# CS 561: Data Systems Architectures

class 4

# Systems & Research Project

Prof. Manos Athanassoulis

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

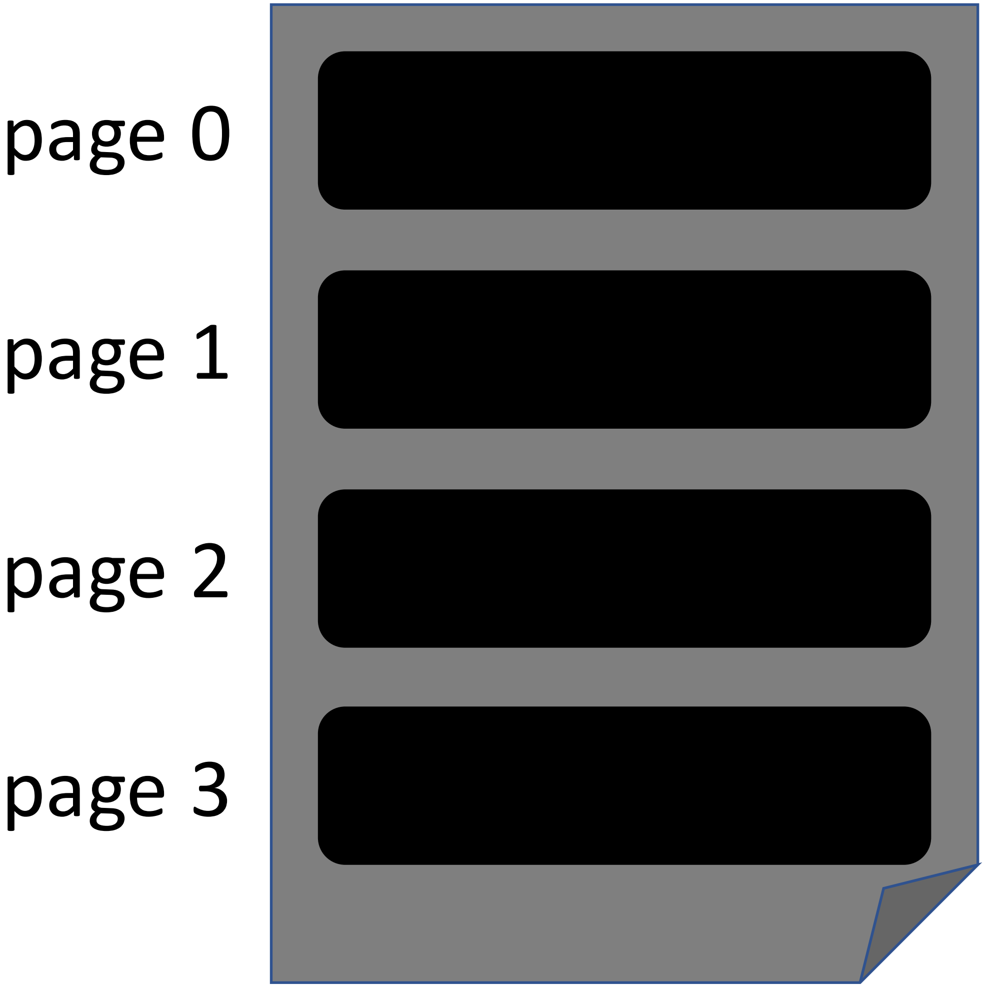
# Let’s revisit Zonemaps

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

# Let’s revisit Zonemaps

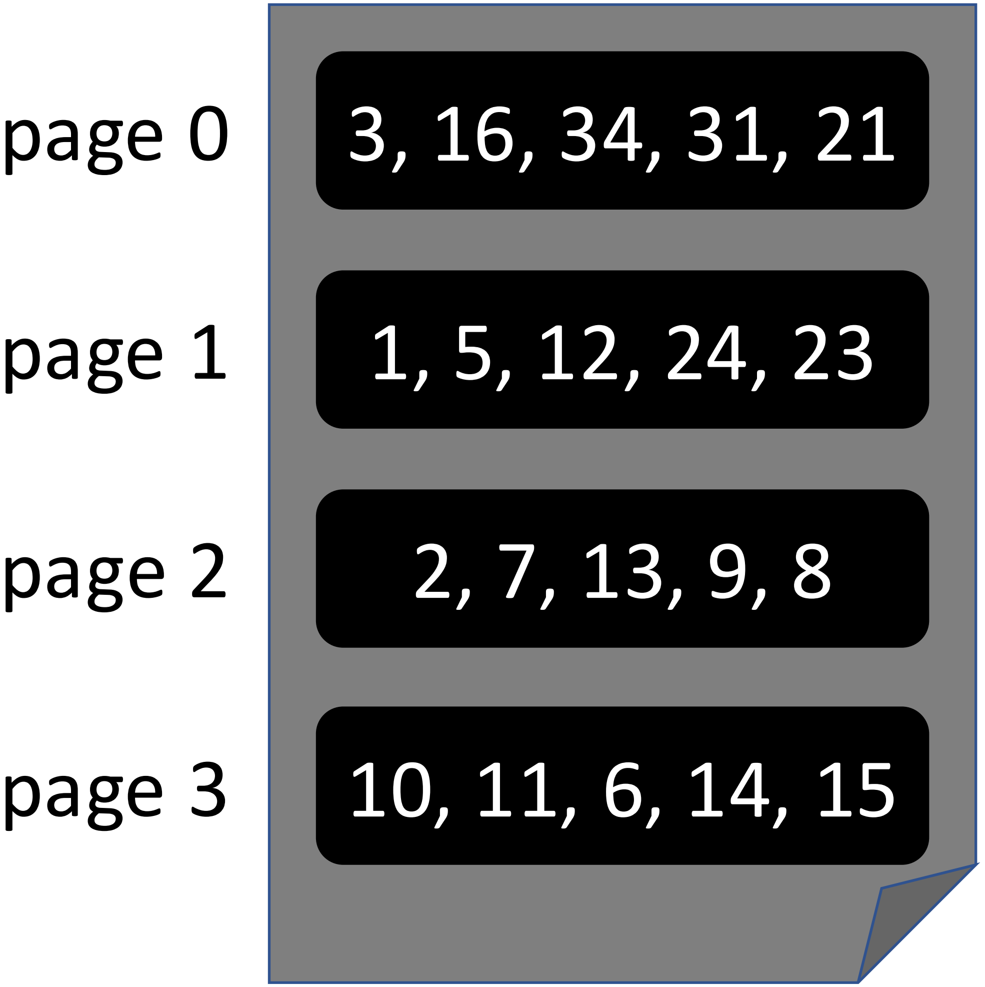
zonemaps

file = collection of pages



# zonemaps

file = collection of pages



page 0 3, 16, 34, 31, 21

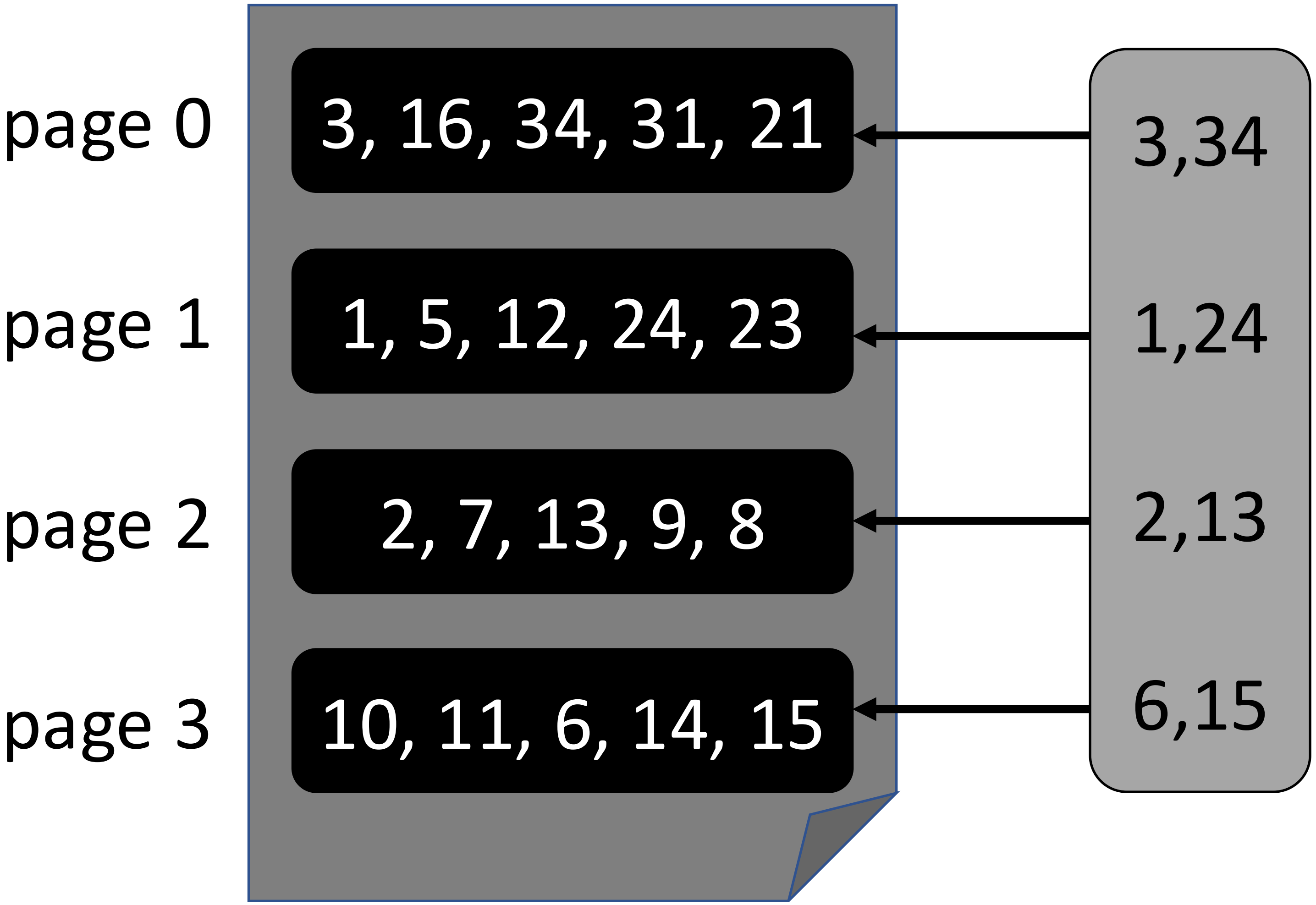
page 1 1, 5, 12, 24, 23

page 2 2, 7, 13, 9, 8

page 3 10, 11, 6, 14, 15

# zonemaps

file = collection of pages



page 0 3, 16, 34, 31, 21 ← 3,34

page 1 1, 5, 12, 24, 23 ← 1,24

page 2 2, 7, 13, 9, 8 ← 2,13

page 3 10, 11, 6, 14, 15 ← 6,15

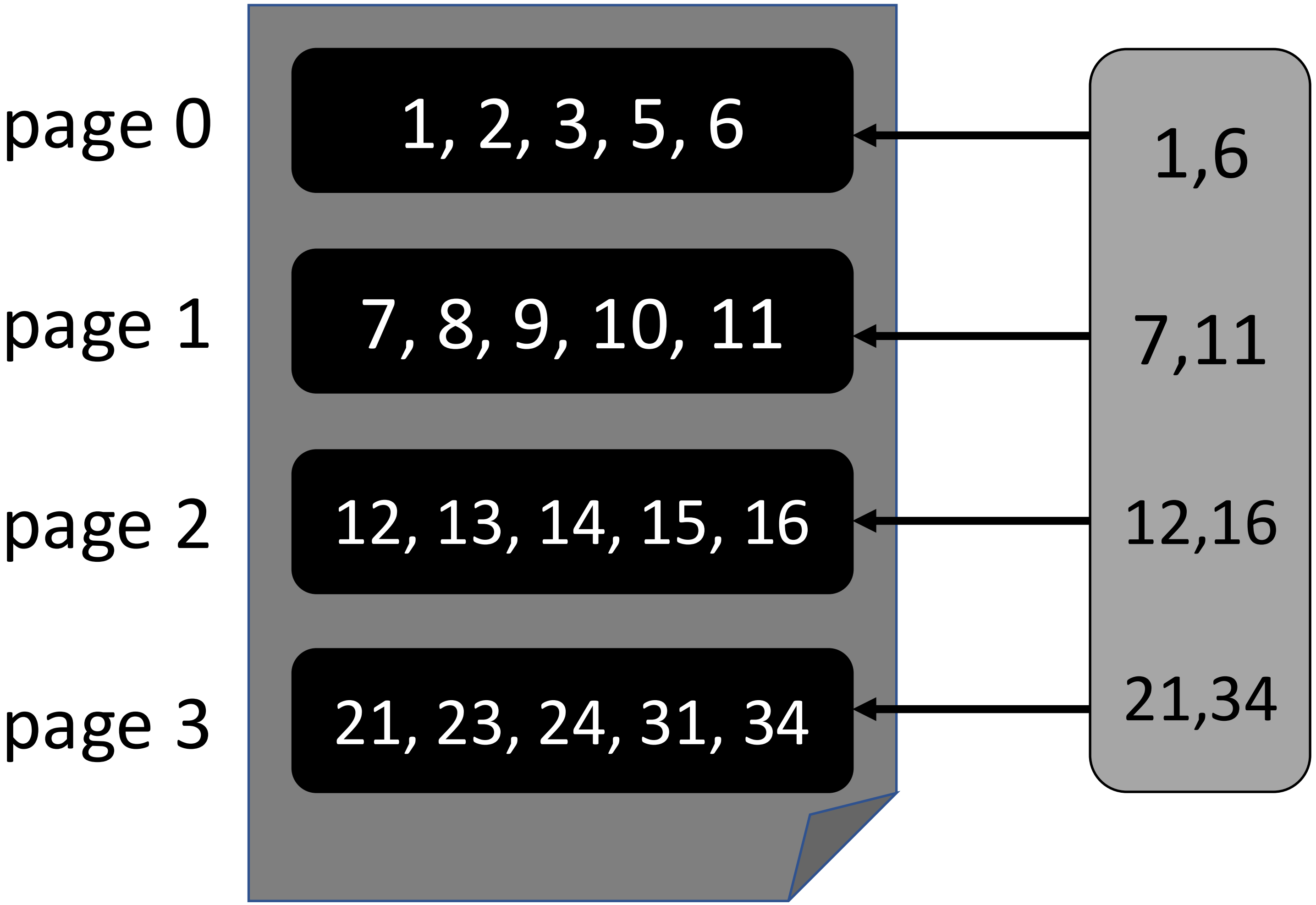
*light-weight*

*typically retained in memory*

But what if the data is sorted?

