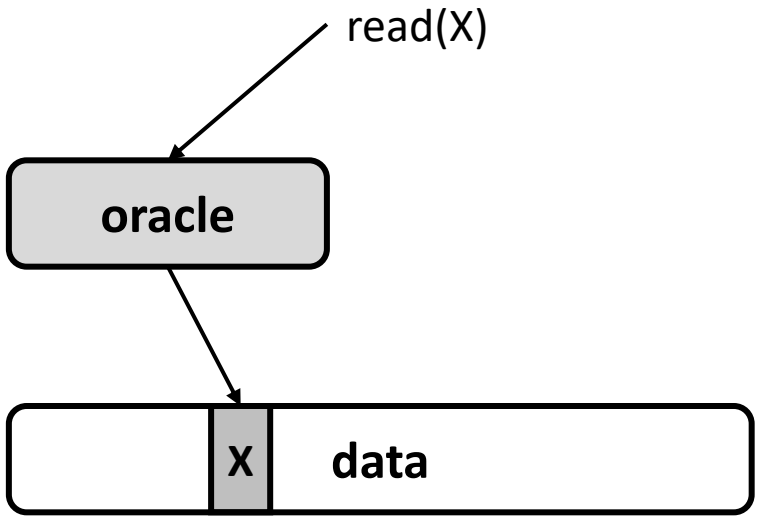
the three of which form a competing triangle



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

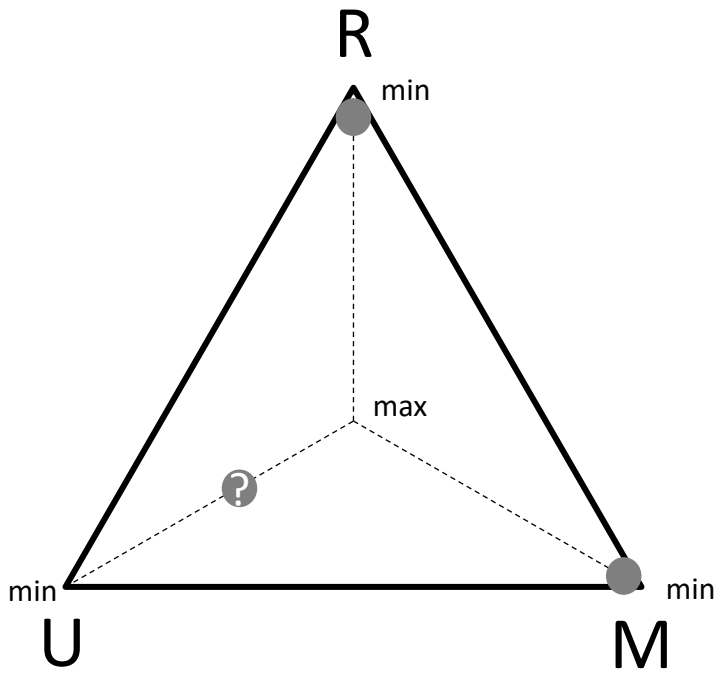
BOSTON UNIVERSITY

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

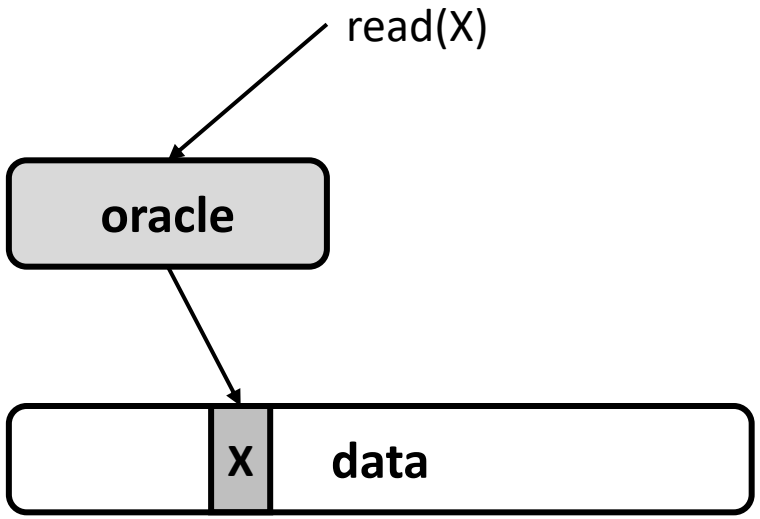


read(x) accesses only the bytes of object X

how *free* can an oracle be?

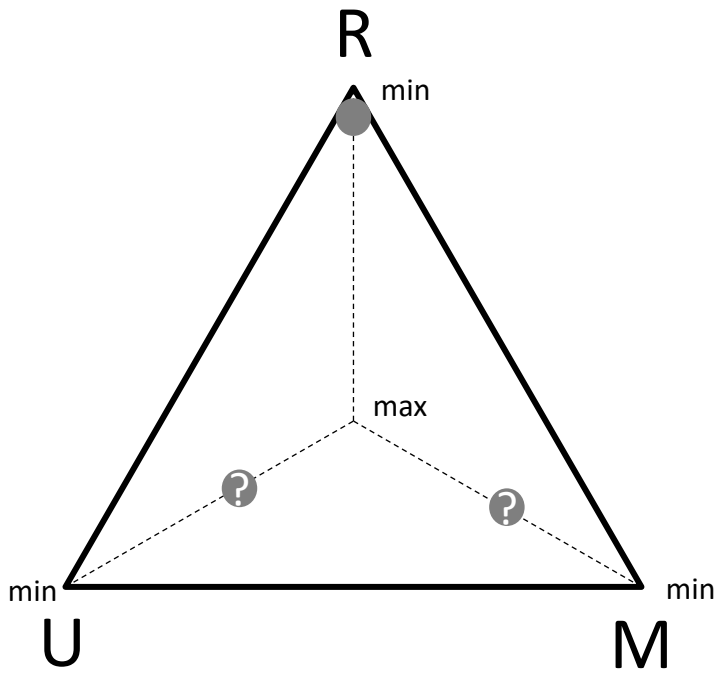


what would be an **optimal read** behavior?

