

CS660: Intro to Database Systems

Class 10: Log-Structured-Merge Trees

Instructor: Manos Athanassoulis

<https://bu-disc.github.io/CS660/>

Reads vs Writes: The two extremes

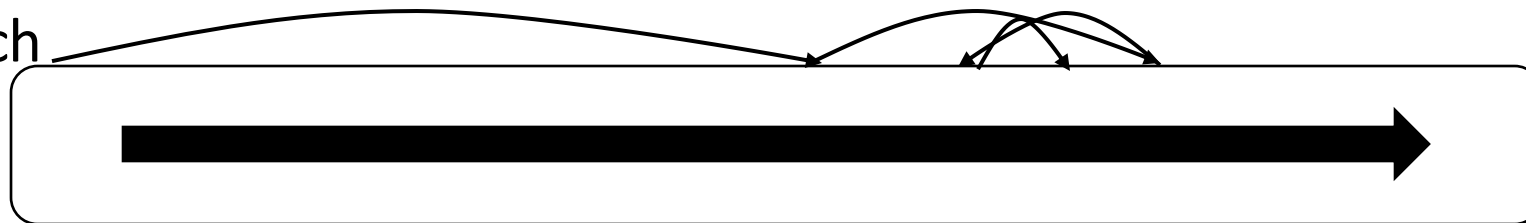
Assume **no index** – what is the **best way to physical store** our data?



Case 1: I have a static datasets and I **only receive reads**

how to read?

binary search

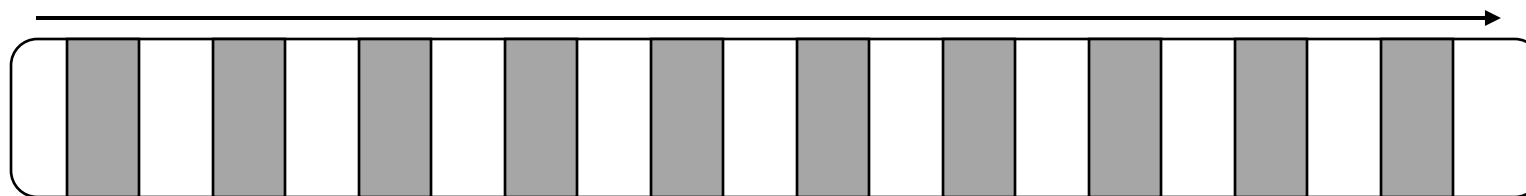


Sorted!



Case 2: I **only receive new updates**, which I never try to read

scan

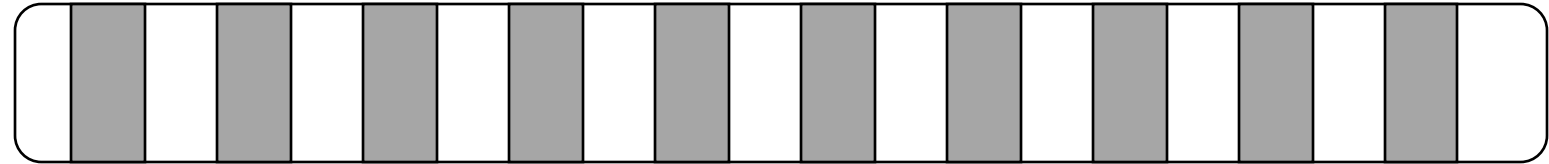


Append (log)

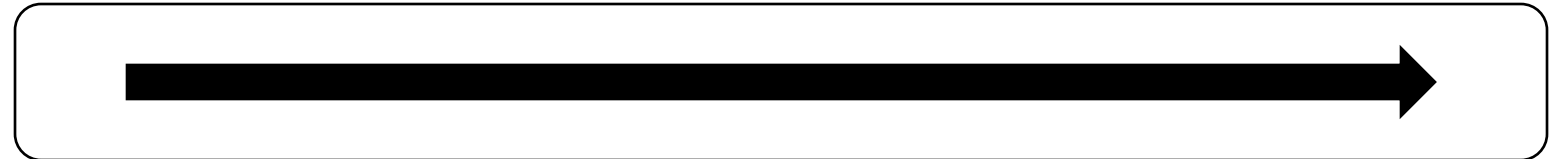
How to bridge the two?

Consider a workload with **bursts of new data**, followed by queries!

Append:

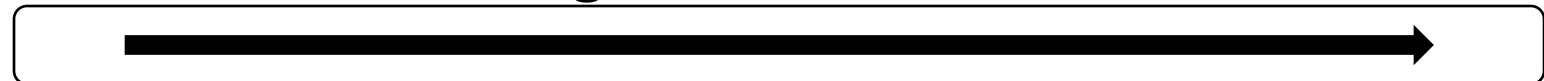


Once we accumulate “enough” data, we sort, and we write to the disk



What to do if we still receive incoming data?

Keep the sorted file



&

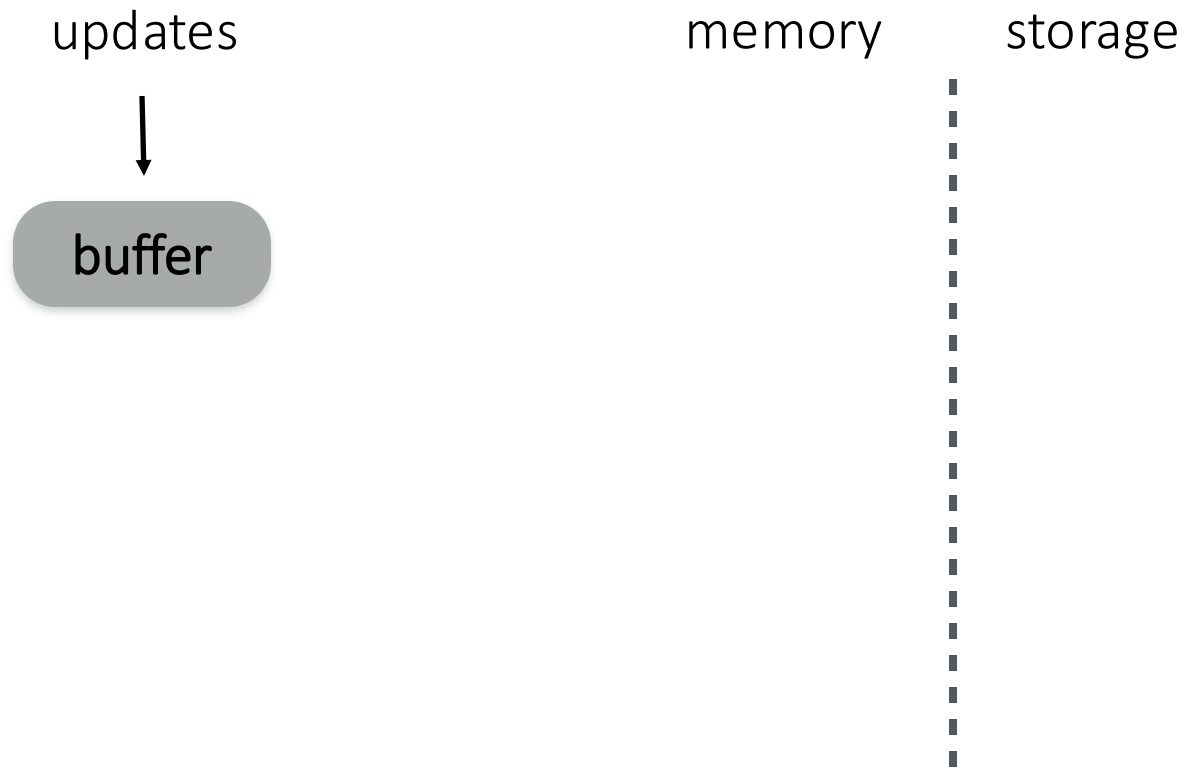
append to a new one

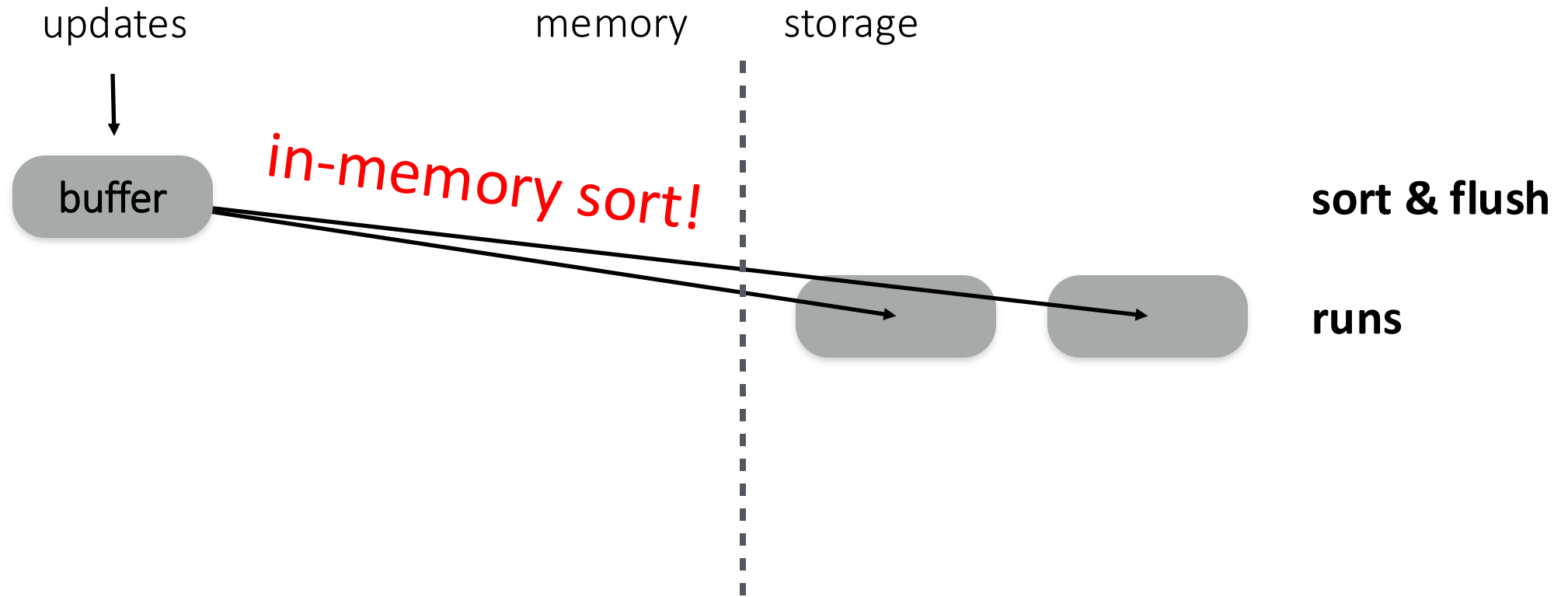


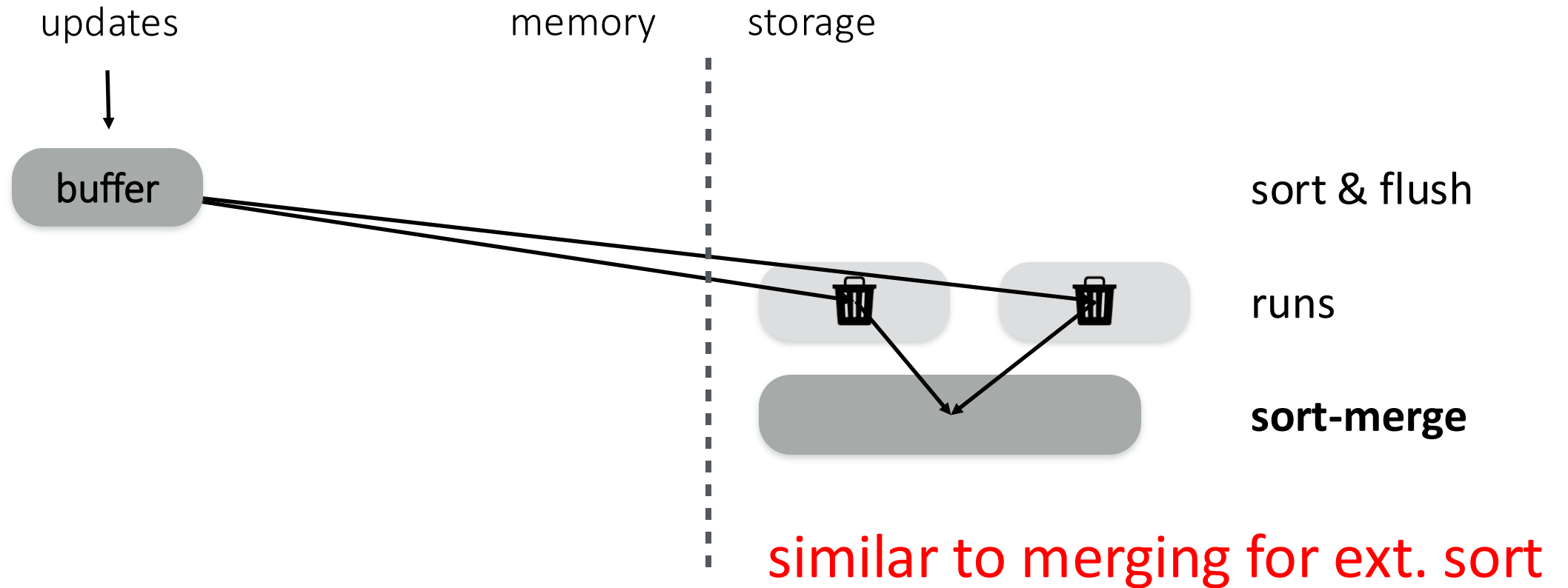


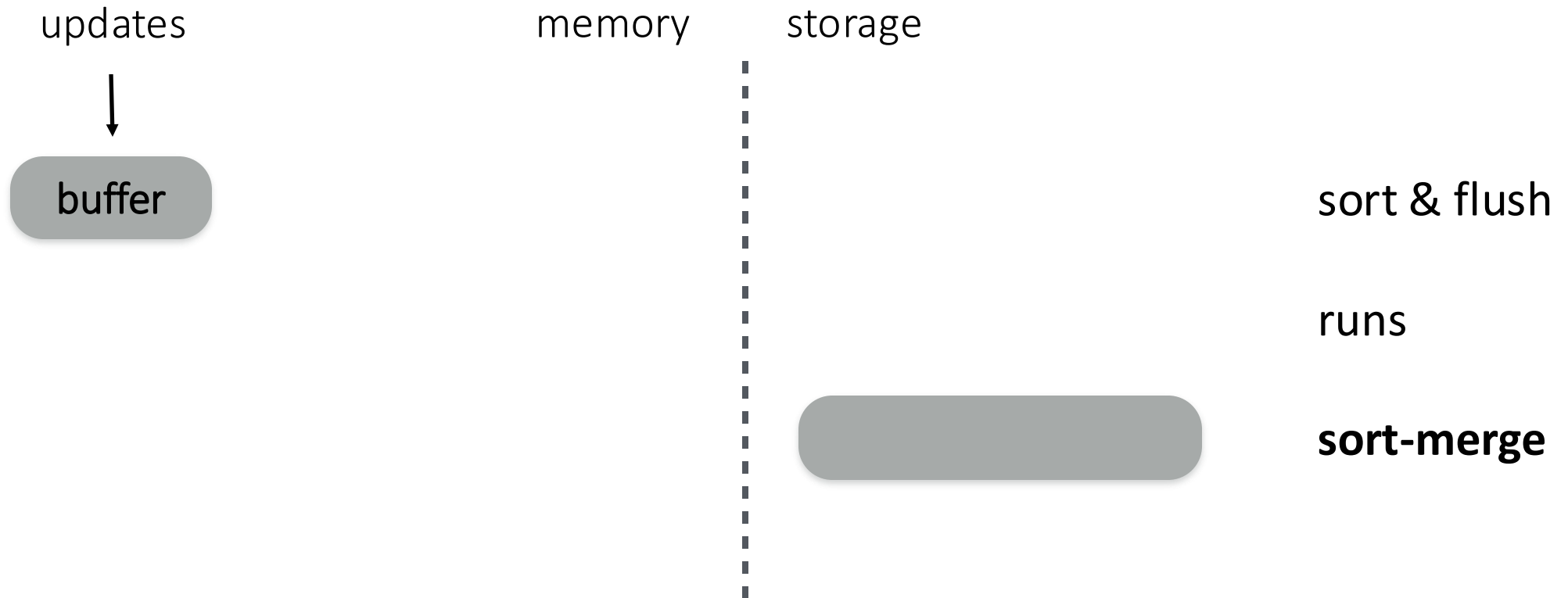
What to do with many sorted files?

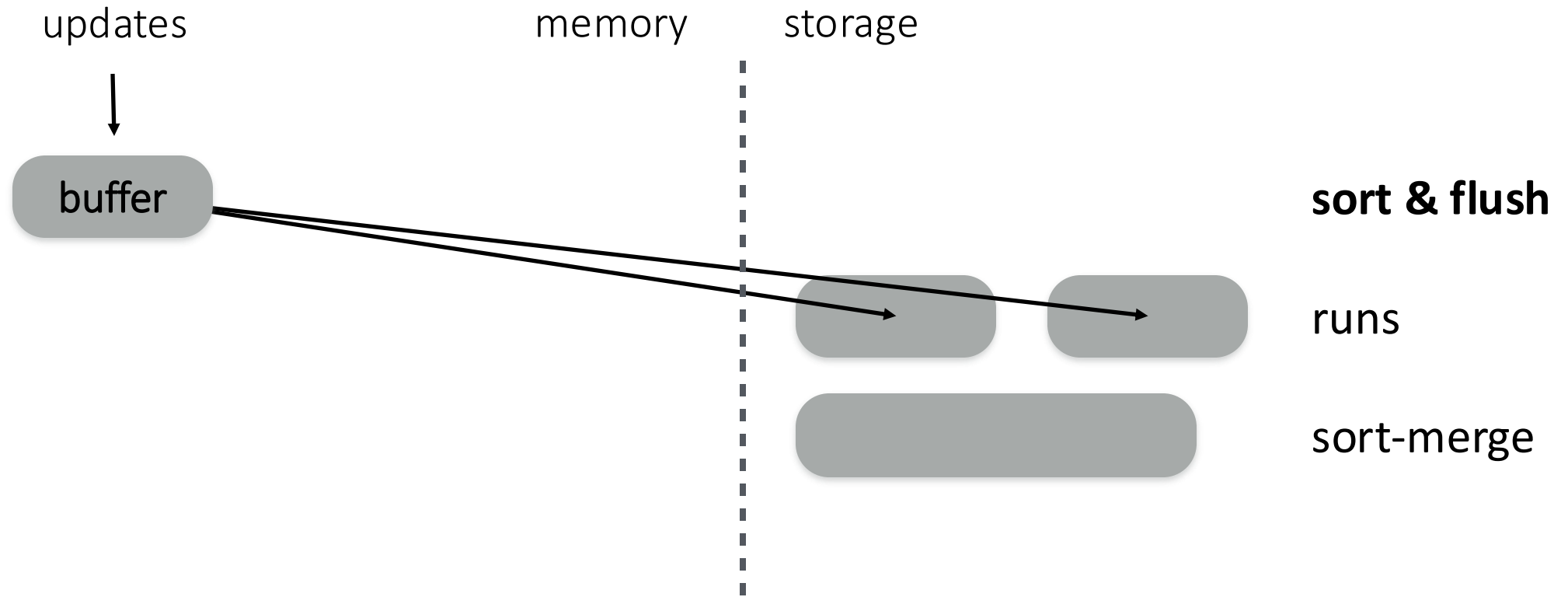
Merge them!