# zonemaps

file = collection of pages



page 0

1, 2, 3, 5, 6

page 1

7, 8, 9, 10, 11

page 2

12, 13, 14, 15, 16

page 3

21, 23, 24, 31, 34

1,6

7,11

12,16

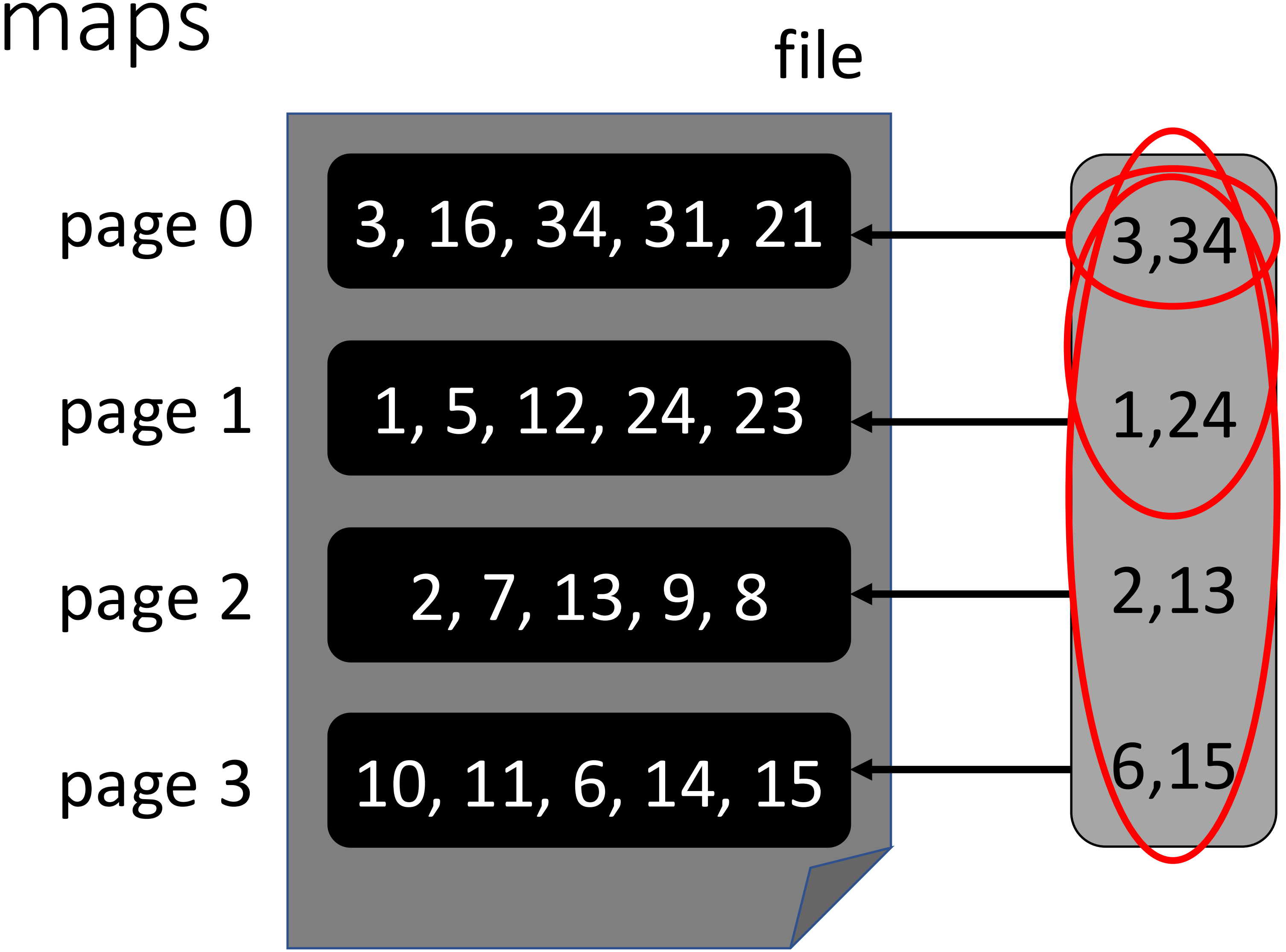
21,34

*light-weight*

*typically retained in memory*

But what if the data is sorted?

# zonemaps



file

page 0: 3, 16, 34, 31, 21 ← 3,34

page 1: 1, 5, 12, 24, 23 ← 1,24

page 2: 2, 7, 13, 9, 8 ← 2,13

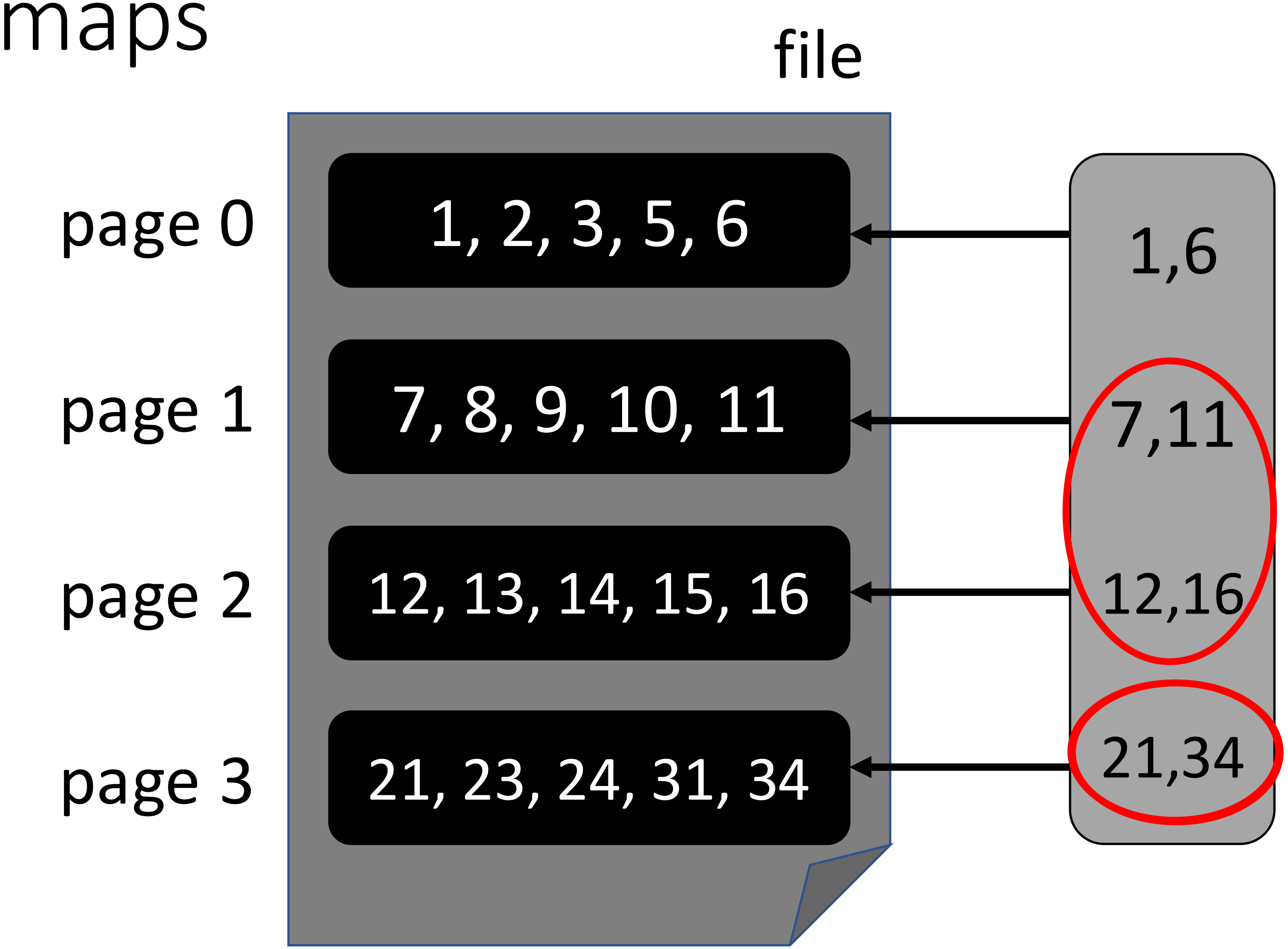
page 3: 10, 11, 6, 14, 15 ← 6,15

Range Queries

| | |
|---|---|
| [25, 100] | 1 page |
| [20, 25] | 2 pages |
| [10, 13] | 4 pages |

But what if the data is sorted?

# zonemaps



file

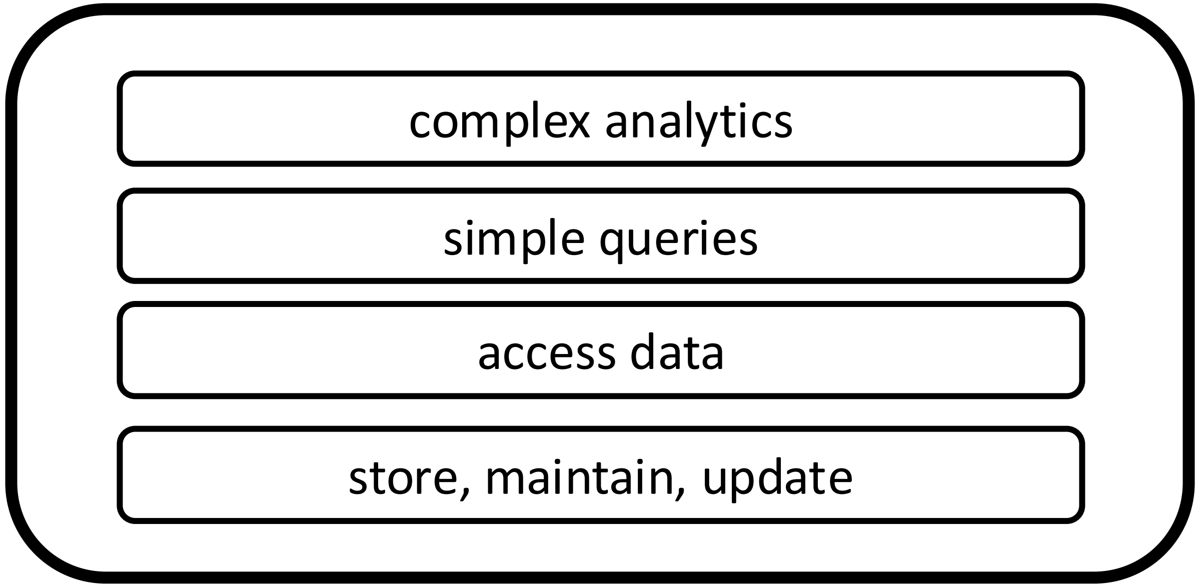
| | | |
|---|---|---|
| page 0 | 1, 2, 3, 5, 6 | 1,6 |
| page 1 | 7, 8, 9, 10, 11 | 7,11 |
| page 2 | 12, 13, 14, 15, 16 | 12,16 |
| page 3 | 21, 23, 24, 31, 34 | 21,34 |

Range Queries

| | |
|---|---|
| [25, 100] | 1 page |
| [20, 25] | 1 page |
| [10, 13] | 2 pages |

Zonemaps efficiency depends on data & queries!

