BOSTON UNIVERSITY

CS 561: **Data Systems Architectures**

class 4

Systems & Research Project

Prof. Manos Athanassoulis

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

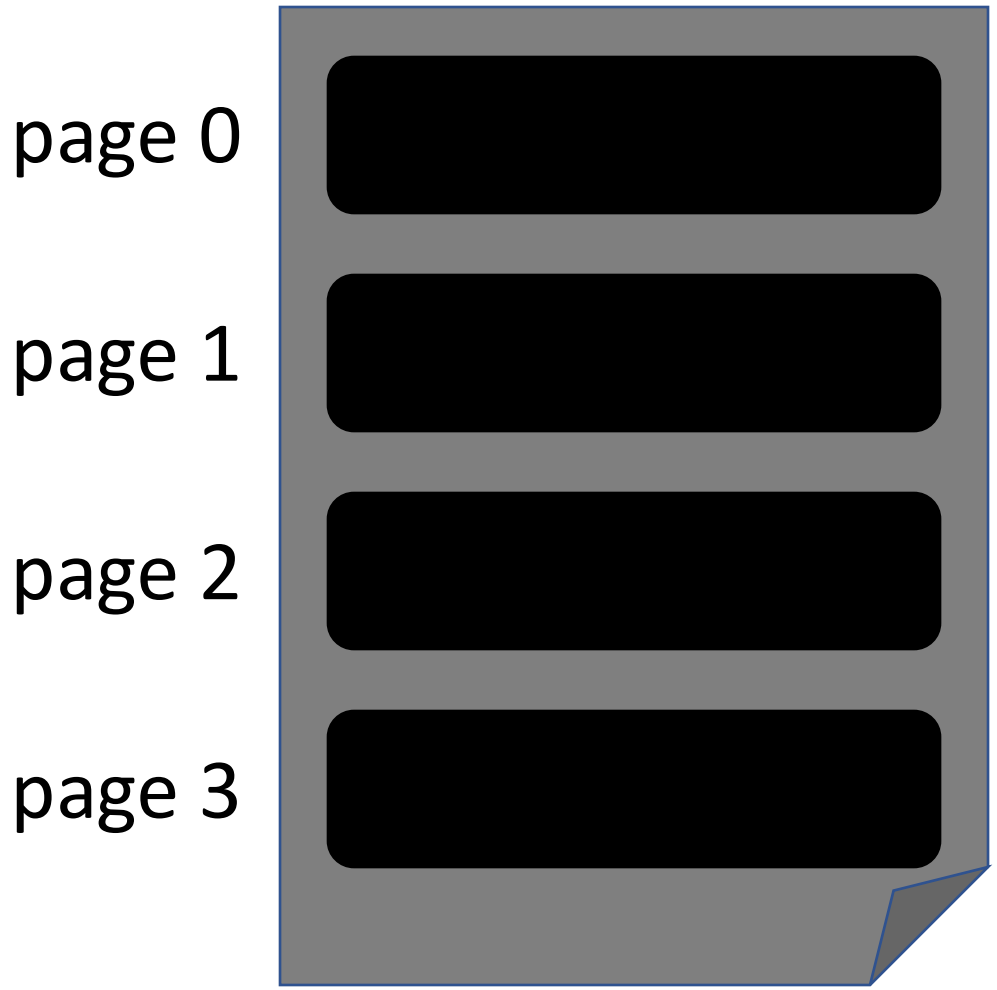
Let’s revisit Zonemaps

- Light-weight auxiliary data structure (*“scan accelerator”*)

