BOSTON UNIVERSITY

CS 561: Data Systems Architectures

The *design space* of data structures

Prof. Manos Athanassoulis

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

data structures are in the core of: 

b+ trees

hash tables

zonemaps

radix trees

bitmap indexes

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

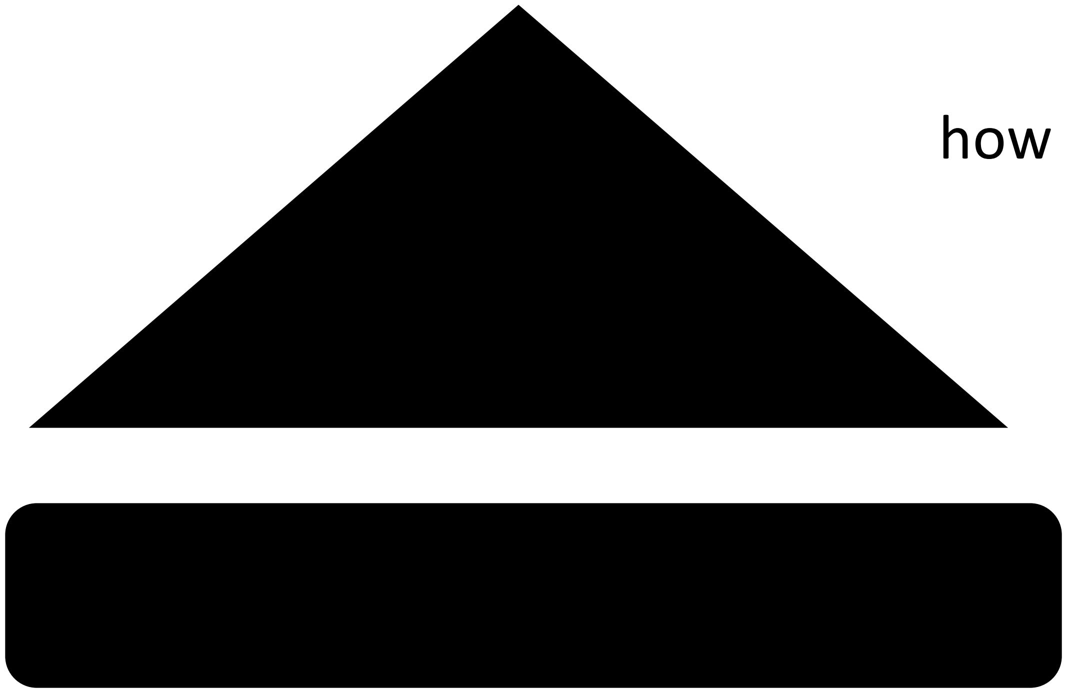
next → hardware (memory/storage/network/compute)

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?

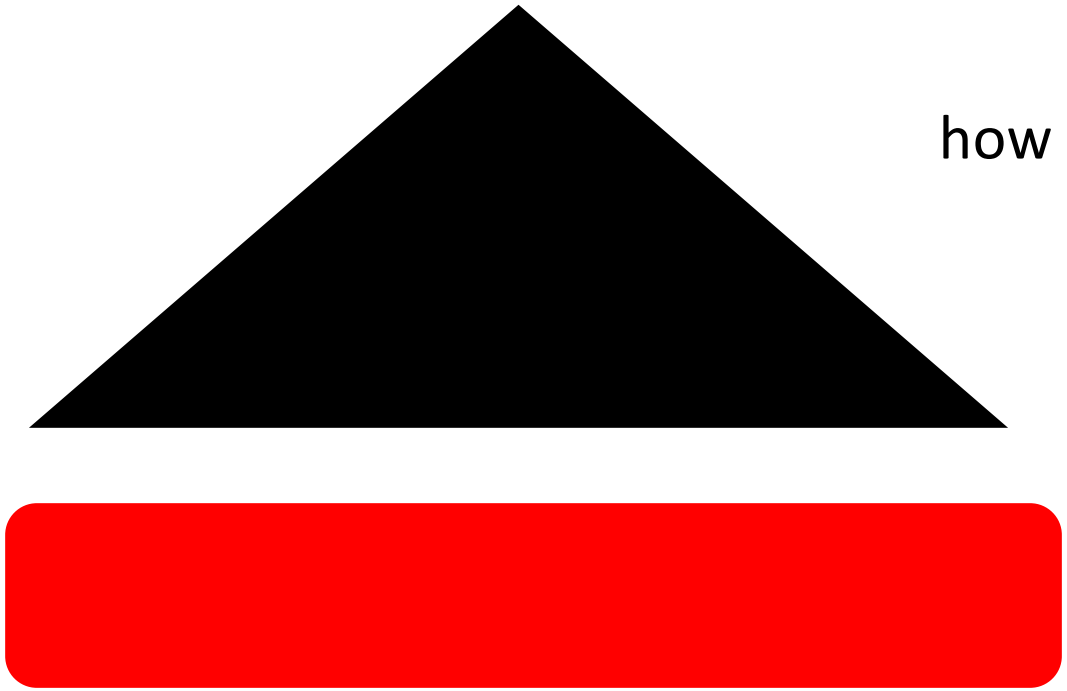
how to update or add new data?

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?

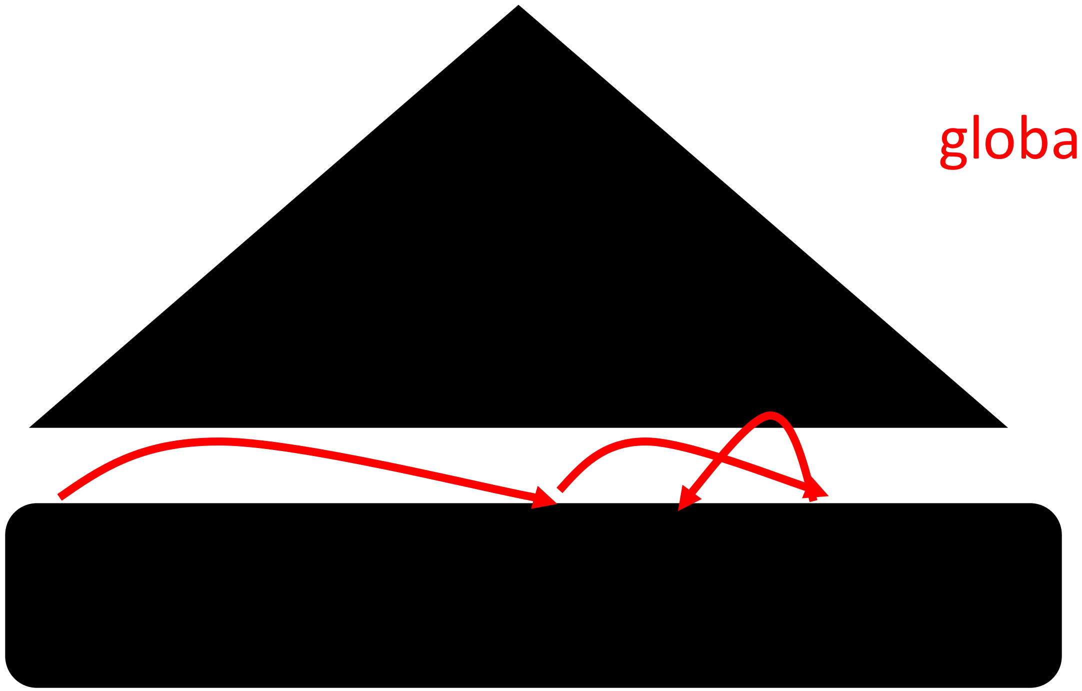
multiple levels of nested organization?

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?

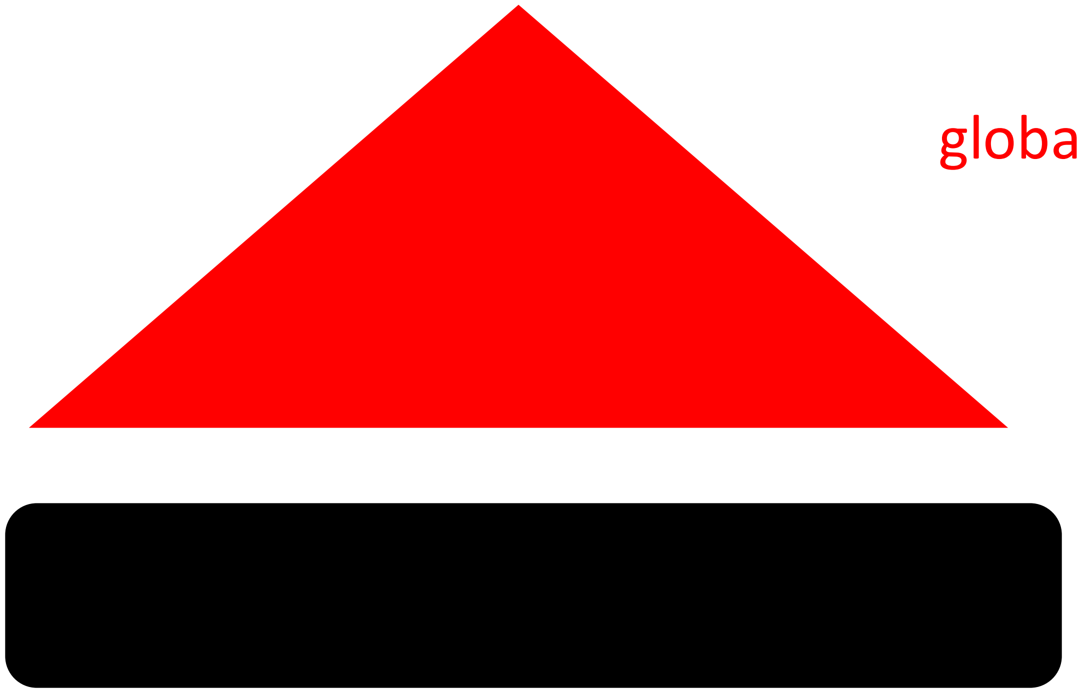
multiple levels of nested organization?

how to update or add new data?

how to exploit additional memory/storage?

should the above decisions be applied eagerly or lazily?

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



global data organization

global search algorithm

metadata for searching

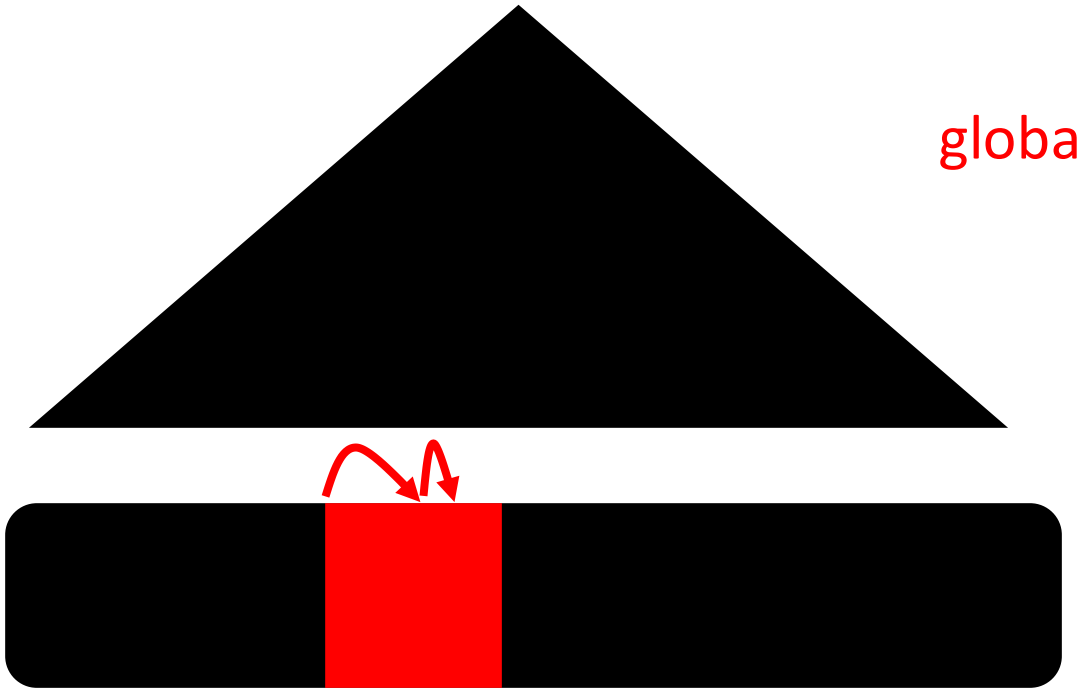
multiple levels of nested organization?

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

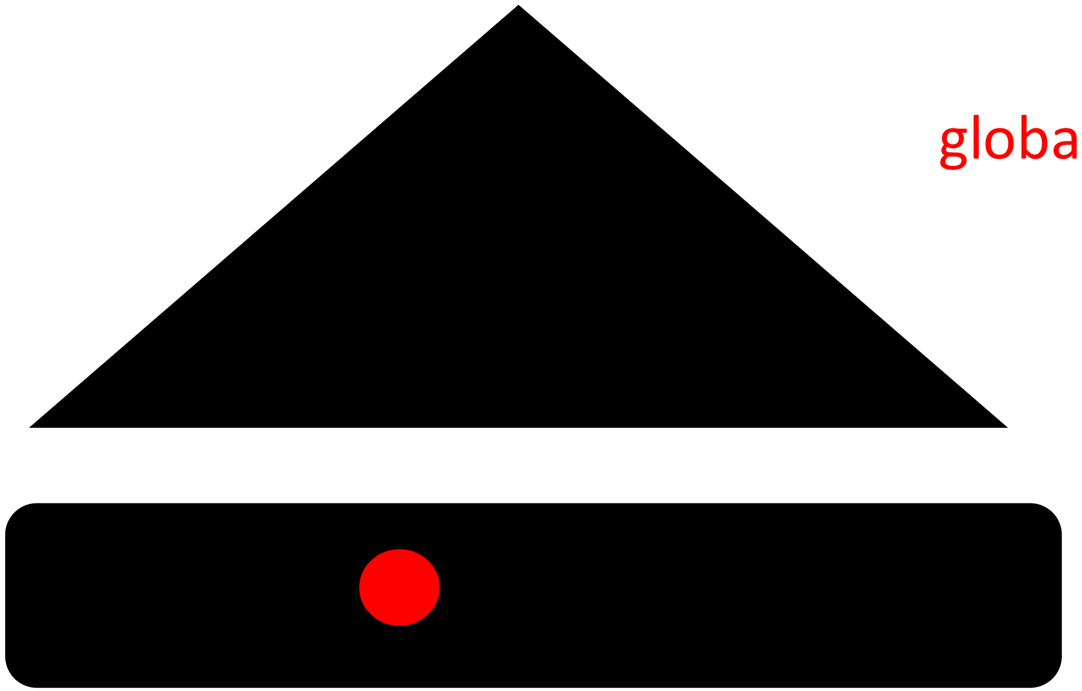
how to update or add new data?

how to exploit additional memory/storage?

should the above decisions be applied eagerly or lazily?