# data systems



complex analytics

simple queries

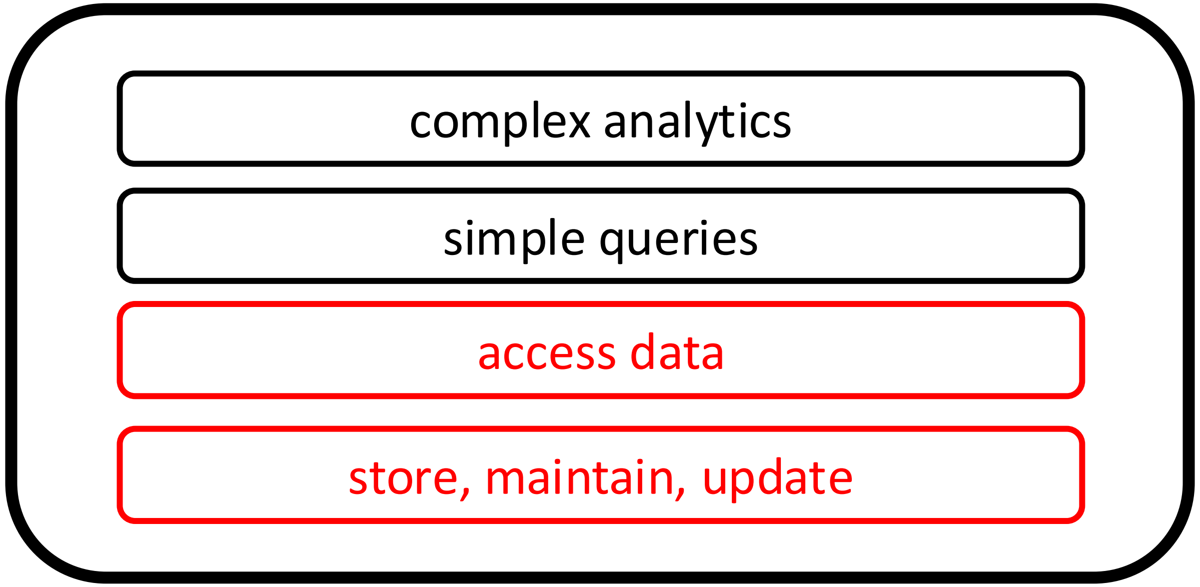
access data

store, maintain, update

# data systems

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

[The Forbes, 2016]


complex analytics

simple queries

access data

store, maintain, update

***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***

amazon DynamoDB

APACHE HBASE

Google BigTable

RocksDB

AsterixDB

Couchbase

Cassandra

mongoDB

***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***"

BOSTON UNIVERSITY

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

# how to use a key-value store?

***basic interface***

put(k,v)

{v} = get(k) $\{v_1, v_2, \ldots\}$ = get(k)

$\{v_1, v_2, \ldots\}$ = get_range($k_{min}, k_{max}$) $\{v_1, v_2, \ldots\}$ = full_scan()

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

more

*deletes: delete(k)*

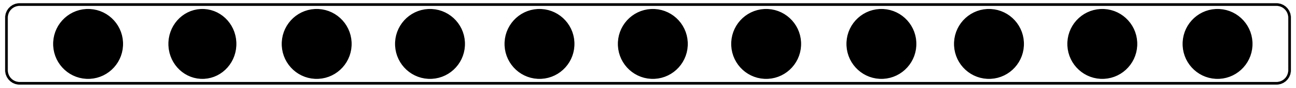
*updates: update(k,v)* is it different than put?

get set: $\{v_1, v_2, \ldots\}$ = get_set($k_1, k_2, \ldots$)

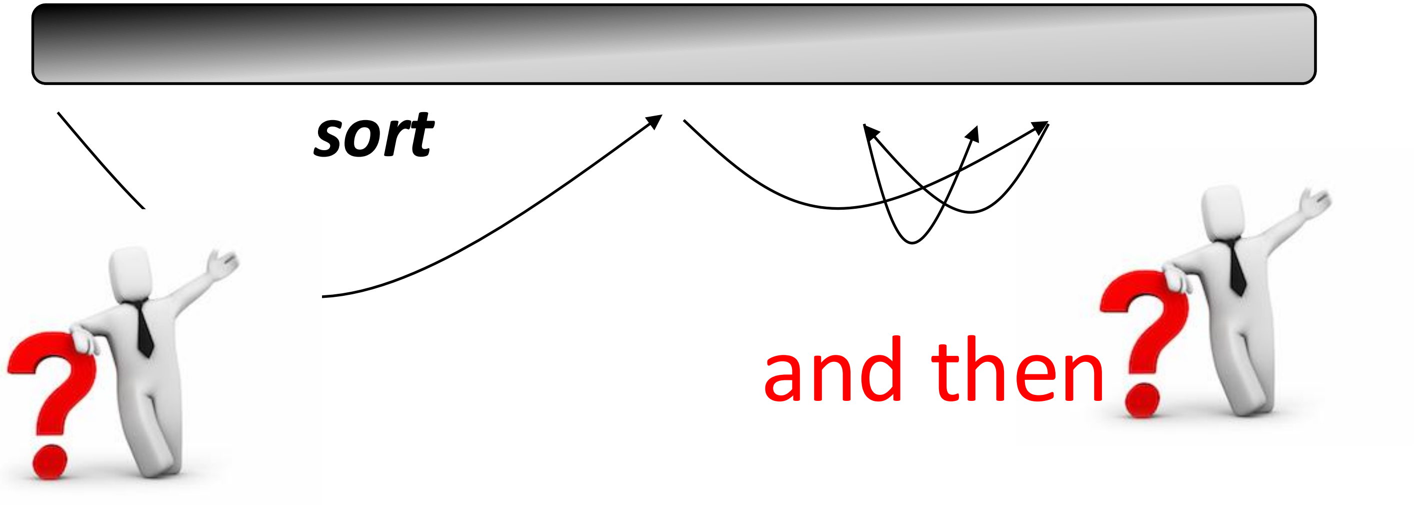
# how to build a key-value store?

***append***

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



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



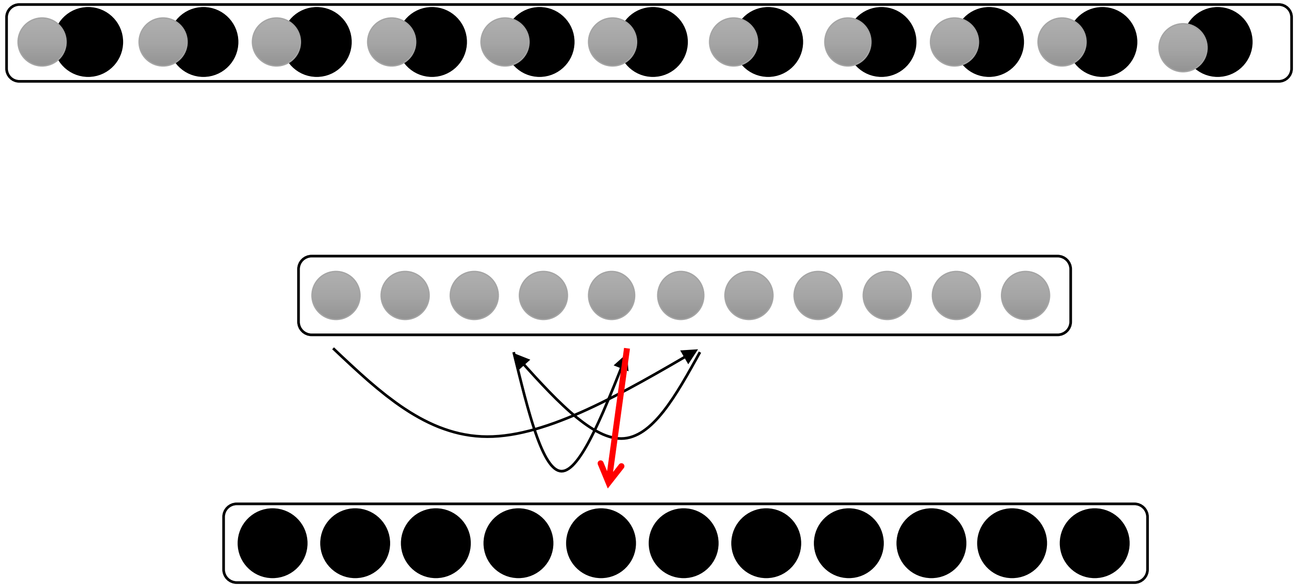
***sort***

and then

what about full scan?

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*

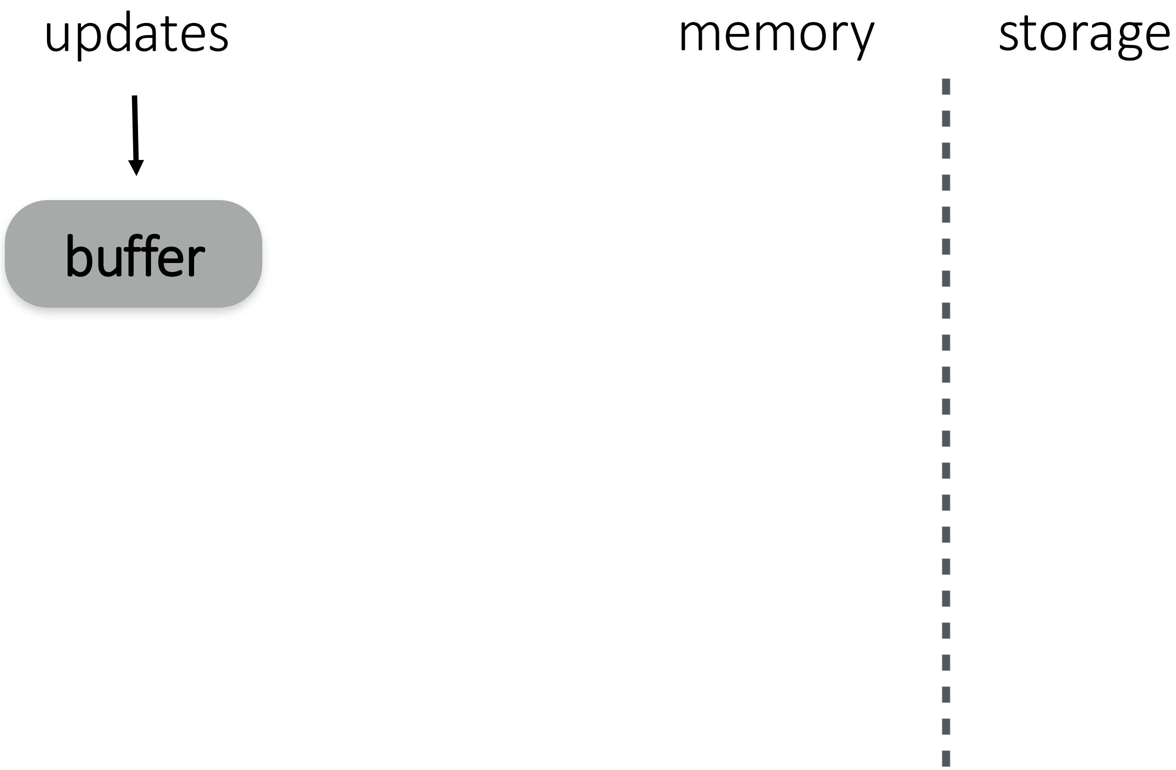
*avoid  resorting after every update*

how to bridge?

# LSM-tree
# Key-Value Stores

What are they really?