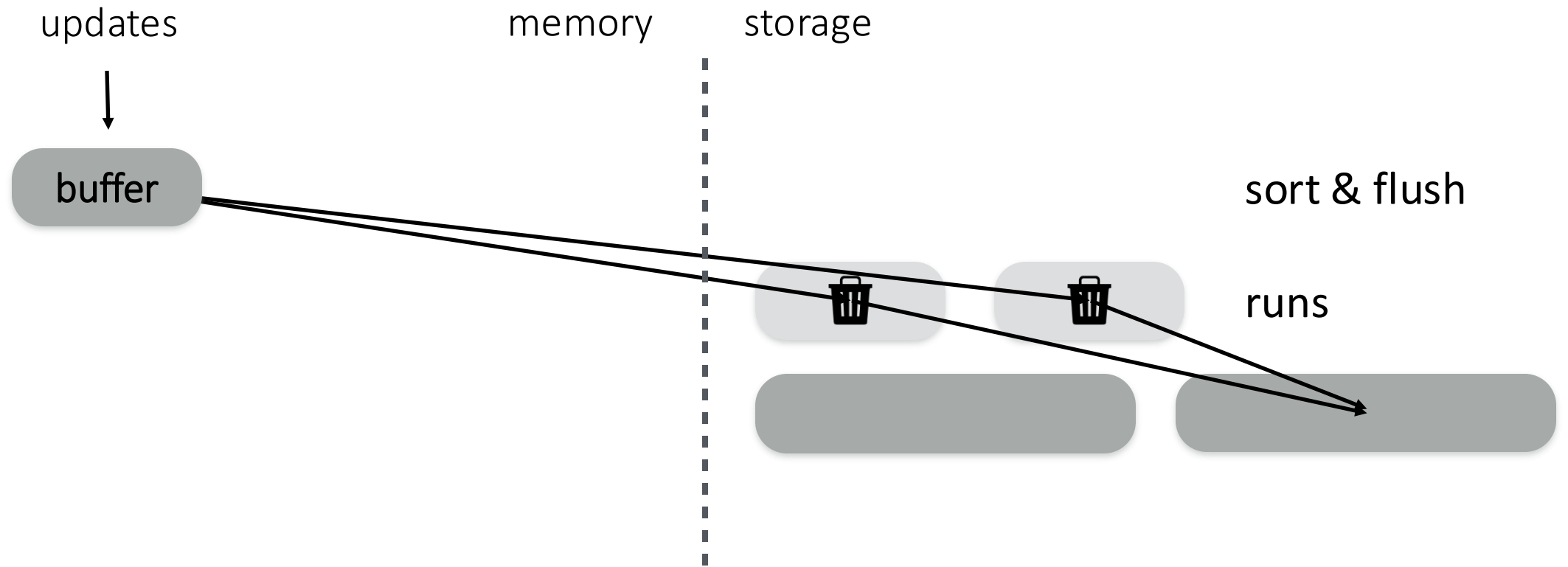


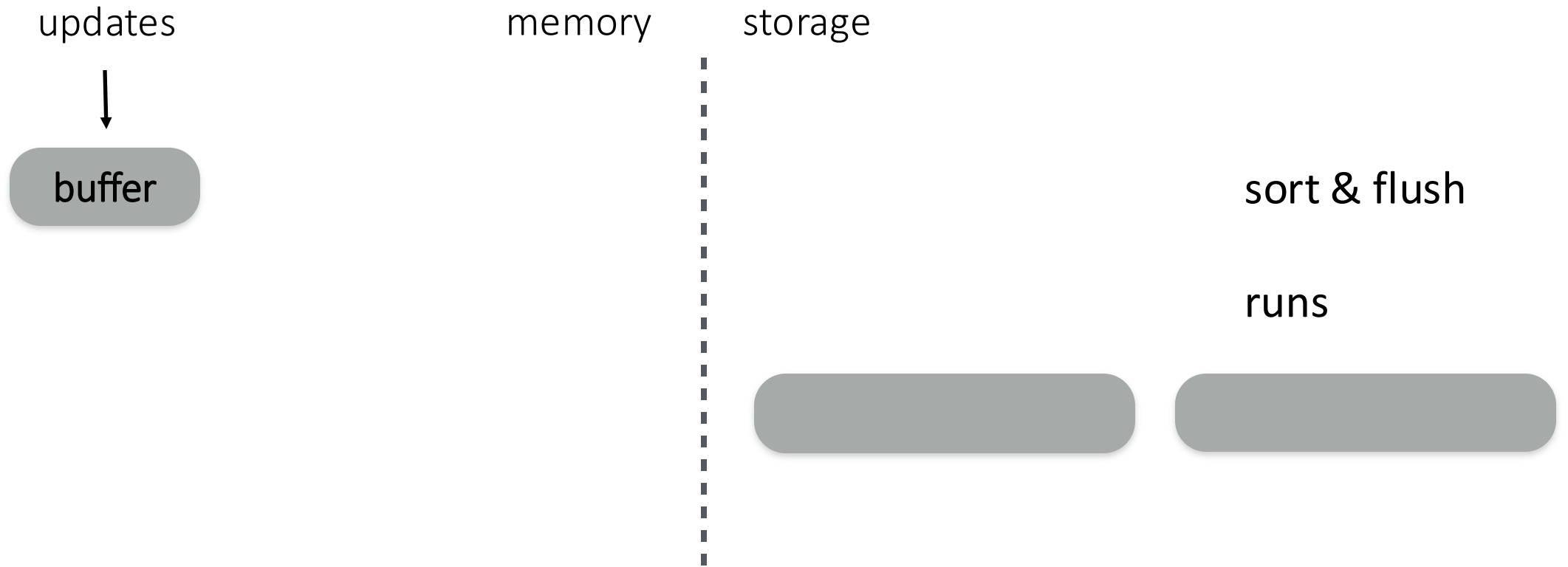


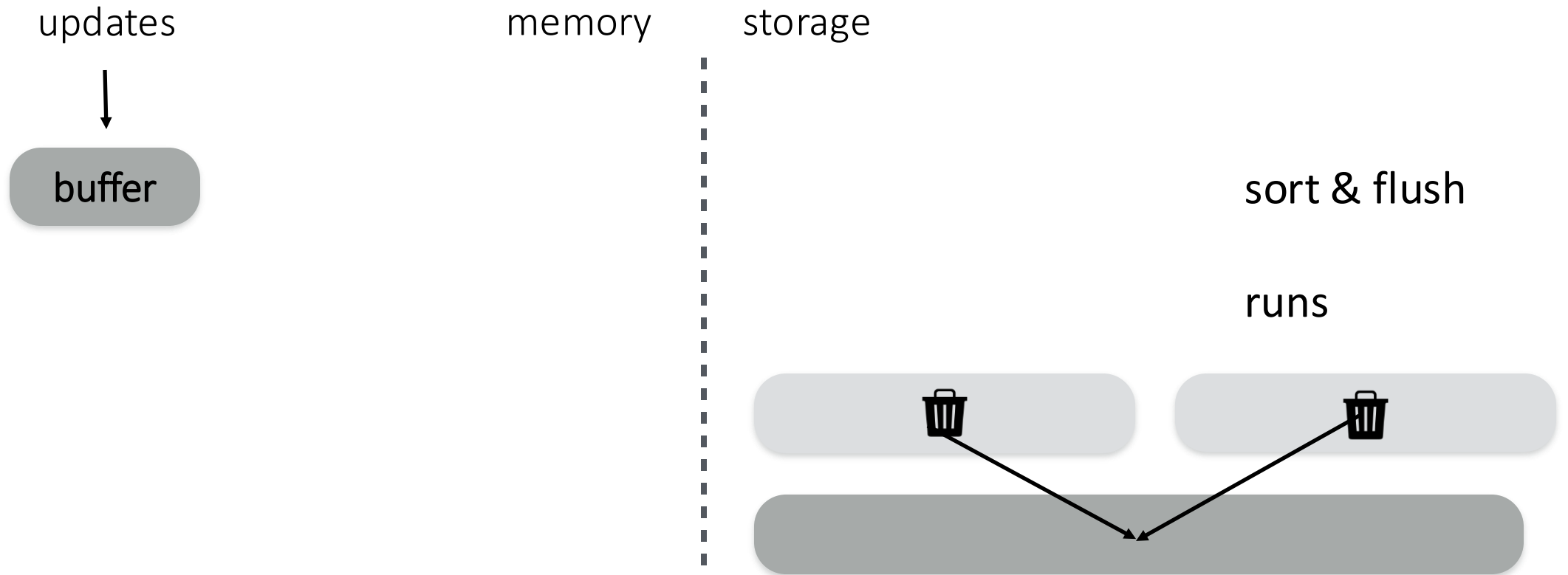


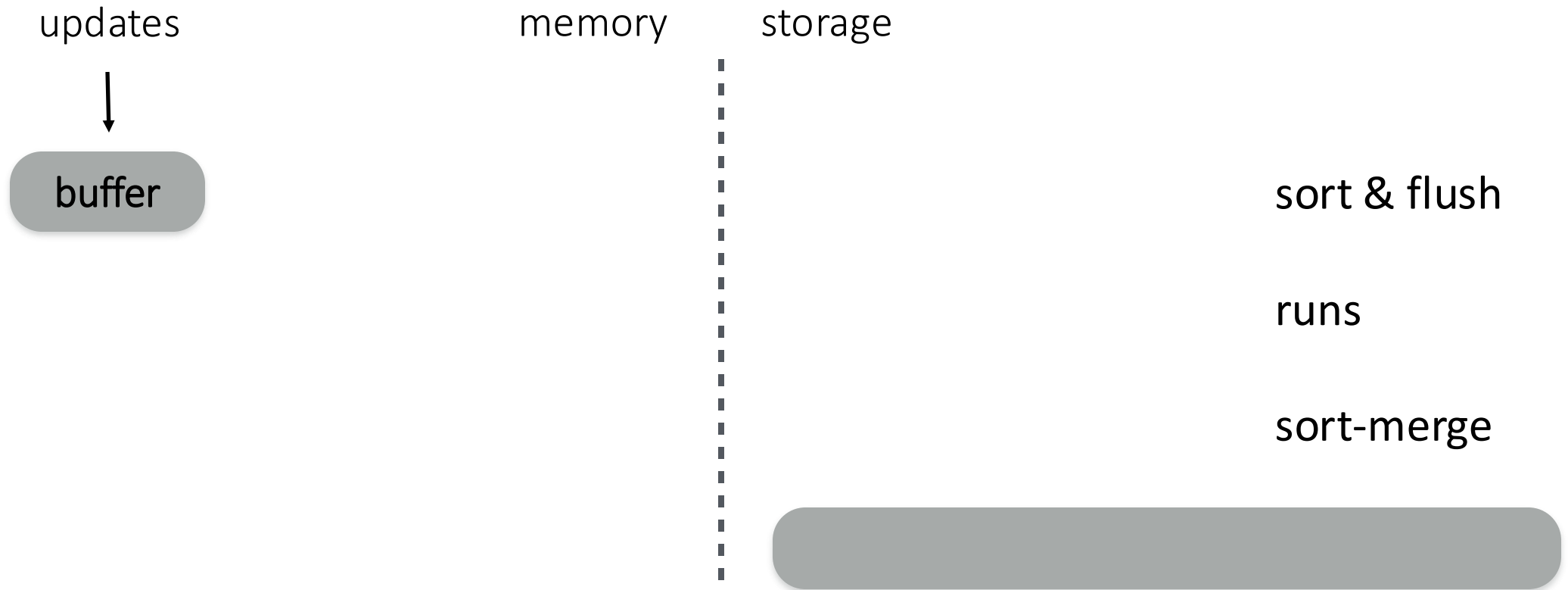


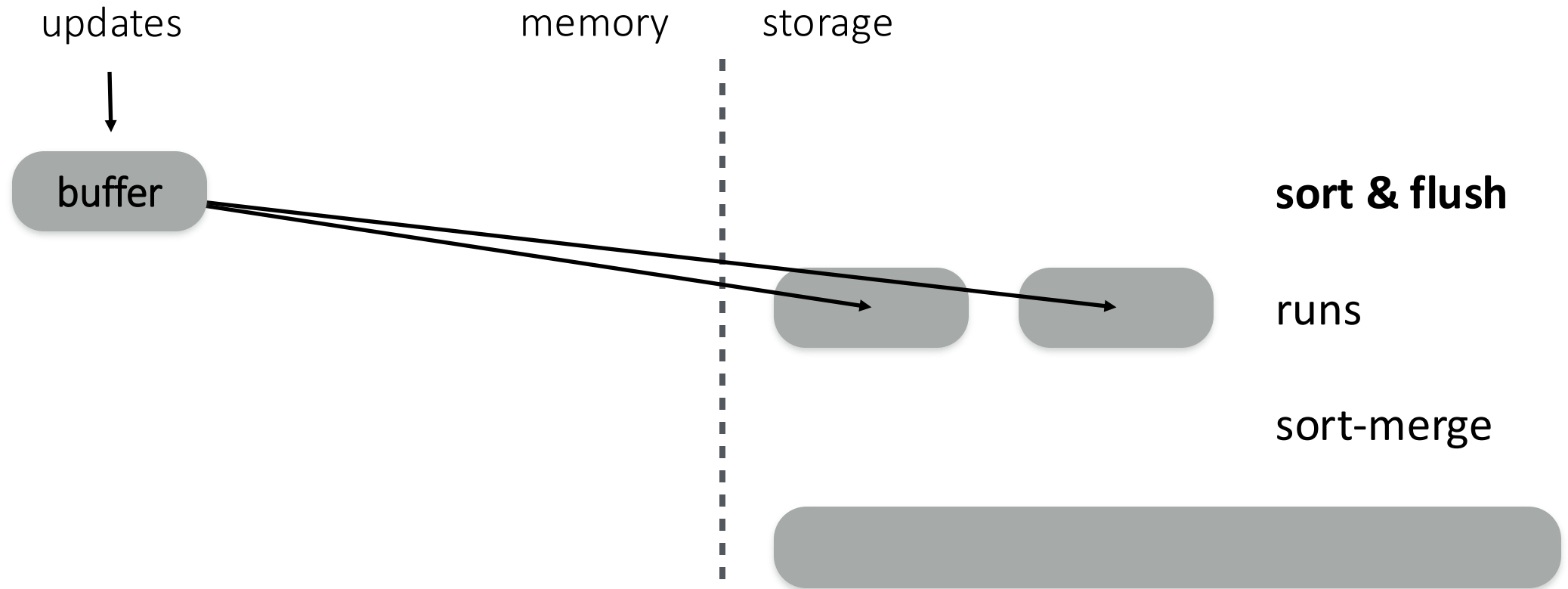


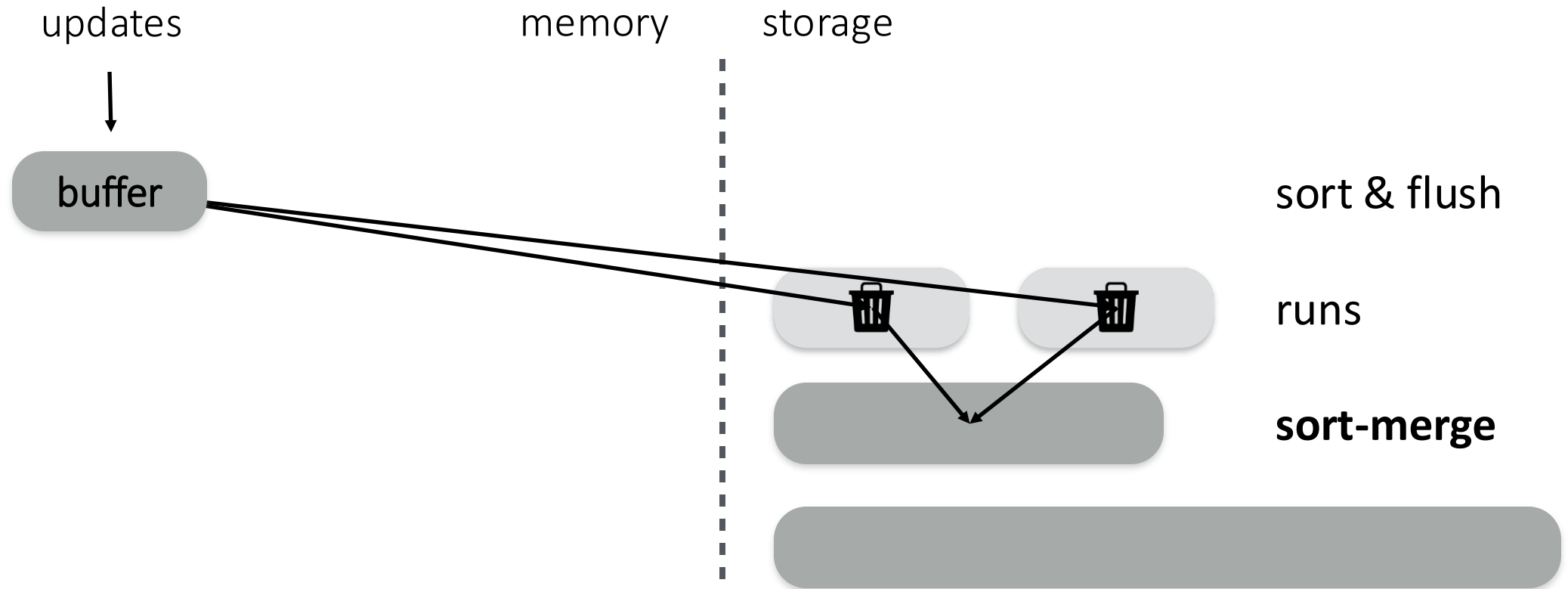


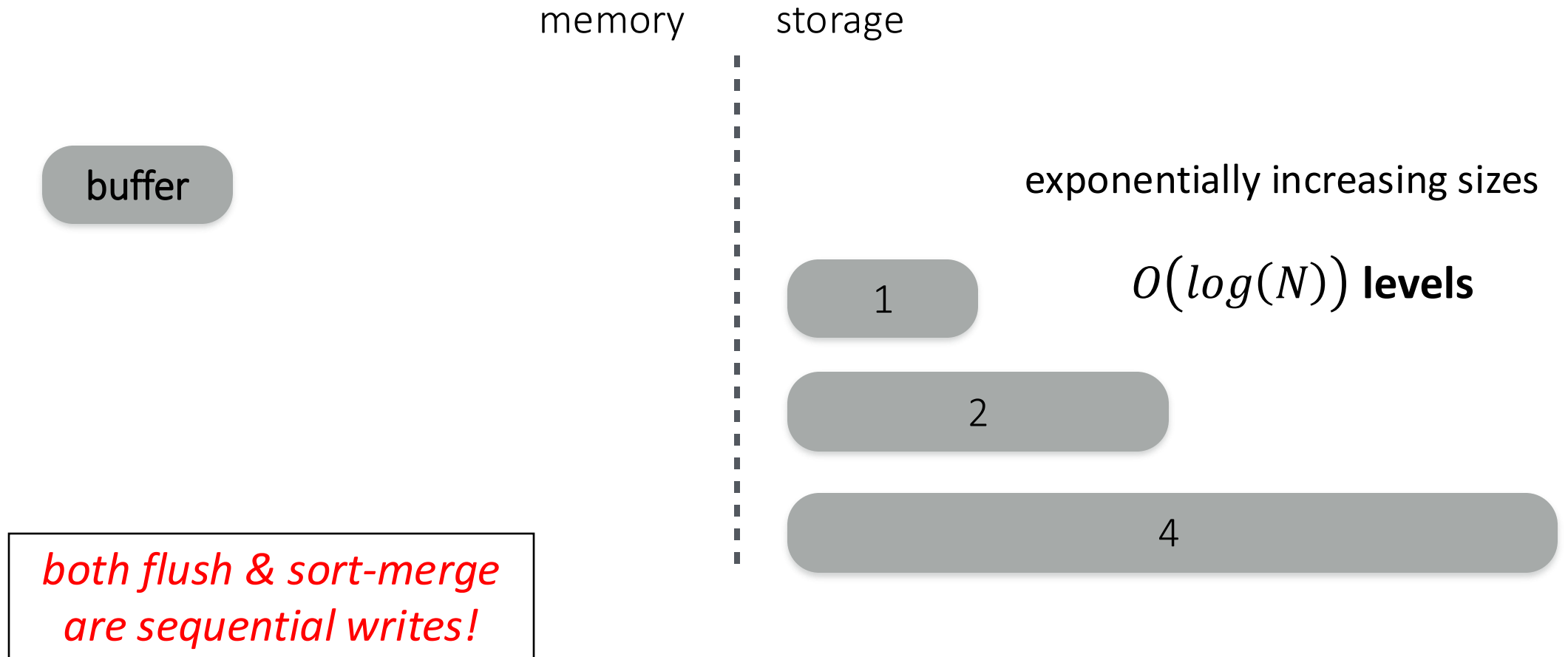












LSM-tree

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

1996

Patrick O'Neil¹, Edward Cheng²
Dieter Gawlick³, Elizabeth O'Neil¹
To be published: Acta Informatica

Patrick O'Neil
UMass Boston

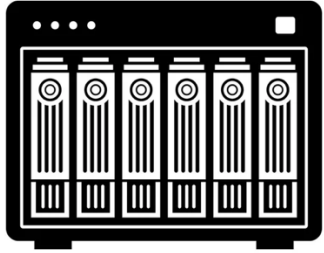


LSM-tree
O'Neil *et al.*

1996

✓ good sequential reads & writes

✓ good random writes



array
of discs

why?

RAID, striping ← ?



how many IOPS?

10KRPM

max seek time 1.5ms

100 disks

10KRPM: 10K rev in 60s

$60/10000=6\text{ms}$ per rev

avg. rot. delay: 3ms (6ms/2)

avg. seek time: 0.75ms (1.5ms/2)

1 I/O / 3.75ms: 267 IOPS

100 disks: 26,700 IOPS

✗ LSM not explicitly needed

LSM-tree
O'Neil *et al.*

so, arrays of disks were enough!



Bigtable

1980s

1996

2006

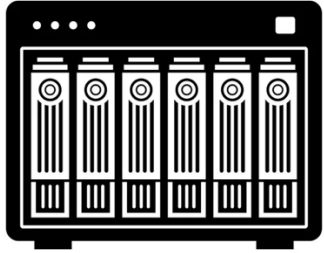
a decade

✓ good sequential reads & writes

✓ good random writes

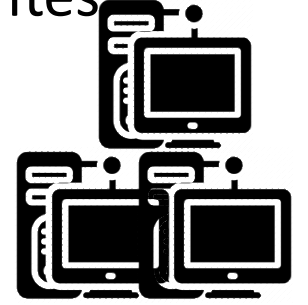
✗ worse sequential access

✗ bad random writes



array of discs

what happened in 2006?



commodity hardware

LSM-tree
O'Neil *et al.*

We set up a Bigtable cluster with N tablet servers to measure the performance and scalability of Bigtable as N is varied. The tablet servers were configured to use 1 GB of memory and to write to a GFS cell consisting of 1786 machines with two 400 GB IDE hard drives each.



Bigtable

1980s

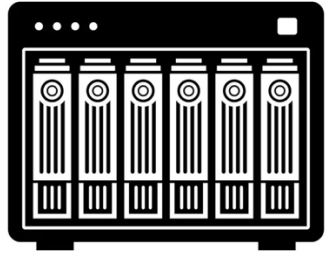
1996

2006

a decade

✓ good sequential reads & writes

✓ good random writes



array of discs



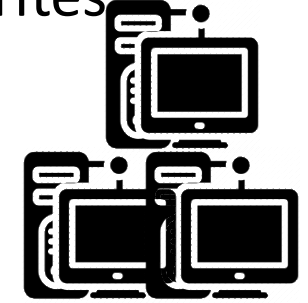
SSD wear-friendly

competitive rand. reads

fast ingestion (sequential)

✗ worse sequential access

✗ bad random writes



commodity hardware



LSM-tree
O'Neil *et al.*



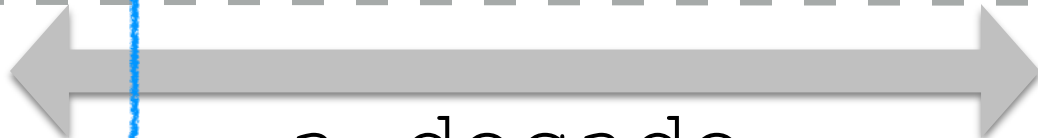
Bigtable

1980s

1996

2006

a decade



LSM-tree
O'Neil *et al.*

1996



Bigtable

2006

APACHE
HBASE 

2007

LSM-tree
O'Neil *et al.*

1996



Bigtable

2006

APACHE
HBASE 

2007



cassandra

2010

LSM-tree
O'Neil *et al.*

1996



2006



2007



2010



levelDB

2011

LSM-tree
O'Neil *et al.*

1996



Bigtable

2006

APACHE
HBASE 

2007



cassandra

2010



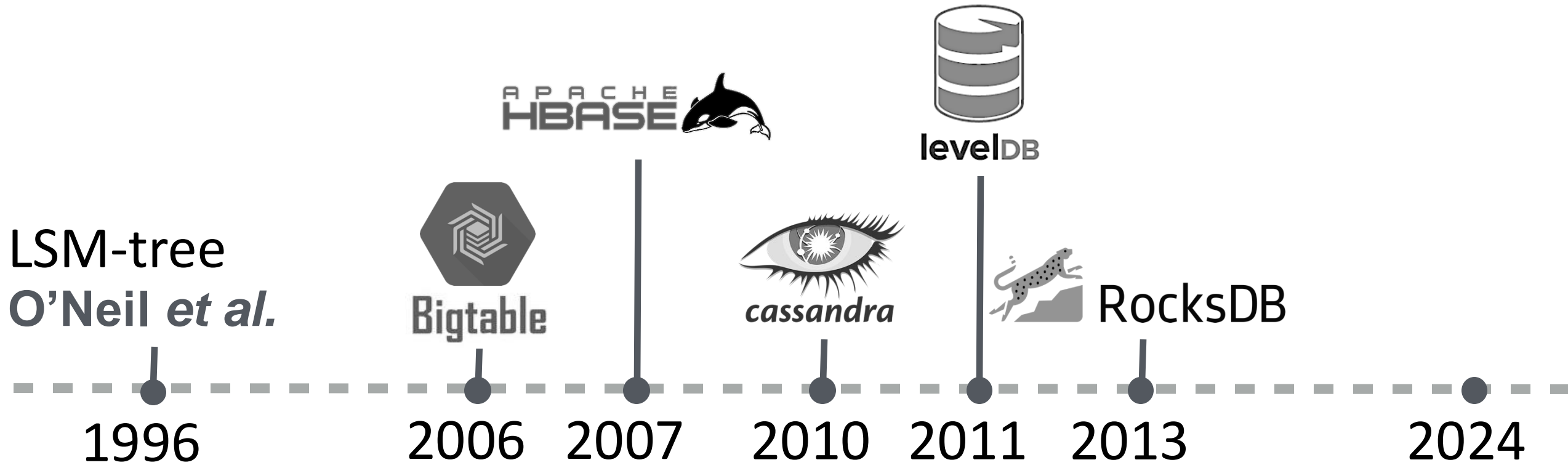
levelDB

2011



RocksDB

2013



LSM-tree

NoSQL

This block contains logos for various NoSQL databases. The logos are arranged in two rows. The top row includes RocksDB (leopard), WT (stylized letters), levelDB (cylinder), SCYLLA (skull), and riak (network diagram). The bottom row includes cassandra (eye), tarantool (Venn diagram), Bigtable (hexagon), APACHE HBASE (orca), DynamoDB (cylinder), speedb (dotted shape), and ACCUMULO (grid pattern).

This block contains the SQLite logo (feather) and a square icon with a whale, representing a relational database.

relational

This block contains the influxdb logo (cube) and the QuasarDB logo (curved grid).

time-series

2024

LSM-tree

NoSQL

This block contains logos for various NoSQL databases. The logos are arranged in two rows. The top row includes RocksDB (cheetah), WT (stylized letters), levelDB (green cylinder), SCYLLA (blue monster), and riak (grey text with orange dots). The bottom row includes cassandra (eye), tarantool (red circles), Bigtable (blue hexagon), APACHE HBASE (red text with orca), DynamoDB (blue cylinder), and speedb (blue dots). The ACCUMULO logo is also present at the bottom center.

This block contains two logos: SQLite (blue square with feather) and a dark blue square with a white whale, representing a relational database.

relational

This block contains two logos: influxdb (blue cube) and QuasarDB (blue grid pattern).

time-series

2024

How does LSM-tree compare with prior approaches?

Compare and contrast data structures.

What to use when?

Data Structure	Lookup cost	Insertion cost
Sorted array		
Log		
B-tree		
Basic LSM-tree		
Leveled LSM-tree		
Tiered LSM-tree		

Results Catalogue

Compare and contrast data structures.

What to use when?

Data Structure	Lookup cost	Insertion cost
Sorted array		
Log		
B-tree		
Basic LSM-tree		
Leveled LSM-tree		
Tiered LSM-tree		

Sorted Array

Measure Performance in I/Os

n entries

B entries fit into a disk block

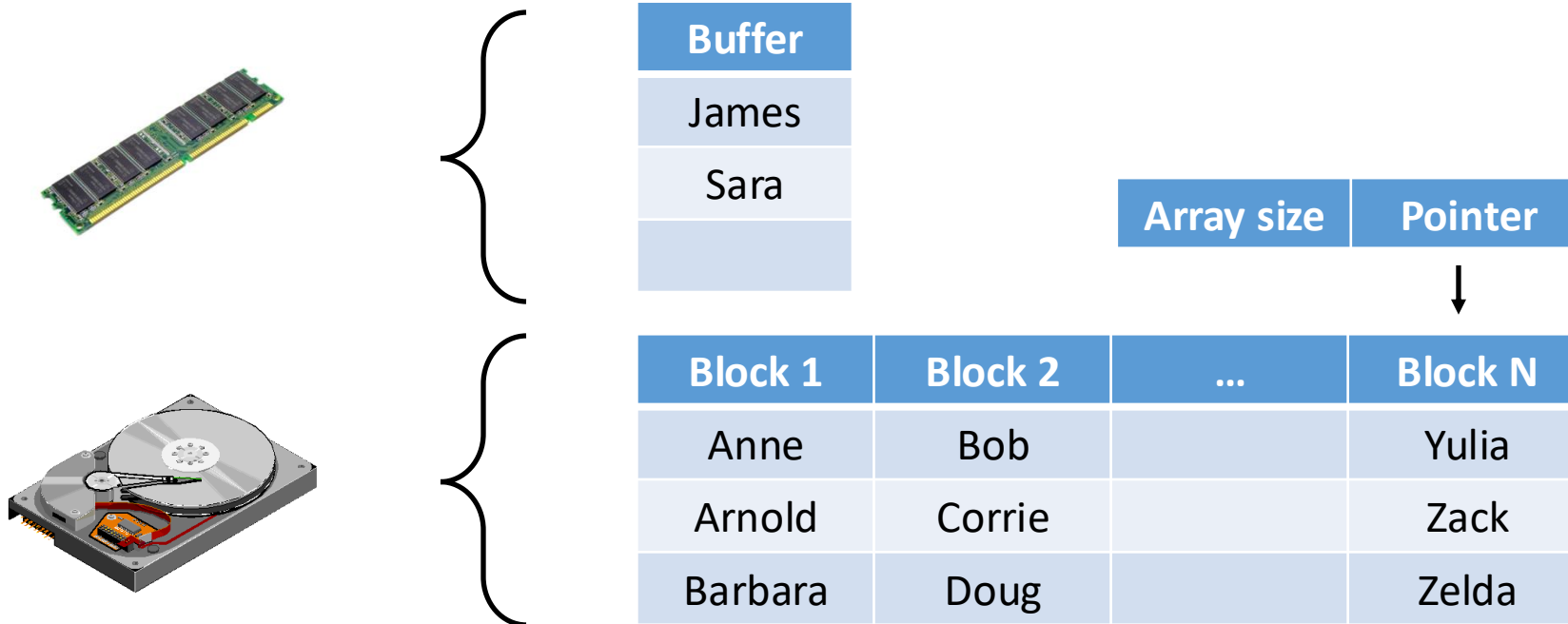
Array spans $N = \frac{n}{B}$ disk blocks

Lookup method & cost?

Binary search: $O(\log_2(N))$ I/Os

Insertion cost?

Push entries: $O(N/2)$ I/Os



Results Catalogue

	Lookup cost	Insertion cost
Sorted array	$O(\log_2(N))$	$O(N/2)$
Log		
B-tree		
Basic LSM-tree		
Leveled LSM-tree		
Tiered LSM-tree		

Results Catalogue

	Lookup cost	Insertion cost
Sorted array	$O(\log_2(N))$	$O(N/2)$
Log		
B-tree		
Basic LSM-tree		
Leveled LSM-tree		
Tiered LSM-tree		

Log (append-only array)

n entries

B entries fit into a disk block

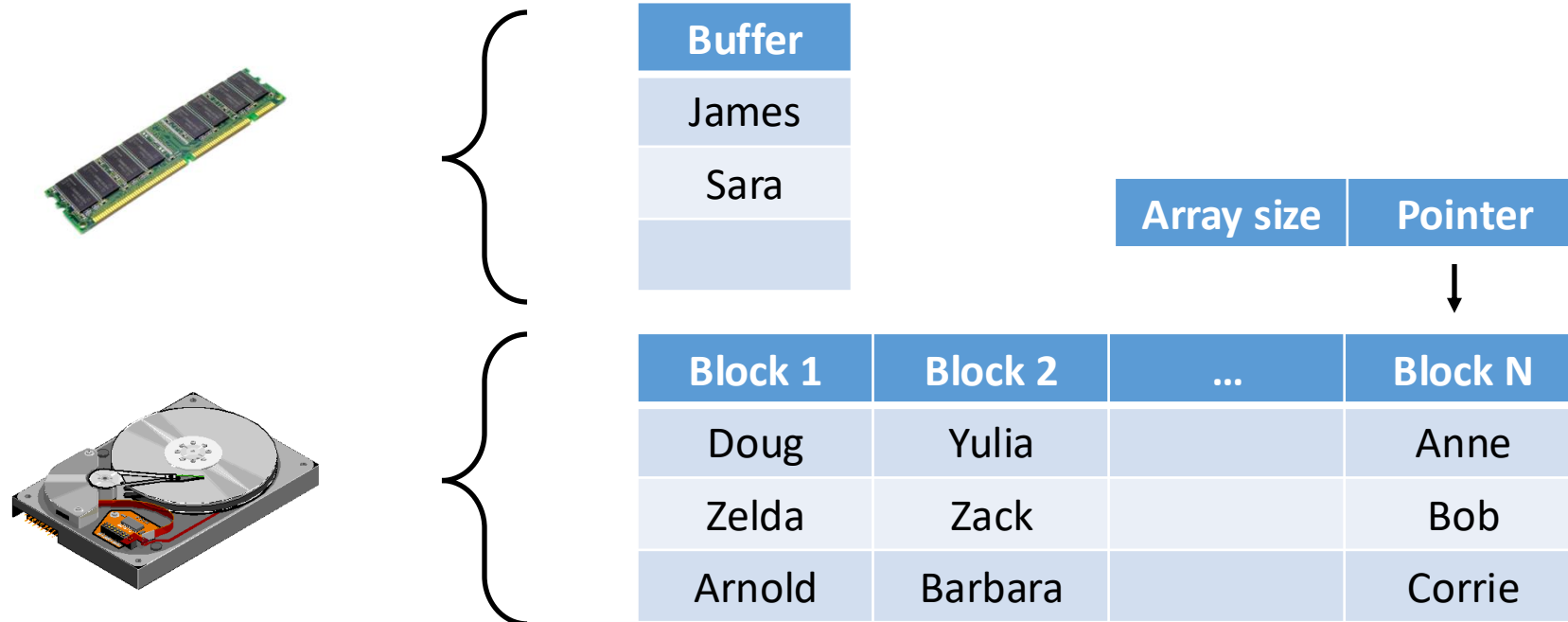
Array spans $N = \frac{n}{B}$ disk blocks

Lookup method & cost?

Scan: $O(N)$

Insertion cost?

Append: $O\left(\frac{1}{B}\right)$



Results Catalogue

	Lookup cost	Insertion cost
Sorted array	$O(\log_2(N))$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree		
Basic LSM-tree		
Leveled LSM-tree		
Tiered LSM-tree		

Results Catalogue

	Lookup cost	Insertion cost
Sorted array	$O(\log_2(N))$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree		
Basic LSM-tree		
Leveled LSM-tree		
Tiered LSM-tree		

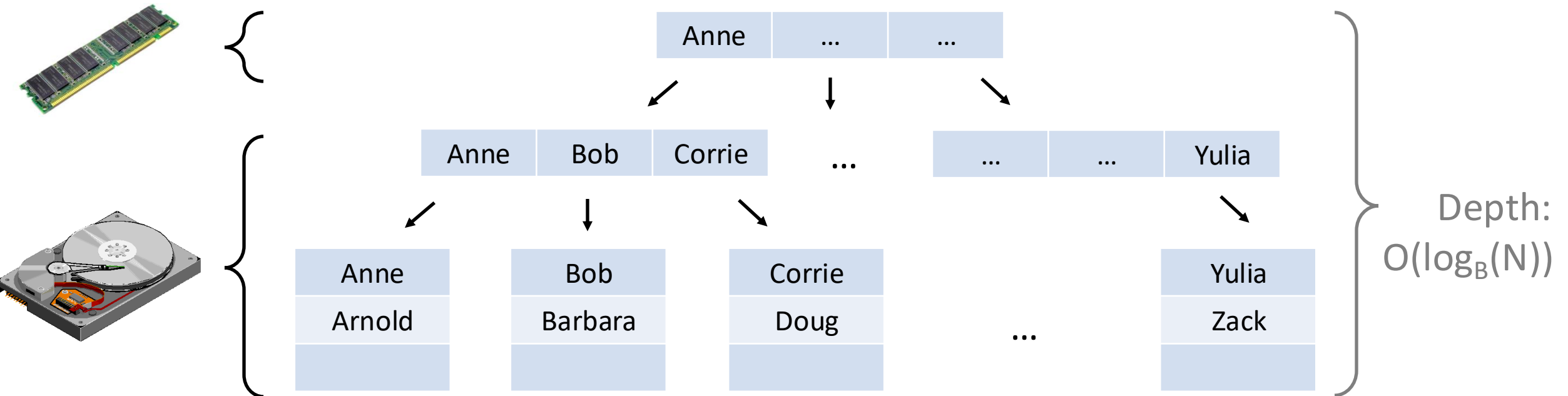
B-tree

Lookup method & cost?

Tree search: $O(\log_B(N))$

Insertion method & cost?

Tree search & append: $O(\log_B(N))$



Results Catalogue

	Lookup cost	Insertion cost
Sorted array	$O(\log_2(N))$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree		
Leveled LSM-tree		
Tiered LSM-tree		

B-trees



“It could be said that the world’s information is at our fingertips because of B-trees”

Goetz Graefe

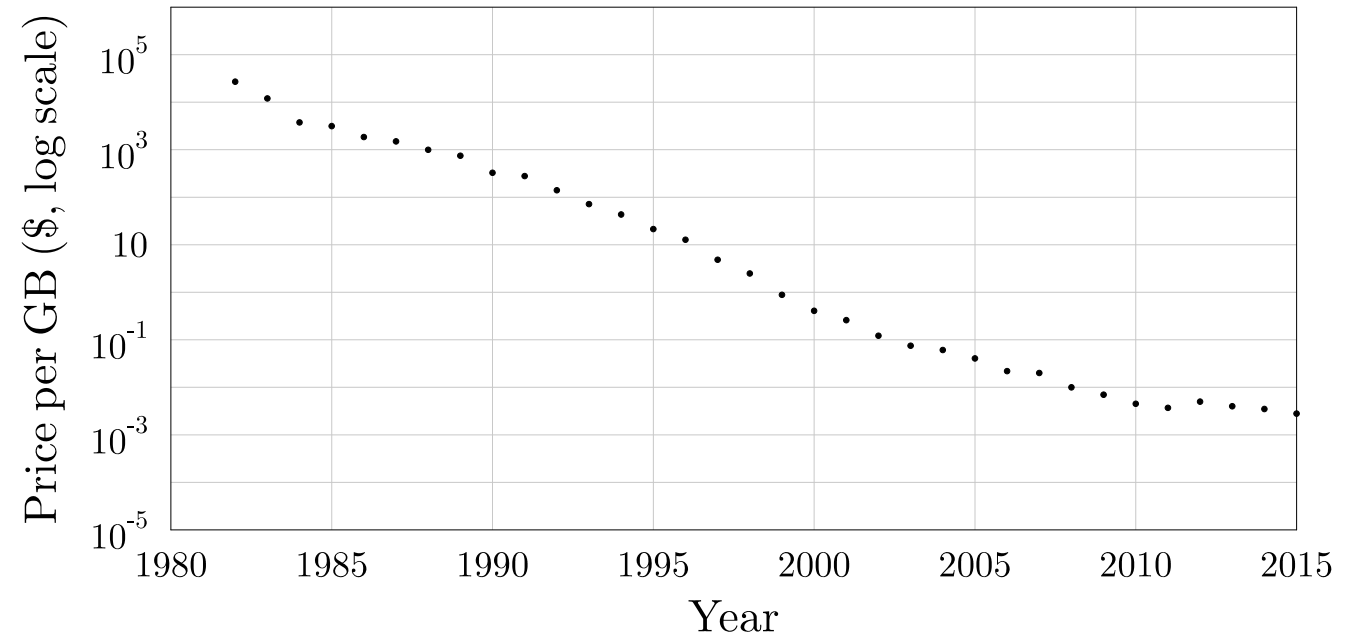
Microsoft, HP Fellow, now Google
ACM Software System Award

B-trees are no longer sufficient

Cheaper storage

Workloads more **insert-intensive**

We need **better insert-performance**



Results Catalogue

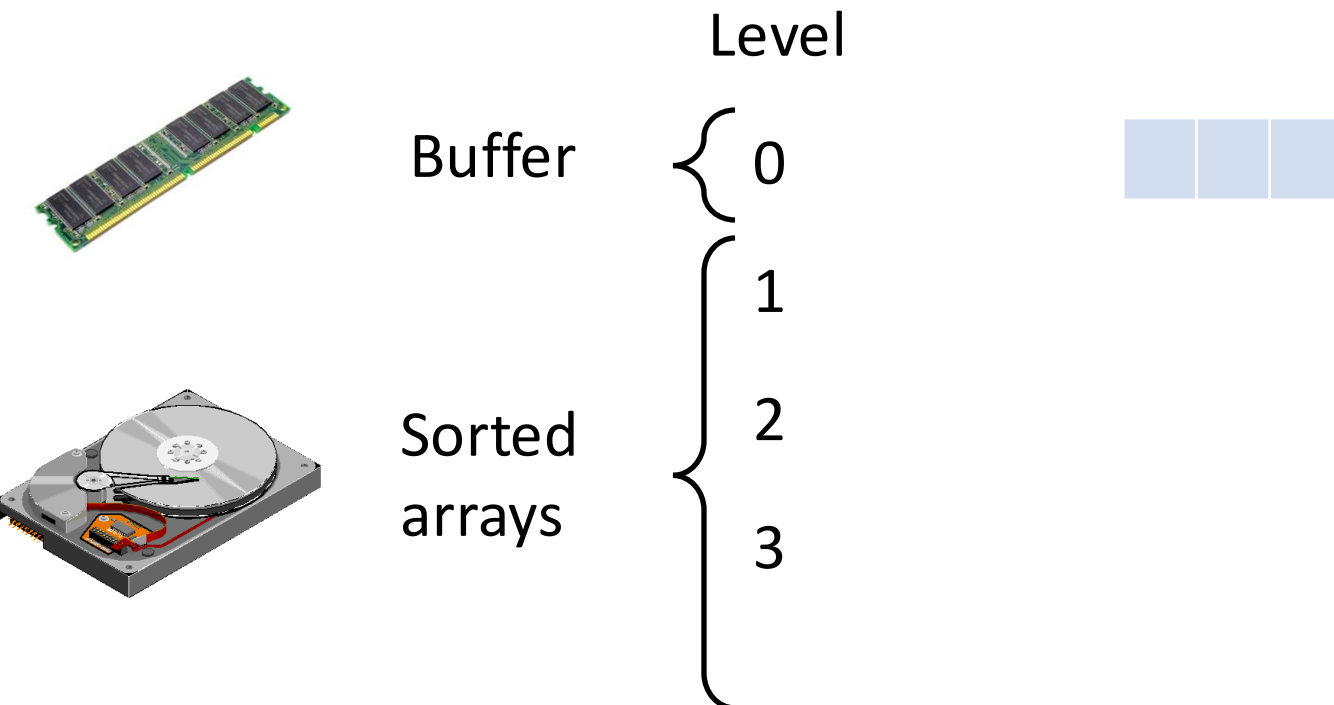
Goal: combine

(low) **constant insertion** cost
logarithmic lookup cost

	Lookup cost	Insertion cost
Sorted array	$O(\log_2(N))$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree		
Leveled LSM-tree		
Tiered LSM-tree		

