CS660: Intro to Database Systems

Class 10: Log-Structured-Merge Trees

Instructor: Manos Athanassoulis

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

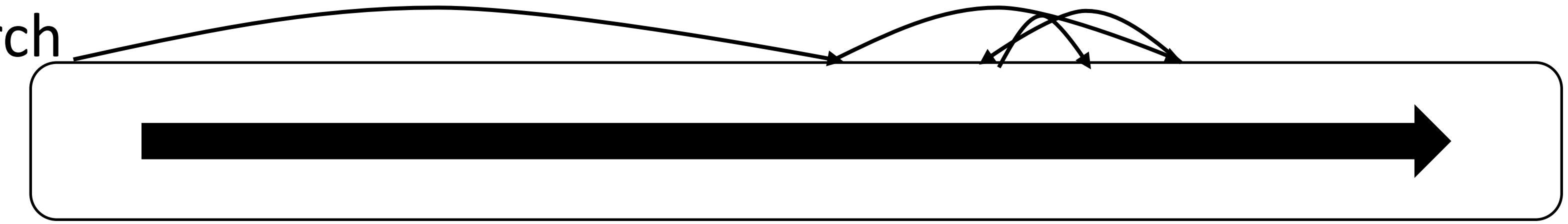
Reads vs Writes: The two extremes

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?

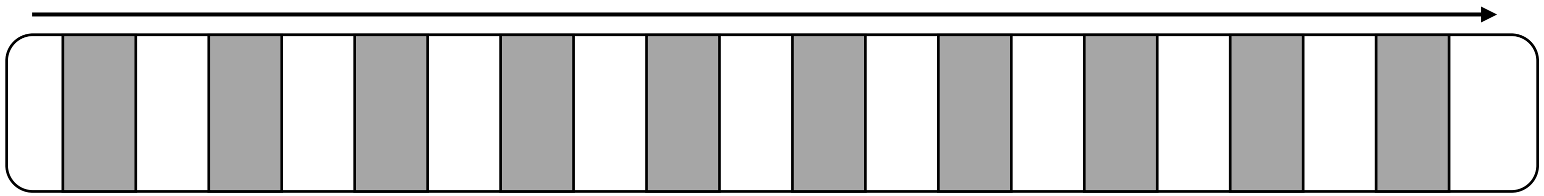
binary search



Sorted!

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

scan

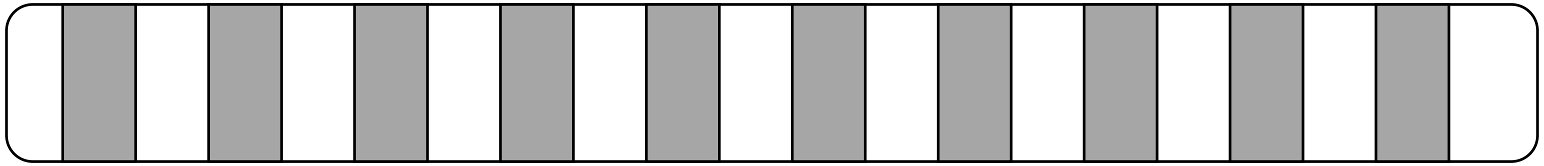


Append (log)

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?

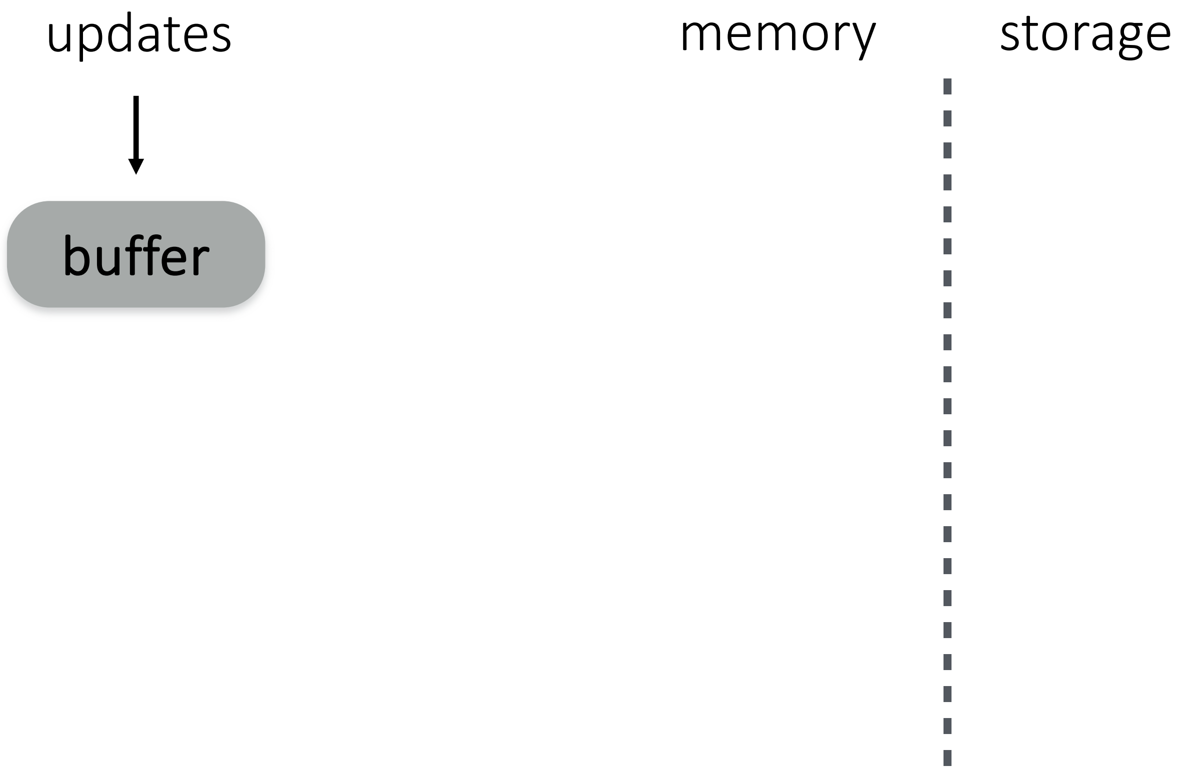
Keep the sorted file

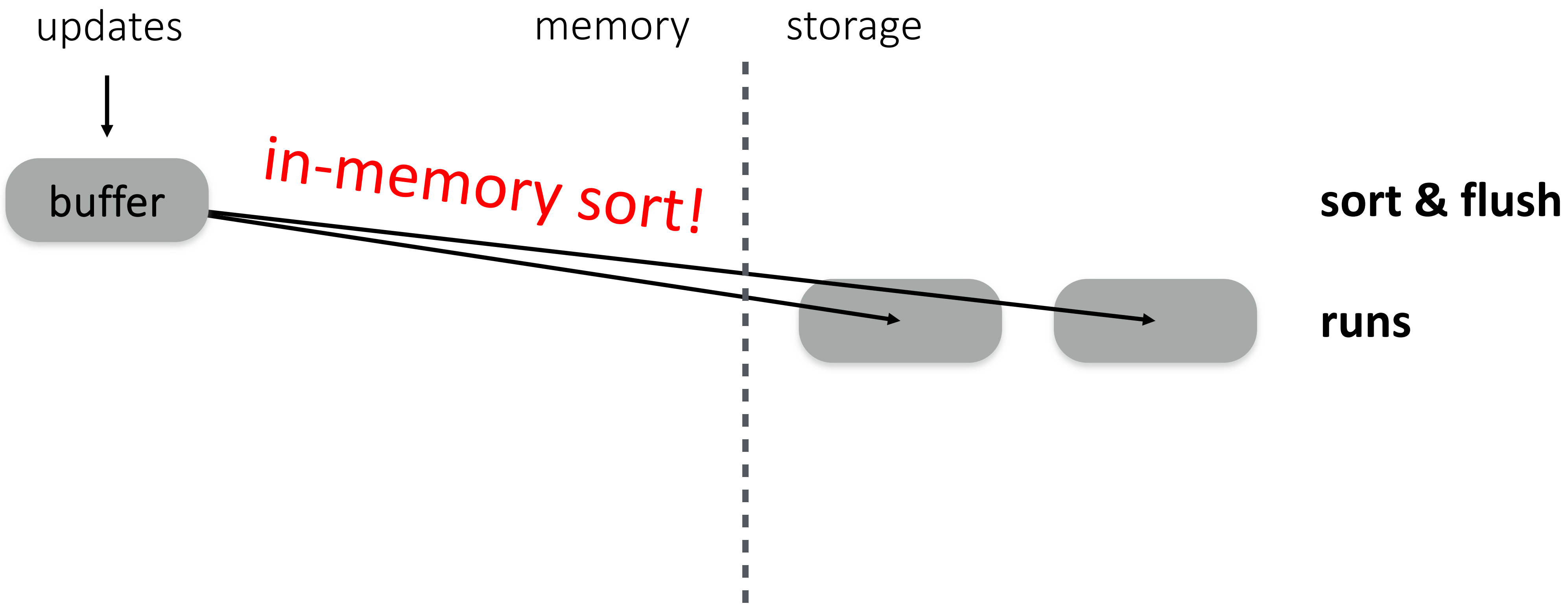
&

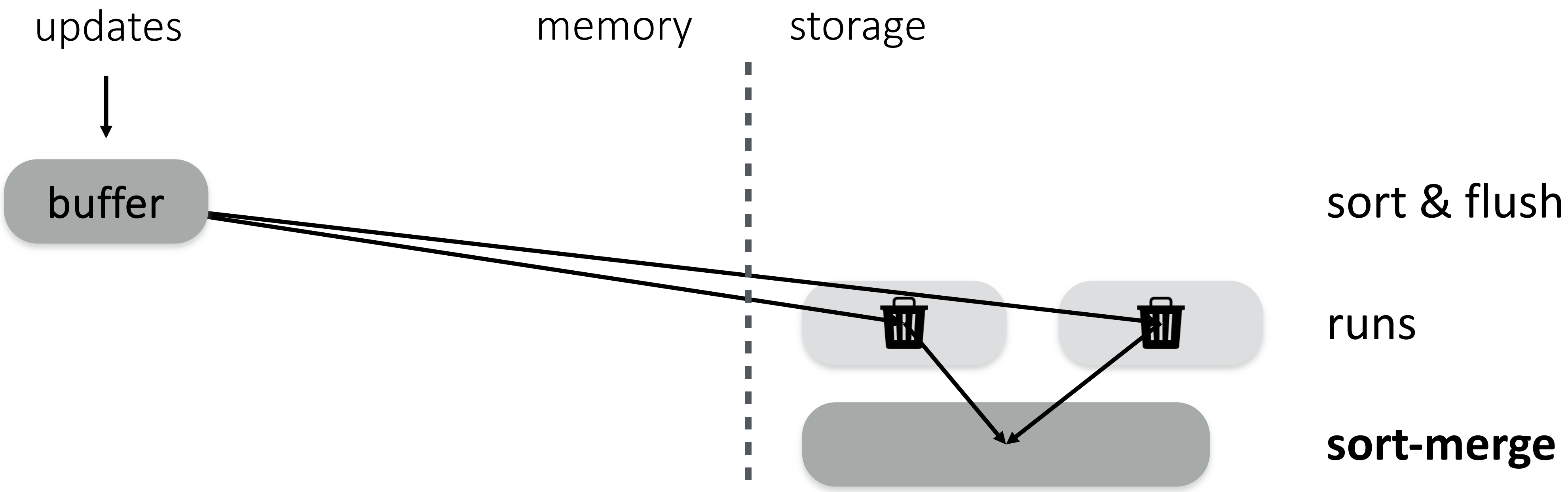
append to a new one

What to do with many sorted files?

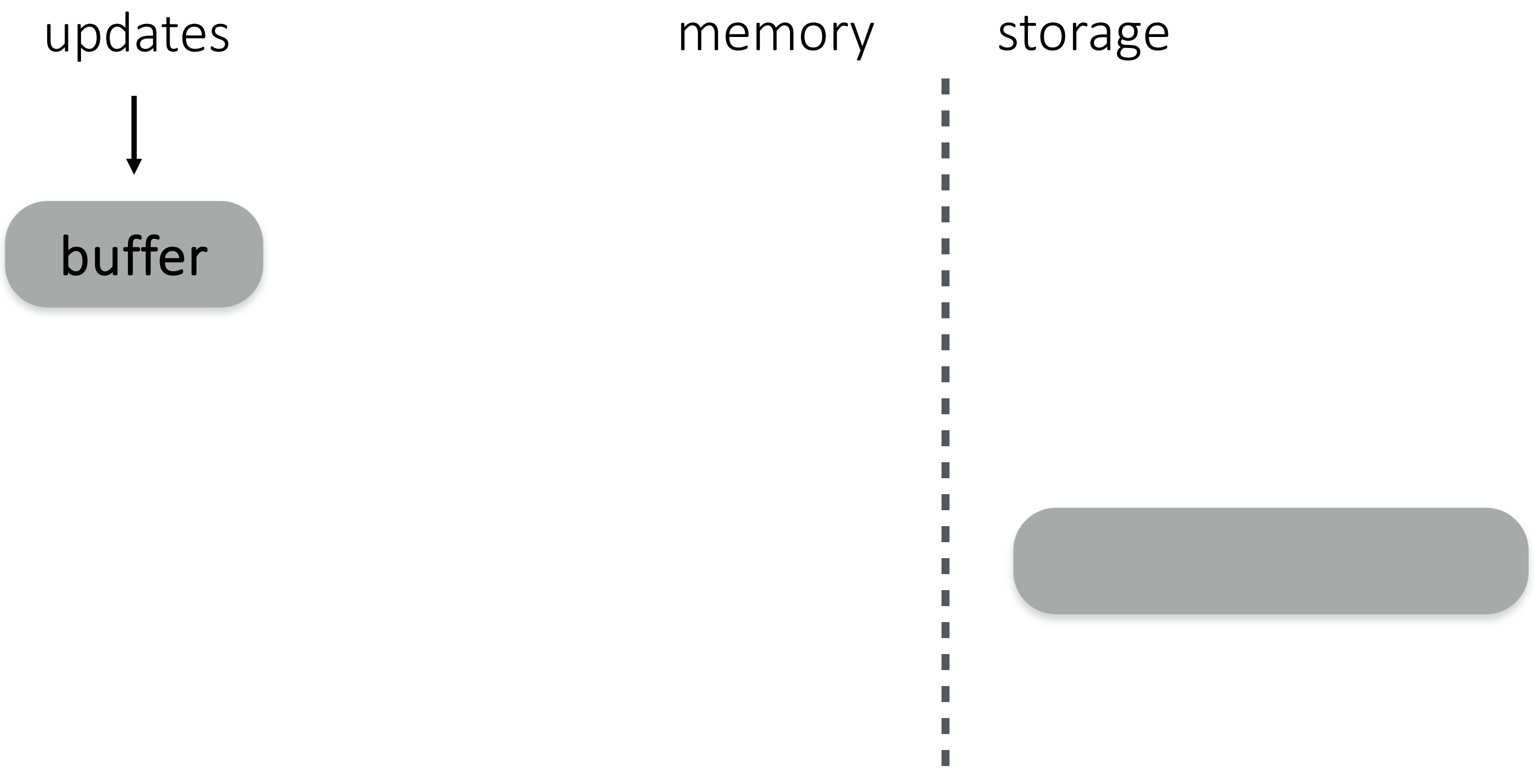
Merge them!