BOSTON UNIVERSITY

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



global data organization

global search algorithm

metadata for searching

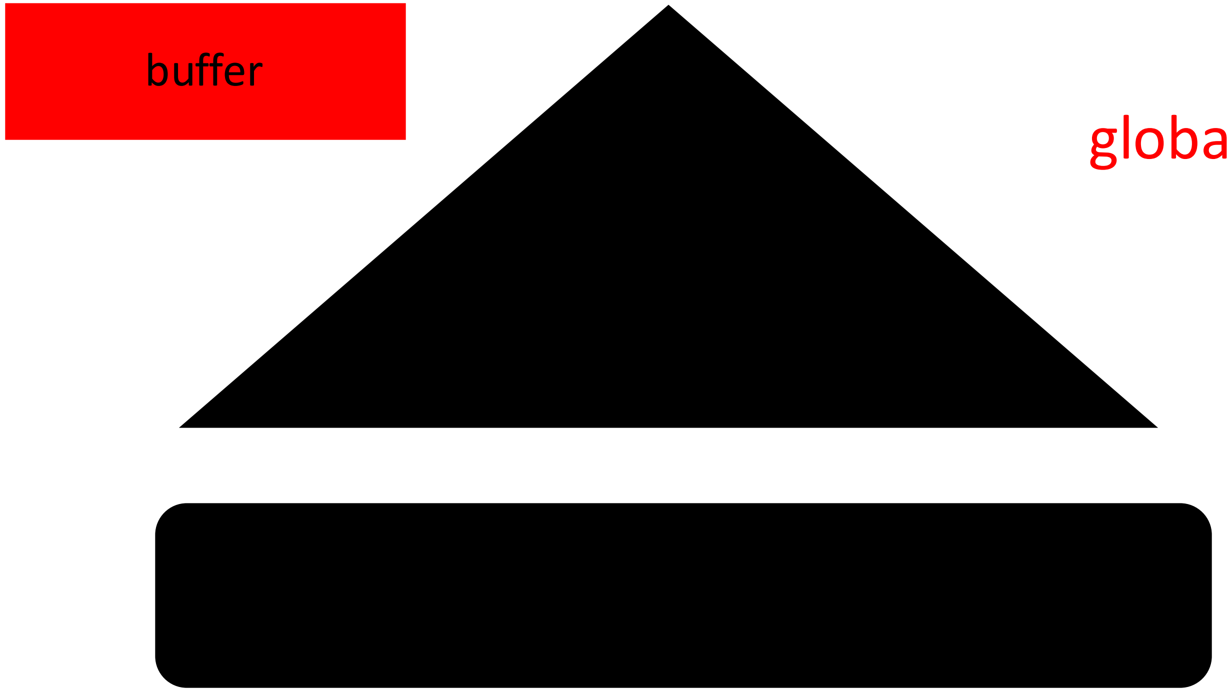
local data organization & search algorithm

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

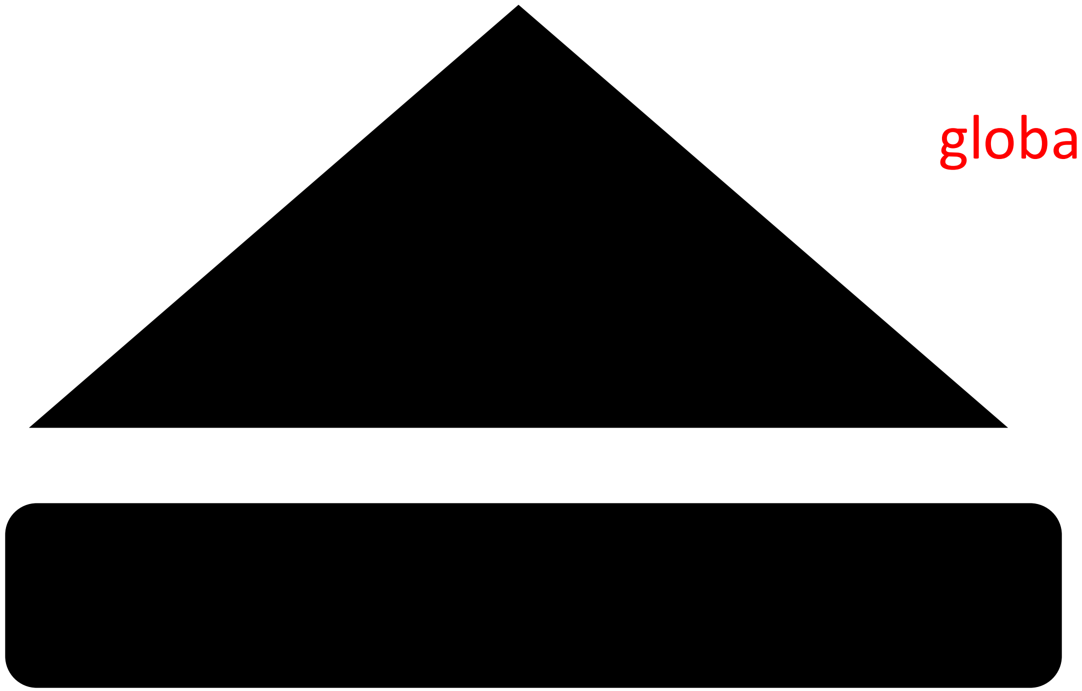
local data organization & search algorithm

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

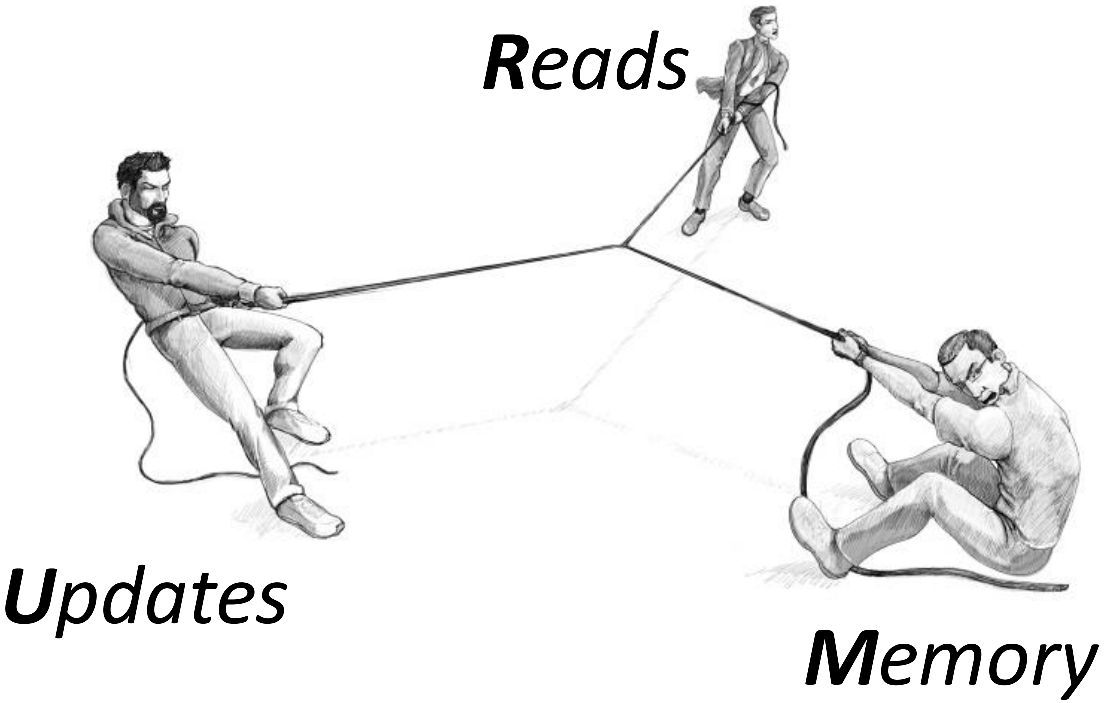
modification policy

batching via buffering

adaptivity

BOSTON UNIVERSITY

data structure designs navigate a three-way tradeoff



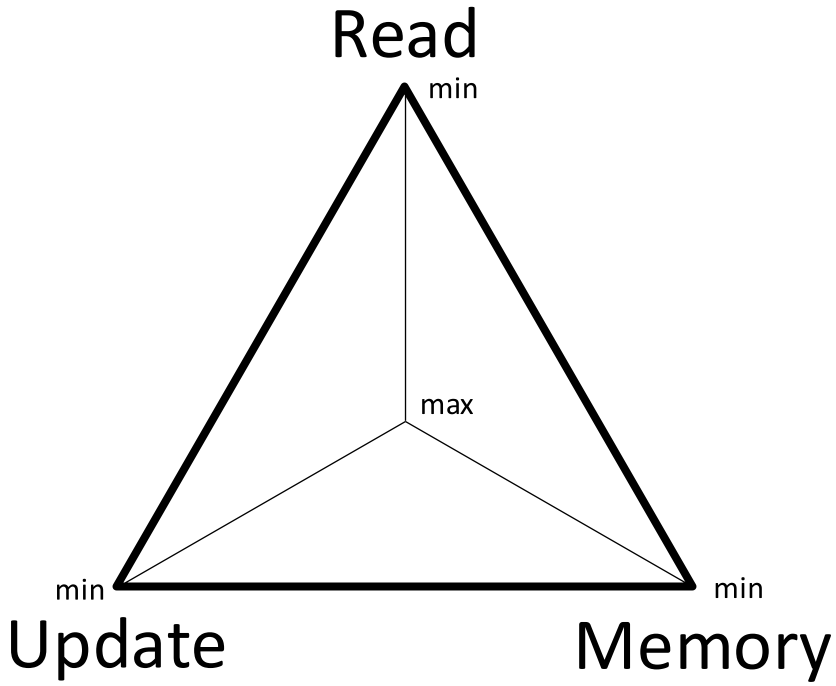
BOSTON UNIVERSITY

The RUM Conjecture

every access method has a (quantifiable)

- read overhead
- update overhead
- memory overhead

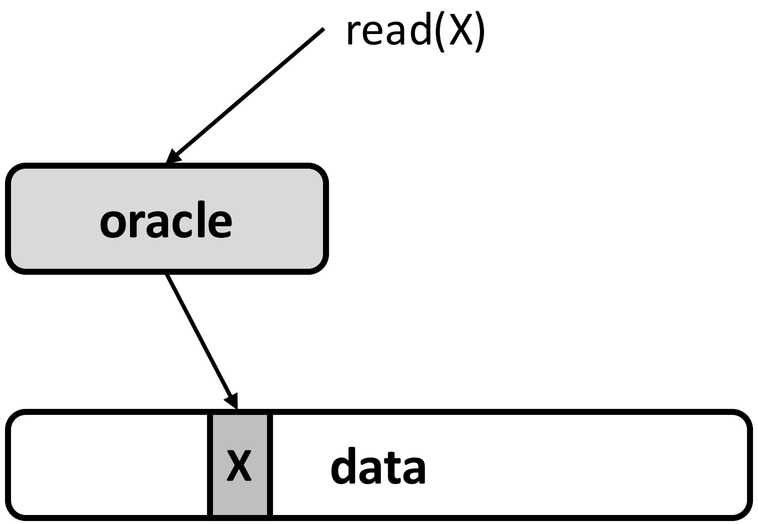
the three of which form a competing triangle



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

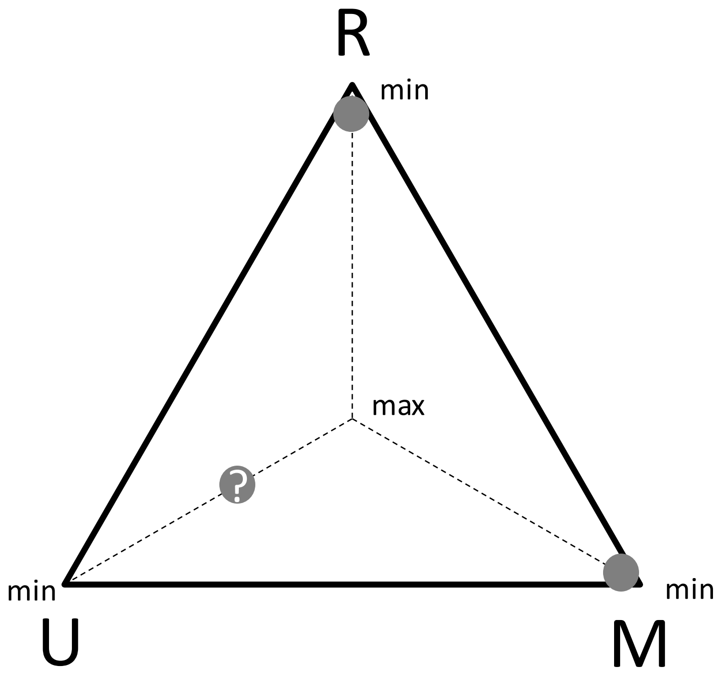
BOSTON UNIVERSITY

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

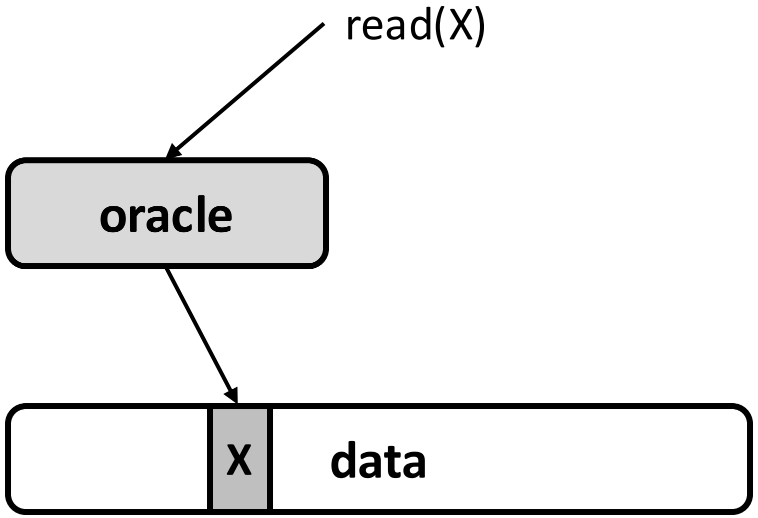


read(x) accesses only the bytes of object X

how *free* can an oracle be?

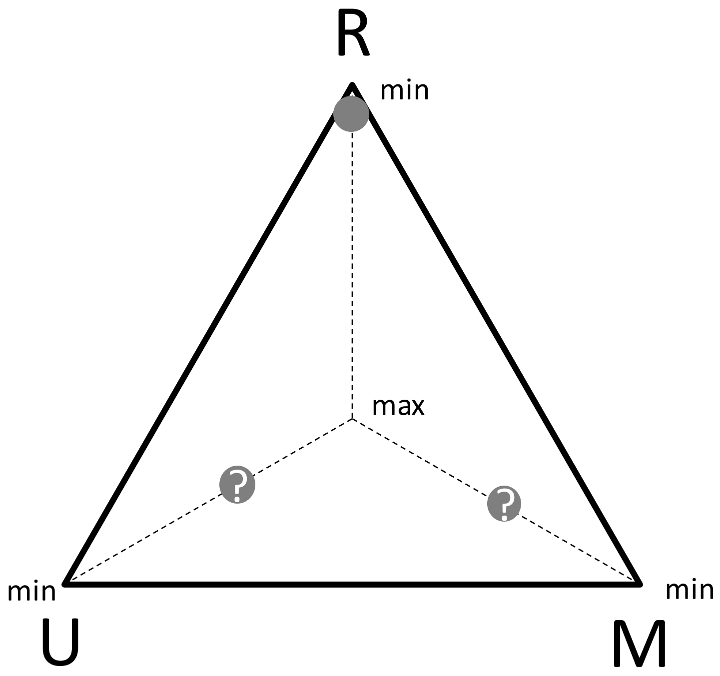


what would be an **optimal read** behavior?



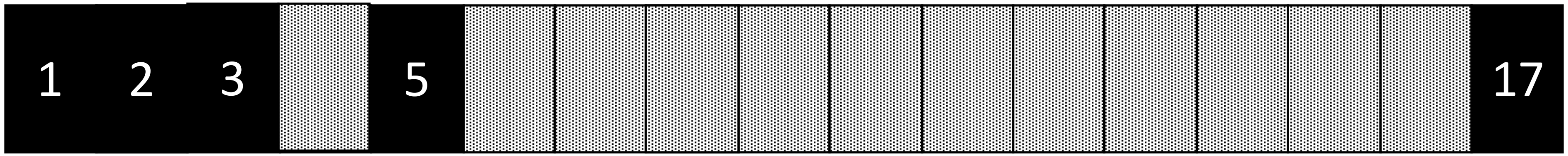
read(x) accesses only the bytes of object X

how *free* can an oracle be?