Let’s revisit Zonemaps

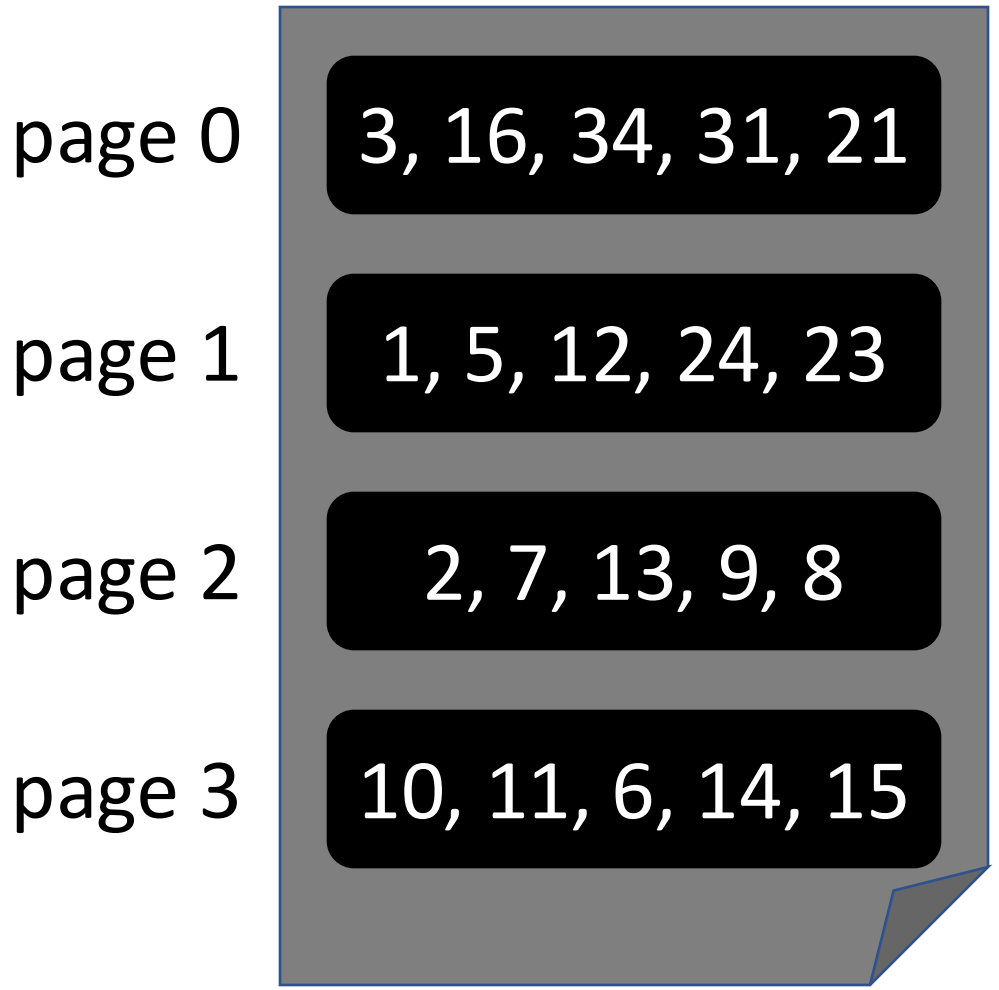
zonemaps

file = collection of pages



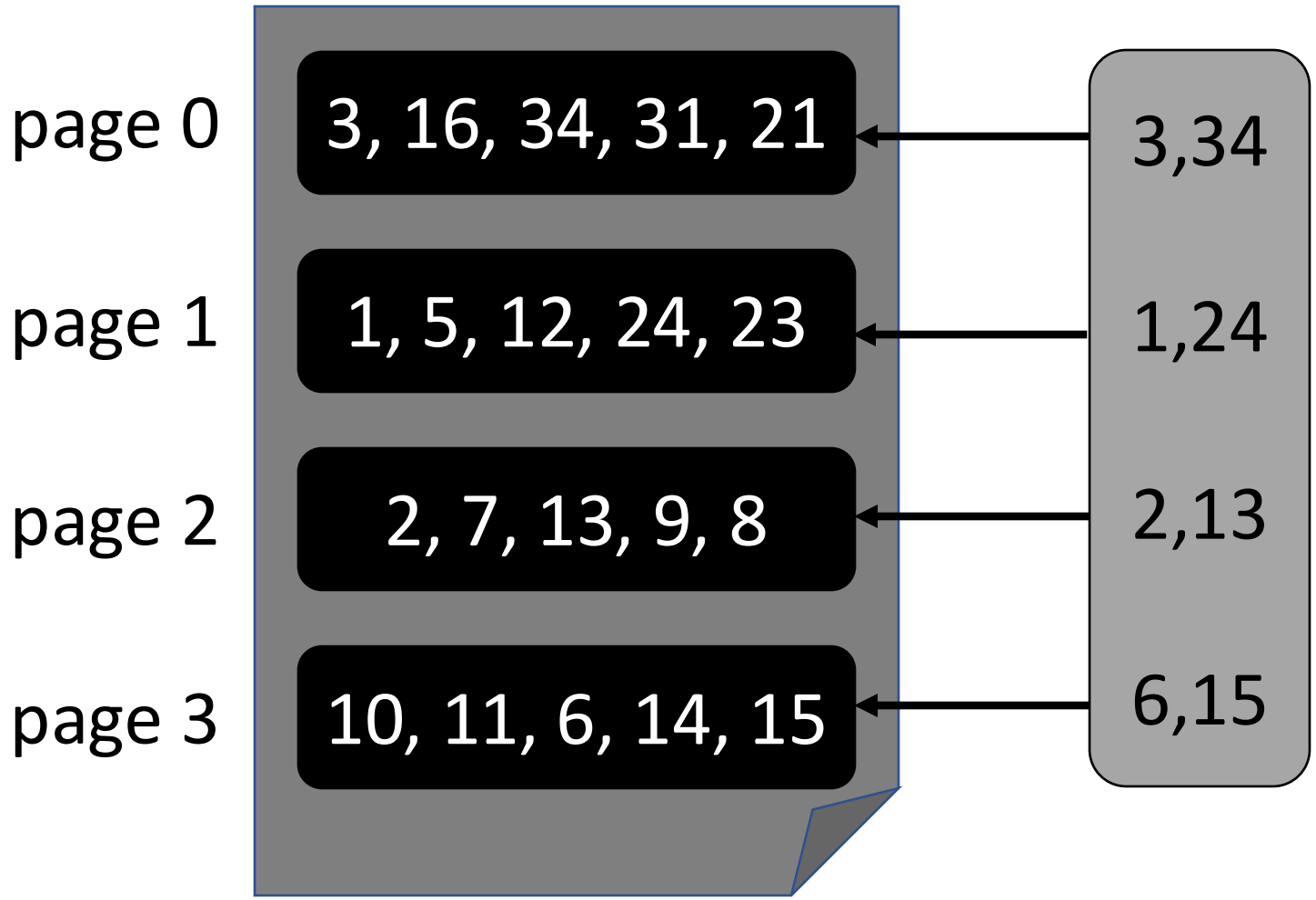
zonemaps

file = collection of pages



zonemaps

file = collection of pages



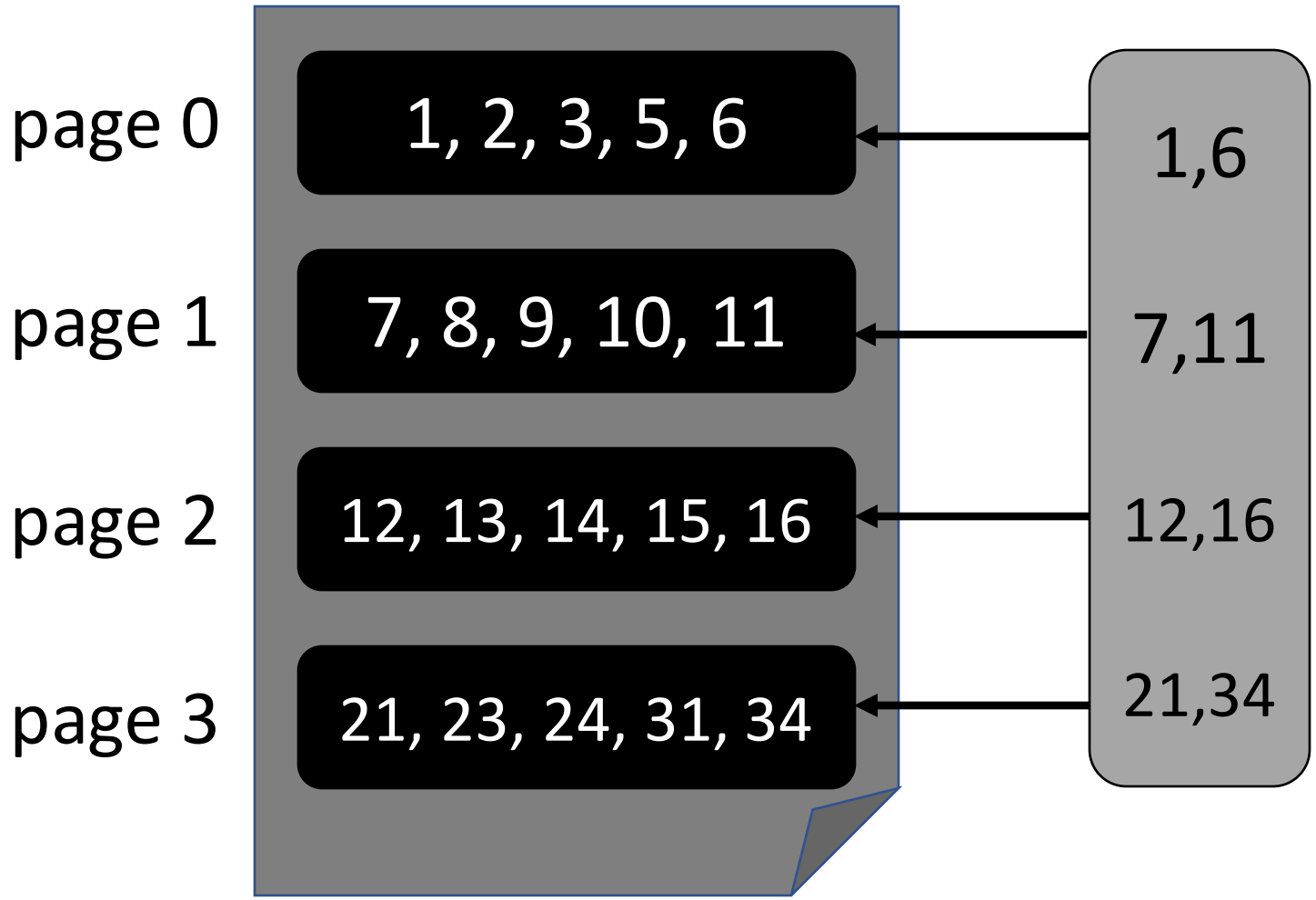
light-weight

typically retained in memory

But what if the data is sorted?

zonemaps

file = collection of pages

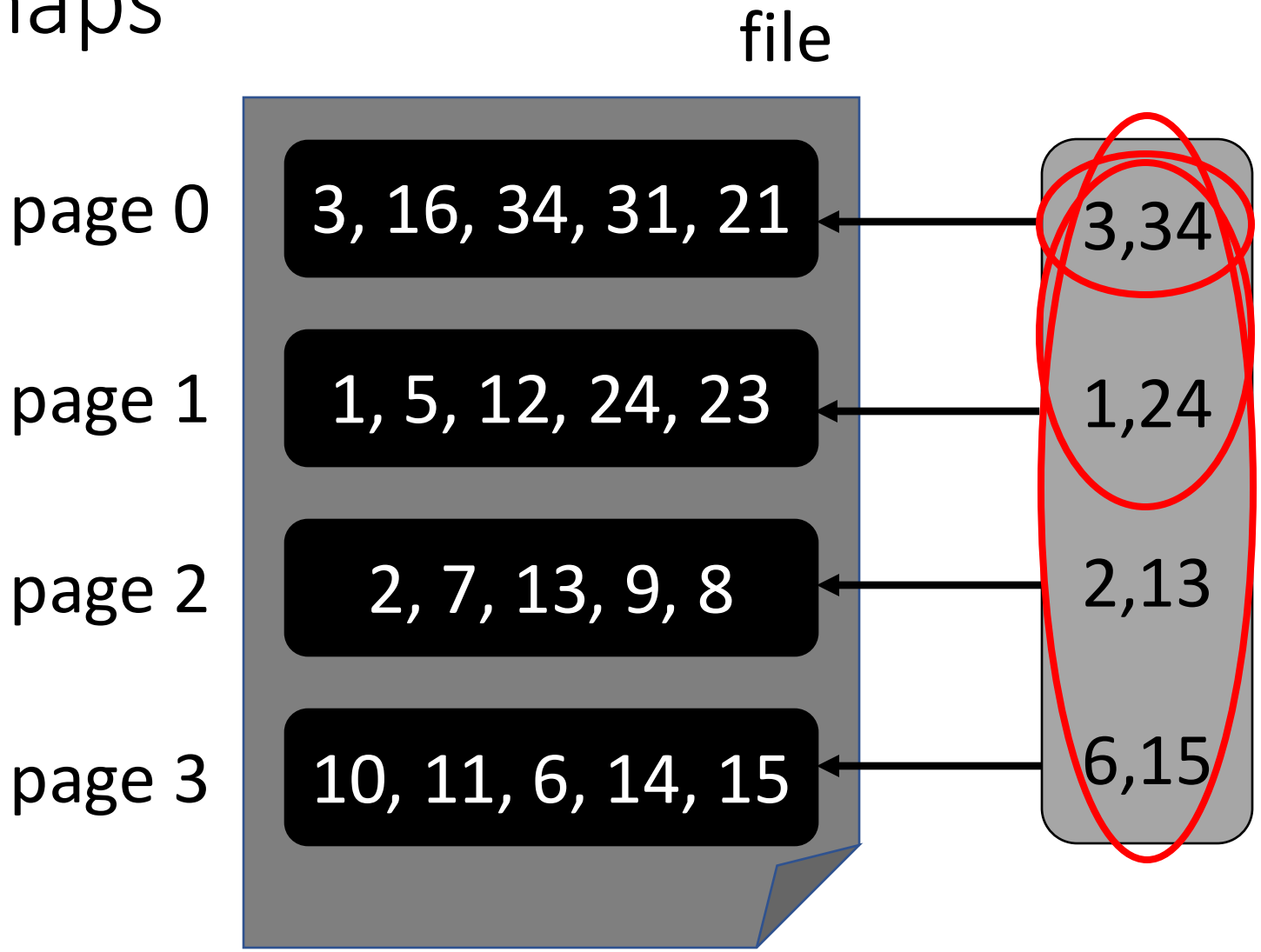


light-weight

typically retained in memory

But what if the data is sorted?

zonemaps



Range Queries

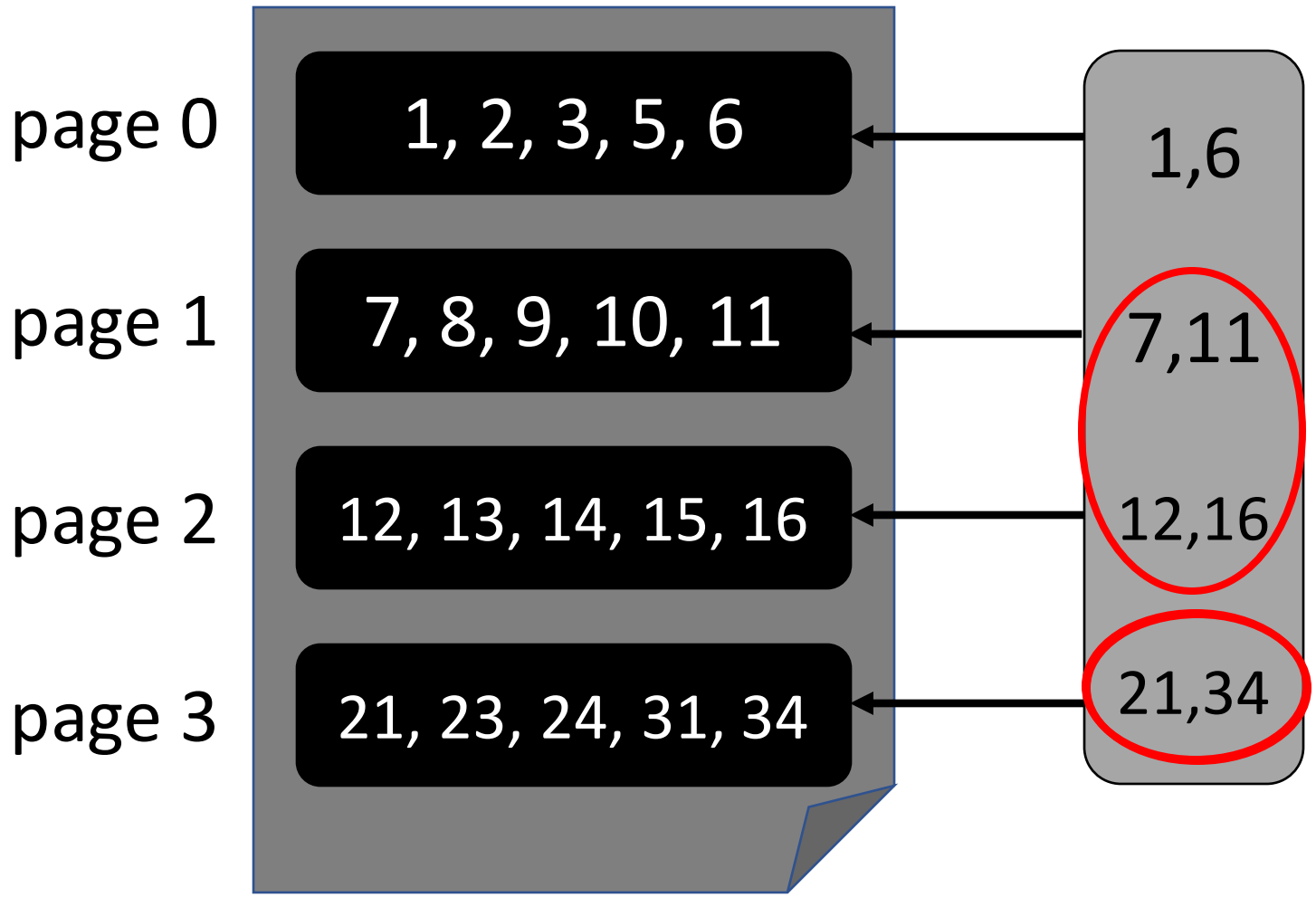
[25, 100] 1 page

[20, 25] 2 pages

[10, 13] 4 pages

zonemaps

But what if the data is sorted?



Range Queries

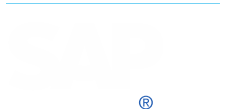
[25, 100] 1 page

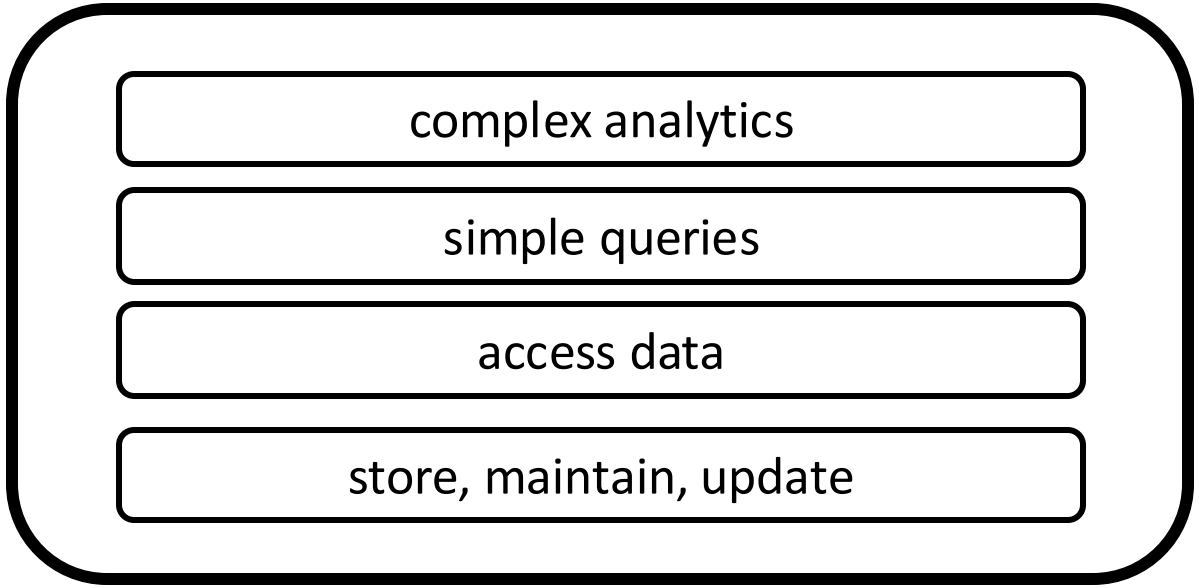
[20, 25] 1 page

[10, 13] 2 pages

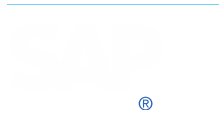
Zonemaps efficiency depends on data & queries!