updates
buffer
memory
storage


updates
memory
storage
buffer
in-memory sort!
sort & flush
runs



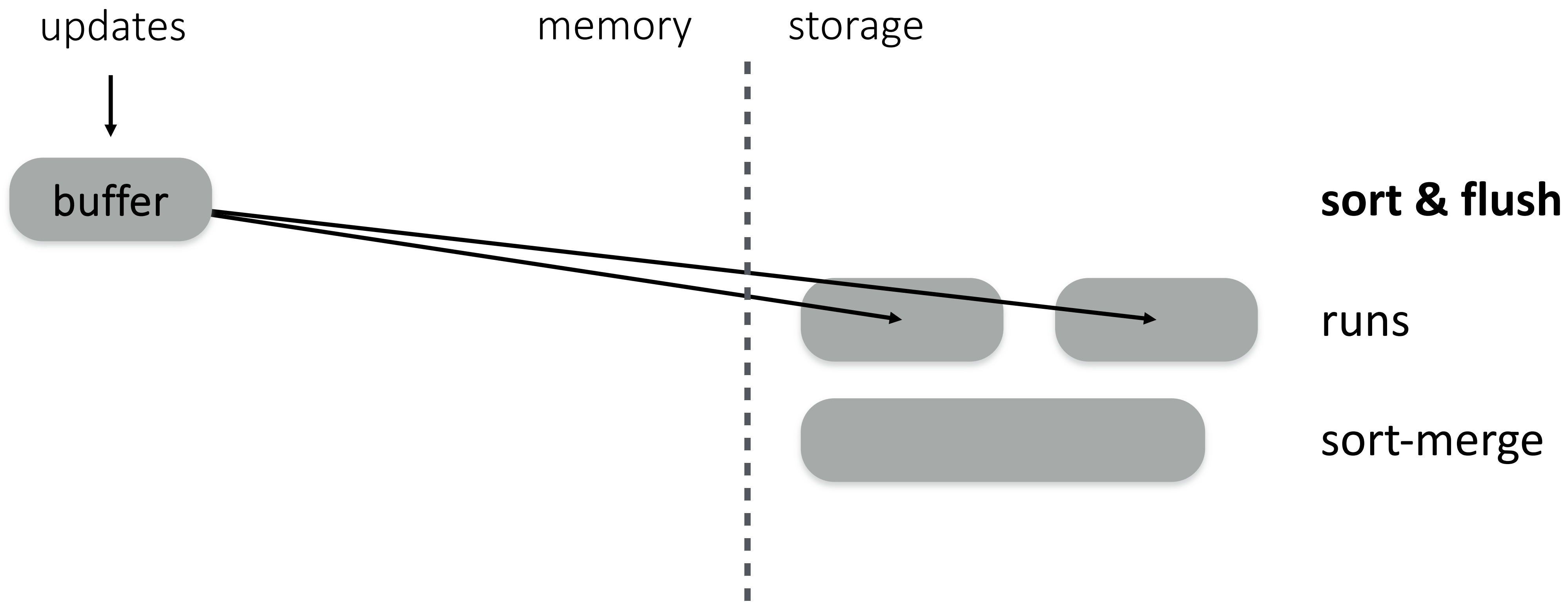
similar to merging for ext. sort



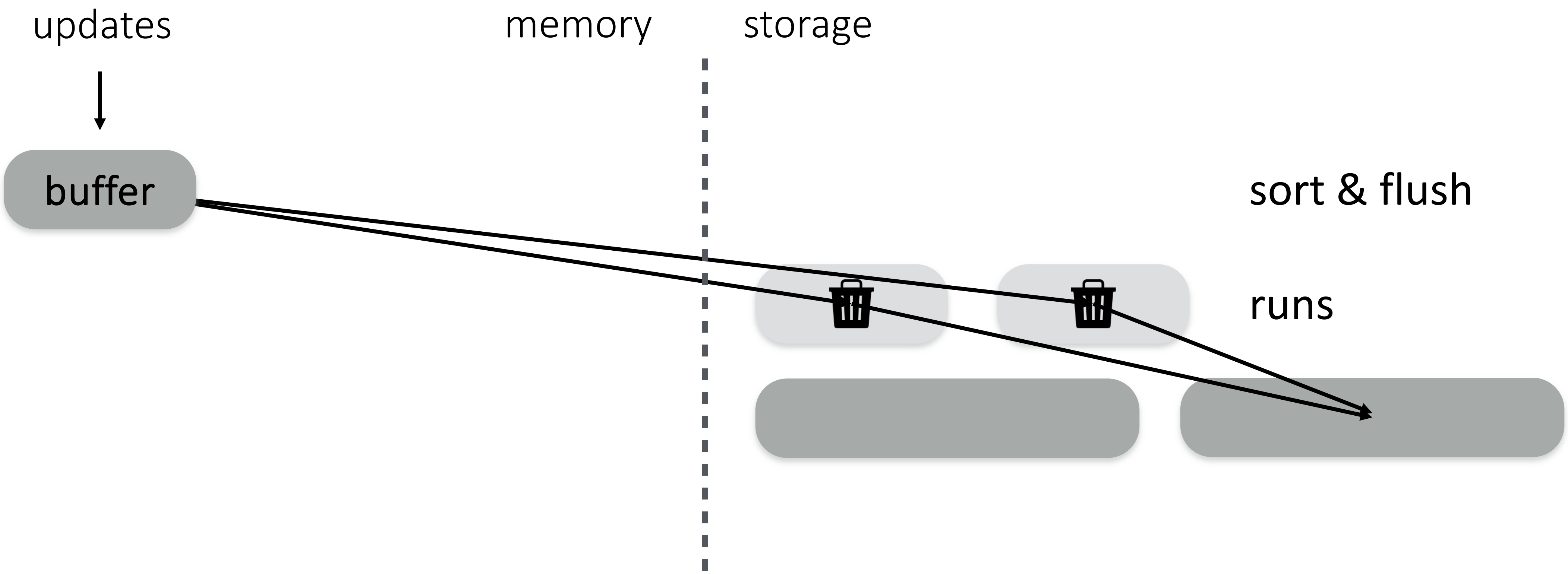
sort & flush

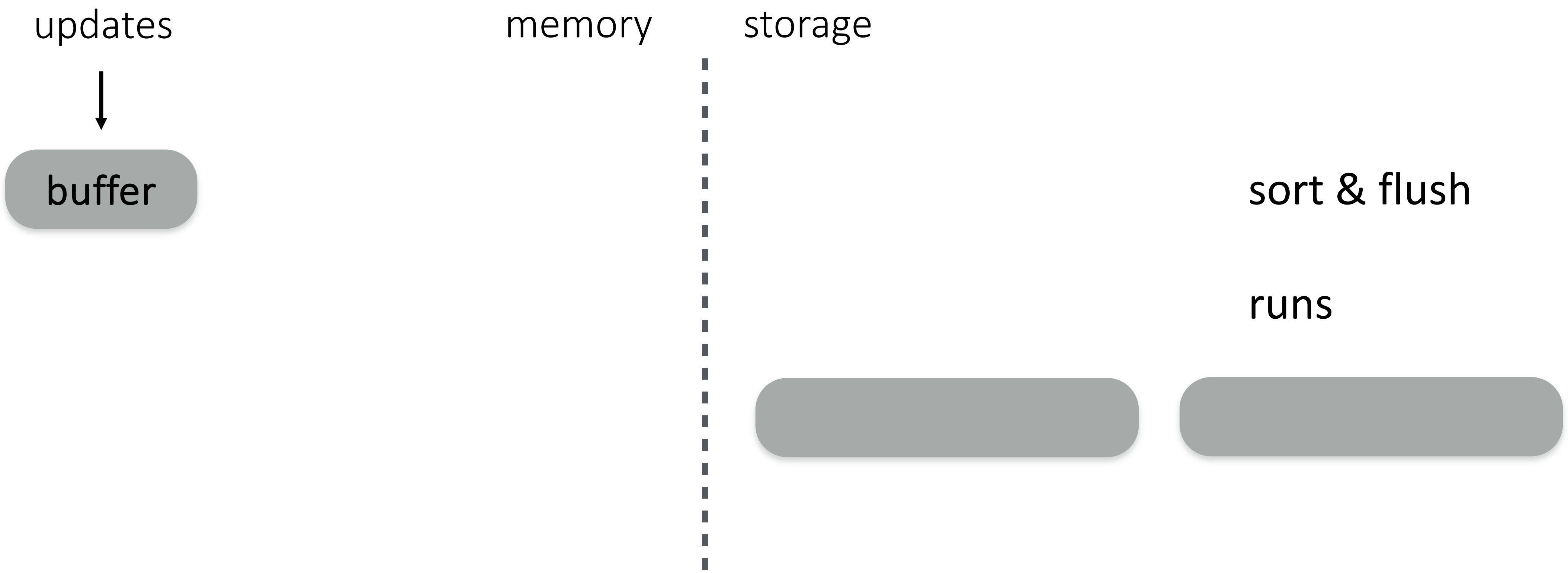
runs

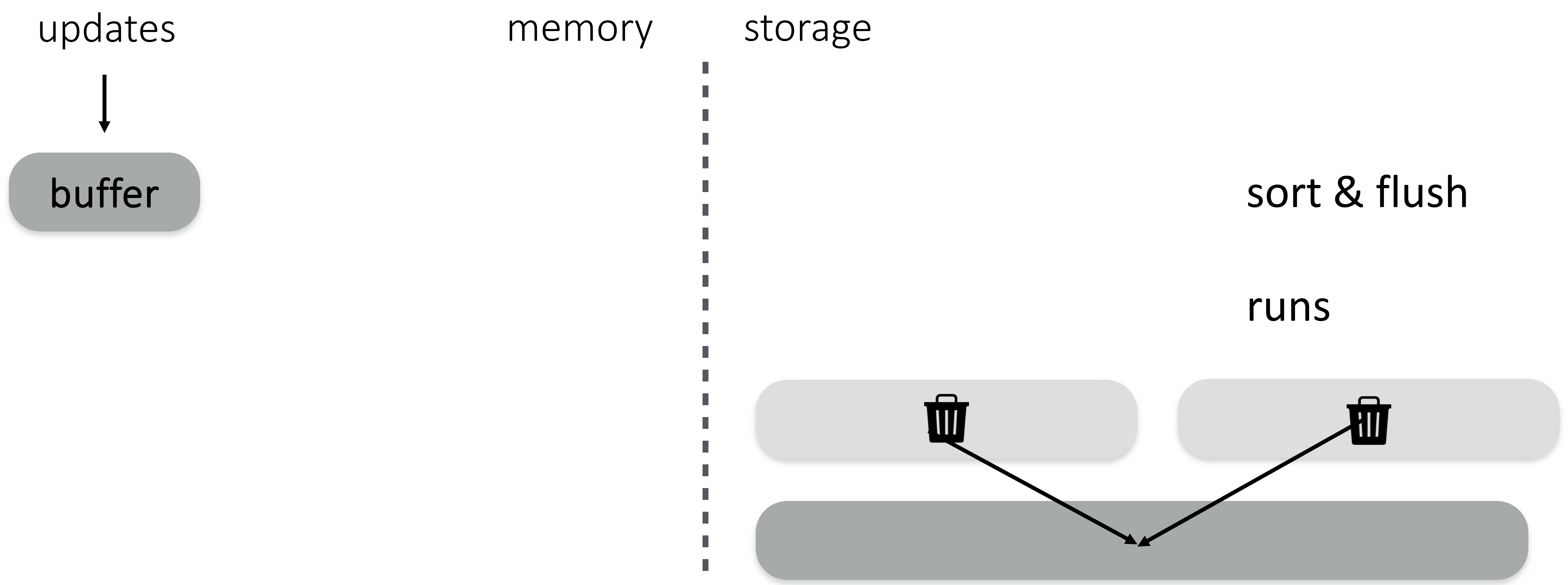
sort-merge

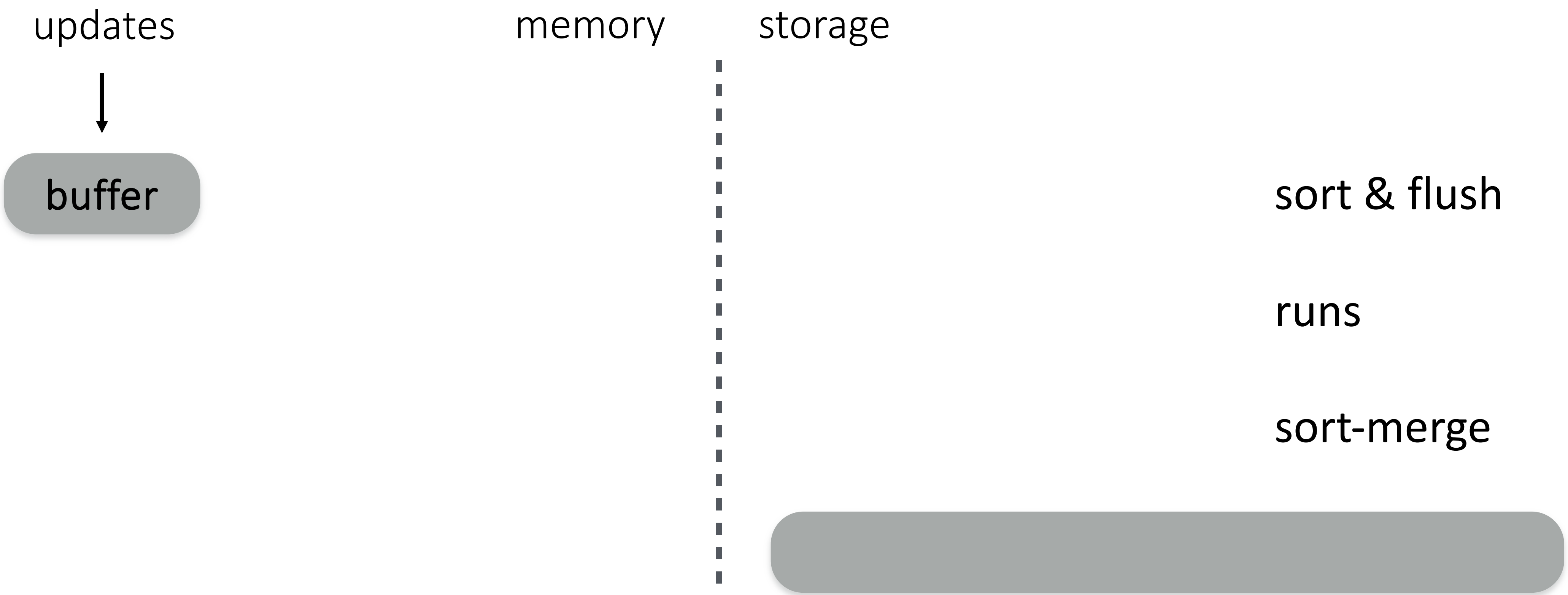


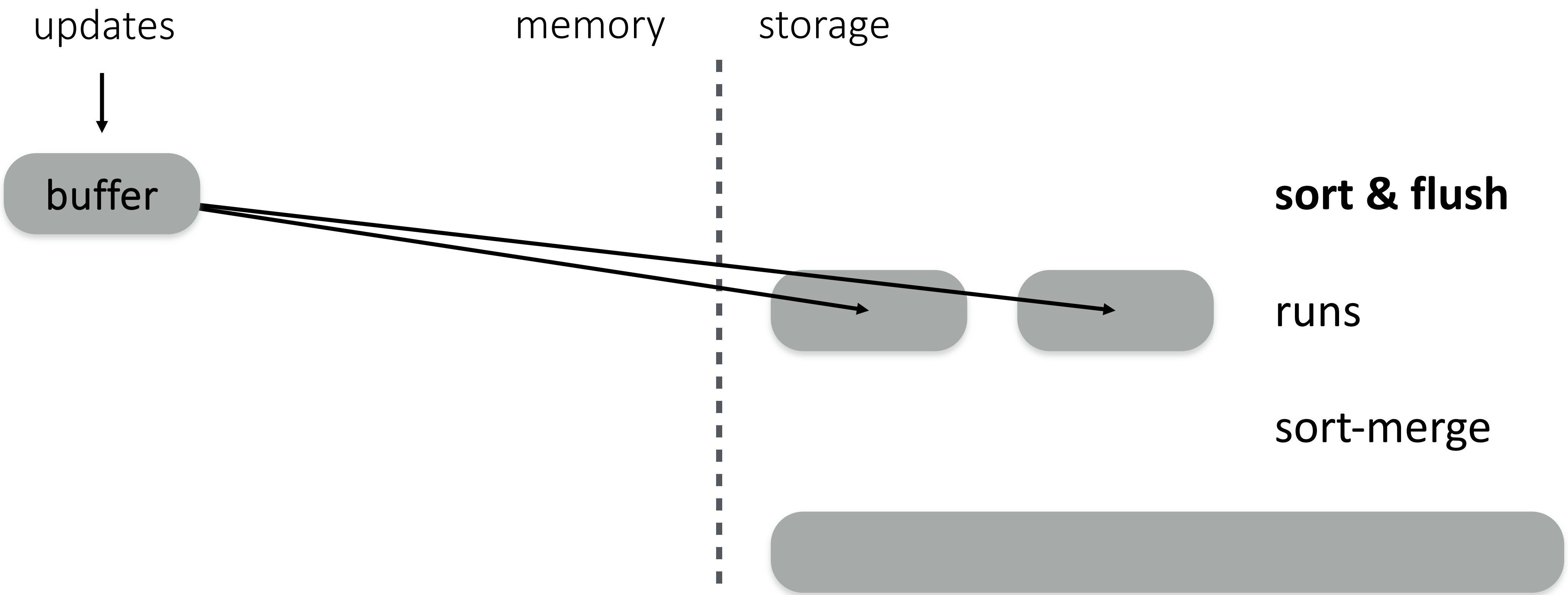
updates
memory
storage
buffer
sort & flush
runs
sort-merge

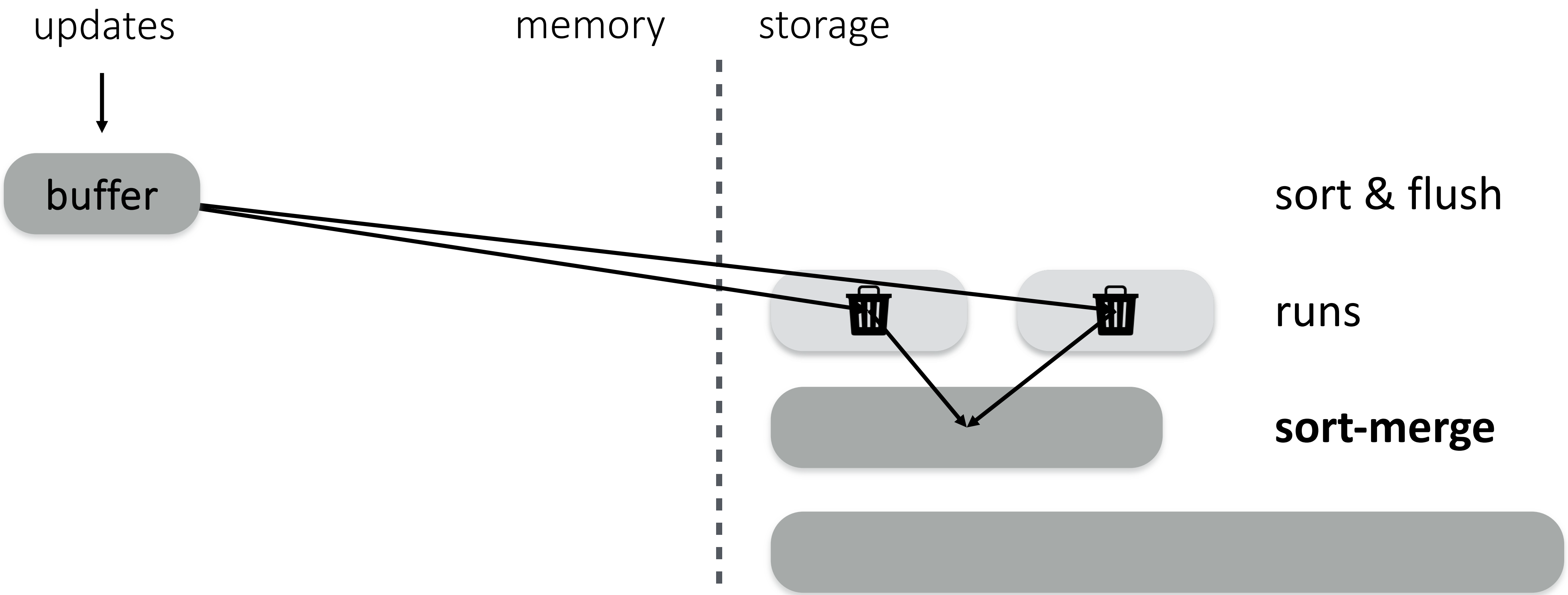

updates
memory
storage
buffer
sort & flush
runs

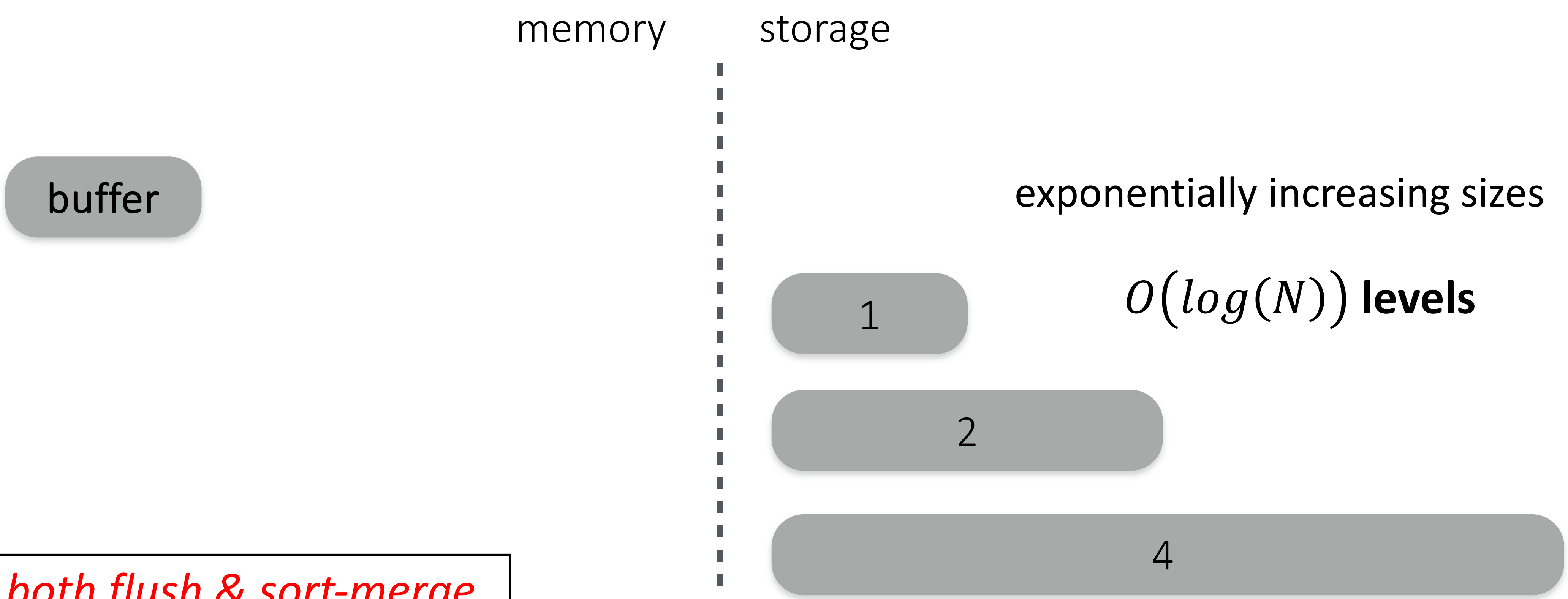

updates
buffer
memory
storage
sort & flush
runs


updates
buffer
memory
storage
sort & flush
runs


updates
buffer
memory
storage
sort & flush
runs
sort-merge


updates
memory
storage
buffer
sort & flush
runs
sort-merge


updates
memory
storage
buffer
sort & flush
runs
sort-merge



both flush & sort-merge are sequential writes!

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

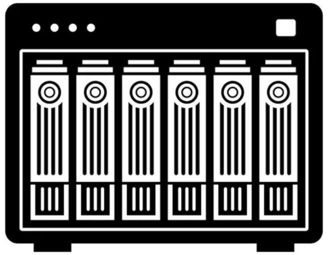
Patrick O'Neil
UMass Boston

LSM-tree
O'Neil *et al.*

1996

✅ good sequential reads & writes

✅ good random writes



array
of discs

why?

RAID, striping ← 

❌ LSM not explicitly needed

LSM-tree
O'Neil *et al.*

so, **arrays of disks** were **enough**!

Bigtable

how many IOPS?

10KRPM
max seek time 1.5ms
100 disks

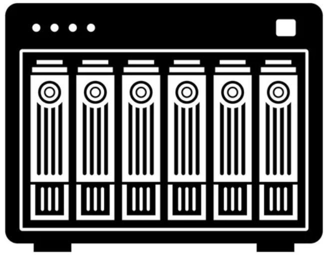
10KRPM: 10K rev in 60s
60/10000=6ms per rev
avg. rot. delay: 3ms (6ms/2)
avg. seek time: 0.75ms (1.5ms/2)