BOSTON UNIVERSITY

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



update 17 -> 3

minimum read overhead

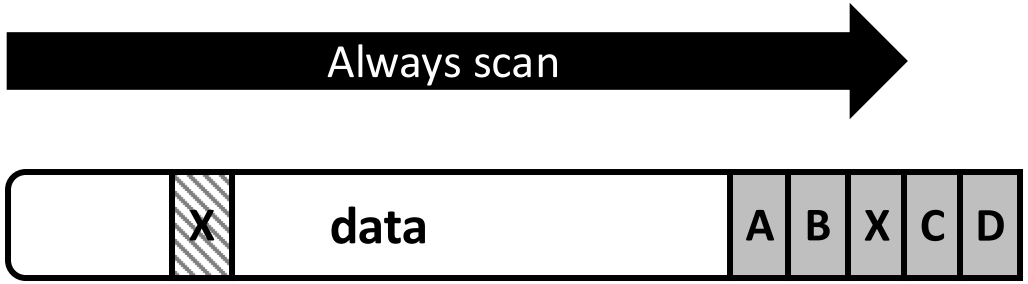
bound update overhead

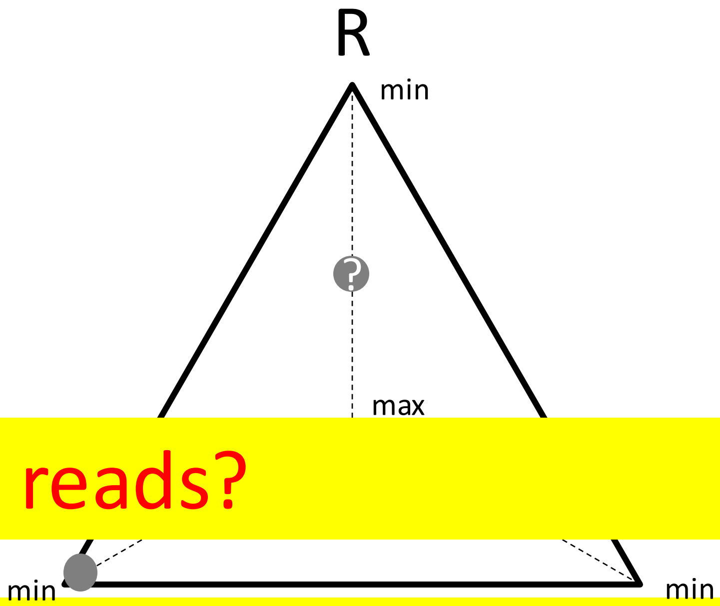
unbounded memory overhead

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

always *append*, and on update *invalidate*

update (X) changes the minimal number of bytes





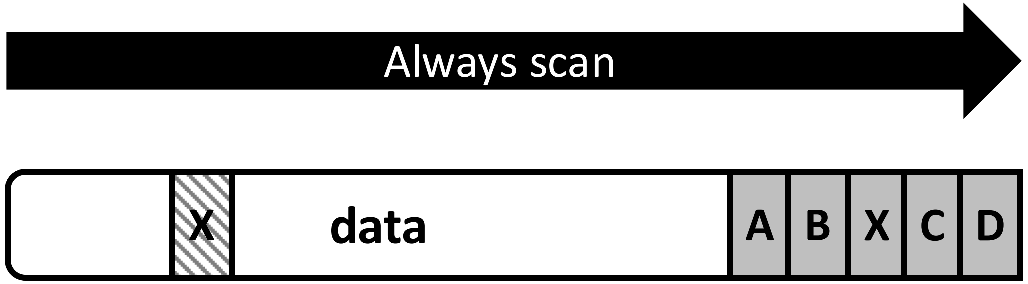
what about reads?

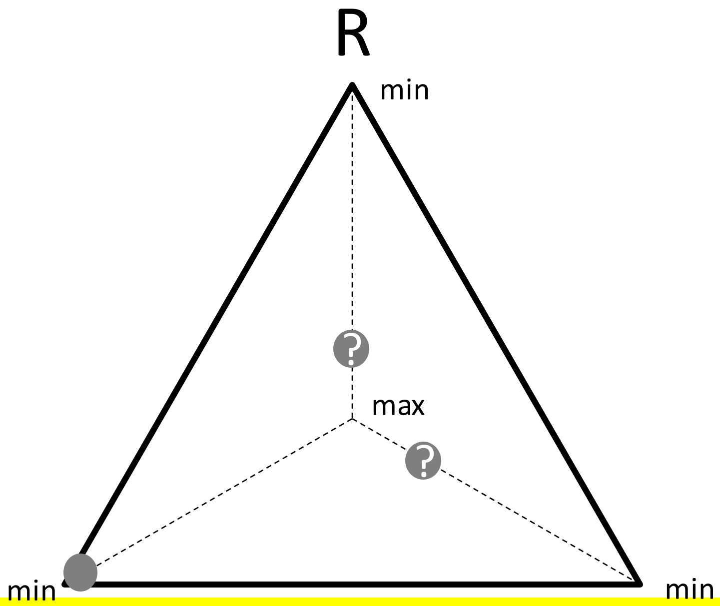
more data?

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

always *append*, and *invalidate* on update

update (X) changes the minimal number of bytes



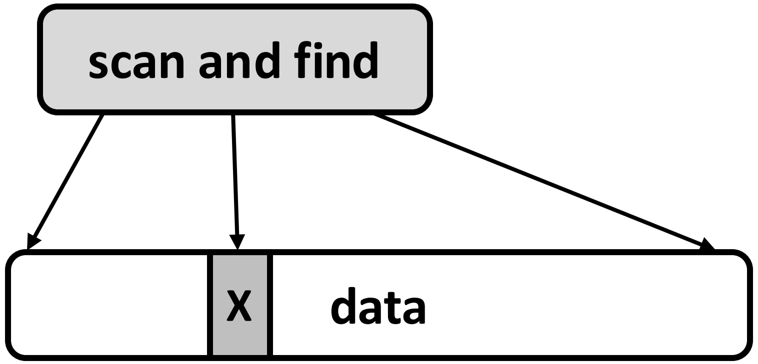


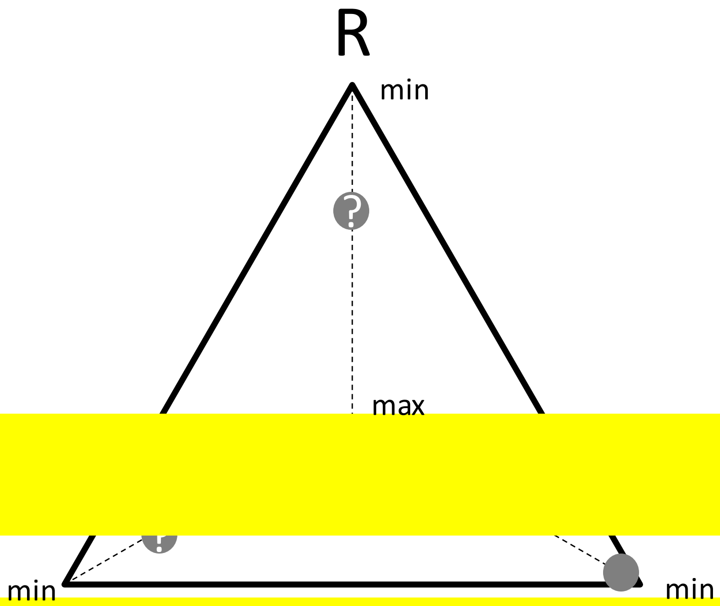
higher read and memory overhead

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

no metadata whatsoever, would result in the smallest memory footprint

scan and in-place updates

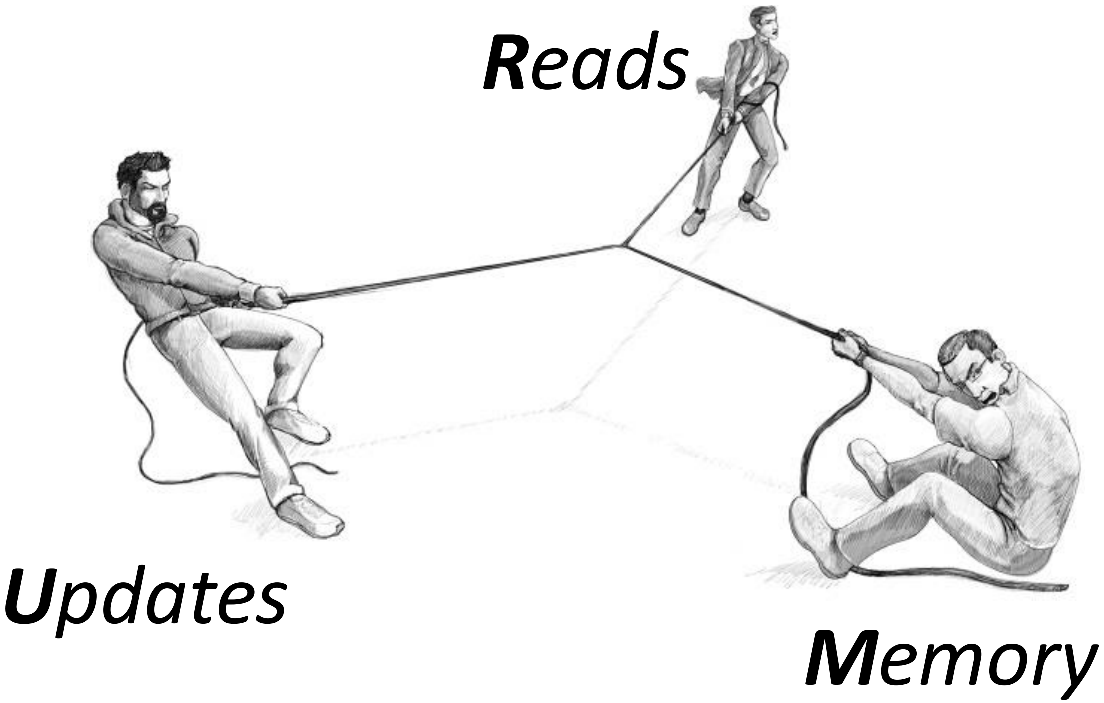




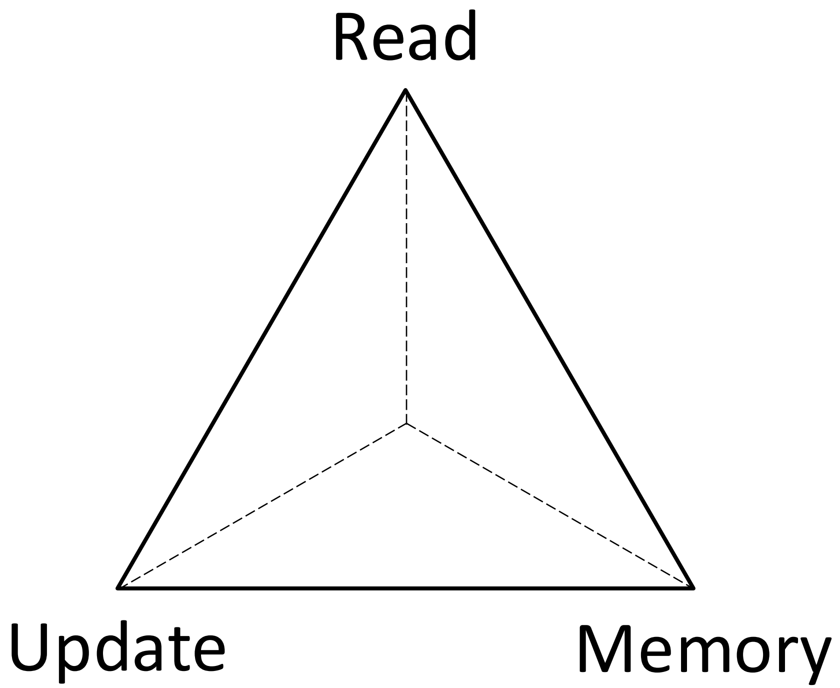
No!

do we need to reach the optimal(s)?

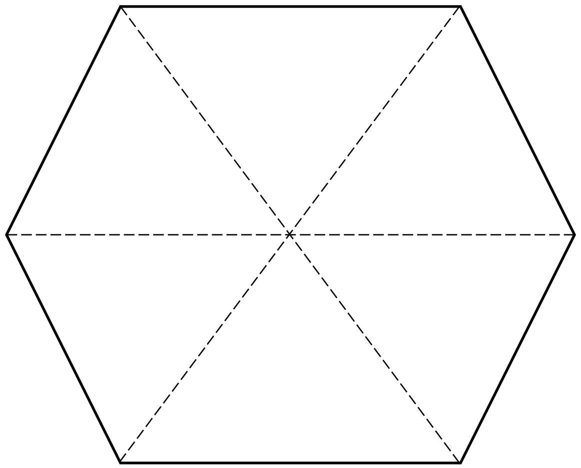
are there only three overheads?



are there only three overheads?



are there only three overheads?



PyRUMID overheads

BOSTON UNIVERSITY

data structures *design dimensions and their values*

global data organization

global search algorithm

metadata for searching

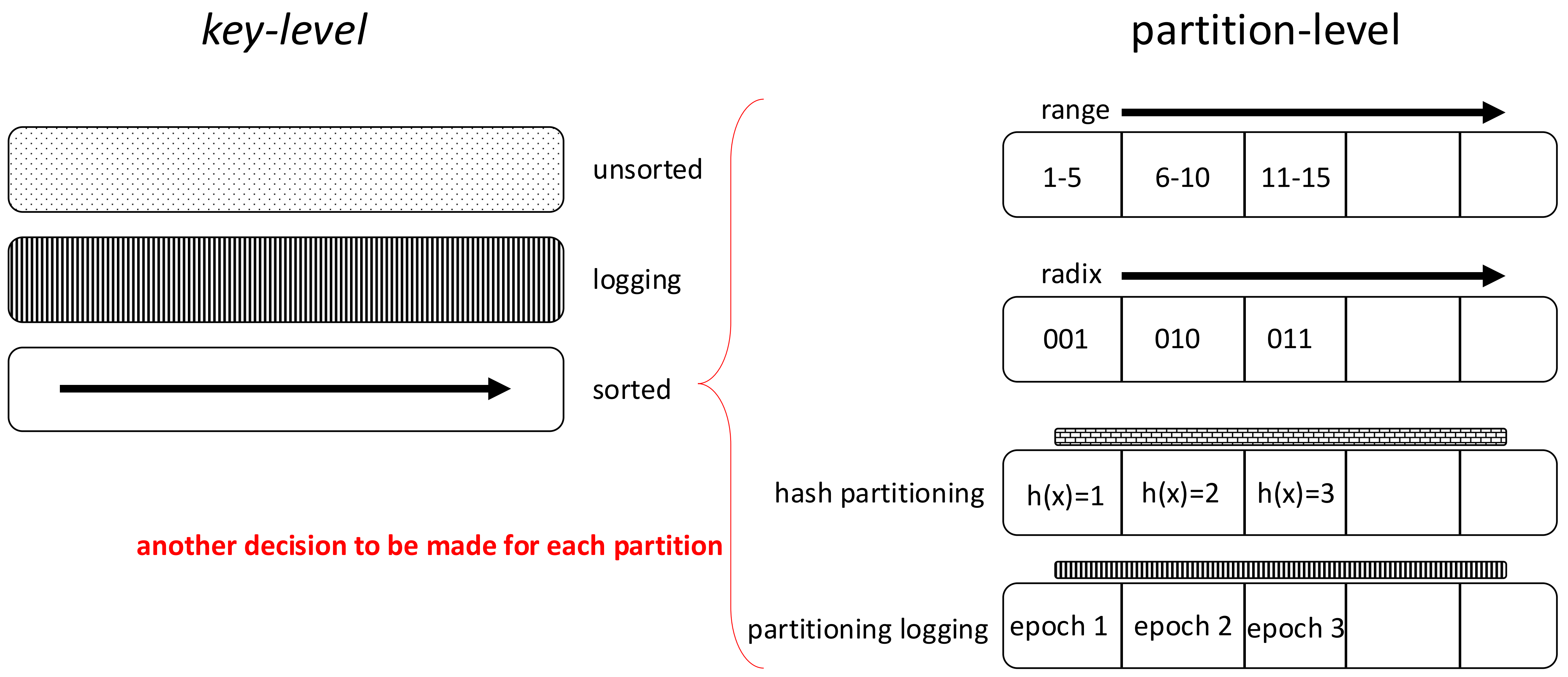
local data organization & search algorithm

modification policy

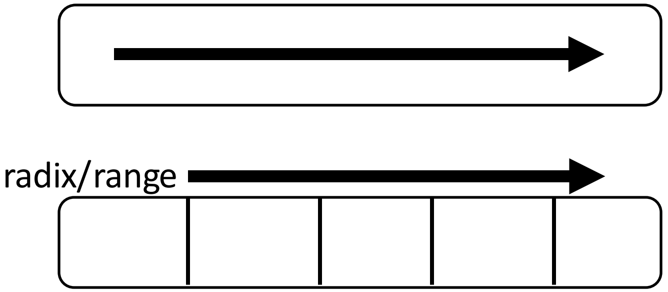
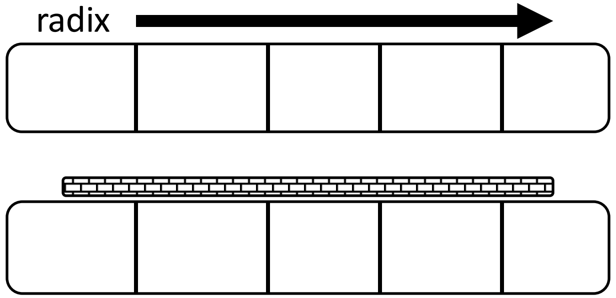
batching via buffering

adaptivity

global data organization



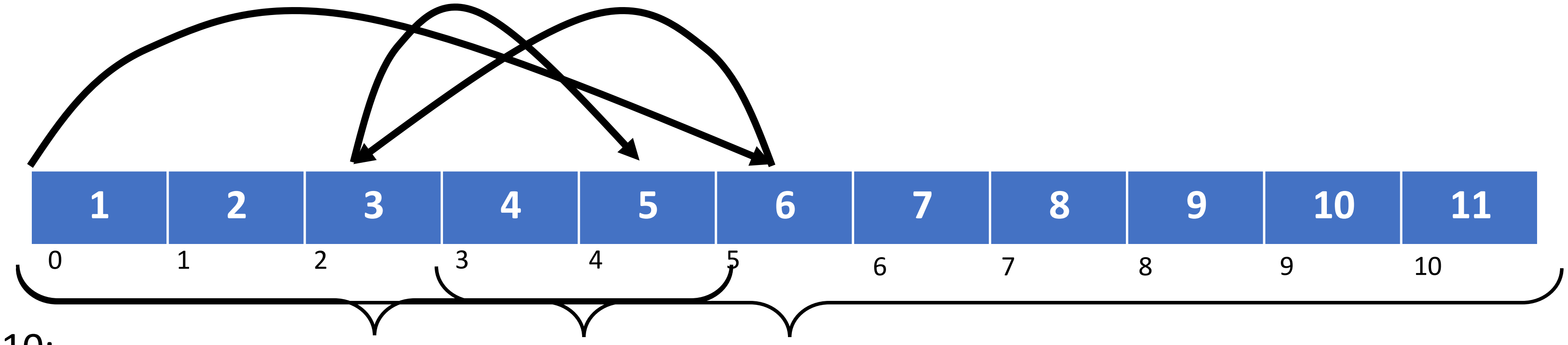
global search algorithm

	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**

BOSTON UNIVERSITY

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!!