data systems



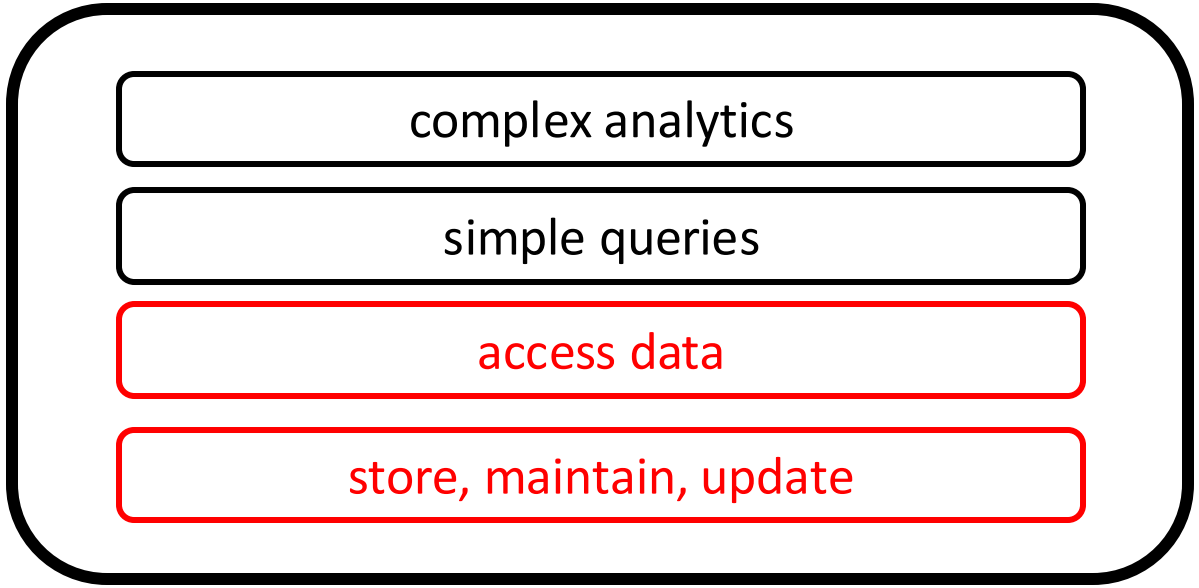


data systems



>$200B by 2020, growing at 11.7% every year

[The Forbes, 2016]



access methods*

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

data systems core: storage engines

main decisions

how to ***store*** data?

how to ***access*** data?

how to ***update*** data?

let's simplify: **key-value** storage engines

collection of keys-value pairs

query on the key, return both key and value

remember

Couchbase

state-of-the-art design

how general is a key value store?

can we store relational data?

yes! {<primary_key>,<rest_of_the_row>}

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

what is the caveat?

how to index these attributes?

other problems?

index: { **name**, { **student_id** } }

index: { **yob**, { **student_id$_1$**, **student_id$_2$**, **…** } }

how general is a key value store?

can we store relational data?

yes! {<primary_key>,<rest_of_the_row>}

how to efficiently code if we do not know the structure of the "***value***"

index: { **yob**, { **student_id$_1$, student_id$_2$, …** } }

how to use a key-value store?

basic interface

put(k,v)

{v} = get(k) {v_1, v_2, …} = get(k)

{v_1, v_2, …} = get_range(k_{min}, k_{max}) {v_1, v_2, …} = full_scan()

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

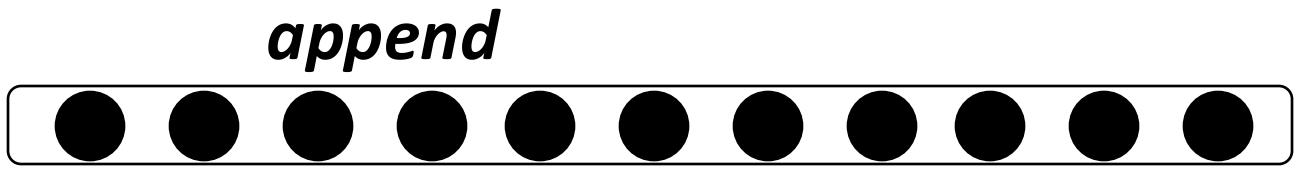
deletes: delete(k)

updates: update(k,v) is it different than put?

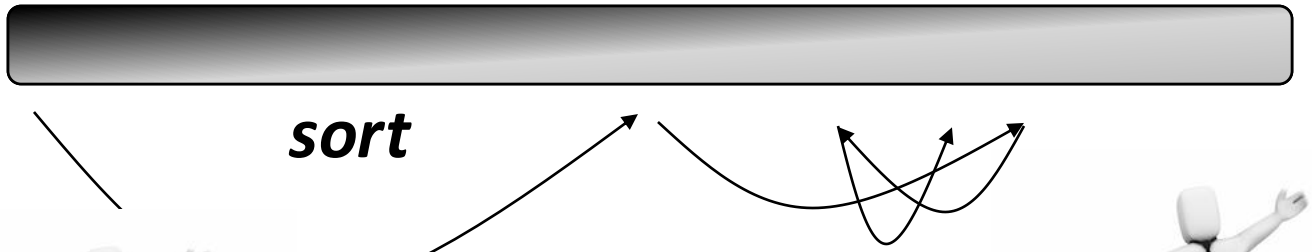
get set: {v_1, v_2, …} = get_set(k_1, k_2, …)

how to build a key-value store?

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



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

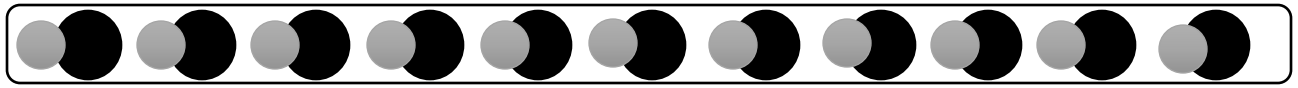


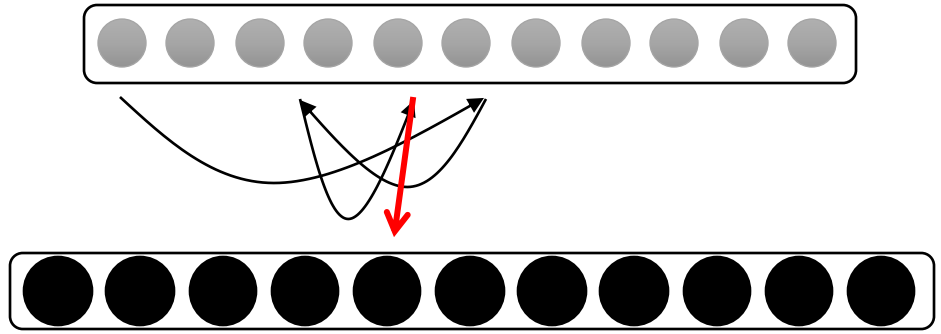
what about full scan?

and then

range queries?

can we separate keys and values?





at what price?

locality? code?

read queries
(point or range)

inserts
(or updates)