1 I/O / 3.75ms: 267 IOPS
100 disks: 26,700 IOPS

1980s — 1996 — 2006

a decade

✅ good sequential reads & writes

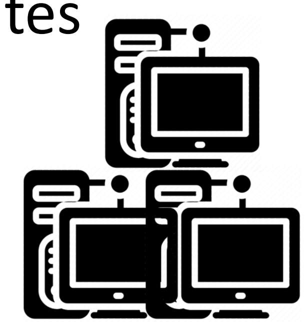
✅ good random writes



array
of discs

❌ worse sequential access

❌ bad random writes



commodity
hardware

what happened in 2006?

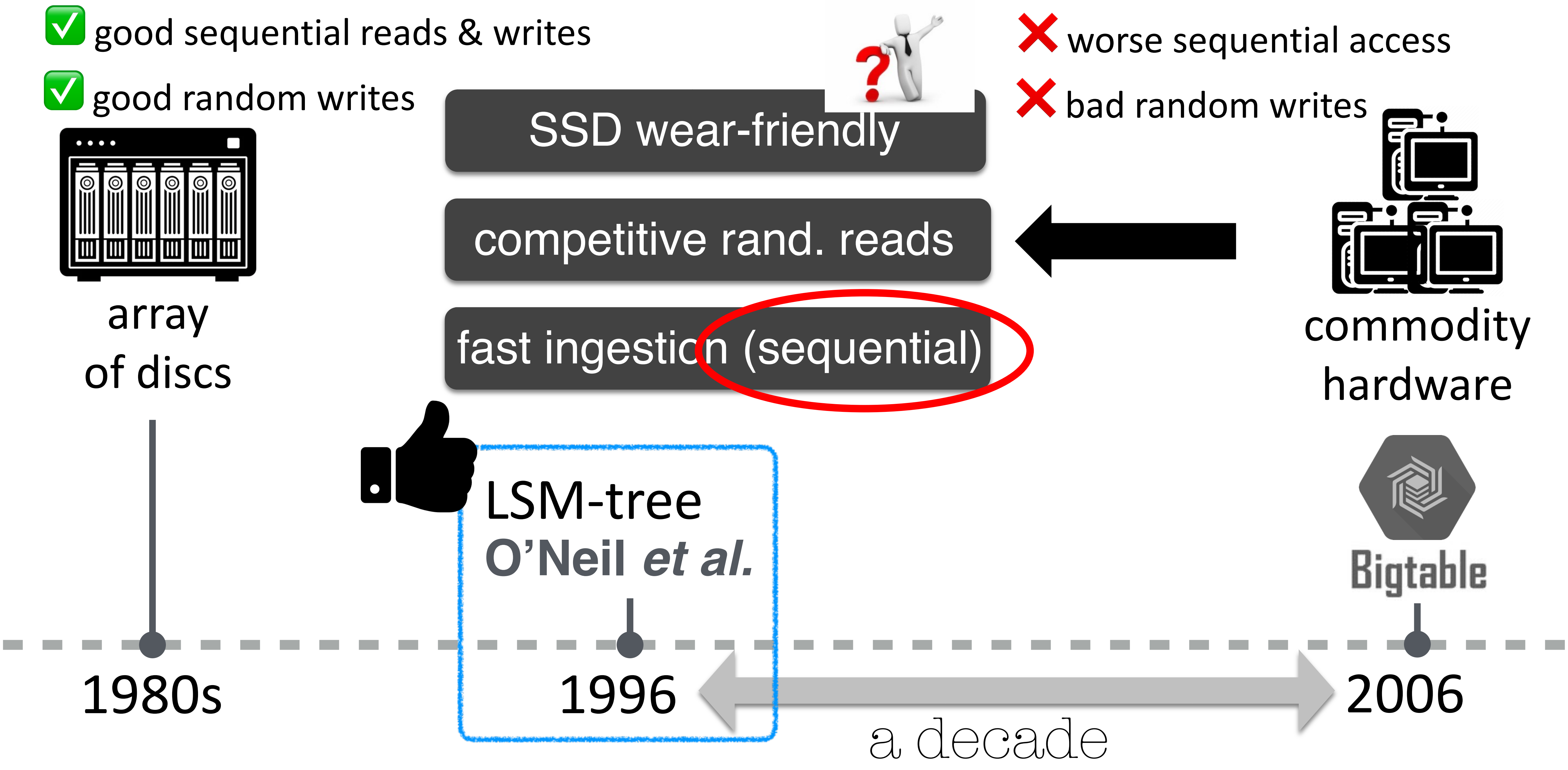
LSM-tree
O'Neil *et al.*

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

APACHE HBASE

Bigtable

LSM-tree
O'Neil *et al.*

1996

2006

2007

APACHE HBASE

Bigtable

cassandra

LSM-tree
O'Neil *et al.*

1996

2006

2007

2010

LSM-tree
O'Neil *et al.*
1996

Bigtable
2006

APACHE HBASE
2007

cassandra
2010

levelDB
2011

LSM-tree
O'Neil *et al.*

Bigtable

APACHE HBASE

cassandra

levelDB

RocksDB

1996 2006 2007 2010 2011 2013

LSM-tree
O'Neil *et al.*
1996

Bigtable
2006

APACHE HBASE
2007

cassandra
2010

levelDB
2011

RocksDB
2013

2023

LSM-tree

NoSQL

RocksDB
WT
levelDB
SCYLLA
riak
cassandra
tarantool
Bigtable
APACHE HBASE
DynamoDB
accumulo™
speedb

SQLite

relational

influxdb
QuasarDB

time-series

2023

LSM-tree

NoSQL

RocksDB
WT
levelDB
SCYLLA
riak
cassandra
tarantool
Bigtable
APACHE HBASE
DynamoDB
accumulo™
speedb

SQLite

relational

influxdb
QuasarDB

time-series

2023

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

$\mathfrak{n}$ entries

$\boldsymbol{B}$ entries fit into a disk block

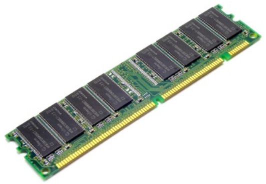
Array spans $\mathbf{N} = \frac{\mathfrak{n}}{\boldsymbol{B}}$ disk blocks

Lookup method & cost?

