CS 561: ***Data Systems Architecture***

Class 4

Systems & Research Projects

Dr. Subhadeep Sarkar

BOSTON UNIVERSITY

DiSC lab

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

Updates: Logistics

Some students in the waitlist have been added (deadline: **tomorrow**).

Order for Classes 4 through 7 has been changed.

First **technical question** is due on **02/07**.

First **review** is due on **02/14**.

Deadline for **Project 0** extended till **02/05**.

Reminder: Presentations

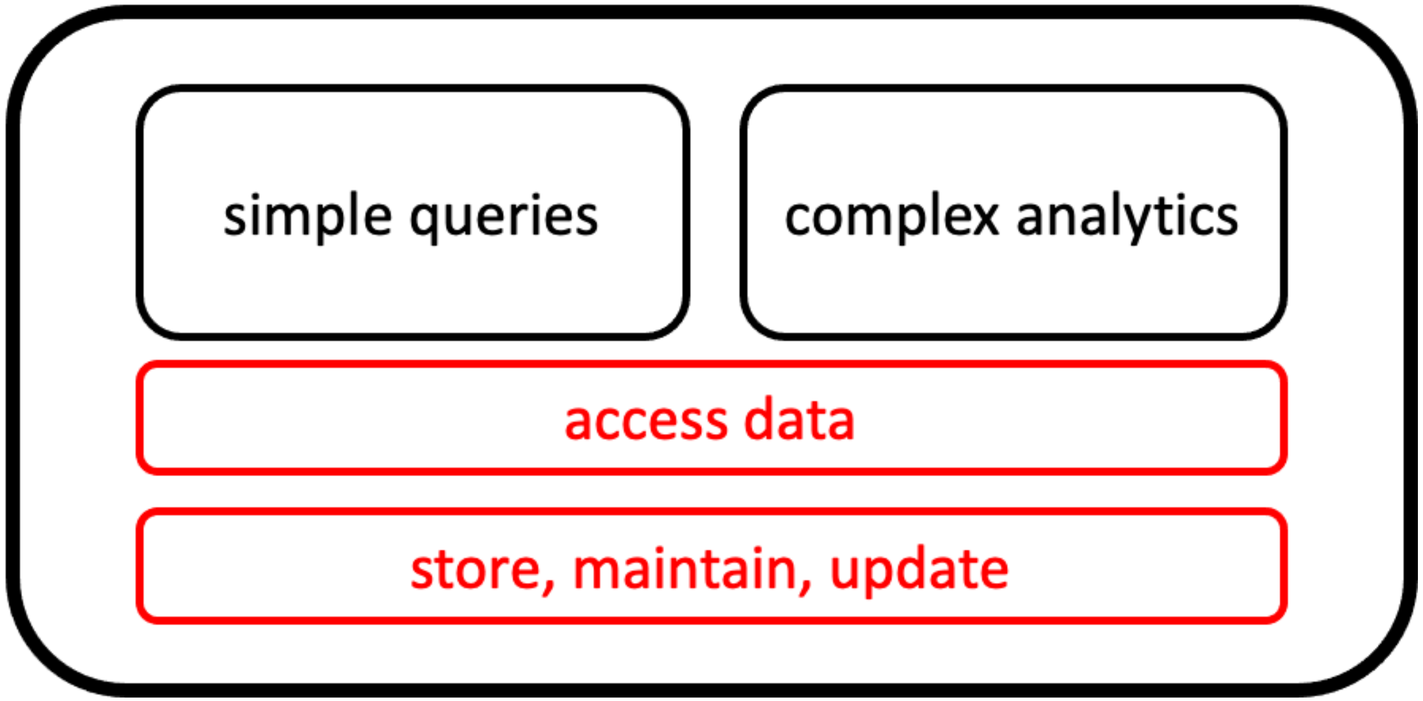
The first **student presentation** is in two weeks (on **02/14**)!

If you haven't done already, **select the paper** you will work on for your **presentation** (groups of 3-4 students)

https://docs.google.com/document/d/1Lr-pwN7YYjL9Qk7riVvABwZyOhvVmAZsCjToghIYs8E/

A week before the presentation, discuss the slides with me in OH.

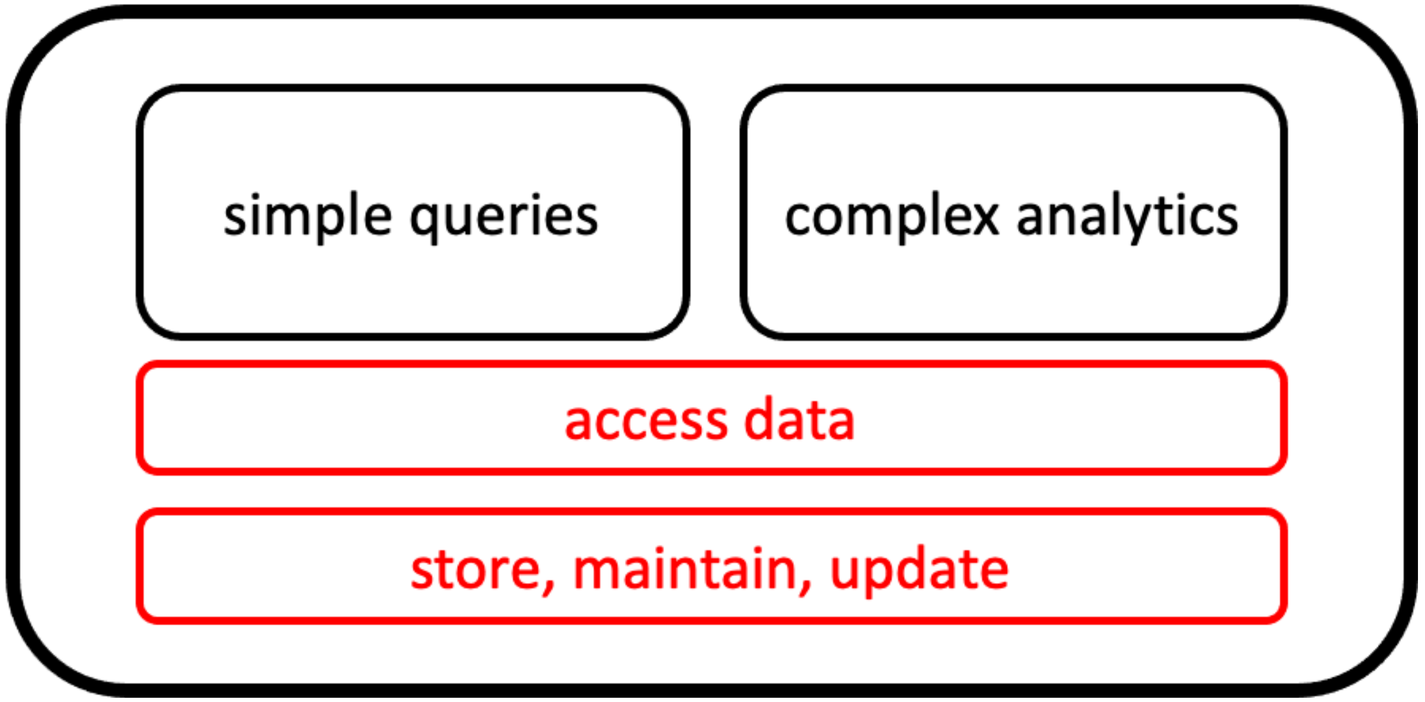
Data Systems



access methods

algorithms and **data structures**
for organizing and accessing data

Data Systems



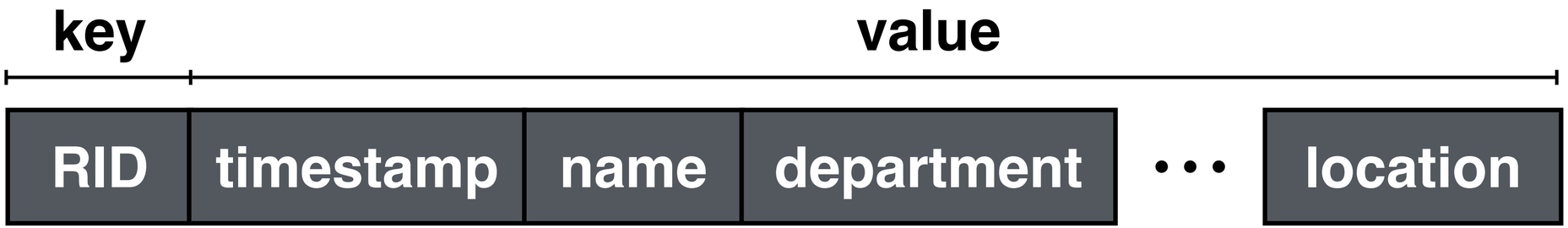
- how to **store** data?
- how to **access** data?
- how to **update** data?