sort data

simply append

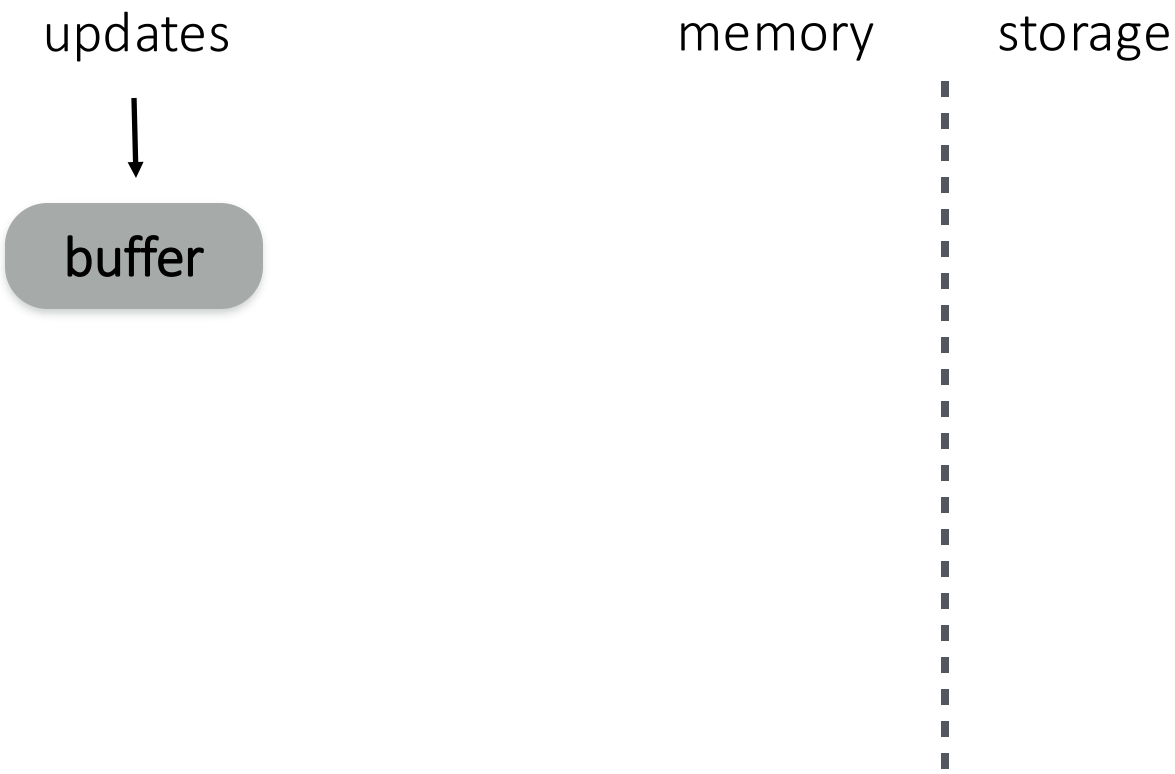
amortize sorting cost

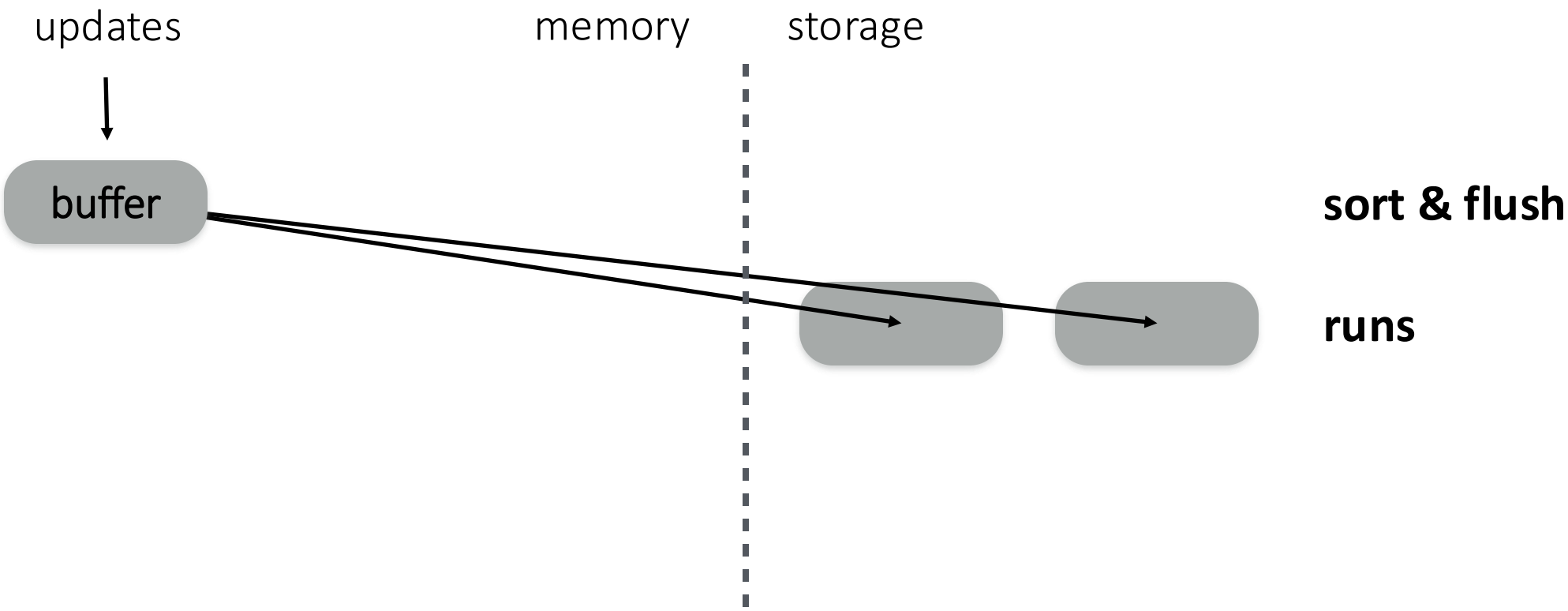
avoid resorting after every update

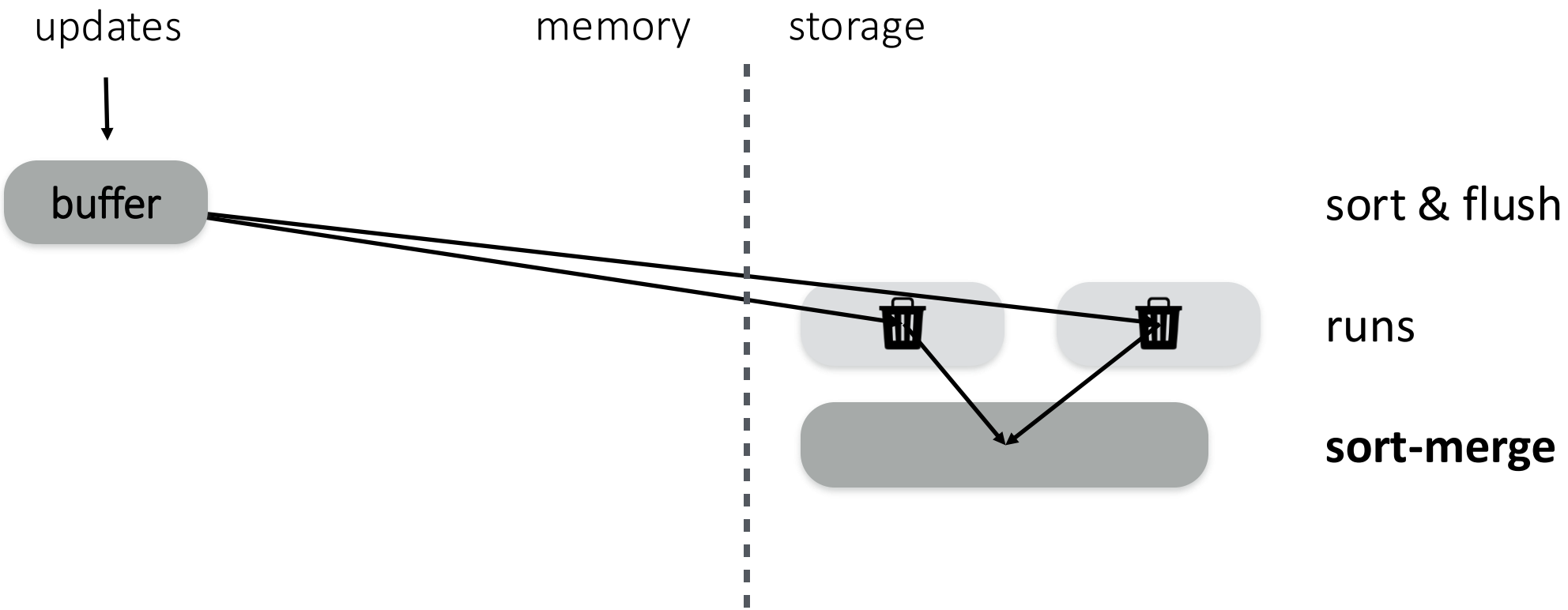
how to bridge?

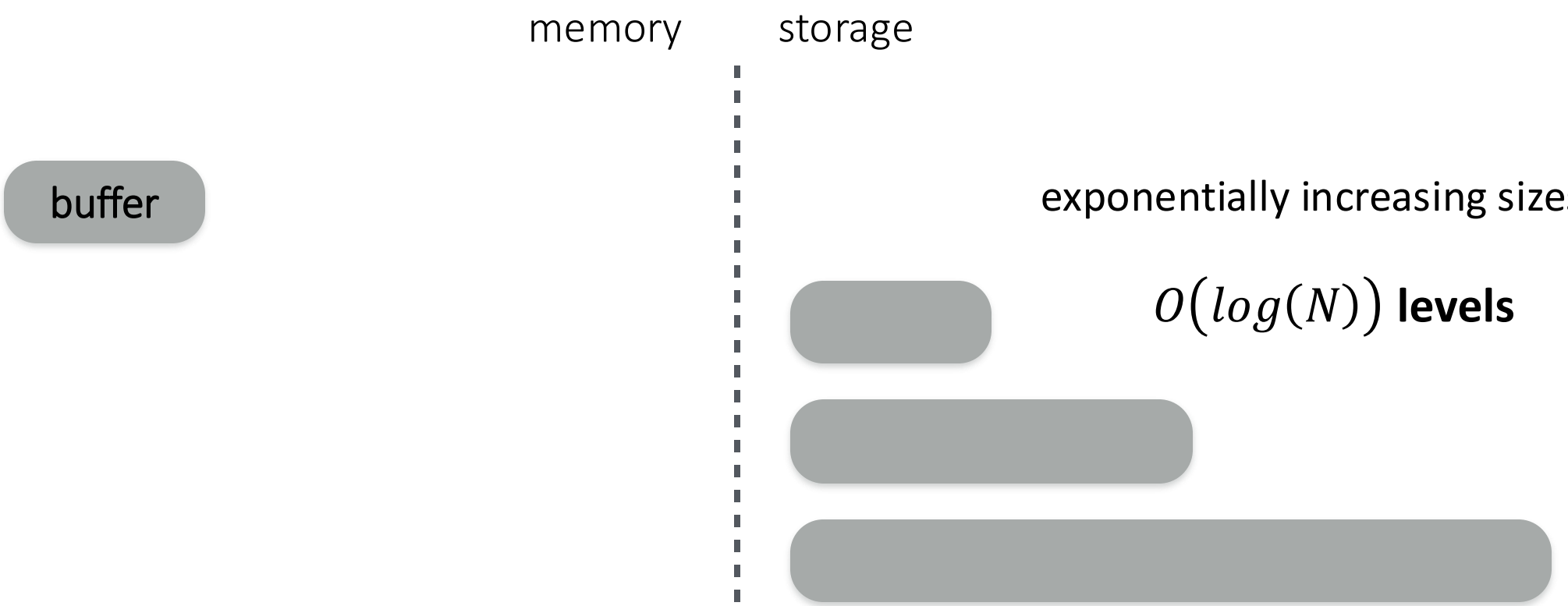
LSM-tree
Key-Value Stores

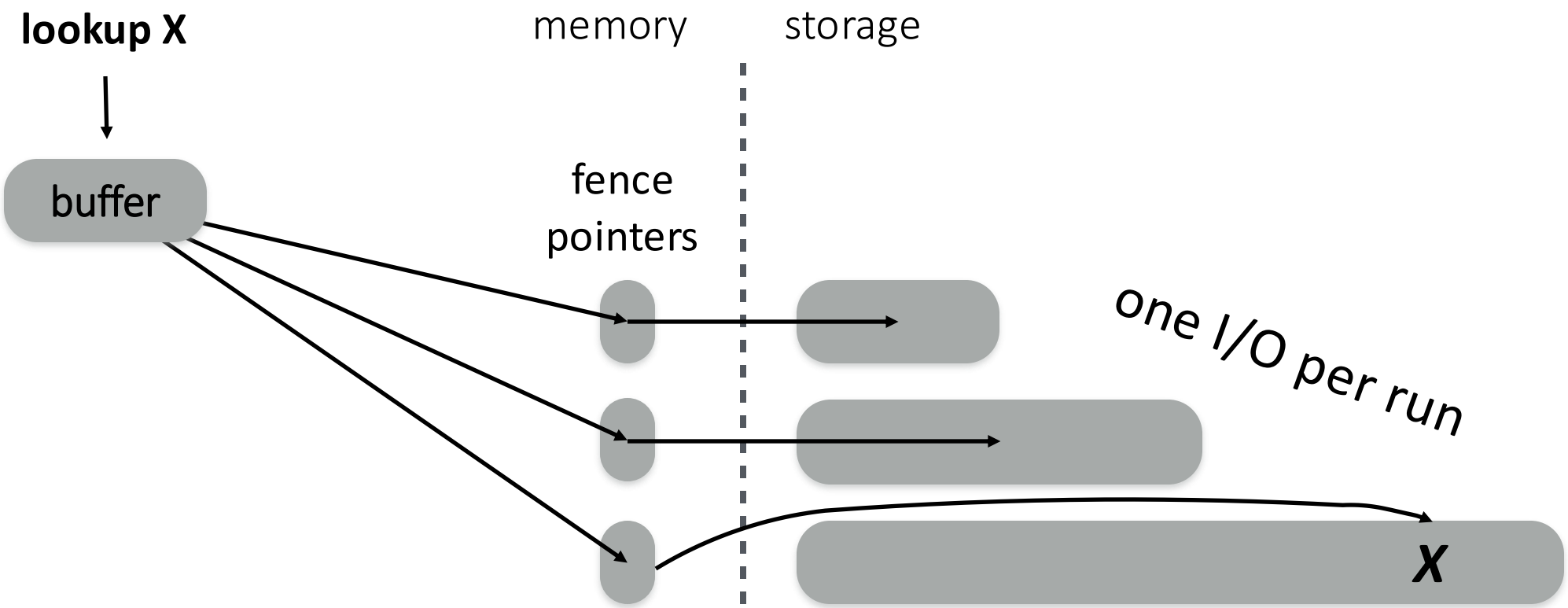
What are they really?