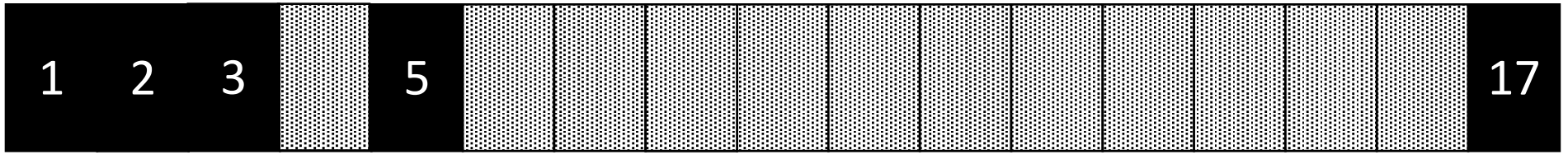


read(x) accesses only the bytes of object X

how *free* can an oracle be?



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



update 17 -> 3

minimum read overhead

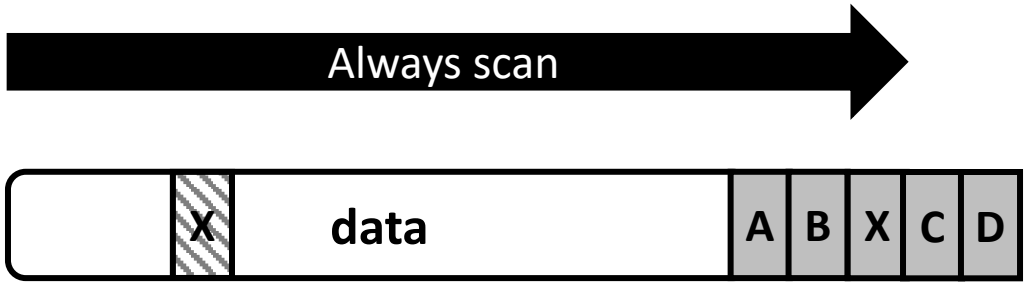
bound update overhead

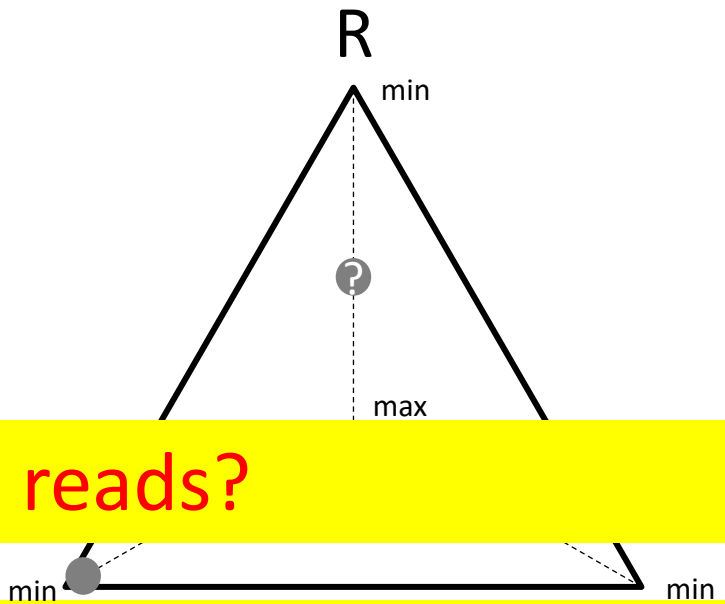
unbounded memory overhead

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

always *append*, and on update *invalidate*

update (X) changes the minimal number of bytes





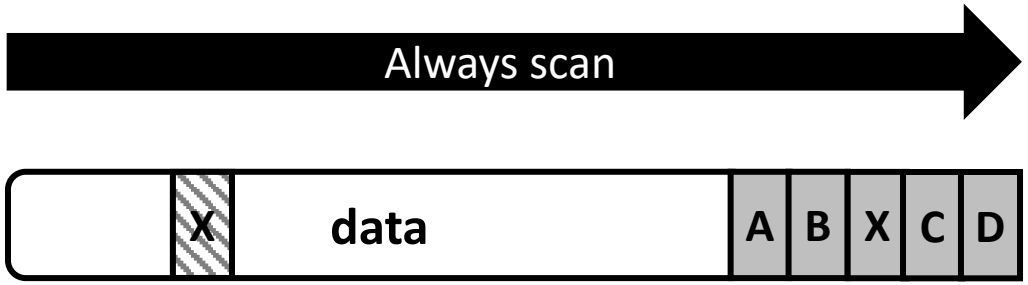
what about reads?

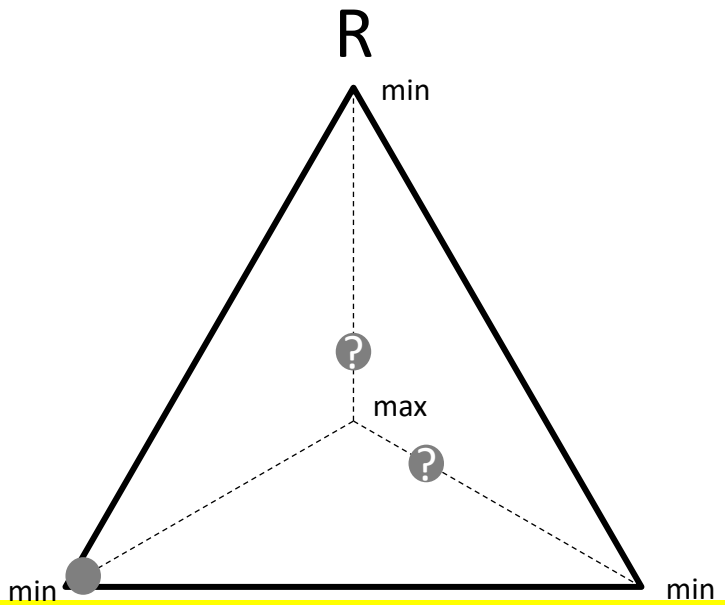
more data?