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

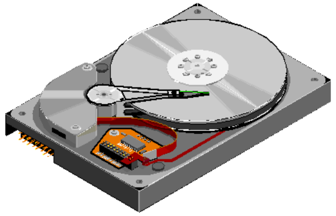
Insertion cost?

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



Buffer
James
Sara

Array size	Pointer



Block 1	Block 2	…	Block N
Anne	Bob		Yulia
Arnold	Corrie		Zack
Barbara	Doug		Zelda

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)

$\boldsymbol{\mathfrak{n}}$ entries

$\boldsymbol{B}$ entries fit into a disk block

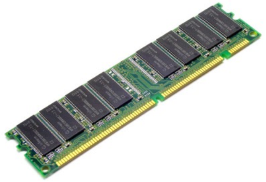
Array spans $\mathbf{N} = \frac{\boldsymbol{\mathfrak{n}}}{\boldsymbol{B}}$ disk blocks

Lookup method & cost?

Scan: $O(N)$

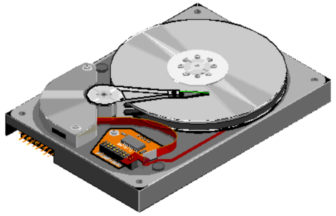
Insertion cost?

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



Buffer
James
Sara

Array size	Pointer



Block 1	Block 2	…	Block N
Doug	Yulia		Anne
Zelda	Zack		Bob
Arnold	Barbara		Corrie

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		

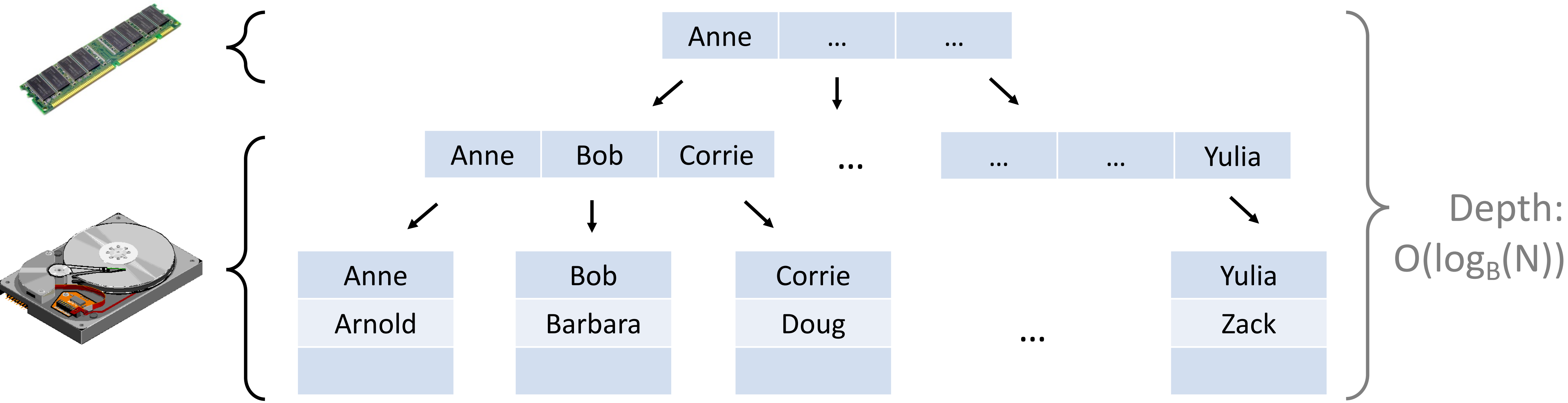
B-tree

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

Goetz Graefe
Microsoft, HP Fellow, now Google
ACM Software System Award

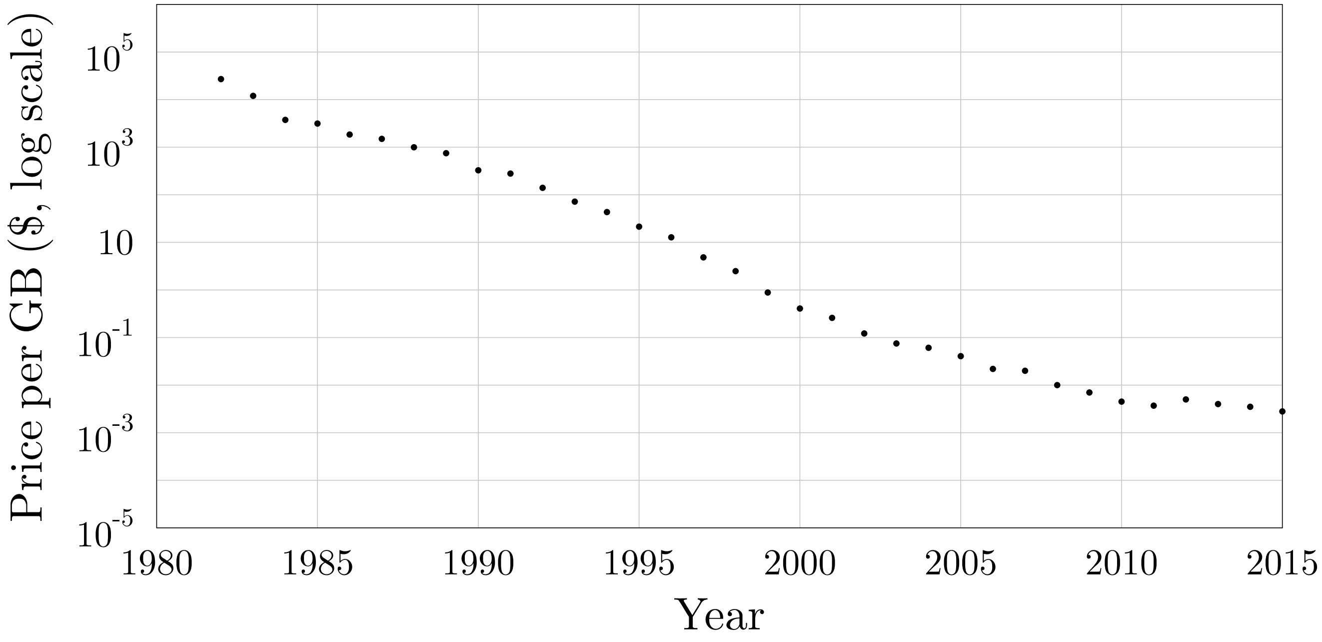
“It could be said that the world’s information is at our fingertips because of B-trees”

B-trees are no longer sufficient

Cheaper storage

Workloads more **insert-intensive**

We need **better insert-performance**



Results Catalogue

Goal to combine

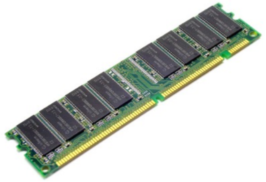
sub-constant insertion cost
logarithmic lookup cost

	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

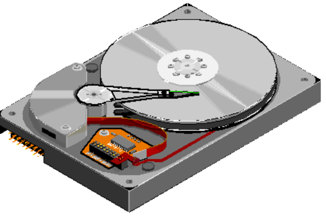
Level



Buffer

0

1



Sorted arrays

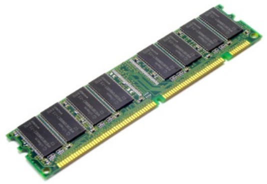
2

3

Basic LSM-tree

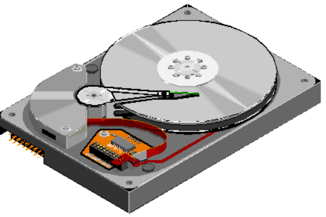
Design principle #1: optimize for insertions by buffering

Level



Buffer

0



Sorted arrays

1

2

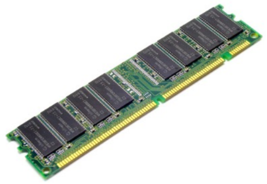
3

Basic LSM-tree

Design principle #1: optimize for insertions by buffering

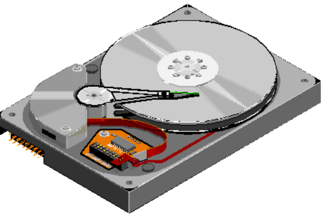
Inserts

Level



Buffer 0

…	…	…



Sorted arrays 1

2

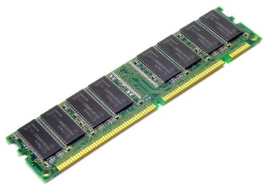
3

Basic LSM-tree

Design principle #1: optimize for insertions by buffering

Inserts

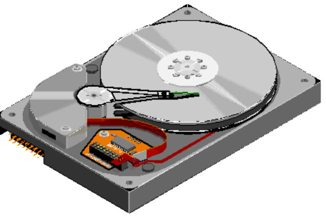
Level



Buffer 0

sort & flush buffer

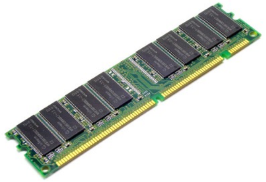
... ...



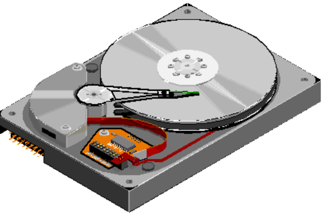
Sorted arrays 1 2 3

Basic LSM-tree

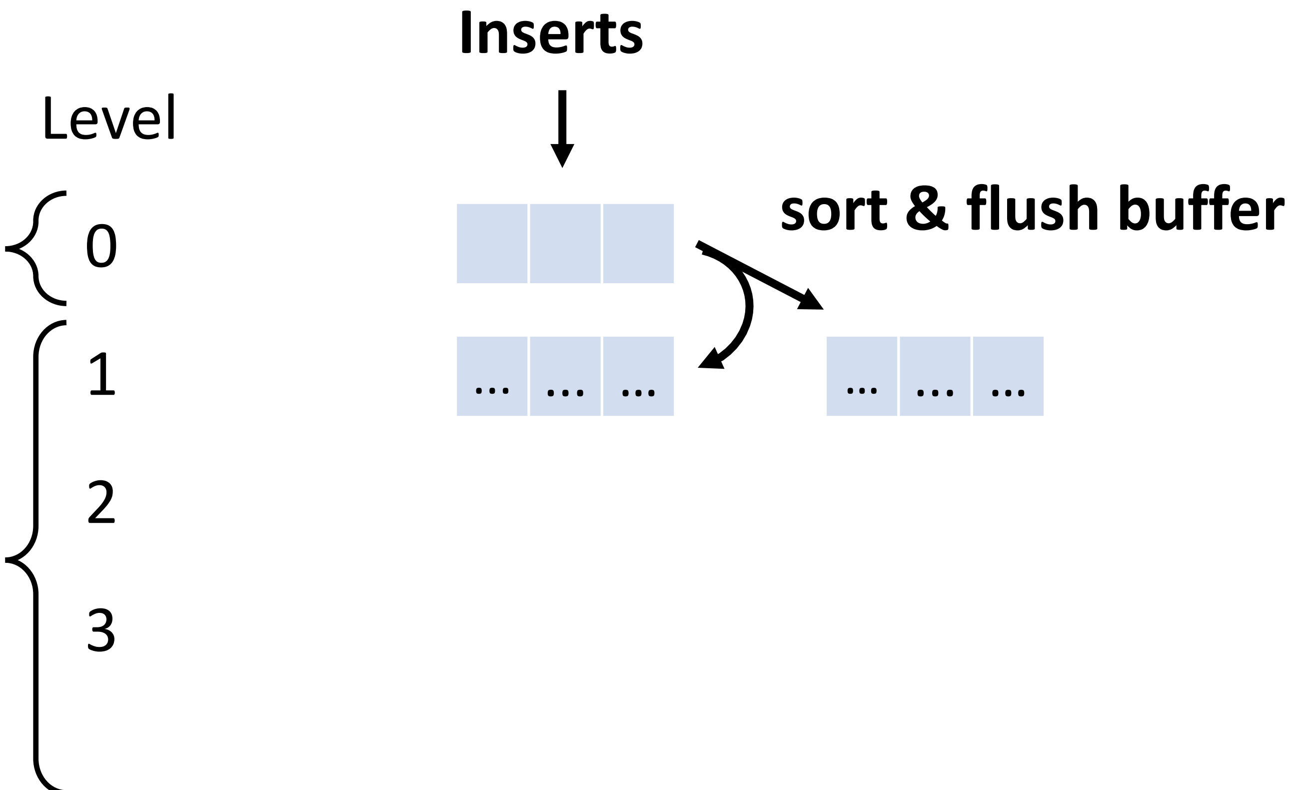
Design principle #1: optimize for insertions by buffering



Buffer



Sorted arrays

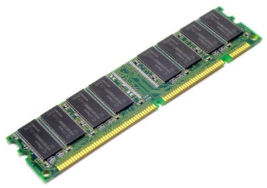


Basic LSM-tree

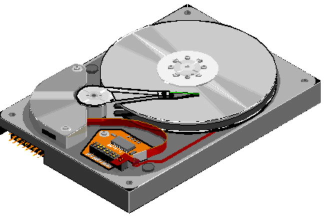
Design principle #1: optimize for insertions by buffering