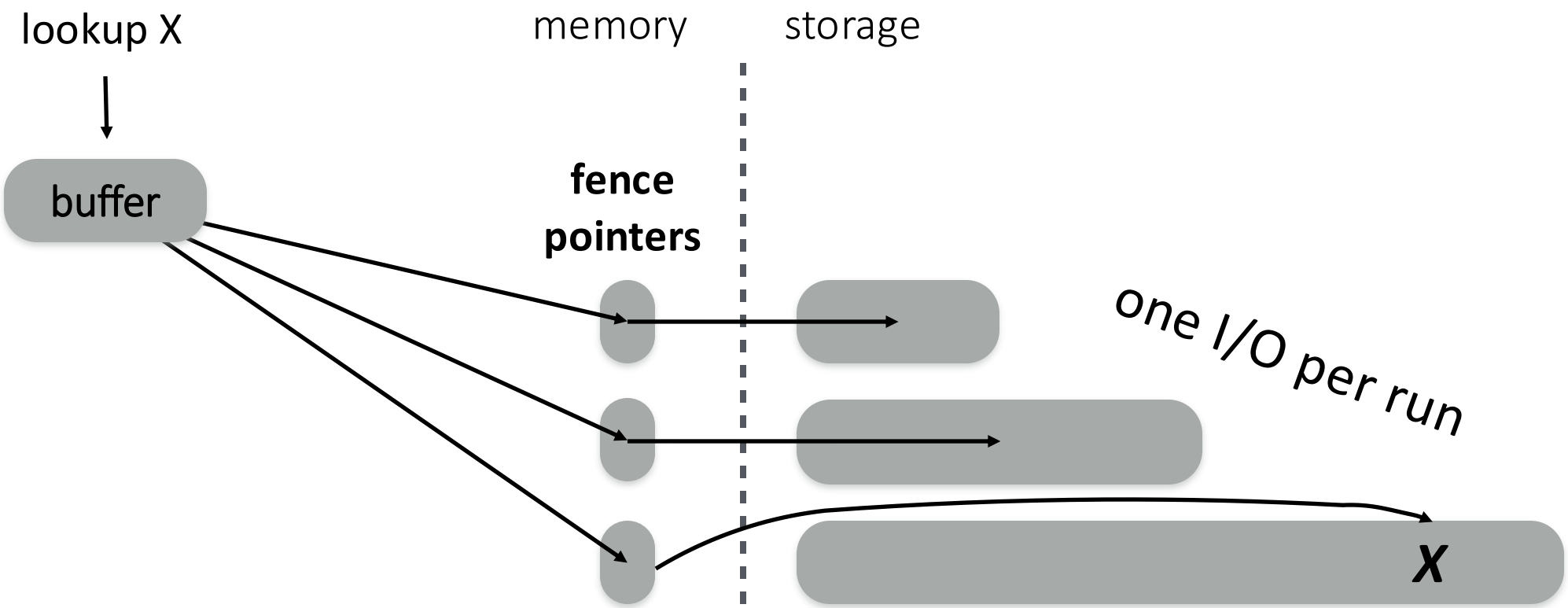


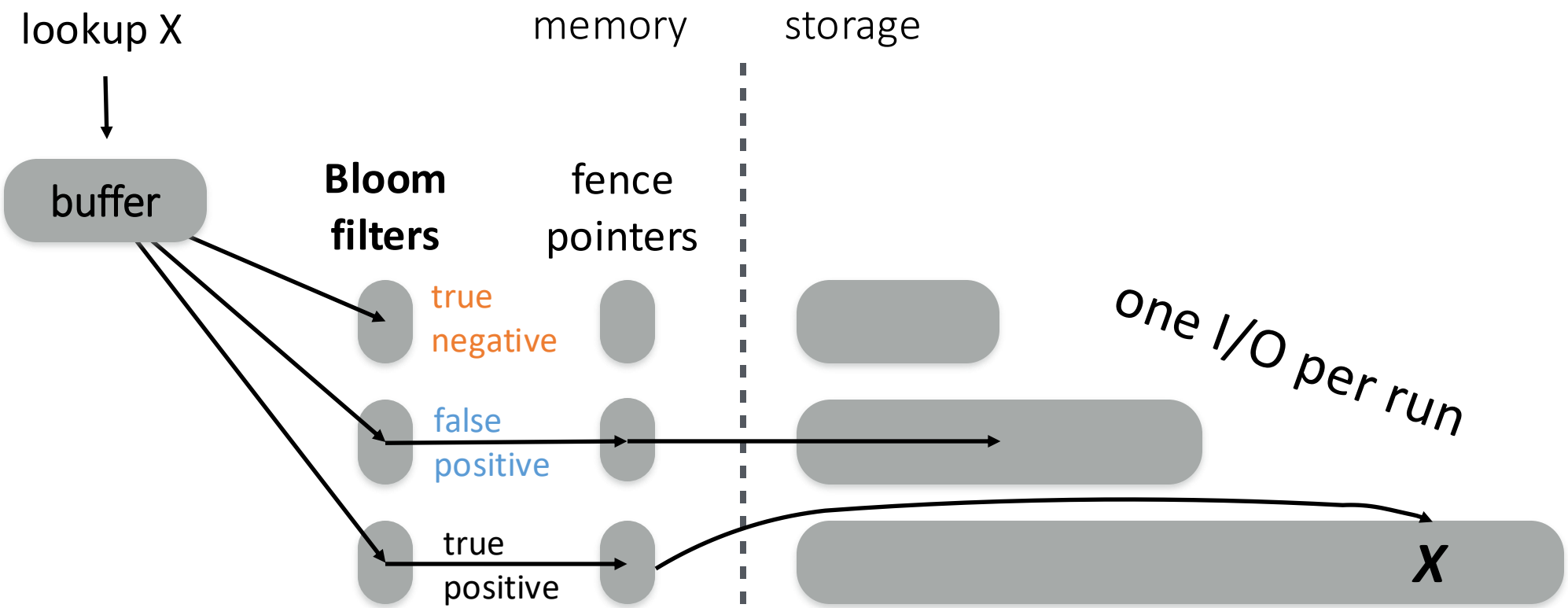




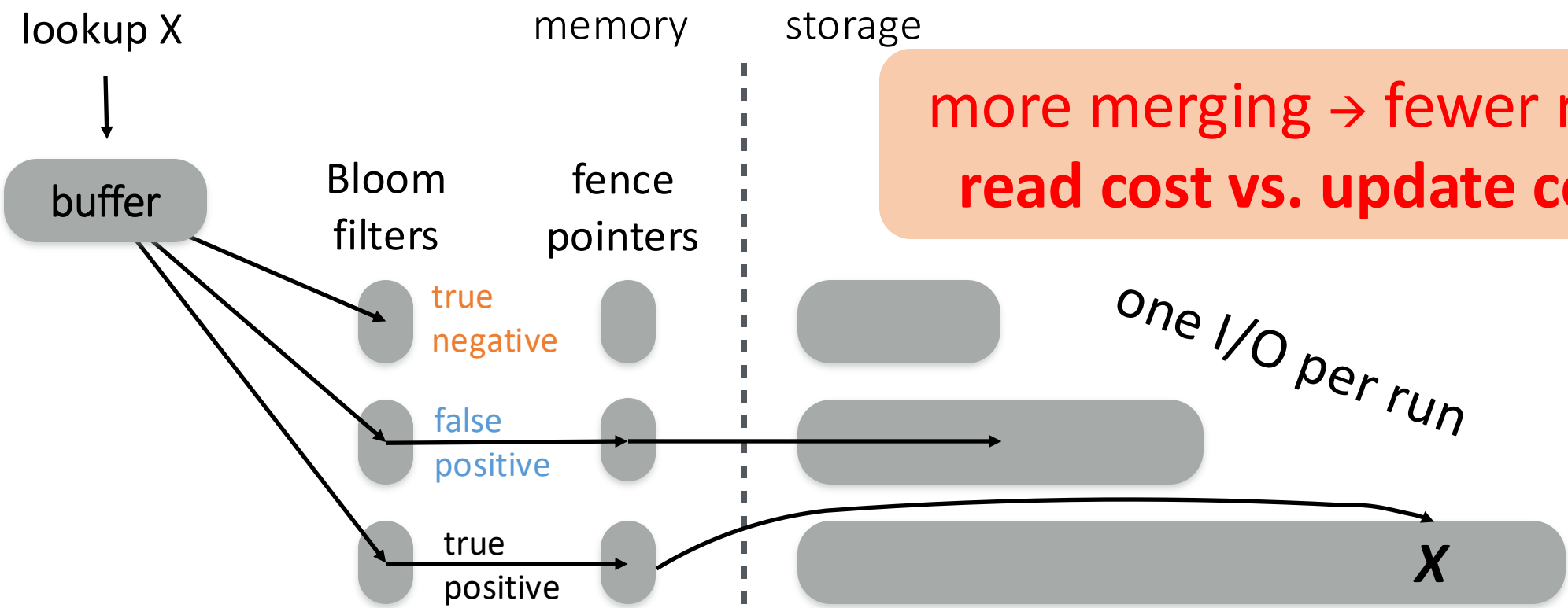








performance & cost trade-offs



more merging → fewer runs

read cost vs. update cost

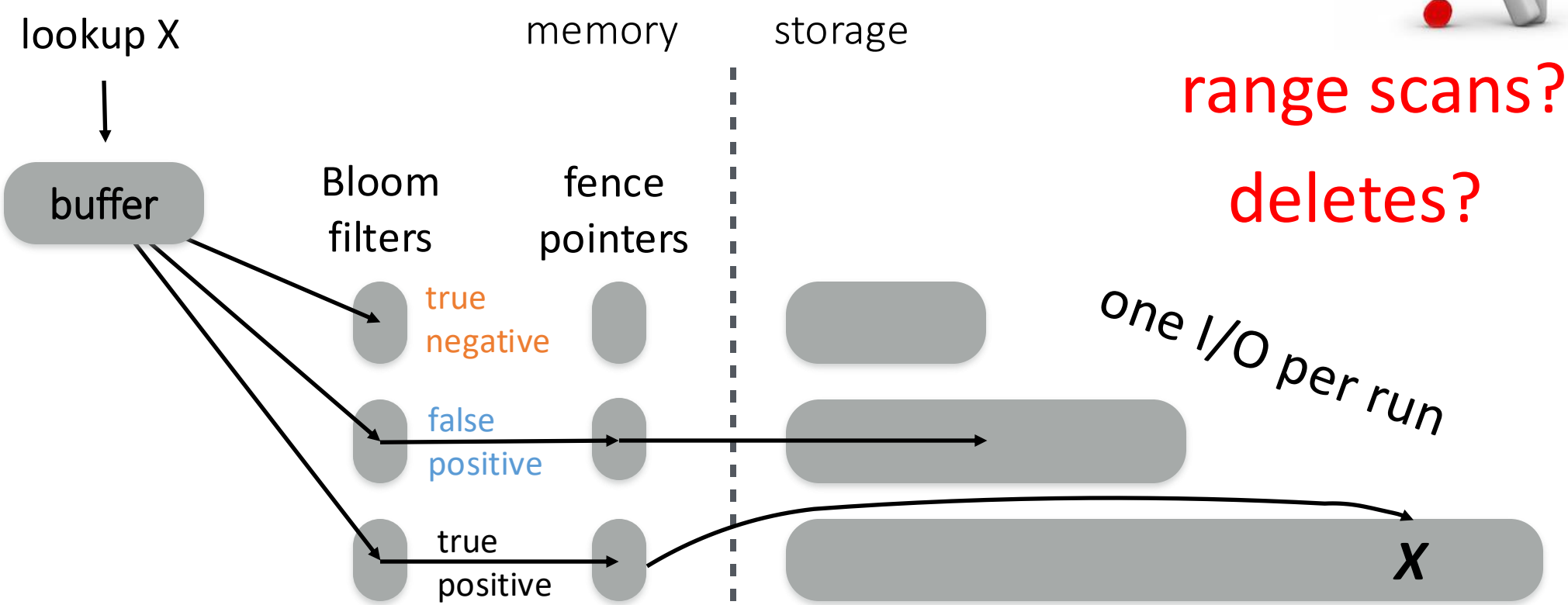
bigger filters → fewer false positives

memory space vs. read cost

other operations

range scans?

deletes?