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

always *append*, and *invalidate* on update

update (X) changes the minimal number of bytes



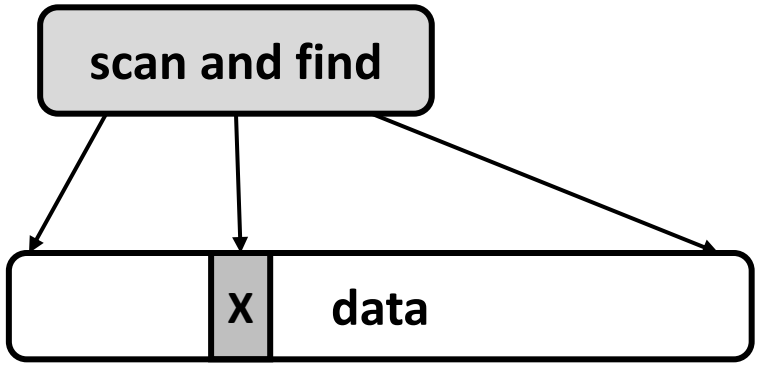


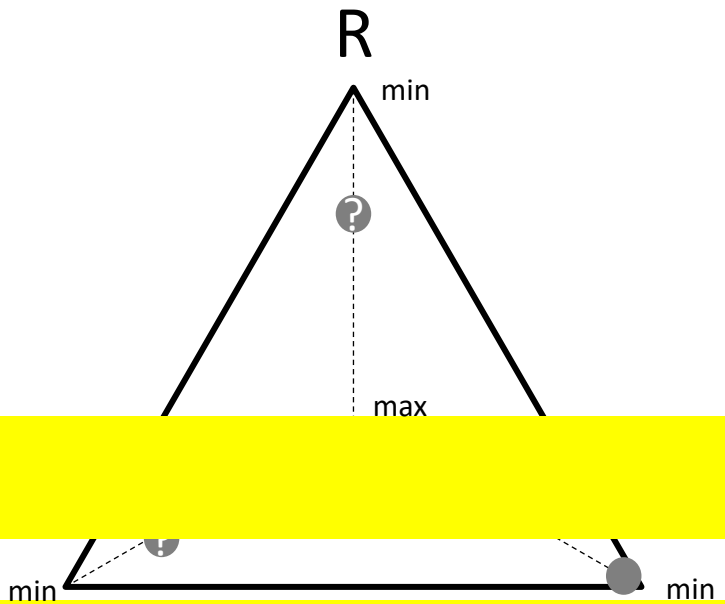
higher read and memory overhead

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

no metadata whatsoever, would result in the smallest memory footprint

scan and in-place updates

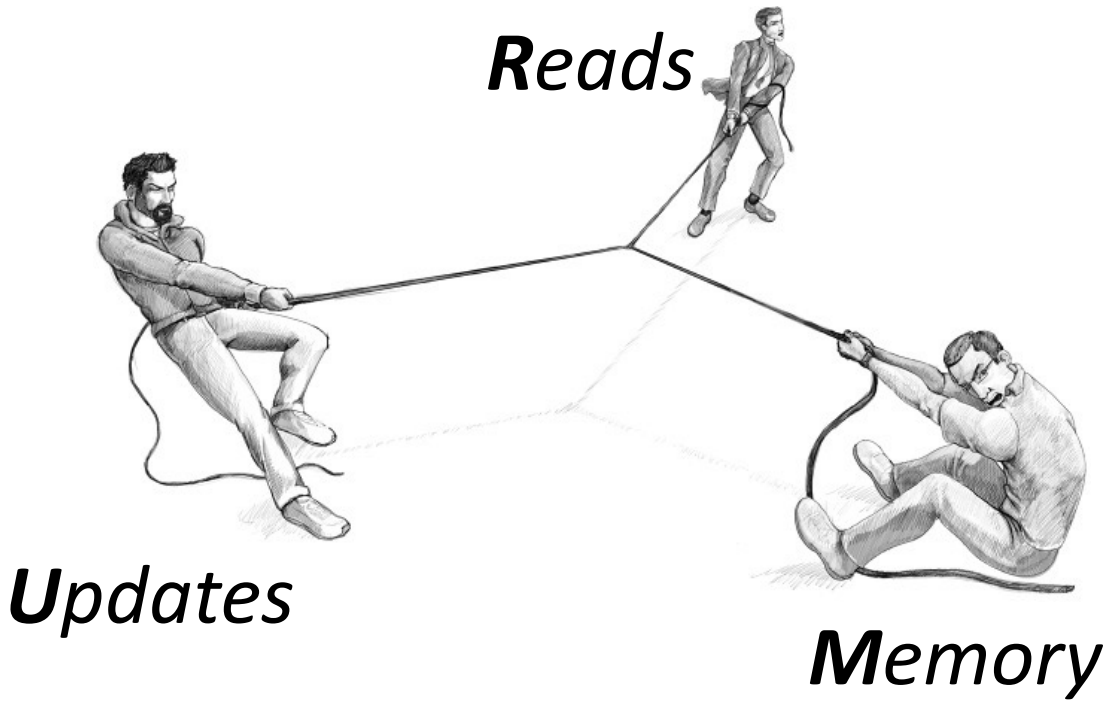




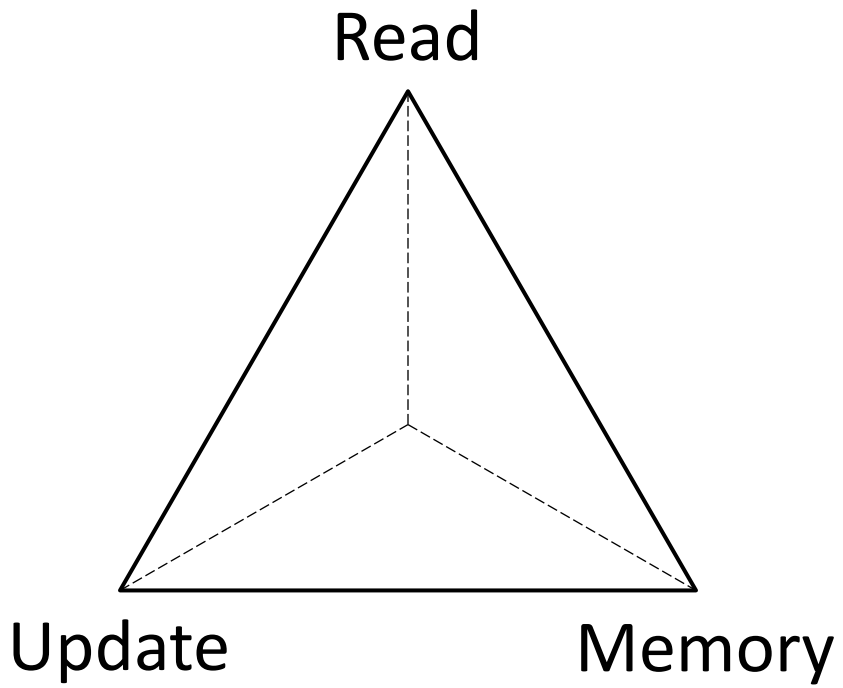
No!

do we need to reach the optimal(s)?