BOSTON UNIVERSITY

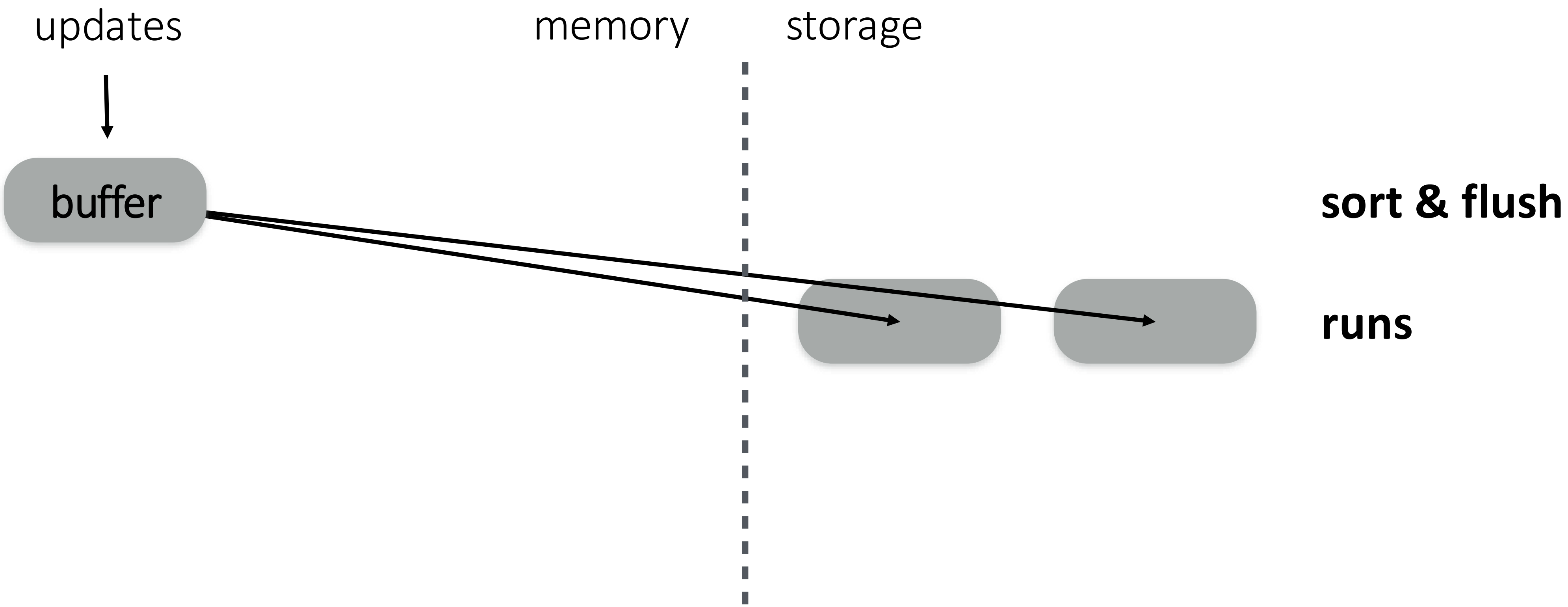


updates

buffer

memory

storage



updates

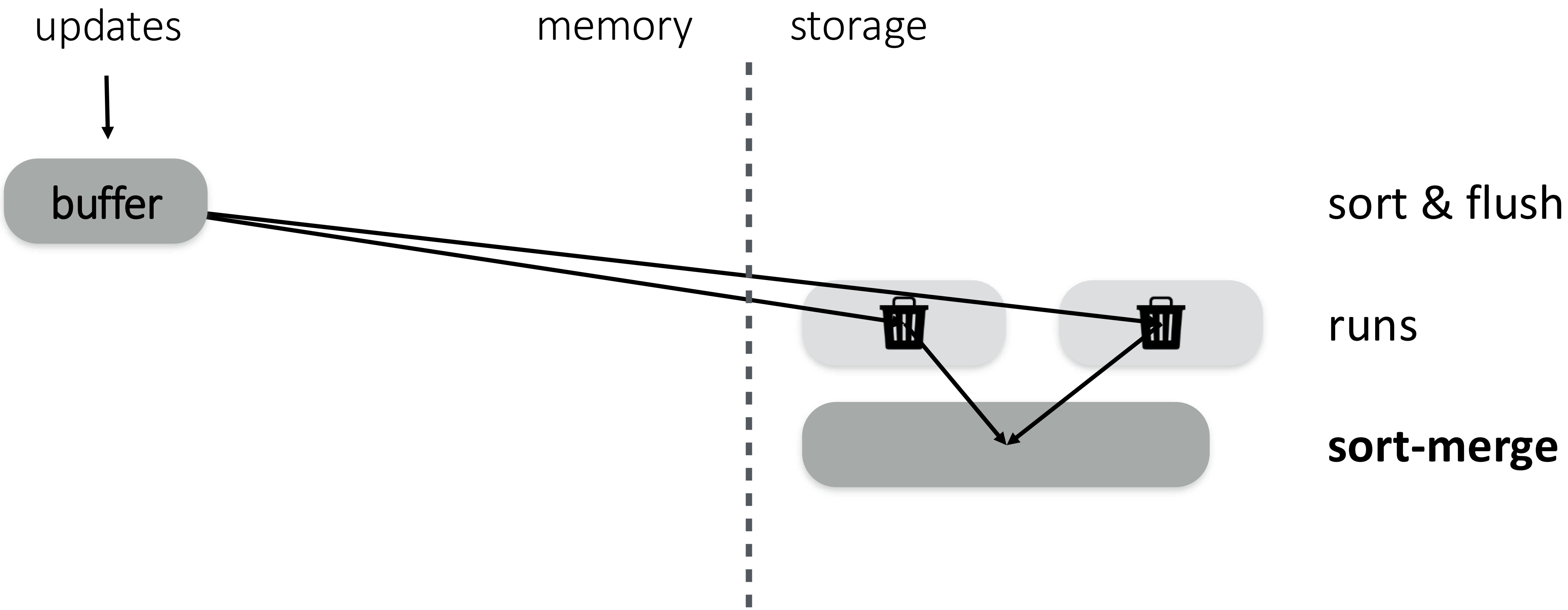
buffer

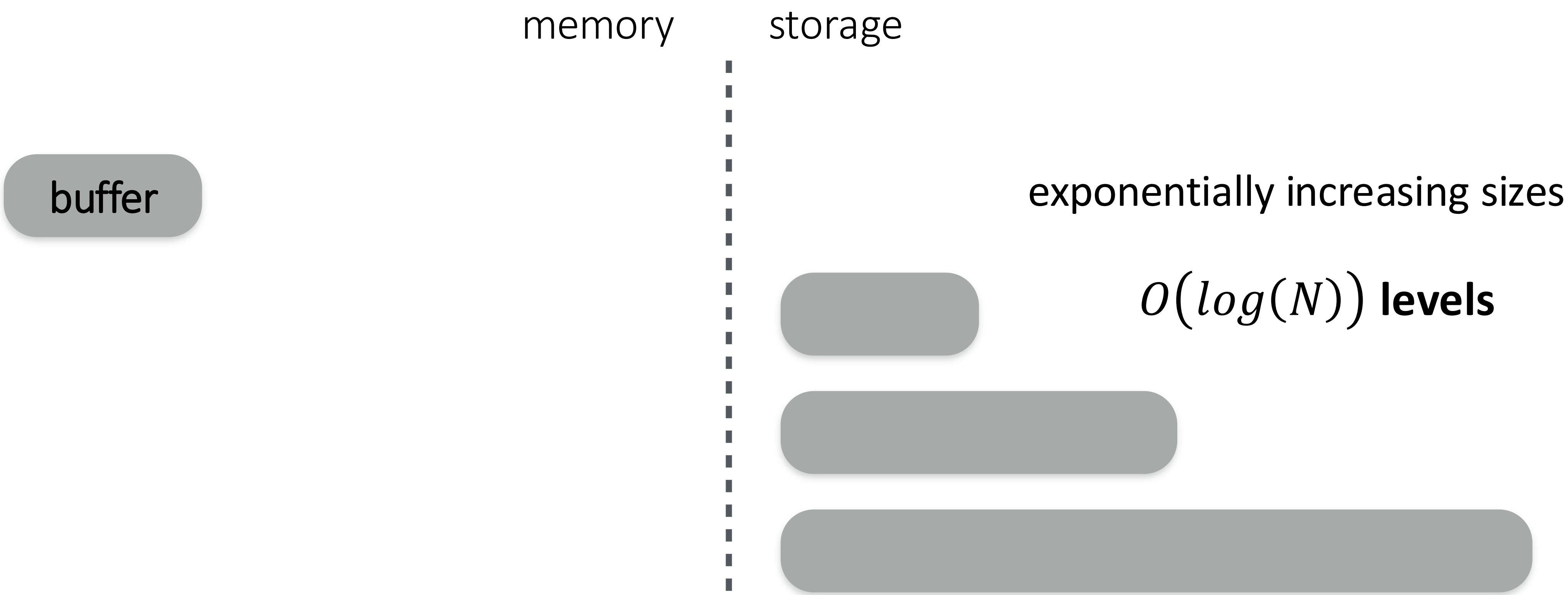
memory

storage

**sort & flush**

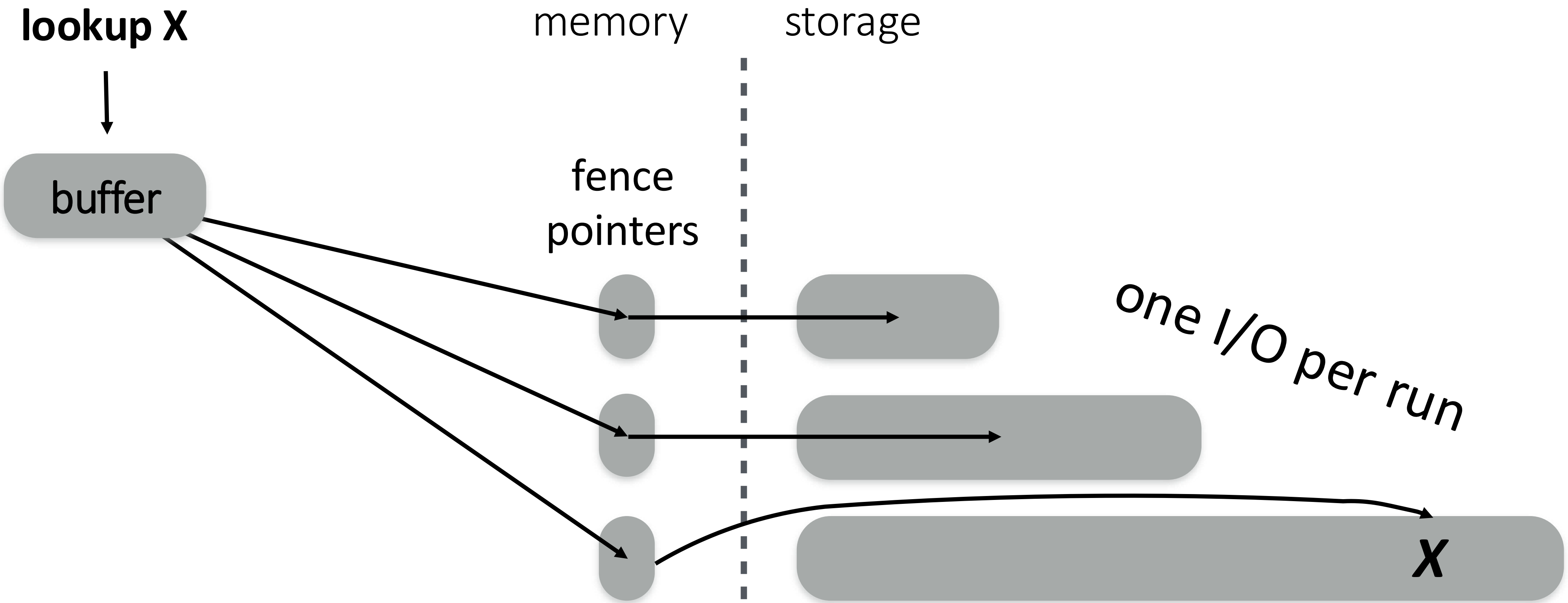
**runs**

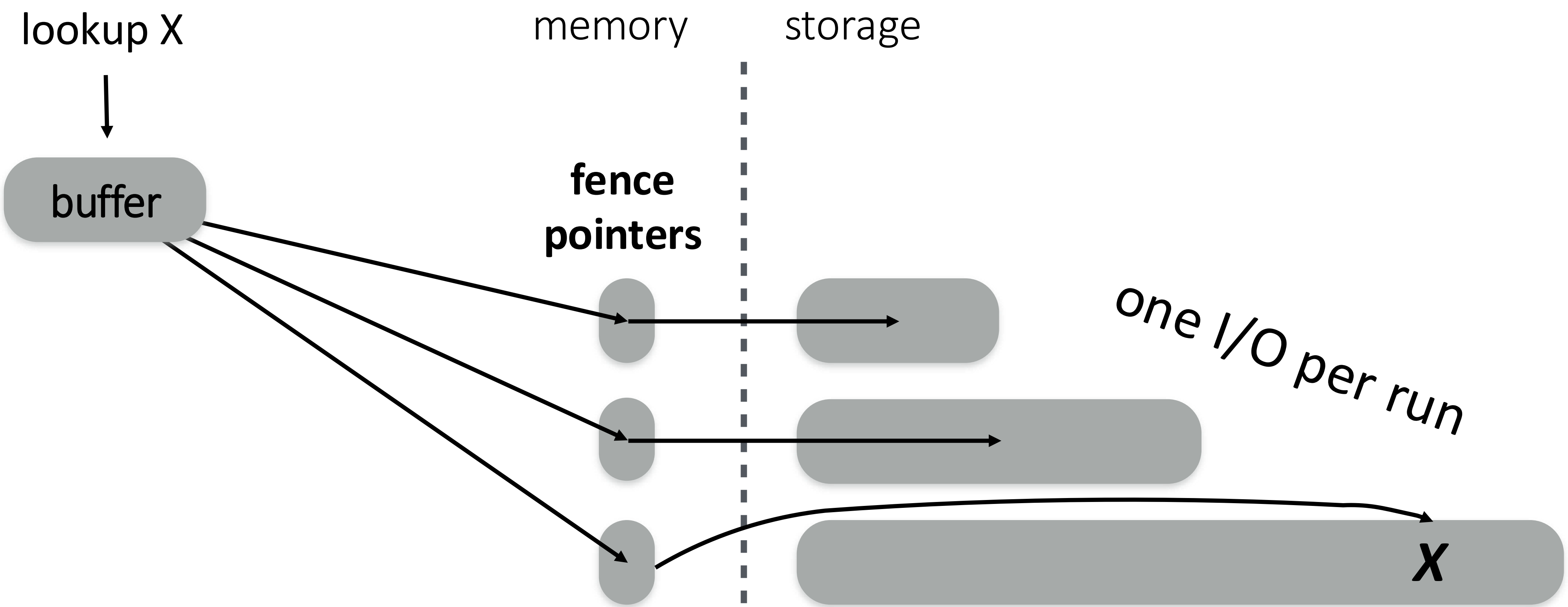

updates
memory
storage
buffer
sort & flush
runs
**sort-merge**

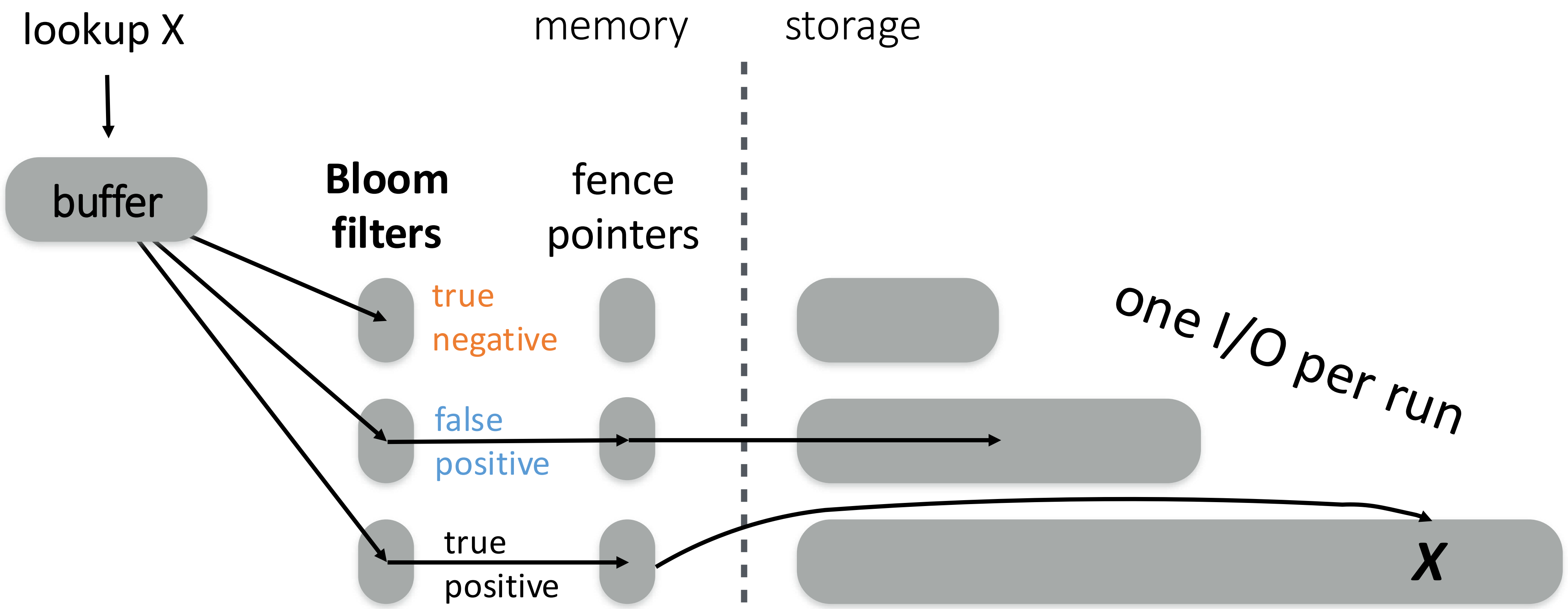


exponentially increasing sizes

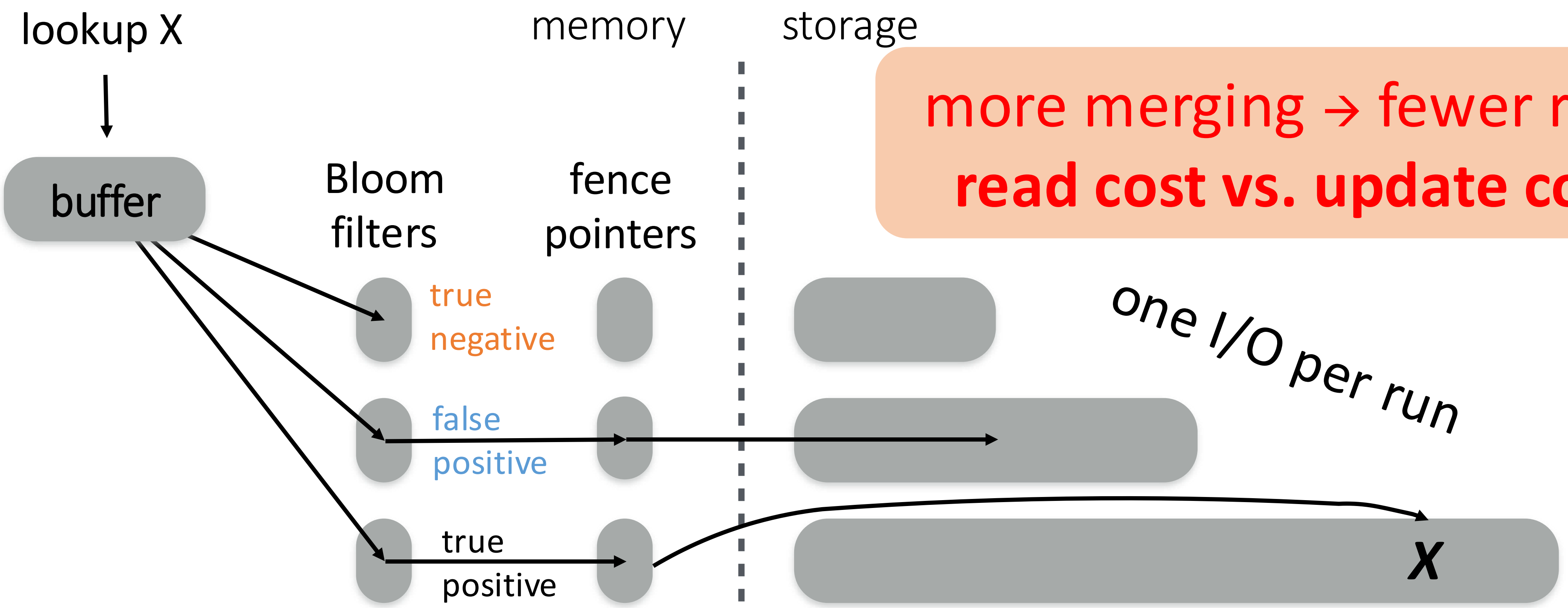
$O\left(log(N)\right)$ **levels**