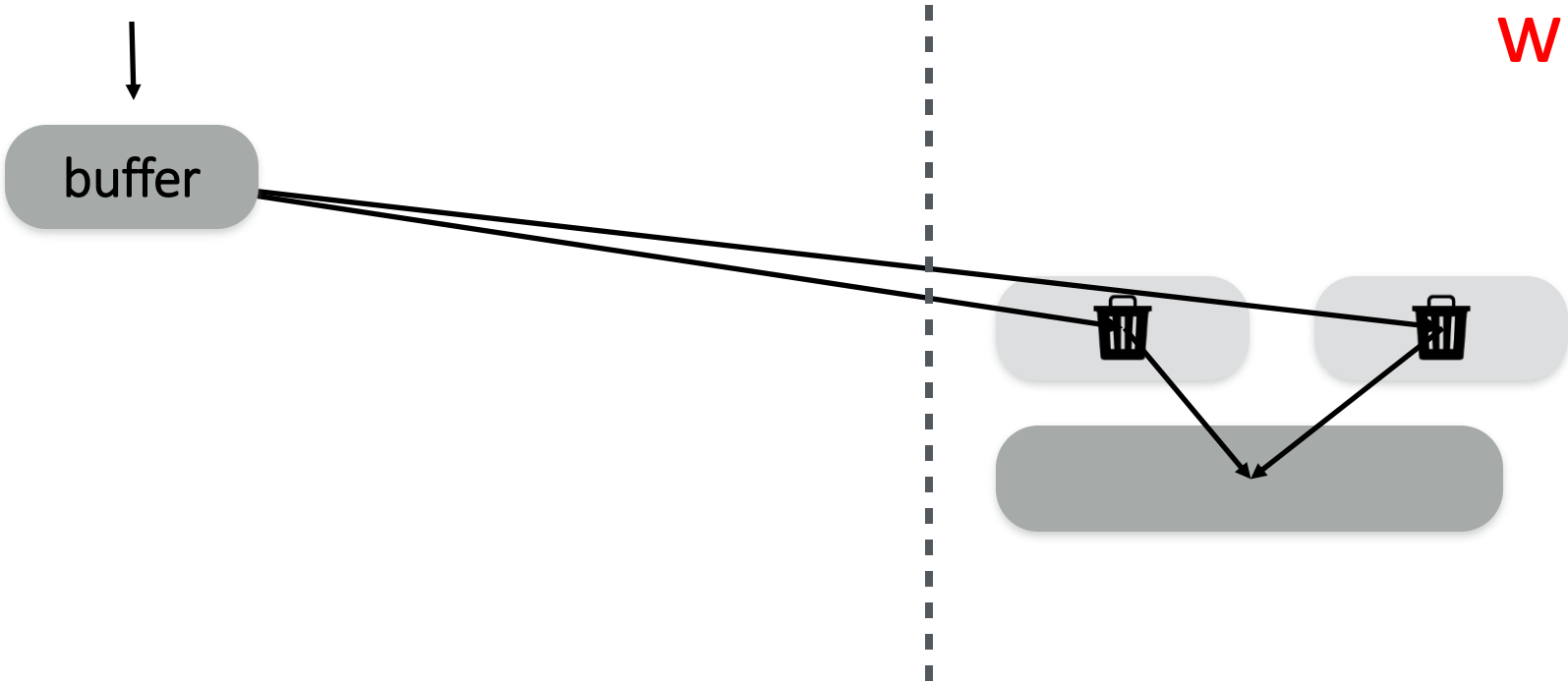
remember merging?

updates

memory

storage

buffer



what strategies?