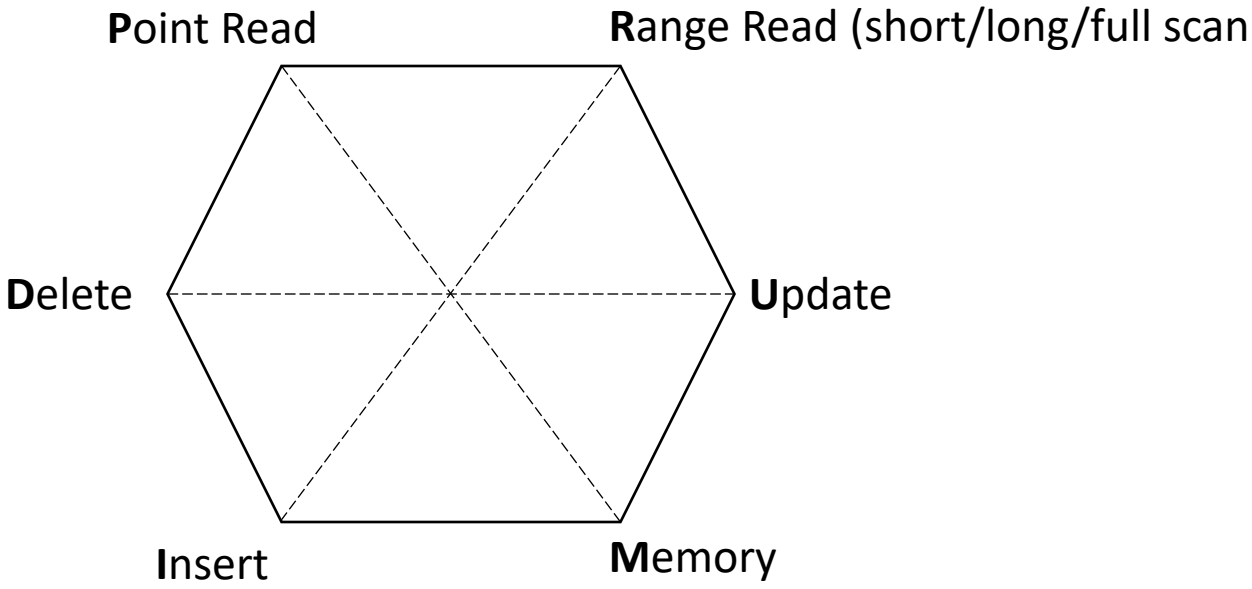
are there only three overheads?



are there only three overheads?



are there only three overheads?



PyRUMID overheads

data structures *design dimensions and their values*

global data organization

global search algorithm

metadata for searching

local data organization & search algorithm

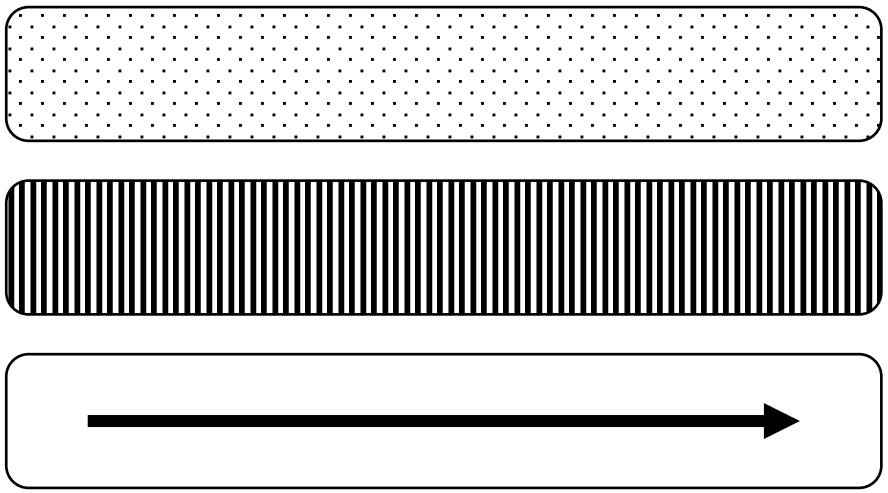
modification policy

batching via buffering

adaptivity

global data organization

key-level



unsorted

logging

sorted

another decision to be made for each partition

partition-level

range

1-5	6-10	11-15		

radix

001	010	011		

hash partitioning

h(x)=1	h(x)=2	h(x)=3		

partitioning logging

epoch 1	epoch 2	epoch 3		

global search algorithm

data organizations that can use it?