Basic LSM-trees

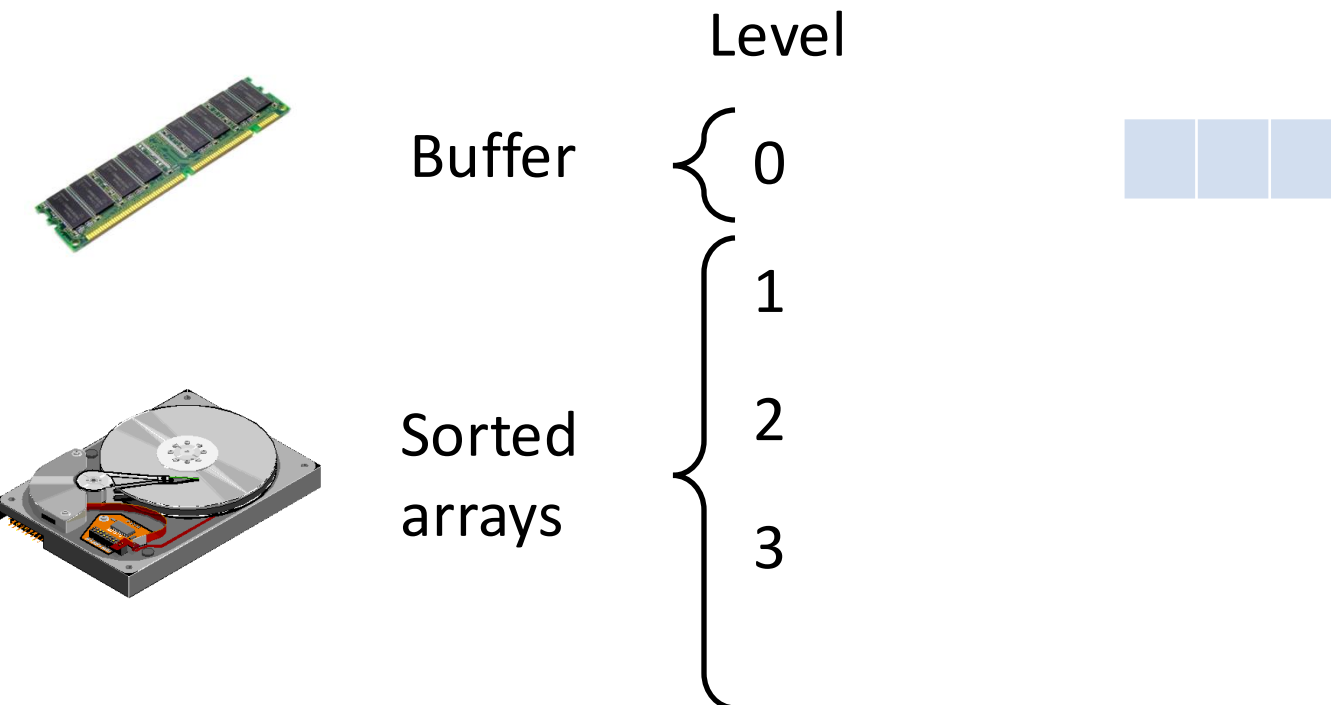
Basic LSM-tree



Basic LSM-tree

Design principle #1:

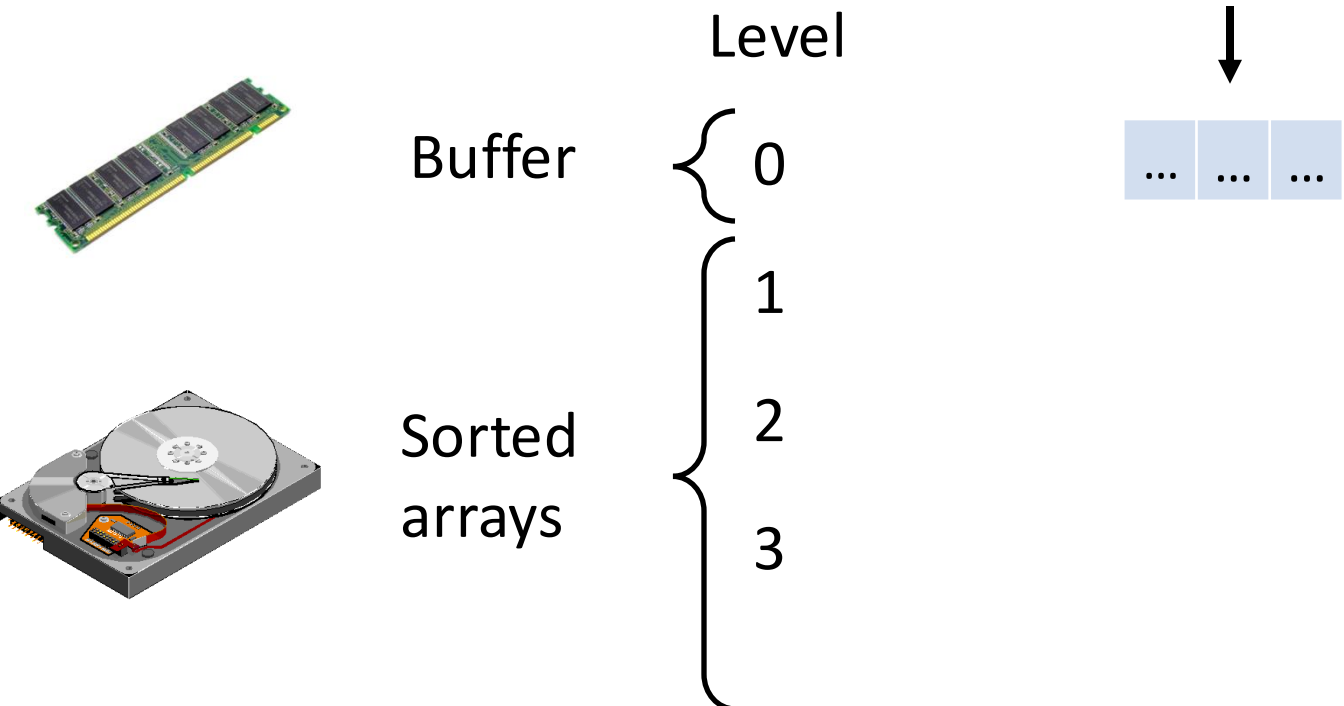
optimize for insertions by buffering



Basic LSM-tree

Design principle #1:

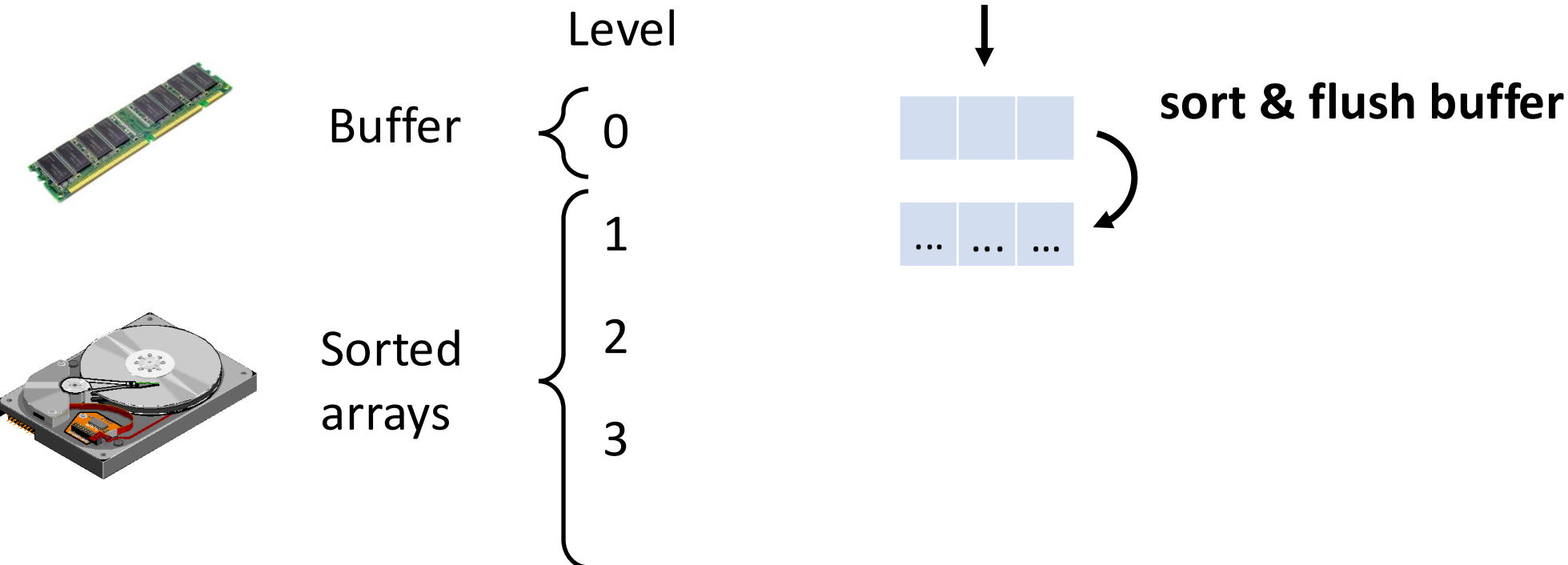
optimize for insertions by buffering



Basic LSM-tree

Design principle #1:

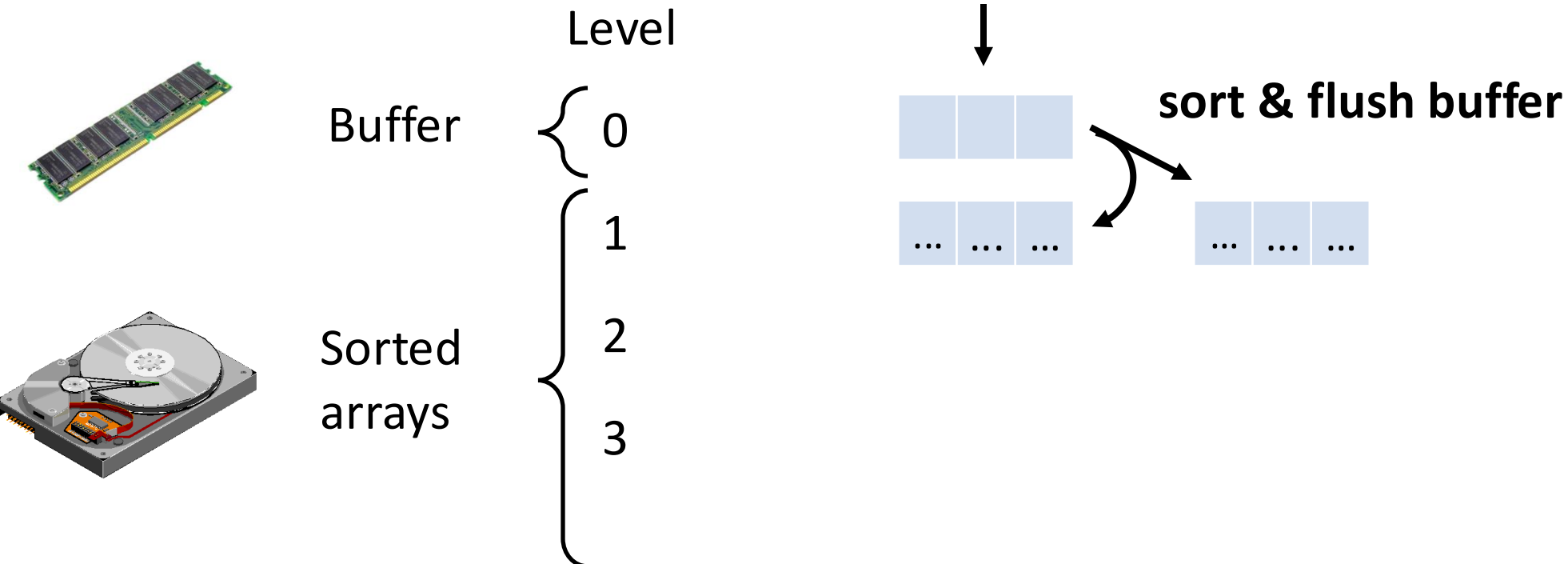
optimize for insertions by buffering



Basic LSM-tree

Design principle #1:

optimize for insertions by buffering



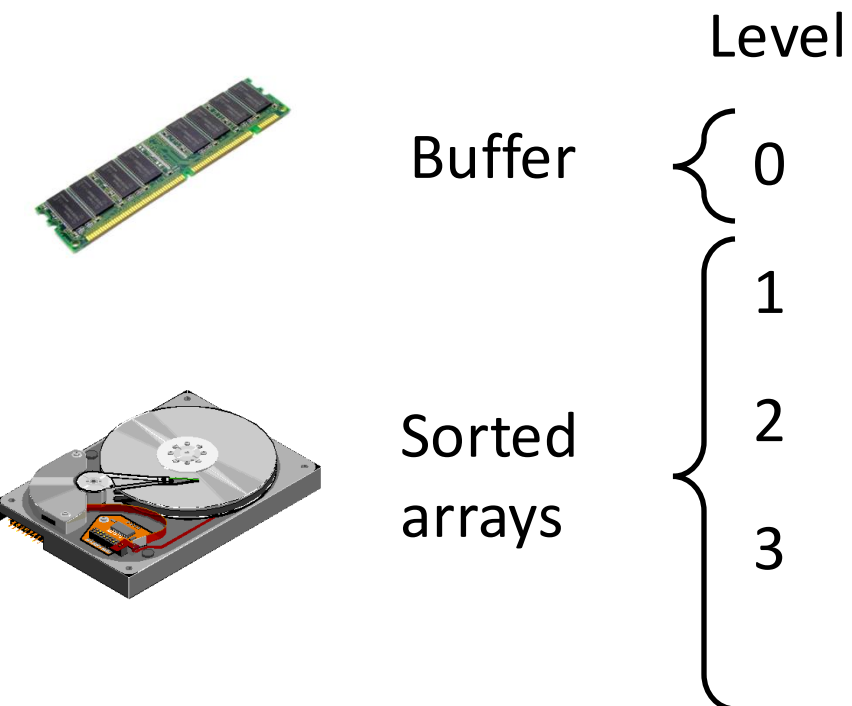
Basic LSM-tree

Design principle #1:

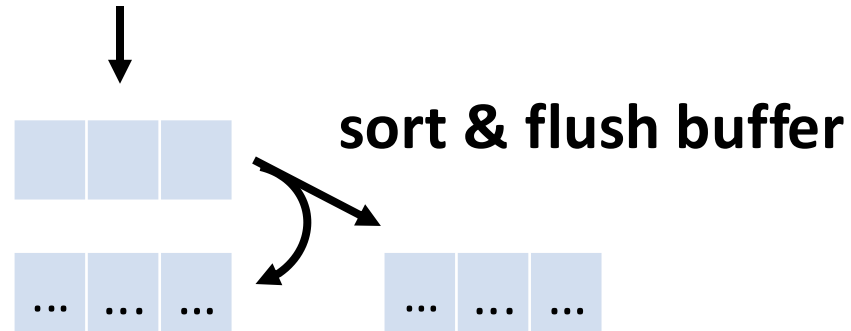
optimize for insertions by buffering

Design principle #2:

optimize for lookups by sort-merging arrays



Inserts



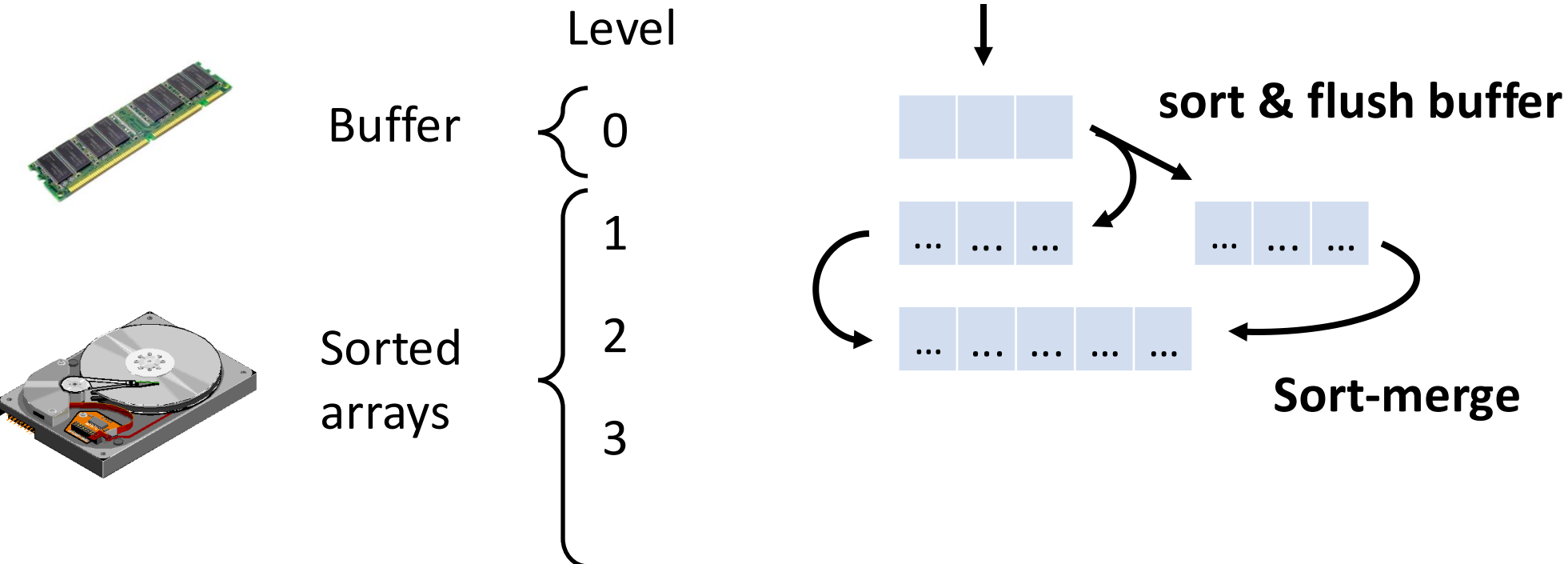
Basic LSM-tree

Design principle #1:

optimize for insertions by buffering

Design principle #2:

optimize for lookups by sort-merging arrays



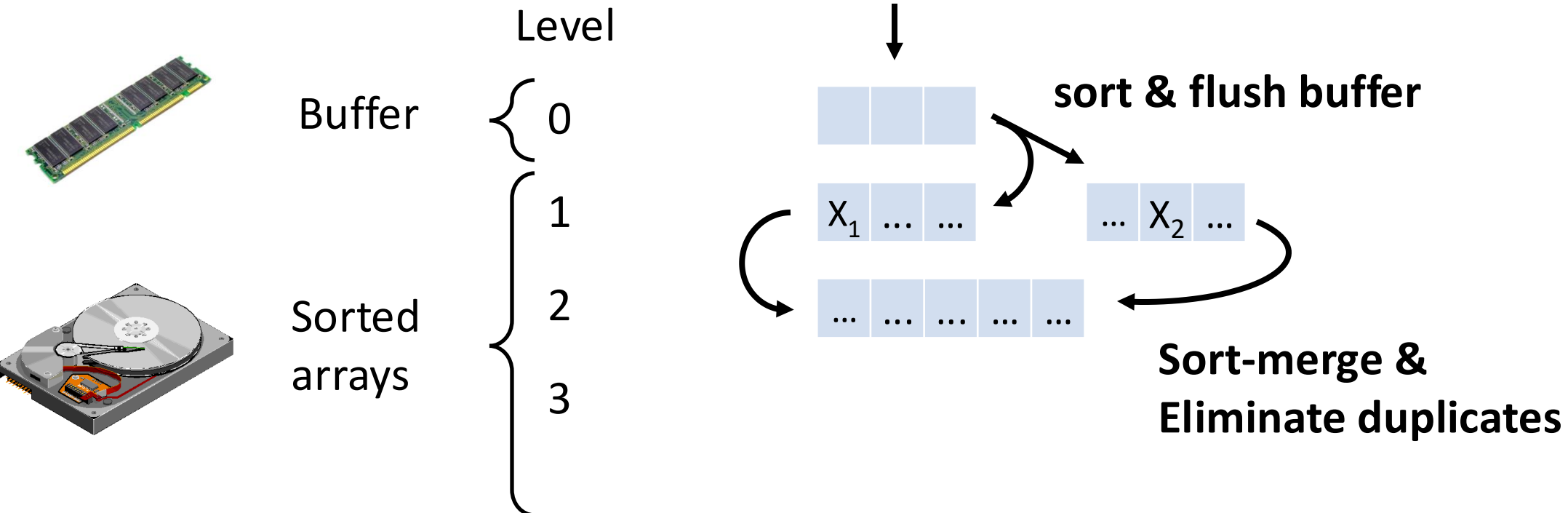
Basic LSM-tree

Design principle #1:

optimize for insertions by buffering

Design principle #2:

optimize for lookups by sort-merging arrays



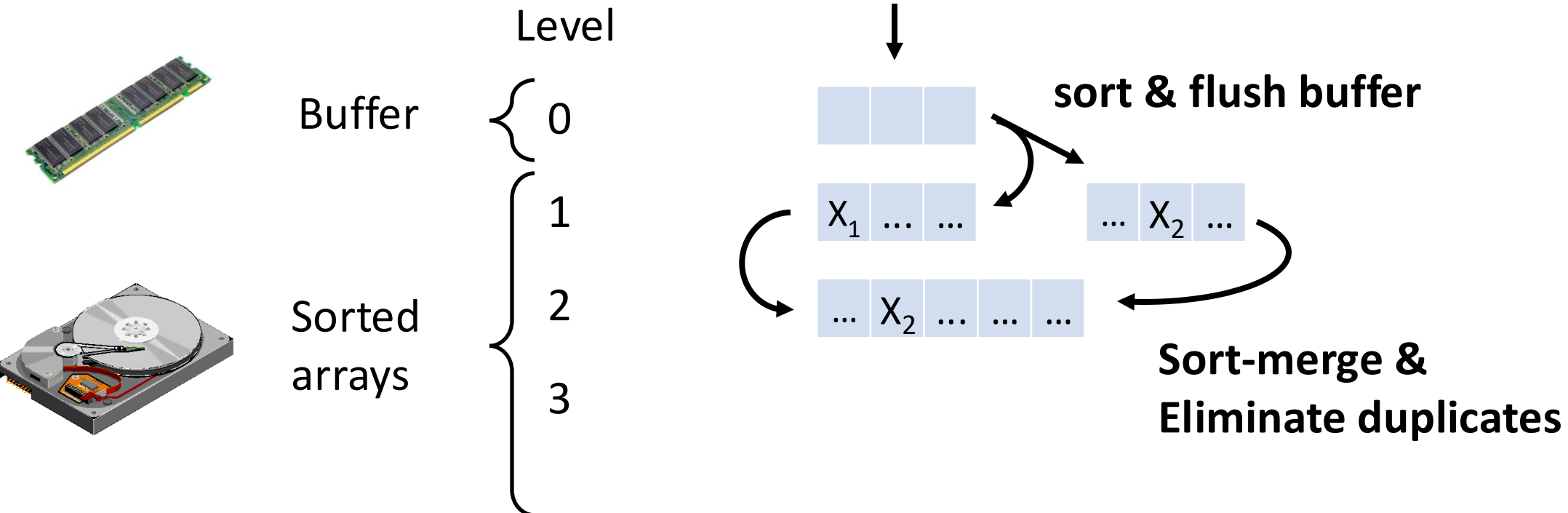
Basic LSM-tree

Design principle #1:

optimize for insertions by buffering

Design principle #2:

optimize for lookups by sort-merging arrays



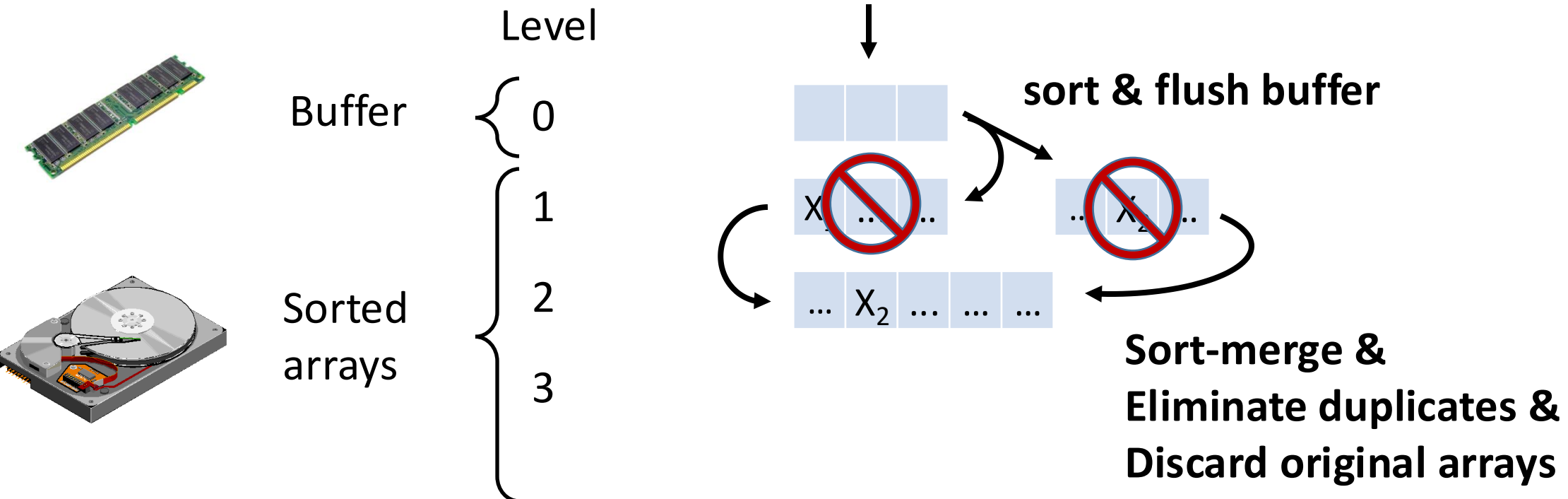
Basic LSM-tree

Design principle #1:

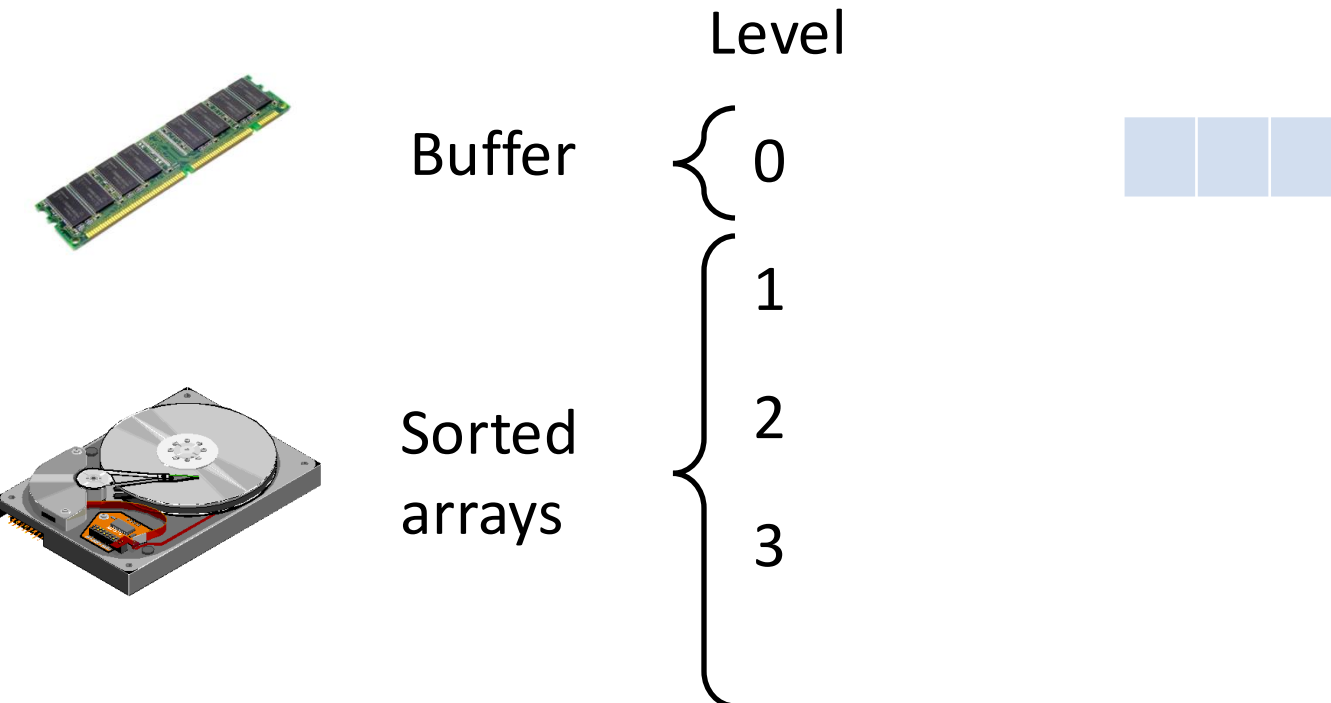
optimize for insertions by buffering

Design principle #2:

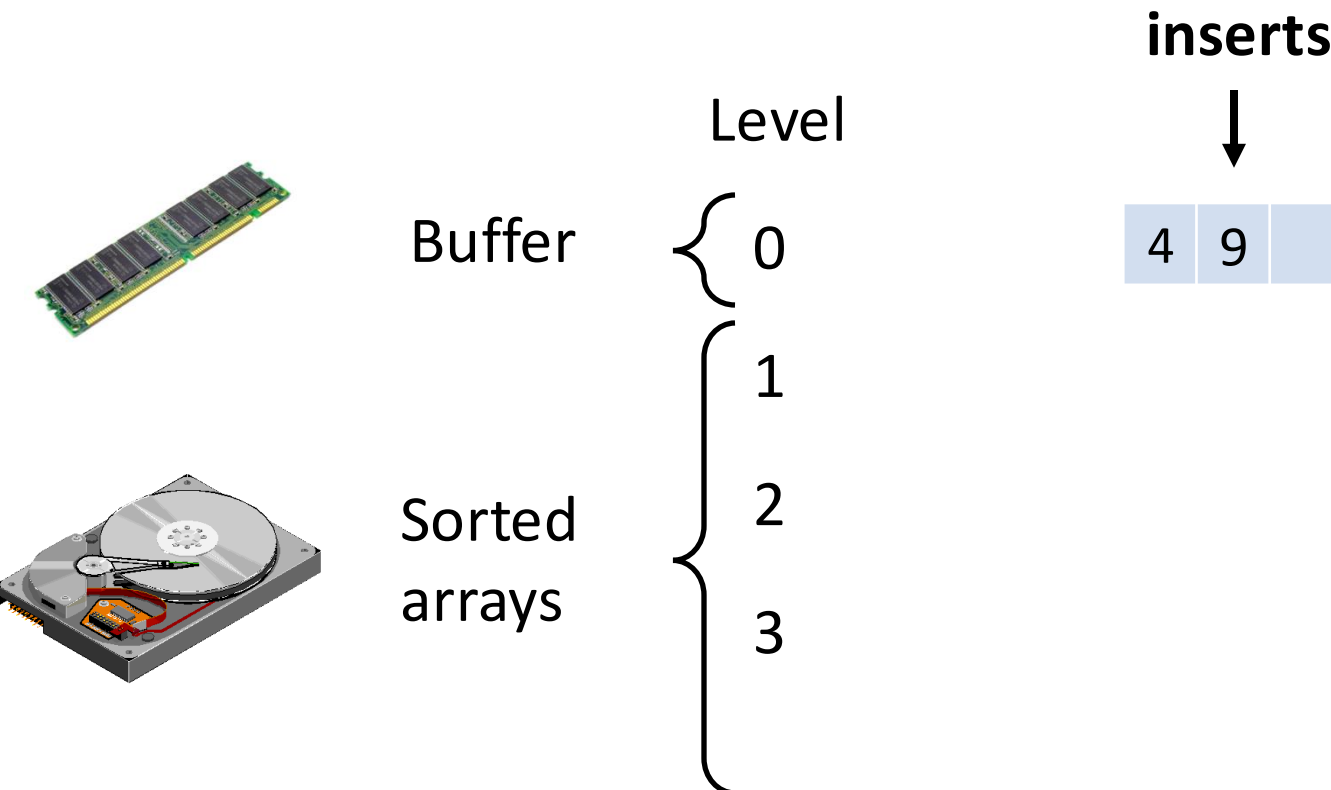
optimize for lookups by sort-merging arrays