**lookup X**
memory
storage
buffer
fence pointers
one I/O per run
*X*


lookup X
memory
storage
buffer
fence pointers
one I/O per run
X


lookup X
memory
storage
buffer
Bloom filters
fence pointers
true negative
false positive
true positive
one I/O per run
X

# performance & cost trade-offs



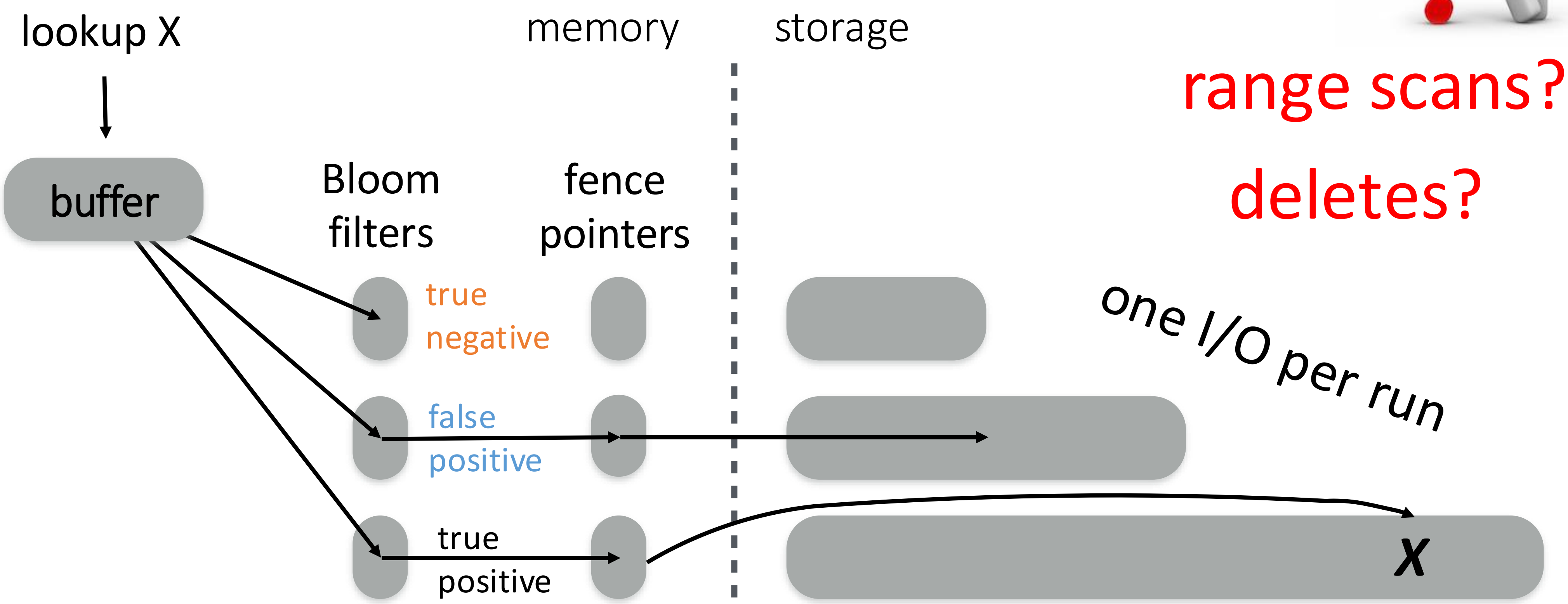
more merging → fewer runs

**read cost vs. update cost**

bigger filters → fewer false positives

**memory space vs. read cost**

# other operations

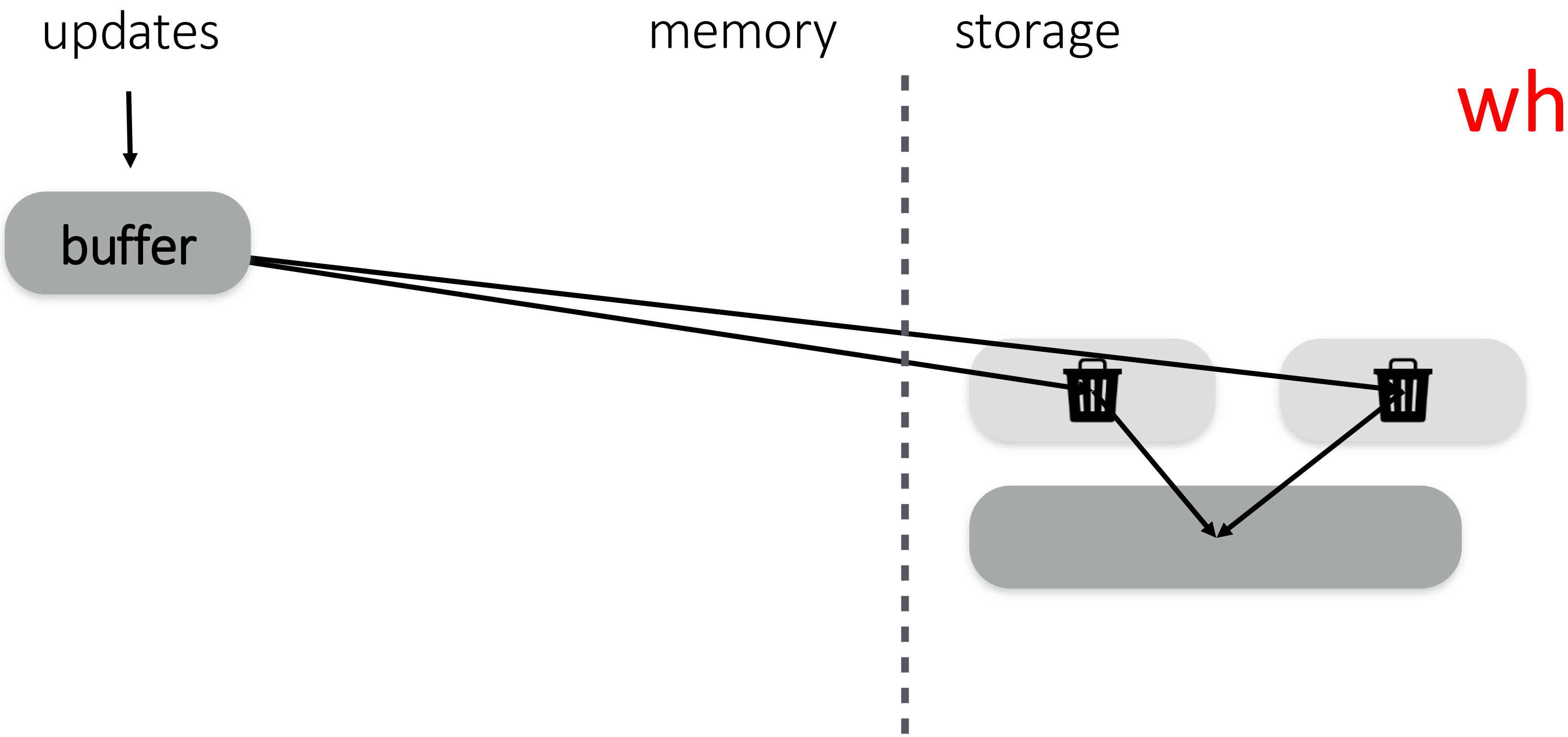


range scans?

deletes?