comments

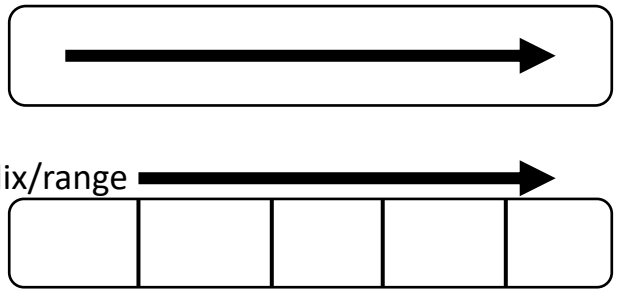
SCAN

any data organization

more suited for long range queries

---

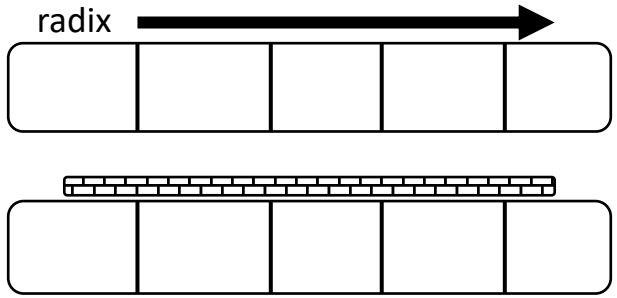
binary search



point or range queries

---

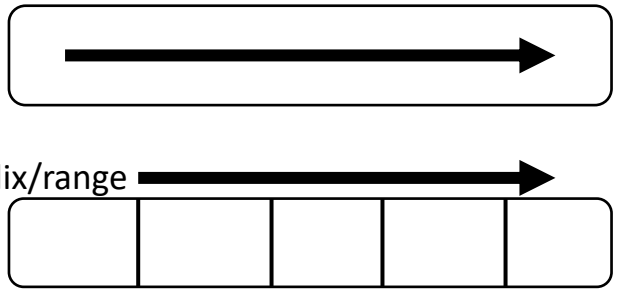
direct addressing



more suited for point queries

---

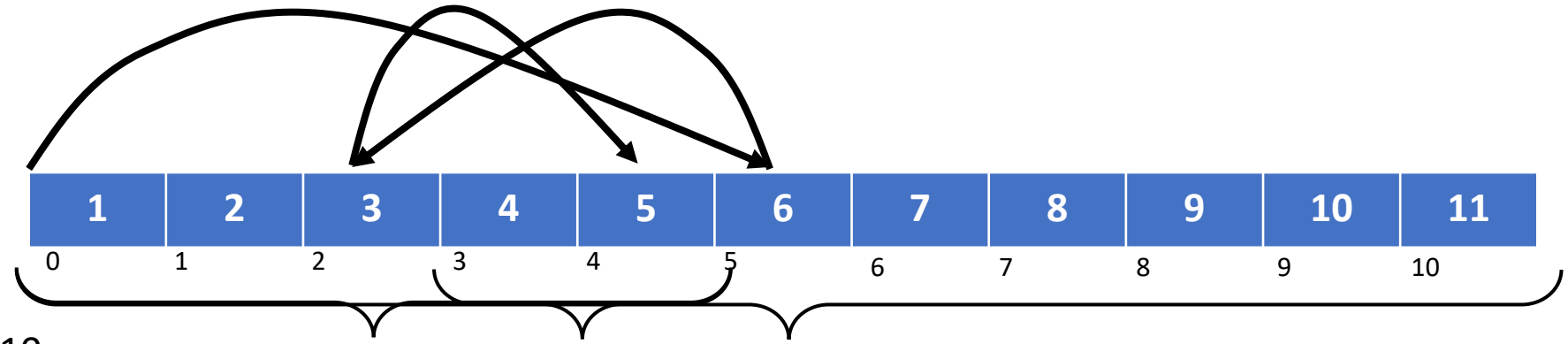
data-driven search



better match the data
example: **interpolation search**

Binary vs interpolation search

search for x=5



low = 0; high = 10;

mid = low + (high - low) / 2 = 5

val[mid] = val[5] = 6; so x < val[mid] ➔ high = mid - 1 = 4

low = 0; high = 4;

mid = low + (high - low) / 2 = 2

val[mid] = val[2] = 3; so x > val[mid] ➔ low = mid + 1 = 3

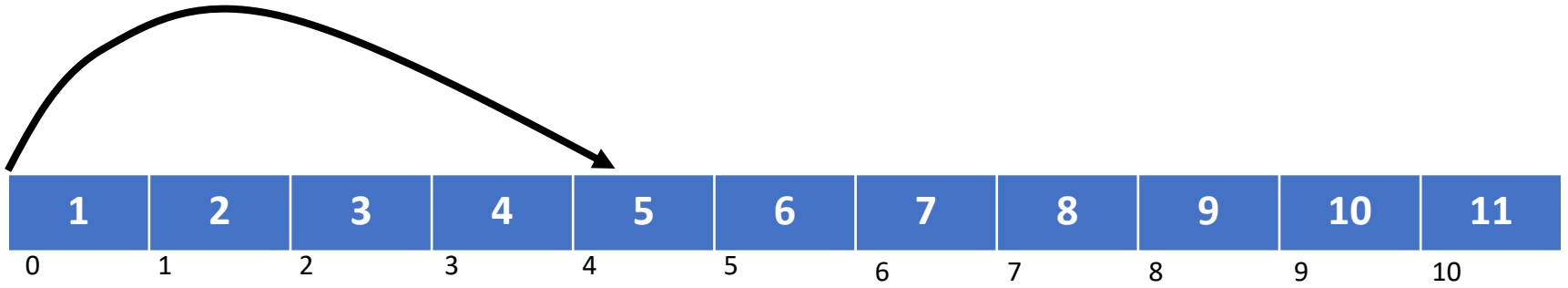
low = 3; high = 4;

mid = low + (high - low) / 2 = 3.5 (rounding to 4)

val[mid] = val[4] = 5; so x == val[mid] ➔ success!!

Binary vs **interpolation** search

search for x=5



low = 0; high = 10;

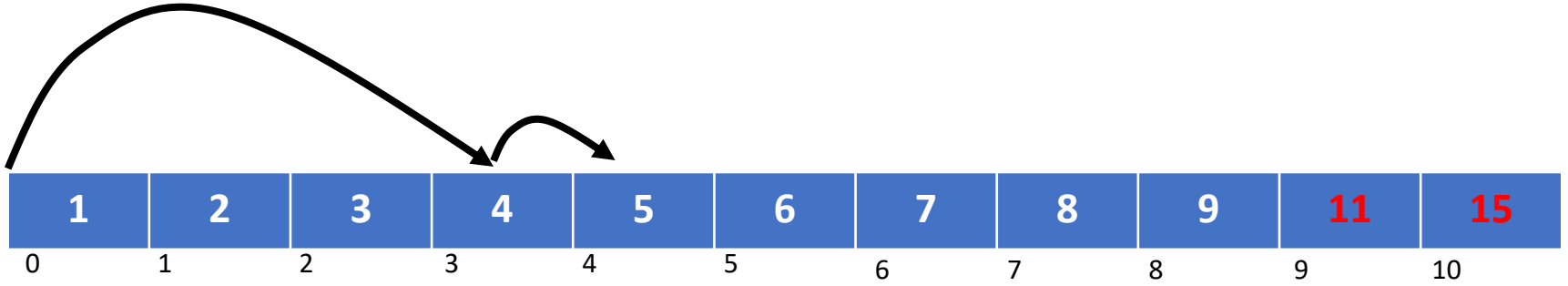
mid = low + ((x - val[low]) * (high - low) / (val[high] - val[low])) = (5-1)*(10-0)/(11-1) = 4

val[mid] = val[4] = 5 ➔ success!

does it always need 1 hop?