BOSTON UNIVERSITY

Binary vs <u>interpolation</u> search

search for x=5

1	2	3	4	5	6	7	8	9	10	11
0	1	2	3	4	5	6	7	8	9	10

low = 0; high = 10;

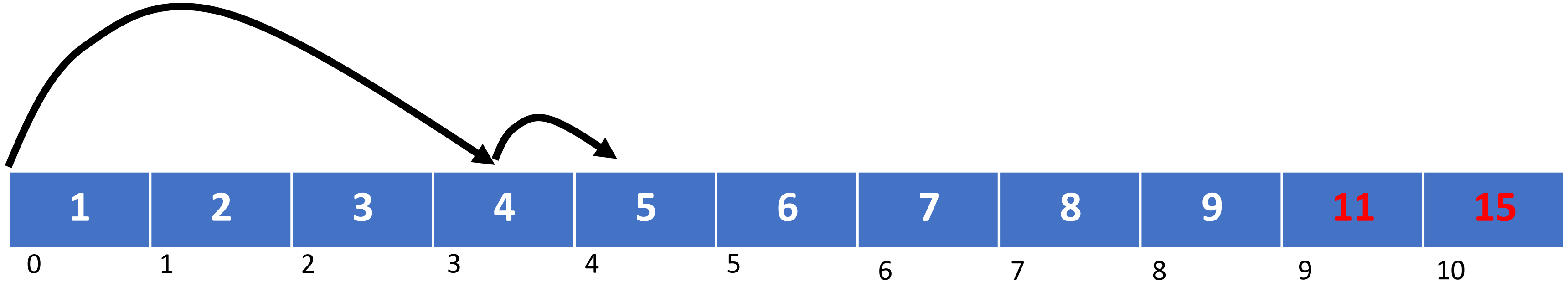
mid = low + ((x - val[low]) * (high - low) / (val[high] - val[low])) = (5-1)*(10-0)/(11-1) = 4

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

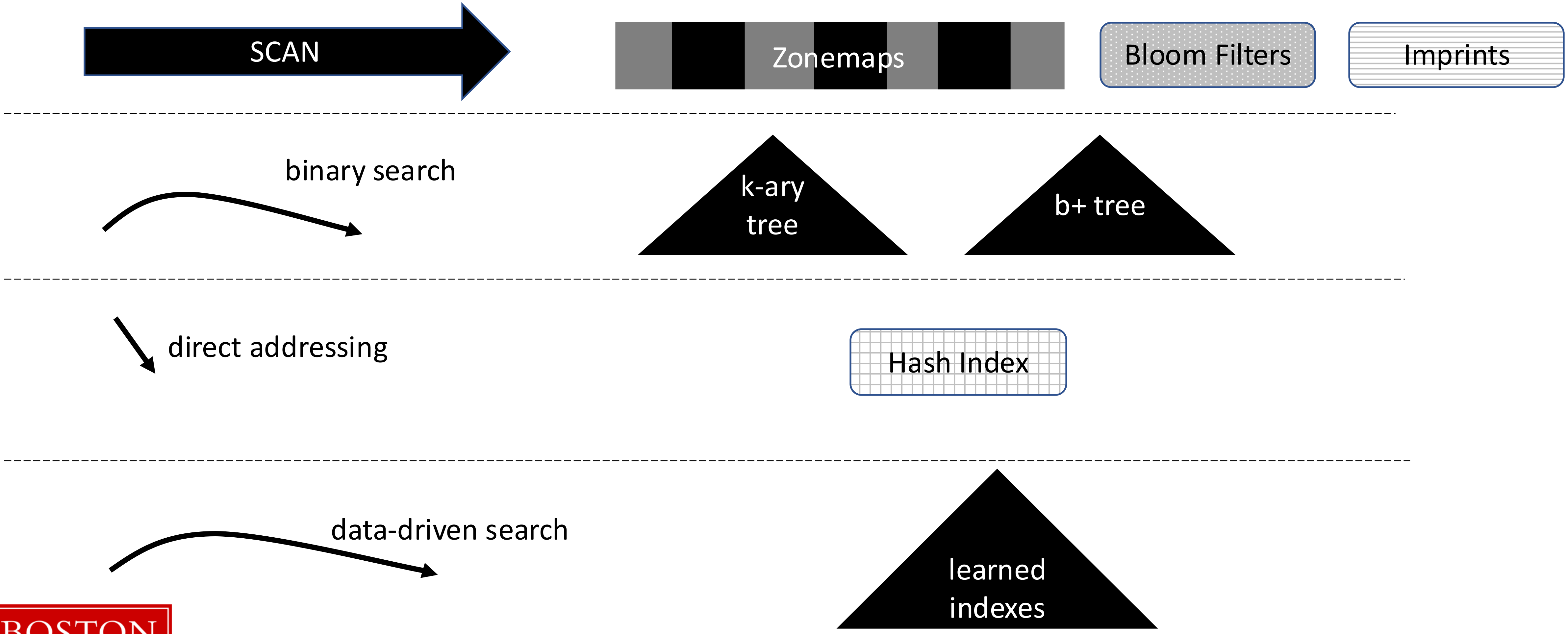
val[mid] = val[4] = 5 ➔ success!

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



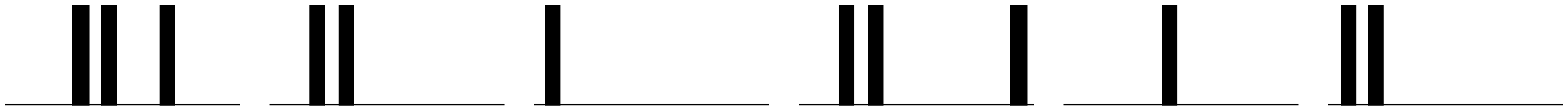
BOSTON UNIVERSITY

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**

Scan vs. Index

$$APS \text{ ratio} = \frac{Index\ Cost}{Scan\ Cost}$$

$>$ 1 scan

$<$ 1 index

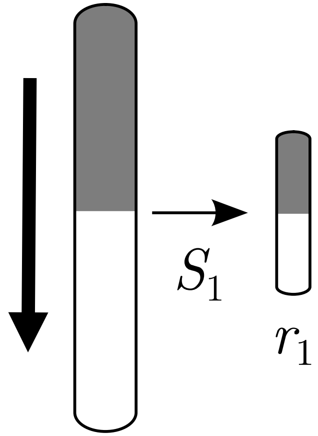
let us model

BOSTON UNIVERSITY

access path selection ***modeling***

N	rows
ts	tuple size
BW	memory bandwidth
sel	query selectivity

scan data



$$ResultWrite = sel \cdot \frac{N \cdot ts}{BW}$$

$$ScanData = \frac{N \cdot ts}{BW}$$

what if we have ***q queries?***

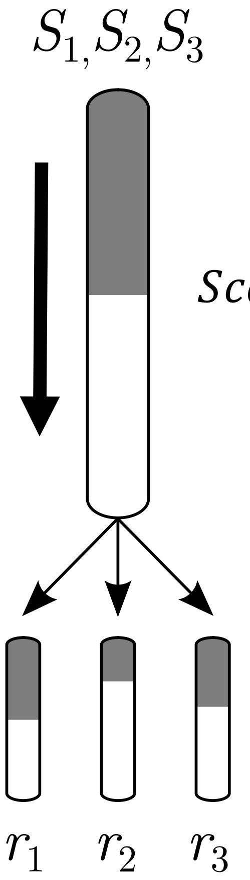
BOSTON UNIVERSITY

access path selection ***modeling***

***q concurrent** queries*

N	rows
ts	tuple size
BW	memory bandwidth
sel	query selectivity
q	queries
p	predicate evaluation cost (CPU)

scan data



$$ScanData = max\left(\frac{N \cdot ts}{BW}, N(p \cdot q)\right)$$

CPU cost per query (may be too high!)

selectivity per query

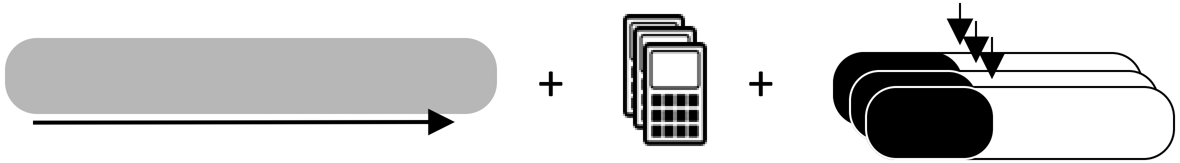
$$ResultWrite = \sum_{i=1}^{q} sel_i \cdot \frac{N \cdot ts}{BW} = S_q \cdot \frac{N \cdot ts}{BW}, \quad s.t. \quad S_q = \sum_{i=1}^{q} sel_i$$

total query selectivity (sum)

scan → data

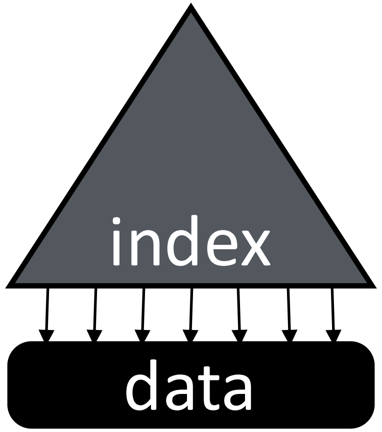
BOSTON UNIVERSITY

$$ScanCost = max\left(\frac{N \cdot ts}{BW}, N \cdot p \cdot q\right) + S_q \cdot \frac{N \cdot ts}{BW}$$



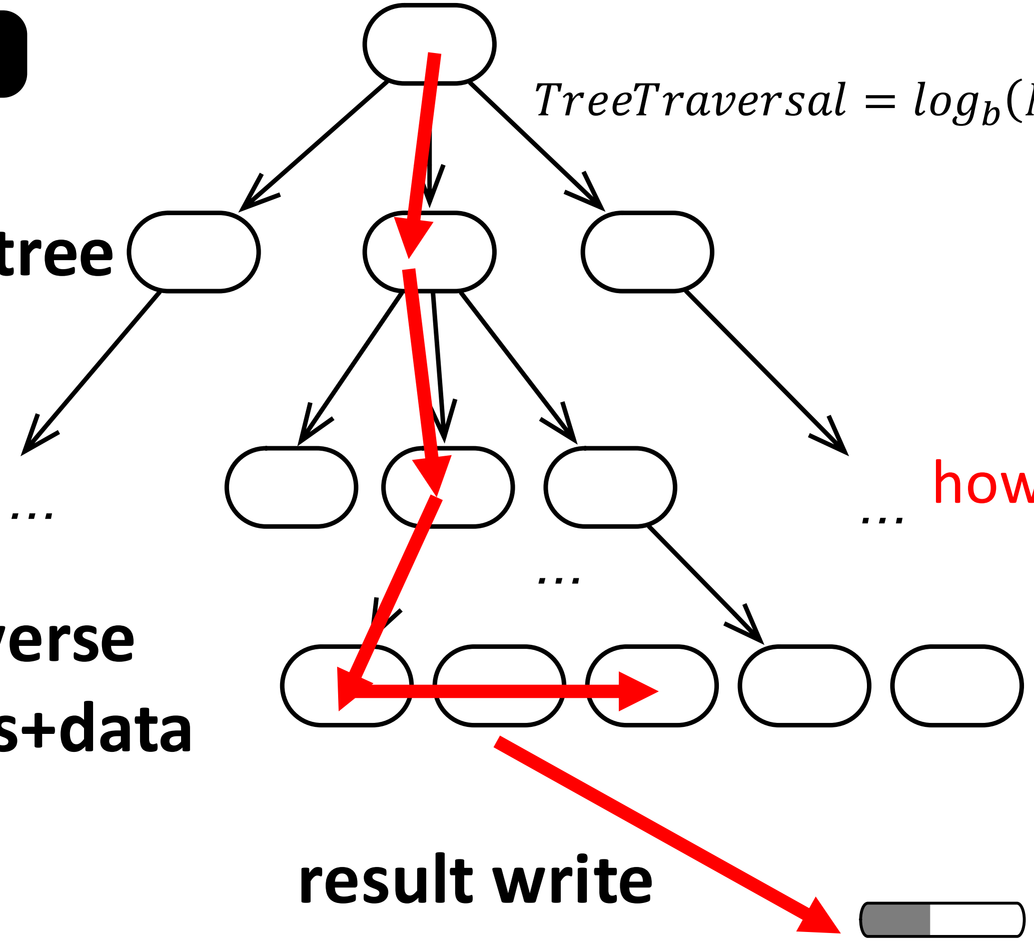
$ScanData$ $\quad PredicateEval$ $\quad ResultWrite$

access path selection *modeling*



N	rows
ts	tuple size
BW	memory bandwidth
sel	query selectivity
q	queries
p	predicate evaluation cost (CPU)
b	branching factor (fanout)
C_M	cache miss latency
its	index tuple size

traverse tree



$$TreeTraversal = log_b(N) \cdot C_M$$

how to make index drop-in replacement?

traverse leaves+data

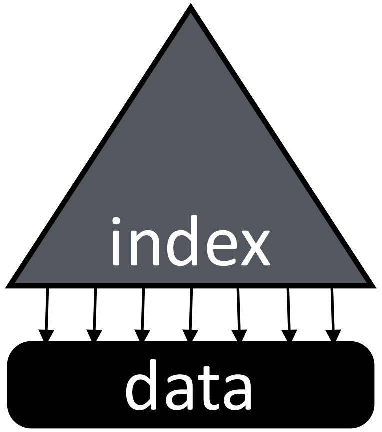
$$LeavesTraversal = sel \cdot \frac{N}{b} \cdot C_M$$

$$DataTraversal = sel \cdot \frac{N \cdot its}{BW}$$

result write

$$ResultWrite = sel \cdot \frac{N \cdot ts}{BW}$$

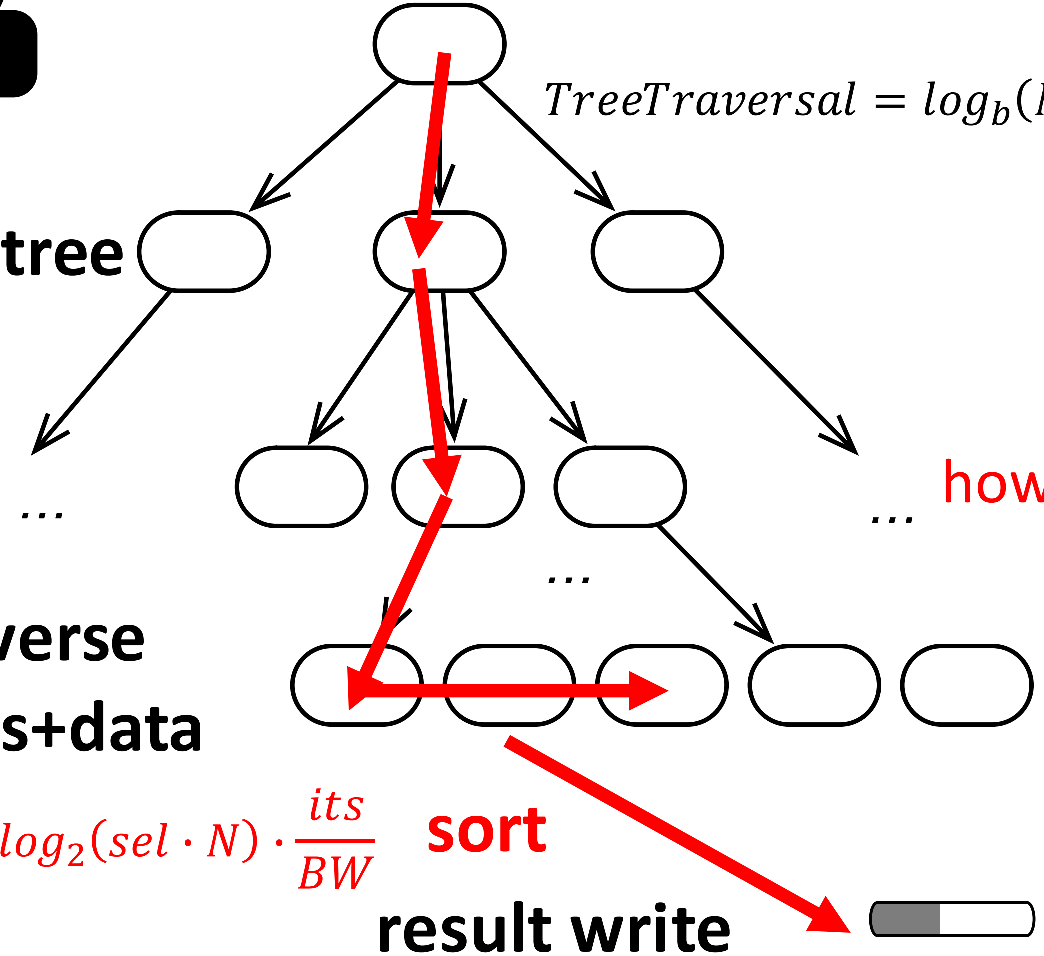
BOSTON UNIVERSITY



access path selection ***modeling***

N	rows
ts	tuple size
BW	memory bandwidth
sel	query selectivity
q	queries
p	predicate evaluation cost (CPU)
b	branching factor (fanout)
C_M	cache miss latency
its	index tuple size

traverse tree



$$TreeTraversal = log_b(N) \cdot C_M$$

how to make index drop-in replacement?

