

CS 6530: Advanced Database Systems Fall 2022

Lecture 05

The Design Space of Data Structures

Prashant Pandey

prashant.pandey@utah.edu

Acknowledgement: Slides taken from Prof. Manos Athanassoulis, BU

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

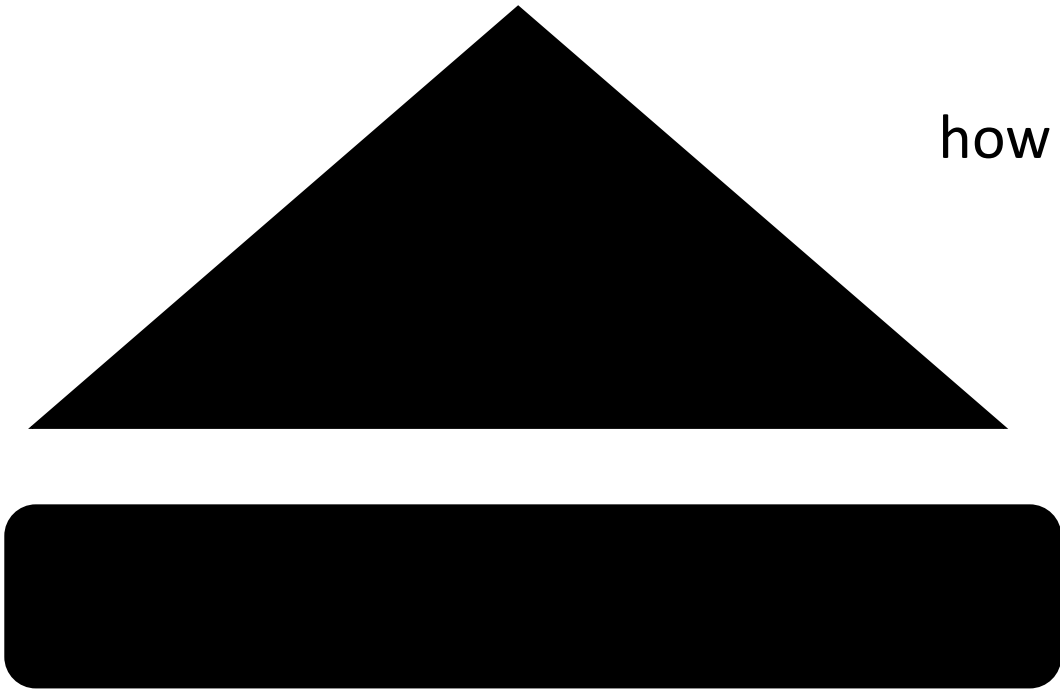


hardware (memory/storage/network/compute)

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?

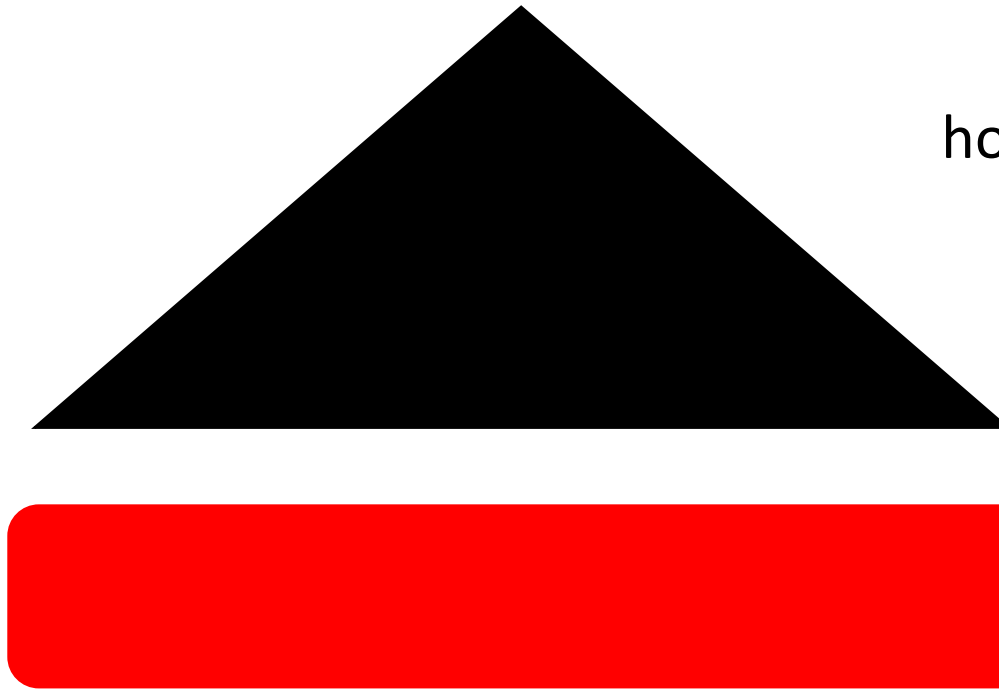
can I accelerate search through metadata?

multiple levels of nested organization?

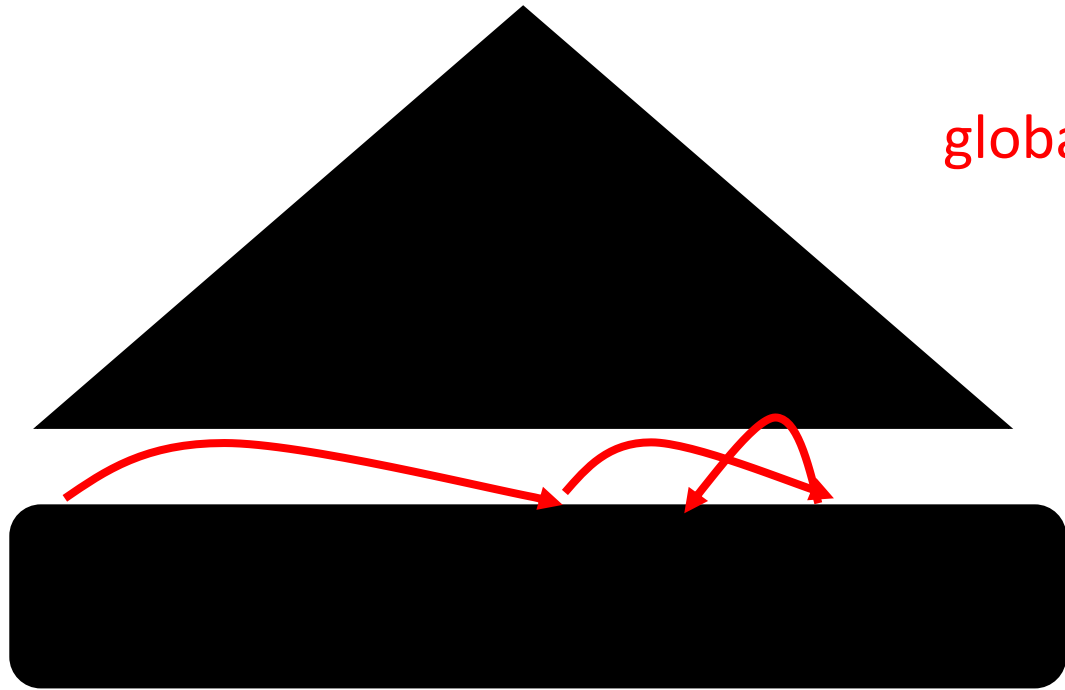
how to update or add new data?

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?

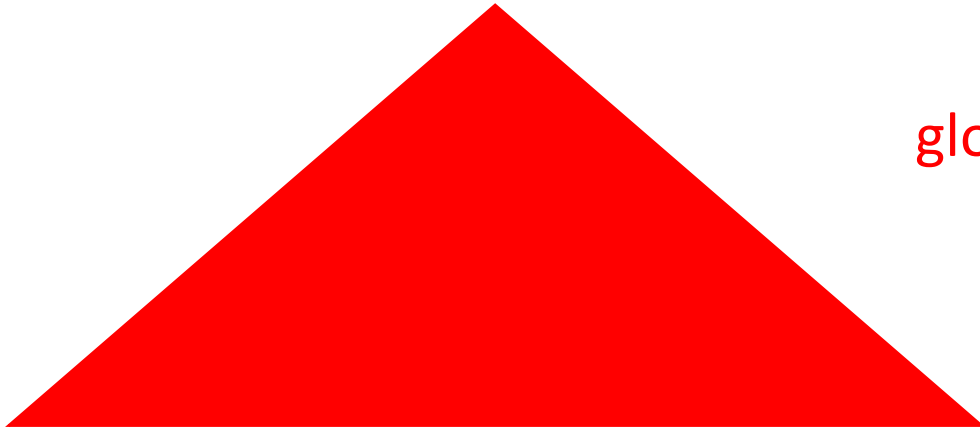
multiple levels of nested organization?