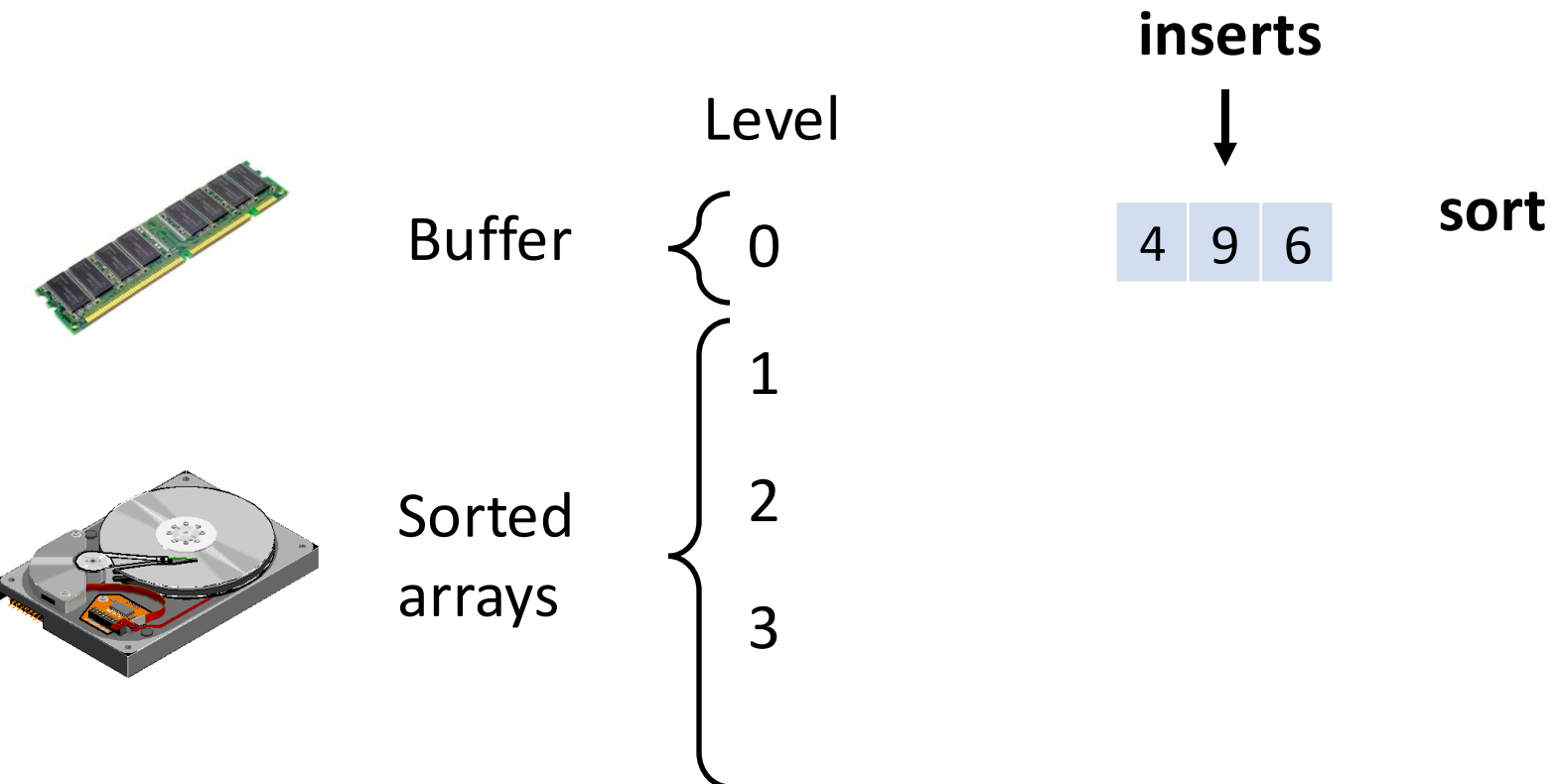
Basic LSM-tree – Example



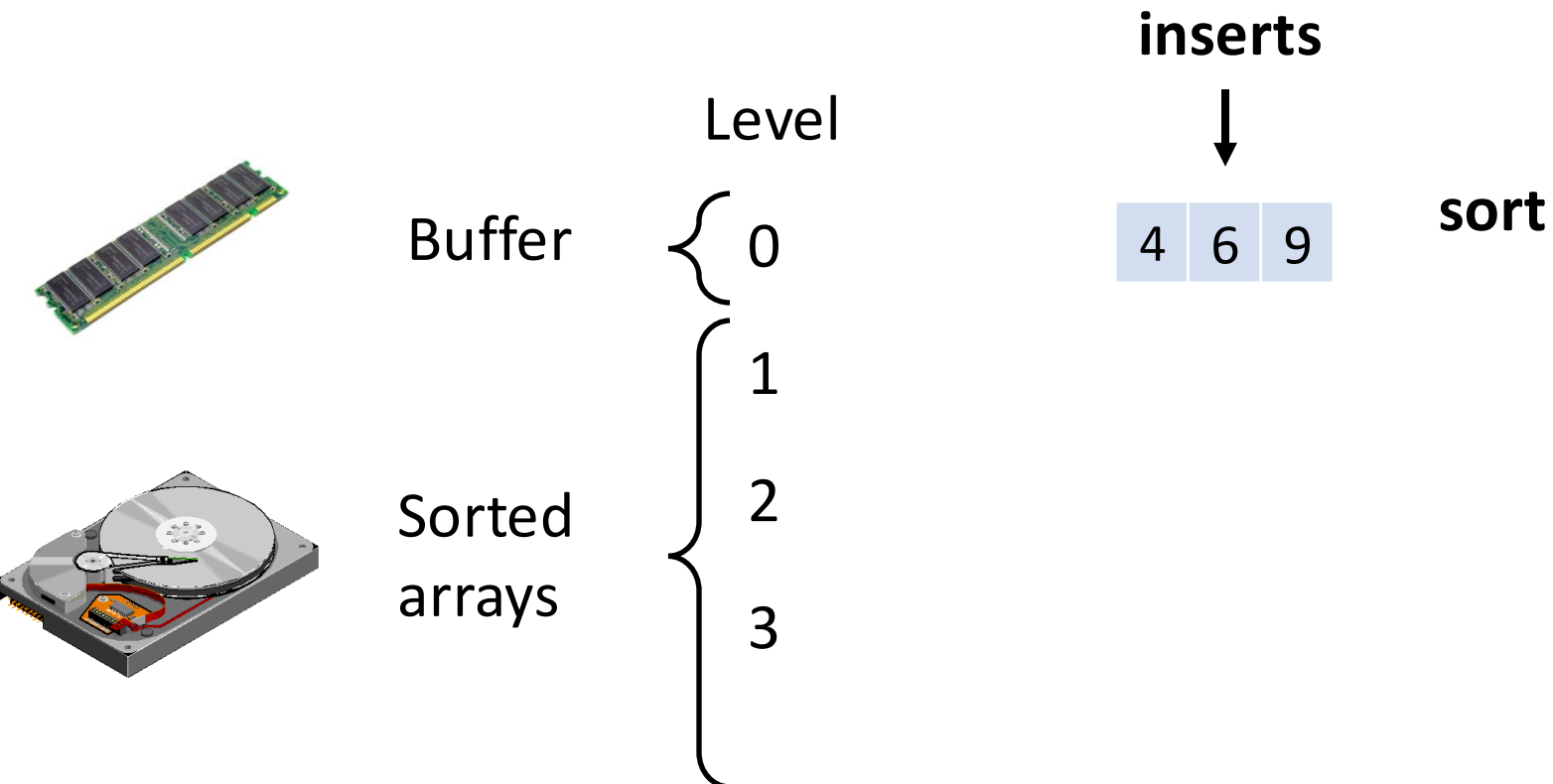
Basic LSM-tree – Example



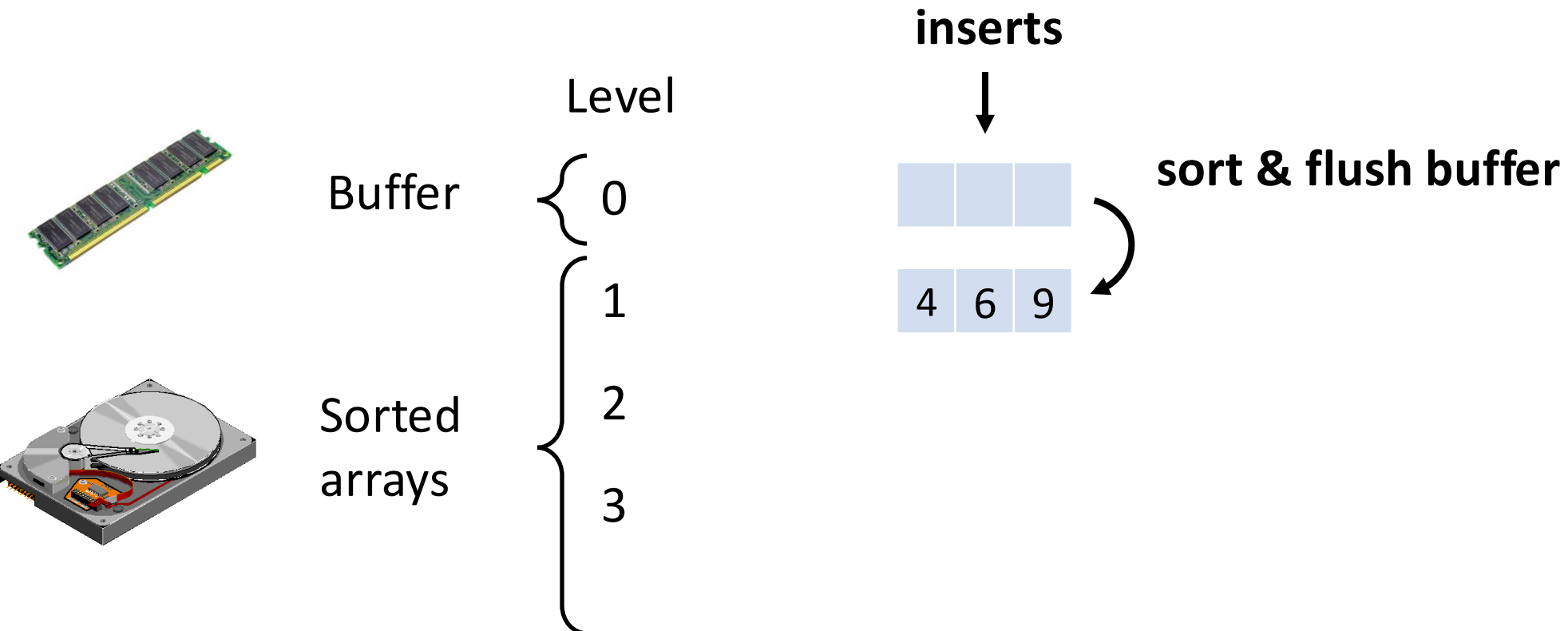
Basic LSM-tree – Example



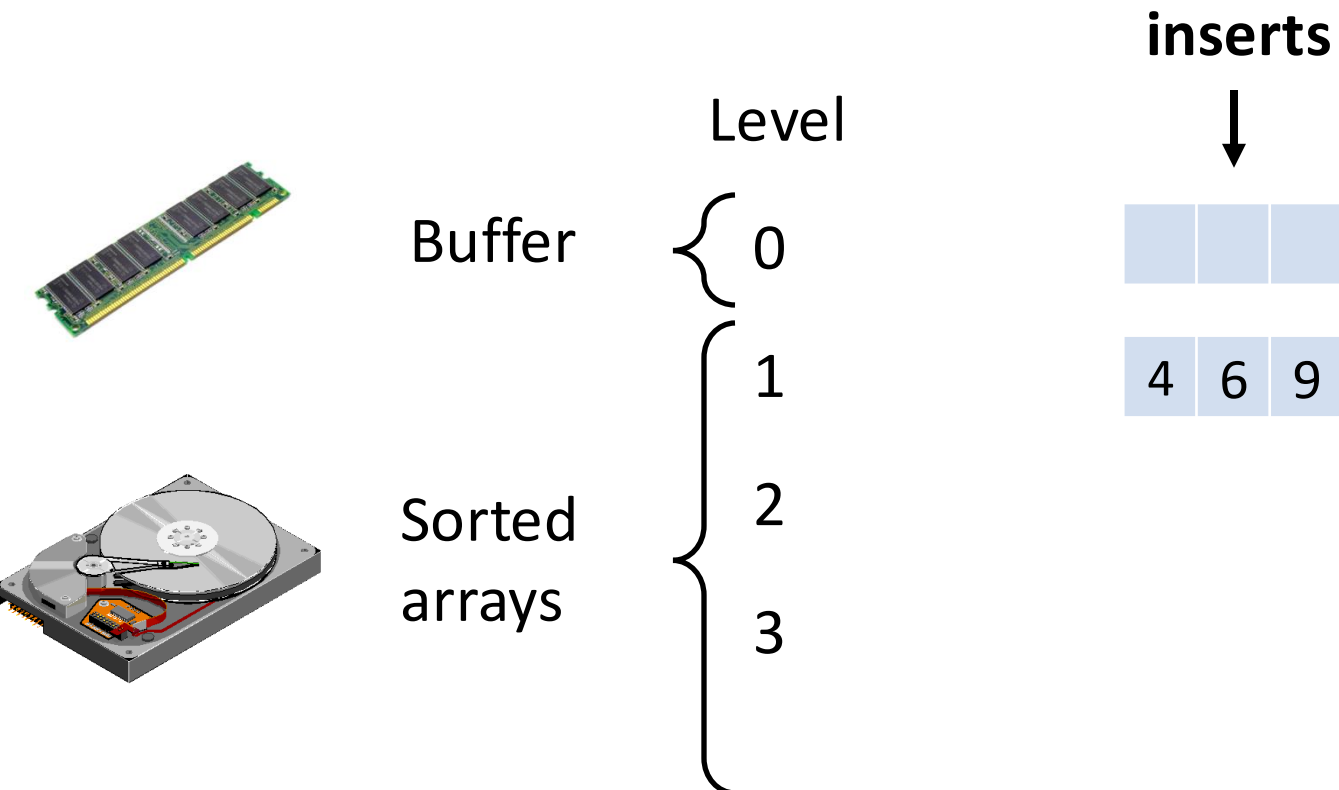
Basic LSM-tree – Example



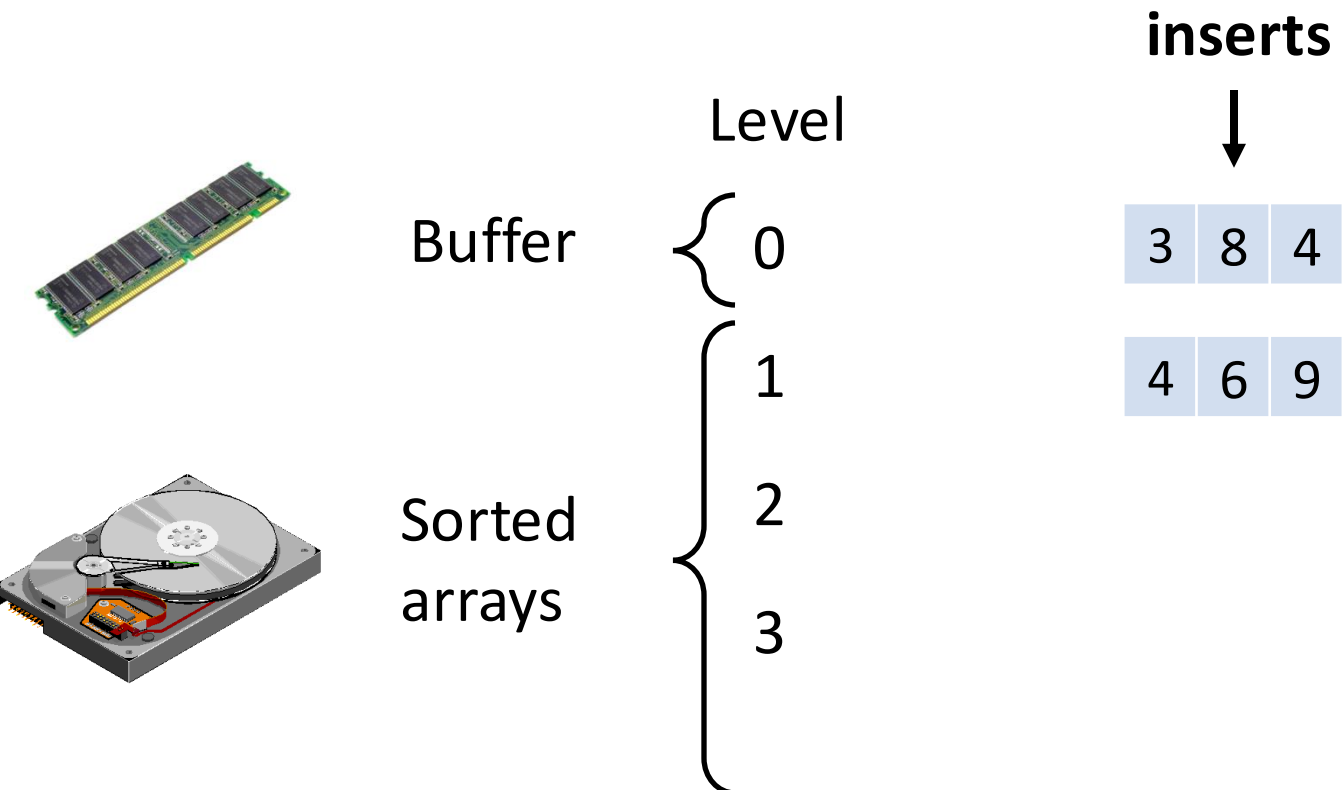
Basic LSM-tree – Example



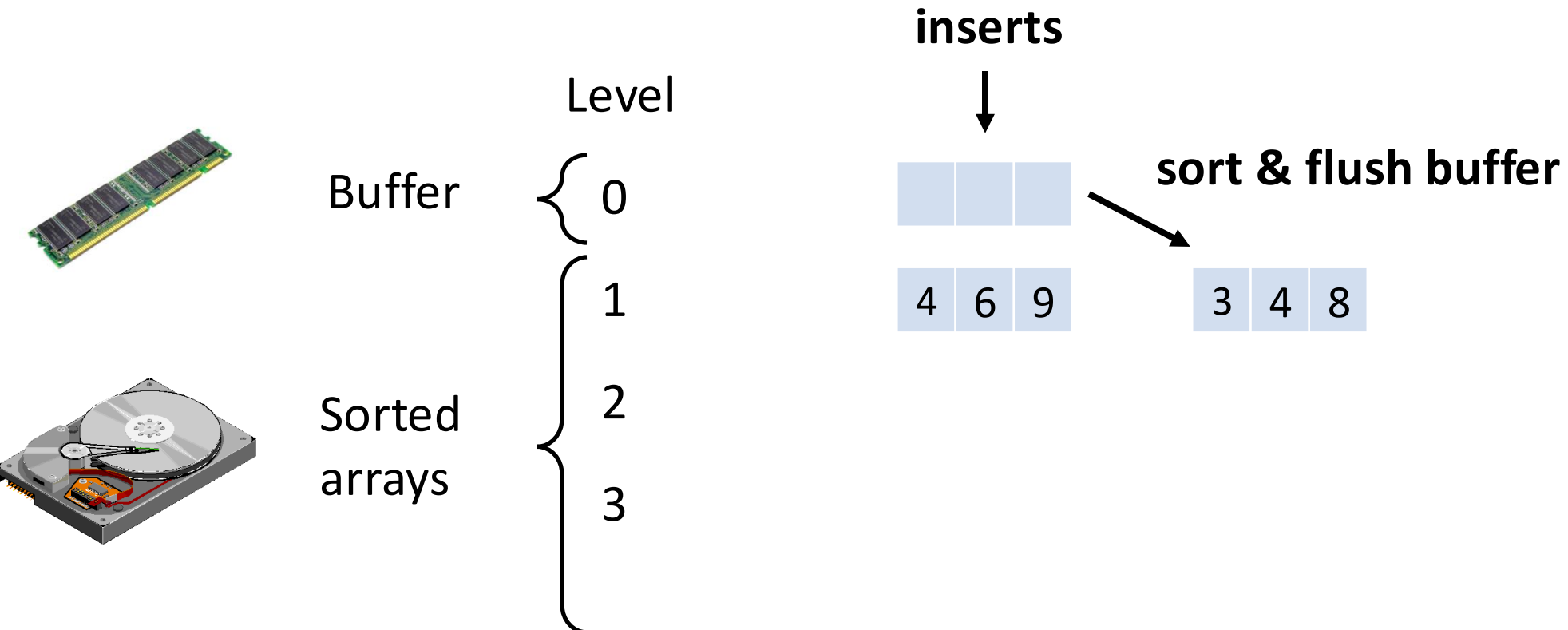
Basic LSM-tree – Example



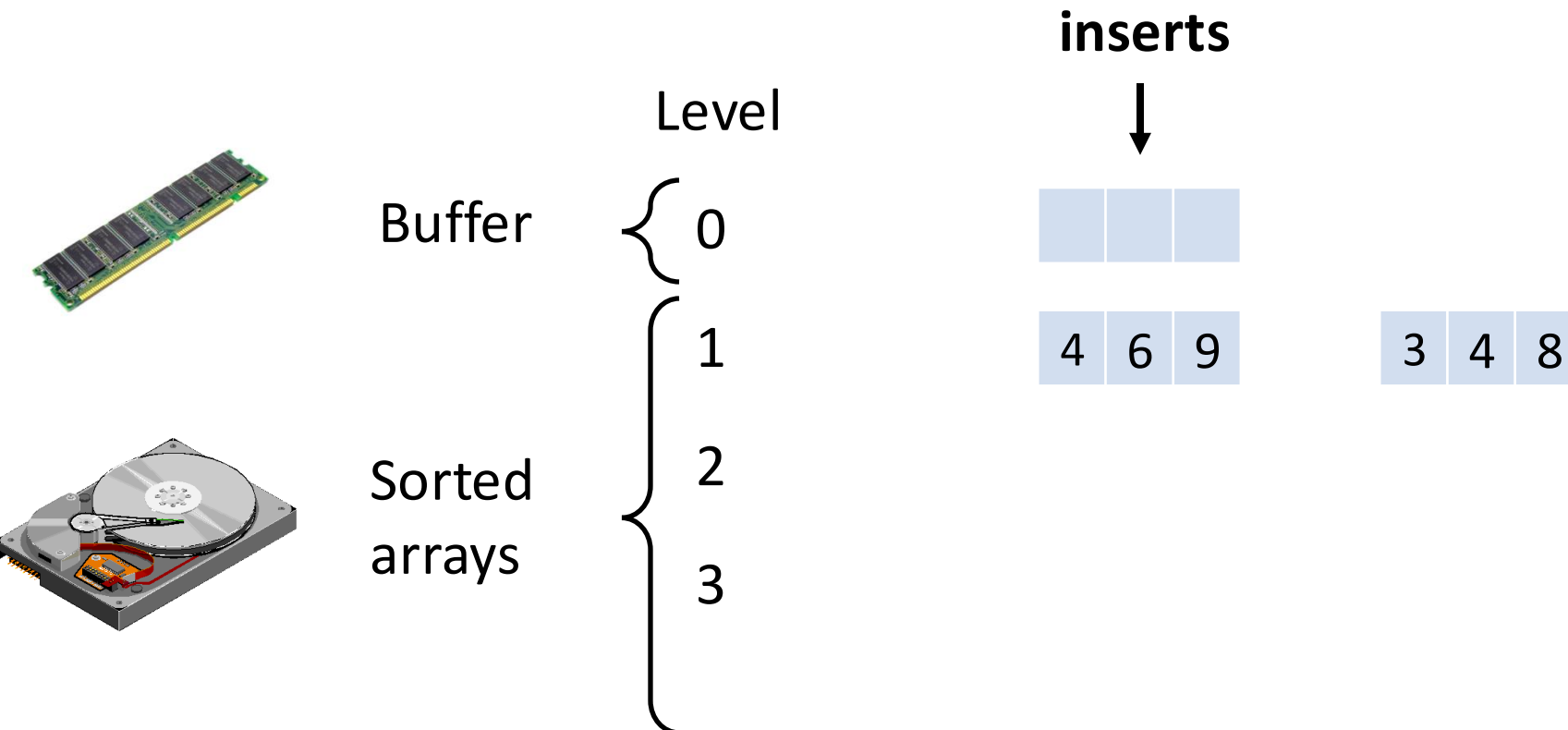
Basic LSM-tree – Example



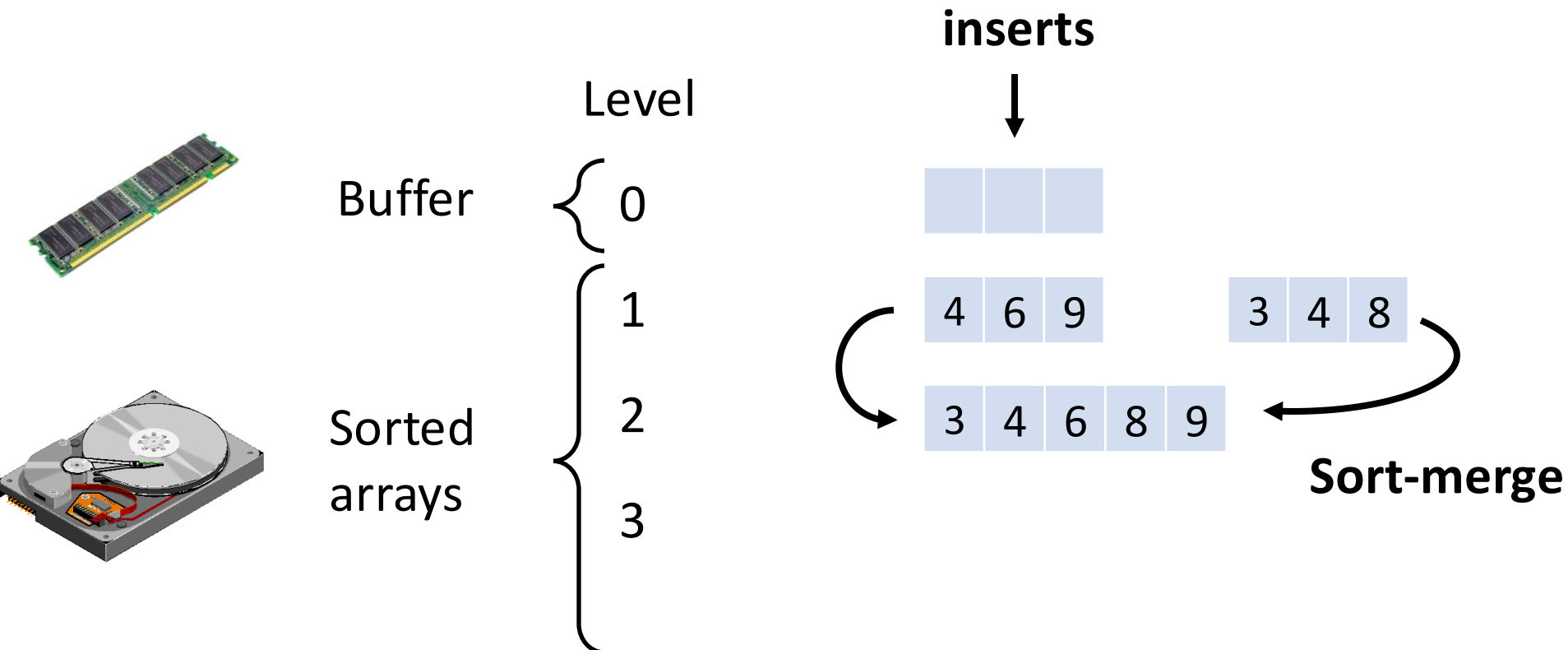
Basic LSM-tree – Example



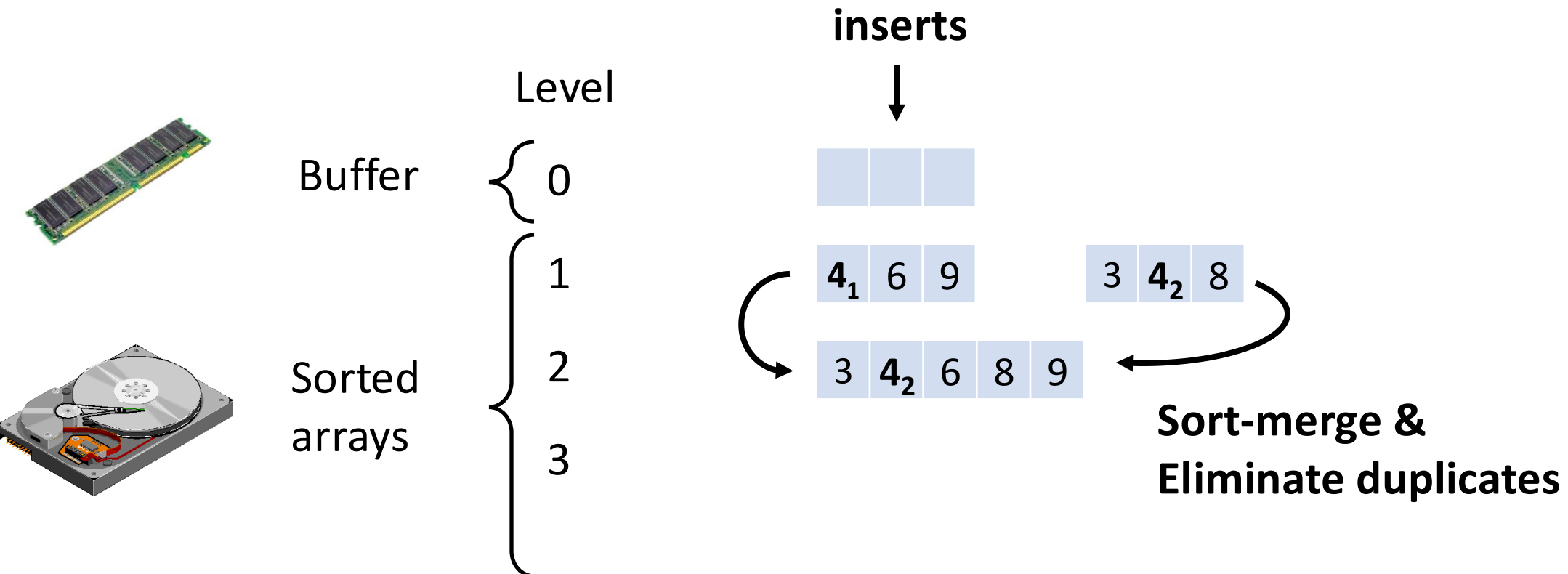
Basic LSM-tree – Example



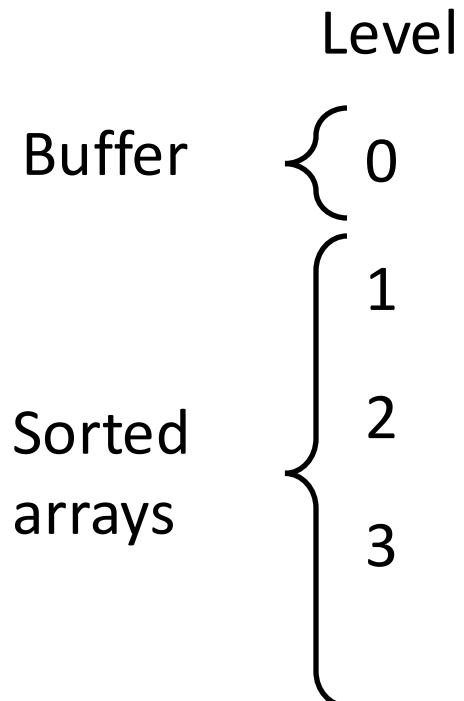
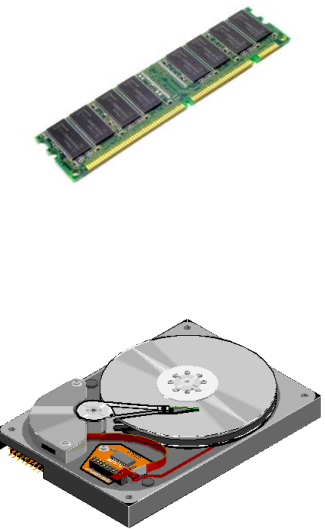
Basic LSM-tree – Example