traverse leaves+data

$$LeavesTraversal = sel \cdot \frac{N}{b} \cdot C_M$$

$$DataTraversal = sel \cdot \frac{N \cdot its}{BW}$$

$$Sort = sel \cdot N \cdot log_2(sel \cdot N) \cdot \frac{its}{BW}$$

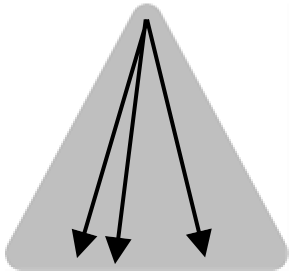
sort

result write

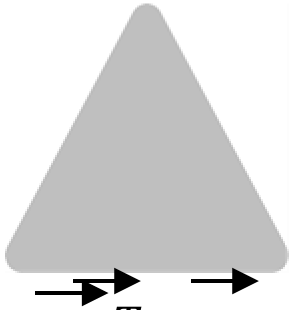
$$ResultWrite = sel \cdot \frac{N \cdot ts}{BW}$$

BOSTON UNIVERSITY

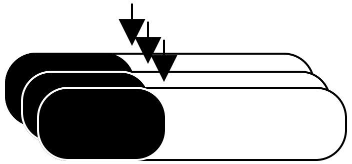
$$IndexCost = q \cdot log_b(N) \cdot C_M + S_q \cdot \left(\frac{N}{b} \cdot C_M + \frac{N \cdot its}{BW} + \frac{N \cdot ts}{BW}\right)$$



+



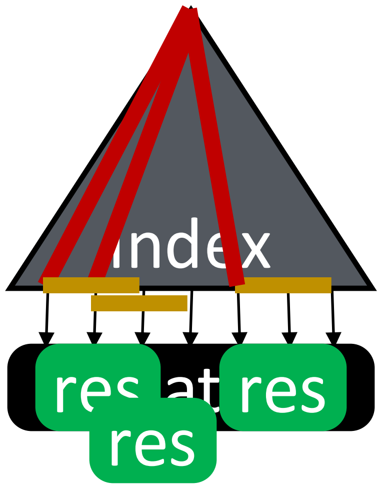
+



$TreeTraversal$ $LeavesTraversal$ $ResultWrite$

$DataTraversal$

new access path selection



index design

data size & hardware

$$APS\ \text{ratio} = \frac{q \cdot TreeTraversal + S_q \cdot (LeavesTraversal + DataTraversal + ResultWrite)}{max(ScanData, CPU\ cost\ of\ q\ predicates) + S_q \cdot ResultWrite}$$

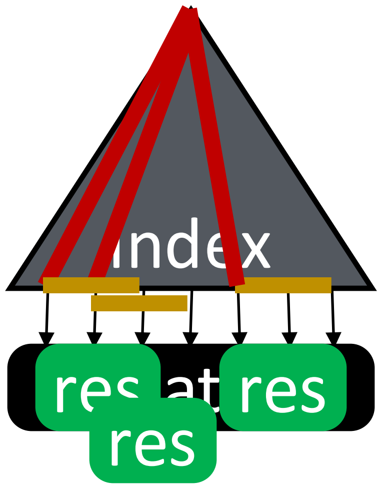
data size (for result)

Dynamic Parameter

q #concurrent read queries

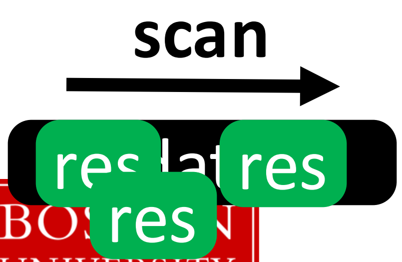
S_q sum of query selectivity of ***q*** queries $S_q = \sum_{i=1}^{q} sel_i$

new access path selection



N	rows
ts	tuple size
BW	memory bandwidth
sel	query selectivity
q	queries
p	predicate evaluation cost (CPU)
b	branching factor (fanout)
C_M	cache miss latency
its	index tuple size

$$APS\text{ ratio} = \frac{q \cdot log_b(N) \cdot C_M + S_q \cdot \left(\frac{N}{b} \cdot C_M + \frac{N \cdot its}{BW} + \frac{N \cdot ts}{BW}\right)}{max\left(\frac{N \cdot ts}{BW}, q \cdot p \cdot N\right) + S_q \cdot \frac{N \cdot ts}{BW}}$$



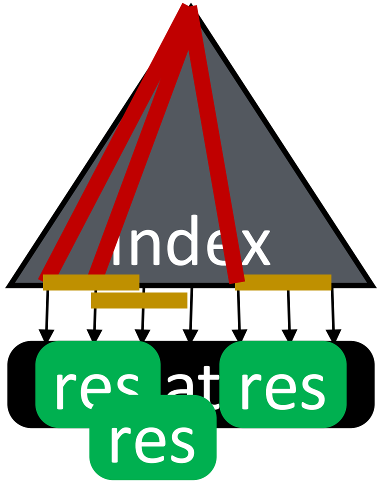
Dynamic Parameter

q #concurrent read queries

S_q sum of query selectivity of ***q*** queries $S_q = \sum_{i=1}^{q} sel_i$

BOSTON UNIVERSITY

new access path selection



index design

data size

hardware characteristics

data layout

$$APS\ \text{ratio} = \frac{q\,\frac{\log_b(N)}{N}\,BW \cdot C_M + S_q \cdot \left(\frac{BW \cdot C_M}{b} + (ts + its)\right)}{max(ts, q\,(p \cdot BW)) + S_q \cdot ts}$$

N	rows
ts	tuple size
BW	memory bandwidth
sel	query selectivity
q	queries
p	predicate evaluation cost (CPU)
b	branching factor (fanout)
C_M	cache miss latency
its	index tuple size

Dynamic Parameter

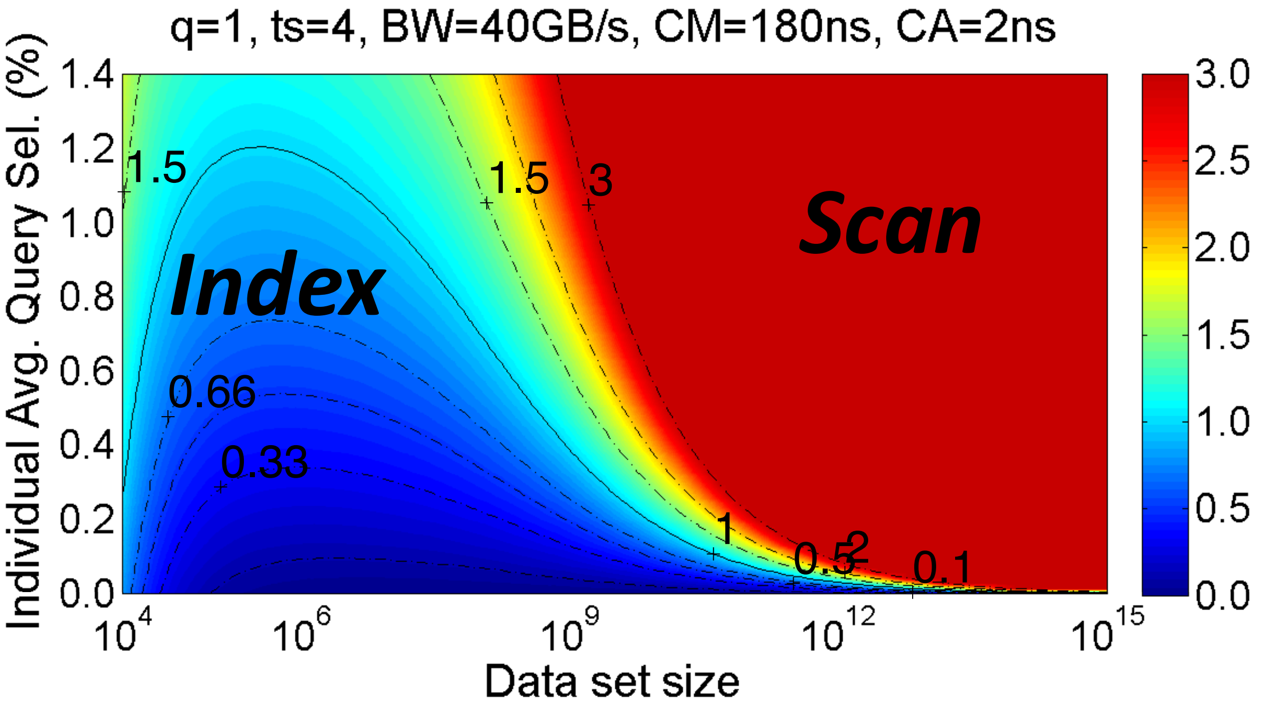
q #concurrent read queries

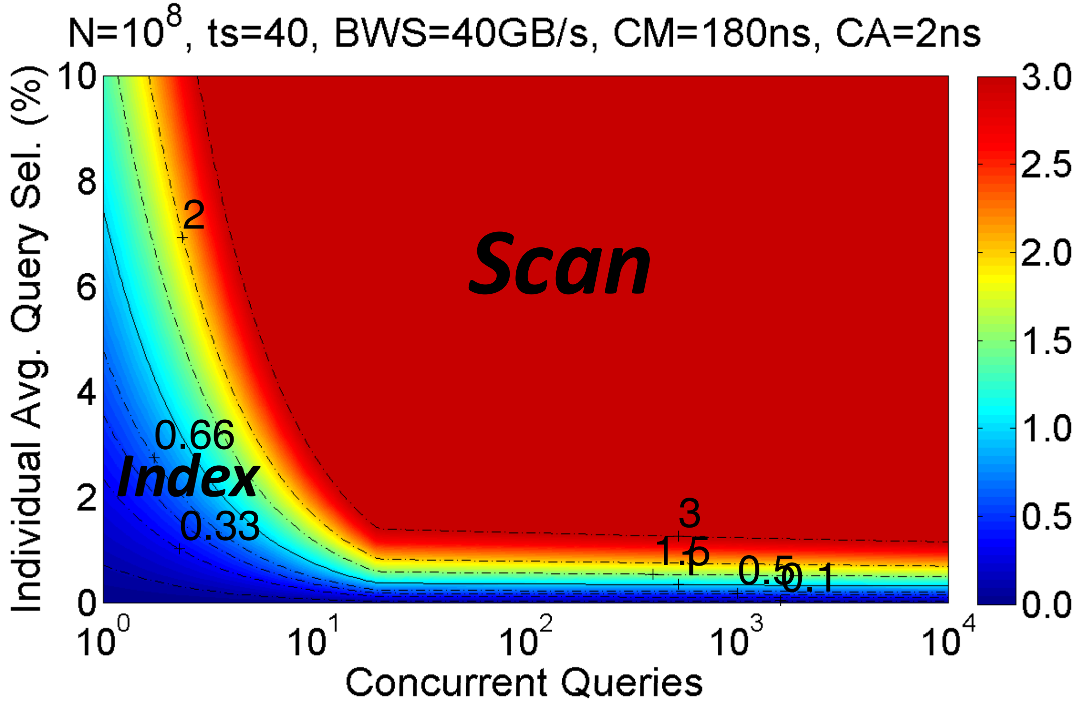
S_q sum of query selectivity of ***q*** queries $S_q = \sum_{i=1}^{q} sel_i$