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

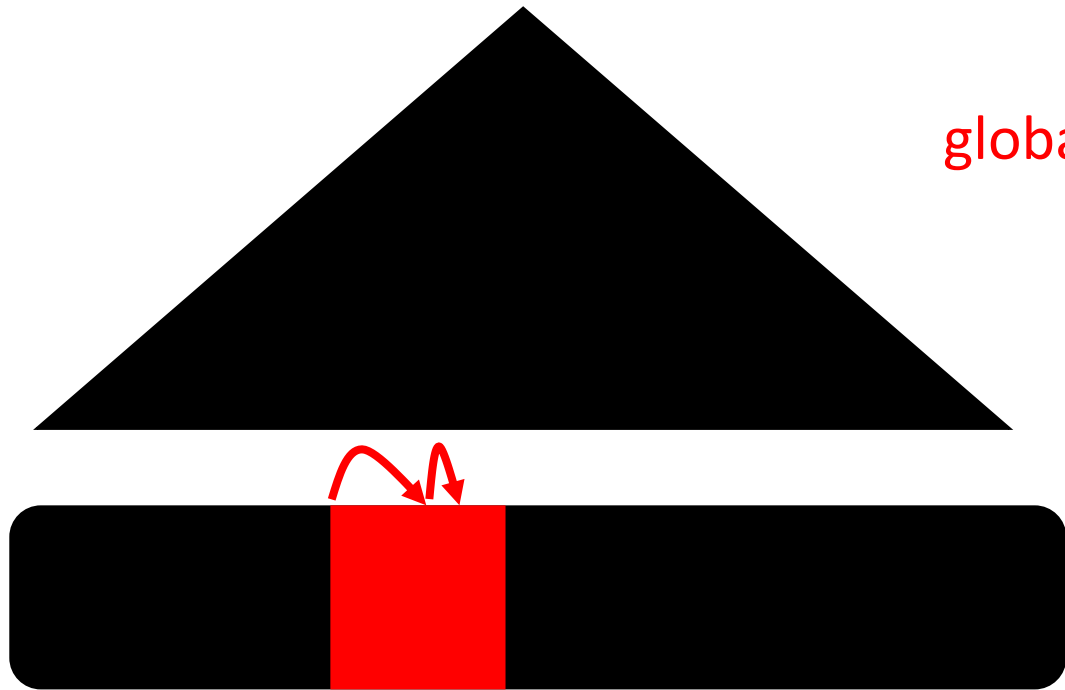
multiple levels of nested organization?

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

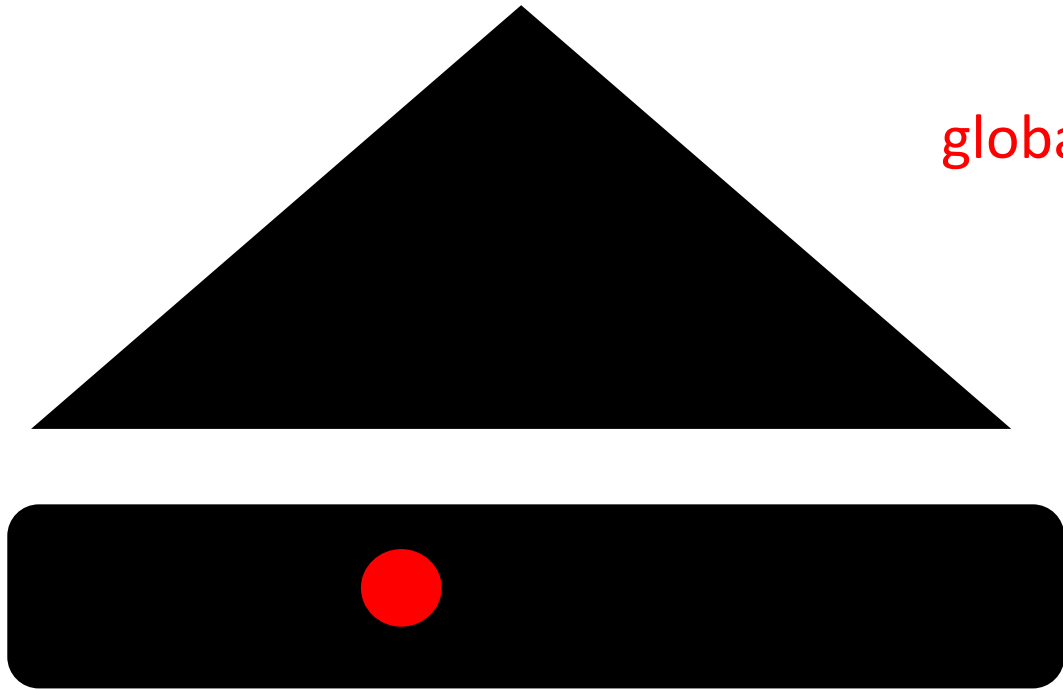
local data organization & search algorithm

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

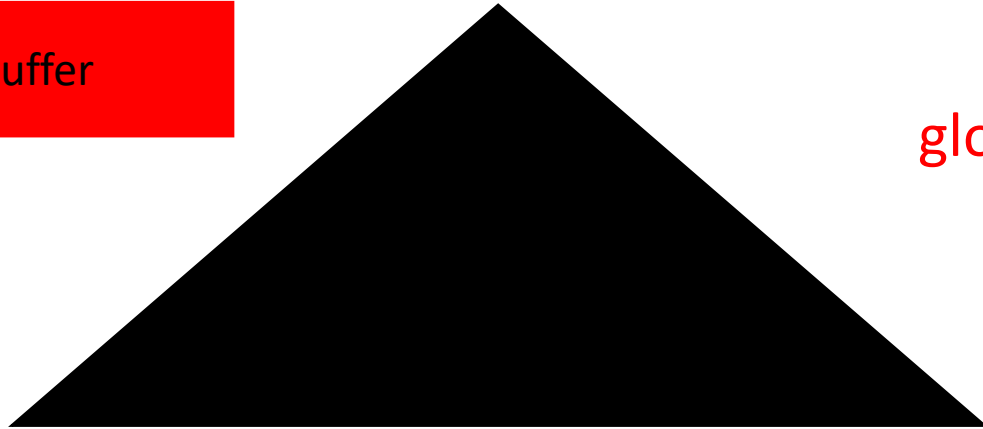
how to exploit additional memory/storage?

should the above decisions be applied eagerly or lazily?