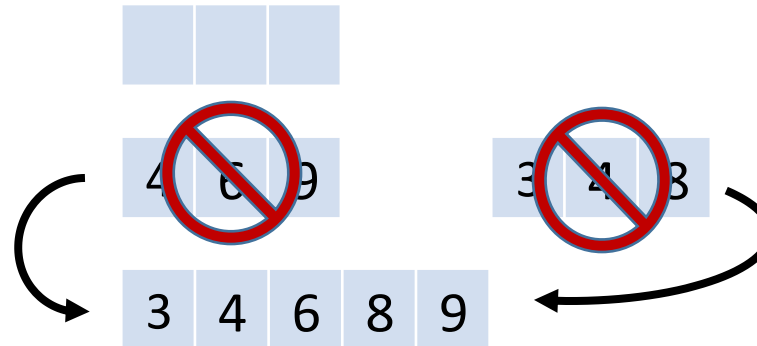
Basic LSM-tree – Example



Basic LSM-tree – Example

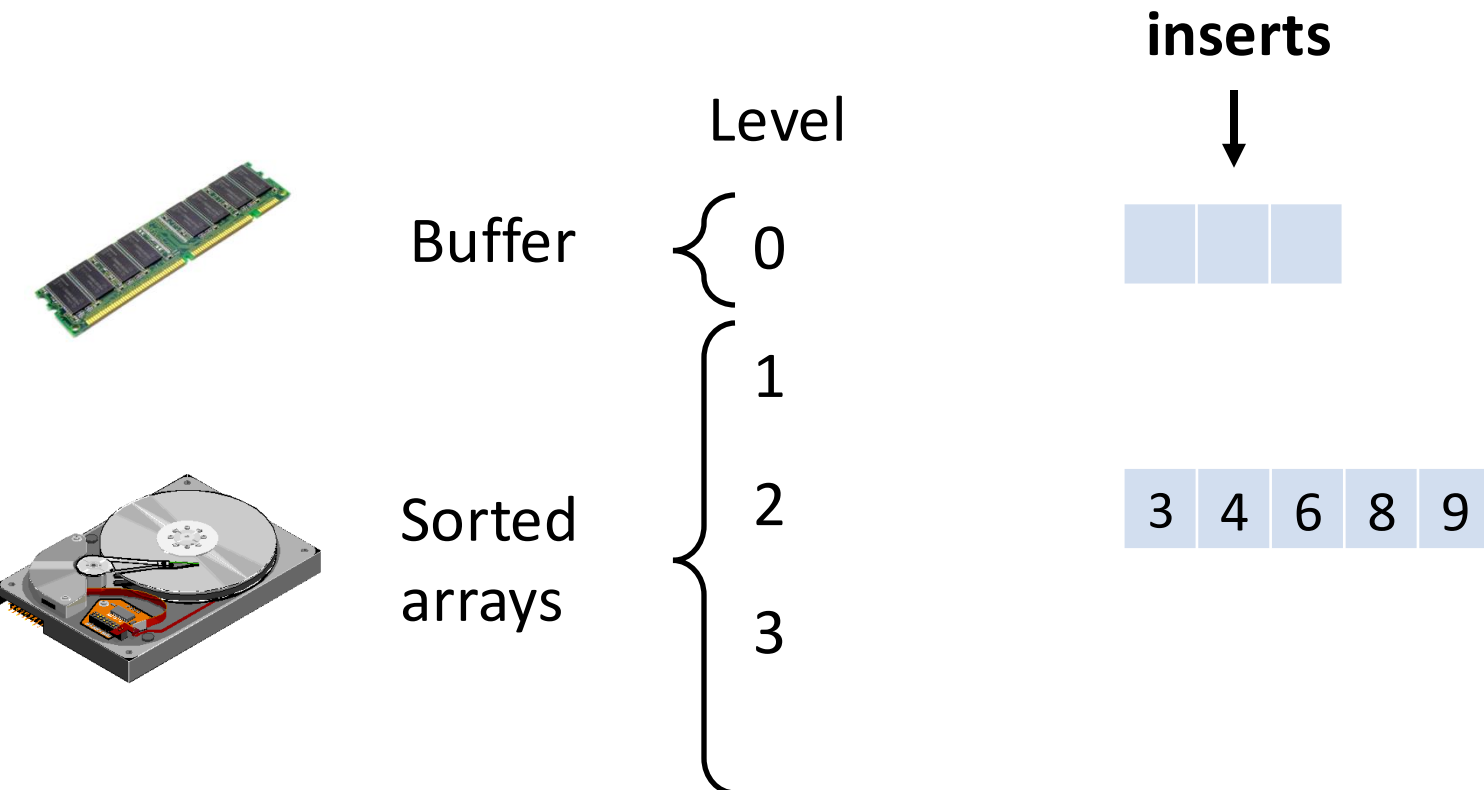


inserts

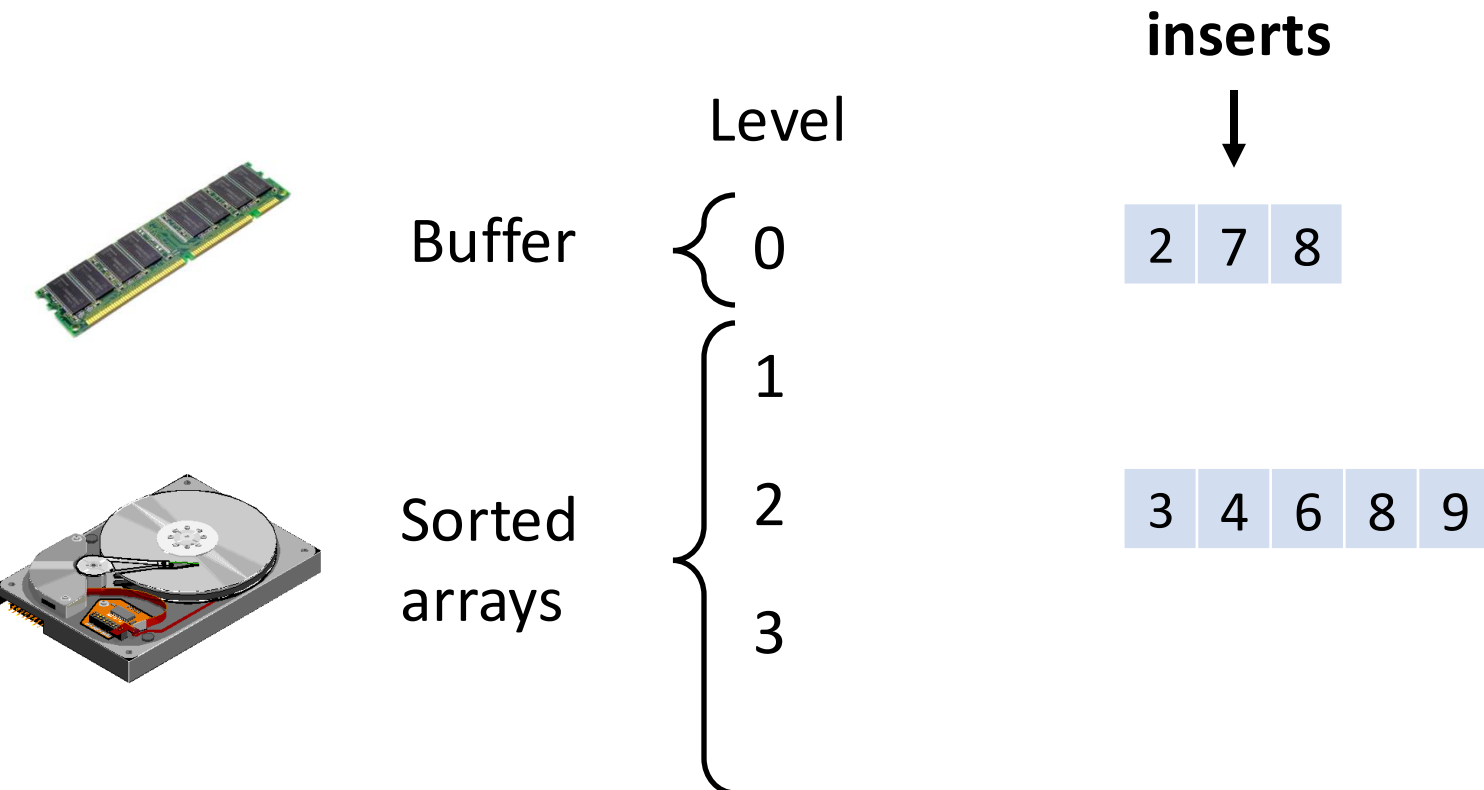


**Sort-merge &
Eliminate duplicates &
Discard original arrays**

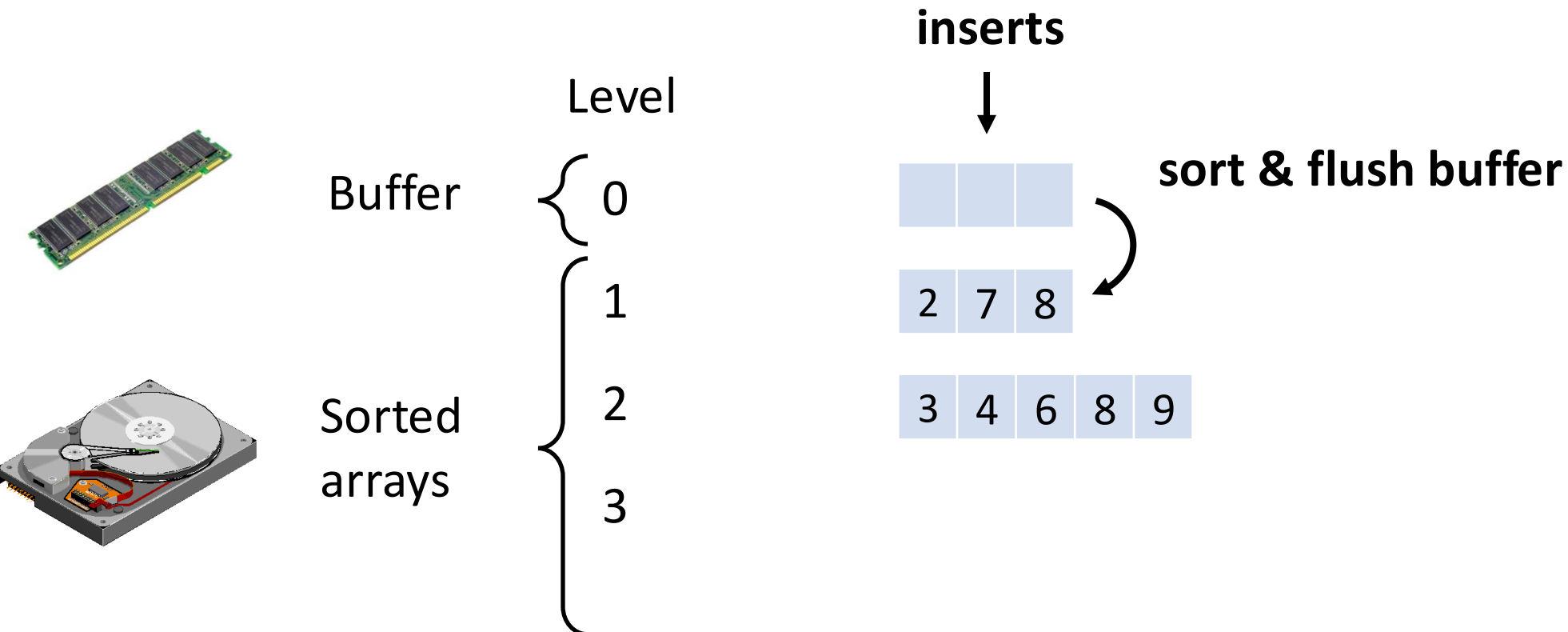
Basic LSM-tree – Example



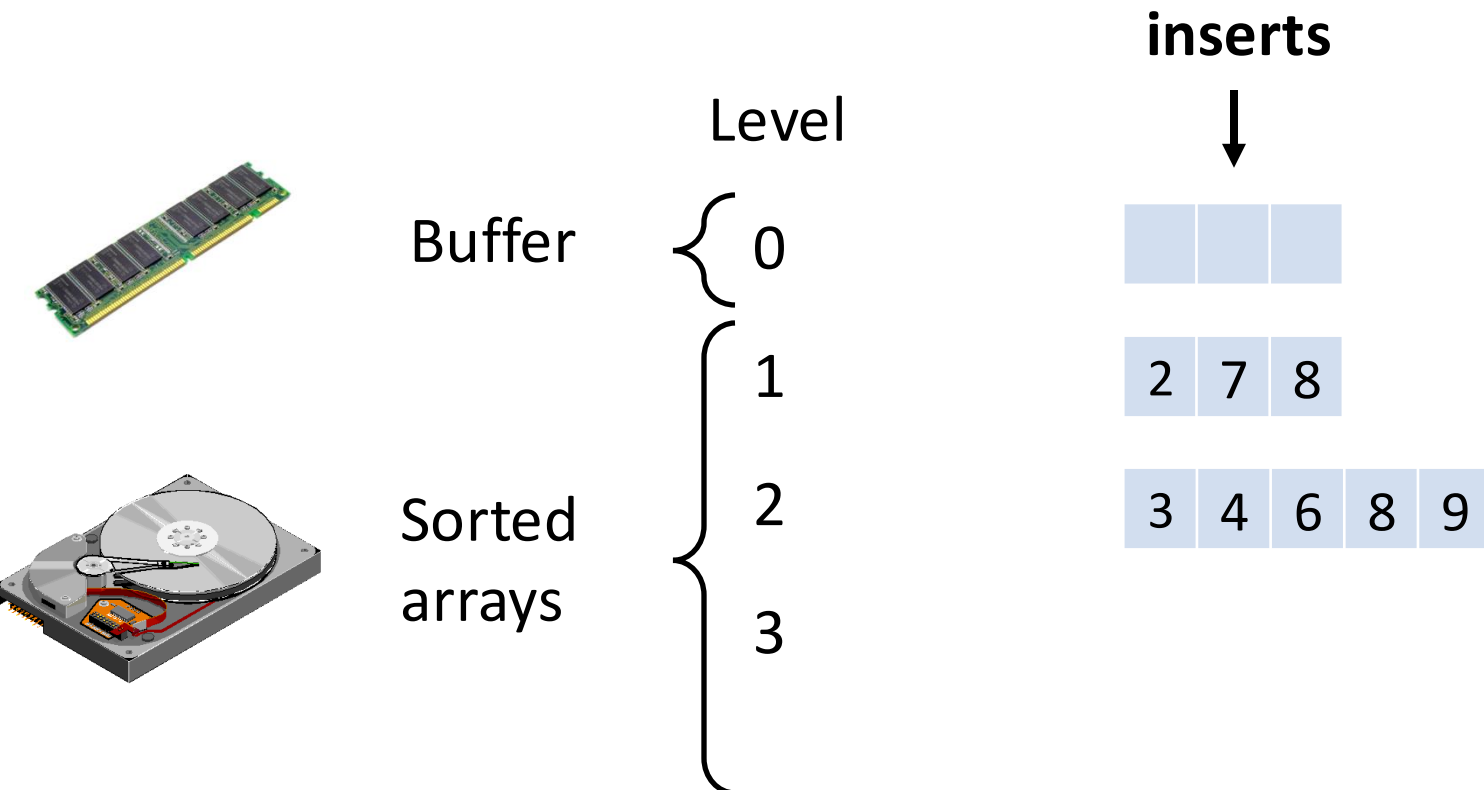
Basic LSM-tree – Example



Basic LSM-tree – Example



Basic LSM-tree – Example



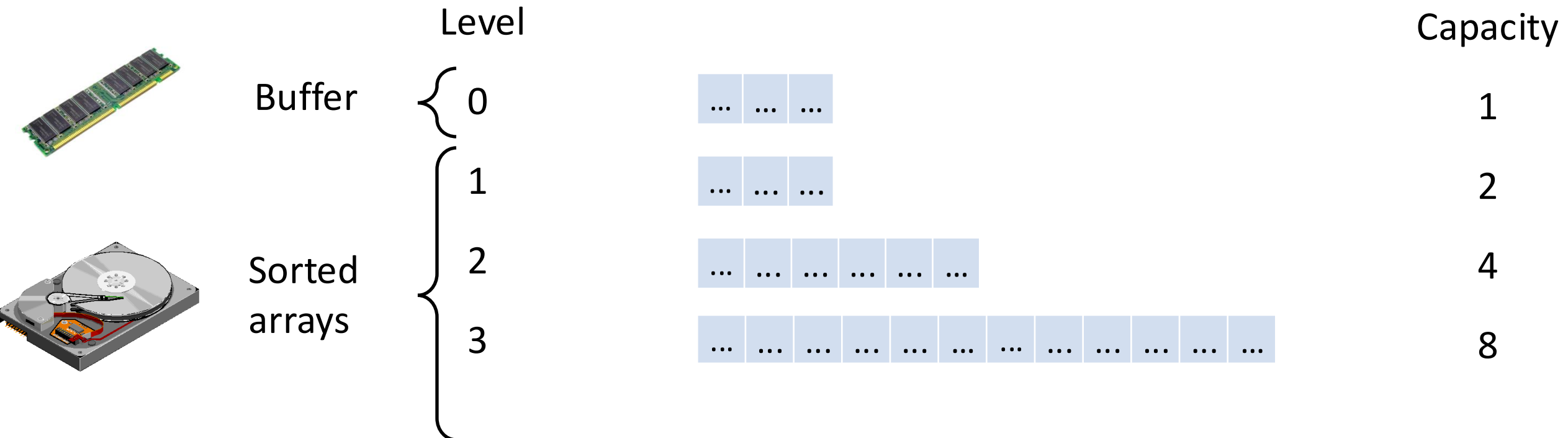
Basic LSM-tree

Levels have exponentially increasing capacities.

How many levels?



$\log_2(N)$



Basic LSM-tree – Lookup cost

Lookup method?

Search youngest to oldest.

$$O(\log_2(N))$$

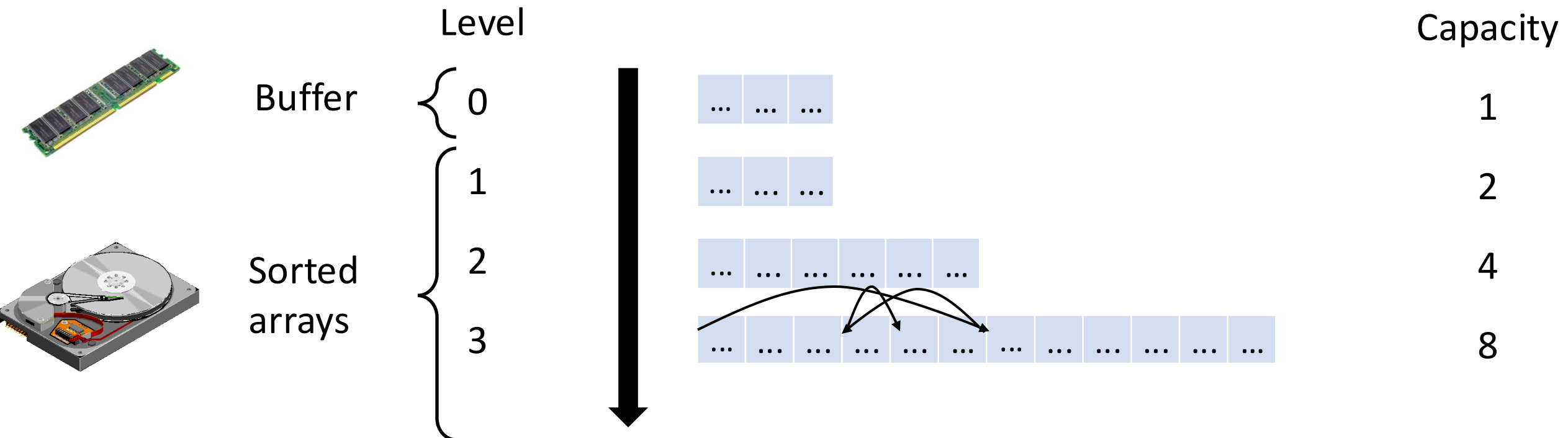
How?

Binary search.

$$O(\log_2(N_i)) = O(\log_2(N)) \quad N > N_i$$

Lookup cost?

$$O(\log_2(N)^2)$$



Basic LSM-tree – Insertion cost

How many times is each entry copied?

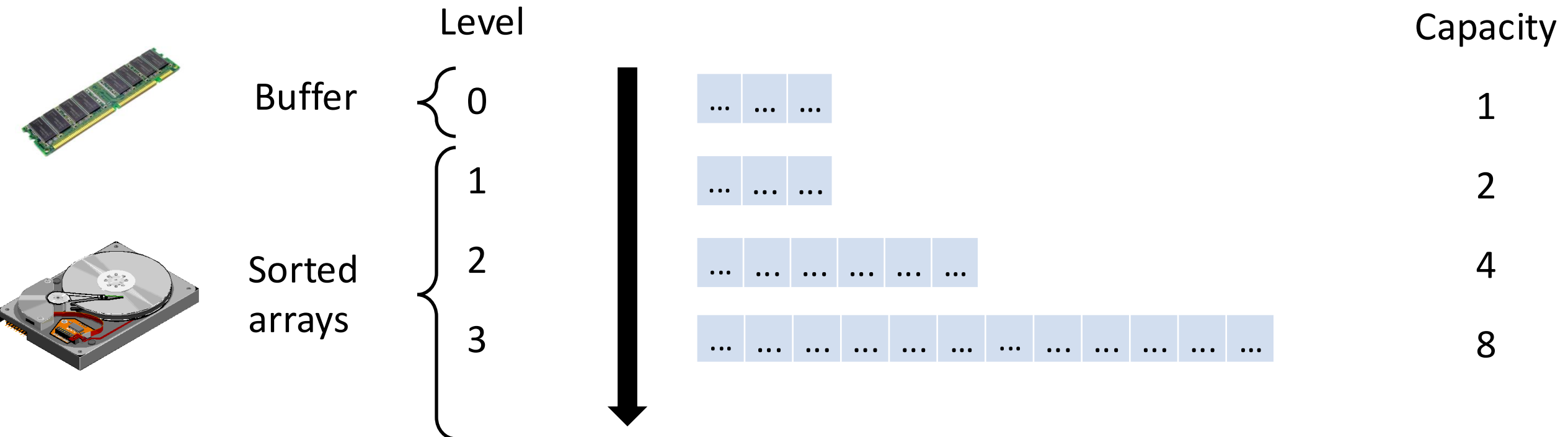
$O(\log_2(N))$, once per level

What is the price of each copy?

$O(1/B)$, amortized

Total insert cost?

$O((1/B) \cdot \log_2(N))$



Results Catalogue

	Lookup cost	Insertion cost
Sorted array	$O(\log_2(N))$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N)^2)$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree		
Tiered LSM-tree		

Results Catalogue

Better insert cost and **worse lookup cost** compared with B-trees

	Lookup cost	Insertion cost
Sorted array	$O(\log_2(N))$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N)^2)$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree		
Tiered LSM-tree		

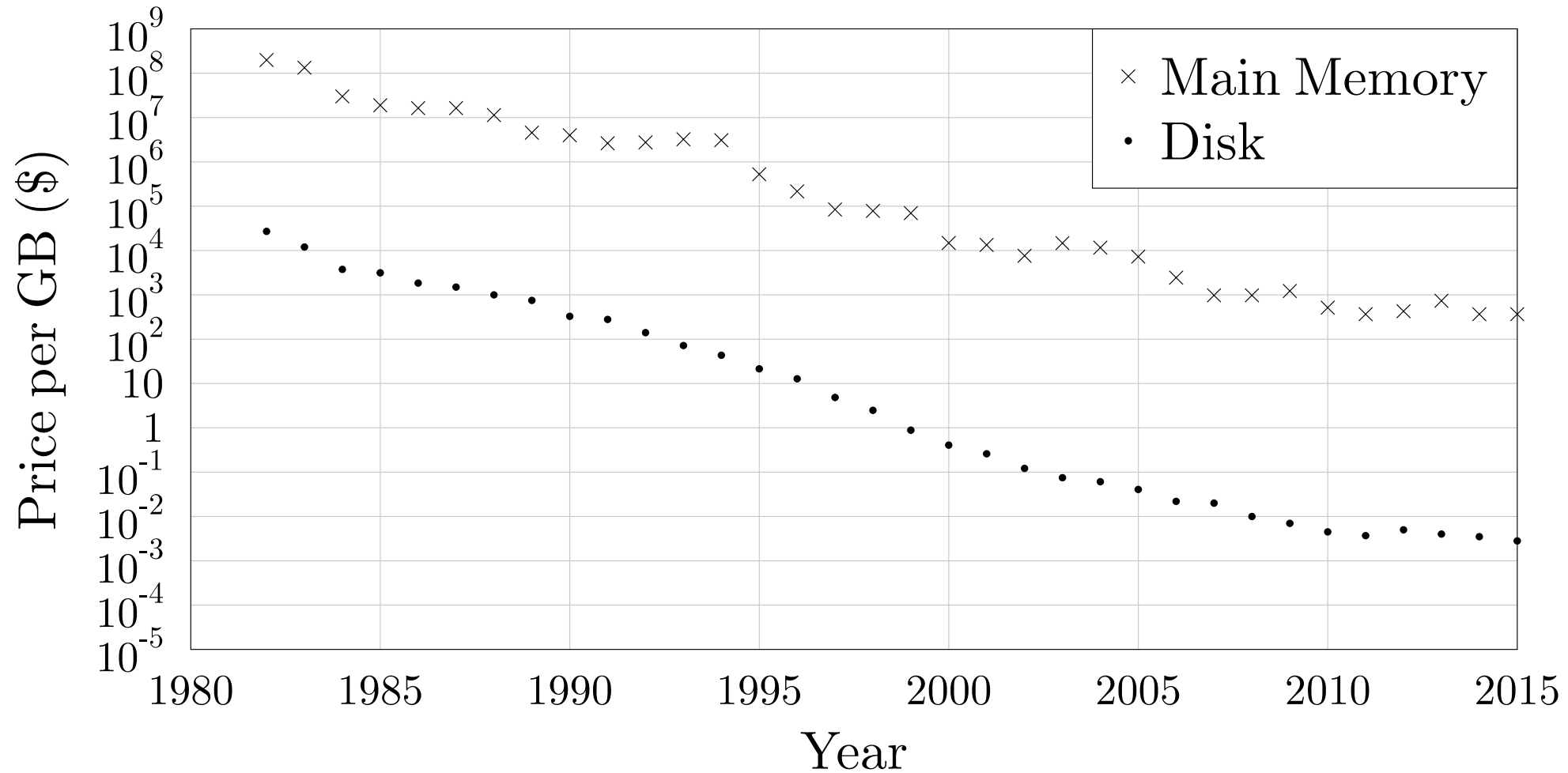
Results Catalogue

Better insert cost and **worse lookup cost** compared with B-trees

Can we improve the lookup cost?