Design principle #2: optimize for lookups by sort-merging arrays

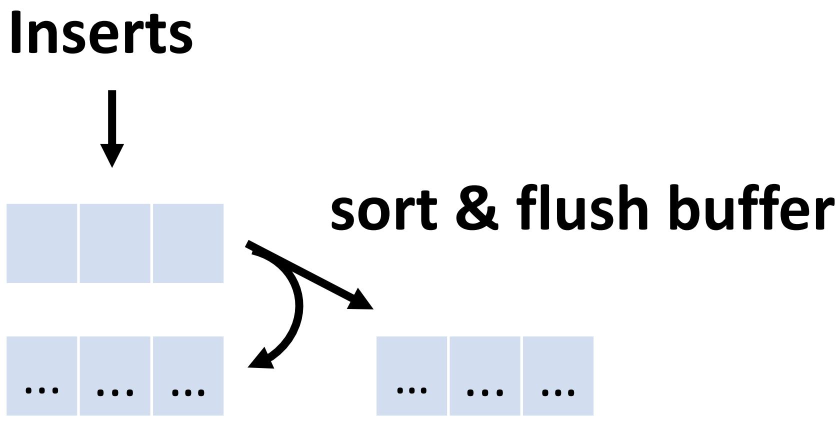
Level



Buffer { 0



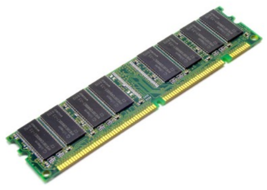
Sorted arrays { 1, 2, 3



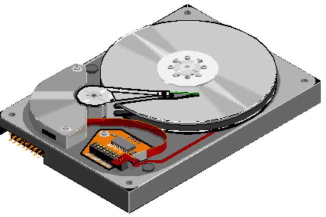
Basic LSM-tree

Design principle #1: optimize for insertions by buffering

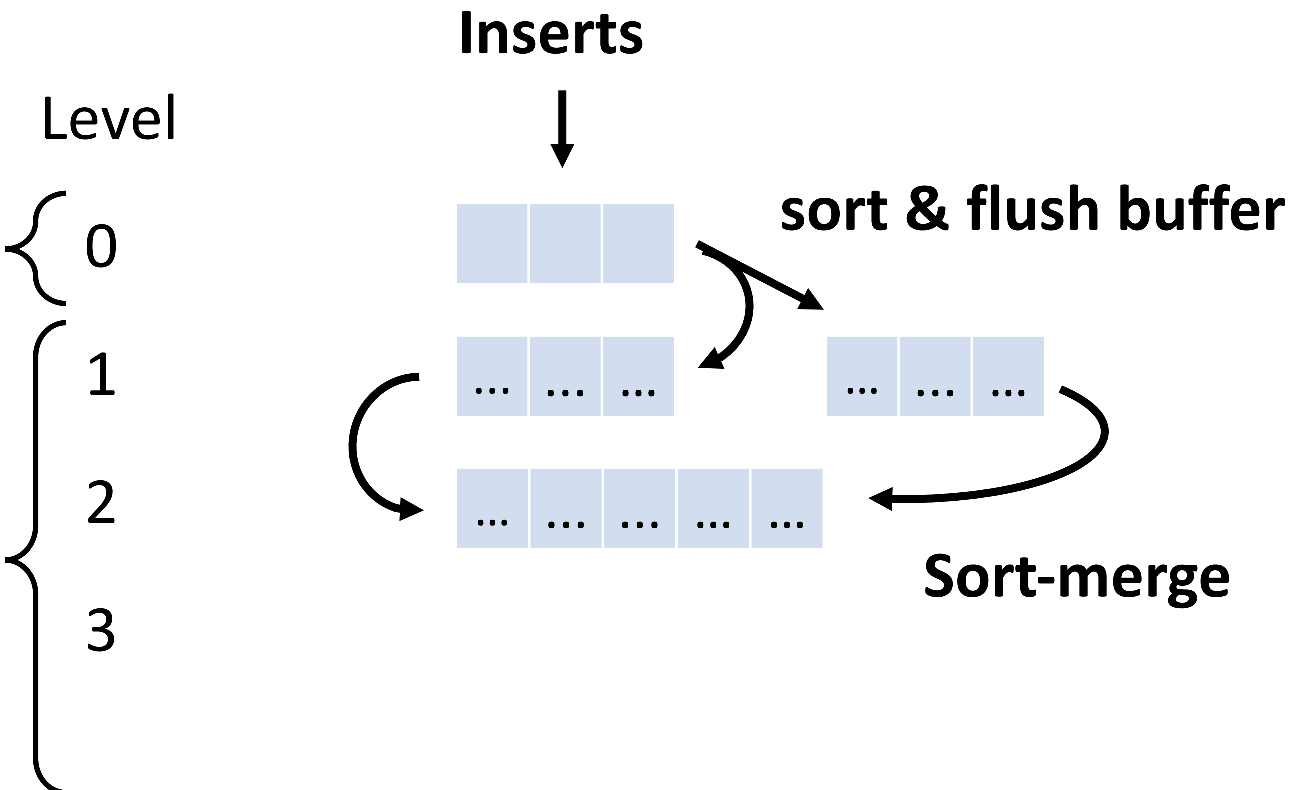
Design principle #2: optimize for lookups by sort-merging arrays



Buffer



Sorted arrays

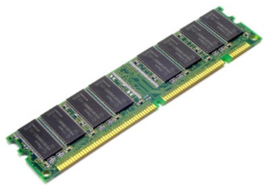


Basic LSM-tree

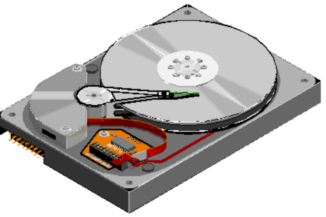
Design principle #1: optimize for insertions by buffering

Design principle #2: optimize for lookups by sort-merging arrays

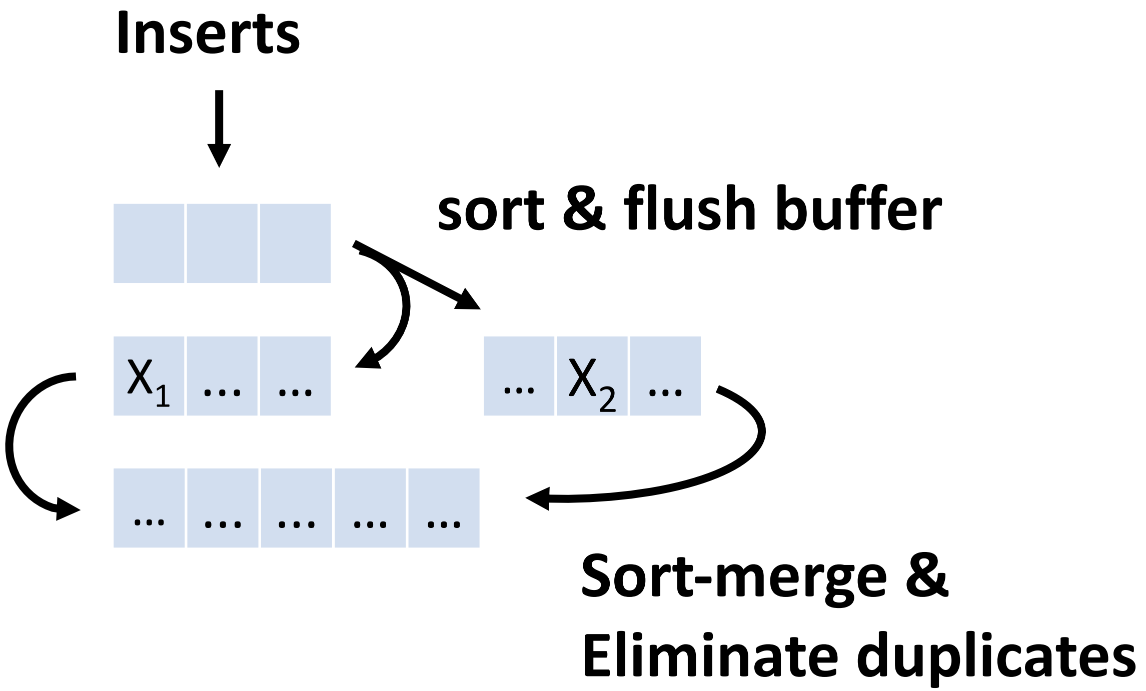
Level



Buffer — 0



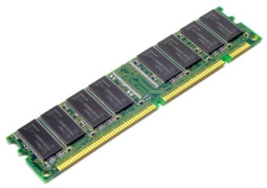
Sorted arrays — 1, 2, 3



Basic LSM-tree

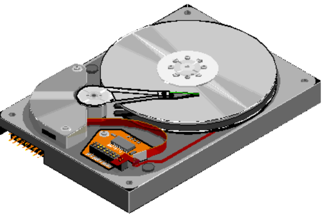
Design principle #1: optimize for insertions by buffering

Design principle #2: optimize for lookups by sort-merging arrays

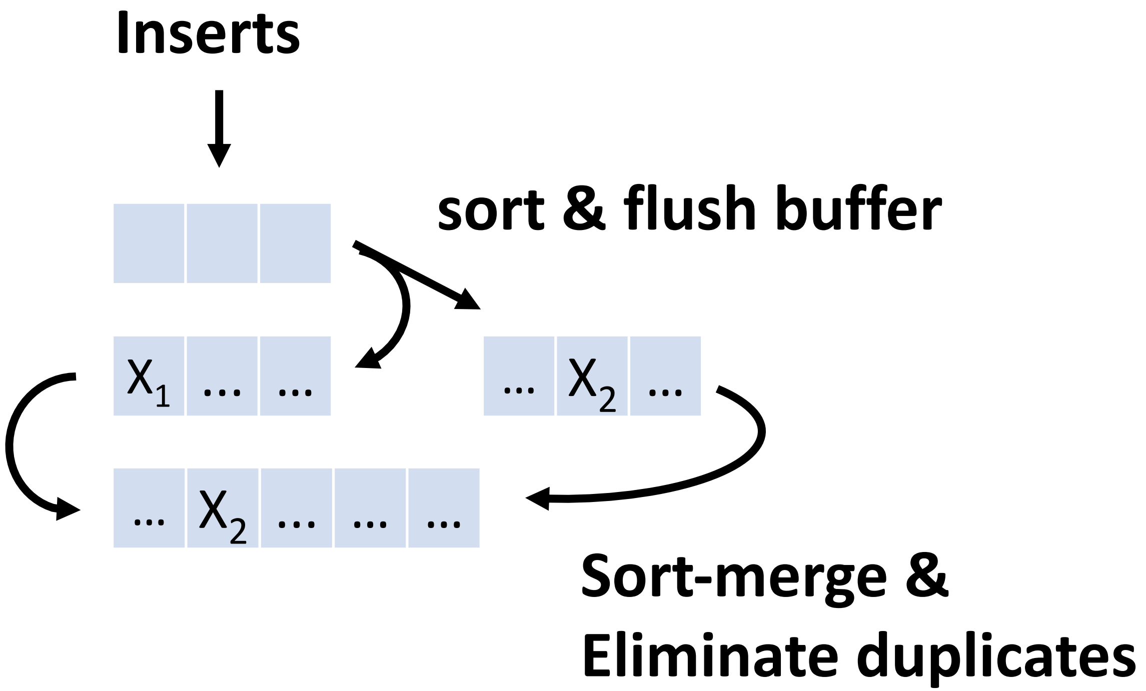


Level

Buffer — 0



Sorted arrays — 1, 2, 3



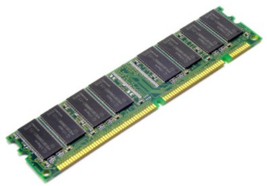
Basic LSM-tree

Design principle #1: optimize for insertions by buffering

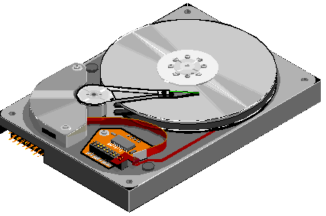
Design principle #2: optimize for lookups by sort-merging arrays