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



buffer



global data organization

global search algorithm

metadata for searching

local data organization & search algorithm

modification policy

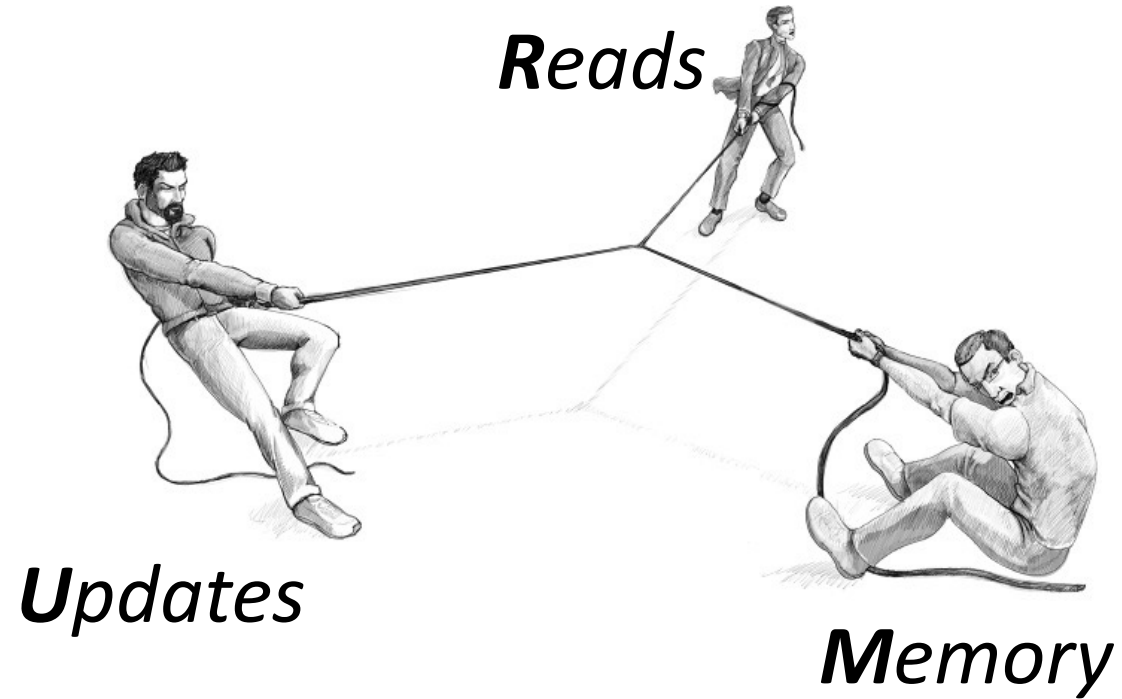
batching via buffering

should the above decisions be applied eagerly or lazily?

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



data structure designs navigate a three-way tradeoff





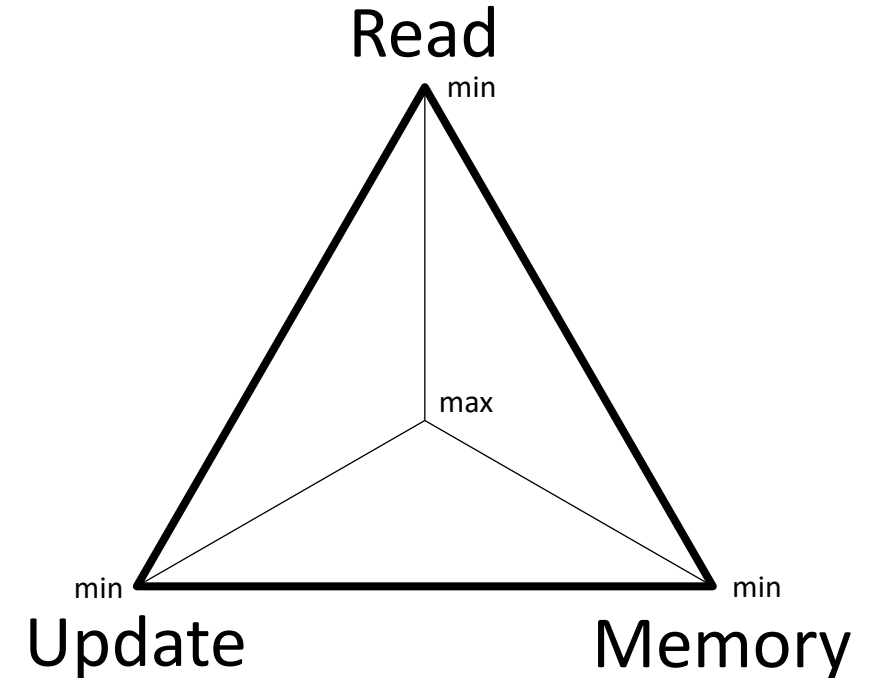
The RUM Conjecture

every access method has a (quantifiable)

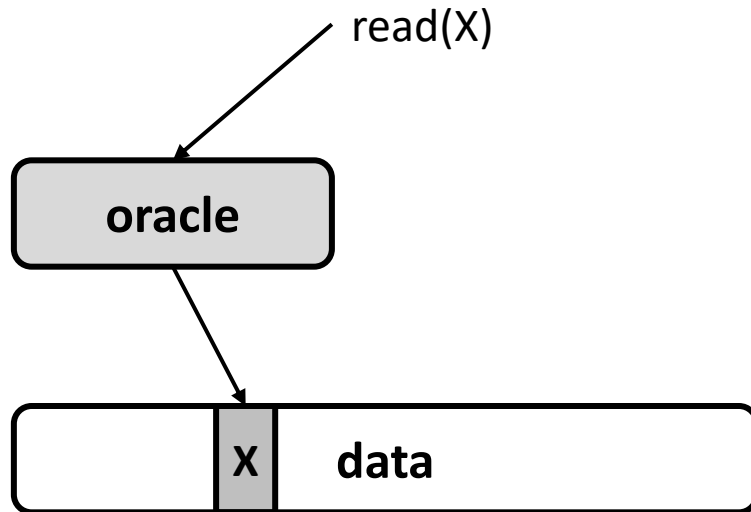
- read overhead
- update overhead
- memory overhead

the three of which form a competing triangle

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

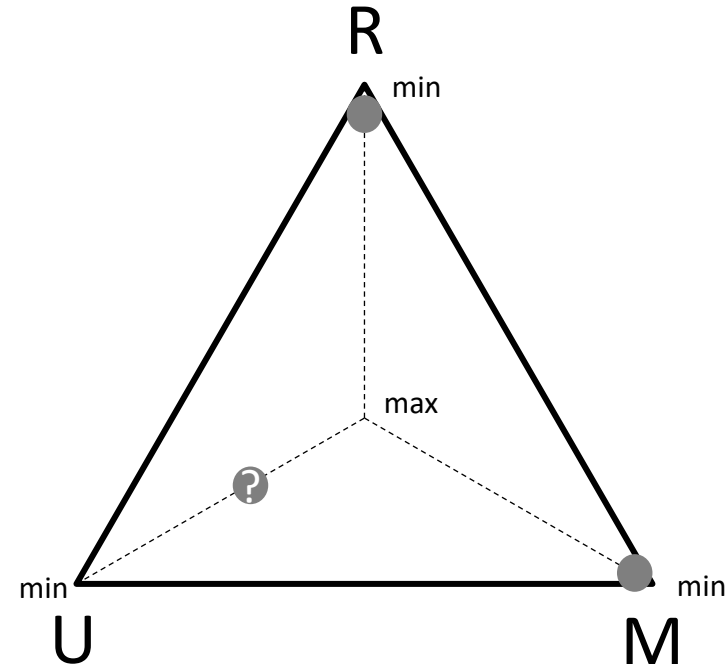


what would be an optimal read behavior?

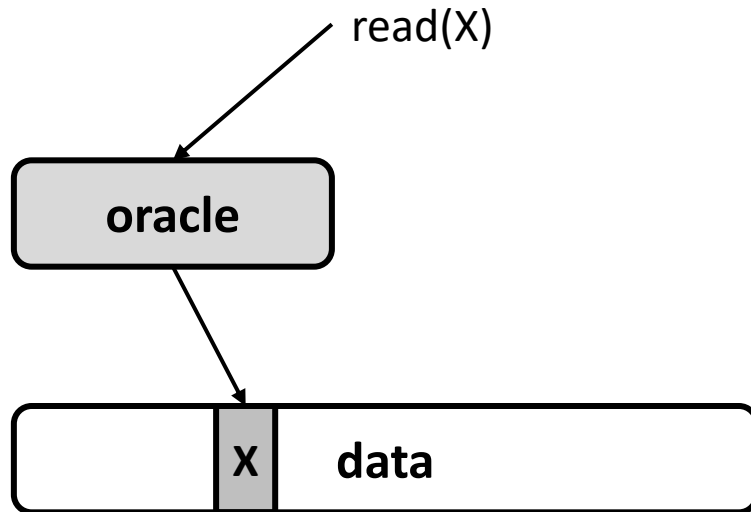


$read(x)$ accesses only the bytes of object X

how *free* can an oracle be?