BOSTON UNIVERSITY

# remember merging?



what strategies?

sort & flush

runs

**sort-merge**

# Merge Policies

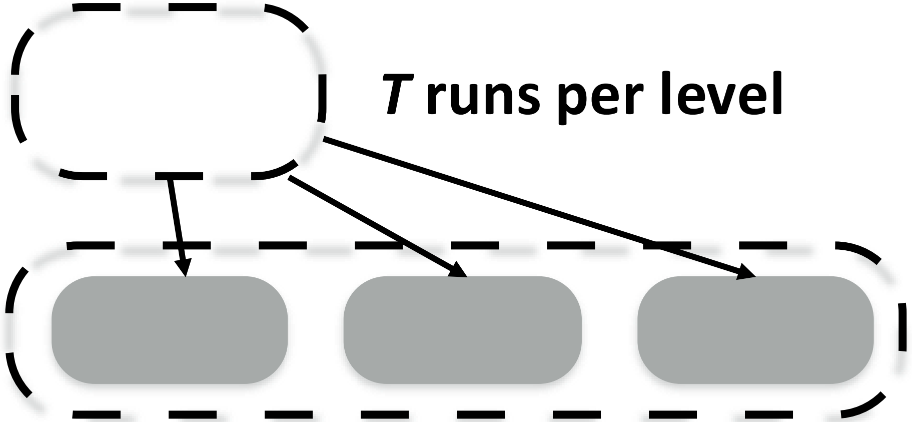
**Tiering**

write-optimized

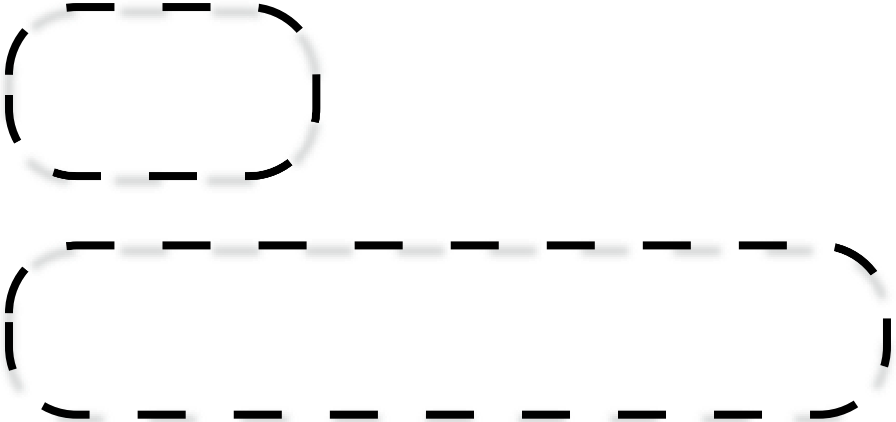
**Leveling**

read-optimized

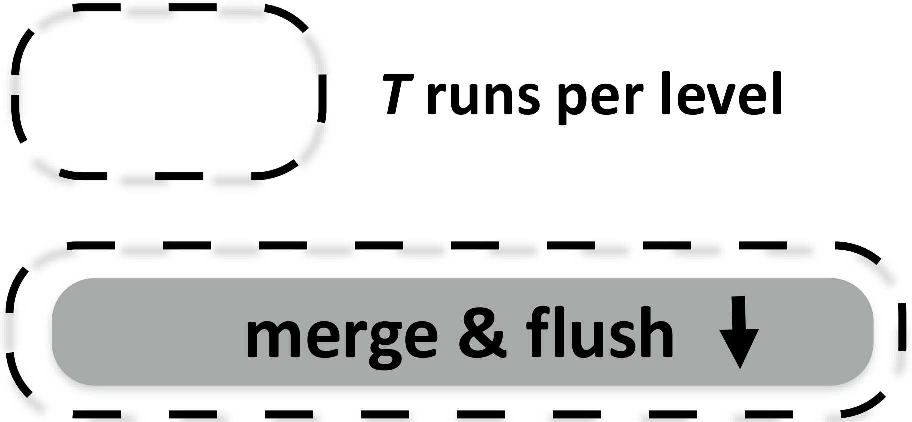
Tiering
write-optimized



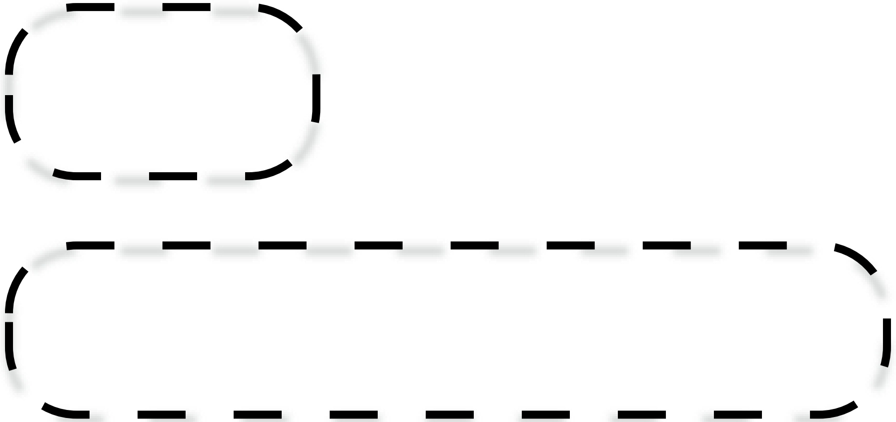
Leveling
read-optimized



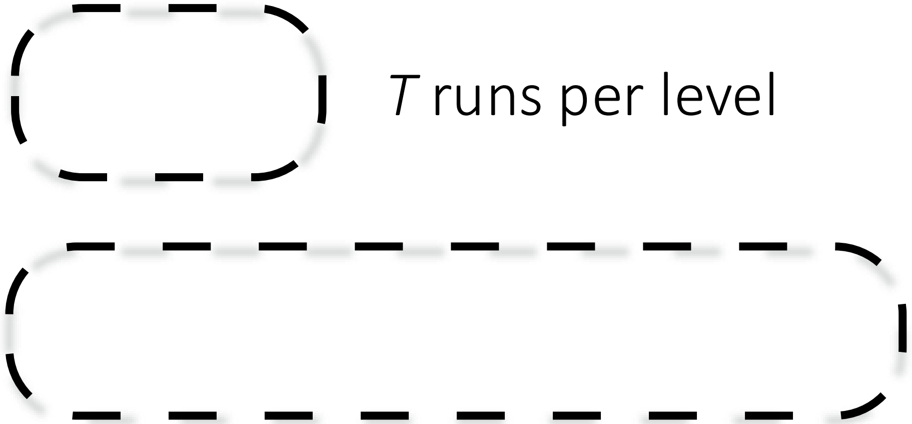
Tiering
write-optimized



Leveling
read-optimized

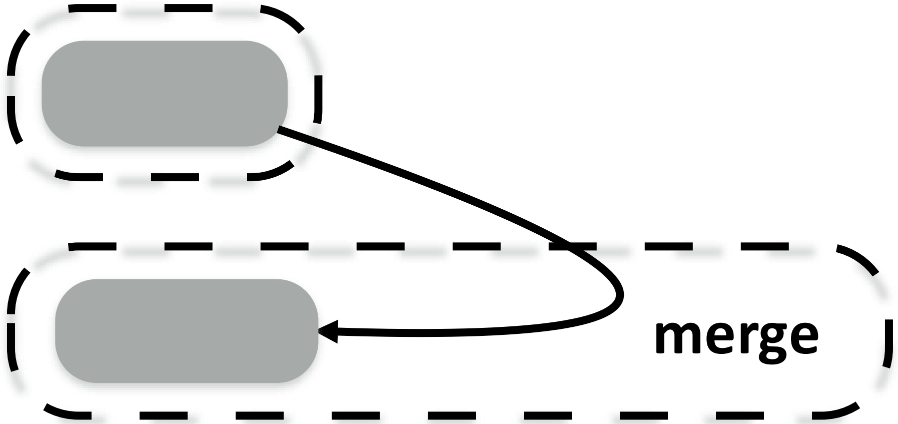


Tiering
write-optimized

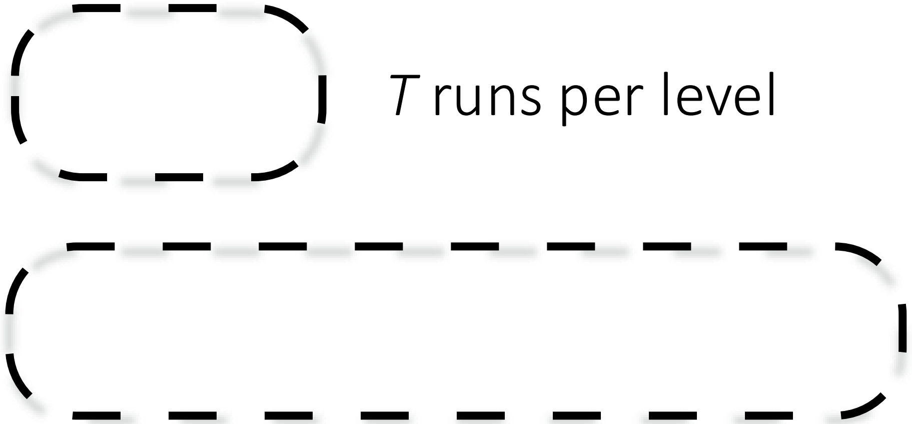


*T* runs per level

Leveling
read-optimized

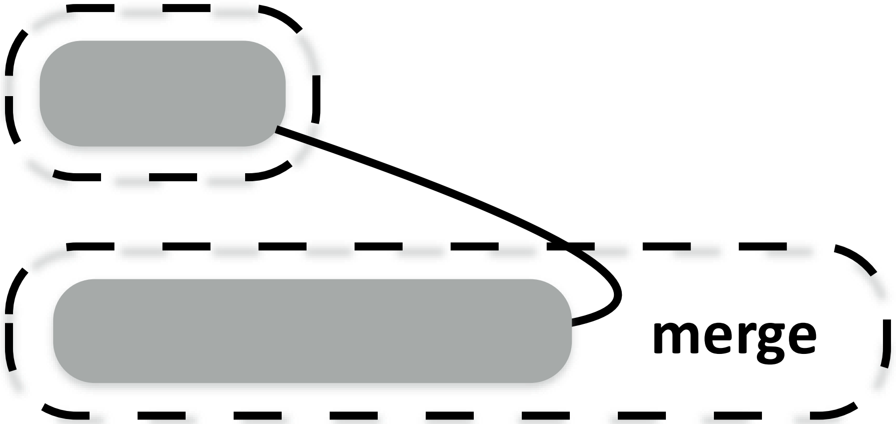


## Tiering
## write-optimized

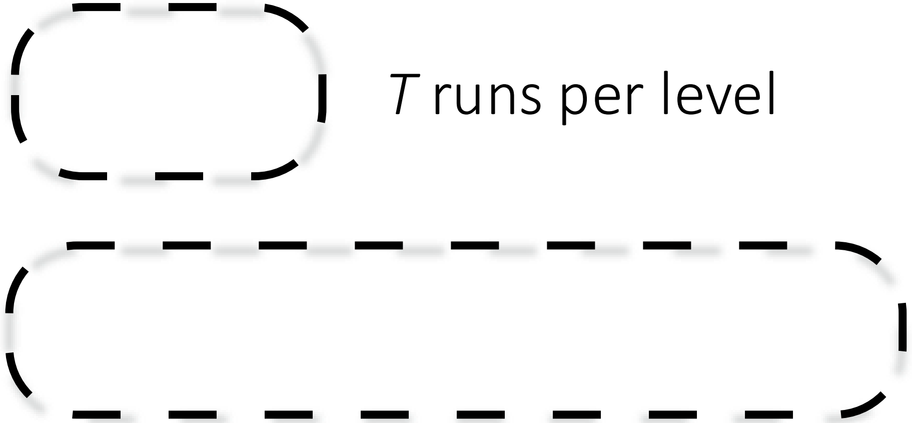


*T* runs per level

## Leveling
## read-optimized

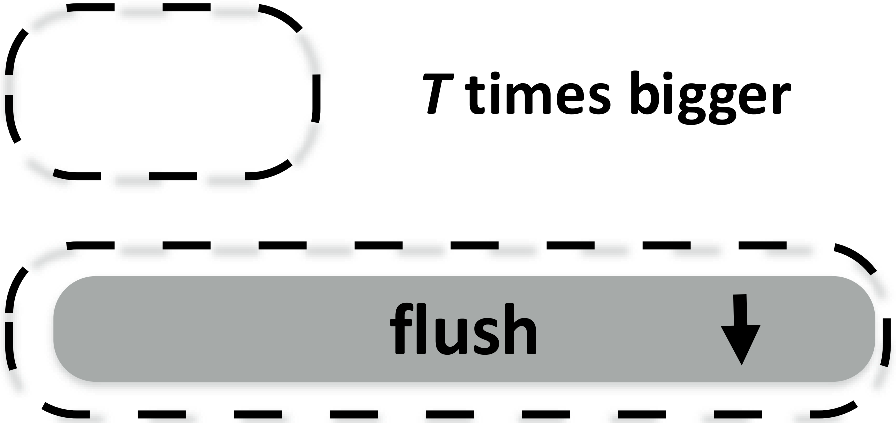


## Tiering
## write-optimized



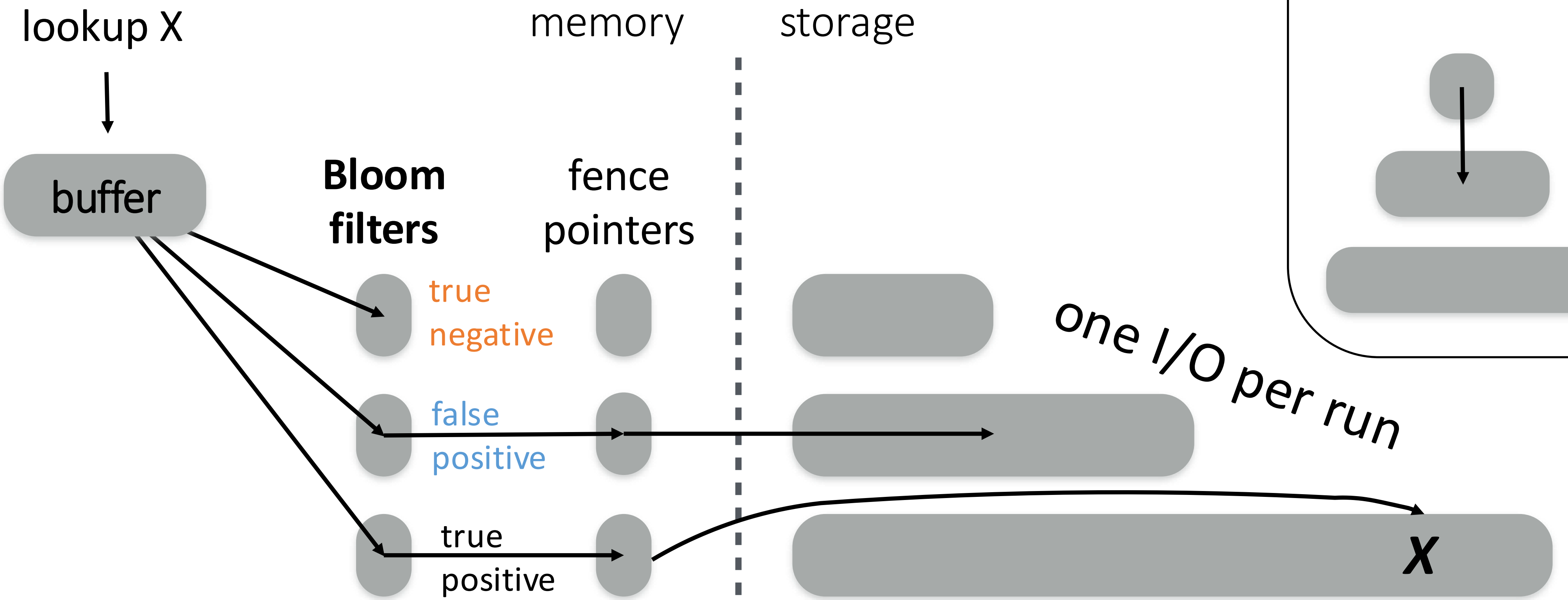
*T* runs per level

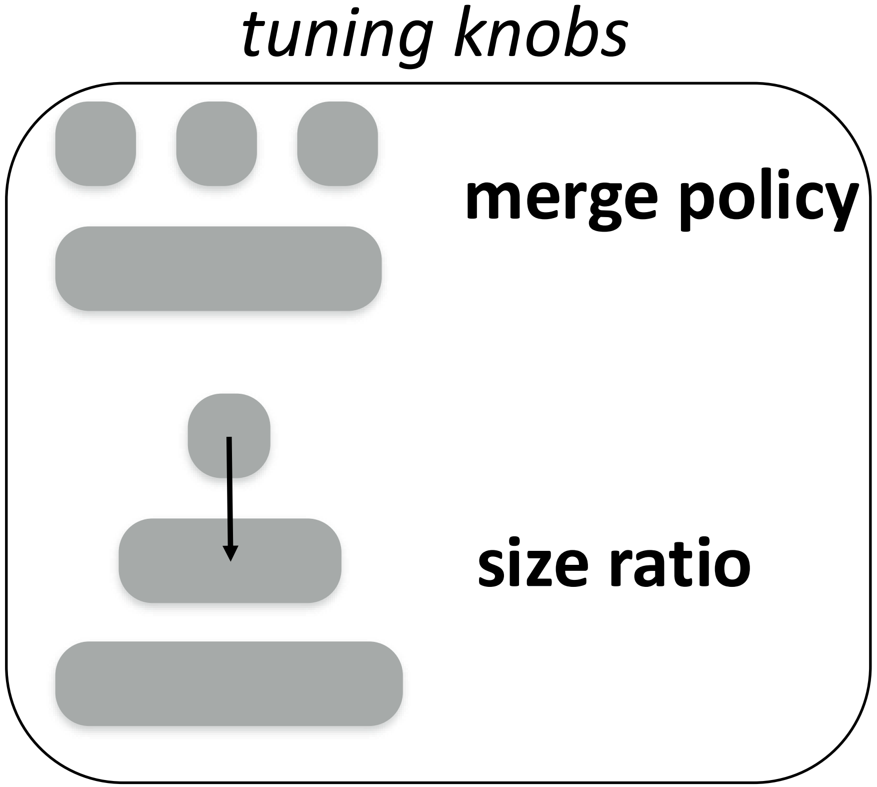
## Leveling
## read-optimized



***T* times bigger**

# Systems Project: LSM-Trees

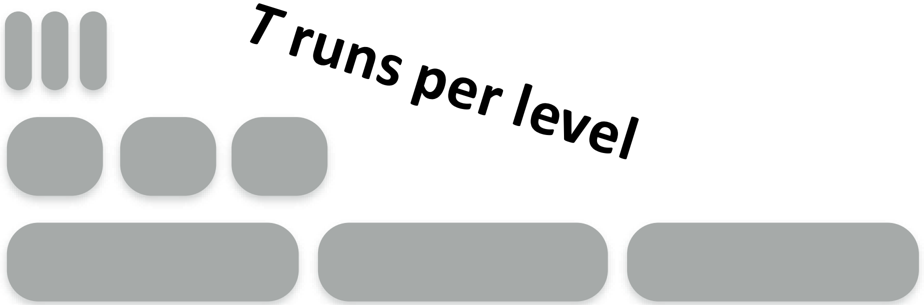




BOSTON UNIVERSITY

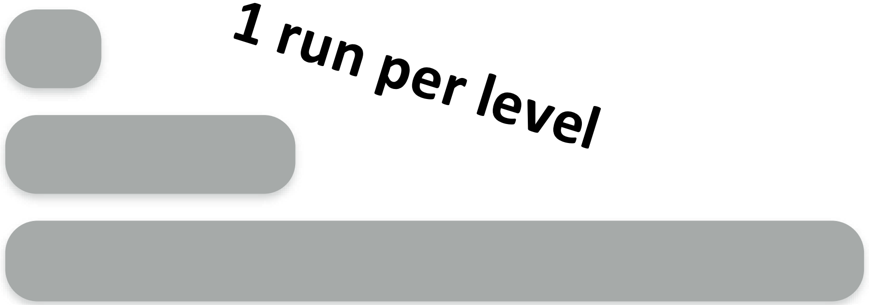
# more on LSM-Tree performance

# Tiering
write-optimized



*T* runs per level

# Leveling
read-optimized



1 run per level