At the core of our Research/Systems Projects!

Projects

Theme: **NoSQL key-value stores**

key-value pairs



Projects

Theme: **NoSQL key-value stores**

key-value pairs

How general is a key-value store?

can we store relational data?

yes! {<primary_key>,<rest_of_the_row>}

example: { **student_id**, { **name**, **loginID**, **yob**, **gpa** } }

what is the caveat?

how to index these attributes?

index: { **loginID**, { **student_id** } }

How to efficiently code if entry size is variable?

How to use a key-value store?

The key-value API

insert: put(k,v)

PQ: {v} = get(k) {v_1,v_2,...} = get(k) {v_1,v_2,...} = get_set(k_1,k_2,...)

RQ: {v_1,v_2,...} = get_range(k_{min},k_{max}) {v_1,v_2,...} = full_scan()

count: c = count(k_{min},k_{max})

delete: delete(k) delete(k_{min},k_{max})

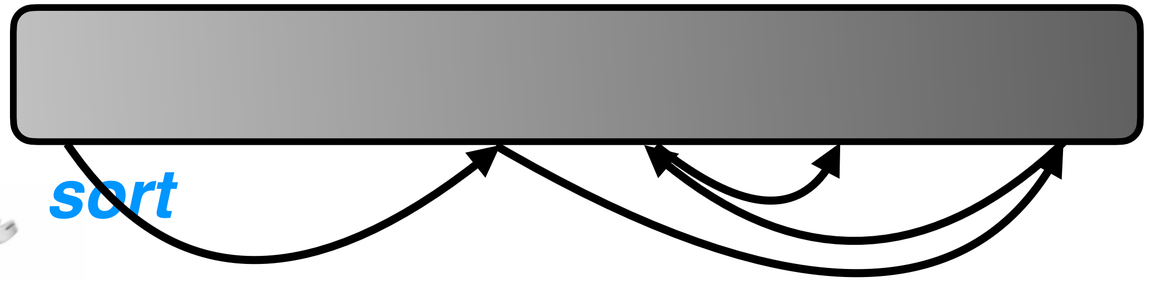
update: update(k,v_{new}) **not very different from put**

How to build a key-value store?

if we have only ***put*** operations

append

if we mostly have ***point get*** operations



if we mostly have ***range get*** operations

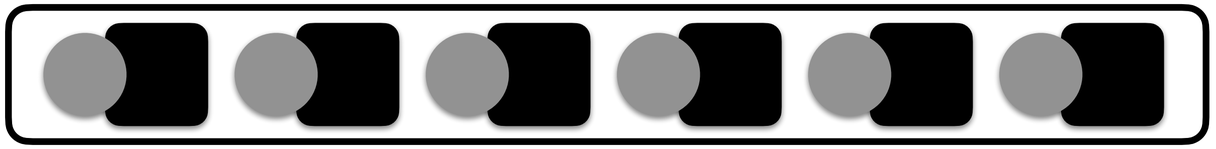
sort, find the min, scan

if we mostly have ***full scan*** operations

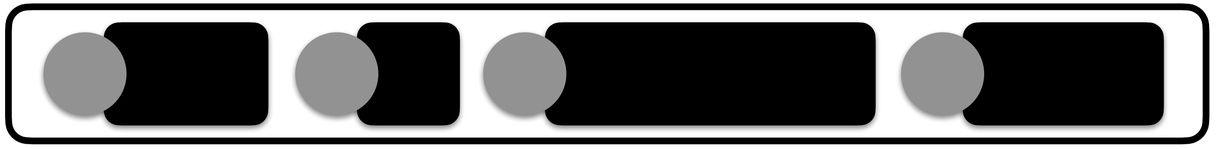
don't sort

What about **variable entry size**?

fixed entry size

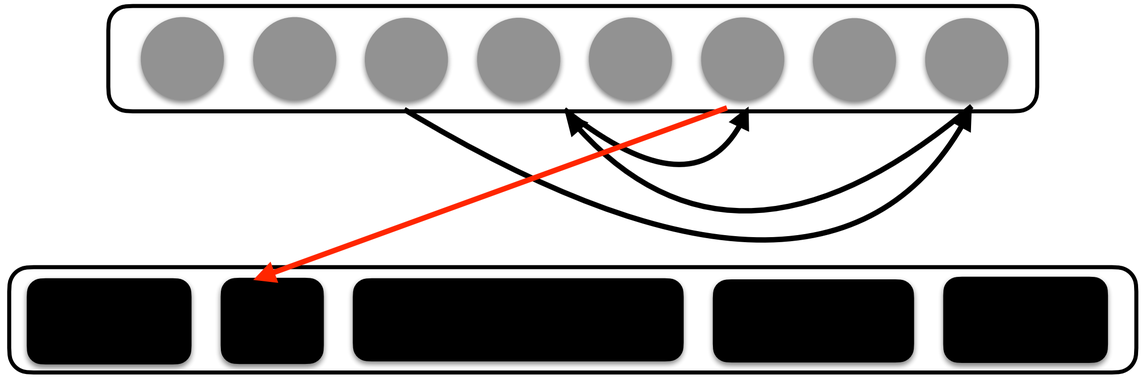


variable entry size



what can we do?

- efficient key storage
- compiler-friendly
- need not sort values



is my lunch free ?

BOSTON UNIVERSITY

- range queries?
- locality?
- code complexity?

Log-**S**tructured **M**erge-tree

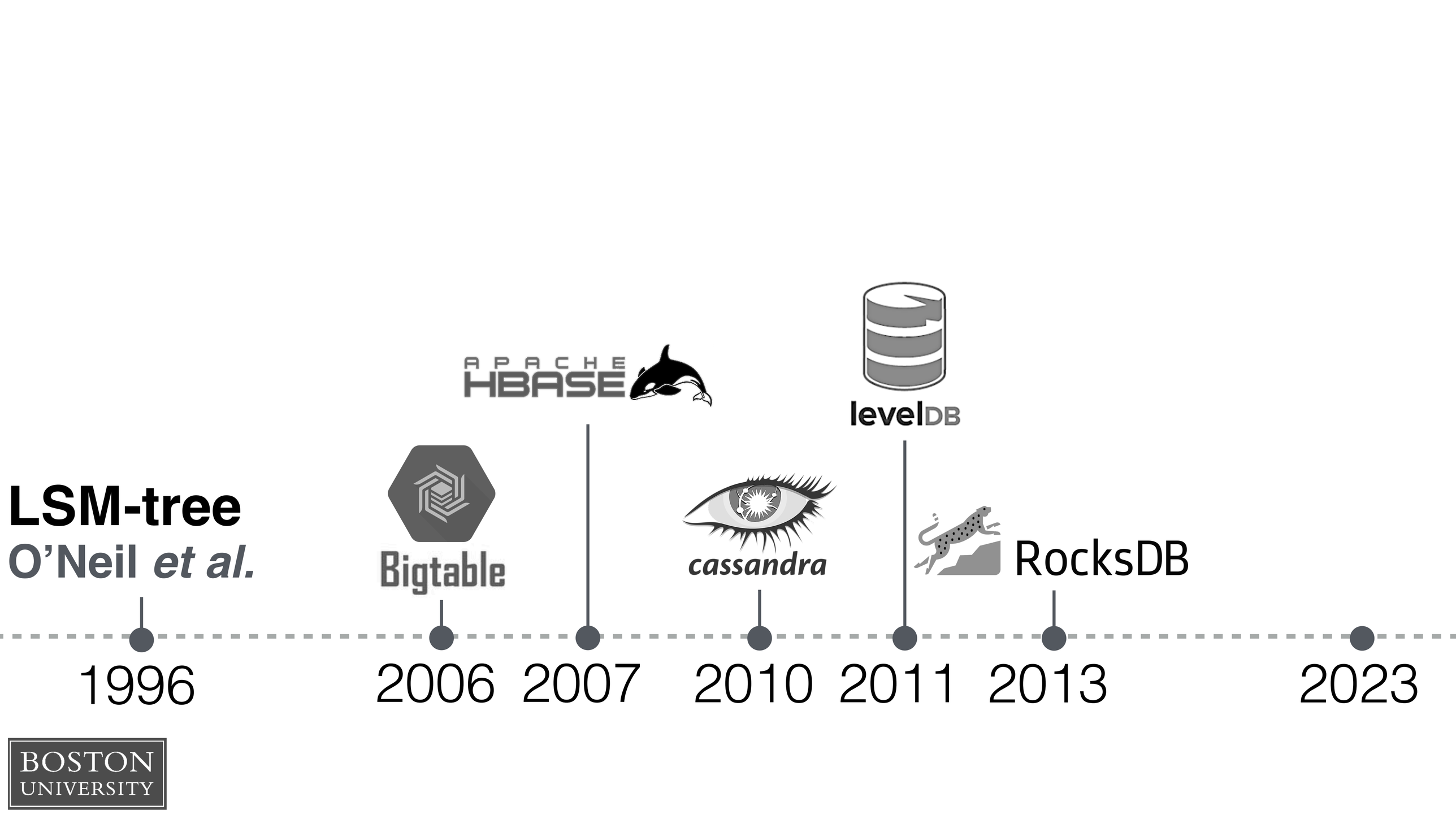
BOSTON
UNIVERSITY

LSM-tree

The Log-Structured Merge-Tree (LSM-Tree)