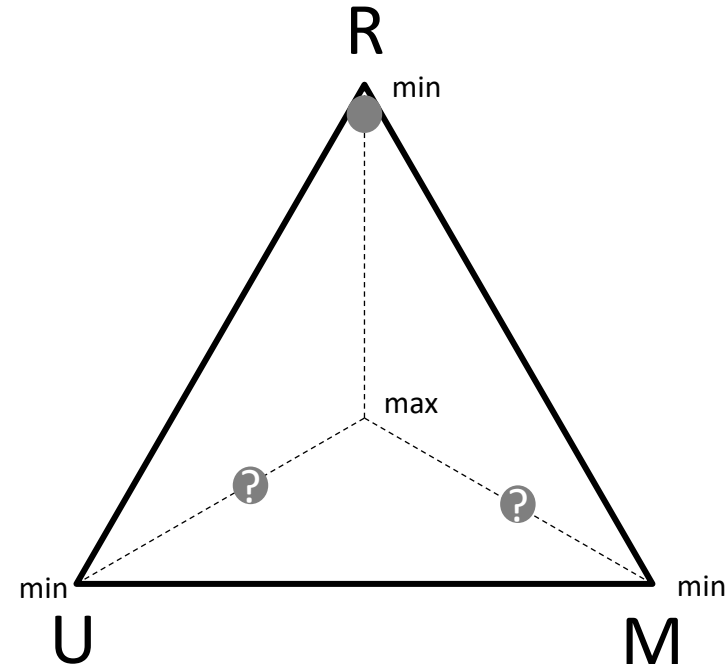


what would be an optimal read behavior?

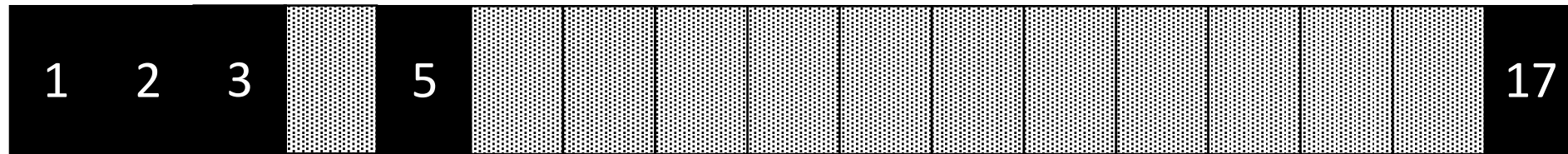


$read(x)$ accesses only the bytes of object X

how *free* can an oracle be?



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



~~update 17~~ -> 3

minimum read overhead

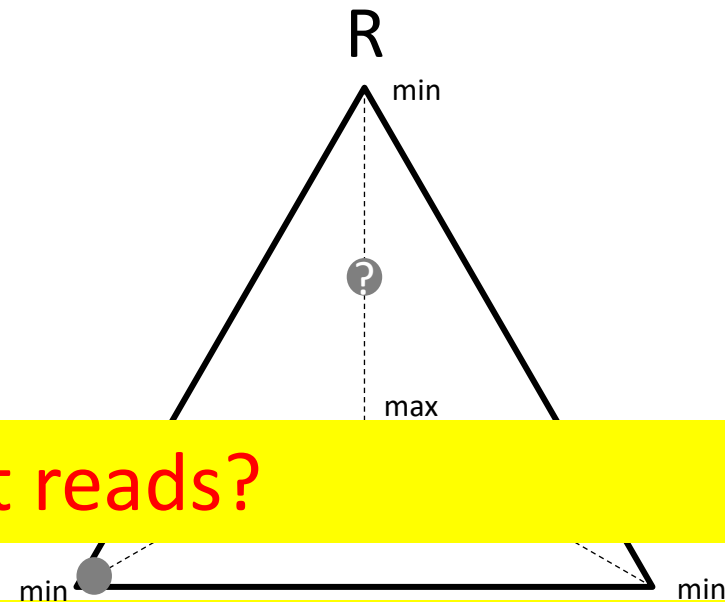
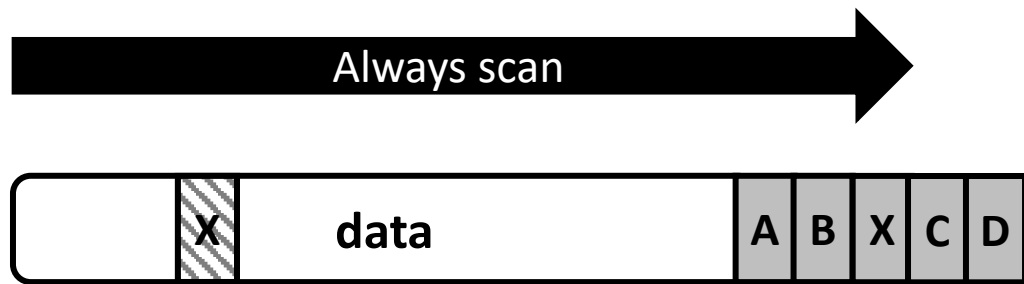
bound update overhead

unbounded memory overhead

what would be an optimal update behavior?

always *append*, and on update *invalidate*

update (X) changes the minimal number of bytes



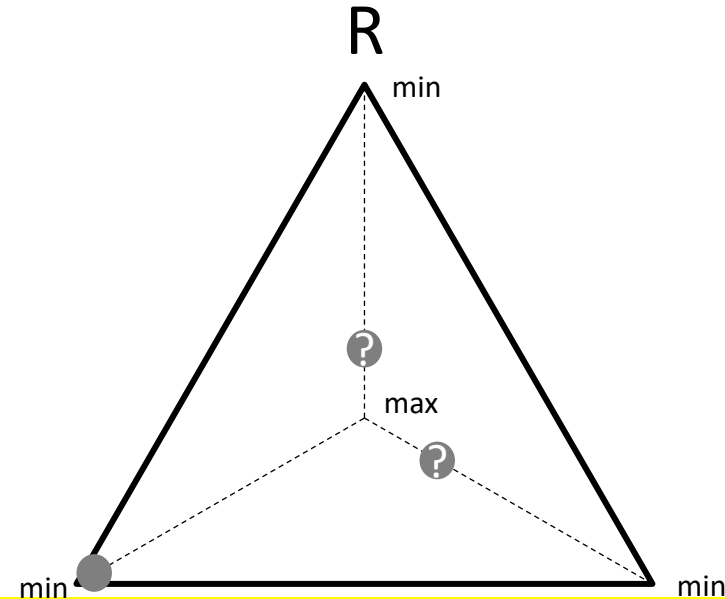
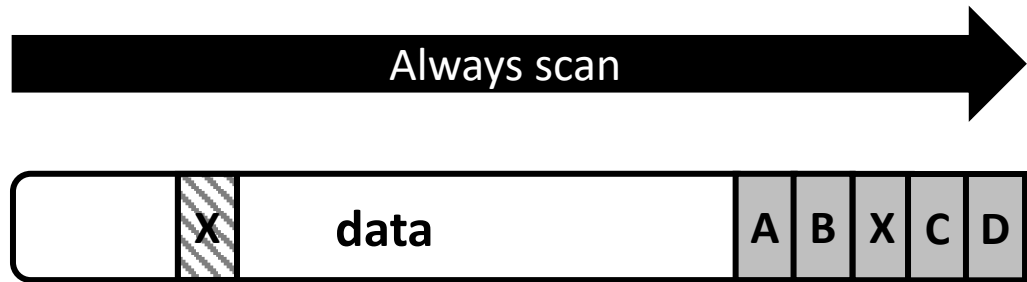
what about reads?

more data?

what would be an optimal update behavior?

always *append*, and *invalidate* on update

update (X) changes the minimal number of bytes

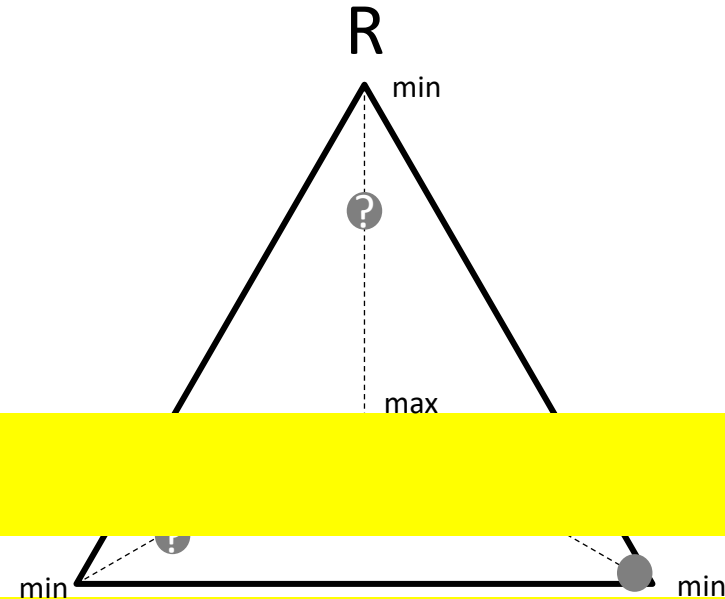
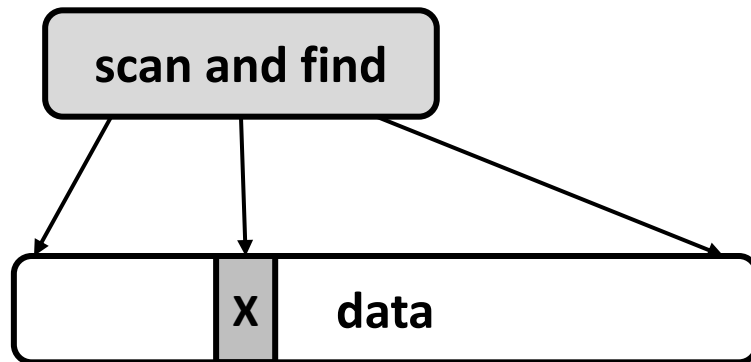


higher read and memory overhead

what would be an optimal memory overhead?