Binary vs interpolation search

search for x=5



low = 0; high = 10;

mid = low + ((x - val[low]) * (high - low) / (val[high] - val[low])) = (5-1)*(10-0)/(15-1) = (rounding to) 3

val[mid] = val[3] = 4 ; so x > val[mid] ➔ low = mid + 1 = 4

low = 4; high = 10;

mid = low + ((x - val[low]) * (high - low) / (val[high] - val[low])) = 4 + (5-5)*(10-4)/(15-5) = 4

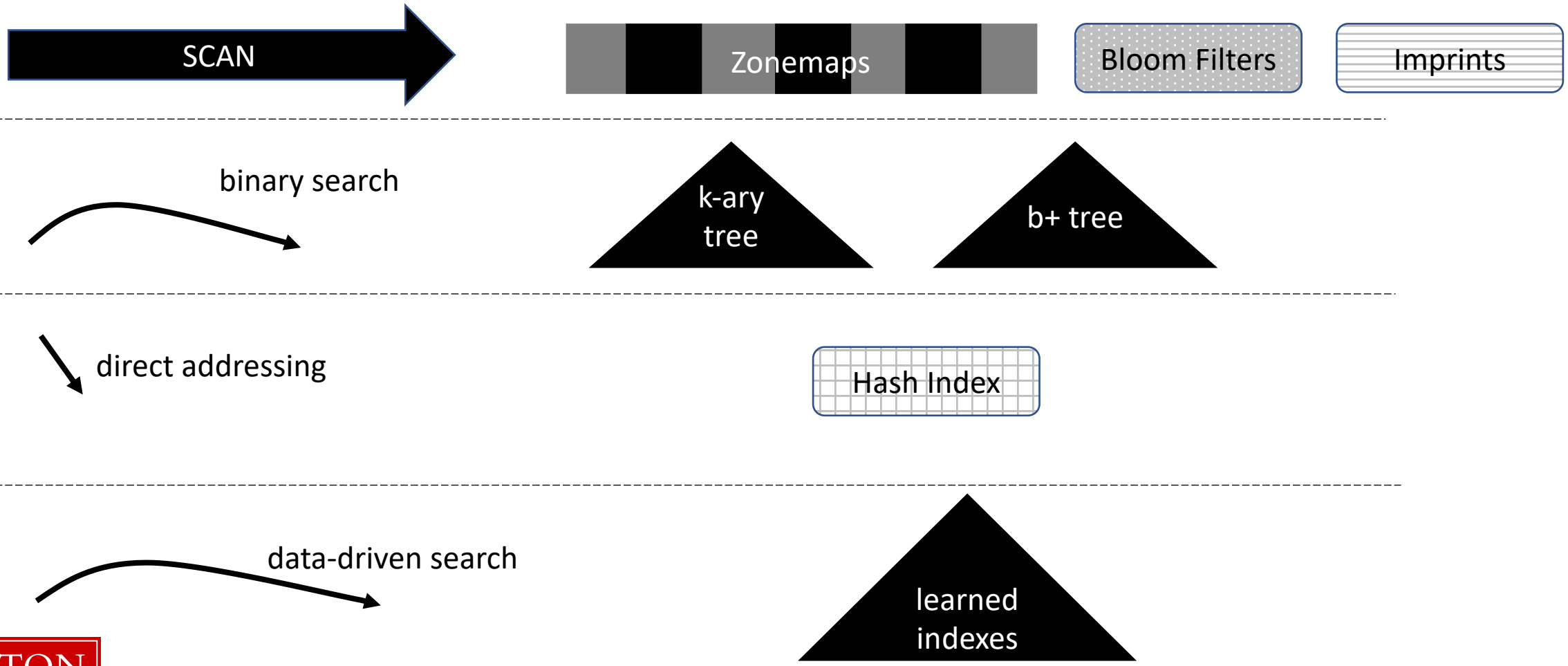
val[mid] = val[4] = 5 ➔ success!

still better than binary!

works well with uniform distribution

global search using metadata (indexing)

every search algorithm can be materialized and further optimized using indexing

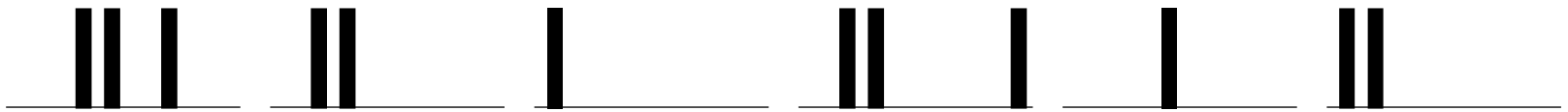


Imprints

similar to zonemaps

Z1: [32,72]	Z2: [13,45]	Z3: [1,10]	Z4: [21,99]	Z5: [28,35]	Z6: [5,12]

C1: [32,72]	C2: [13,45]	C3: [1,10]	C4: [21,99]	C5: [28,35]	C6: [5,12]