**lookup cost:**

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

runs
per level

levels

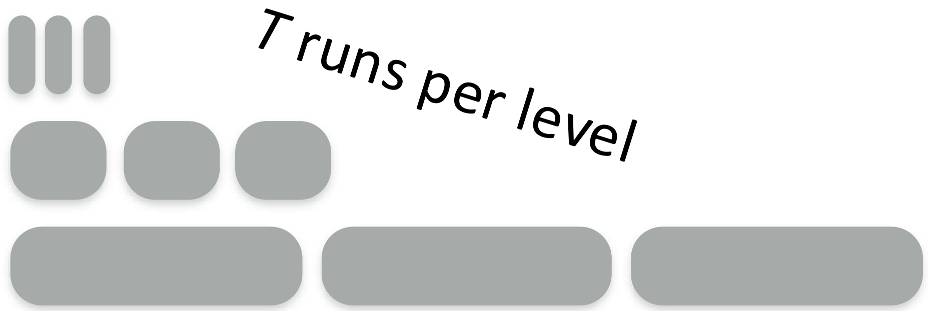
false
positive rate

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

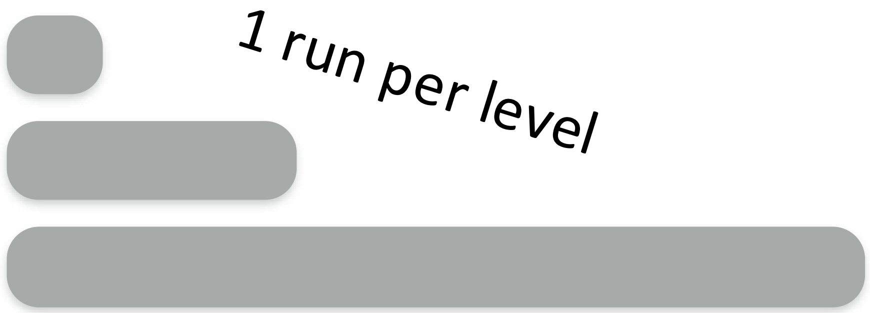
levels

false
positive rate

## Tiering
## write-optimized

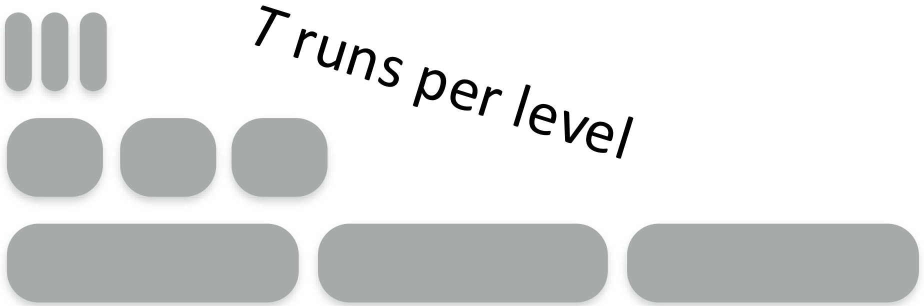


## Leveling
## read-optimized

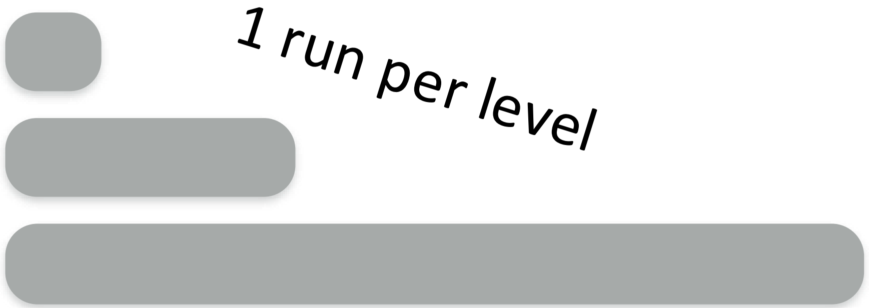


| | Tiering | Leveling |
|---|---|---|
| lookup cost: | $O\left(T \cdot log_T(N) \cdot e^{-M/N}\right)$ | $O\left(log_T(N) \cdot e^{-M/N}\right)$ |
| **update cost:** | $\boldsymbol{O\left(\underbrace{log_T}_{\text{levels}}(N)\right)}$ | $\boldsymbol{O\left(\underbrace{T}_{\text{merges per level}} \cdot \underbrace{log_T(N)}_{\text{levels}}\right)}$ |

## Tiering
## write-optimized



## Leveling
## read-optimized

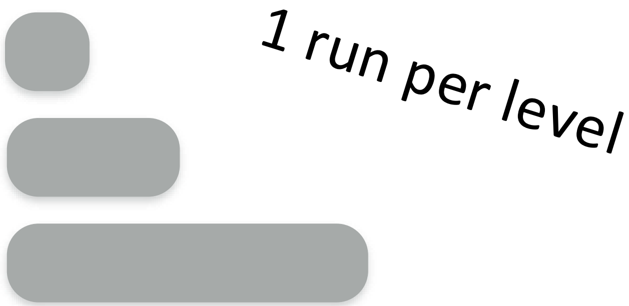


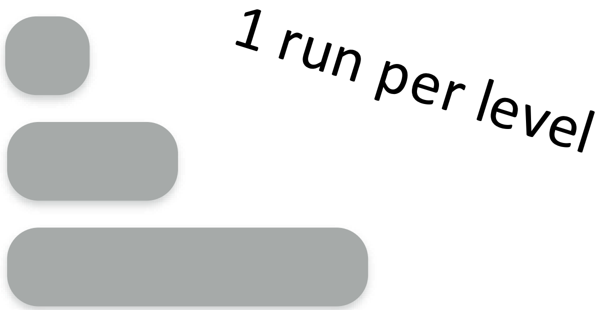
| | Tiering | Leveling |
|---|---|---|
| lookup cost: | $O\left(T \cdot log_T(N) \cdot e^{-M/N}\right)$ | $O\left(log_T(N) \cdot e^{-M/N}\right)$ |
| update cost: | $O\left(log_T(N)\right)$ | $O\left(T \cdot log_T(N)\right)$ |

**for size ratio T** ≫

Tiering
write-optimized

Leveling
read-optimized



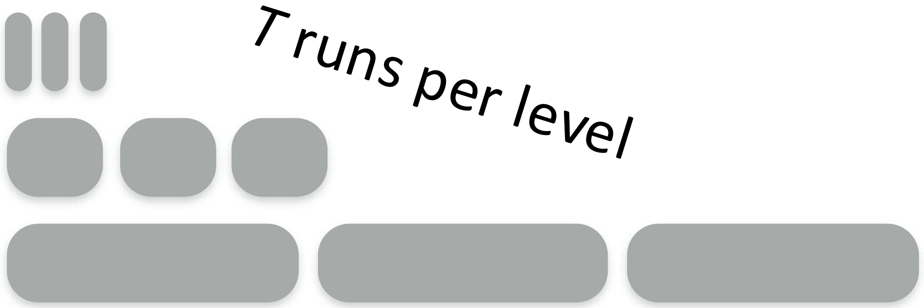


lookup cost: $O\left(log_T(N) \cdot e^{-M/N}\right) = O\left(log_T(N) \cdot e^{-M/N}\right)$