no metadata whatsoever, would result in the smallest memory footprint

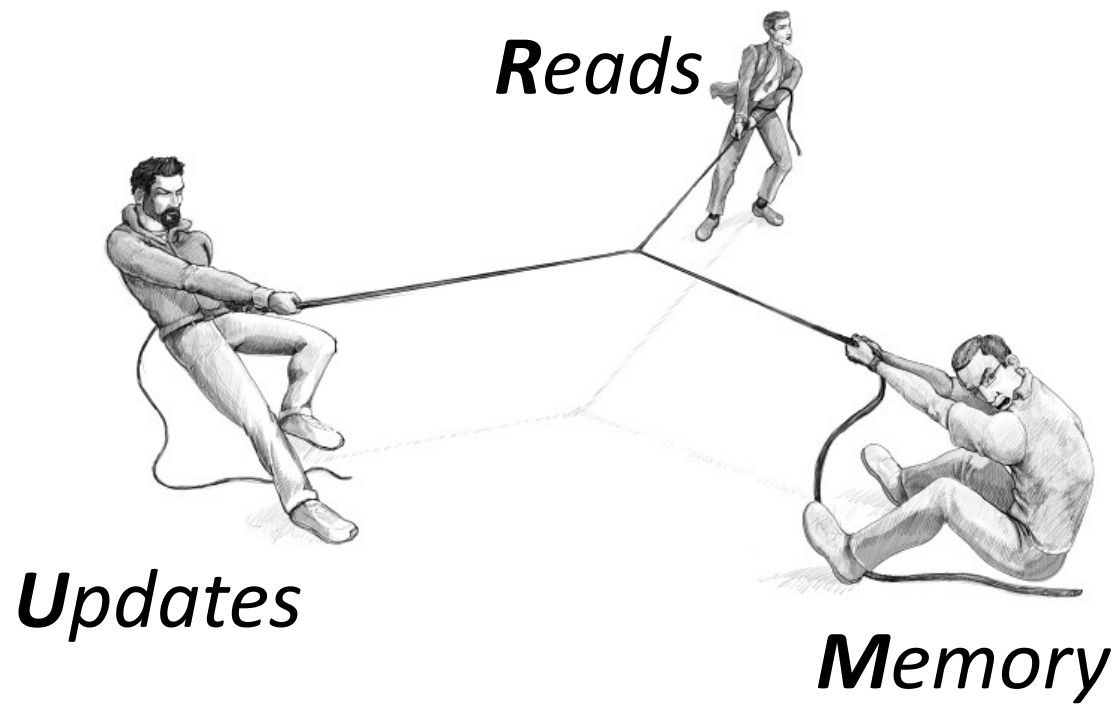
scan and in-place updates



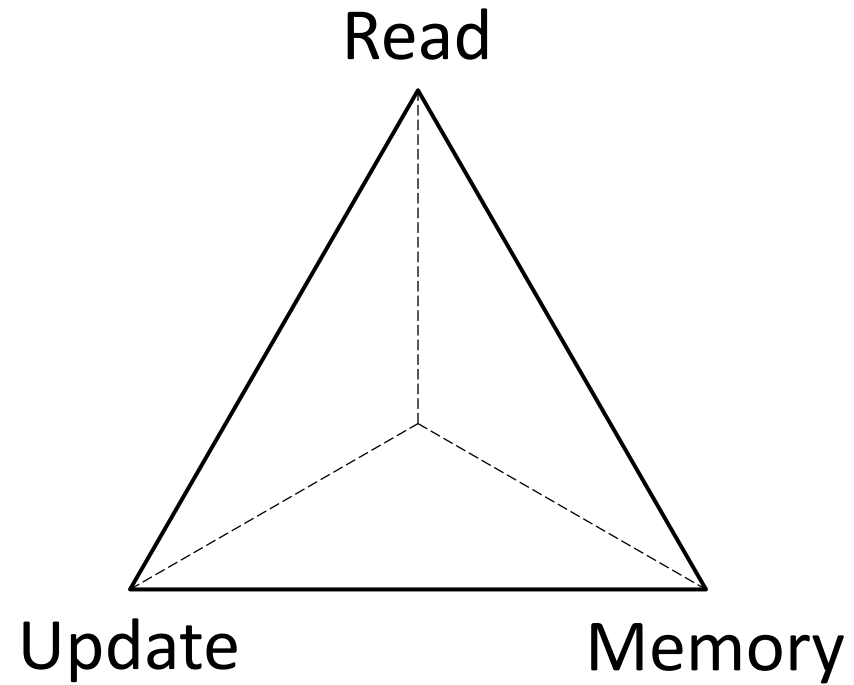
No!

do we need to reach the optimal(s)?

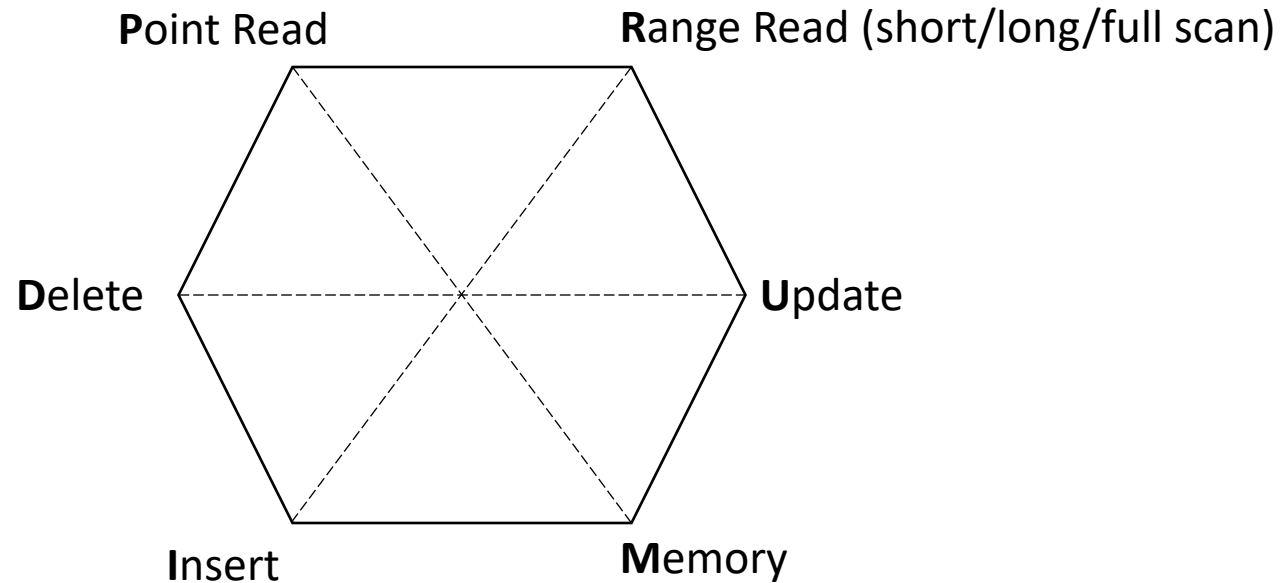
are there only three overheads?