new access path selection

Modeling

high selectivity → scan

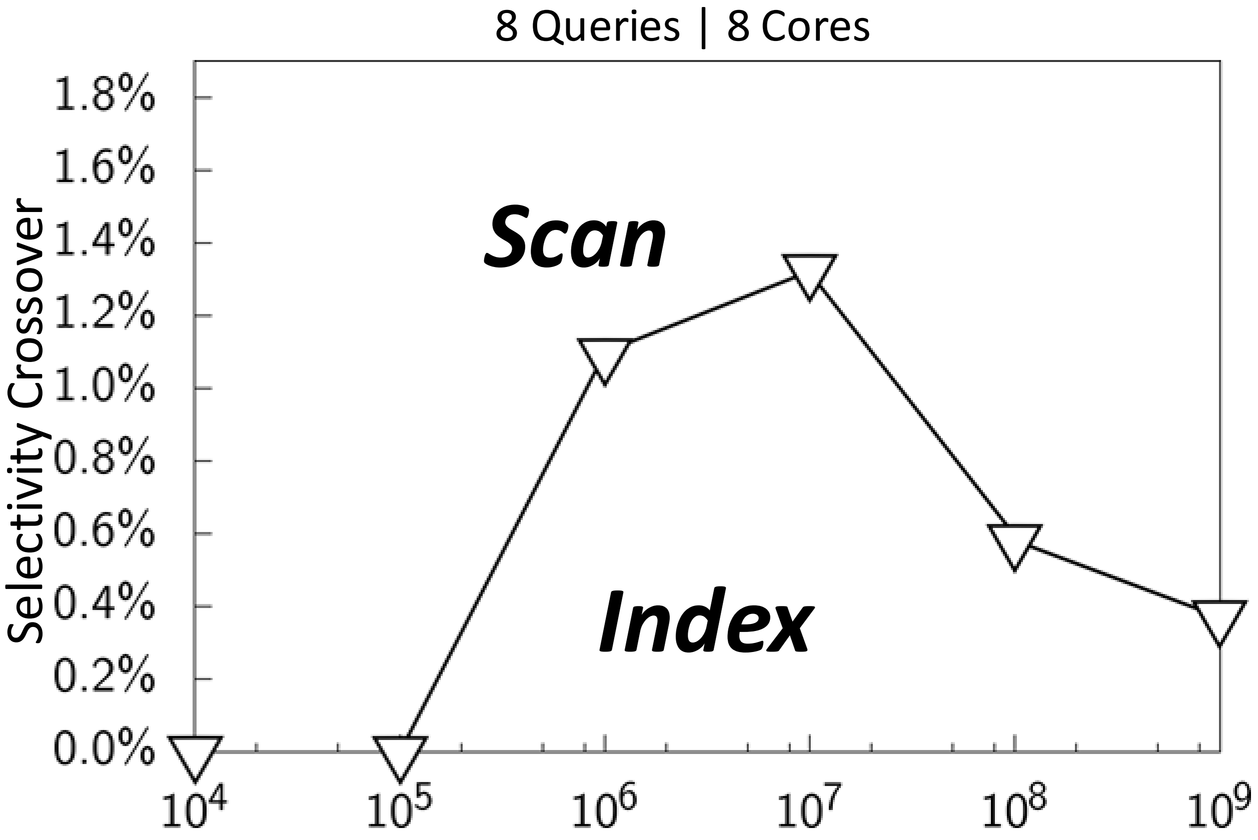


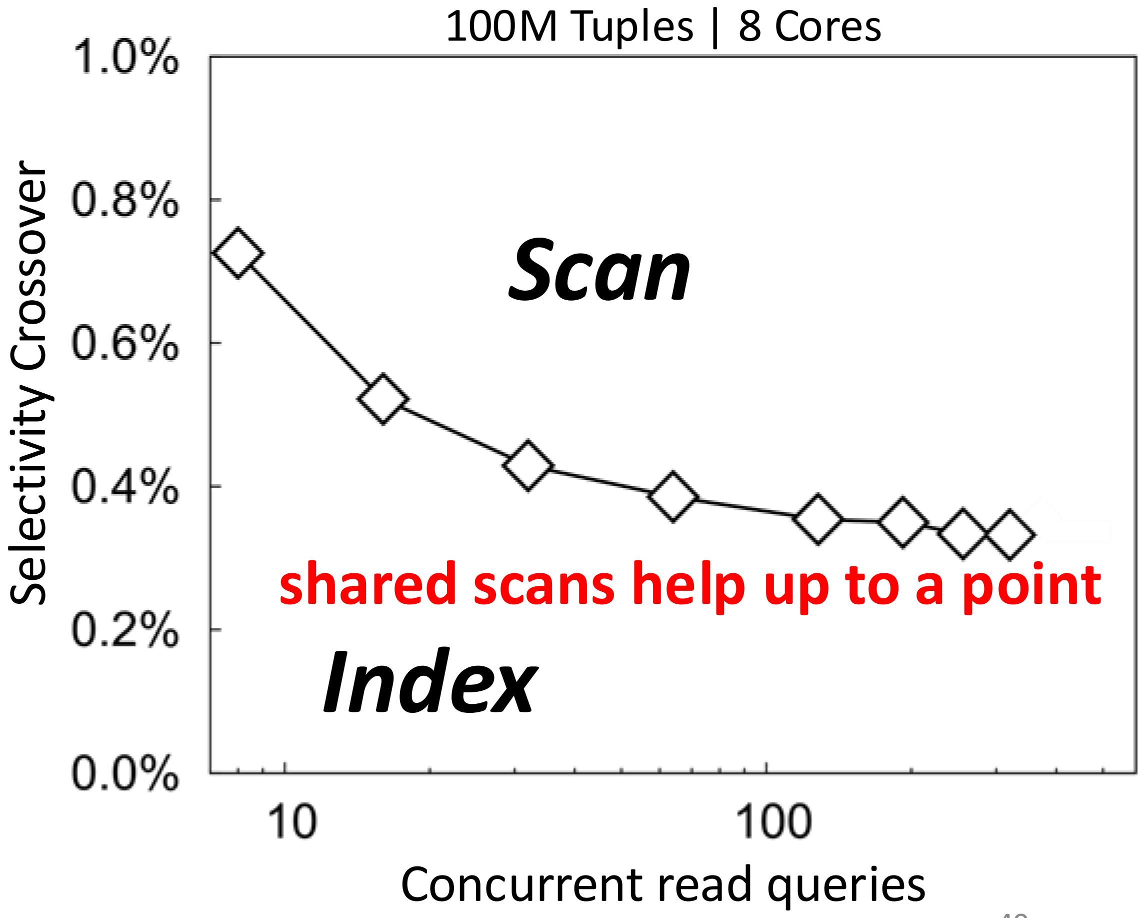


large data → scan **high concurrency matters → scan**

new access path selection

Experiments

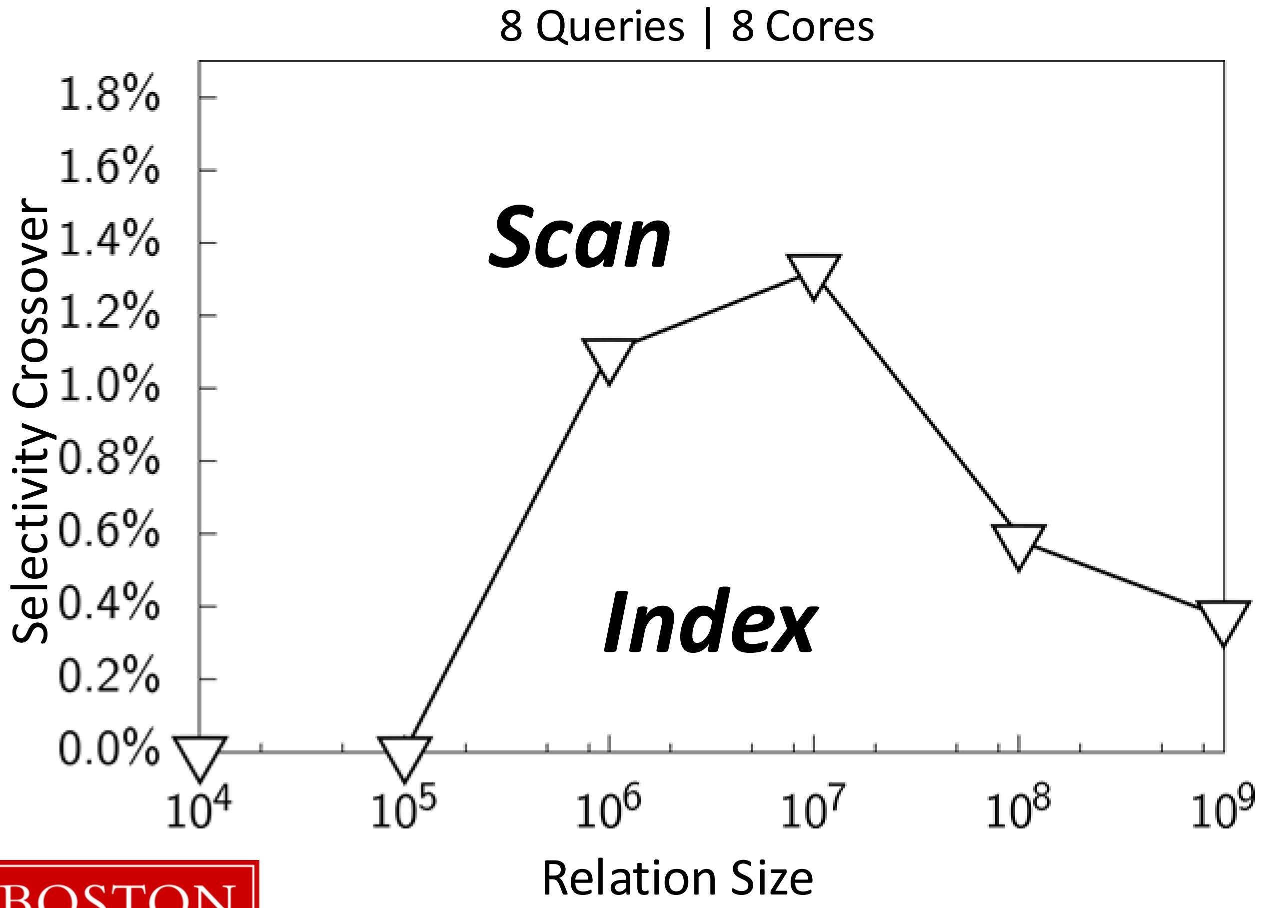


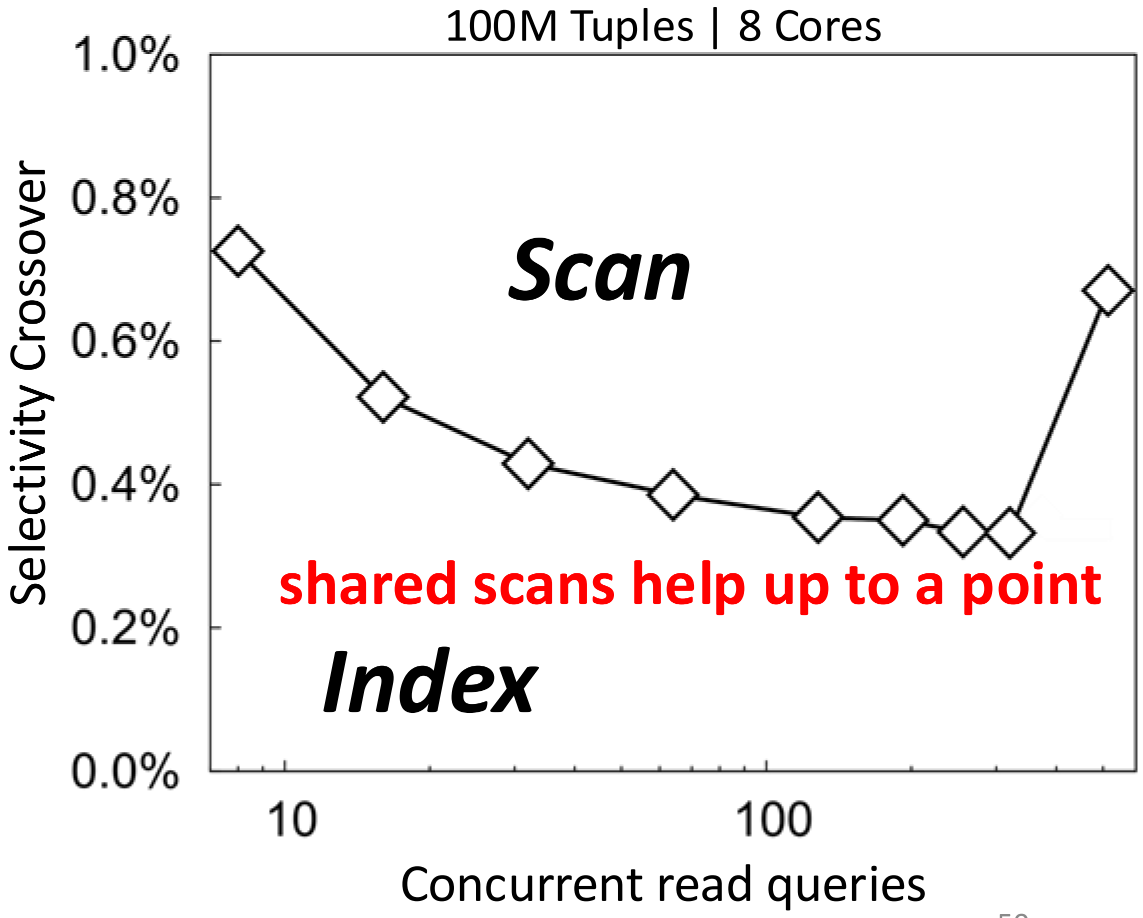


BOSTON UNIVERSITY

new access path selection

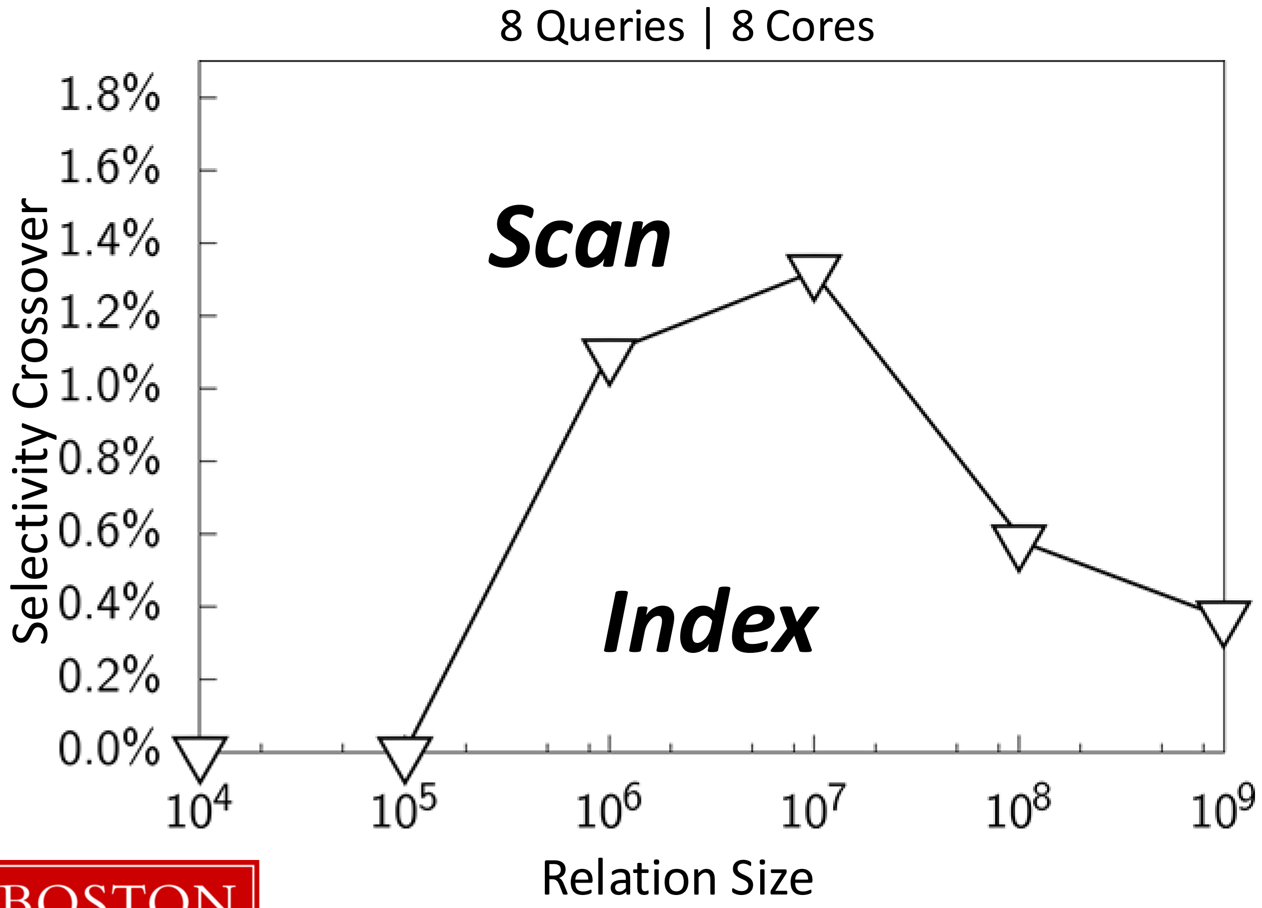
Experiments

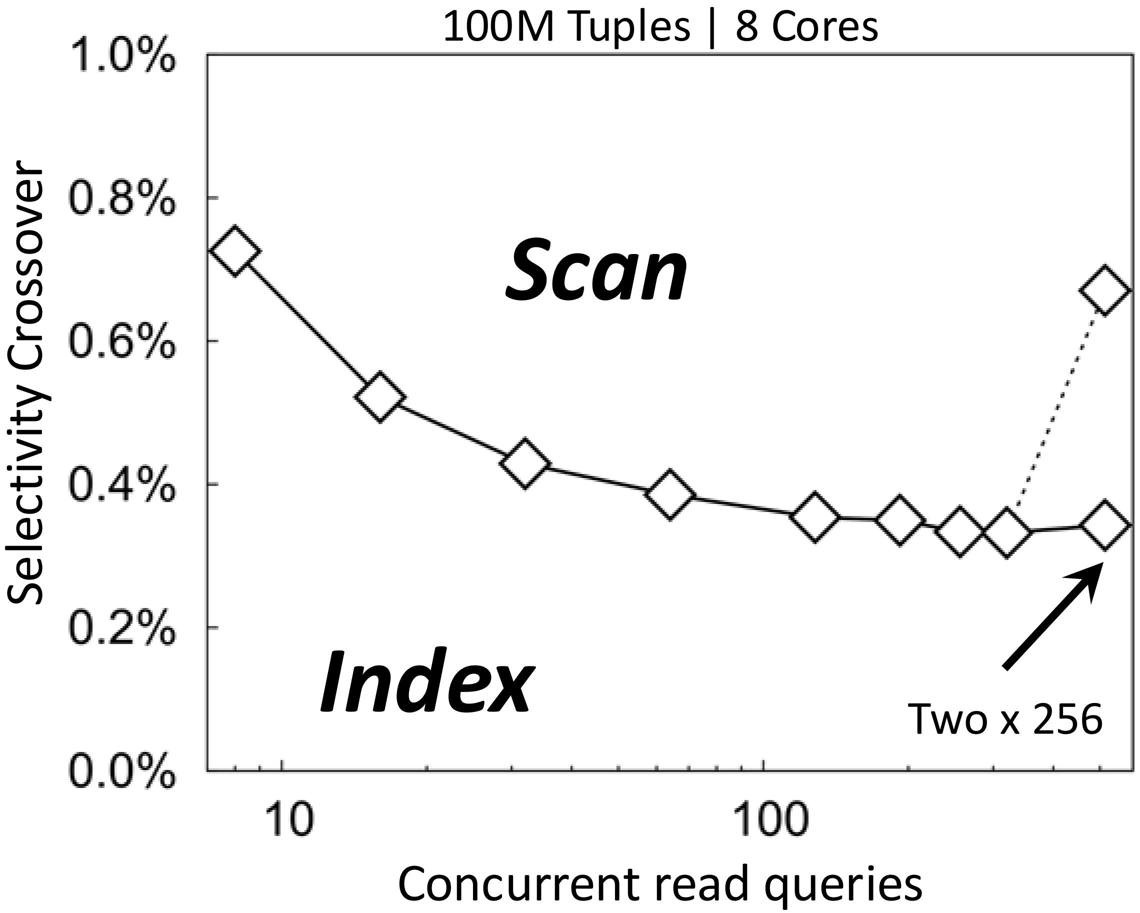