1996

Patrick O'Neil[1], Edward Cheng[2]
Dieter Gawlick[3], Elizabeth O'Neil[1]
To be published: Acta Informatica


LSM-tree
O'Neil et al.
Bigtable
APACHE HBASE
levelDB
cassandra
RocksDB
1996
2006
2007
2010
2011
2013
2023

LSM-tree

NoSQL

relational

time-series

2023

LSM-tree

NoSQL

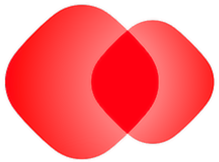

RocksDB


levelDB


SCYLLA


DynamoDB


cassandra


tarantool


Bigtable


APACHE HBASE


riak


accumulo


SQLite

relational


influxdb

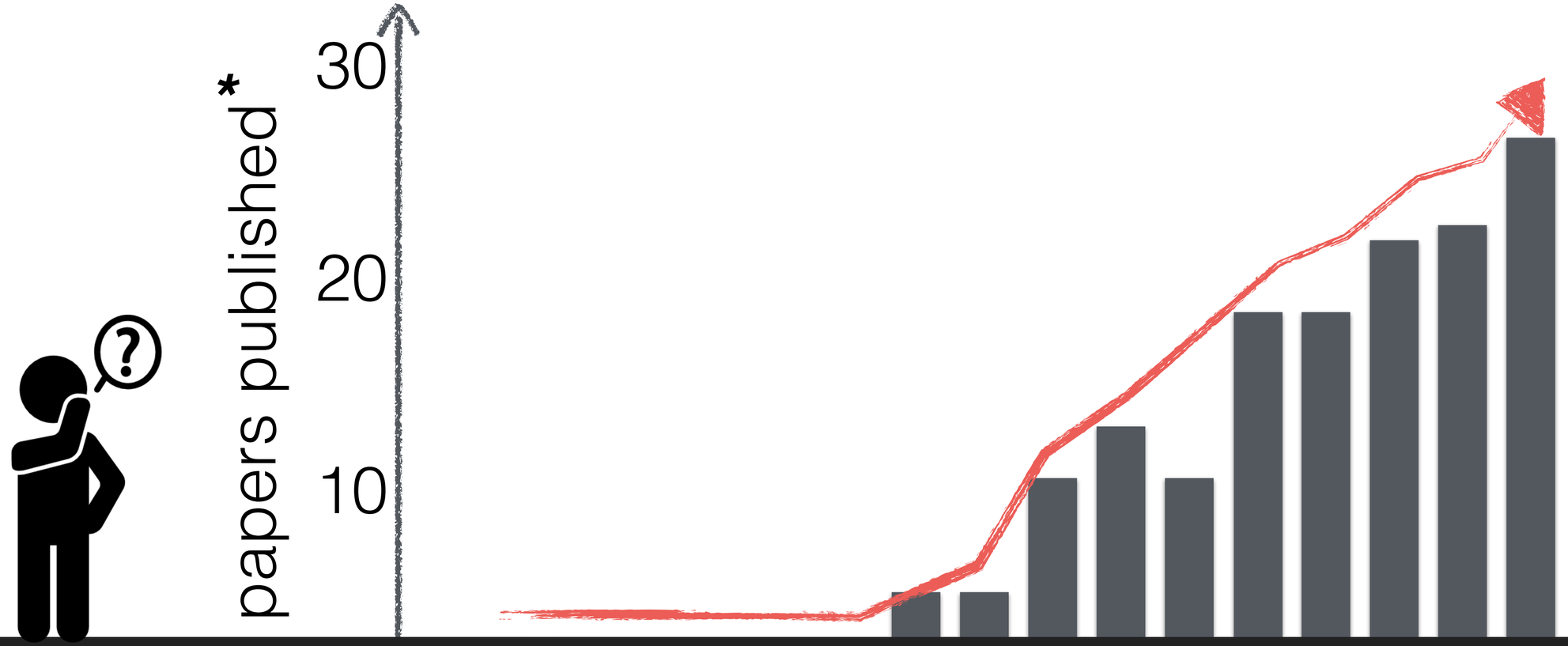

QuasarDB

time-series

2023

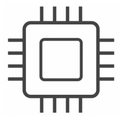
BOSTON UNIVERSITY

Research Trend

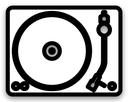


So, how does an LSM-tree look?

LSM **Basics**



buffer



level 1

level 2

level 3

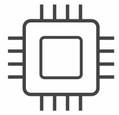
size ratio: T

Great! But, how does it work?

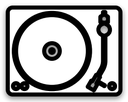
Buffering ingestion

put(6)

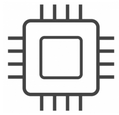
put(2)



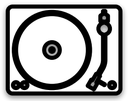
buffer



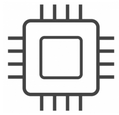
Buffering ingestion



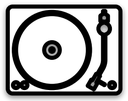
buffer | 2 | 6 | 1 | 4 |



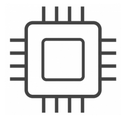
Buffering ingestion



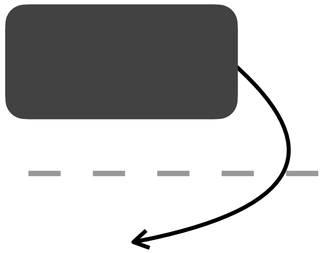
buffer | 1 | 2 | 4 | 6 |

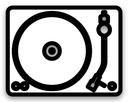


Buffering ingestion



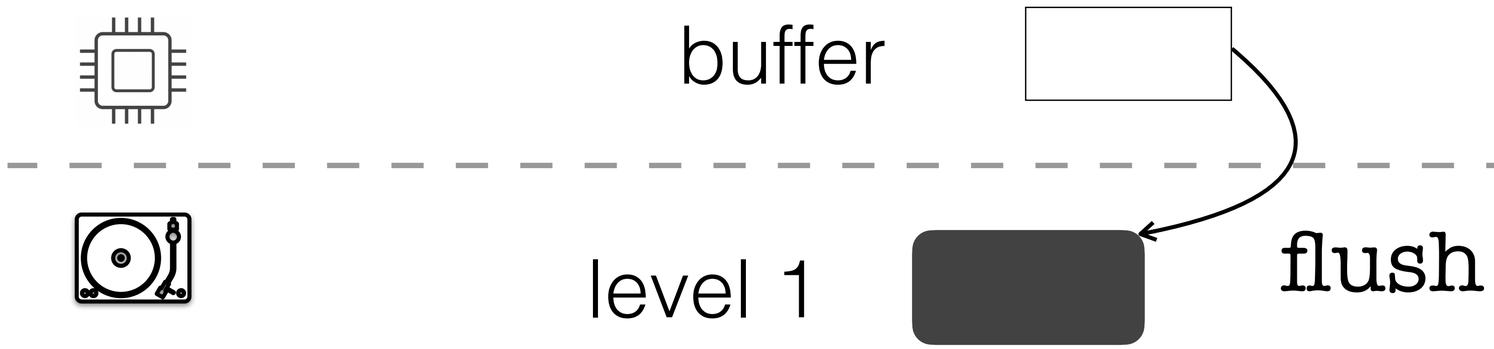
buffer





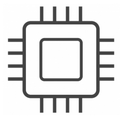
Buffering ingestion

- low ingestion cost
- fast temporal reads

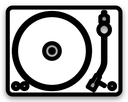


Immutable files on storage

- compact storage
- good ingestion throughput

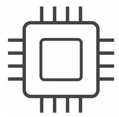


buffer

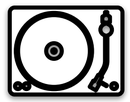


level 1

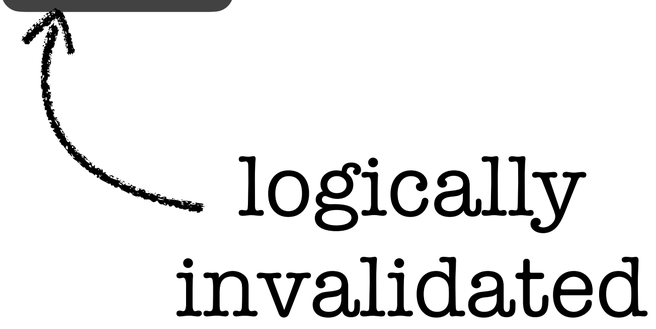
How do we update data?