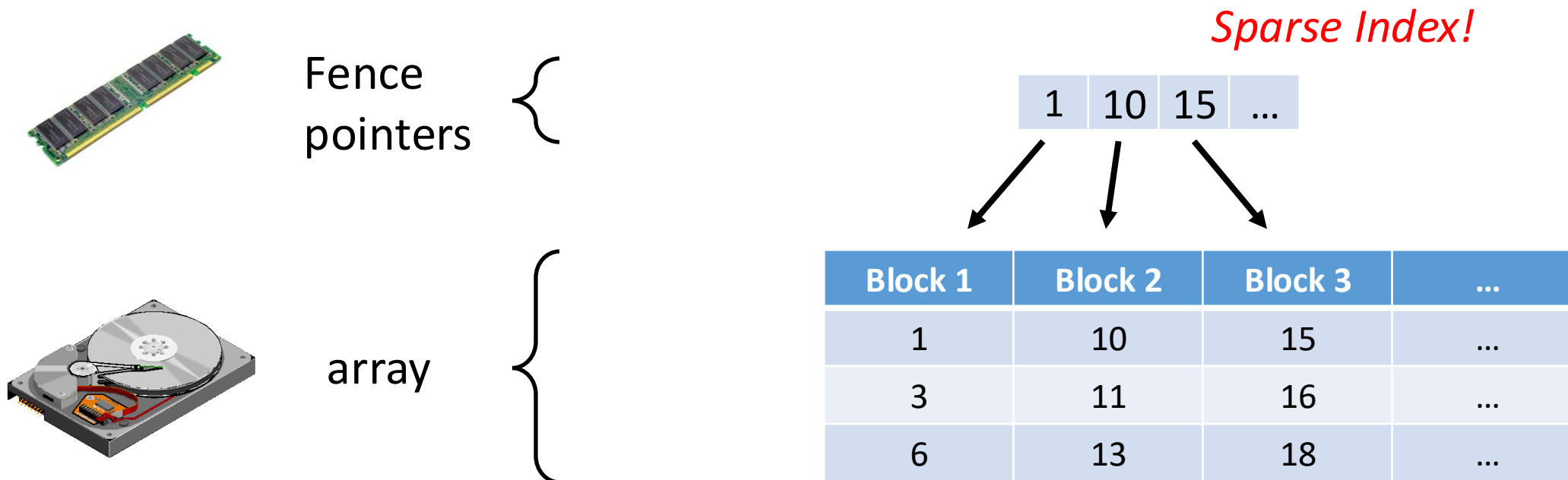
	Lookup cost	Insertion cost
Sorted array	$O(\log_2(N))$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N)^2)$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree		
Tiered LSM-tree		

Declining Main Memory Cost



Declining Main Memory Cost

Store a fence pointer for every block in main memory



Results Catalogue – with fence pointers

	Lookup cost	Insertion cost
Sorted array	$O(\log_2(N))$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N)^2)$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree		
Tiered LSM-tree		

Results Catalogue – with fence pointers

	Lookup cost	Insertion cost
Sorted array	$O(\log_2(N))$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N)^2)$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree		
Tiered LSM-tree		

Results Catalogue – with fence pointers

	Lookup cost	Insertion cost
Sorted array	$O(1)$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N)^2)$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree		
Tiered LSM-tree		

Results Catalogue – with fence pointers

	Lookup cost	Insertion cost
Sorted array	$O(1)$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N)^2)$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree		
Tiered LSM-tree		

Results Catalogue – with fence pointers

	Lookup cost	Insertion cost
Sorted array	$O(1)$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N)^2)$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree		
Tiered LSM-tree		

Results Catalogue – with fence pointers

	Lookup cost	Insertion cost
Sorted array	$O(1)$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N)^2)$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree		
Tiered LSM-tree		

Results Catalogue – with fence pointers

	Lookup cost	Insertion cost
Sorted array	$O(1)$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N))$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree		
Tiered LSM-tree		

Results Catalogue – with fence pointers

Quick sanity check:

suppose

$$N = 2^{32}$$

and

$$B = 2^{10}$$

	Lookup cost	Insertion cost
Sorted array	$O(1)$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N))$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree		
Tiered LSM-tree		

Results Catalogue – with fence pointers

Quick sanity check:

suppose

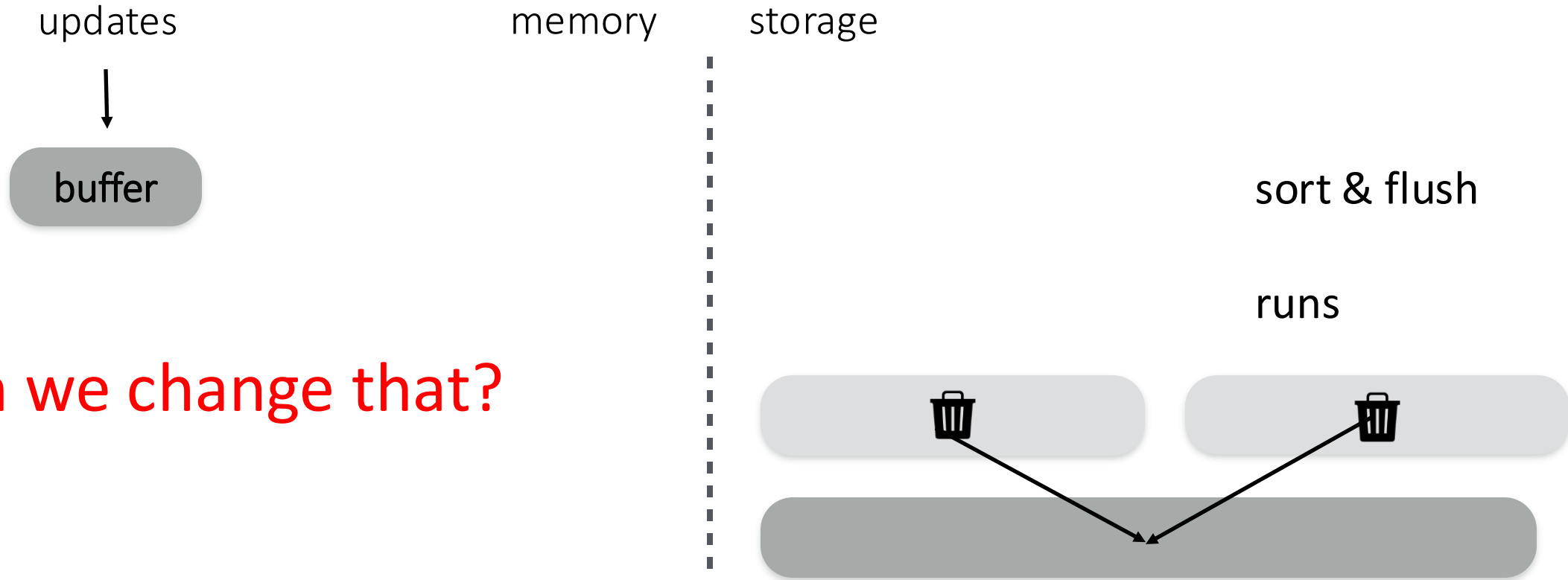
$$N = 2^{32}$$

and

$$B = 2^{10}$$

	Lookup cost	Insertion cost
Sorted array	$O(1)$	$O(2^{31})$
Log	$O(2^{32})$	$O(2^{-10})$
B-tree	$O(4)$	$O(4)$
Basic LSM-tree	$O(32)$	$O(2^{-10} \cdot 32)$
Leveled LSM-tree		
Tiered LSM-tree		

Up until now we always create levels by merging **two** files!



Can we change that?

Leveled LSM-tree

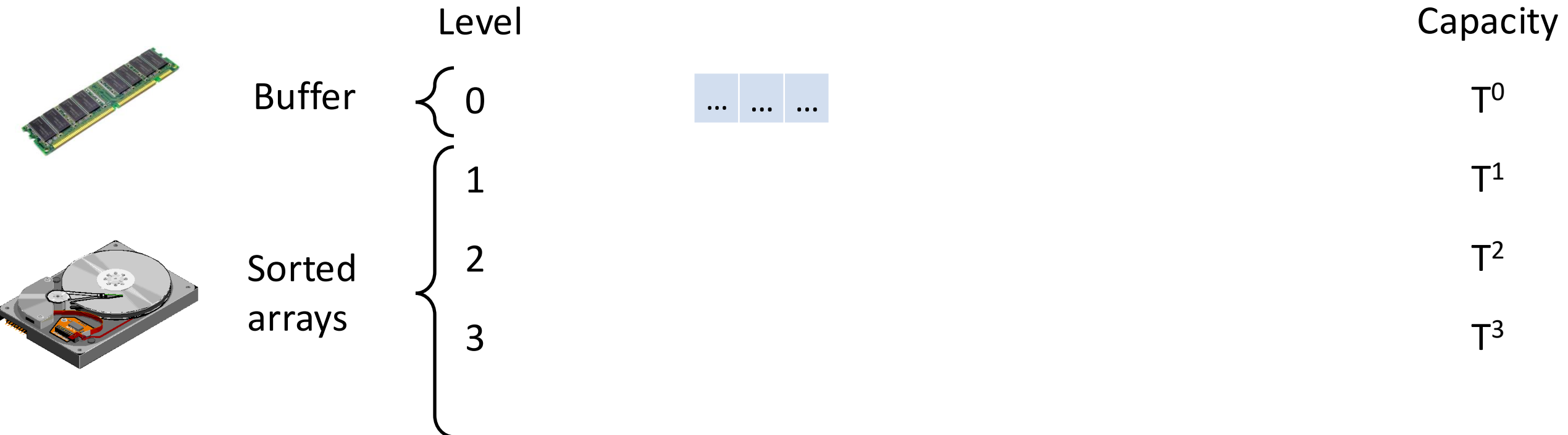
 Lookup cost

 Update cost

Leveled LSM-tree

Lookup cost depends on number of levels
How to reduce it?

Increase size ratio T



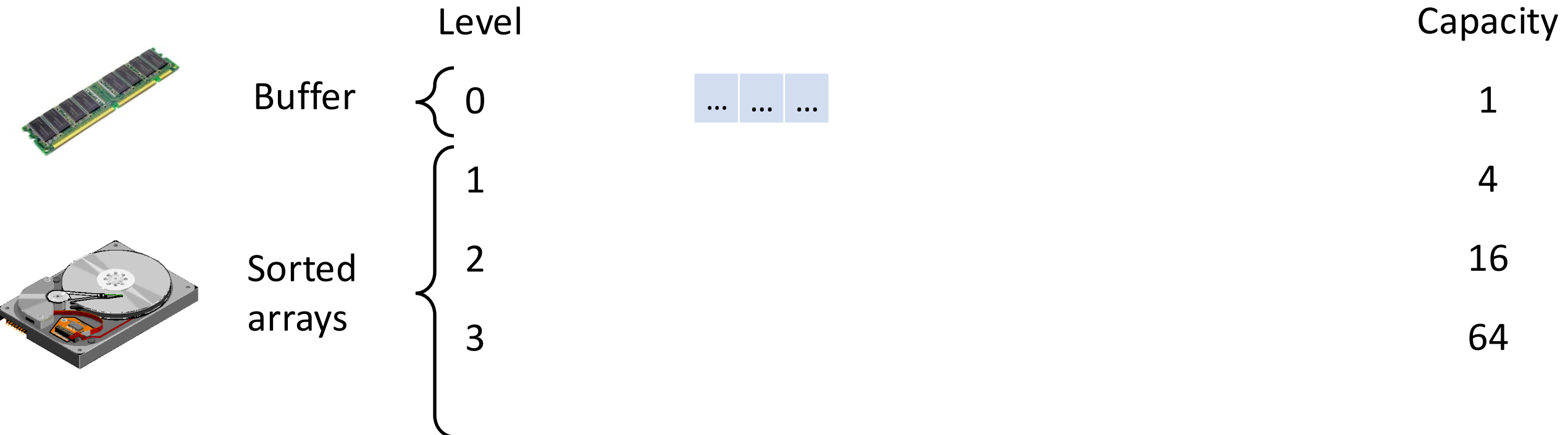
Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

E.g. size ratio of 4

Increase size ratio T



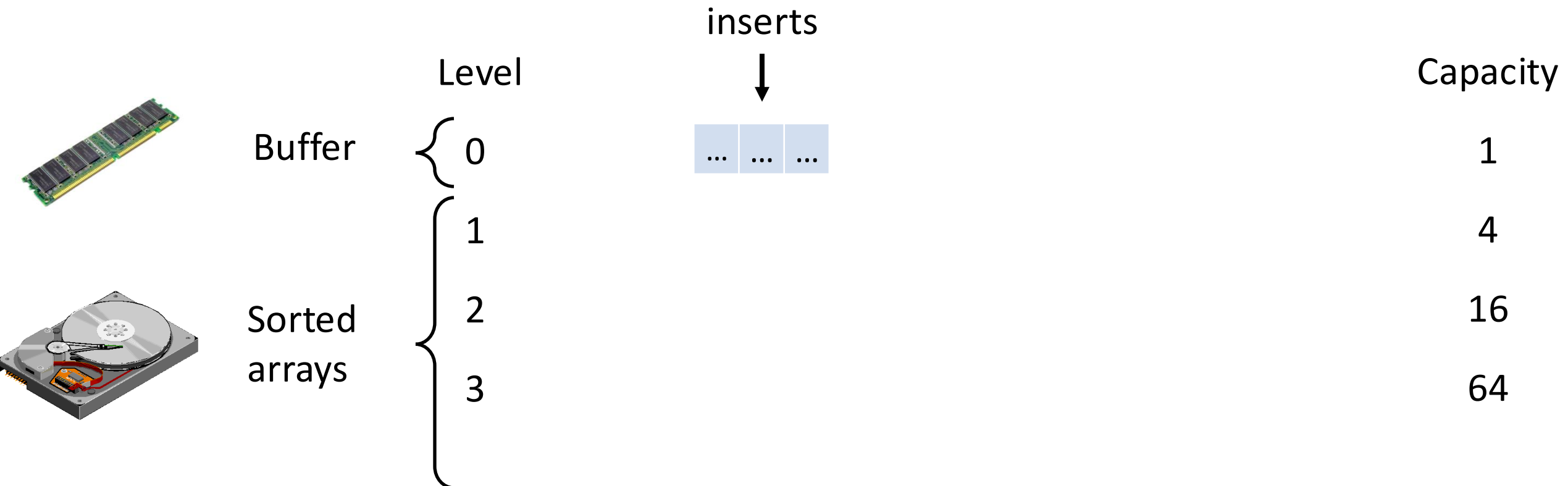
Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

E.g. size ratio of 4

Increase size ratio T



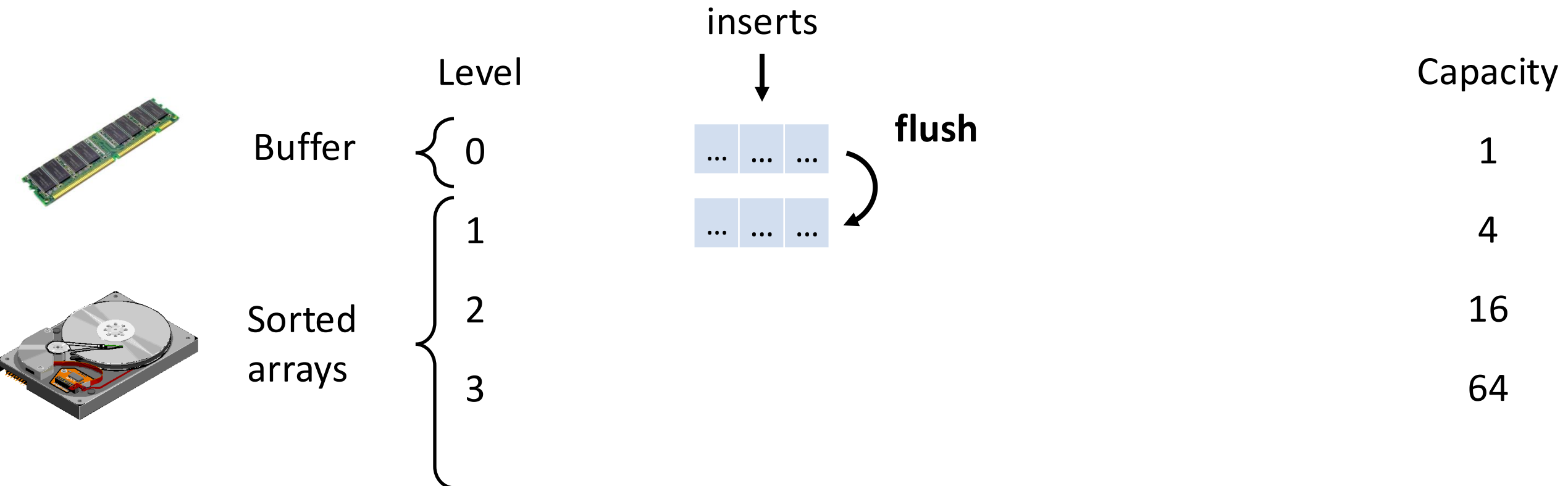
Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

E.g. size ratio of 4

Increase size ratio T



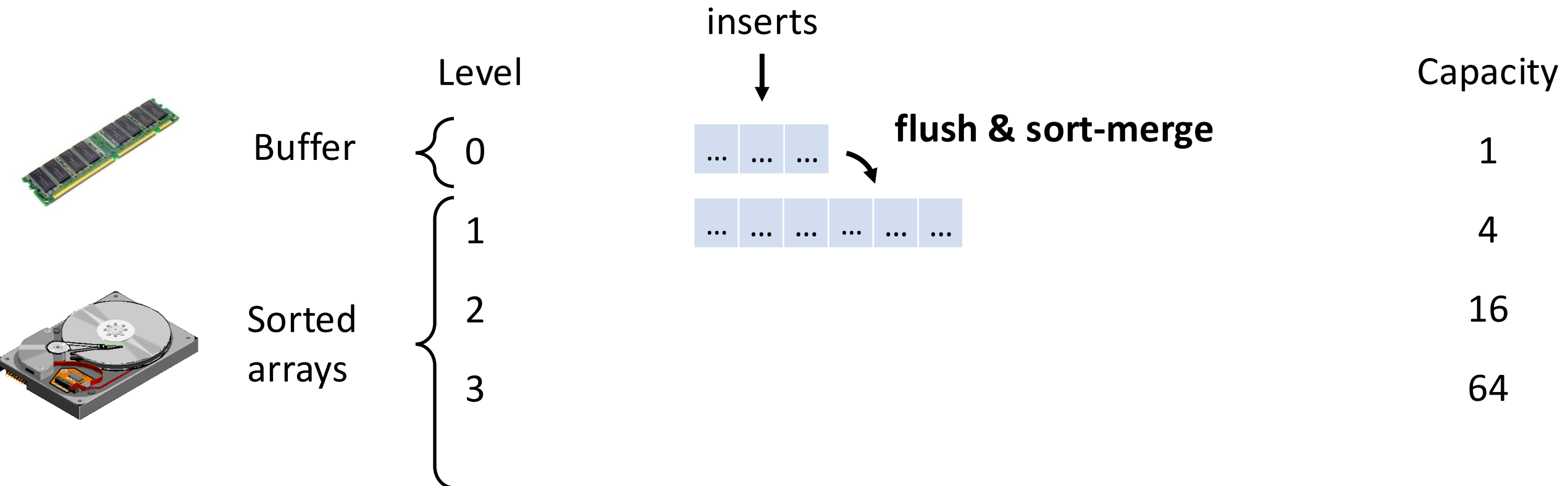
Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

E.g. size ratio of 4

Increase size ratio T



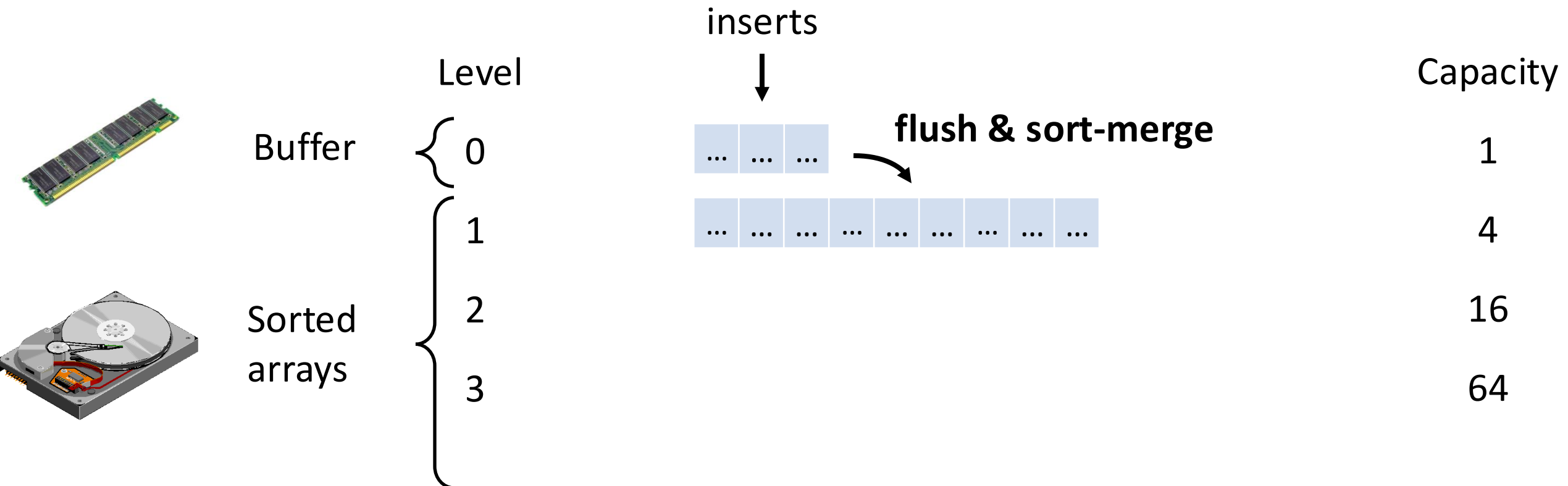
Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

E.g. size ratio of 4

Increase size ratio T



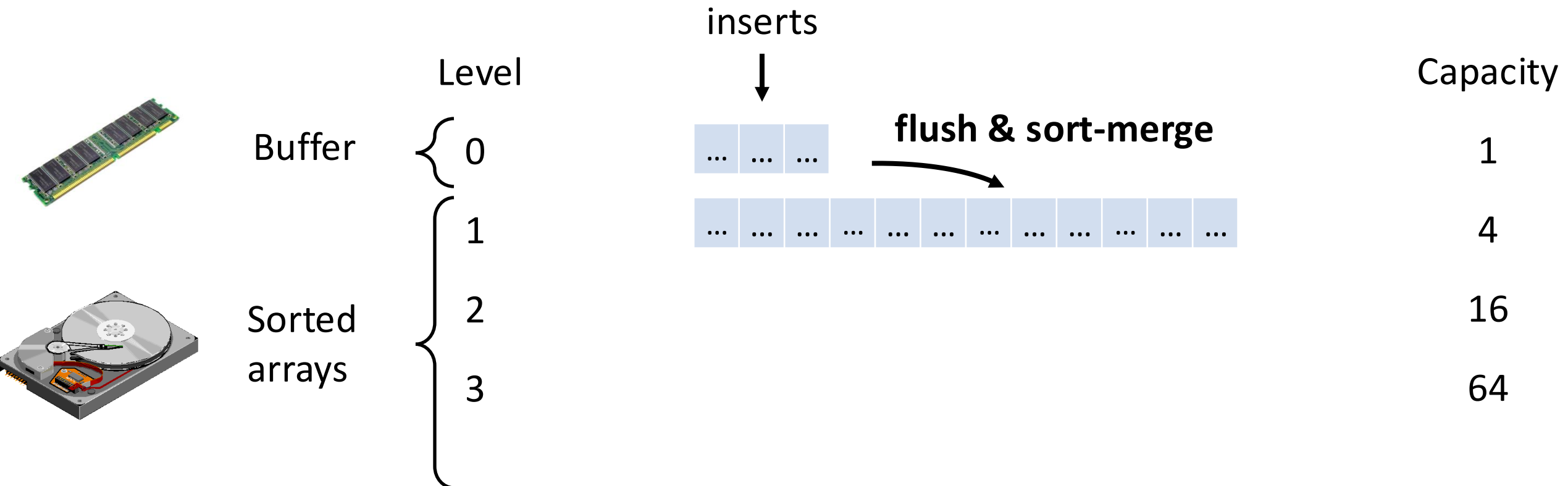
Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

E.g. size ratio of 4

Increase size ratio T



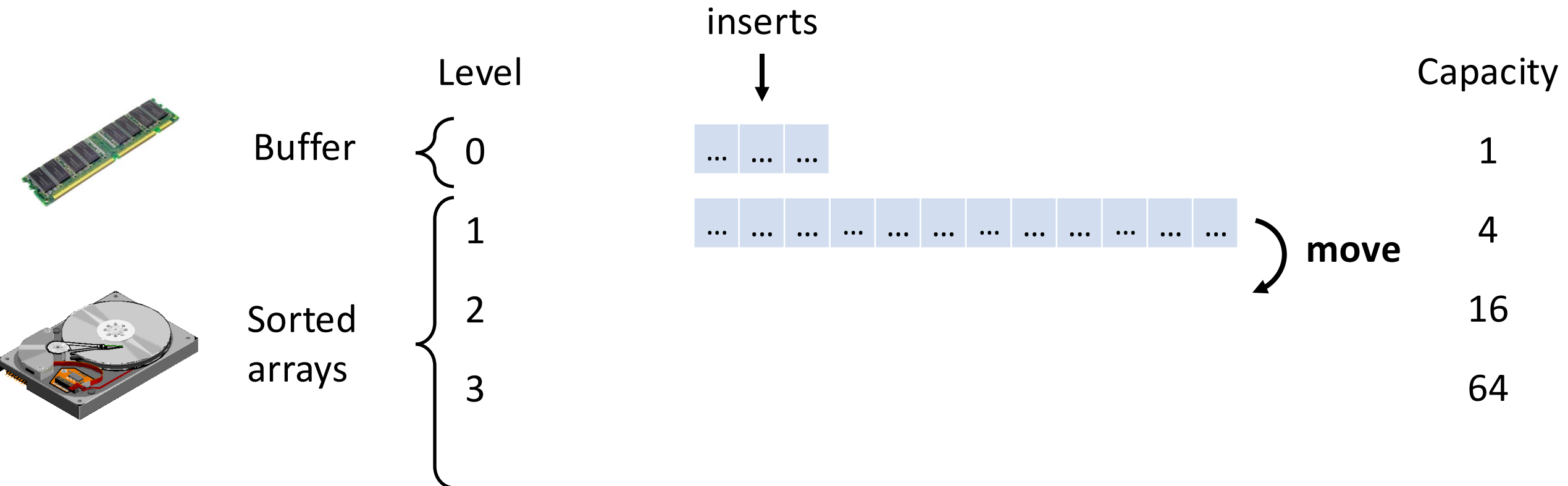
Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

E.g. size ratio of 4

Increase size ratio T



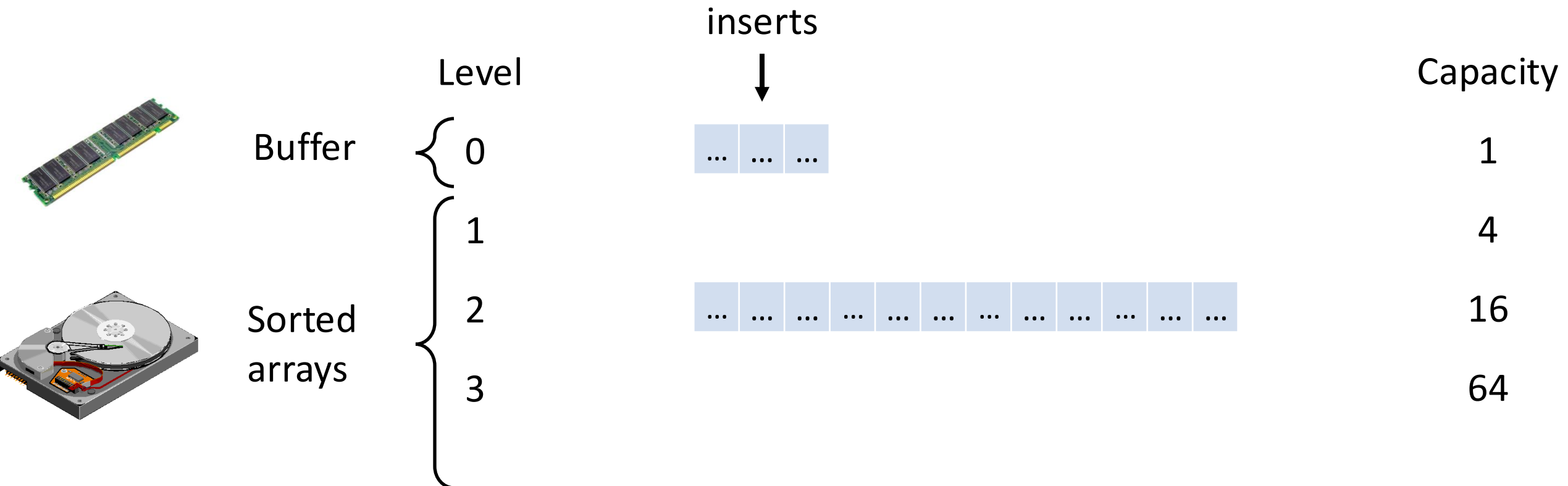
Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

E.g. size ratio of 4

Increase size ratio T



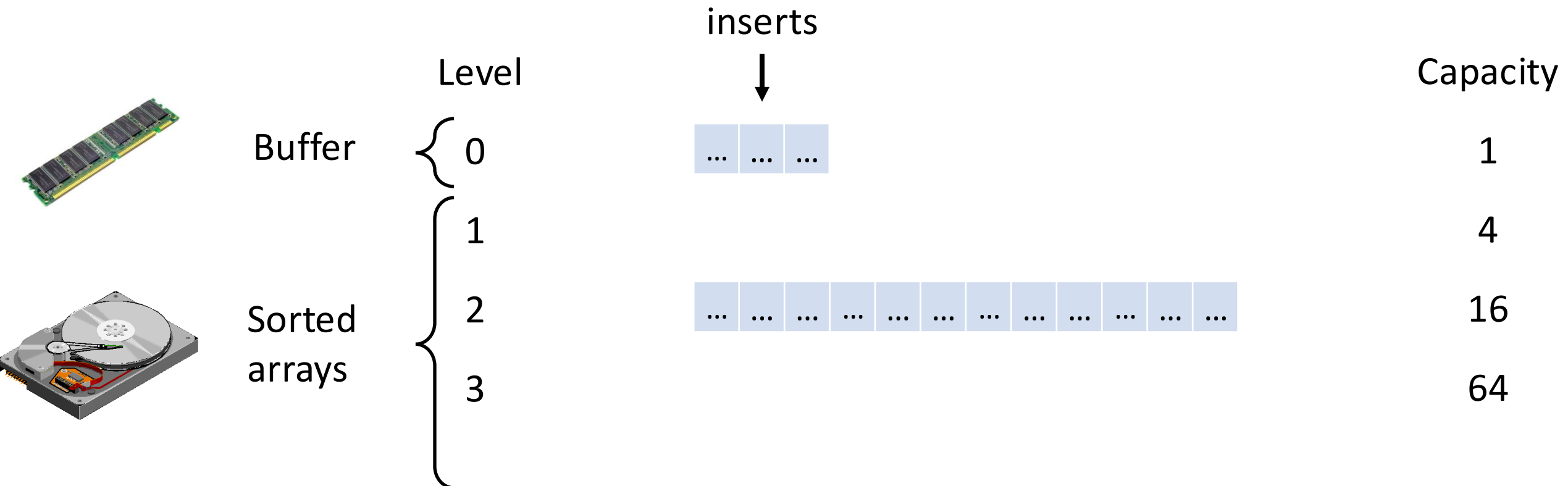
Leveled LSM-tree

Lookup cost?


$$O(\log_T(N))$$


Insertion cost?

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



Leveled LSM-tree

 Lookup cost?
 $O(\log_T(N))$

Insertion cost? 
 $O\left(\frac{T}{B} \cdot \log_T(N)\right)$

What happens as we increase the size ratio T ?

What happens when size ratio T is set to be N ?

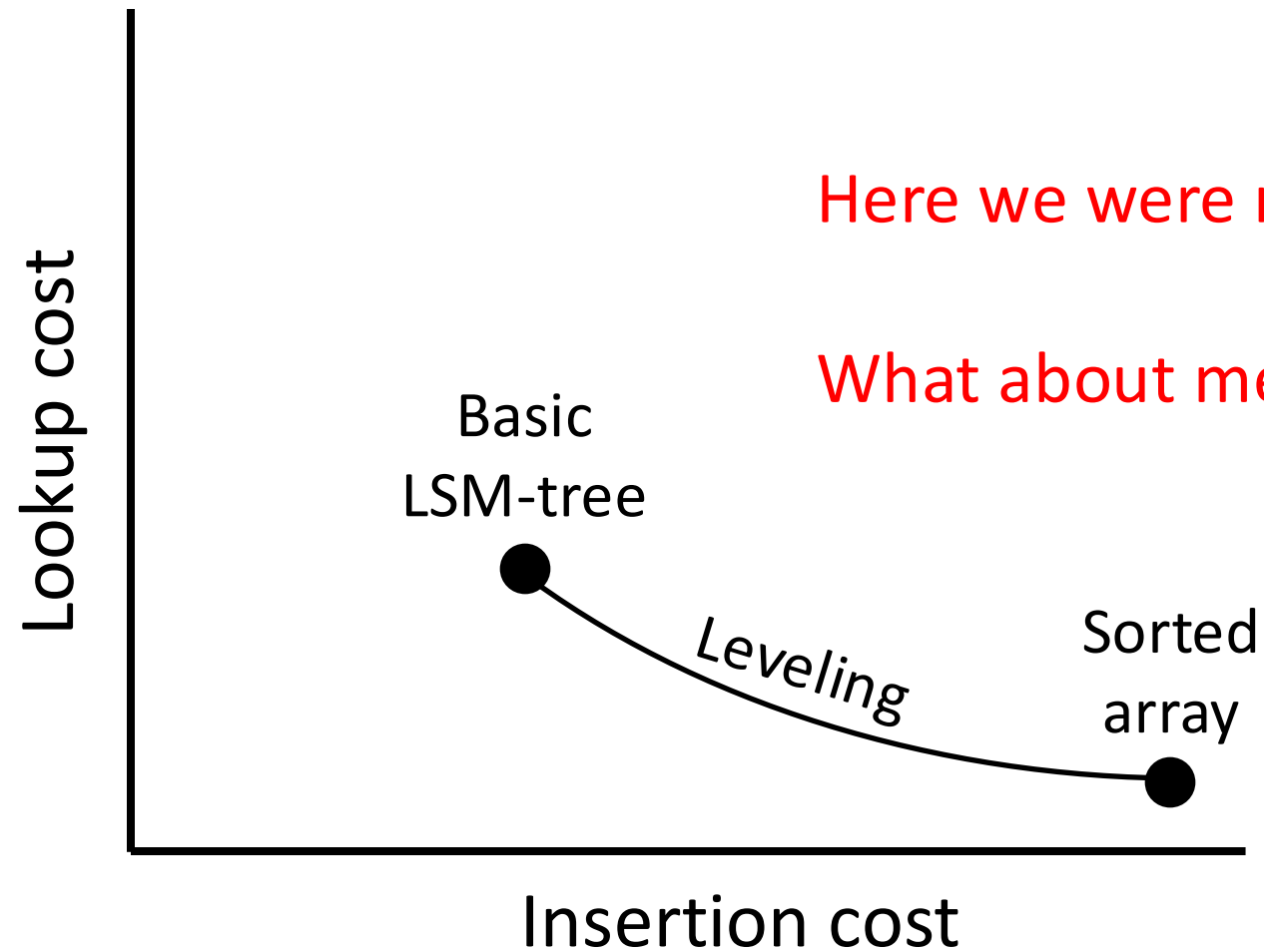
Lookup cost becomes:
 $O(1)$

Insert cost becomes:
 $O(N/B)$

The LSM-tree becomes a sorted array!