sort & flush

runs

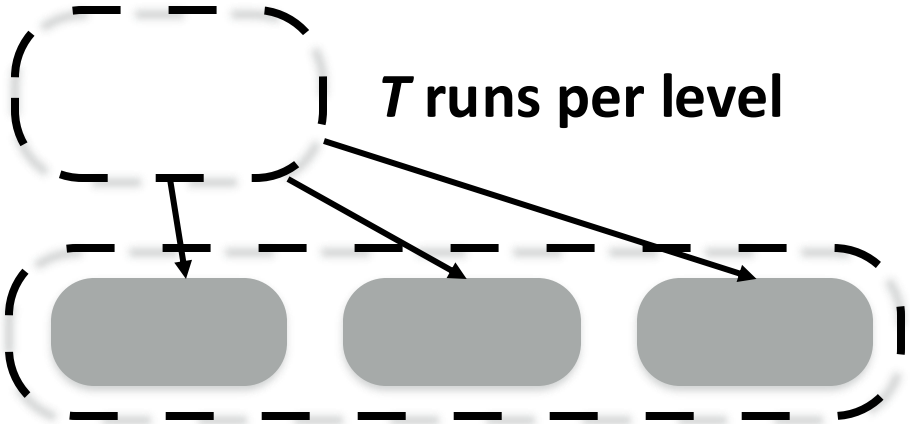
sort-merge

Merge Policies

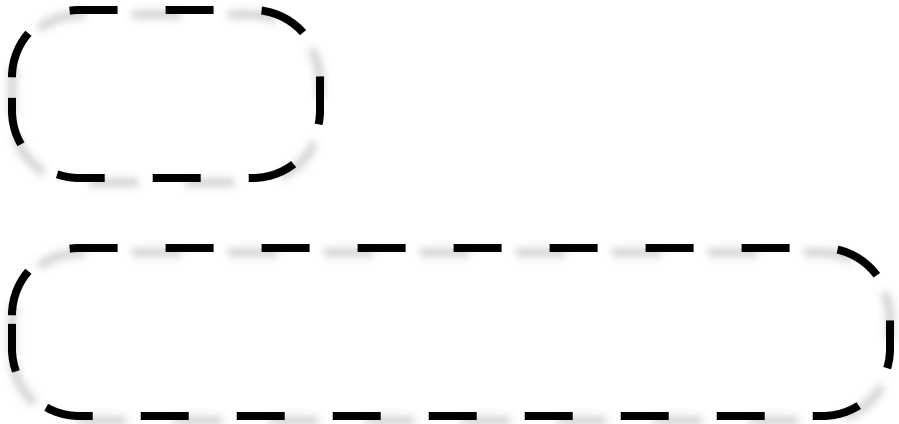
Tiering
write-optimized

Leveling
read-optimized

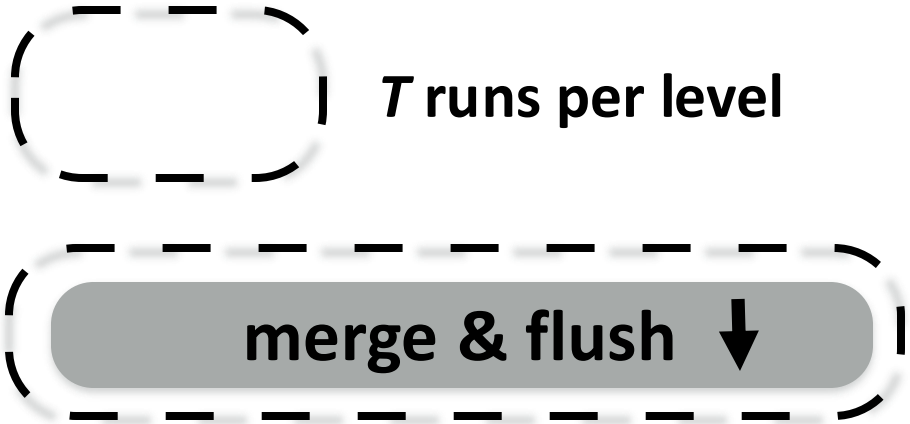
Tiering
write-optimized



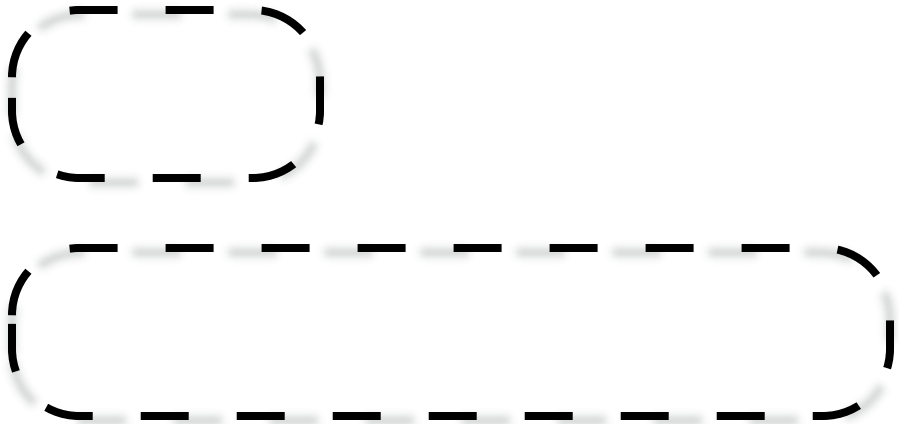
Leveling
read-optimized



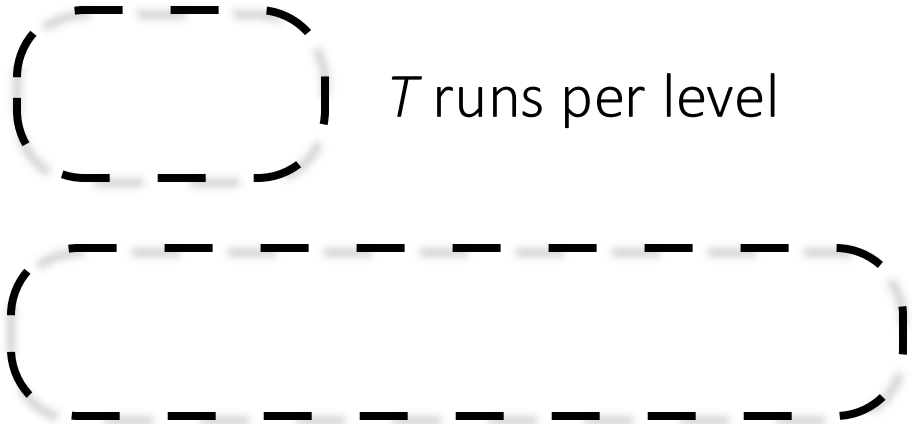
Tiering
write-optimized



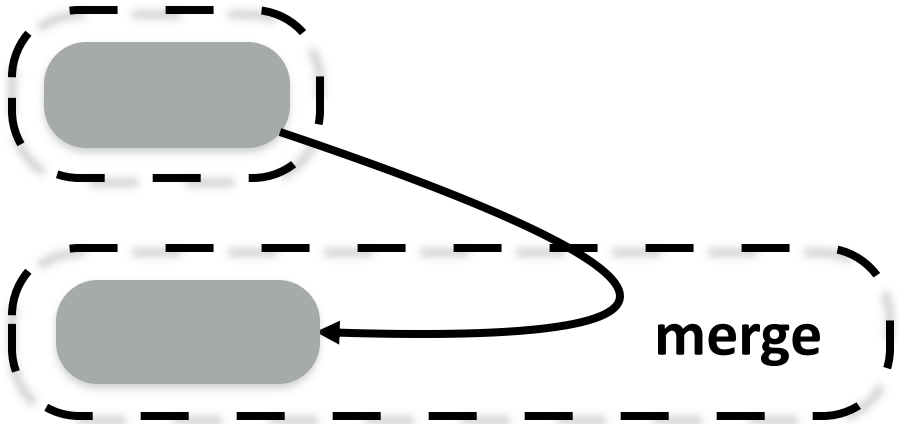
Leveling
read-optimized



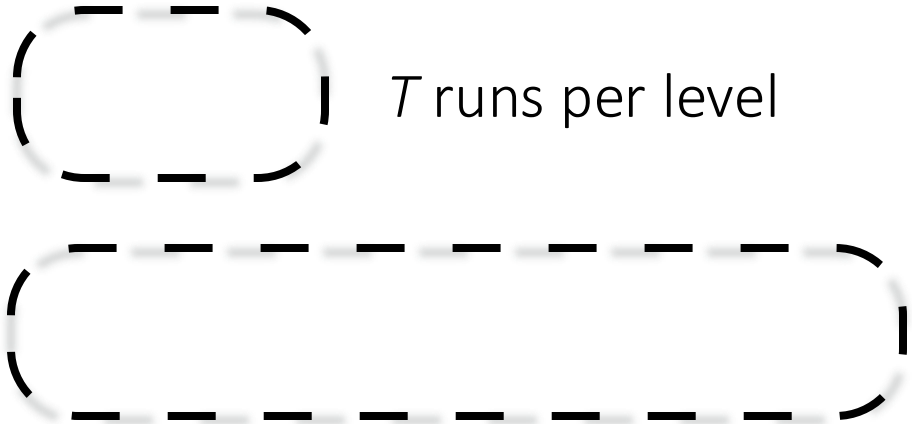
Tiering
write-optimized



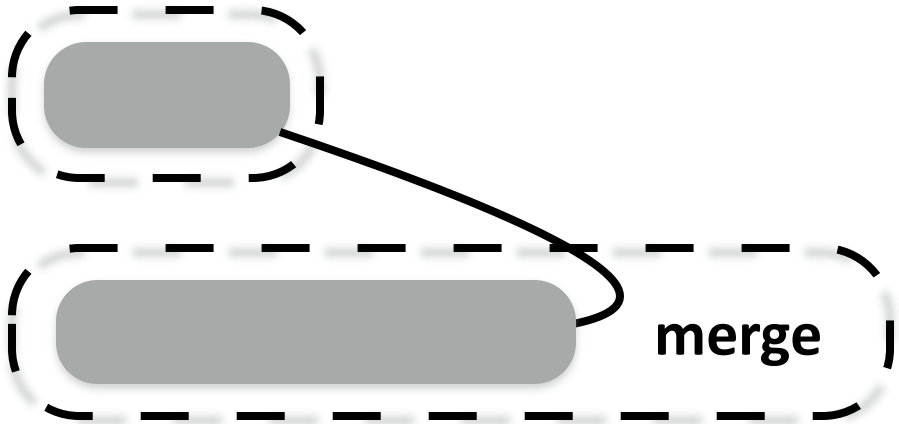
Leveling
read-optimized



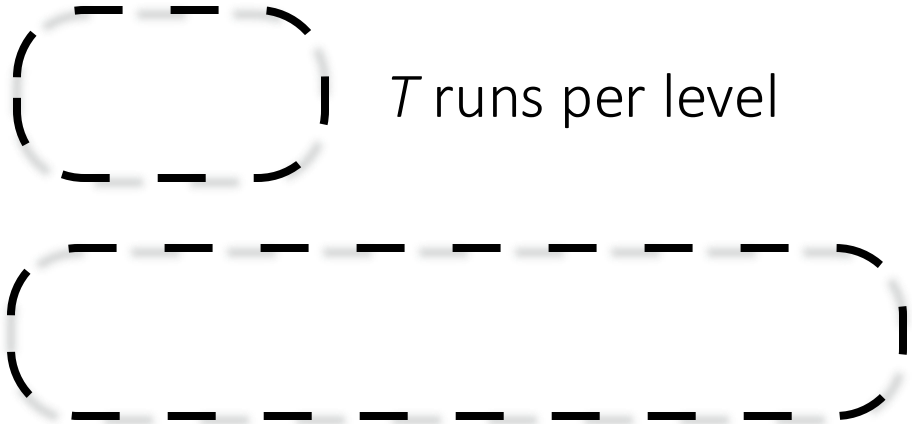
Tiering
write-optimized



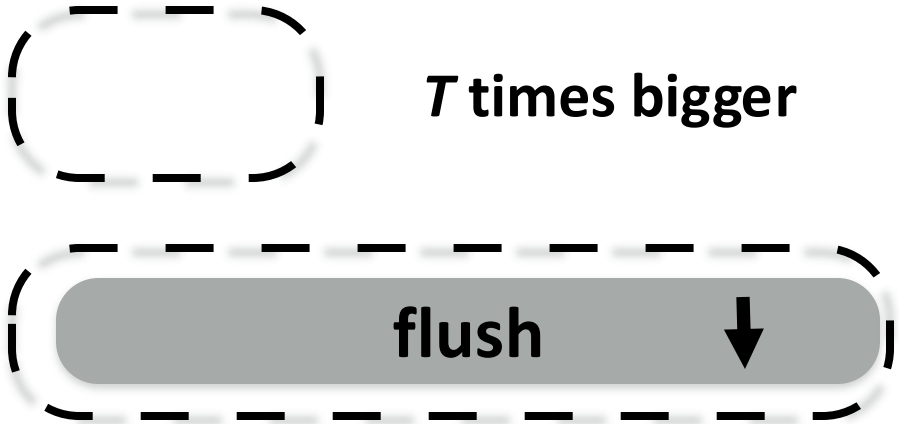
Leveling
read-optimized



Tiering
write-optimized



Leveling
read-optimized



Systems Project: LSM-Trees

tuning knobs

merge policy

size ratio

lookup X

memory

storage

buffer

Bloom filters

fence pointers

true negative

false positive

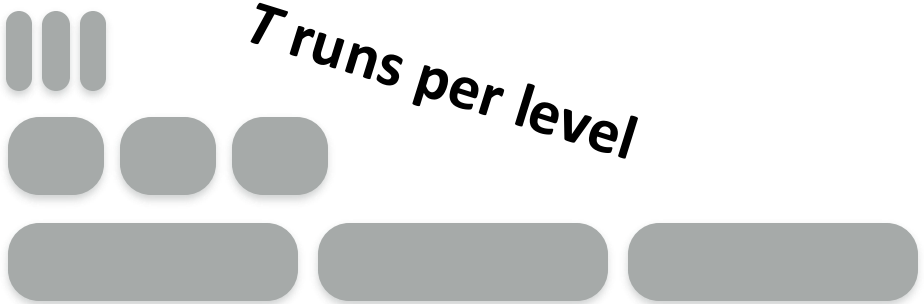
true positive

one I/O per run

X

more on LSM-Tree performance

Tiering
write-optimized



Leveling
read-optimized

lookup cost:

$$O\left(T \cdot log_T(N) \cdot e^{-M/N}\right)$$

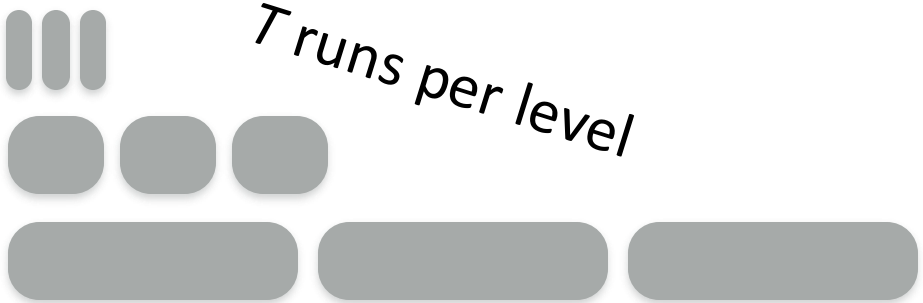
runs per level — levels — false positive rate

$$O\left(log_T(N) \cdot e^{-M/N}\right)$$

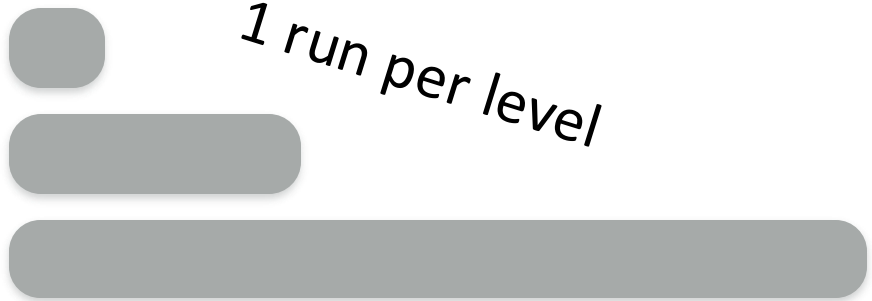
levels — false positive rate

BOSTON UNIVERSITY

Tiering
write-optimized



Leveling
read-optimized



lookup cost: $O\left(T \cdot log_T(N) \cdot e^{-M/N}\right)$

update cost: $\boldsymbol{O\left(log_T(N)\right)}$

levels

$O\left(log_T(N) \cdot e^{-M/N}\right)$

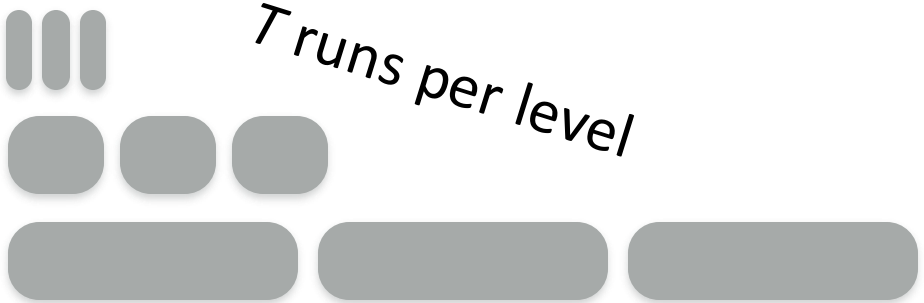
$\boldsymbol{O\left(T \cdot log_T(N)\right)}$

merges per level

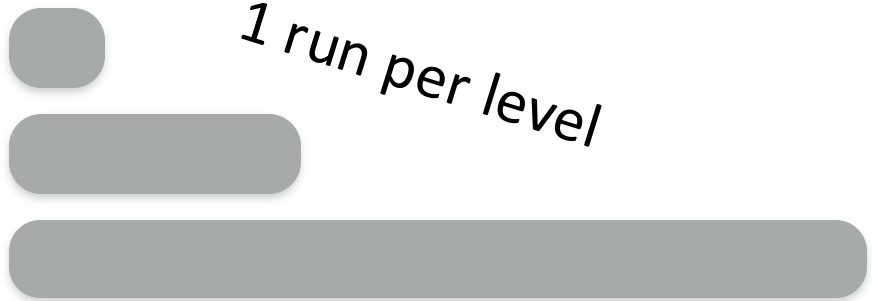
levels

BOSTON UNIVERSITY

Tiering
write-optimized



Leveling
read-optimized

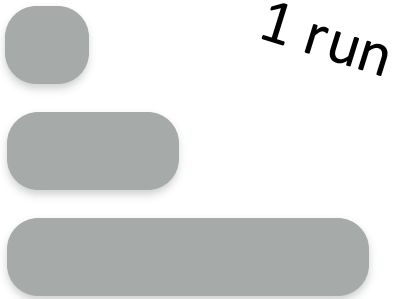


lookup cost: $O\left(T \cdot log_T(N) \cdot e^{-M/N}\right)$ $O\left(log_T(N) \cdot e^{-M/N}\right)$

update cost: $O\left(log_T(N)\right)$ $O\left(T \cdot log_T(N)\right)$

for size ratio T ⌄⌄

Tiering
write-optimized



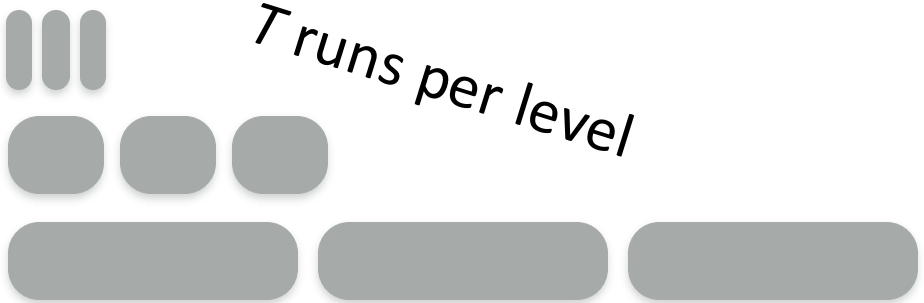
Leveling
read-optimized

lookup cost: $O\left(log_T(N) \cdot e^{-M/N}\right) = O\left(log_T(N) \cdot e^{-M/N}\right)$