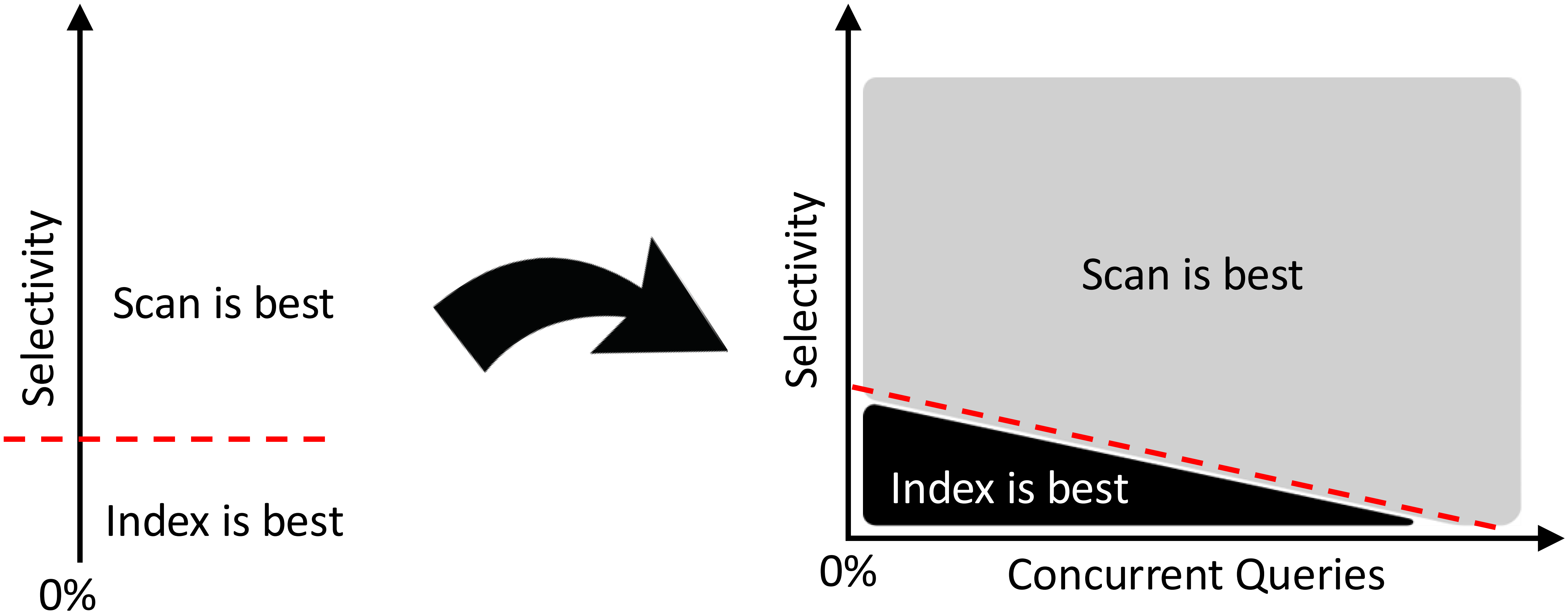
new access path selection

Experiments

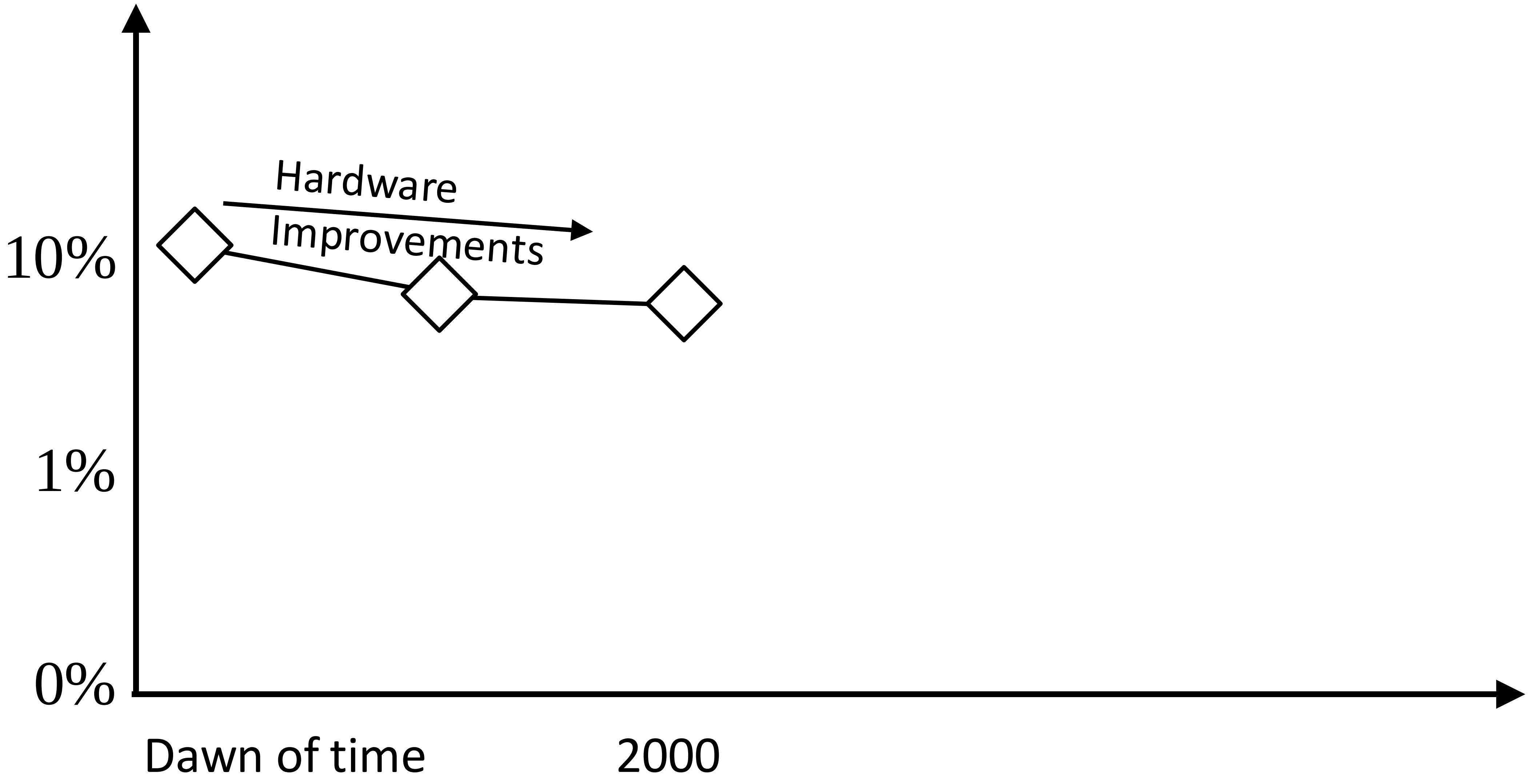


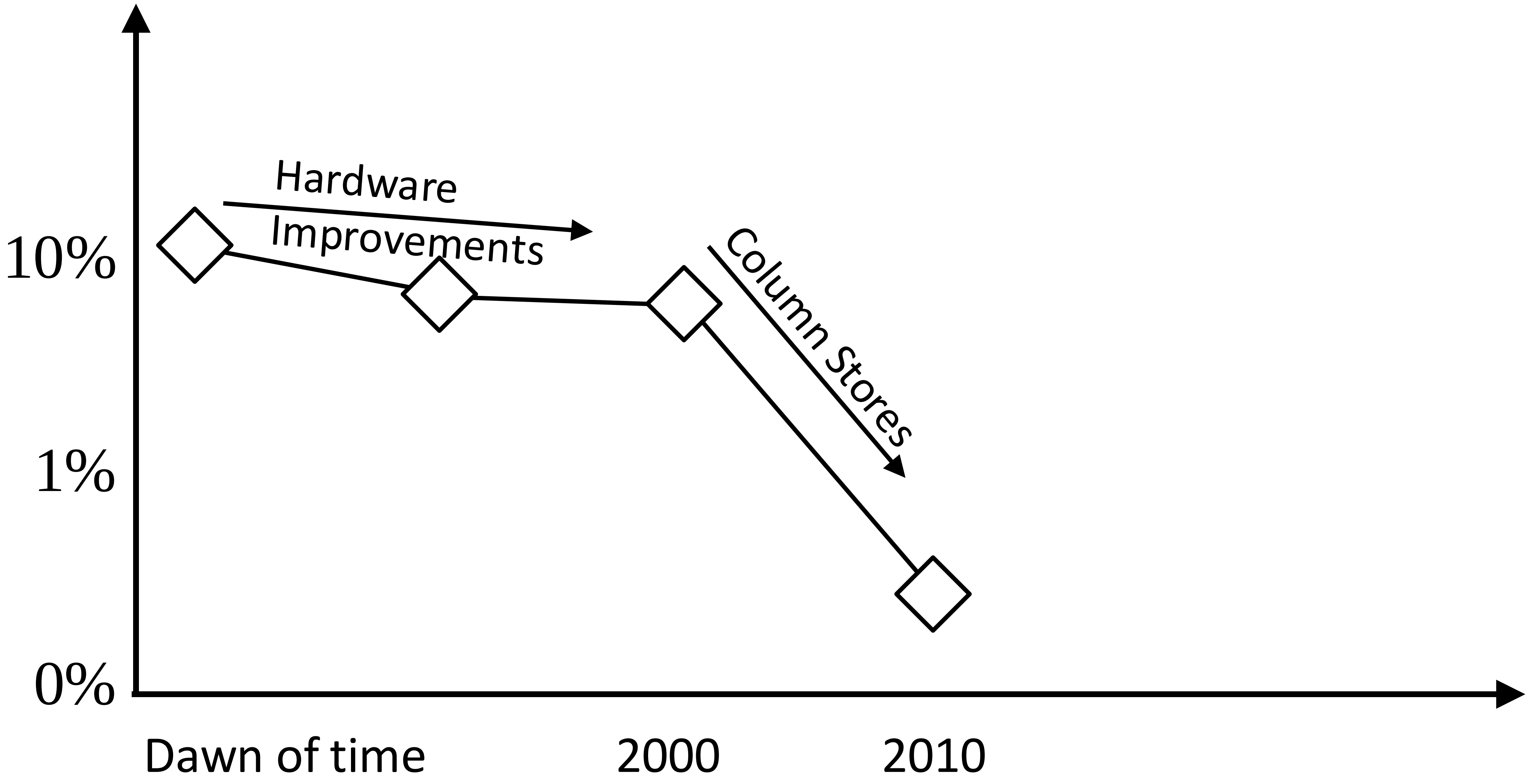


new access path selection

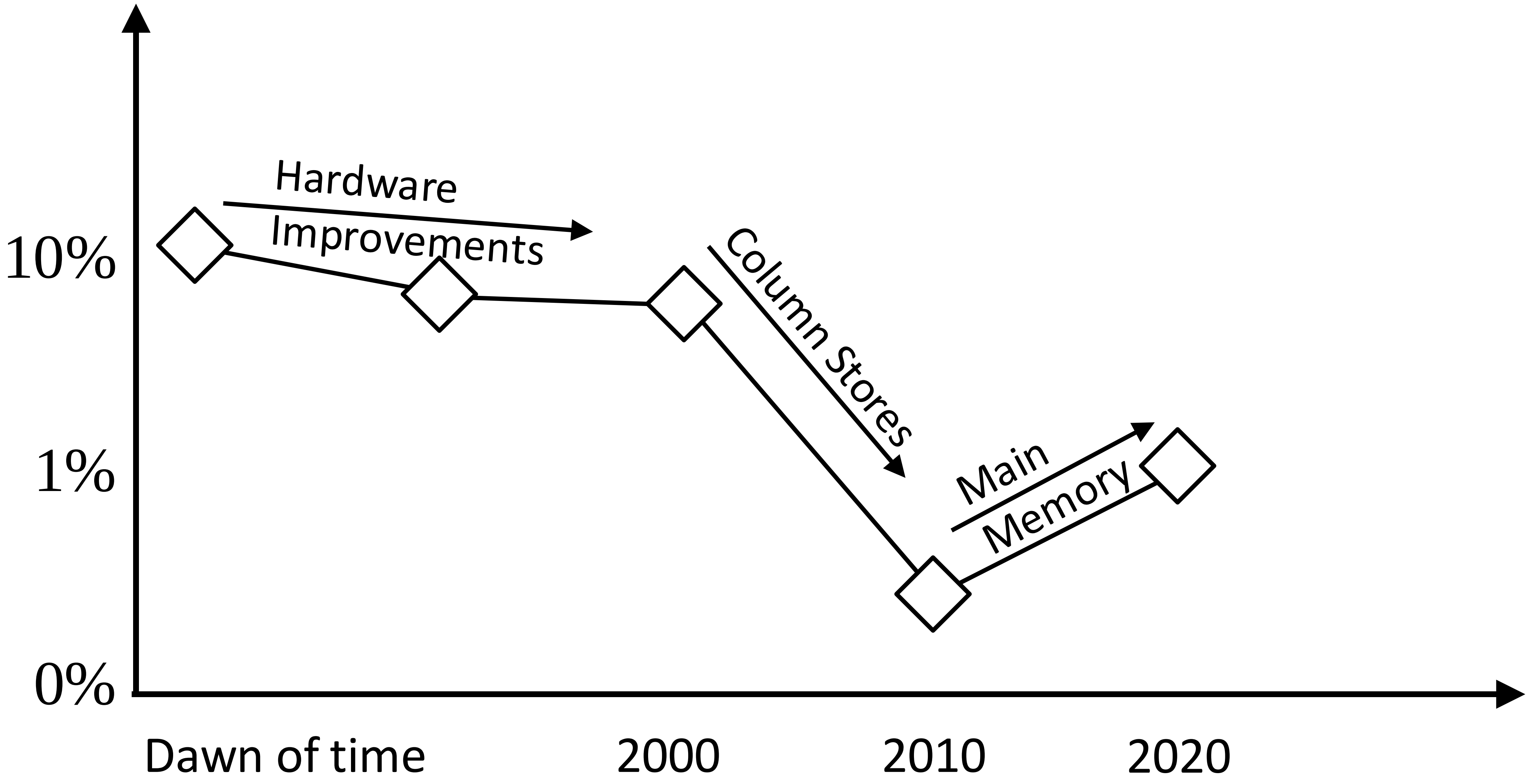


BOSTON UNIVERSITY

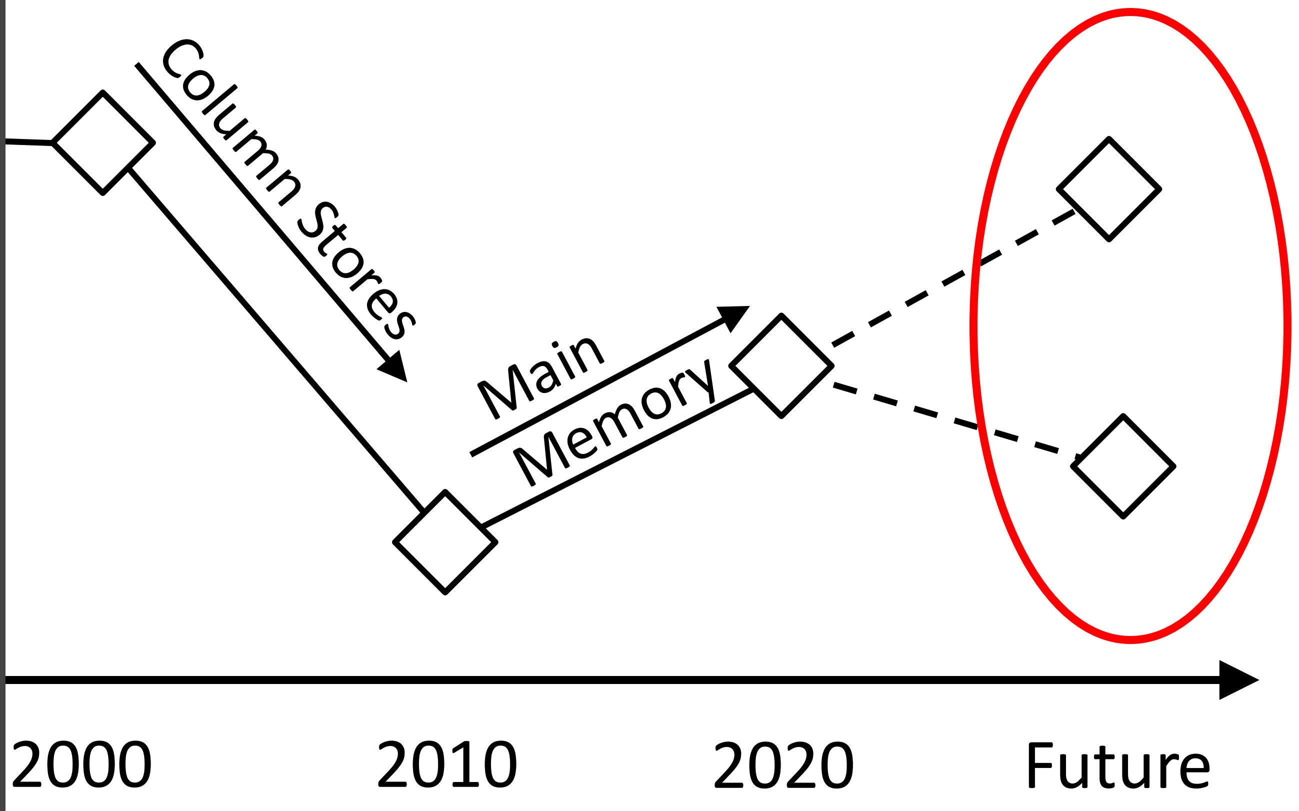

Hardware
Improvements
10%
1%
0%
Dawn of time
2000



BOSTON UNIVERSITY


Hardware
Improvements
Column Stores
Main
Memory
10%
1%
0%
Dawn of time
2000
2010
2020