Results Catalogue – with fence pointers

	Lookup cost	Insertion cost
Sorted array	$O(1)$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N))$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree	$O(\log_T(N))$	$O(T/B \cdot \log_T(N))$
Tiered LSM-tree		



Here we were merging eagerly.

What about merging lazily?

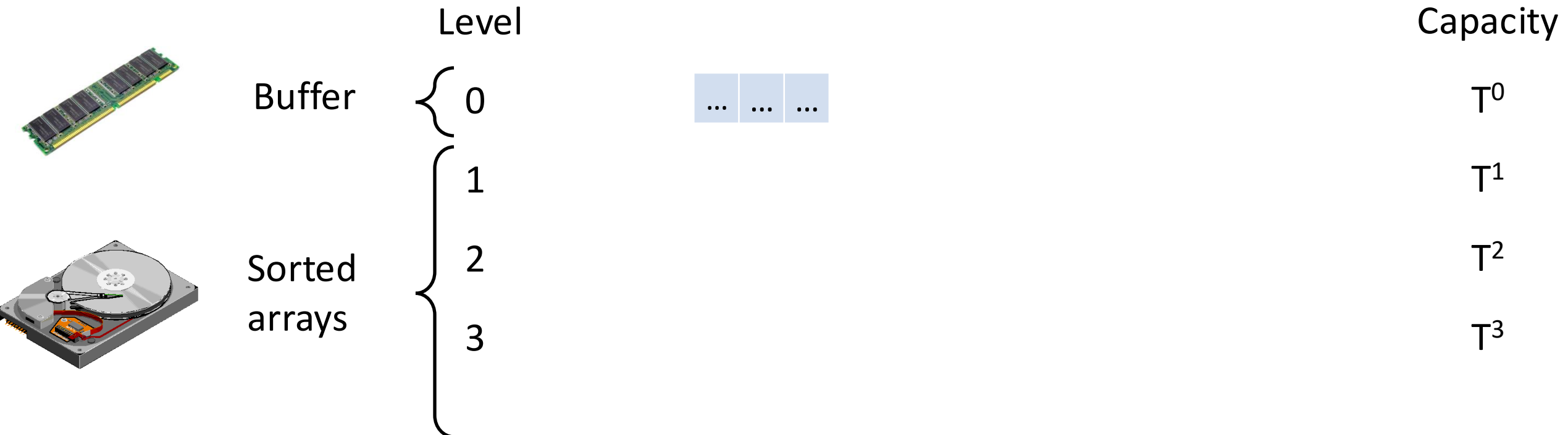
Tiered LSM-tree

 Lookup cost

 Insertion cost

Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.
Do not merge within a level.

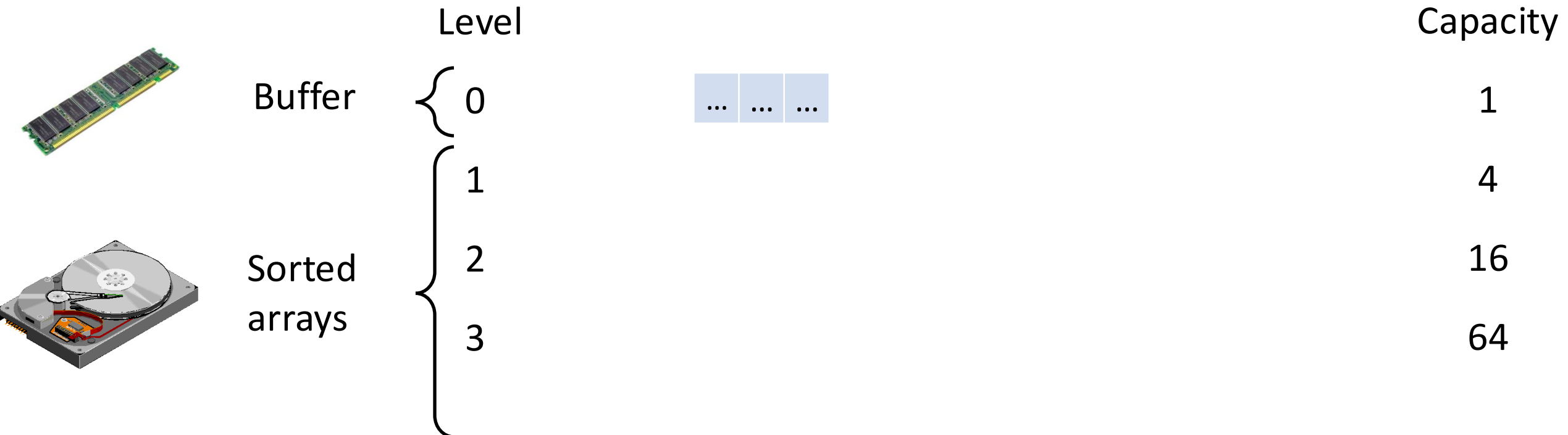


Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.

Do not merge within a level.

E.g. size ratio of 4

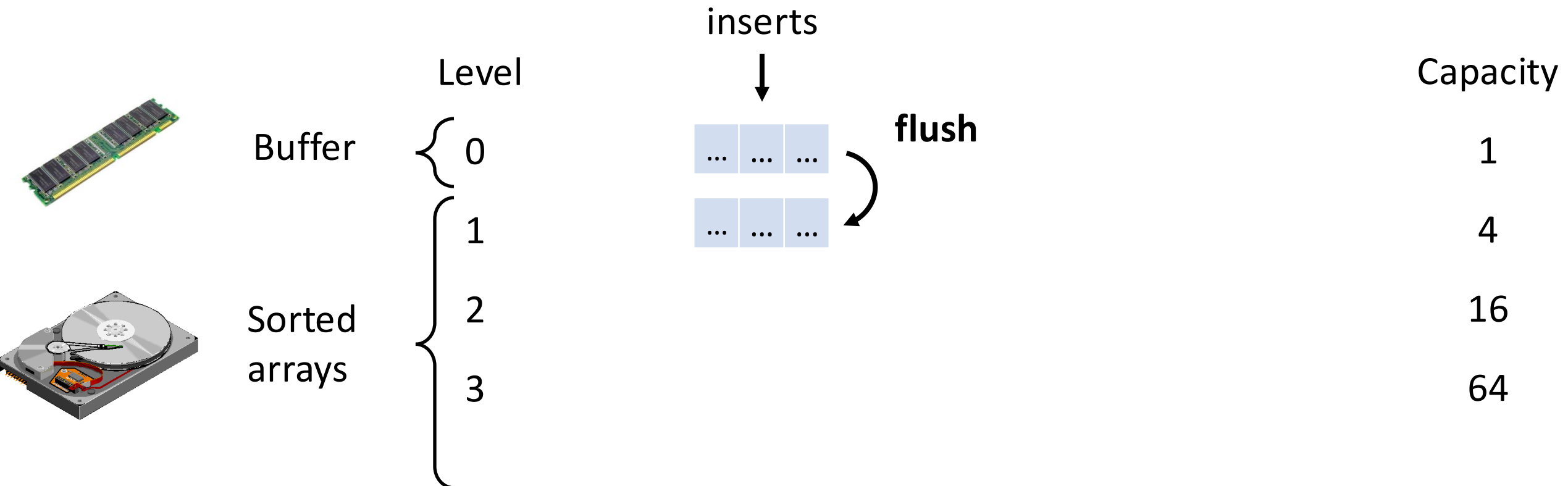


Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.

Do not merge within a level.

E.g. size ratio of 4

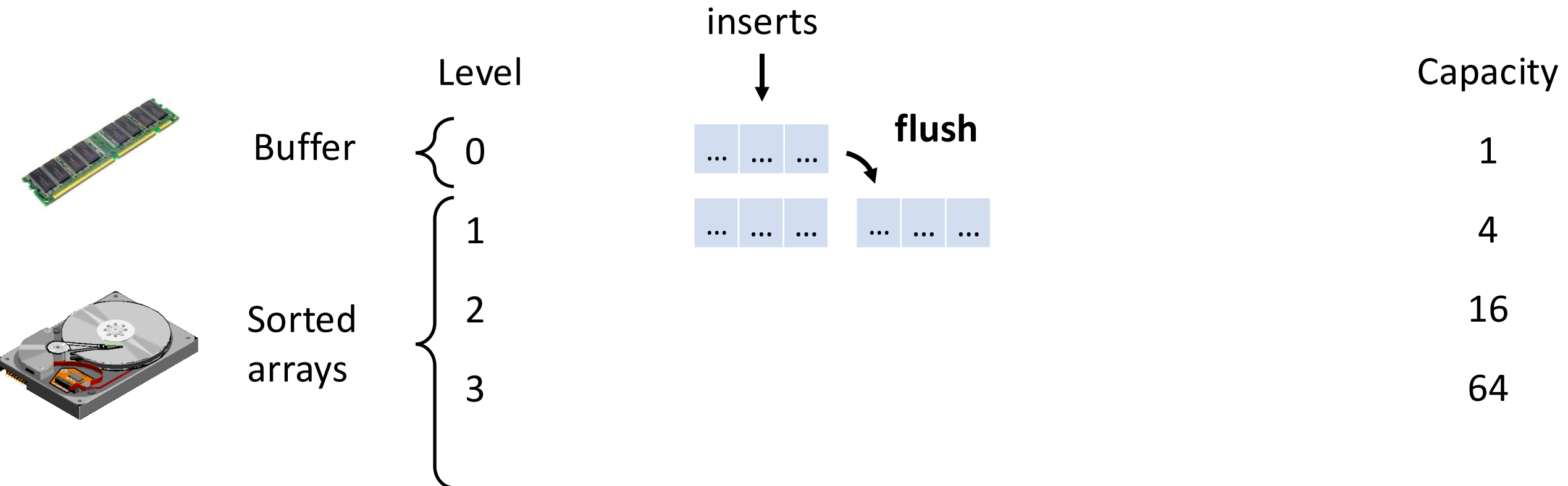


Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.

Do not merge within a level.

E.g. size ratio of 4

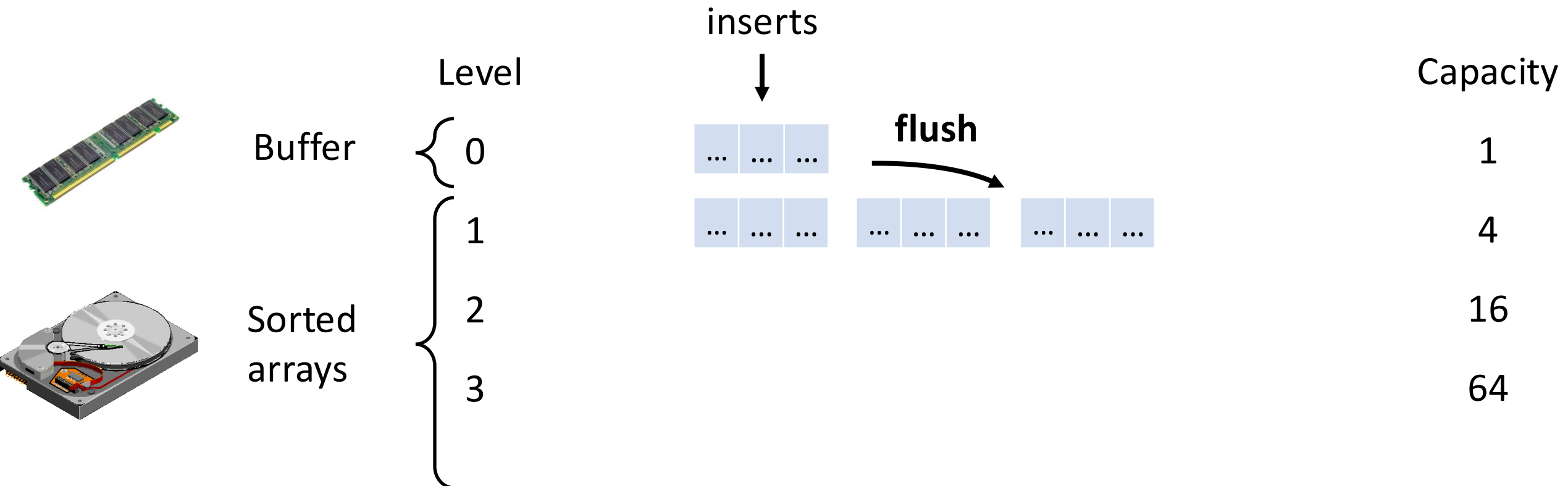


Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.

Do not merge within a level.

E.g. size ratio of 4

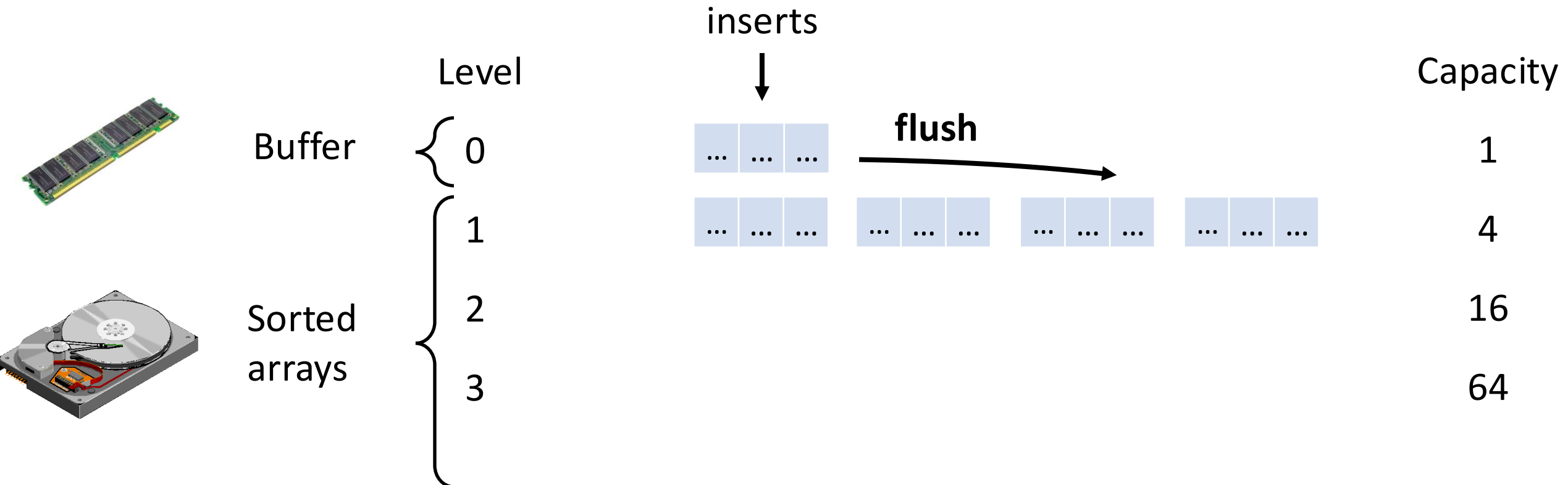


Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.

Do not merge within a level.

E.g. size ratio of 4

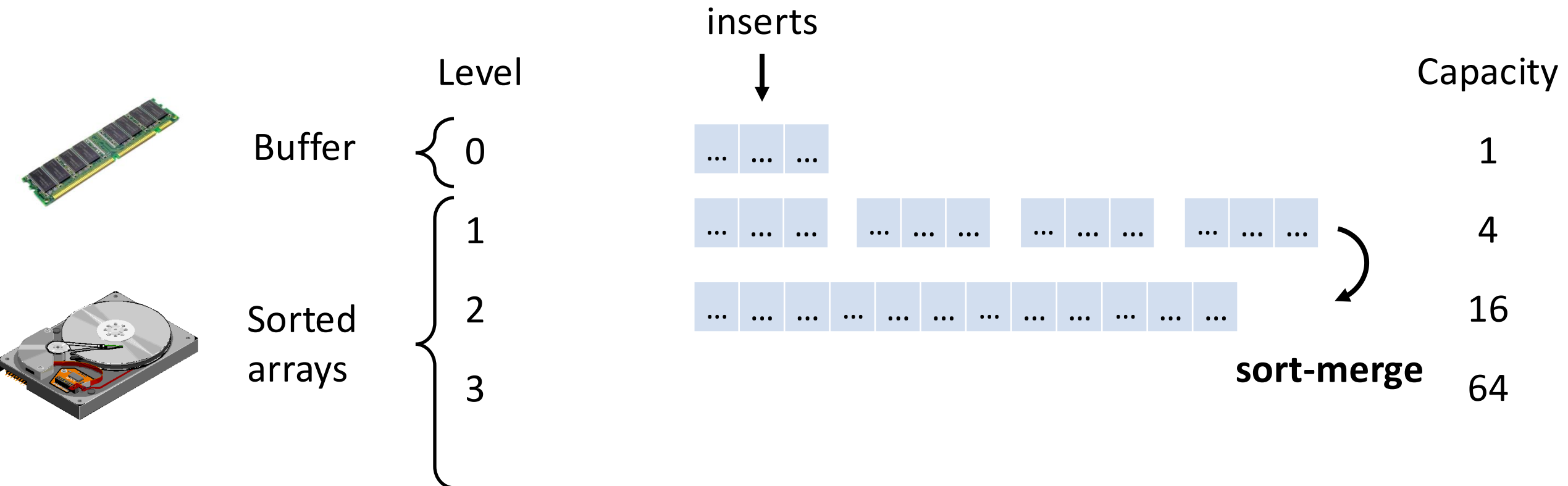


Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.

Do not merge within a level.

E.g. size ratio of 4

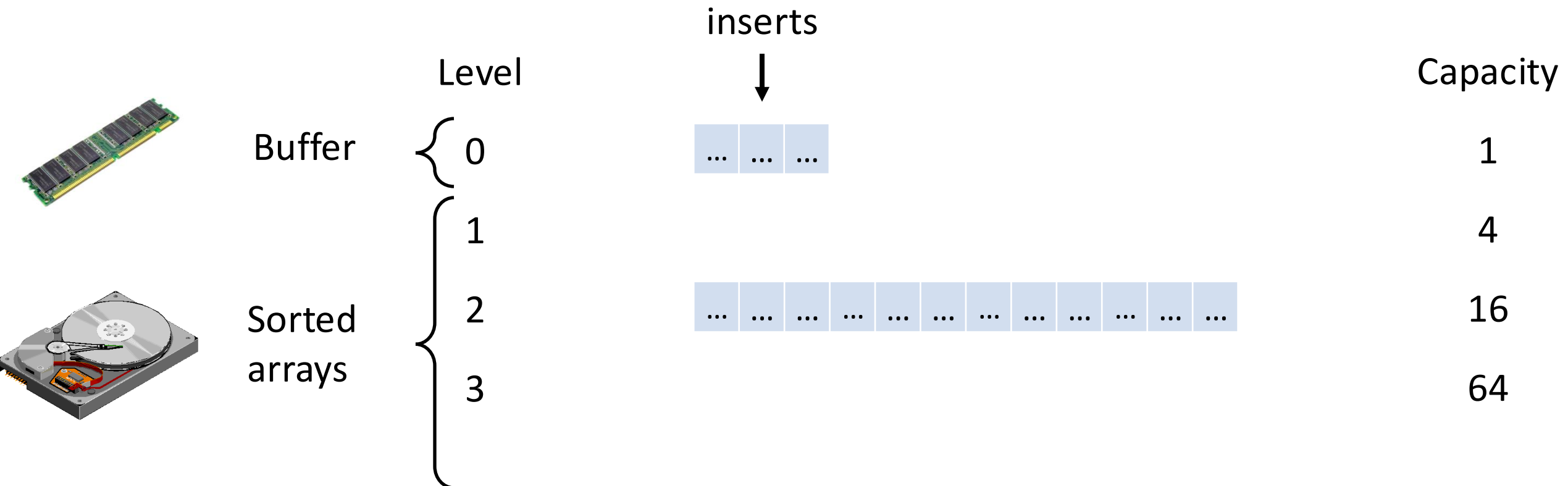


Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.

Do not merge within a level.

E.g. size ratio of 4



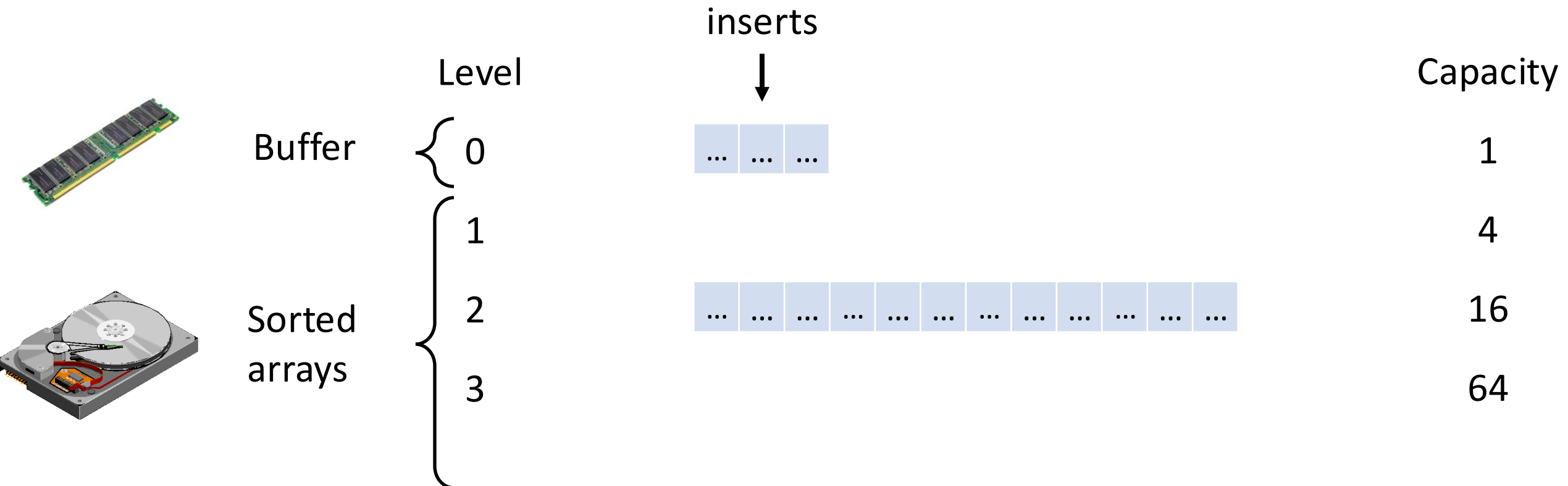
Tiered LSM-tree

Lookup cost?

$$O(T \cdot \log_T(N))$$

Insertion cost?

$$O\left(\frac{1}{B} \cdot \log_T(N)\right)$$



Tiered LSM-tree

↑ Lookup cost?
 $O(T \cdot \log_T(N))$

Insertion cost?
 $O\left(\frac{1}{B} \cdot \log_T(N)\right)$ ↓

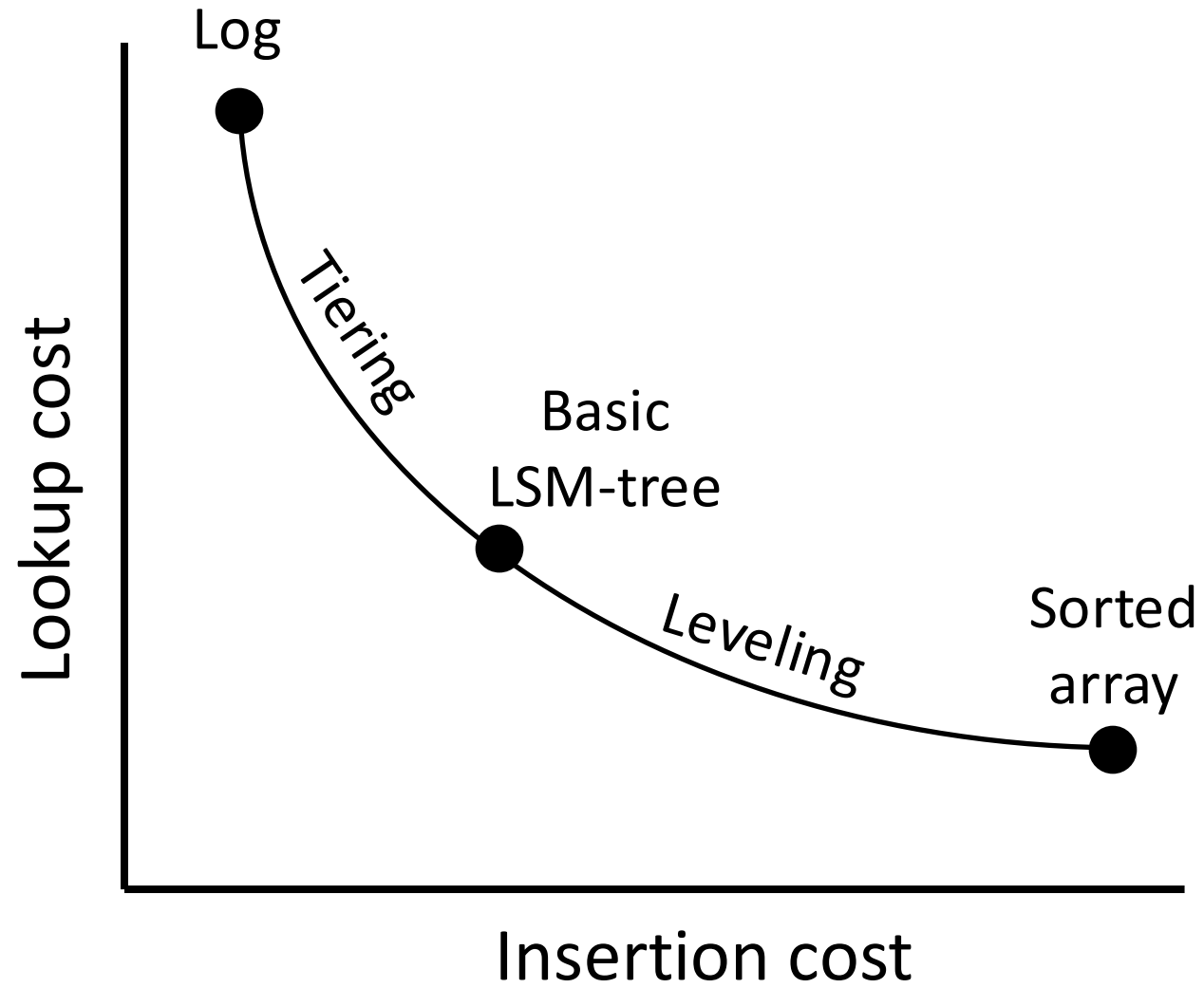
What happens as we increase the size ratio T ?

What happens when size ratio T is set to be N ?

Lookup cost becomes:
 $O(N)$

Insert cost becomes:
 $O(1/B)$

The tiered LSM-tree becomes a log!



Results Catalogue – with fence pointers

	Lookup cost	Insertion cost
Sorted array	$O(1)$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N))$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree	$O(\log_T(N))$	$O(T/B \cdot \log_T(N))$
Tiered LSM-tree	$O(T \cdot \log_T(N))$	$O(1/B \cdot \log_T(N))$

Results Catalogue – with fence pointers

Quick sanity check:

suppose

$$N = 2^{32}$$

and

$$B = 2^{10}$$

and

$$T = 2^2$$

	Lookup cost	Insertion cost
Sorted array	$O(1)$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N))$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree	$O(\log_T(N))$	$O(T/B \cdot \log_T(N))$
Tiered LSM-tree	$O(T \cdot \log_T(N))$	$O(1/B \cdot \log_T(N))$

Results Catalogue – with fence pointers

Quick sanity check:

suppose

$$N = 2^{32}$$

and

$$B = 2^{10}$$

and

$$T = 2^2$$

	Lookup cost	Insertion cost
Sorted array	$2^0=1$	$2^{31}=2B$
Log	$2^{32}=4B$	$2^{-10}=0.001$
B-tree	$2^2=4$	$2^2=4$
Basic LSM-tree	$2^5=32$	$2^{-5}=0.031$
Leveled LSM-tree	$2^4=16$	$2^{-4}=0.063$
Tiered LSM-tree	$2^6=64$	$2^{-6}=0.016$

Results Catalogue – with fence pointers

Quick sanity check:

suppose

$$N = 2^{32}$$

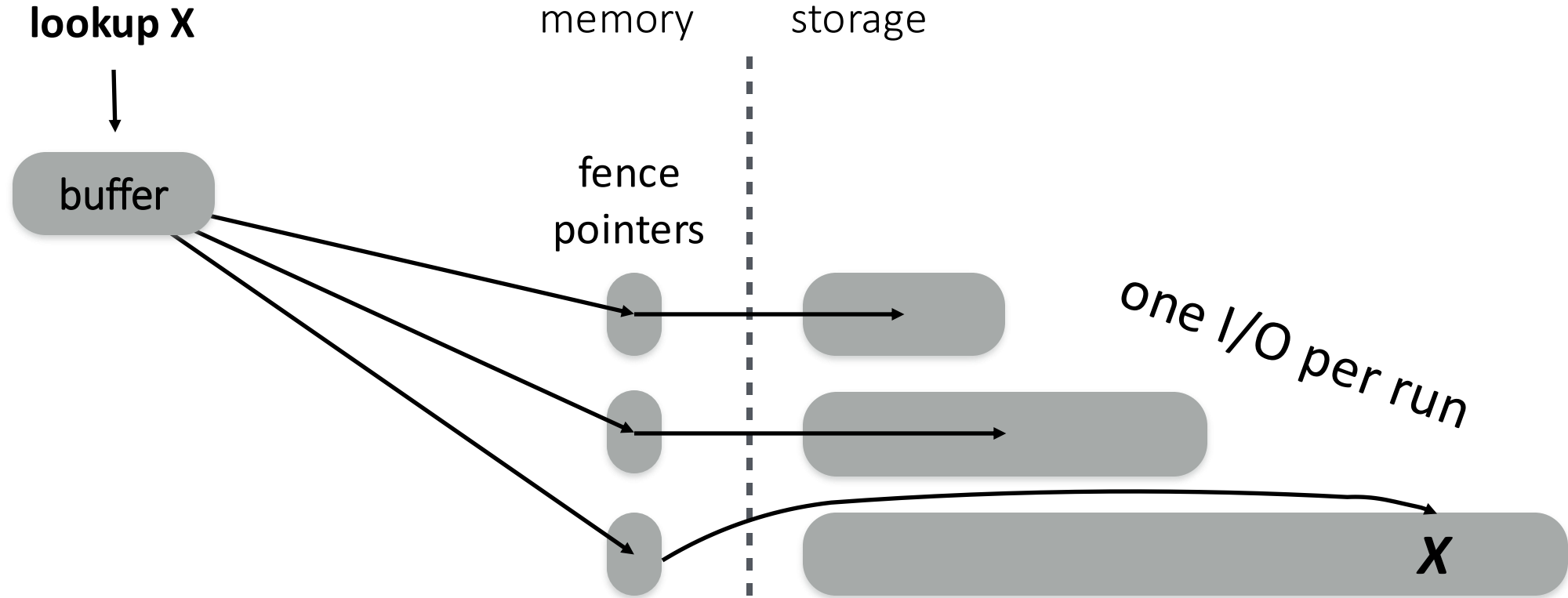
and

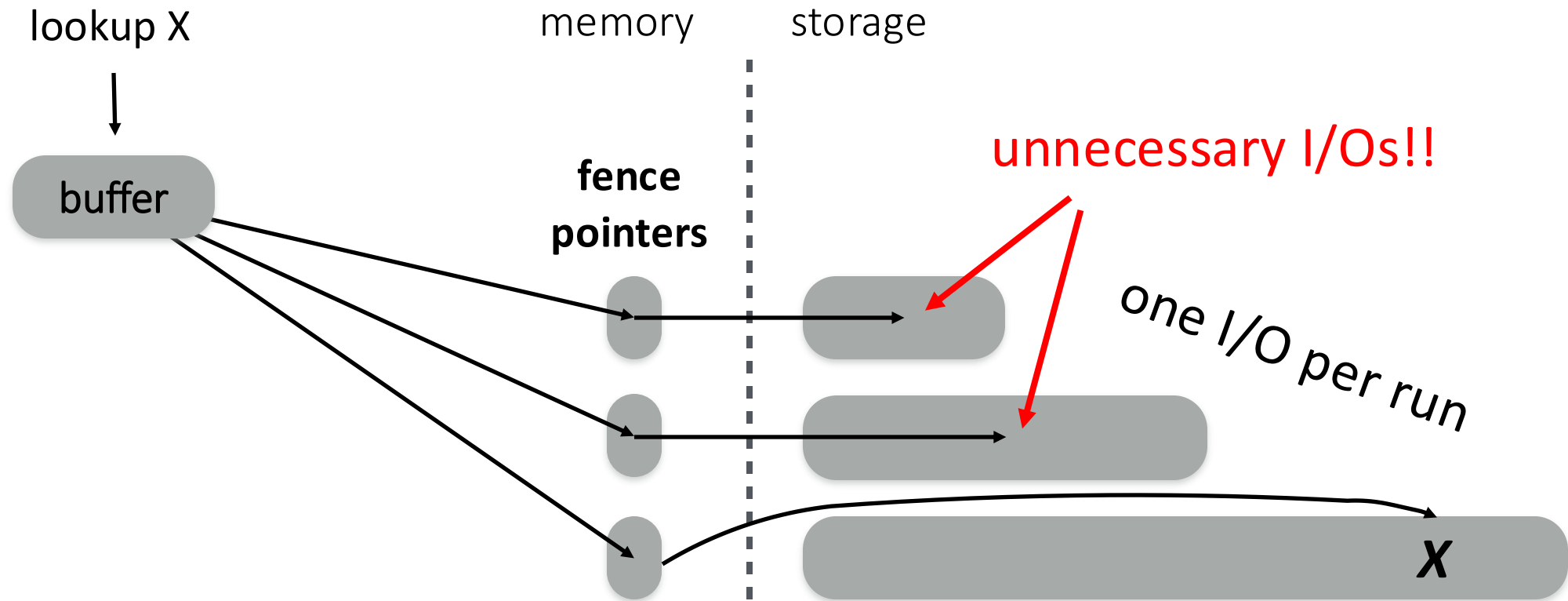
$$B = 2^{10}$$

and

$$T = 10$$

	Lookup cost	Insertion cost
Sorted array	$2^0=1$	$2^{31}=2B$
Log	$2^{32}=4B$	$2^{-10}=0.001$
B-tree	$2^2=4$	$2^2=4$
Basic LSM-tree	$2^5=32$	$2^{-5}=0.031$
Leveled LSM-tree	$\log_{10}(2^{32})=9.6$	$10 \cdot 2^{-10} \cdot \log_{10}(2^{32}) = 0.09$
Tiered LSM-tree	$10 \cdot \log_{10}(2^{32})=96$	$2^{-10} \cdot \log_{10}(2^{32}) = 0.009$





How to avoid them?