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

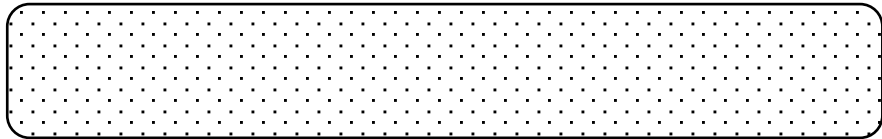
modification policy

batching via buffering

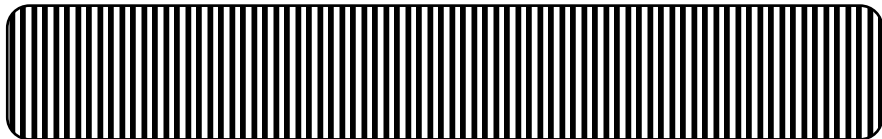
adaptivity

global data organization

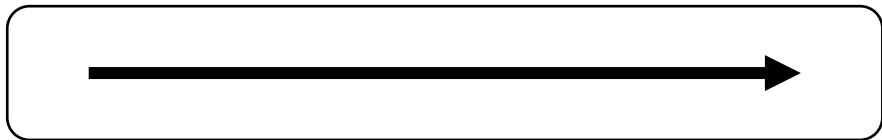
key-level



unsorted



logging



sorted

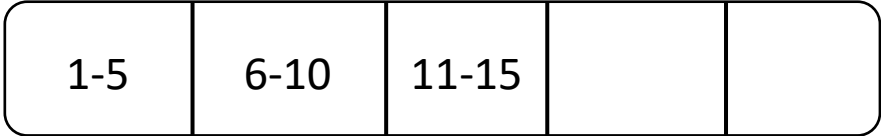
another decision to be made for each partition

hash partitioning

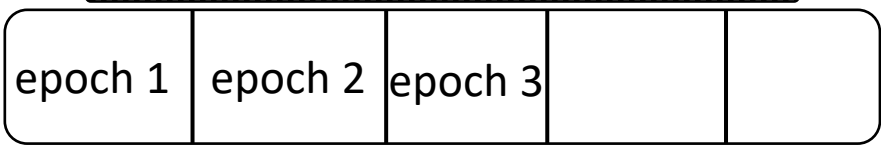
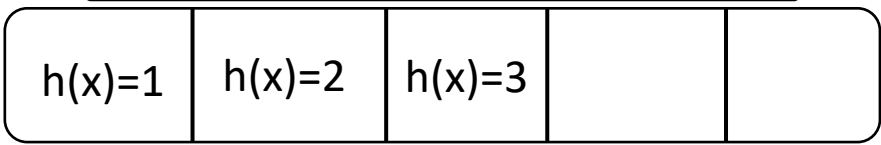
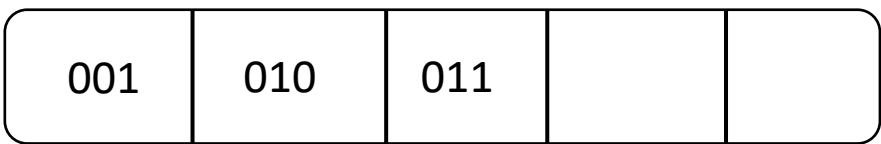
partitioning logging

partition-level

range



radix



global search algorithm

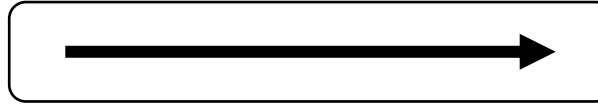
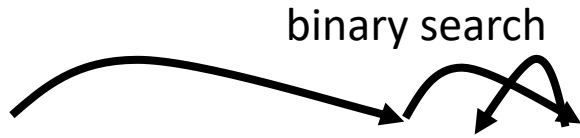


data organizations that can use it?

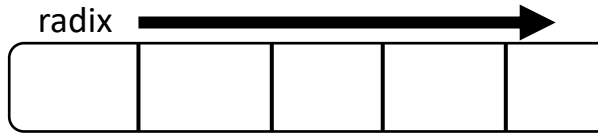
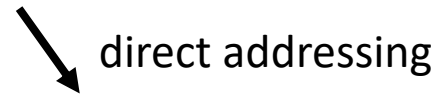
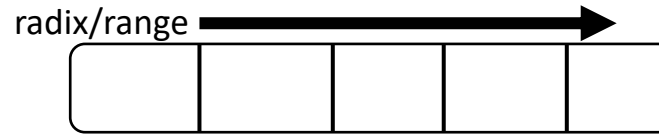
comments



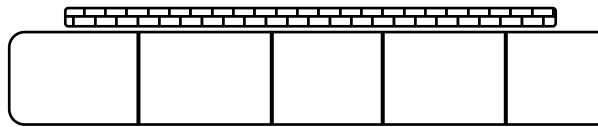
more suited for long range queries



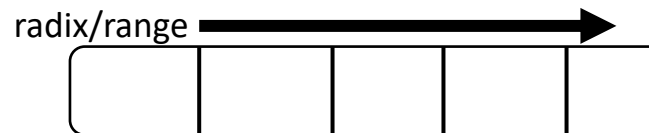
point or range queries



more suited for point queries

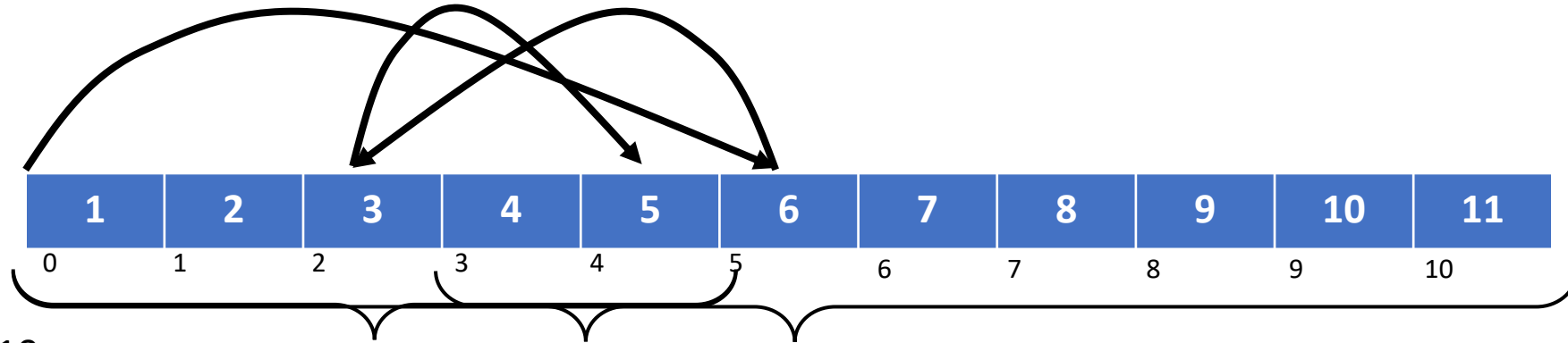


better match the data
example: **interpolation search**



Binary vs interpolation search

search for x=5



low = 0; high = 10;

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

val[mid] = val[5] = 6; so $x < \text{val}[\text{mid}] \rightarrow \text{high} = \text{mid} - 1 = 4$

low = 0; high = 4;

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

val[mid] = val[2] = 3; so $x > \text{val}[\text{mid}] \rightarrow \text{low} = \text{mid} + 1 = 3$

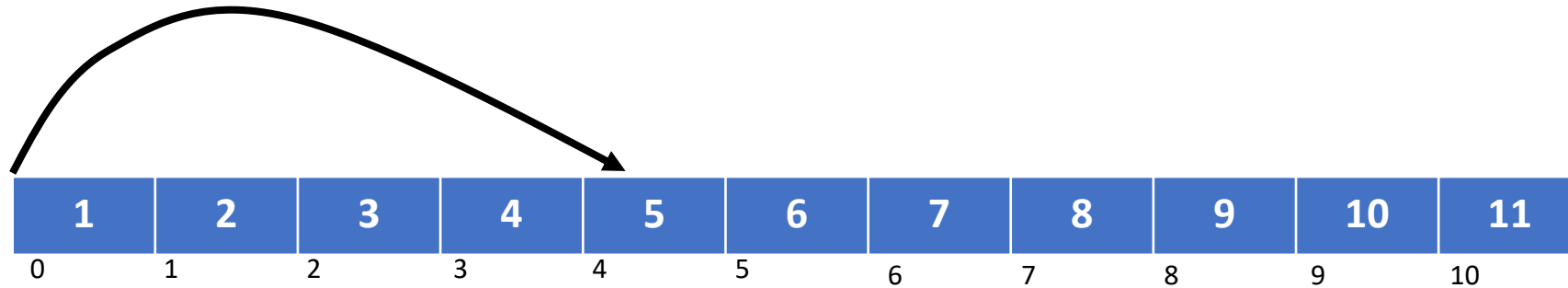
low = 3; high = 4;