“What-if” questions



data structures *design dimensions and their values*

global data organization

global search algorithm

metadata for searching

local data organization & search algorithm

modification policy

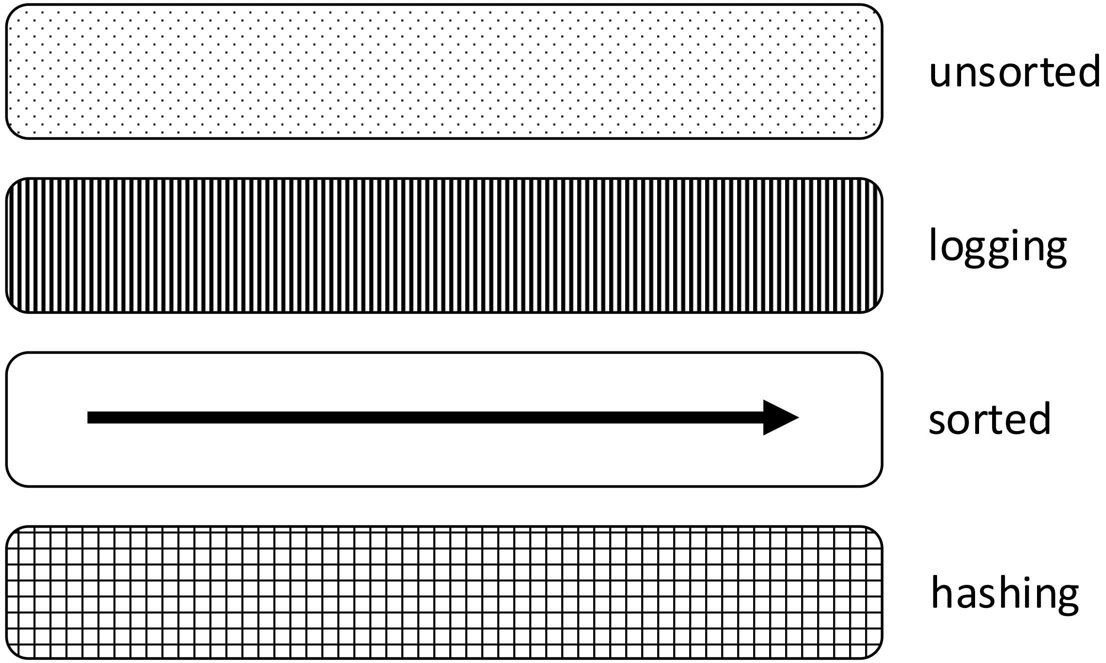
batching via buffering

adaptivity

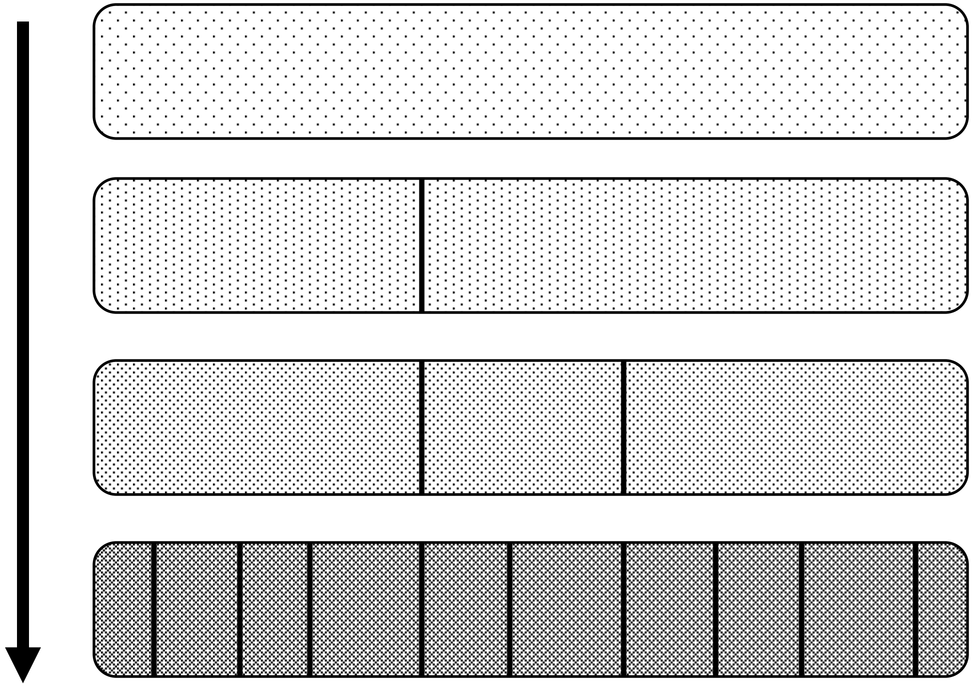
BOSTON UNIVERSITY

local data organization

decision per partition



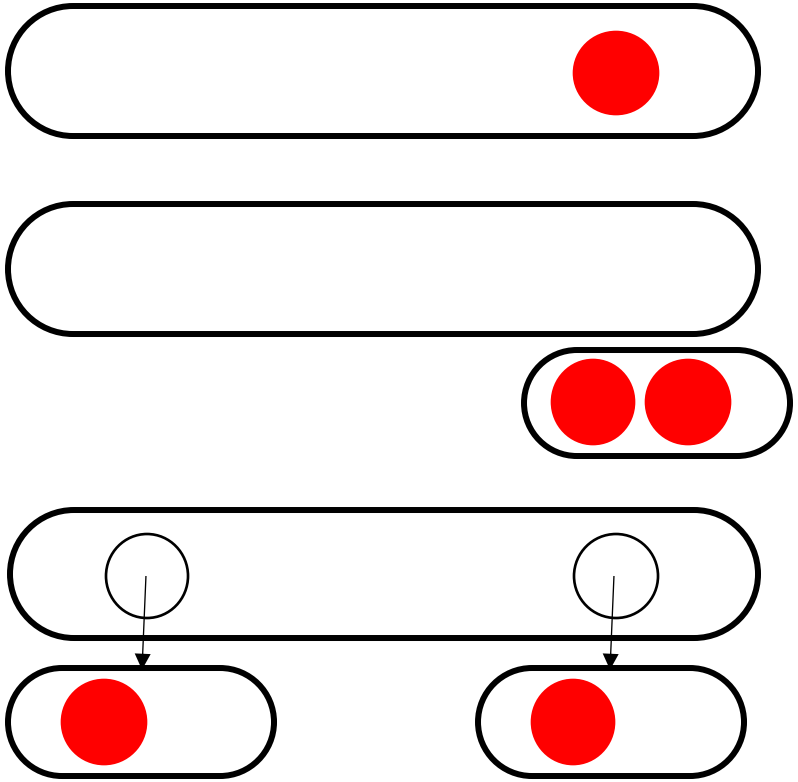
cracking



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

BOSTON UNIVERSITY

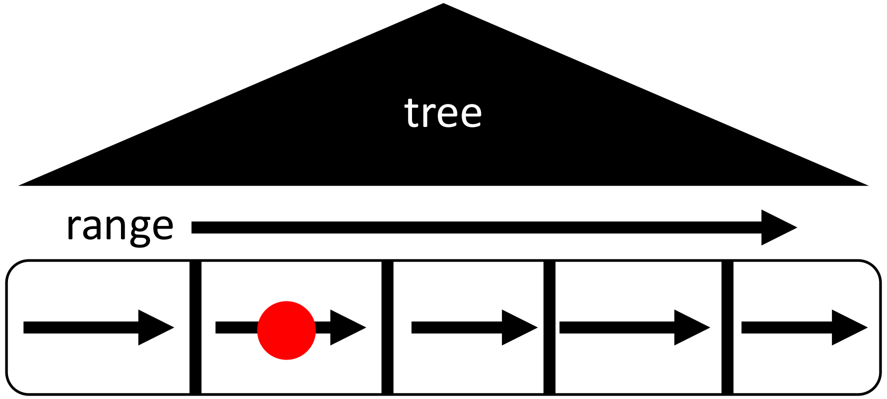
how to break down ***popular designs***
to those design decisions?

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, modifications, and some scans



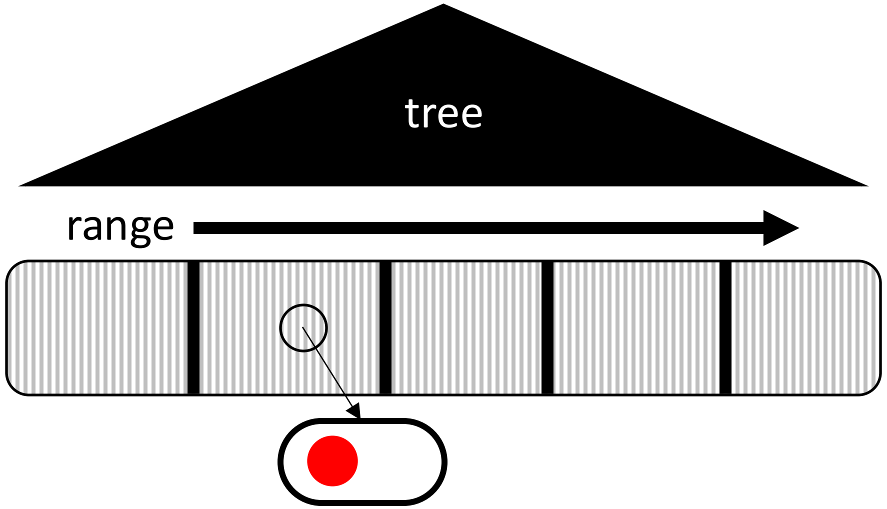
BOSTON UNIVERSITY

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



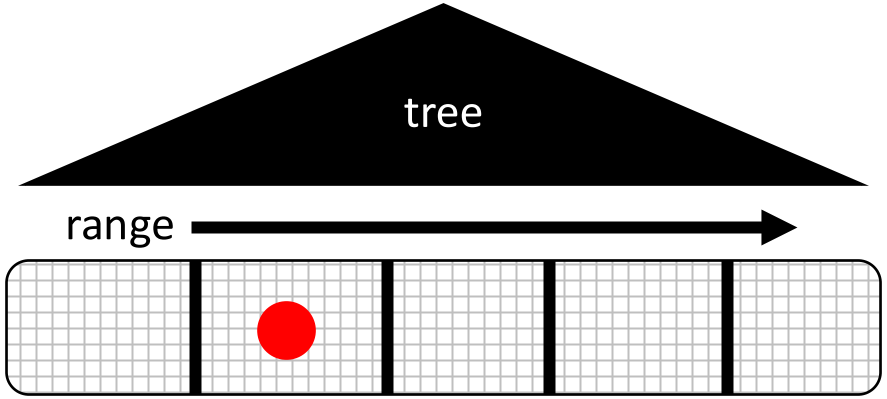
BOSTON UNIVERSITY

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 <u>short range</u> queries



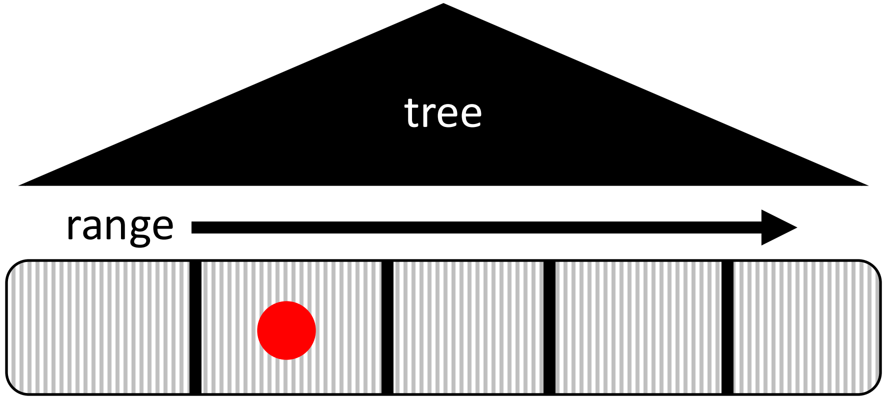
BOSTON UNIVERSITY

Bloom-filter tree

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

Workload?

mixed workload, without short range queries



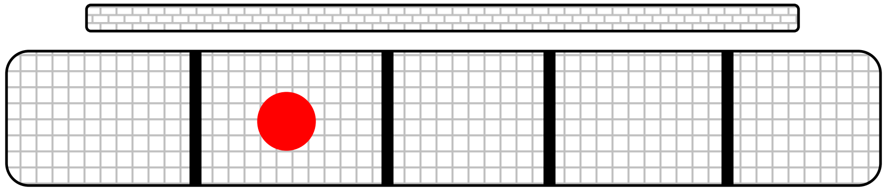
BOSTON UNIVERSITY

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



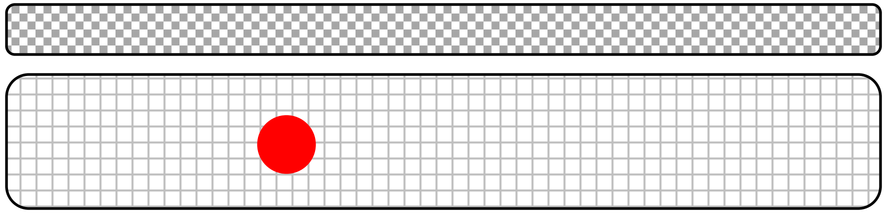
BOSTON UNIVERSITY

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



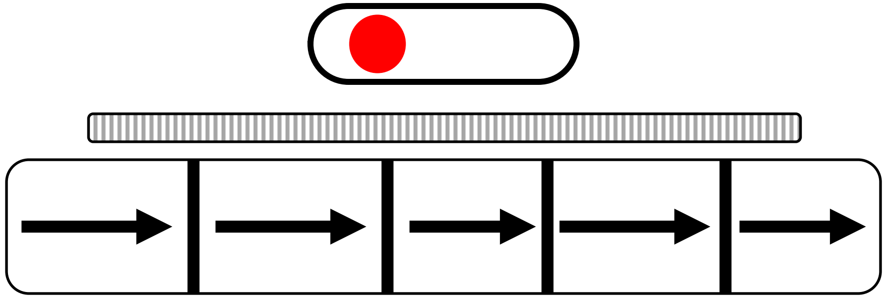
BOSTON UNIVERSITY

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



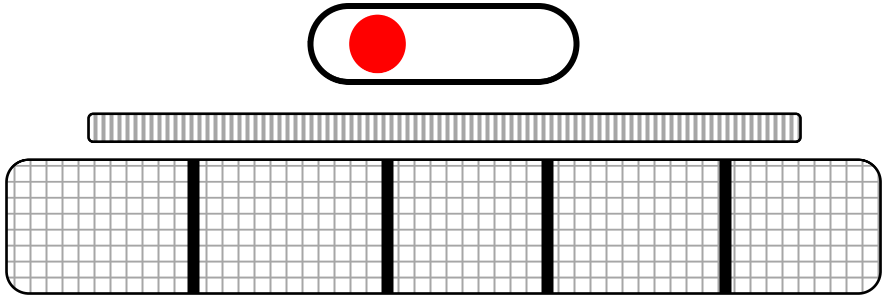
BOSTON UNIVERSITY

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

CS 561: Data Systems Architectures

The *design space* of data structures

Prof. Manos Athanassoulis

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