Inserts

Level

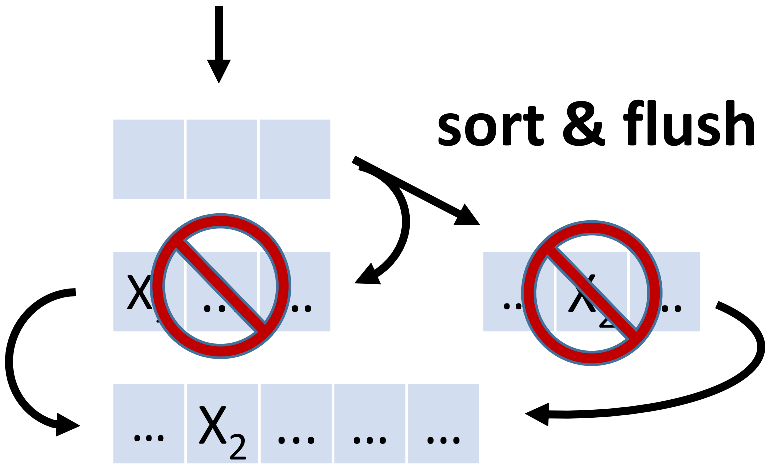


Buffer — 0



Sorted arrays — 1, 2, 3

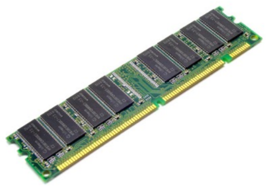
sort & flush buffer



Sort-merge &
Eliminate duplicates &
Discard original arrays

Basic LSM-tree – Example

Level



Buffer

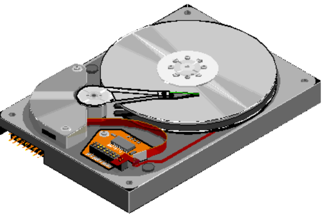
0

Sorted arrays

1

2

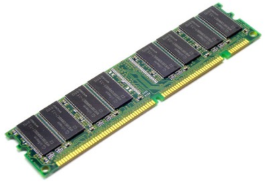
3



Basic LSM-tree – Example

inserts

Level

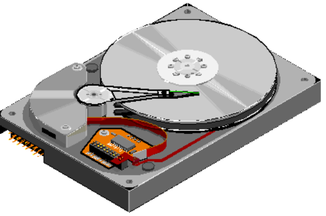


Buffer

0

4	9	

1



Sorted arrays

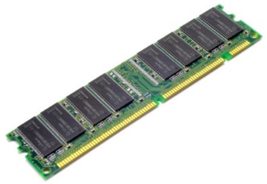
2

3

Basic LSM-tree – Example

inserts

Level



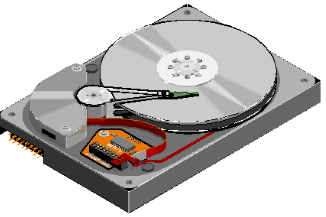
Buffer

0

4	9	6

sort

1

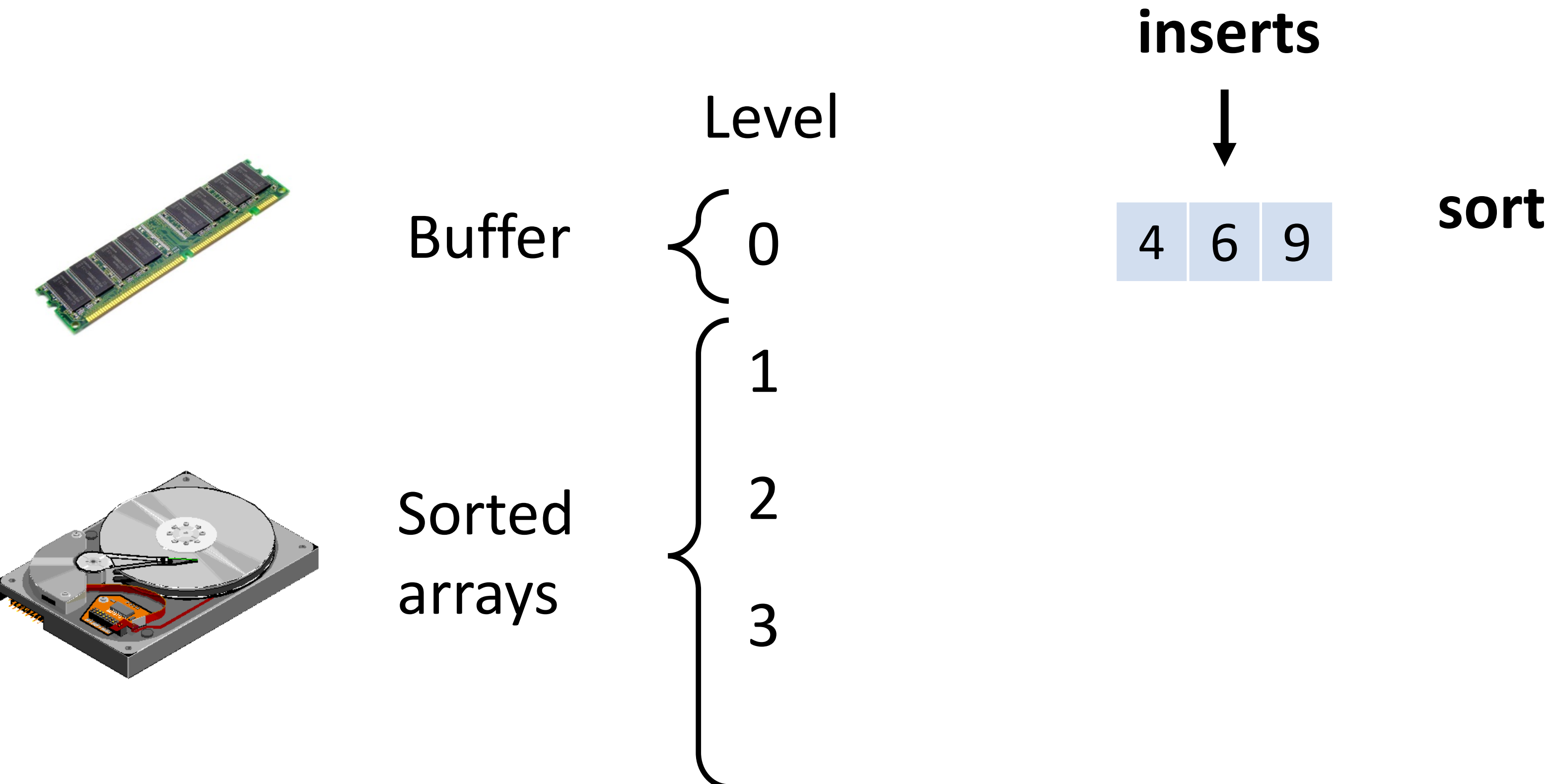


Sorted arrays

2

3

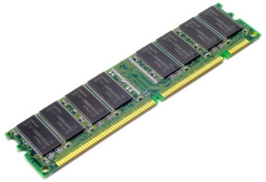
Basic LSM-tree – Example



Basic LSM-tree – Example

inserts

Level



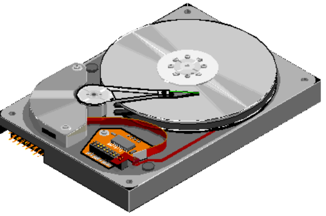
Buffer

0

sort & flush buffer

4	6	9

1



Sorted arrays

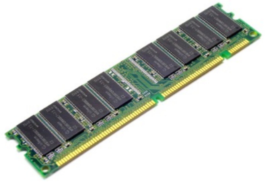
2

3

Basic LSM-tree – Example

inserts

Level

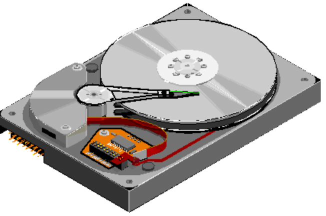


Buffer

0

1

4	6	9



Sorted arrays

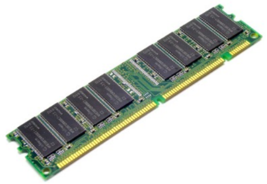
2

3

Basic LSM-tree – Example

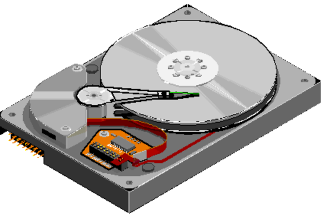
inserts

Level



Buffer

0: 3 8 4



Sorted arrays

1: 4 6 9

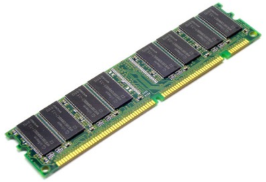
2

3

Basic LSM-tree – Example

inserts

Level



Buffer

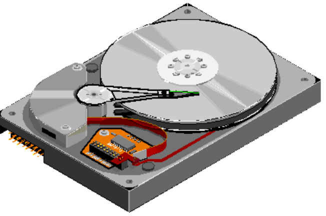
0

sort & flush buffer

1

4	6	9

3	4	8



Sorted arrays

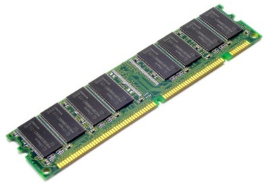
2

3

Basic LSM-tree – Example

inserts

Level

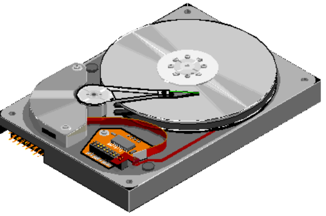


Buffer

0

1

4	6	9		3	4	8

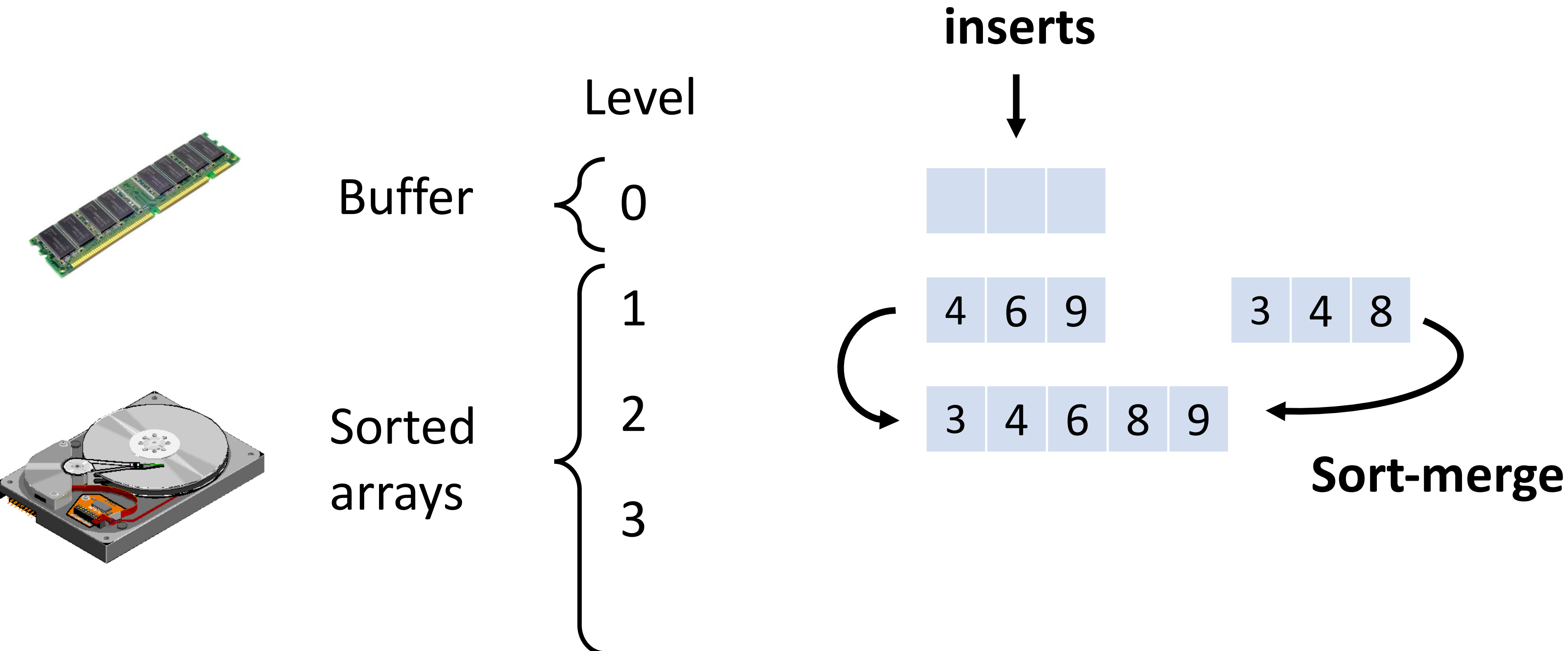


Sorted arrays

2

3

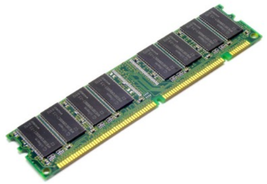
Basic LSM-tree – Example



Basic LSM-tree – Example

inserts

Level



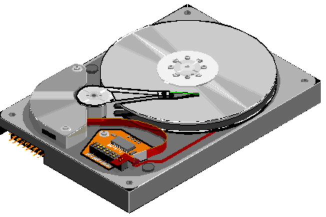
Buffer

0

1

2

3



Sorted arrays

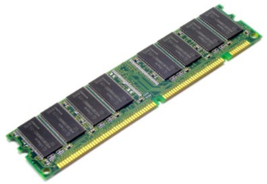
4_1	6	9

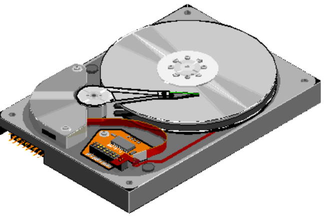
3	**4_2**	8

3	**4_2**	6	8	9

Sort-merge &
Eliminate duplicates

Basic LSM-tree – Example

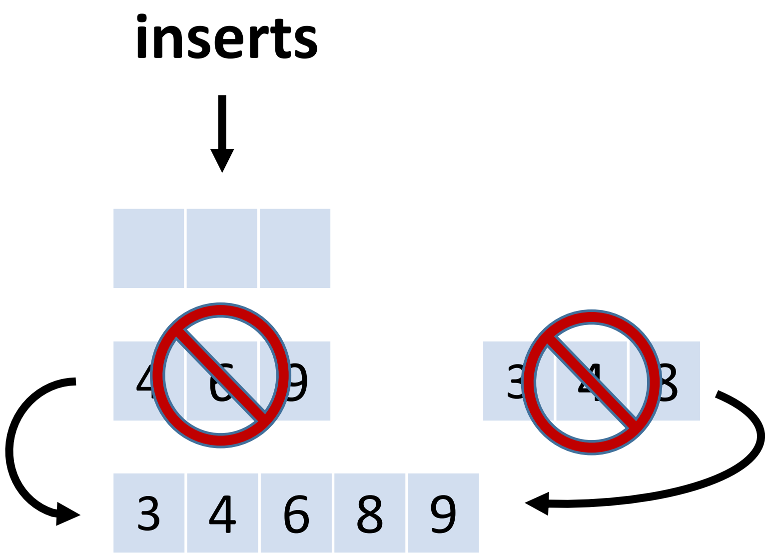




Level

Buffer 0

Sorted arrays 1 2 3

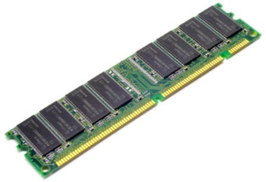


Sort-merge &
Eliminate duplicates &
Discard original arrays

Basic LSM-tree – Example

inserts

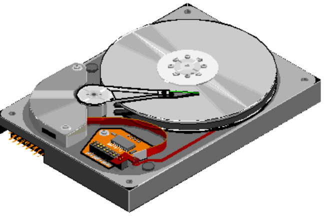
Level



Buffer

0

1



Sorted arrays

2

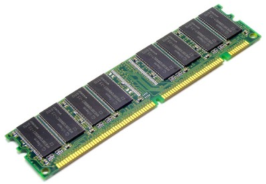
3	4	6	8	9

3

Basic LSM-tree – Example

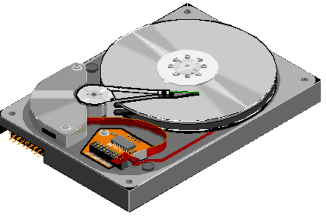
inserts

Level



Buffer

0: | 2 | 7 | 8 |



Sorted arrays

1:

2: | 3 | 4 | 6 | 8 | 9 |

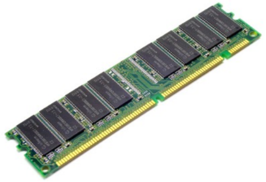
3:

Basic LSM-tree – Example

inserts

Level

sort & flush buffer

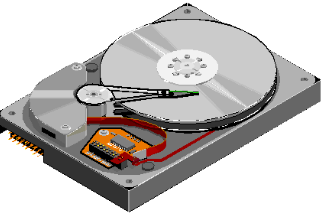


Buffer

0

1

2	7	8



Sorted arrays

2

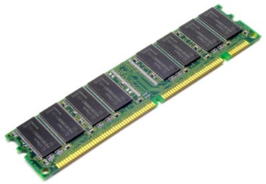
3	4	6	8	9

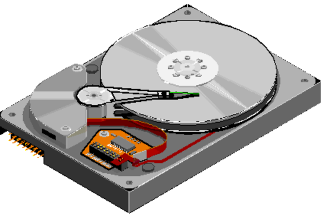
3

Basic LSM-tree – Example

inserts

	Level	
Buffer	0	
Sorted arrays	1	2 7 8
	2	3 4 6 8 9
	3	

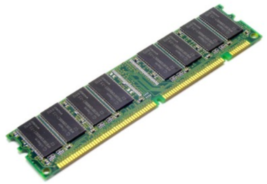


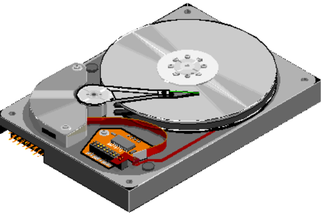


Basic LSM-tree

Levels have exponentially increasing capacities.