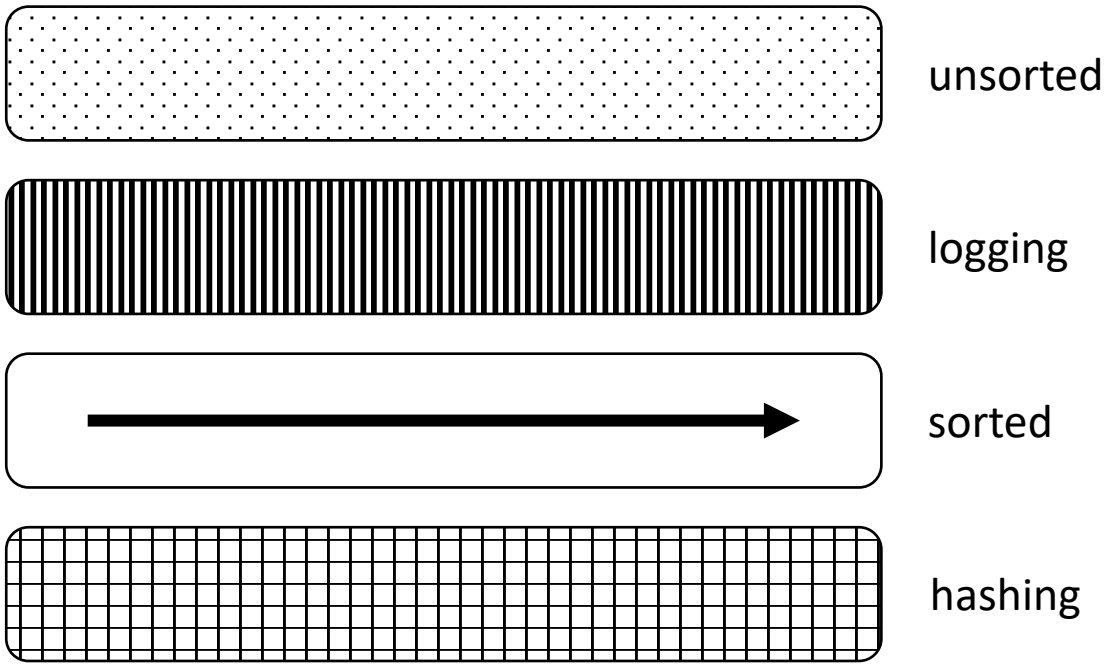


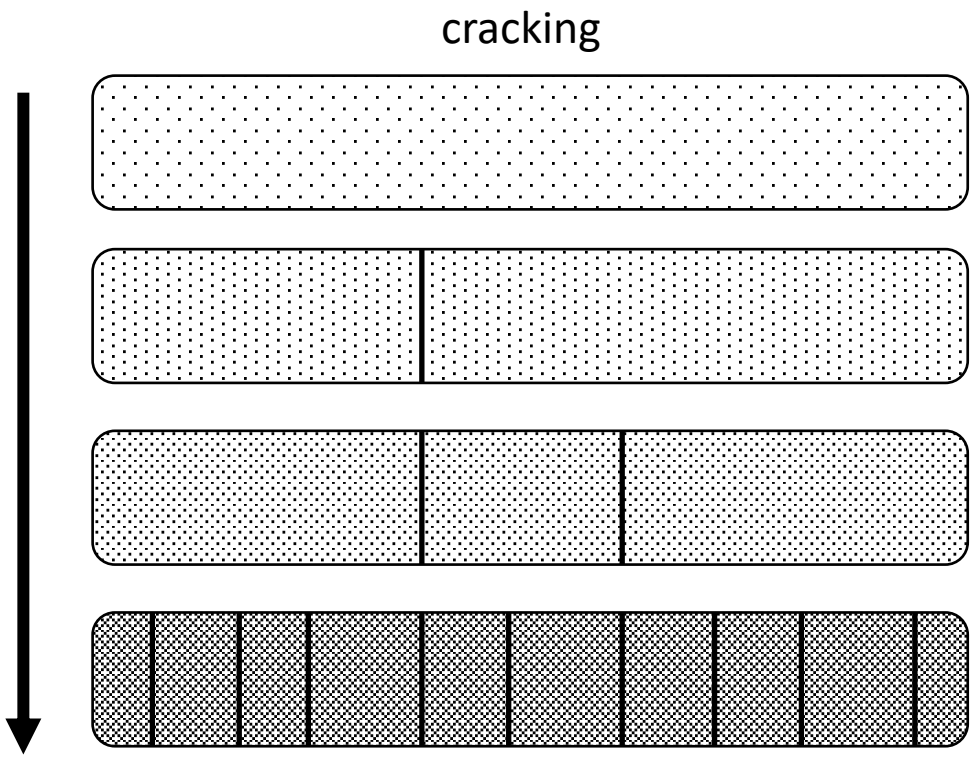
storing a simplified histogram for each block

why?  **it can capture better range queries and avoid useless overlap**

local data organization

decision per partition

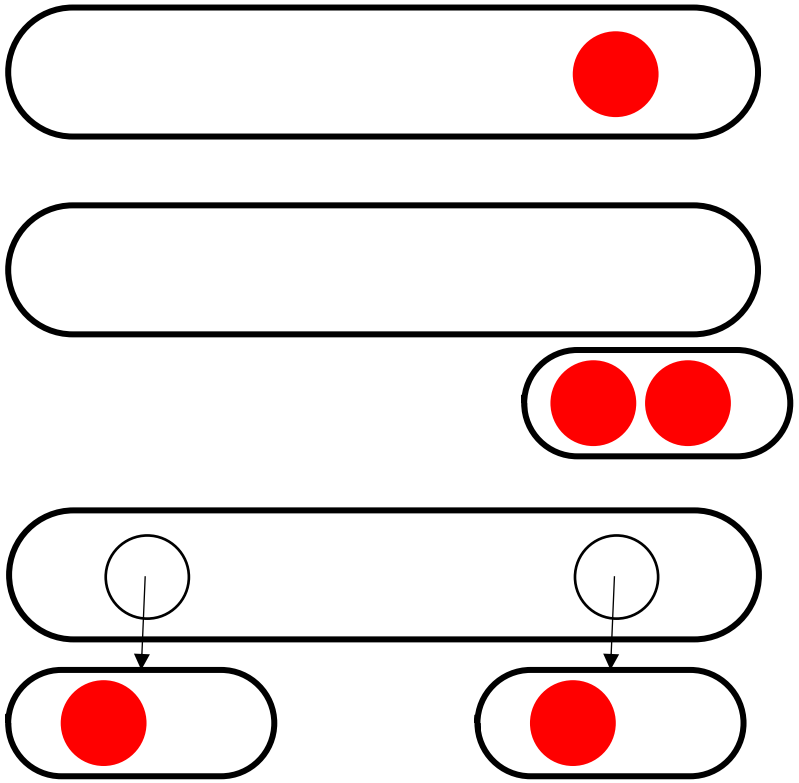




gradually from unsorted towards sorted

local search algorithms

modification policy (updates/deletes/inserts)



in-place	every update needs to find the “correct” position
out-of-place	every read needs to search all data
deferred in-place	will eventually merge

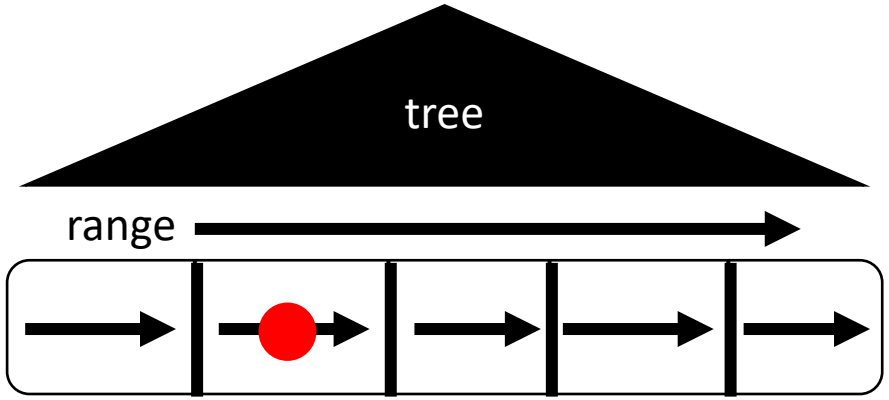
how to break down ***popular designs***
to those design decisions?