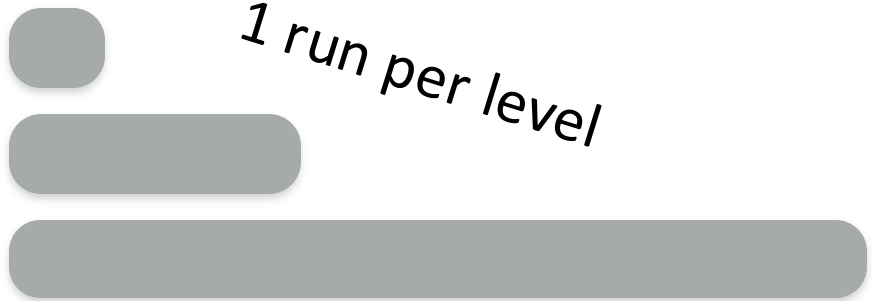
update cost: $O\left(log_T(N)\right) = O\left(log_T(N)\right)$

for size ratio T ⌄⌄

Tiering
write-optimized



Leveling
read-optimized



lookup cost: $O\left(T \cdot log_T(N) \cdot e^{-M/N}\right)$ $O\left(log_T(N) \cdot e^{-M/N}\right)$

update cost: $O\left(log_T(N)\right)$ $O\left(T \cdot log_T(N)\right)$

for size ratio T ⌃⌃

Tiering
write-optimized

$O(N)$ runs per level

log

Leveling
read-optimized

1 run per level

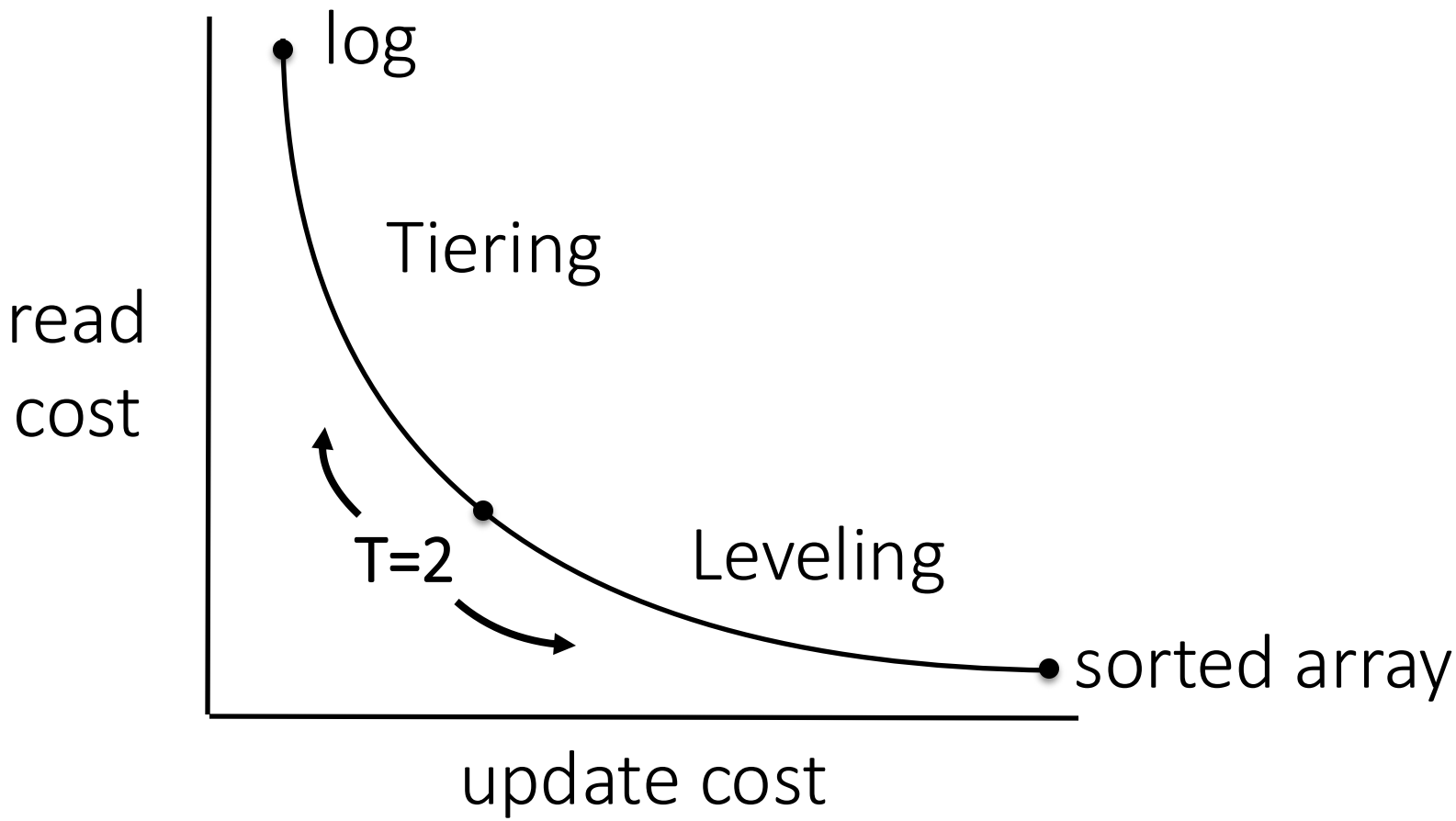
sorted array

lookup cost: $O\left(T \cdot log_T(N) \cdot e^{-M/N}\right)$ $O\left(log_T(N) \cdot e^{-M/N}\right)$

update cost: $O\left(log_N(N)\right) = \boldsymbol{O(1)}$ $O\left(N \cdot log_N(N)\right) = \boldsymbol{O(N)}$

for size ratio T **N** ≫

BOSTON UNIVERSITY



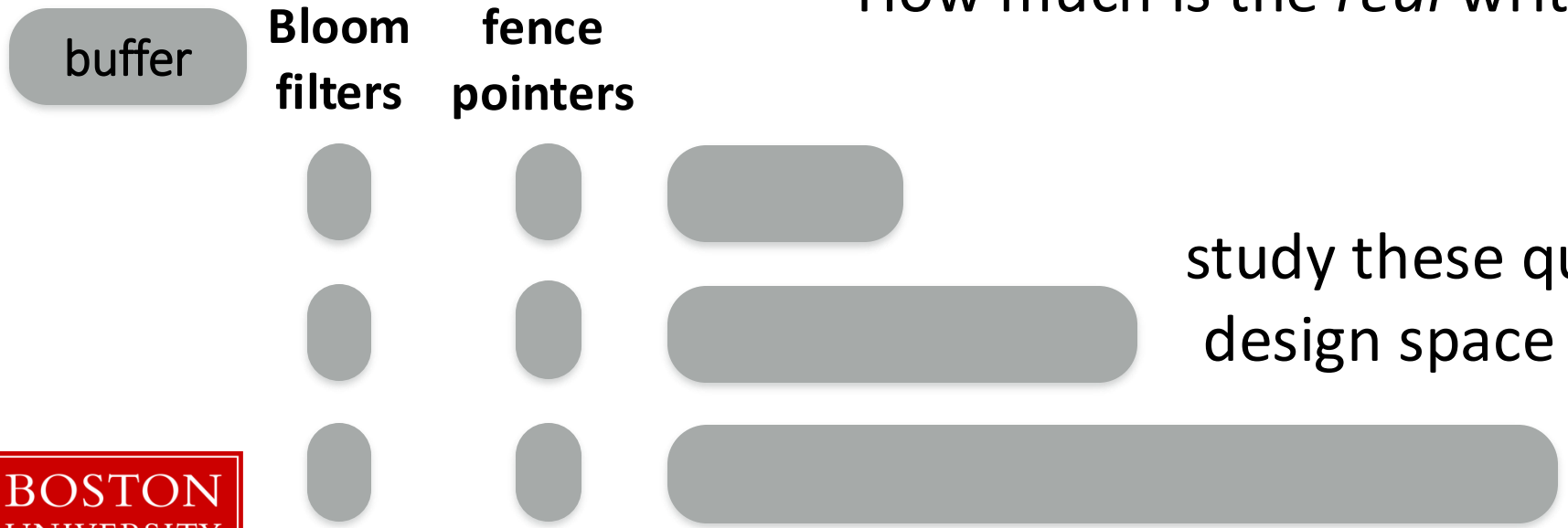
T : size ratio

Research Question on LSM-Trees

how can we minimize the duplicate space during compaction?

how to ensure that we can tune without sharing workload details?

How much is the *real* write-amplification on SSDs?



study these questions and navigate LSM design space using Facebook's RocksDB

BOSTON UNIVERSITY

RocksDB

Research on PostgreSQL

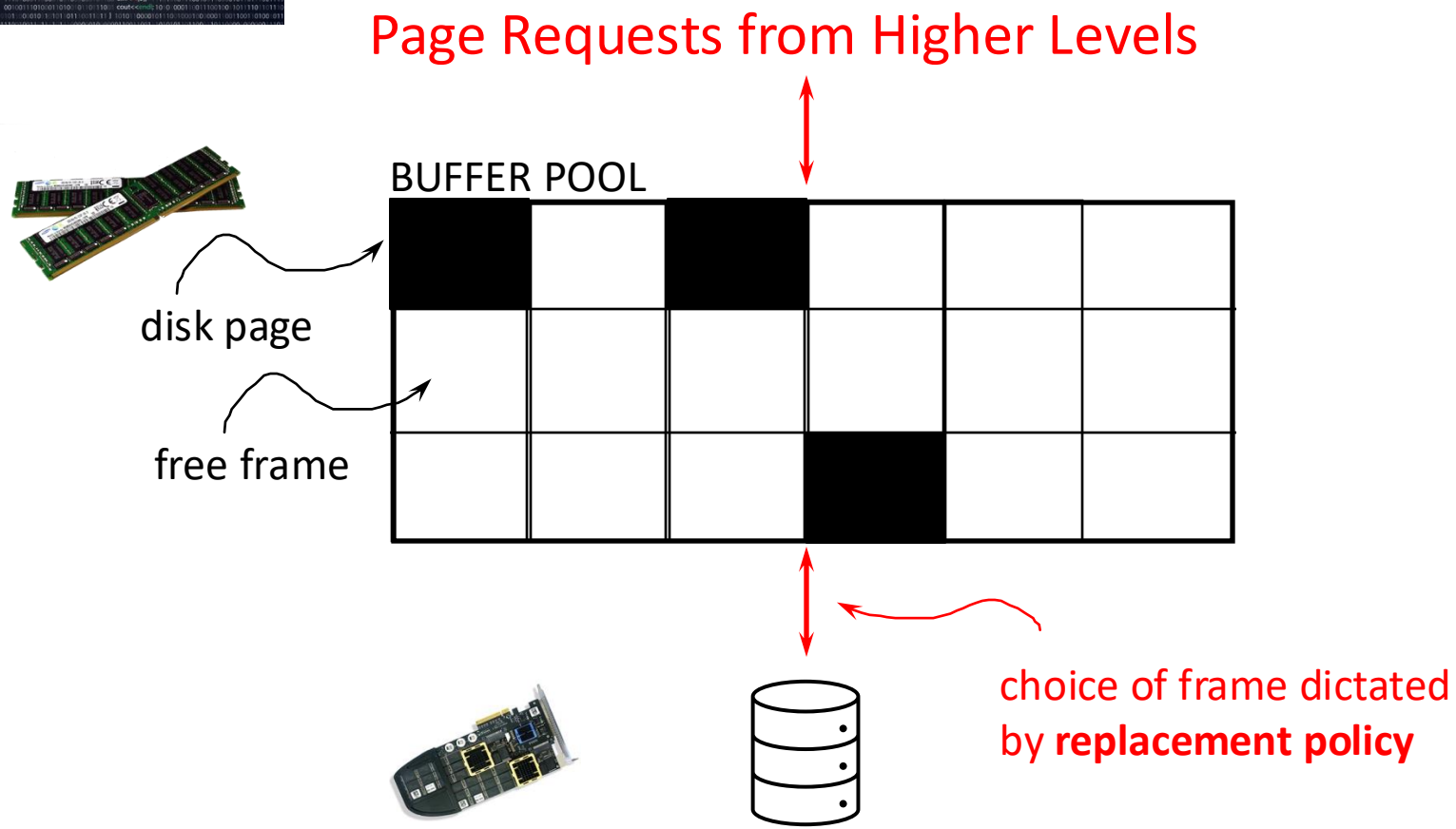
A state-of-the-art relational database

How can we implement a skew-aware efficient join algorithm?

How much a noise in the existing cardinality estimation can impact the selected query plan, and the overall query performance

BOSTON UNIVERSITY

Systems Project: Bufferpool



Implementation of a bufferpool

- **Application requests** a page
  - If **in the bufferpool** return it
  - If **not in the bufferpool** fetch it from the disk
    - If bufferpool is full select page to **evict**

Core Idea: Eviction Policy

- *Least Recently Used*
- *First In First Out*
- *more …*

BOSTON UNIVERSITY

Research Topics on Buffer Management & SSDs

How to deploy ACE buffer management on emulated SSDs?

How can we achieve storage parallelism in ZNS SSDs?

Other Research Topics on Indexing and beyond

How to apply sortedness-aware concepts on Adaptive Radix Tree?

How to build cache-friendly sortedness-aware indexing?

How to design a sortedness-aware join algorithm?

what to do now?

systems project

form groups of 3

(speak to me in OH if you want a different arrangement)

research project

form groups of 3

pick one of the subjects & read background material

define the behavior you will study and address

sketch approach and success metric

(if LSM-related get familiar with RocksDB)

what to do now?

systems project
form groups of 2
(speak to me in OH if you want to work on your own)

research project

come to OH/Labs

submit project 0 this Friday on 1/31

start working on project 1 (due on 2/14)

submit semester project proposal on 2/23

BOSTON UNIVERSITY

CS 561: **Data Systems Architectures**

class 4

Systems & Research Project

Prof. Manos Athanassoulis

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