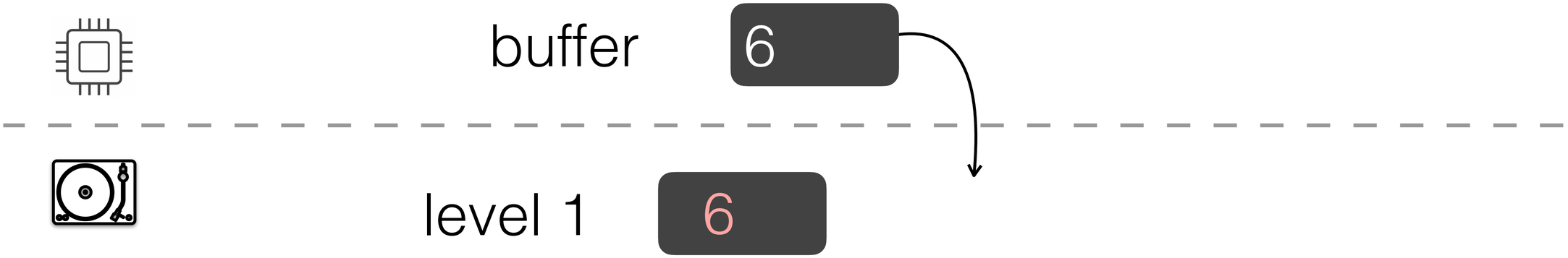


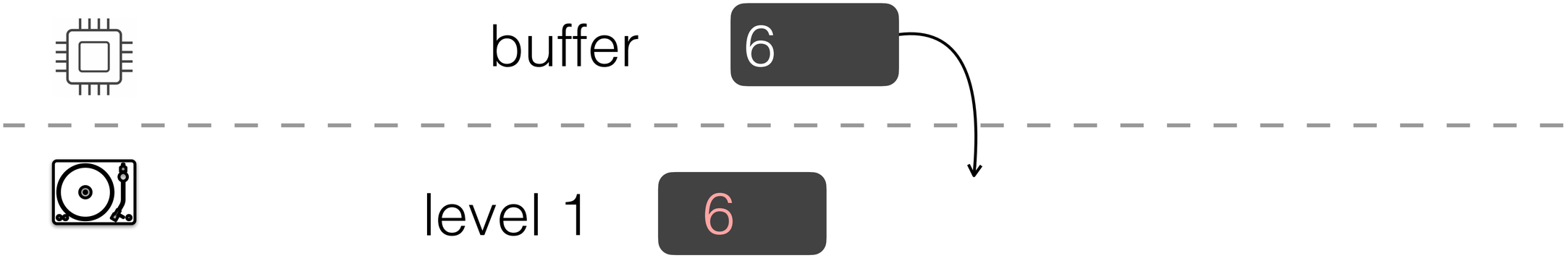
buffer



level 1

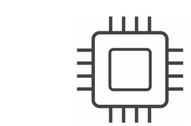




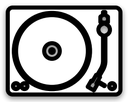


Out-of-place updates

- fast ingestion
- space amplification
- slow reads

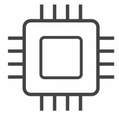


buffer

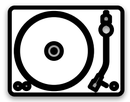


level 1

How do we reduce this space amplification?



buffer

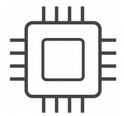


level 1

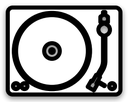
6

compaction

P: pages in buffer
B: entries/page
L: #levels
T: size ratio



buffer $P \cdot B$



level 1 $P \cdot B \cdot T$

level 2 $P \cdot B \cdot T^2$

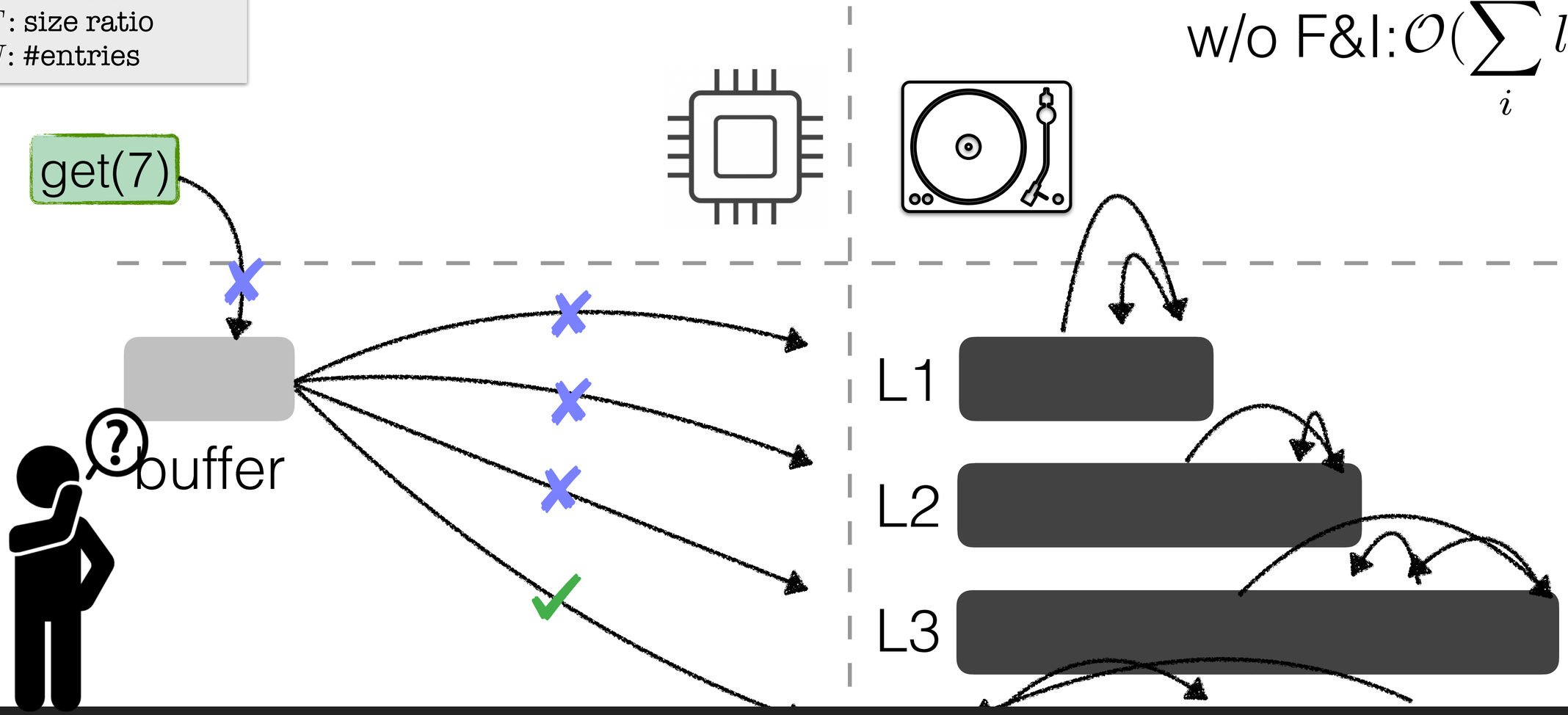
level 3 $P \cdot B \cdot T^3$

How about queries?

P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries

Cost analysis

w/o F&I: $\mathcal{O}(\sum_i log_2(P \cdot T^i))$



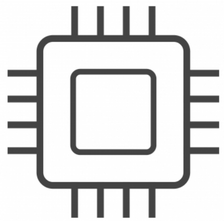
Can we do better?