How many levels?  $\log_2(N)$

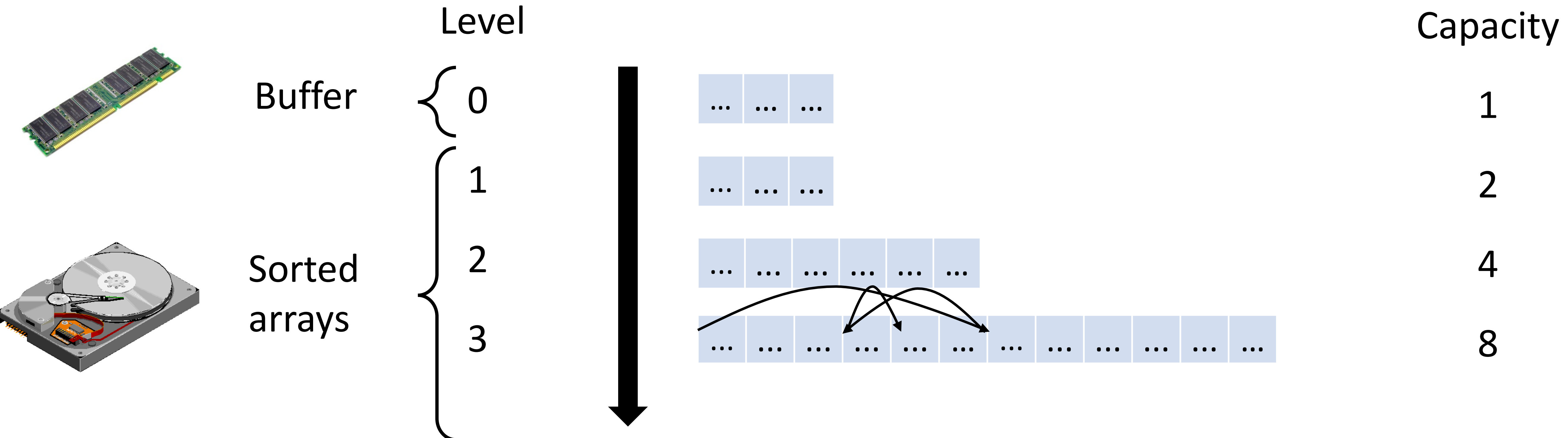




	Level		Capacity
Buffer	0	... ...	1
Sorted arrays	1	... ...	2
	2	... ...	4
	3	... ...	8

Basic LSM-tree – Lookup cost

Lookup method?	Search youngest to oldest.	$O(\log_2(N))$
How?	Binary search.	$O(\log_2(N))$
Lookup cost?		$O(\log_2(N)^2)$

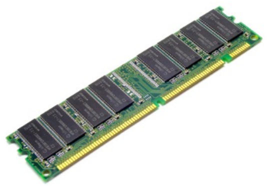


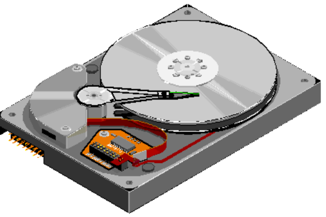
Basic LSM-tree – Insertion cost

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





	Level		Capacity
Buffer	0	… … …	1
Sorted arrays	1	… … …	2
	2	… … … … … …	4
	3	… … … … … … … … … … … …	8

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		

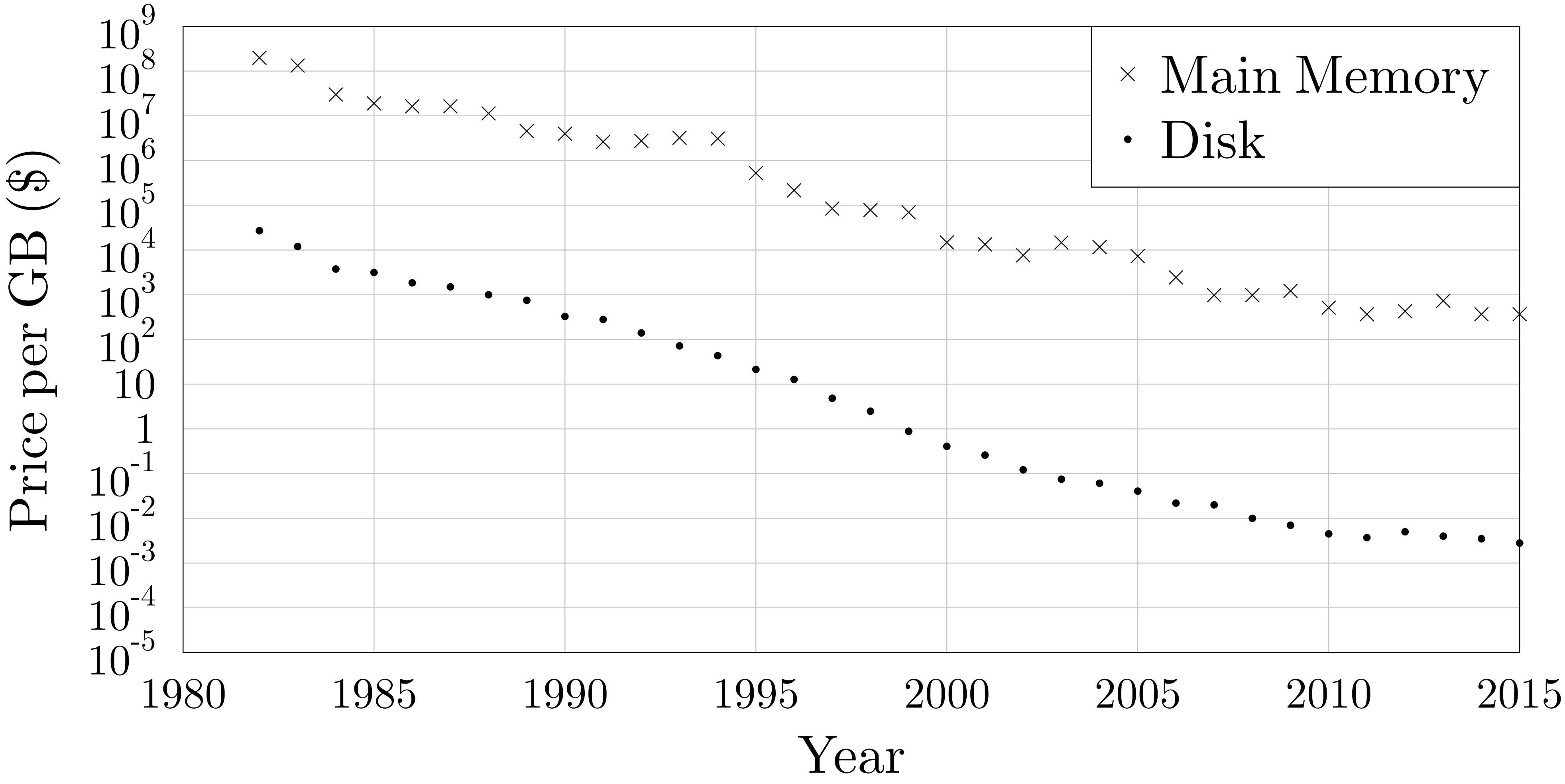
Results Catalogue

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

Can we improve the lookup cost?

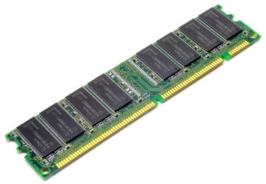
	Lookup cost	Insertion cost
Sorted array	$O(\log_2(N))$	$O(N/2)$
Log	$O(N)$	$O(1/B)$
B-tree	$O(\log_B(N))$	$O(\log_B(N))$
Basic LSM-tree	$O(\log_2(N)^2)$	$O(1/B \cdot \log_2(N))$
Leveled LSM-tree		
Tiered LSM-tree		

Declining Main Memory Cost



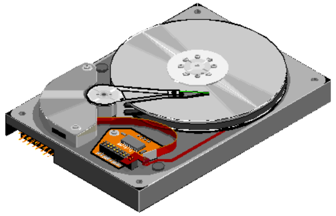
Declining Main Memory Cost

Store a fence pointer for every block in main memory



Fence pointers

1	10	15	…



array

Block 1	Block 2	Block 3	…
1	10	15	…
3	11	16	…
6	13	18	…

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!

updates

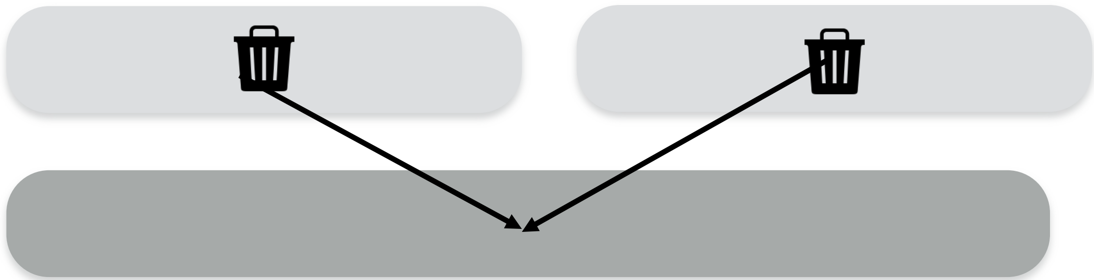
buffer

memory

storage

sort & flush

runs



Can we change that?

Leveled LSM-tree

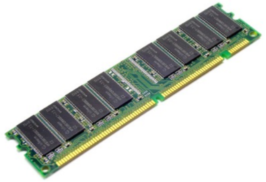
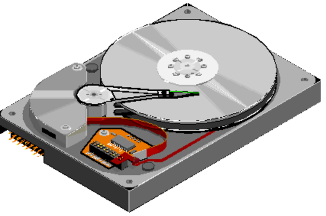
↓ Lookup cost ↑ Update cost

Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

Increase size ratio T

	Level		Capacity
 Buffer	0	... ...	T^0
Sorted arrays	1		T^1
	2		T^2
	3		T^3

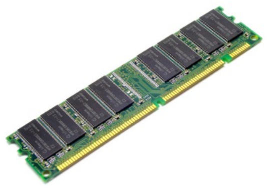
Leveled LSM-tree

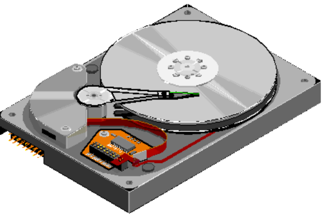
Lookup cost depends on number of levels

How to reduce it?

Increase size ratio T

E.g. size ratio of 4





	Level		Capacity
Buffer	0	... ...	1
Sorted arrays	1		4
	2		16
	3		64

Leveled LSM-tree

Lookup cost depends on number of levels

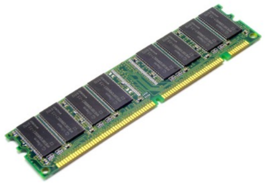
How to reduce it?

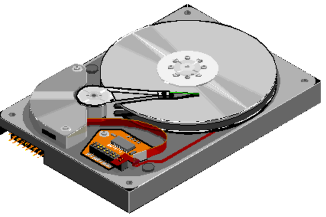
Increase size ratio T

E.g. size ratio of 4

inserts

	Level		Capacity
Buffer	0	… … …	1
Sorted arrays	1		4
	2		16
	3		64





Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

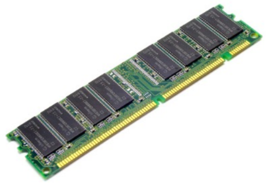
Increase size ratio T

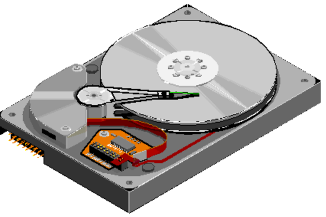
E.g. size ratio of 4

inserts

	Level		Capacity
Buffer	0	… … …	1
Sorted arrays	1	… … …	4
	2		16
	3		64

flush





Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

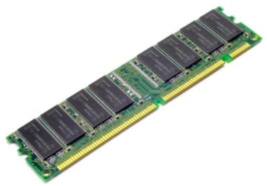
E.g. size ratio of 4

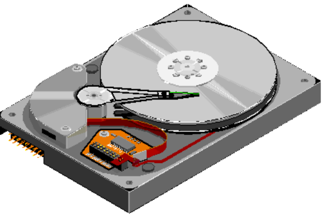
Increase size ratio T

inserts

Level			Capacity
Buffer	0	… … …	1
Sorted arrays	1	… … … … … …	4
	2		16
	3		64

flush & sort-merge





Leveled LSM-tree

Lookup cost depends on number of levels

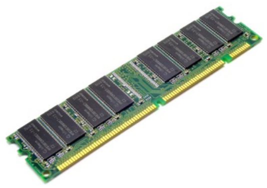
How to reduce it? Increase size ratio T

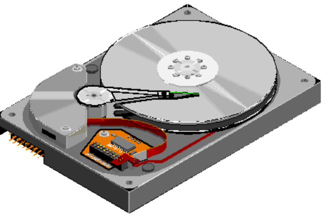
E.g. size ratio of 4

inserts

flush & sort-merge

	Level		Capacity
Buffer	0	… … …	1
Sorted arrays	1	… … … … … … … … …	4
	2		16
	3		64





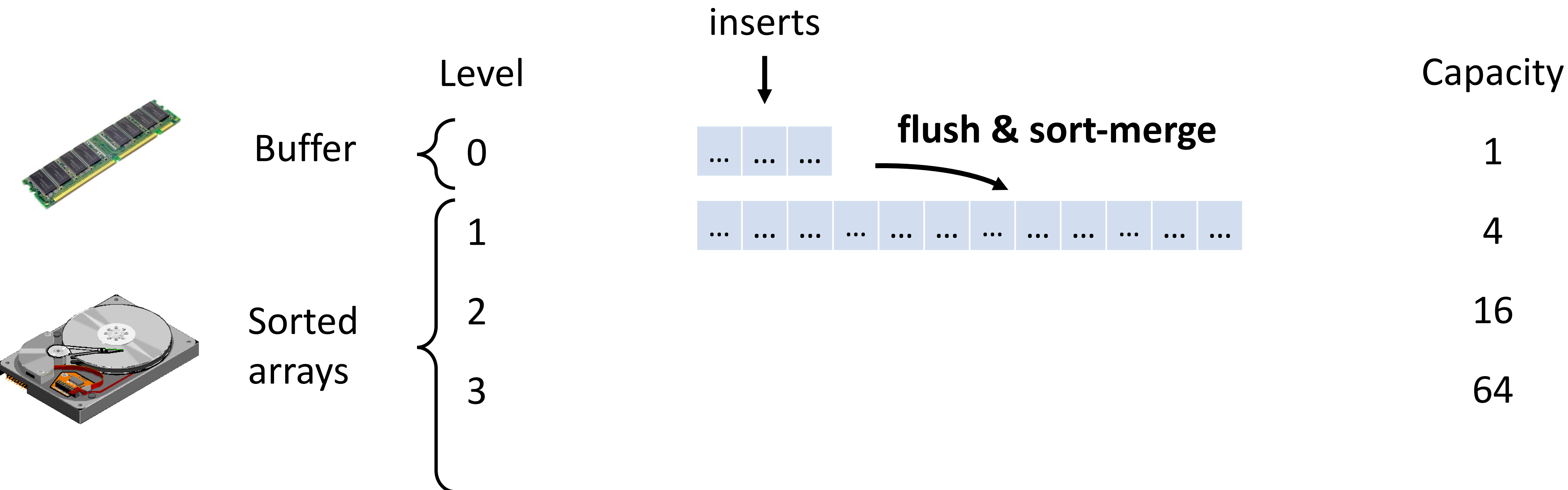
Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

Increase size ratio T

E.g. size ratio of 4



Leveled LSM-tree

Lookup cost depends on number of levels

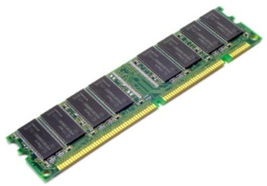
How to reduce it? Increase size ratio T

E.g. size ratio of 4

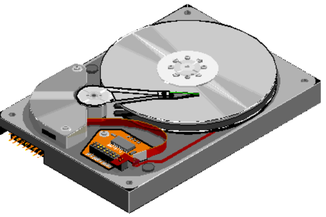
inserts

Level		Capacity
0	... ...	1
1	... ...	4
2		16
3		64

move



Buffer



Sorted arrays

Leveled LSM-tree

Lookup cost depends on number of levels

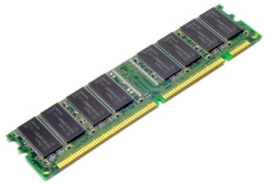
How to reduce it?