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, modifications, and some scans

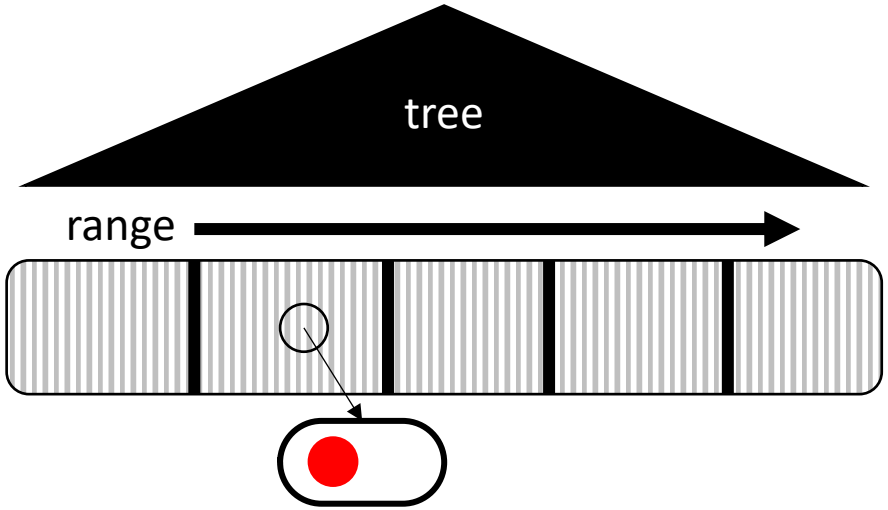


insert optimized b+ trees

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

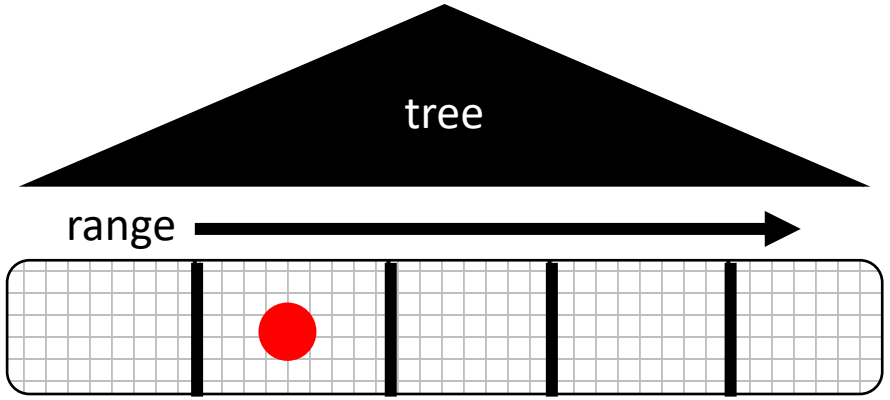


bounded disorder access method

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

Workload? 

mixed workload, without short range queries



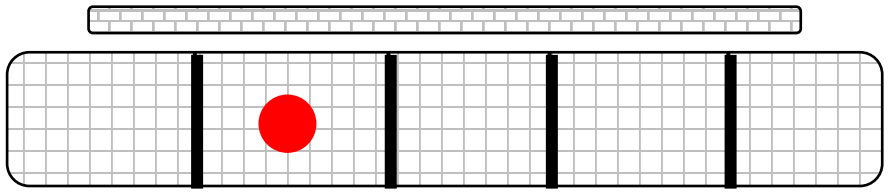
BOSTON UNIVERSITY

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

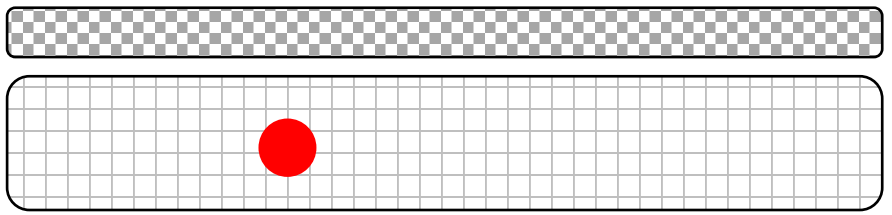


scans with zonemaps

global data organization	**none / logging**
global searching (algorithm or index)	**scan (with filters)**
local data organization	**n/a**
local search algorithm	**n/a**
modification policy	**in-place**

Workload? 

long range queries and modifications

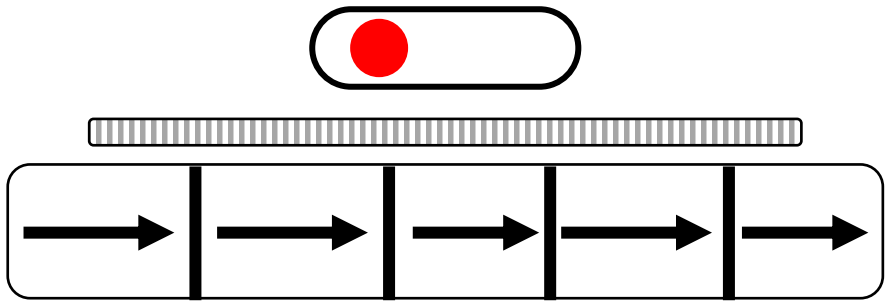


lsm-trees

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



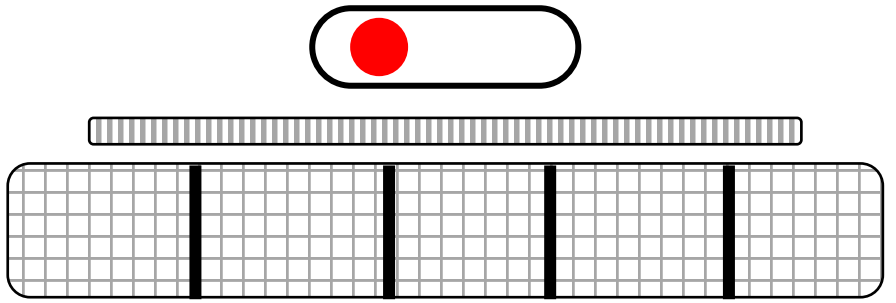
BOSTON UNIVERSITY

lsm-hash

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

Workload? 

modification-heavy with point queries and no range queries



BOSTON UNIVERSITY

BOSTON UNIVERSITY

CS 561: Data Systems Architectures

The *design space* of data structures

Prof. Manos Athanassoulis

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