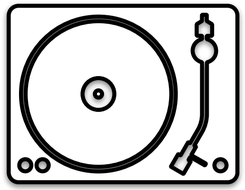
P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries

Cost analysis

w/o F&I: $\mathcal{O}(\sum_i log_2(P \cdot T^i))$

w/ index: $\mathcal{O}(log_T N/(P \cdot B))$





get(7)

buffer

L1

L2

L3

L4

7

fence pointers

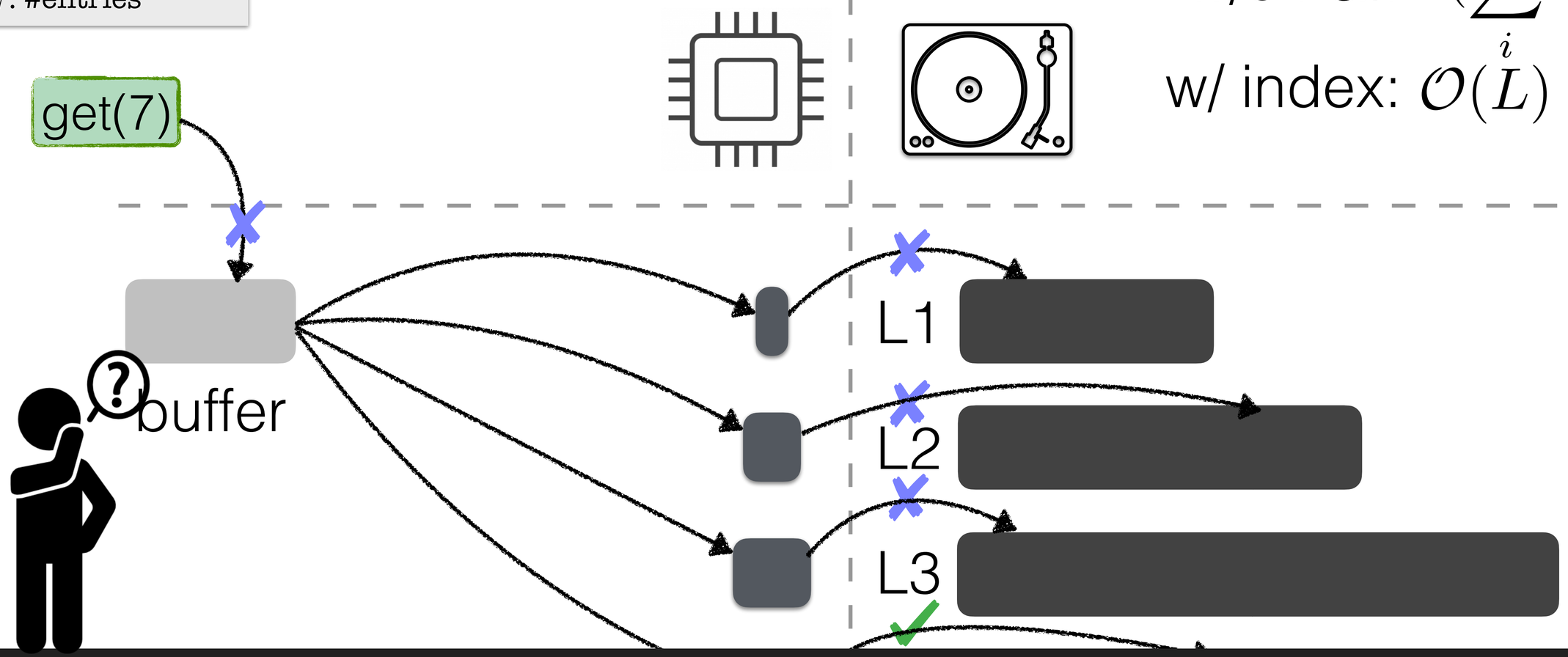
BOSTON UNIVERSITY

P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries

Cost analysis

w/o F&I: $\mathcal{O}(\sum_{i} log_2(P \cdot T^i))$

w/ index: $\mathcal{O}(L)$



Still expensive! Can't we do any better?

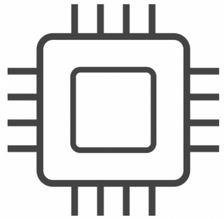
P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries
ϕ : FPR of BF

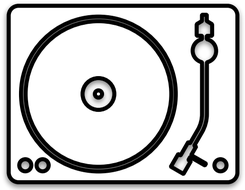
Cost analysis

w/o F&I: $\mathcal{O}(\sum_{i} log_2(P \cdot T^i))$

w/ index: $\mathcal{O}(L)$

w F&I: $\mathcal{O}(\phi \cdot L)$





get(7)

buffer

L1

L2

L3

L4

7

$\phi = e^{-\frac{m_{BF}}{N} \cdot ln2^2}$

false positive

true positive

Bloom filters

fence pointers

BOSTON UNIVERSITY

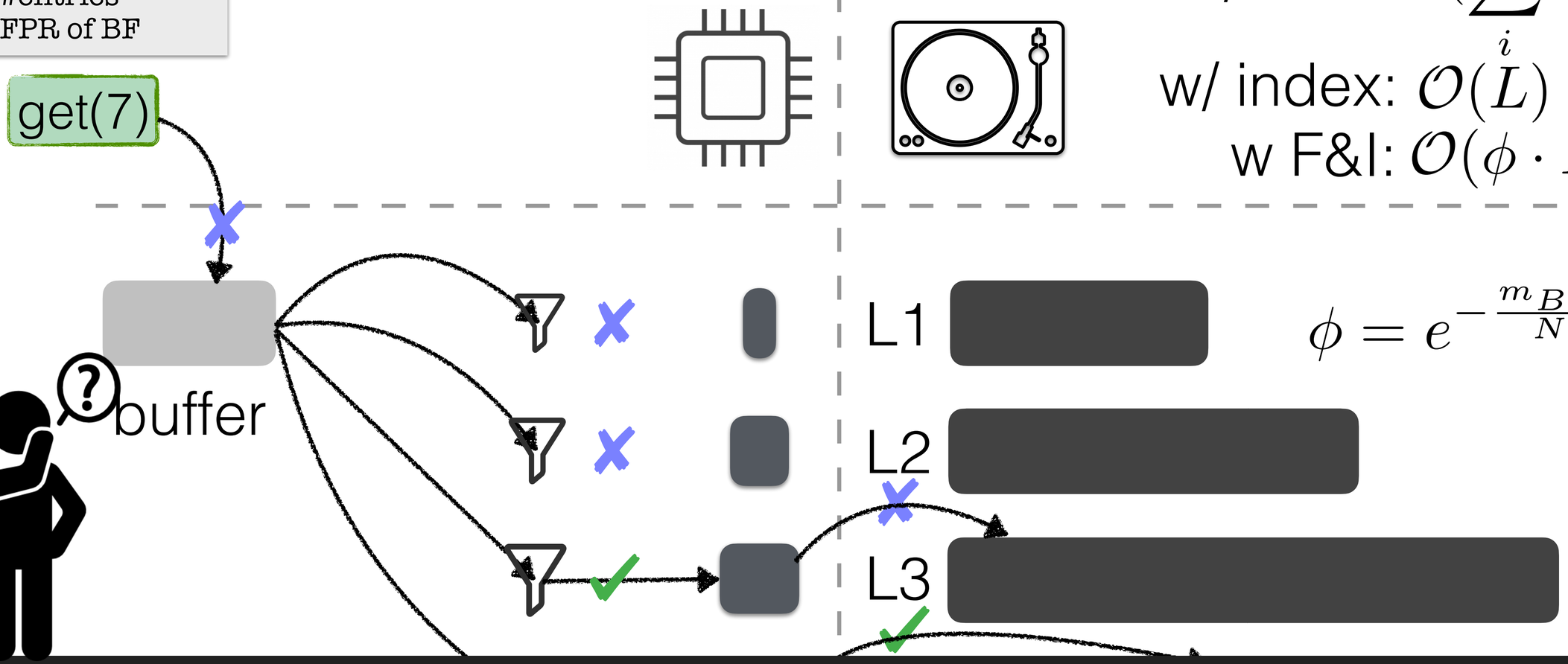
P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries
ϕ : FPR of BF

Cost analysis

w/o F&I: $\mathcal{O}(\sum_{i} log_2(P \cdot T^i))$

w/ index: $\mathcal{O}(L)$

w F&I: $\mathcal{O}(\phi \cdot L)$



$$\phi = e^{-\frac{m_{BF}}{N} \cdot ln2^2} << 1$$

How to manage memory?