An **oracle** that helps us to skip them!

Bloom filters

Answer **set-membership** queries

Small size, typically stored in **memory**



May return **false positives**

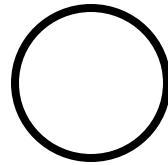
Bloom filters

k hash functions

$h_1(\blacksquare)$

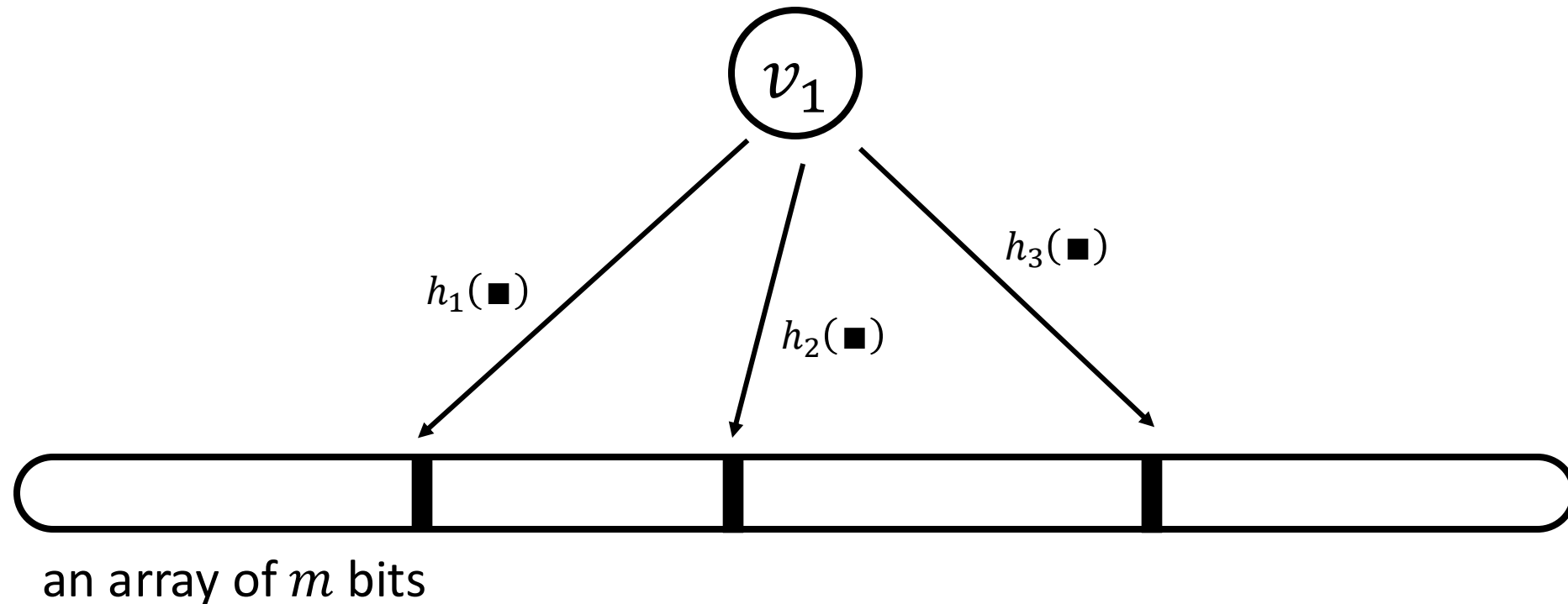
$h_2(\blacksquare)$

$h_3(\blacksquare)$

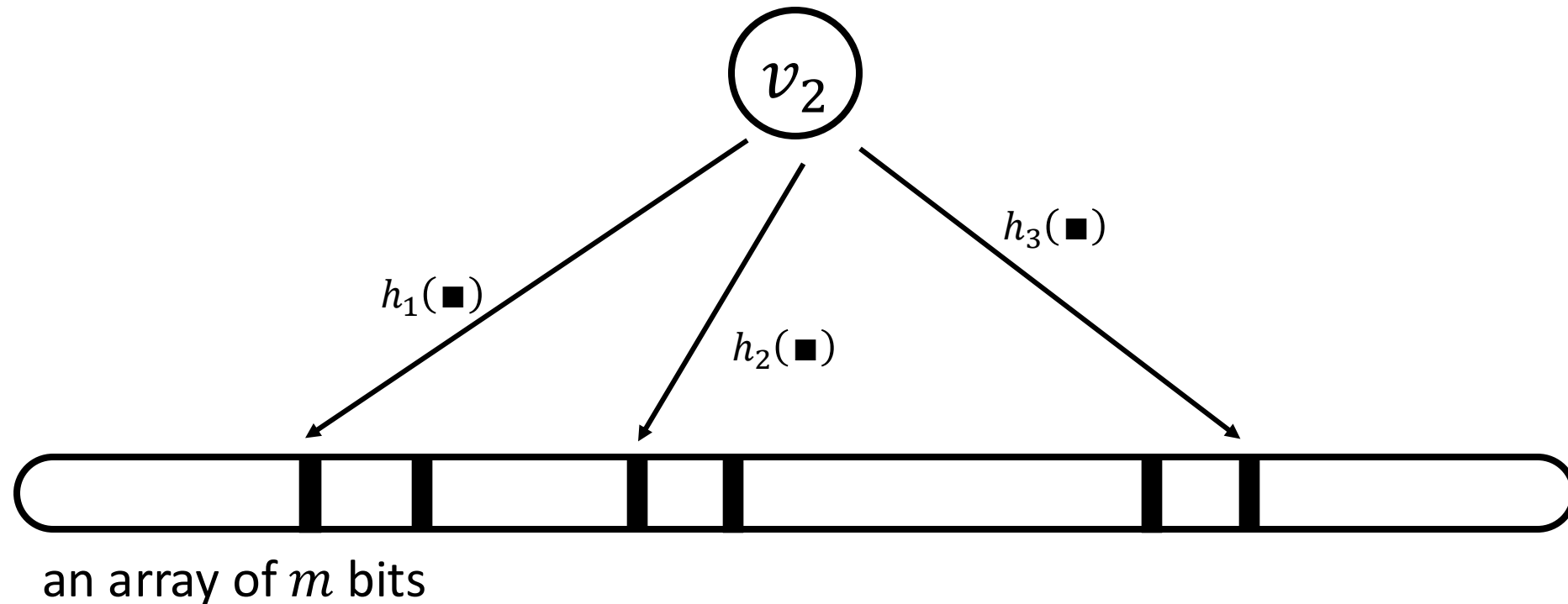


an array of m bits

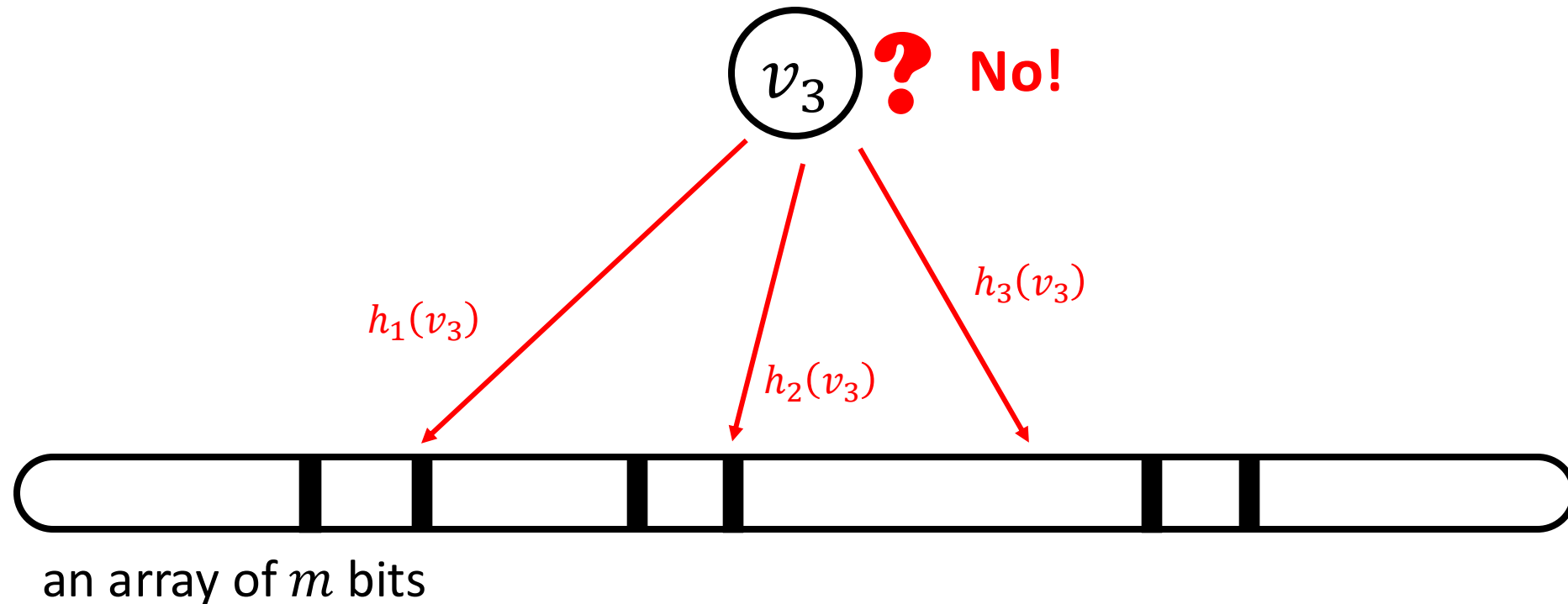
Bloom filters – insert v_1



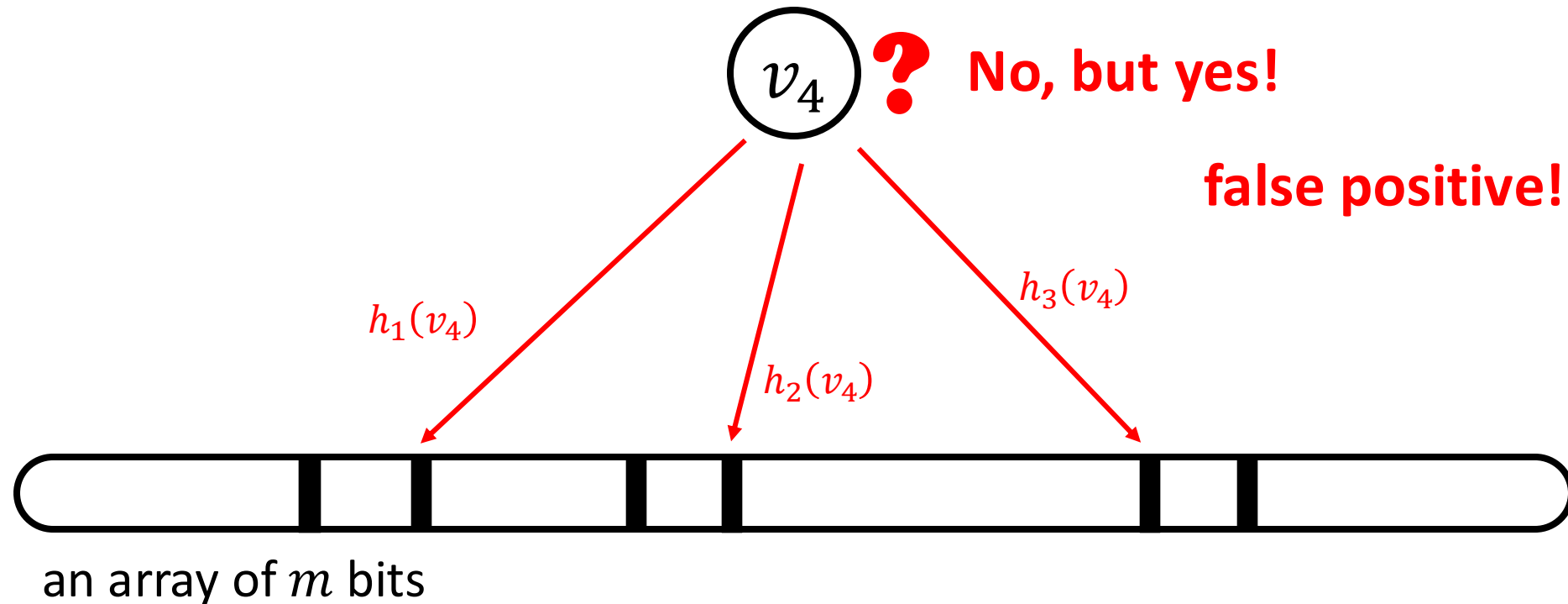
Bloom filters – insert v_2



Bloom filters – query v_3



Bloom filters – query v_4



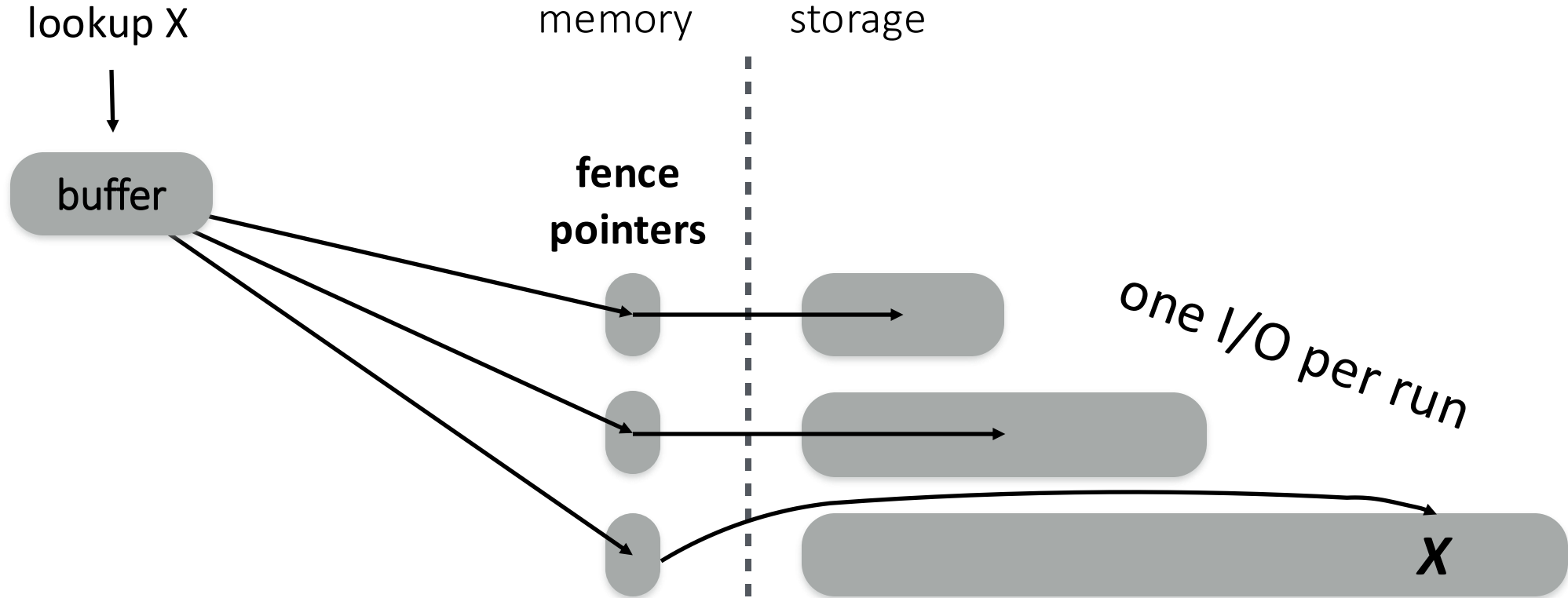
false positive rate: $f = e^{-\frac{m}{n} \cdot (\ln(2))^2}$

sanity check: for $\frac{m}{n} = 10$, $f = 0.00819$

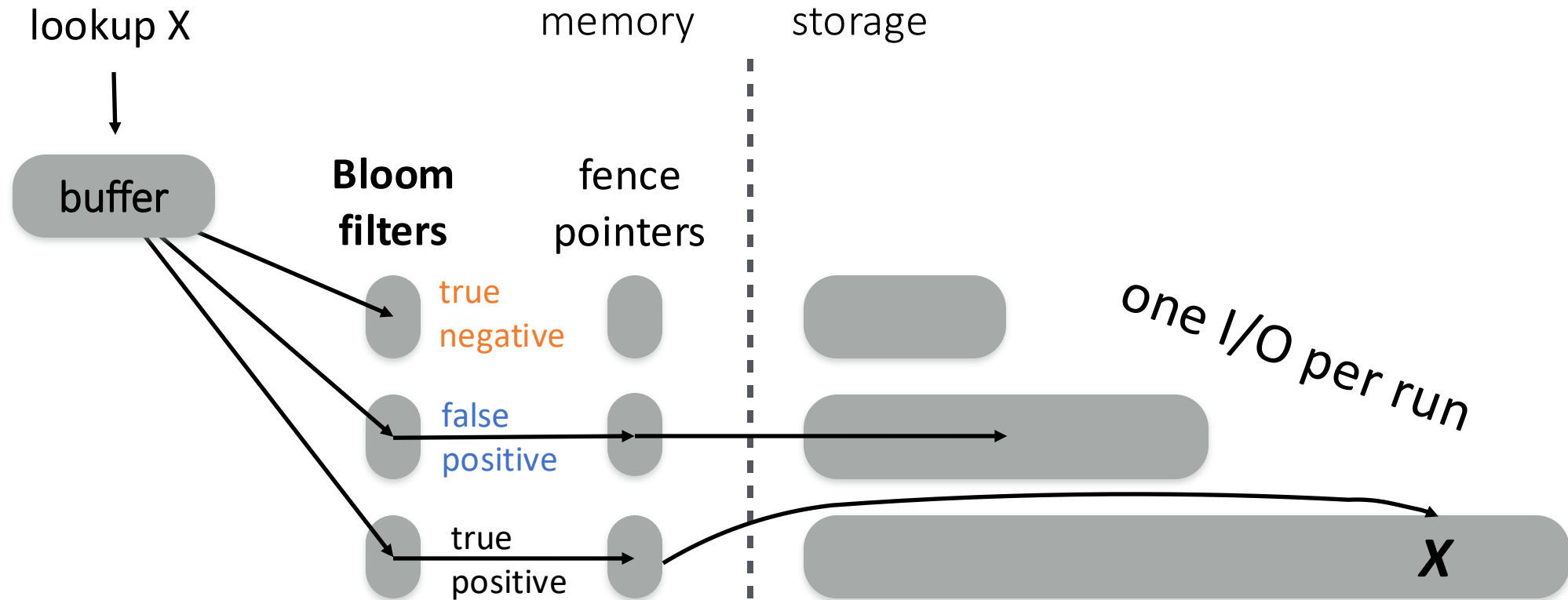
after inserting n elements

→ we have m/n bits per key

Augmenting the LSM design with Bloom filters



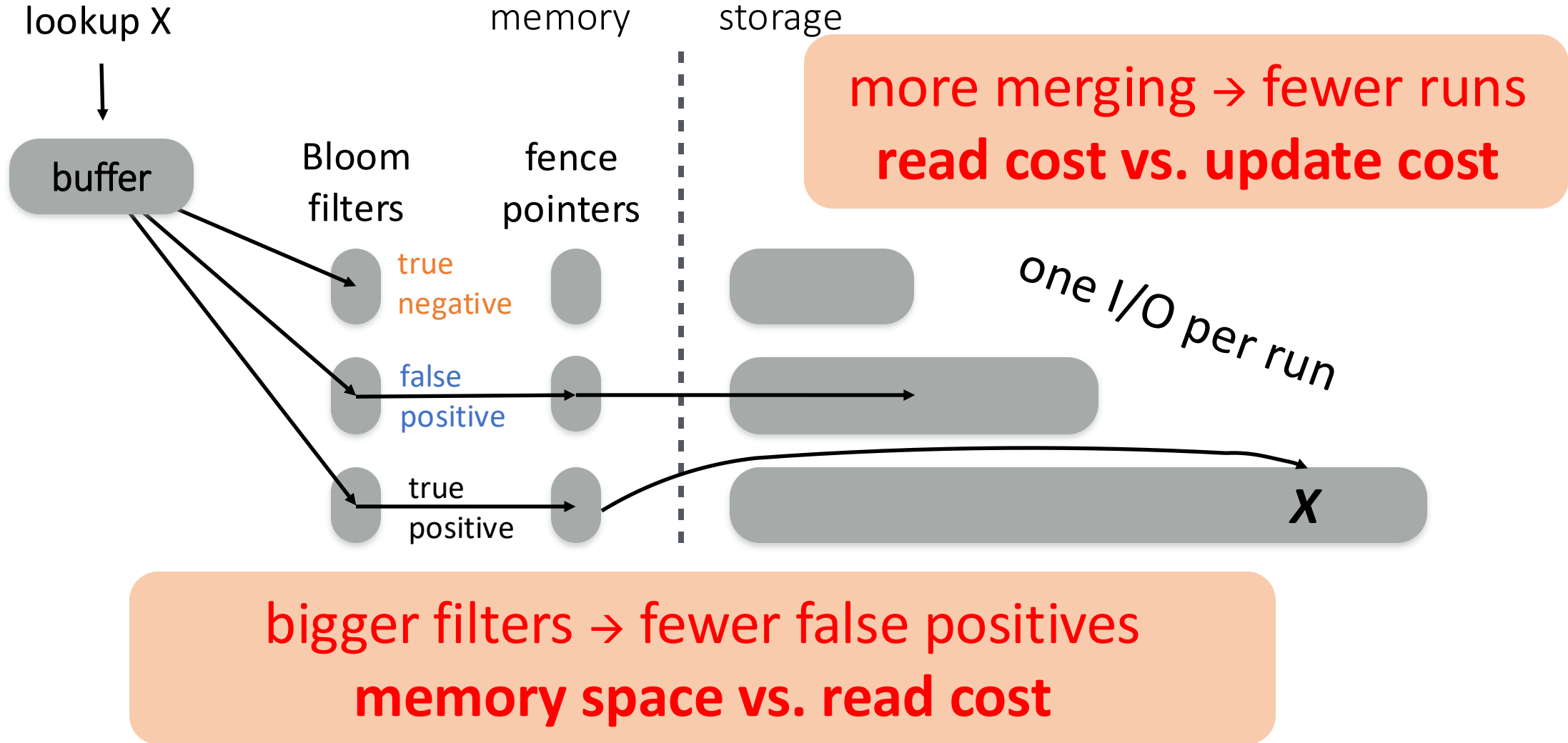
Augmenting the LSM design with Bloom filters



Empty Queries: only FPs

Non-Empty Queries: FPs and one I/O

performance & cost trade-offs



Results Catalogue – with fence pointers & BFs

Empty Queries

	Lookup cost	Insertion cost
Sorted array	$O(1)$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N))$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree	$O(f \cdot \log_T(N))$	$O(T/B \cdot \log_T(N))$
Tiered LSM-tree	$O(f \cdot T \cdot \log_T(N))$	$O(1/B \cdot \log_T(N))$

Results Catalogue – with fence pointers & BFs

Quick sanity check:

suppose

$$N = 2^{32}$$

and

$$B = 2^{10}$$

and

$$T = 10 \text{ and } m/n = 10$$

Empty Queries

	Lookup cost	Insertion cost
Sorted array	$2^0=1$	$2^{31}=2B$
Log	$2^{32}=4B$	$2^{-10}=0.001$
B-tree	$2^2=4$	$2^2=4$
Basic LSM-tree	$2^5=32$	$2^{-5}=0.031$
Leveled LSM-tree	$f \cdot \log_{10}(2^{32})=0.079$	$10 \cdot 2^{-10} \cdot \log_{10}(2^{32}) = 0.09$
Tiered LSM-tree	$f \cdot 10 \cdot \log_{10}(2^{32})=0.79$	$2^{-10} \cdot \log_{10}(2^{32}) = 0.009$

Results Catalogue – with fence pointers & BFs

Non-Empty Queries

	Lookup cost	Insertion cost
Sorted array	$O(1)$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N))$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree	$O(1 + f \cdot \log_T(N))$	$O(T/B \cdot \log_T(N))$
Tiered LSM-tree	$O(1 + f \cdot T \cdot \log_T(N))$	$O(1/B \cdot \log_T(N))$

Results Catalogue – with fence pointers & BFs

Quick sanity check:

suppose

$$N = 2^{32}$$

and

$$B = 2^{10}$$

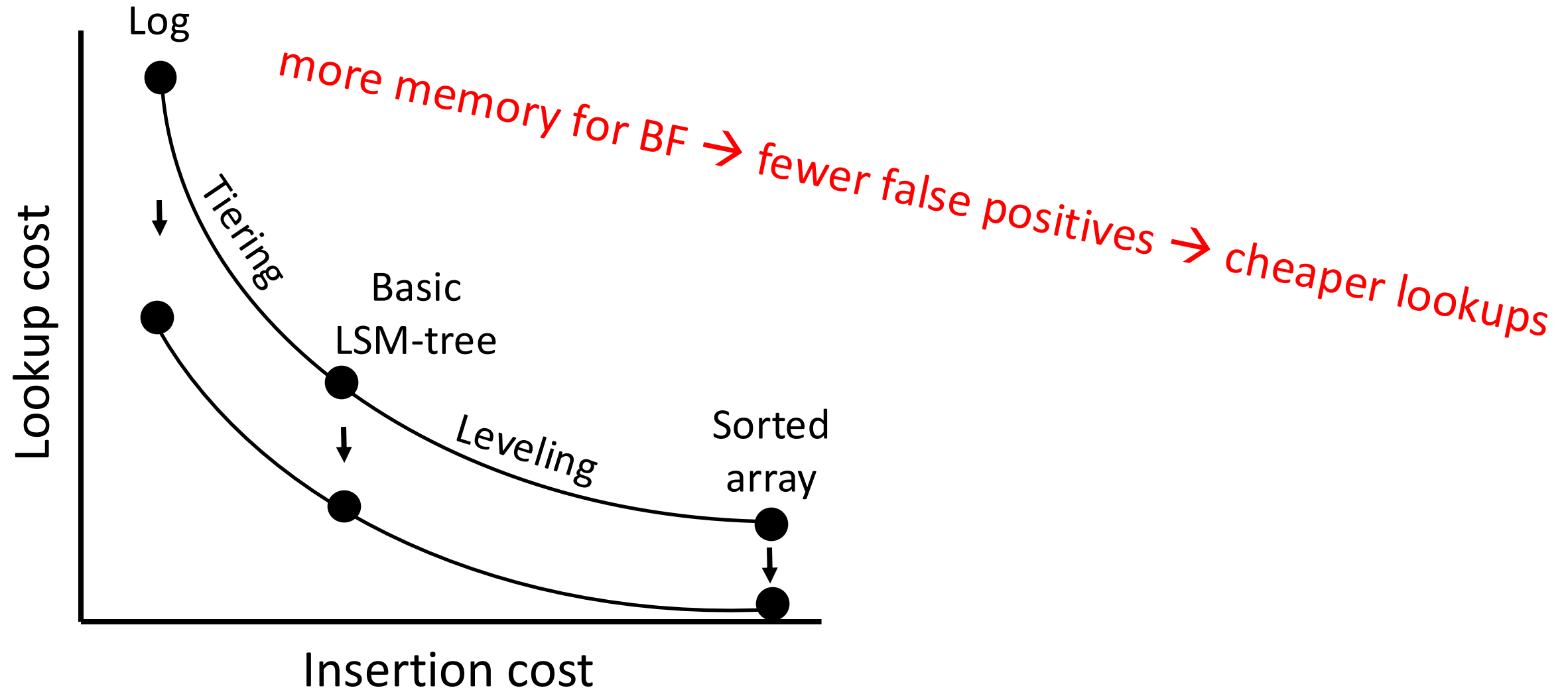
and

$$T = 10 \text{ and } m/n = 10$$

Non-Empty Queries

	Lookup cost	Insertion cost
Sorted array	$2^0=1$	$2^{31}=2B$
Log	$2^{32}=4B$	$2^{-10}=0.001$
B-tree	$2^2=4$	$2^2=4$
Basic LSM-tree	$2^5=32$	$2^{-5}=0.031$
Leveled LSM-tree	$1 + f \cdot \log_{10}(2^{32})=1.079$	$10 \cdot 2^{-10} \cdot \log_{10}(2^{32}) = 0.09$
Tiered LSM-tree	$1 + f \cdot 10 \cdot \log_{10}(2^{32})=1.79$	$2^{-10} \cdot \log_{10}(2^{32}) = 0.009$

Bloom Filters



Conclusions

Write-optimized

Highly tunable

Backbone of many modern systems

Trade-off between lookup and insert cost (tiering/leveling, size ratio)

Trade main memory for lookup cost (fence pointers, Bloom filters)

Thank you!