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

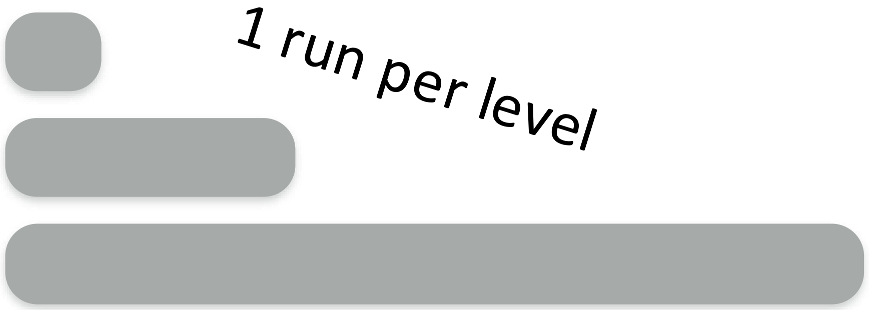
**for size ratio T**

## Tiering
## write-optimized



*T runs per level*

## Leveling
## read-optimized



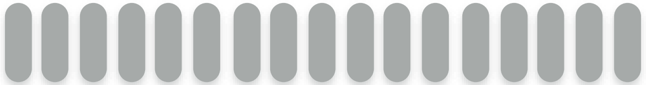
*1 run per level*

| | Tiering | Leveling |
|---|---|---|
| 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**

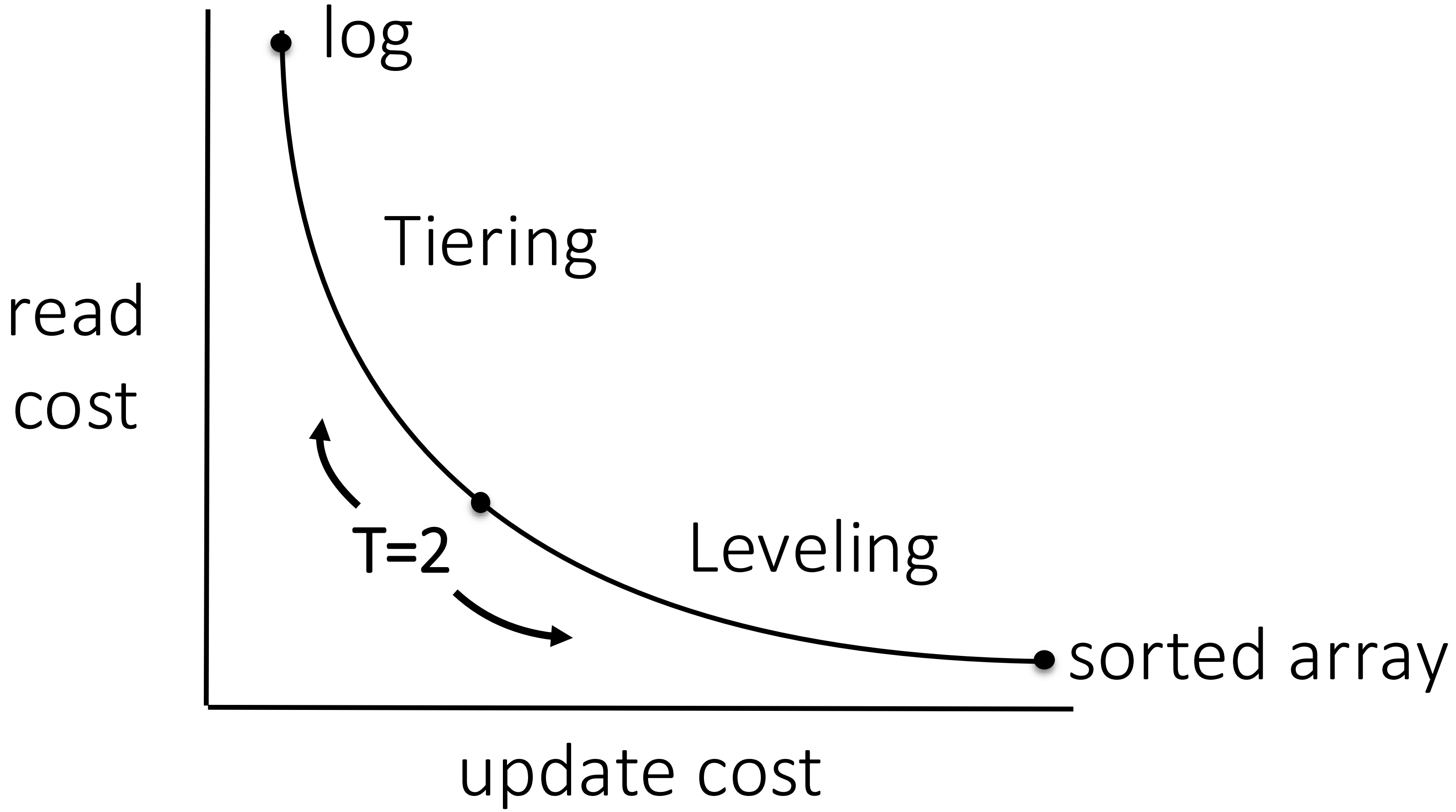
Leveling
read-optimized

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**



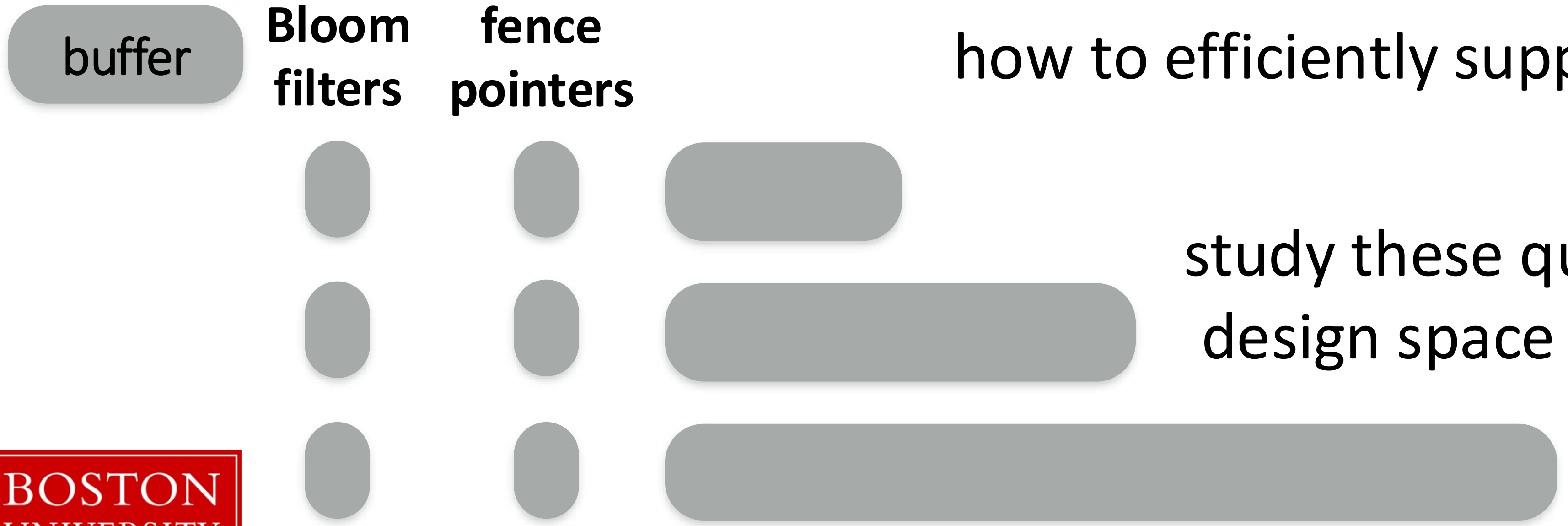
T : size ratio

# Research Question on LSM-Trees

how to reduce temporary space amplification at compaction time?

how to support variable deletion persistent thresholds?

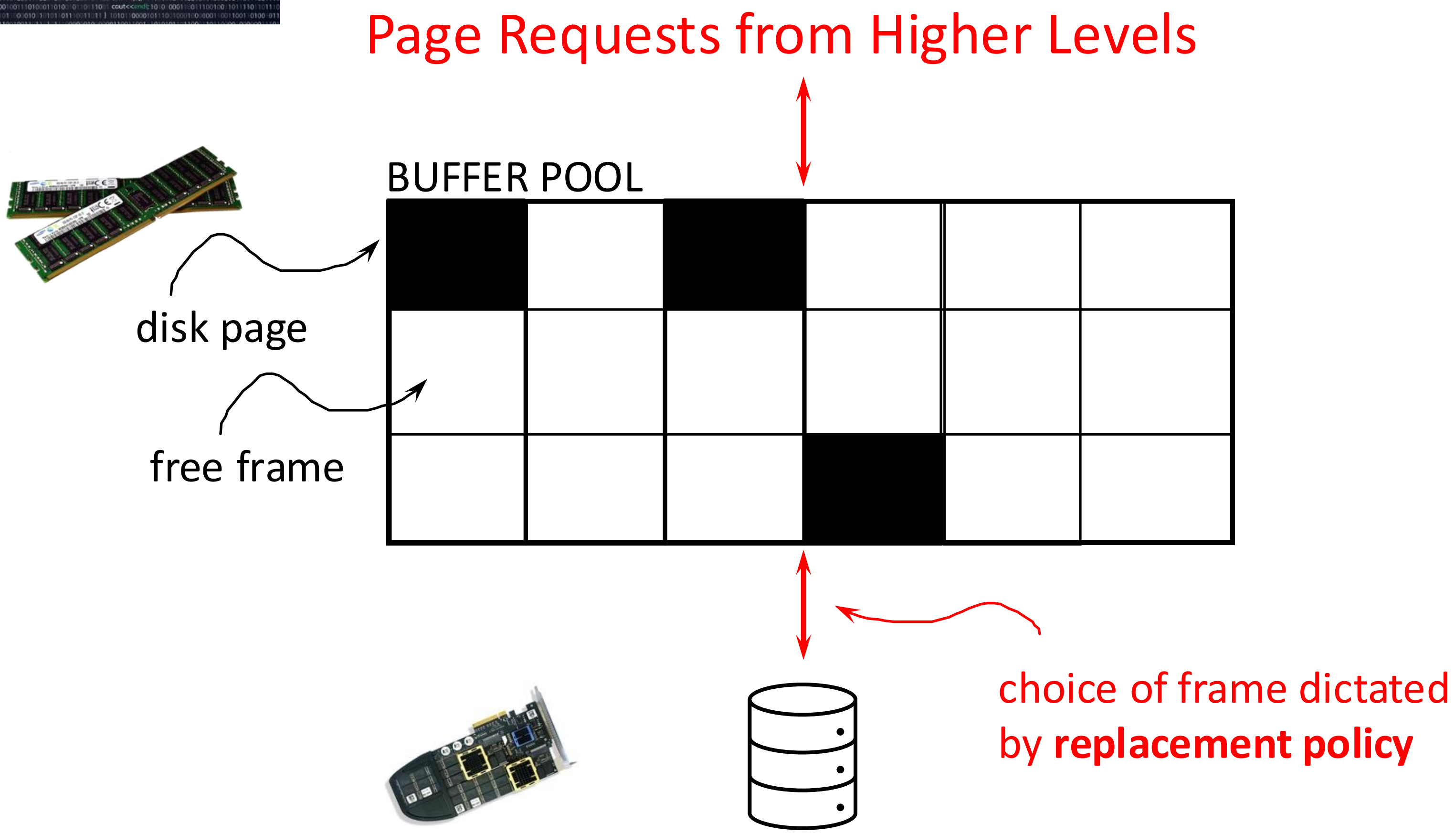
how to address high number of contiguous tombstones?



how to efficiently support range deletes?

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

# Systems Project: Bufferpool


Page Requests from Higher Levels
BUFFER POOL
disk page
free frame
choice of frame dictated by **replacement policy**

## *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 …*

# Research on PostgreSQL

PostgreSQL

***A state-of-the-art relational database***

How can we implement a sortendess-aware join algorithm?

Integrate light-weight eviction policies in PostgreSQL bufferpool

BOSTON UNIVERSITY

# Research Topics on SSDs

how to design a flexible ZNS SSD interface?

understanding Garbage Collection in applications on ZNS SSDs

understanding the impact of Zone Sizes on ZNS SSDs

# Research Topics on Bitmap Indexing

how to do access path selection in the presence of bitmap indexes?

how to execute index-based joins with bitmap indexes?

# 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**
form groups of 2

*come to OH/Labs*

submit project 0 this Sunday on 2/1

start working on project 1 (due on 2/15)

submit semester project proposal on 2/22

BOSTON UNIVERSITY

# CS 561: Data Systems Architectures

class 4

## Systems & Research Project

Prof. Manos Athanassoulis

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