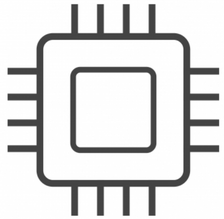
P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries
ϕ : FPR of BF

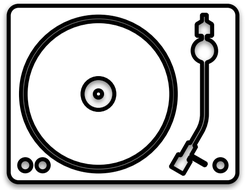
Cost analysis

w/o F&I: $\mathcal{O}(\sum_{i} log_2(P \cdot T^i))$

w/ index: $\mathcal{O}(L)$

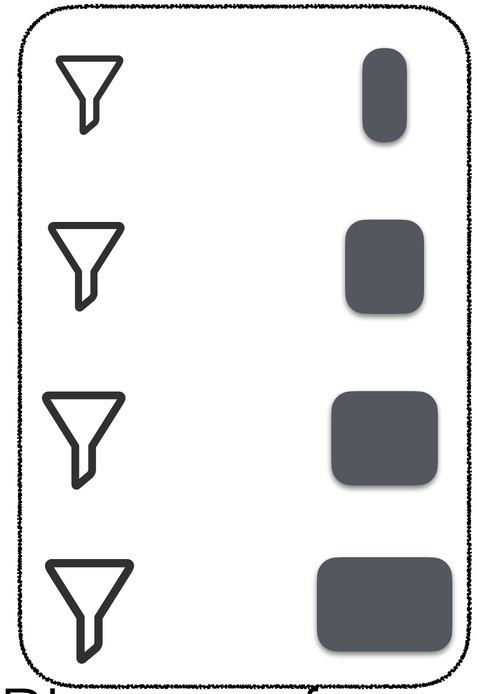
w F&I: $\mathcal{O}(\phi \cdot L)$





buffer

block cache



Bloom filters

fence pointers

L1

L2

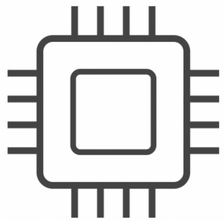
L3

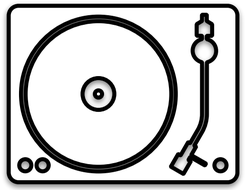
L4

$$\phi = e^{-\frac{m_{BF}}{N} \cdot ln2^2} << 1$$

P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries
ϕ : FPR of BF

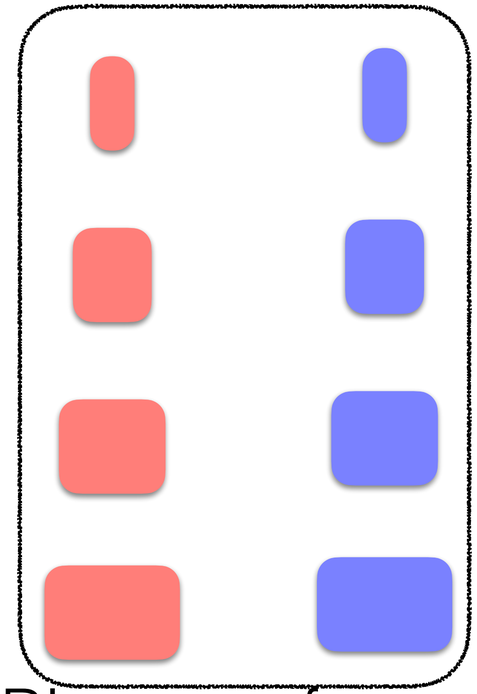
Block Cache





buffer

block cache



Bloom filters

fence pointers

L1

L2

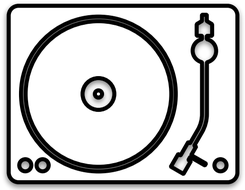
L3

L4

BOSTON UNIVERSITY

P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries
ϕ: FPR of BF

Block Cache

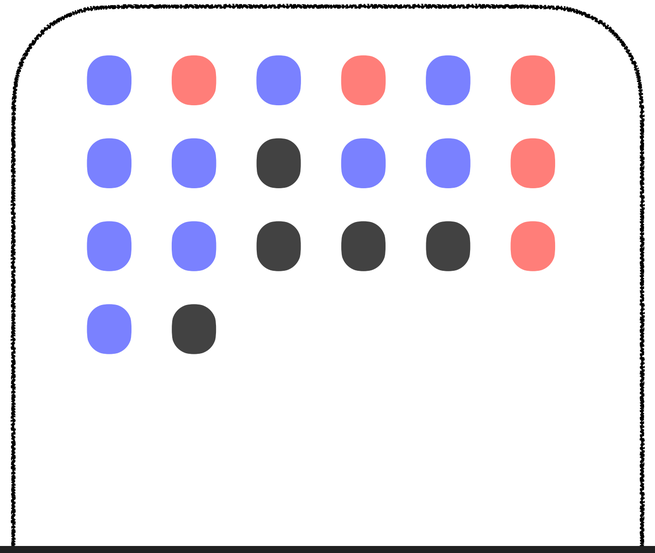


Bloom filters

fence pointers

get(7)

buffer



L1

L2

L3

What about range queries?

P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries
ϕ: FPR of BF

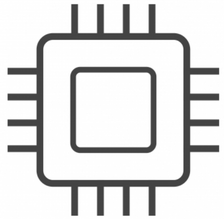
s : selectivity LRQ

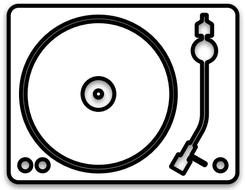
Range Queries

Cost analysis

long range: $\mathcal{O}(s \cdot N/B)$

get(9,90)





buffer

Bloom filters

fence pointers

L1

L2

L3

L4

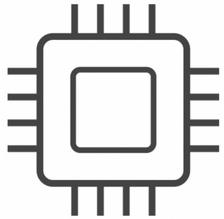
BOSTON UNIVERSITY

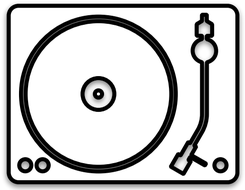
P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries
ϕ: FPR of BF

s : selectivity SRQ

get(9,15)