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

val[mid] = val[4] = 5; so $x == \text{val}[\text{mid}] \rightarrow \text{success!!}$

Binary vs interpolation search

search for x=5



low = 0; high = 10;

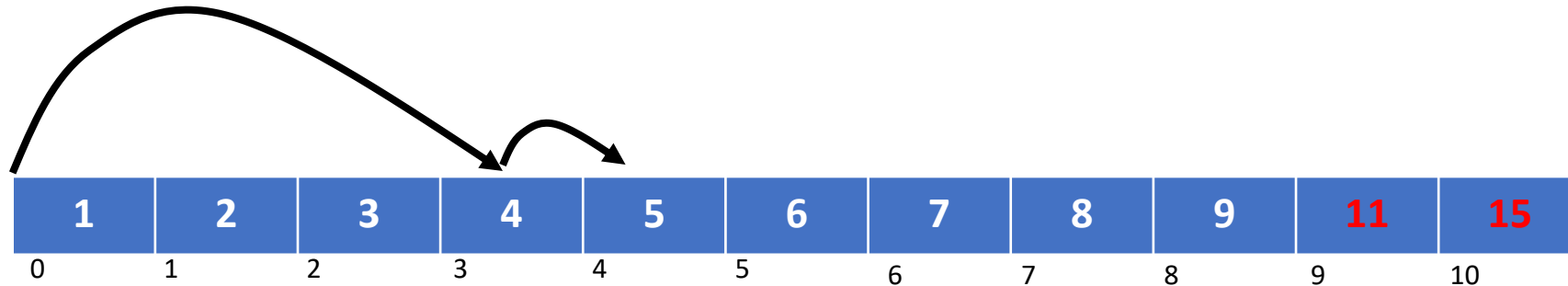
mid = low + ((x - val[low]) * (high - low) / (val[high] - val[low])) = (5-1)*(10-0)/(11-1) = 4

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

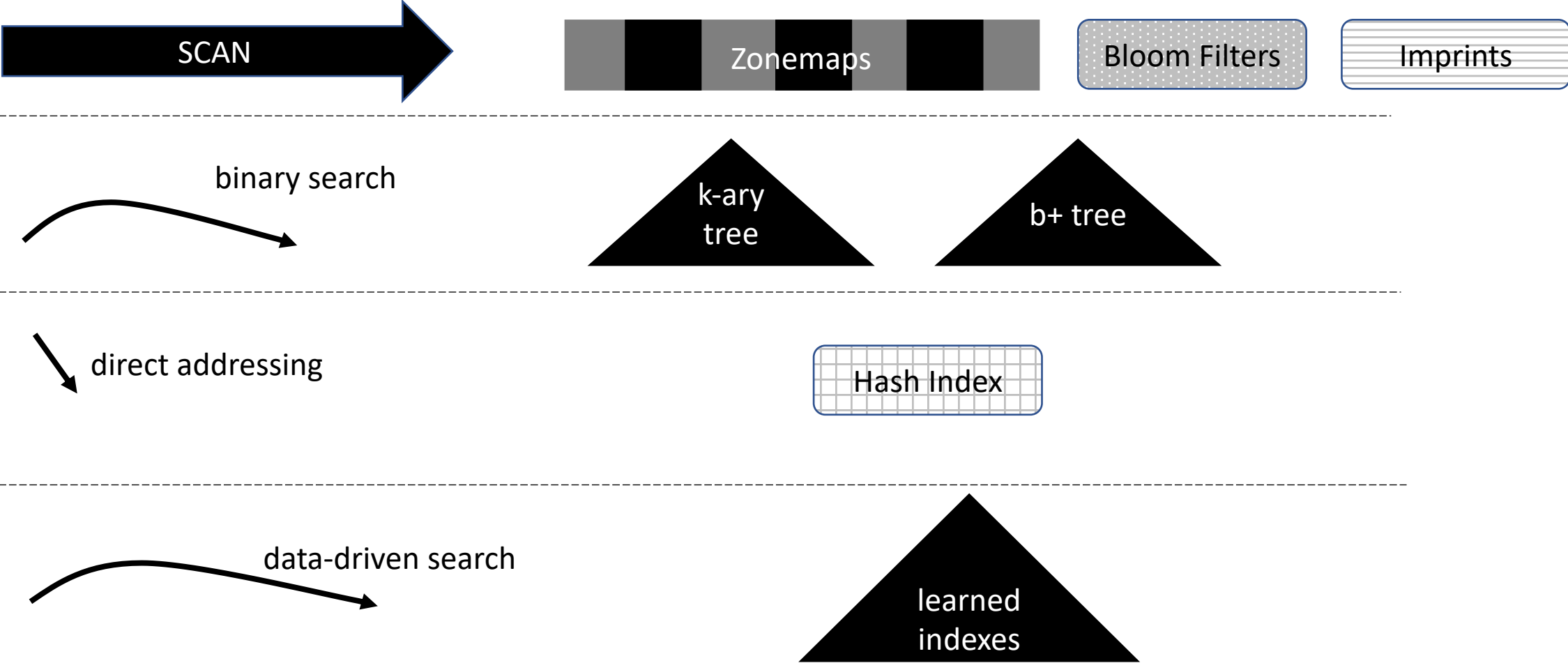
val[mid] = val[4] = 5 → **success!**

still better than binary!