Range Queries





Cost analysis

long range: $\mathcal{O}(s \cdot N/B)$

short range: $\mathcal{O}(L)$

buffer

L1

L2

L3

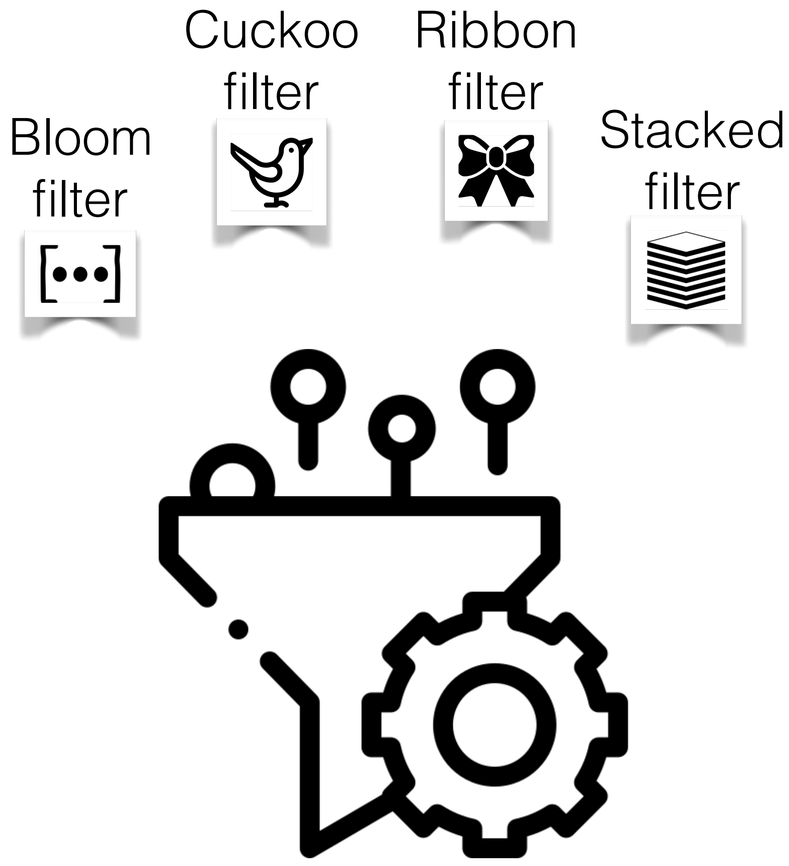
L4

Bloom filters

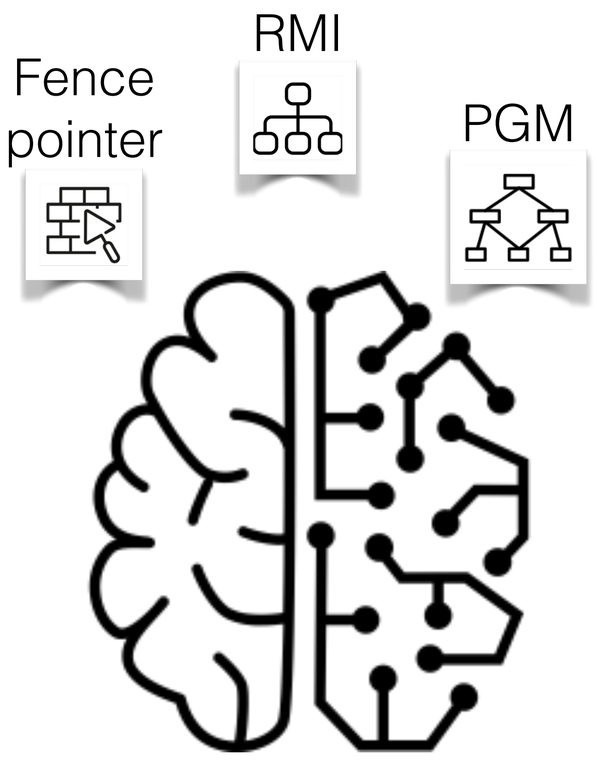
fence pointers

BOSTON UNIVERSITY

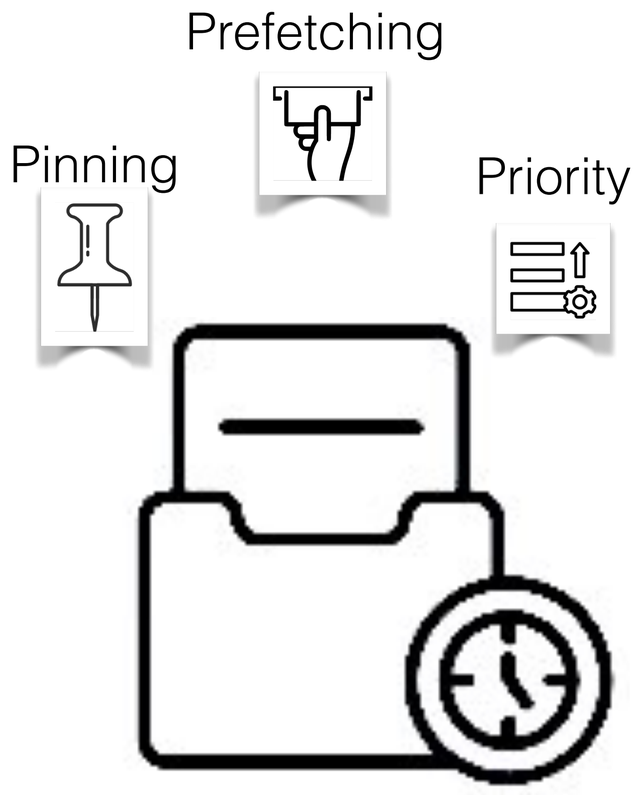
More **Read Optimizations**



filter structures



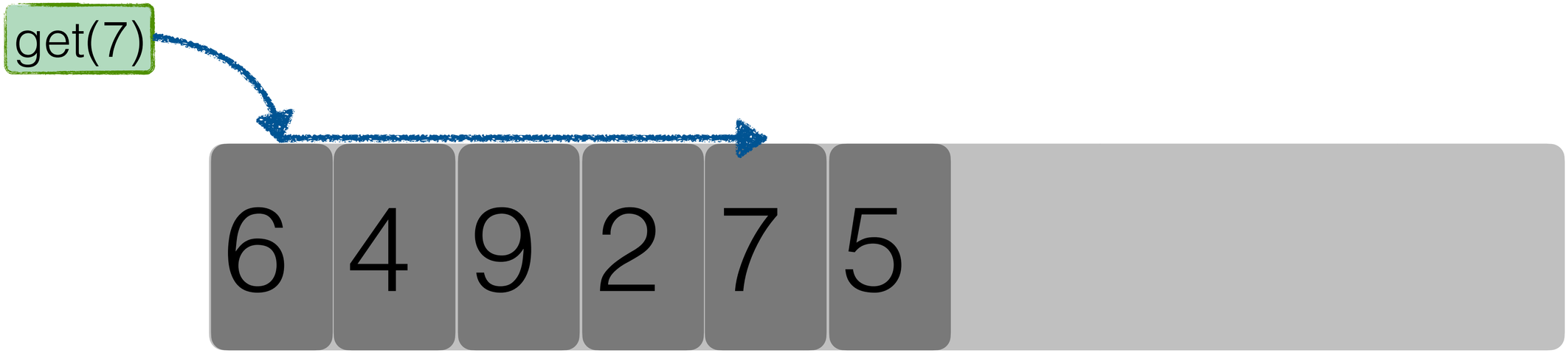
indexes



block cache

P: pages in buffer
B: entries/page

Buffer Implementation: **vector**



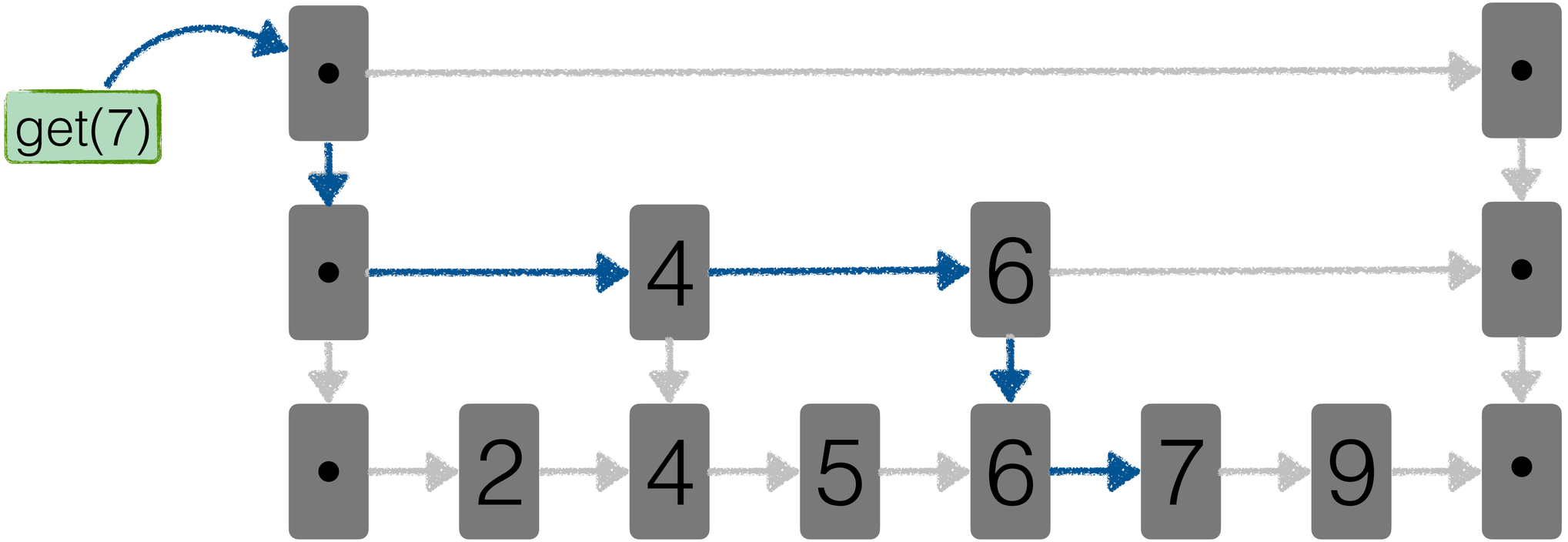
- great for ingestion-heavy w/l
- no extra space needed
- expensive points queries

ingestion cost: $\mathcal{O}(1)$

space complexity: $\mathcal{O}(P \cdot B)$

point query cost: $\mathcal{O}(P \cdot B)$

Buffer Implementation: **skiplist**



- great for mixed w/l
- some extra space needed
- good for points queries

P: pages in buffer
B: entries/page

Buffer Implementation

	vector	**skiplist**	**hashmap**
ingestion cost	$\mathcal{O}(1)$	$\mathcal{O}(log(P \cdot B))$	$\mathcal{O}(1)$
space complexity	$\mathcal{O}(P \cdot B)$	$\mathcal{O}(P \cdot B)$	$\mathcal{O}(P \cdot B)$
point query cost	$\mathcal{O}(P \cdot B)$	$\mathcal{O}(log(P \cdot B))$	$\mathcal{O}(1)$

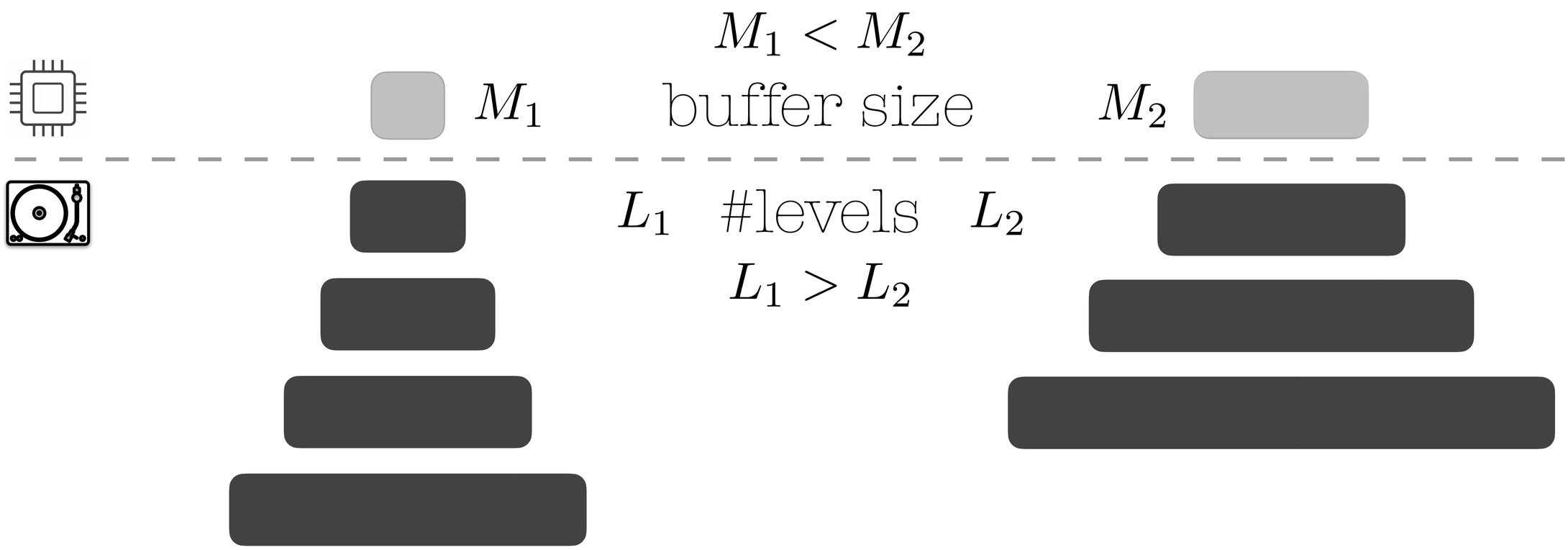
Ingestion-only workloads

Mixed workloads

I/O-bound workloads

BOSTON UNIVERSITY

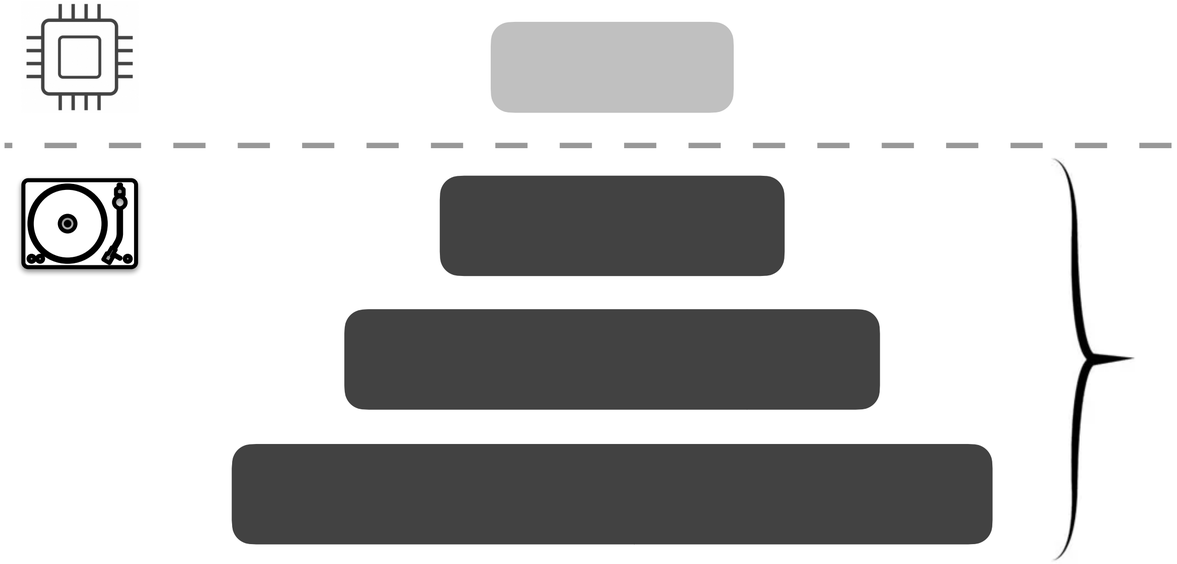
Size of the Buffer



- frequent flushes
- smaller but more levels
- poor read performance

- fewer larger levels
- good for reads
- high tail latency

L: #levels
T: size ratio

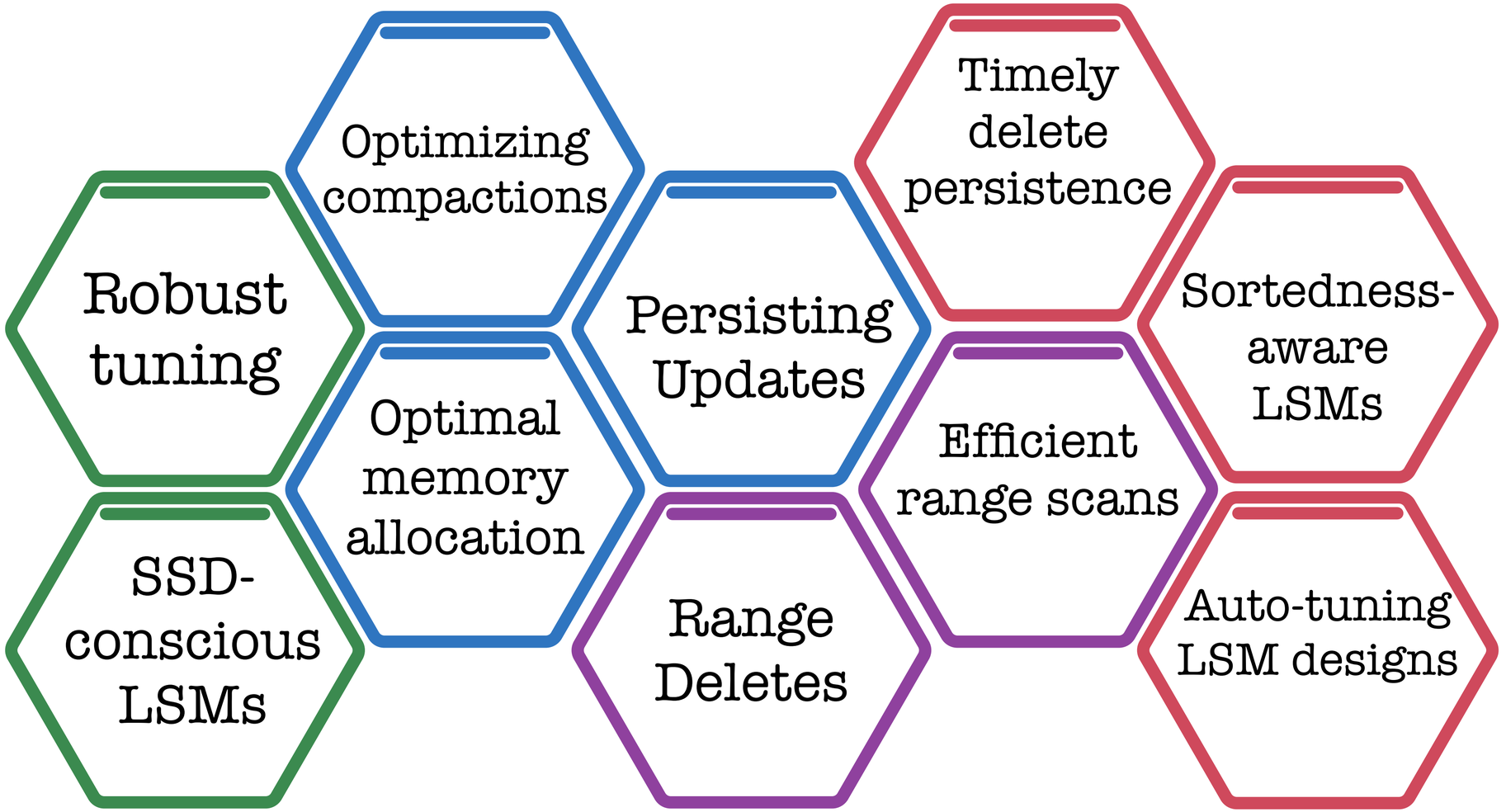


most data
on storage

if $T = 10$ & $L = 4$

99.9% on storage

How does the storage layer affect ingestion?


Optimizing compactions
Timely delete persistence
Robust tuning
Persisting Updates
Sortedness-aware LSMs
Optimal memory allocation
Efficient range scans
SSD-conscious LSMs
Range Deletes
Auto-tuning LSM designs

CS 561: ***Data Systems Architecture***

Class 4

Systems & Research Projects

Dr. Subhadeep Sarkar

BOSTON UNIVERSITY

DiSC lab

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