works well with uniform distribution

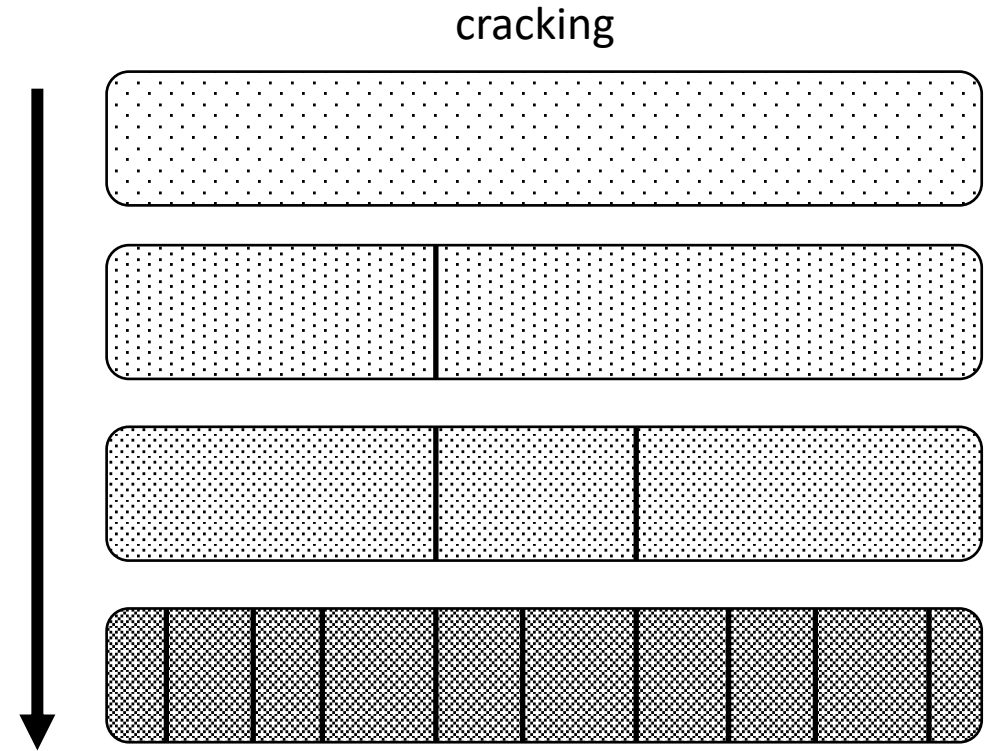
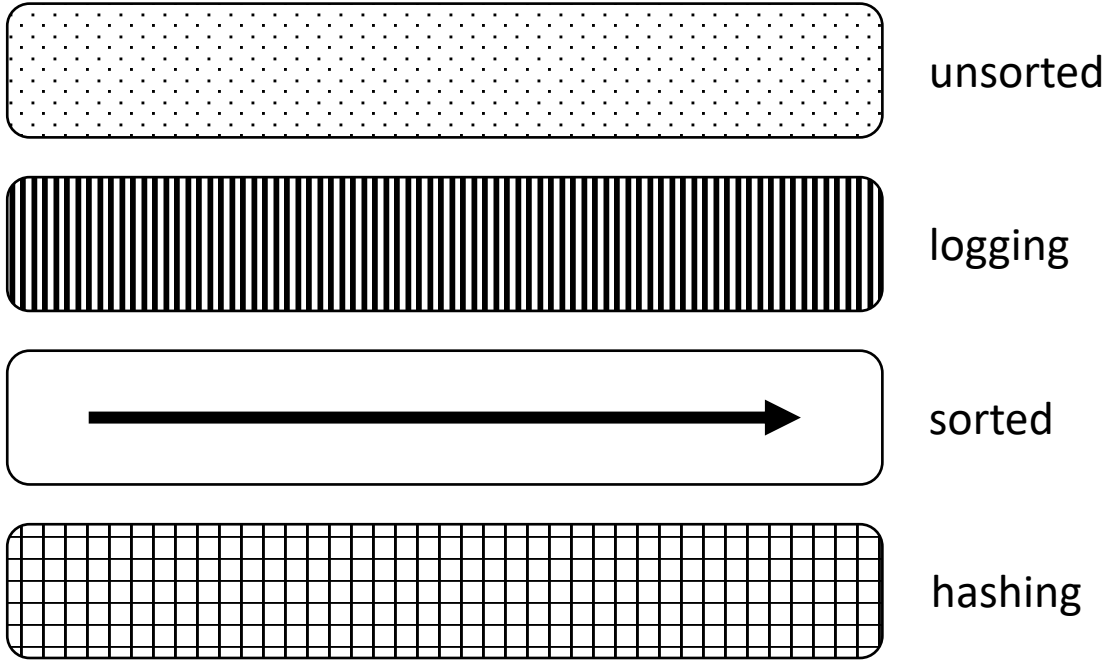
global search using metadata (indexing)

every search algorithm can be materialized and further optimized using indexing



local data organization

decision per partition



local search algorithms



modification policy (updates/deletes/inserts)



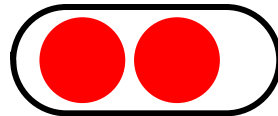
in-place

every update needs to find the “correct” position



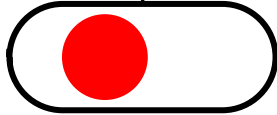
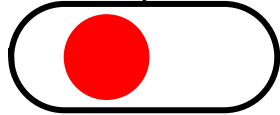
out-of-place

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

global searching (algorithm or index)

local data organization

local search algorithm

modification policy

Workload?



point and range queries, modifications, and some scans



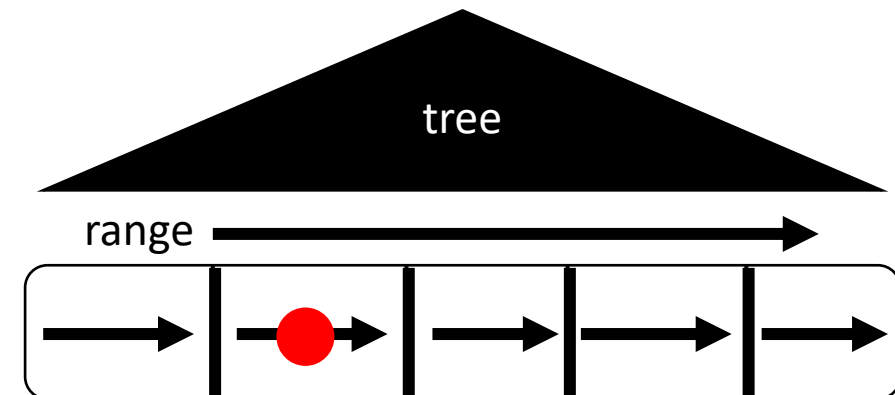
range partitioning

search tree

sorted

binary search / scan

in-place



insert optimized b+ trees

global data organization

global searching (algorithm or index)

local data organization

local search algorithm

modification policy

Workload?



increased number of modifications



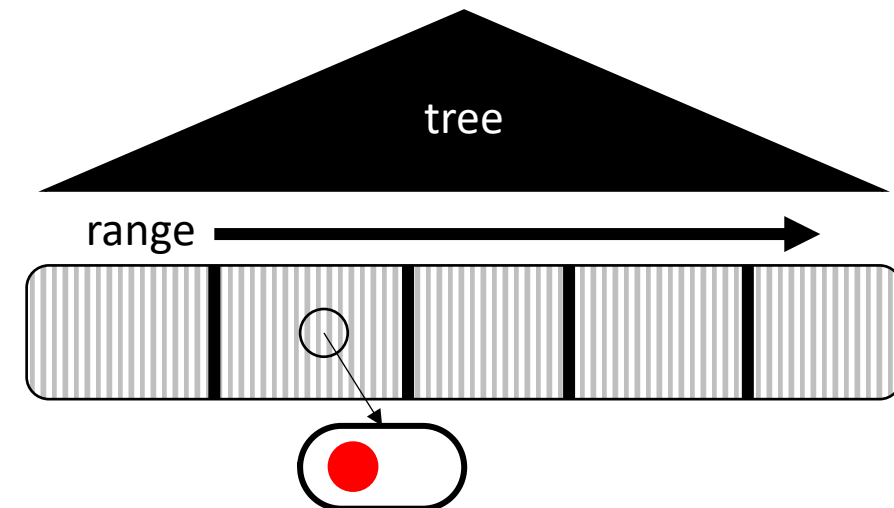
range partitioning

search tree

logging

binary search / scan

deferred in-place



bounded disorder access method



global data organization

global searching (algorithm or index)

local data organization

local search algorithm

modification policy

Workload?



mixed workload, without short range queries

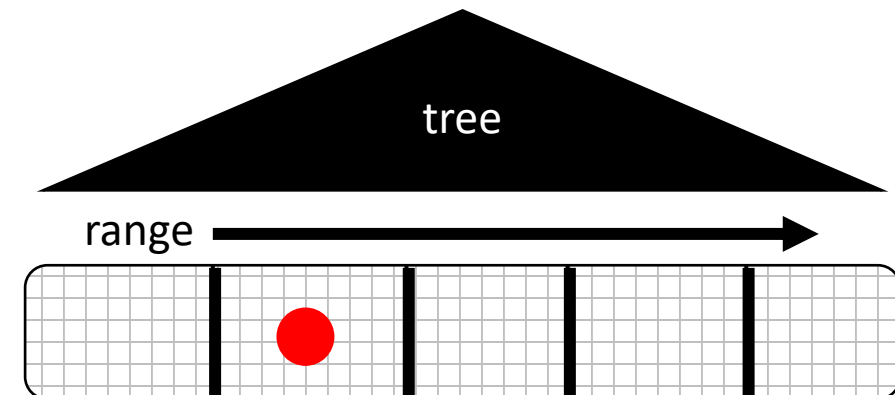
range partitioning

search tree

hashing

hashing

in-place



static hashing

global data organization

global searching (algorithm or index)

local data organization

local search algorithm

modification policy

Workload?



point queries and modifications



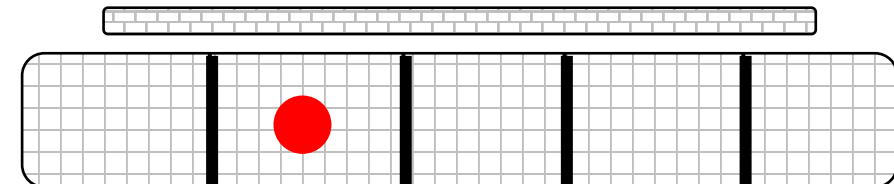
hash partitioning

direct addressing (hashing)

logging

scan

in-place



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