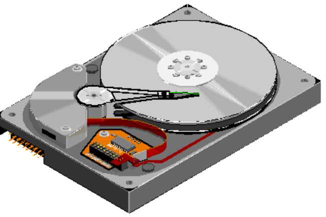
Increase size ratio T

E.g. size ratio of 4

inserts

	Level		Capacity
Buffer	0	... ...	1
Sorted arrays	1		4
	2	... ...	16
	3		64





Leveled LSM-tree

Lookup cost?

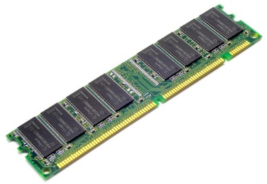
$O(\log_T(N))$

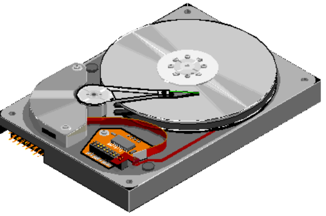
Insertion cost?

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

inserts

	Level		Capacity
Buffer	0	… … …	1
Sorted arrays	1		4
	2	… … … … … … … … … … … …	16
	3		64





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?

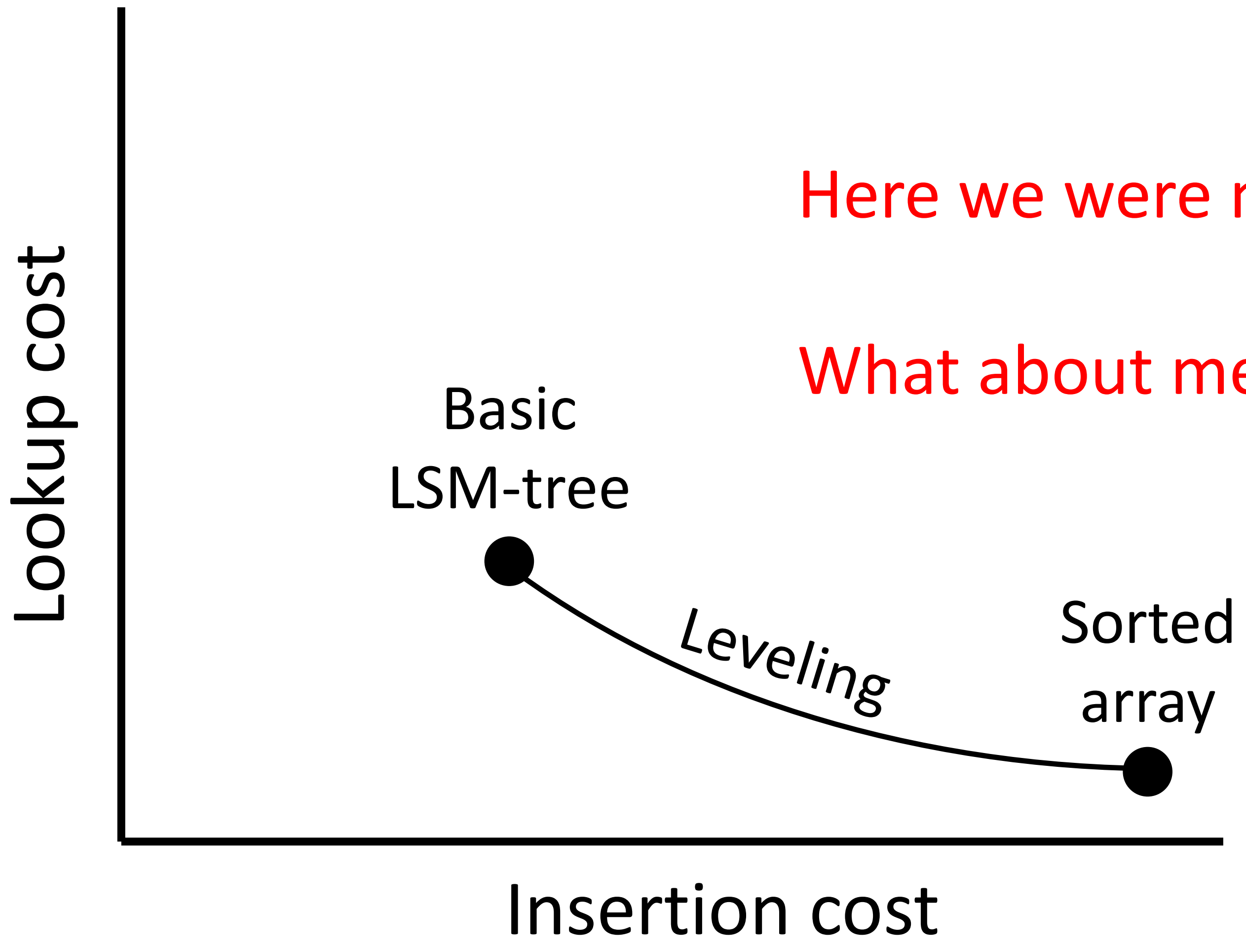
Lookup cost becomes:

O(1)

Insert cost becomes:

O(N/B)

The LSM-tree becomes a sorted array!



Here we were merging eagerly.

What about merging lazily?

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		

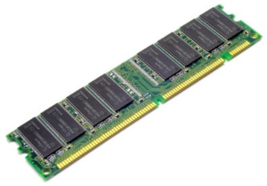
Tiered LSM-tree

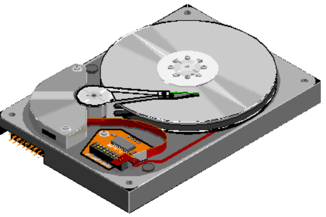
Lookup cost

Insertion cost

Tiered LSM-tree

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

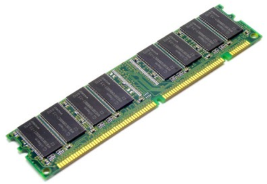


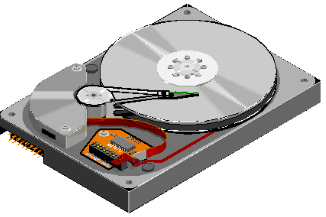


	Level		Capacity
Buffer	0	… … …	T^0
Sorted arrays	1		T^1
	2		T^2
	3		T^3

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





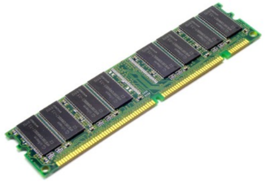
	Level		Capacity
Buffer	0	… … …	1
Sorted arrays	1		4
	2		16
	3		64

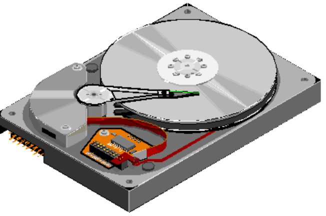
Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.
Do not merge within a level.
E.g. size ratio of 4

inserts

flush





	Level		Capacity
Buffer	0	… … …	1
Sorted arrays	1	… … …	4
	2		16
	3		64

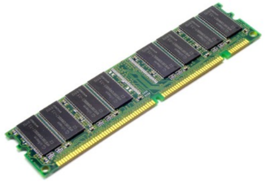
Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.
Do not merge within a level.
E.g. size ratio of 4

inserts

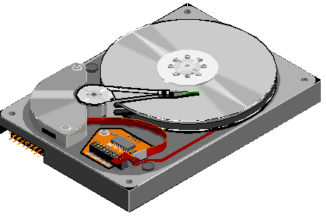
Level

Capacity



Buffer 0 | … … … flush | 1

1 | … … … … … … | 4



Sorted arrays

2 | 16

3 | 64

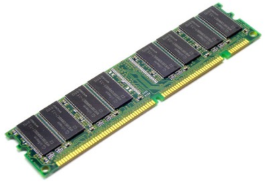
Tiered LSM-tree

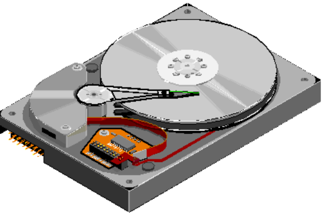
Reduce the number of levels by increasing the size ratio.
Do not merge within a level.
E.g. size ratio of 4

inserts

	Level		Capacity
Buffer	0	… … …	1
Sorted arrays	1	… … … … … … … … …	4
	2		16
	3		64

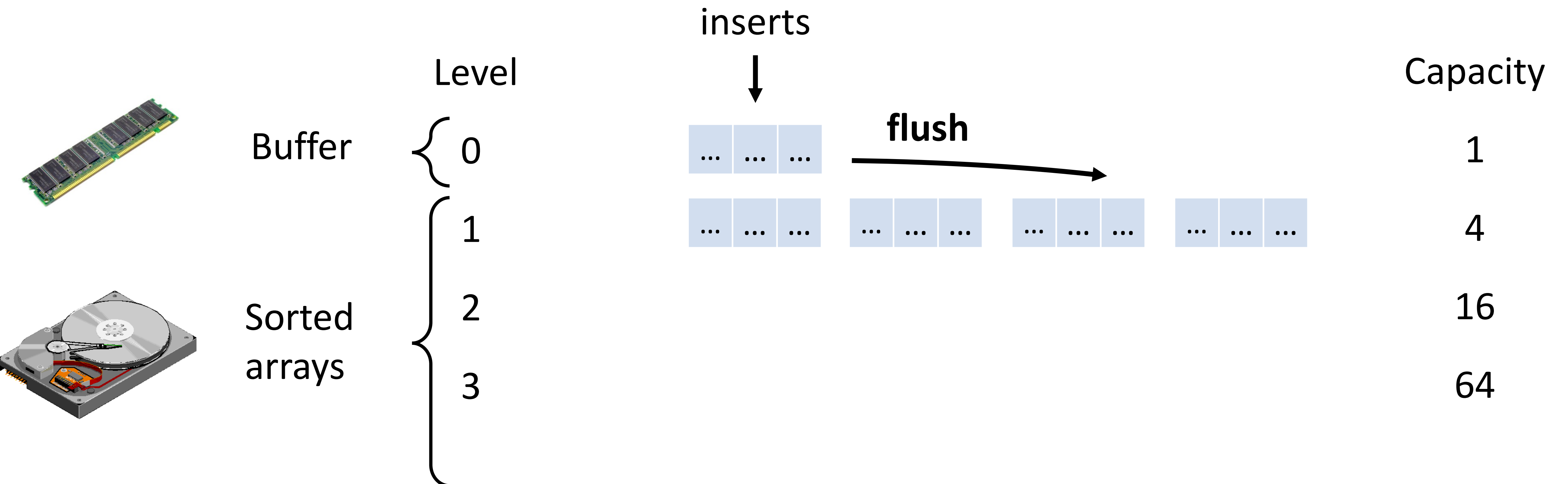
flush





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

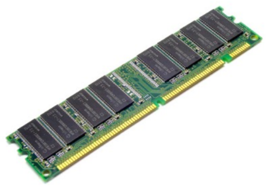


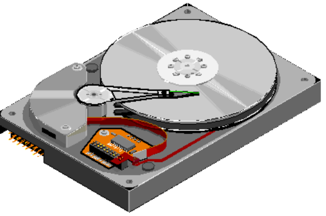
Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.
Do not merge within a level.
E.g. size ratio of 4

inserts

	Level		Capacity
Buffer	0	… … …	1
Sorted arrays	1	… … … … … … … … … … … …	4
	2	… … … … … … … … … … … …	16
	3		64

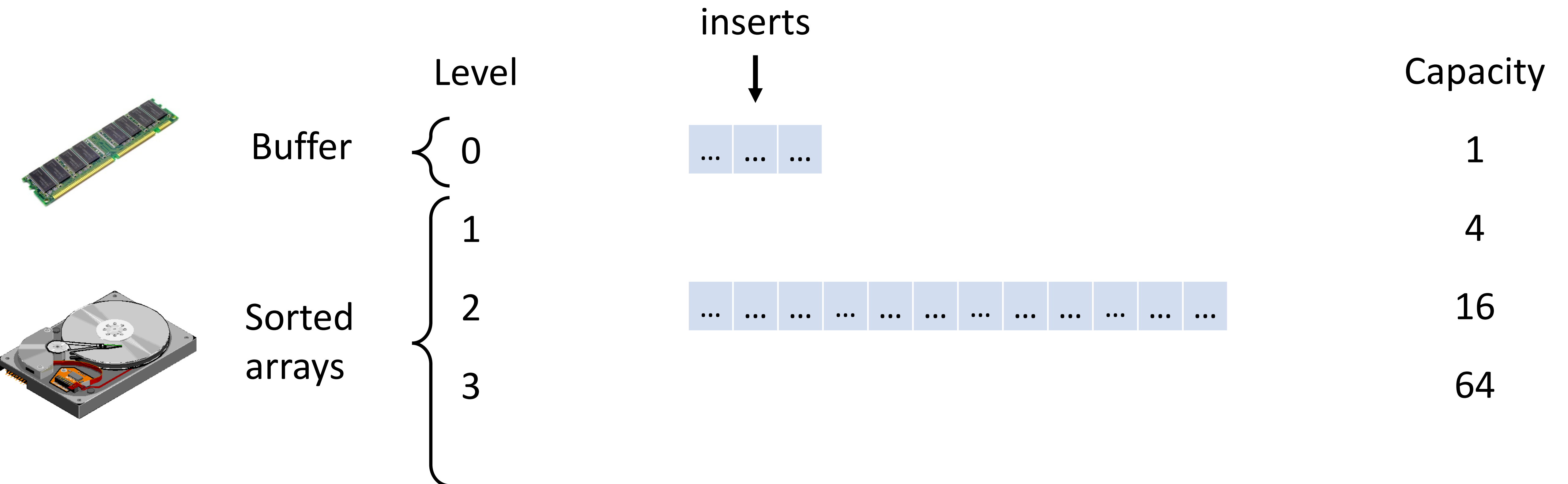




sort-merge

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?

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

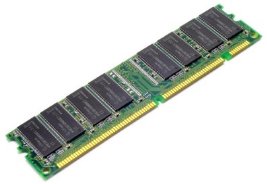
Insertion cost?

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

inserts

Level

Capacity

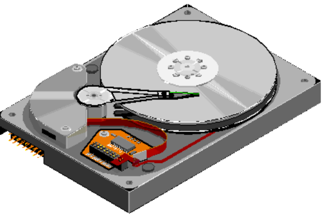


Buffer

0

... ...

1



Sorted arrays

1

4

2

... ...

16

3

64

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?

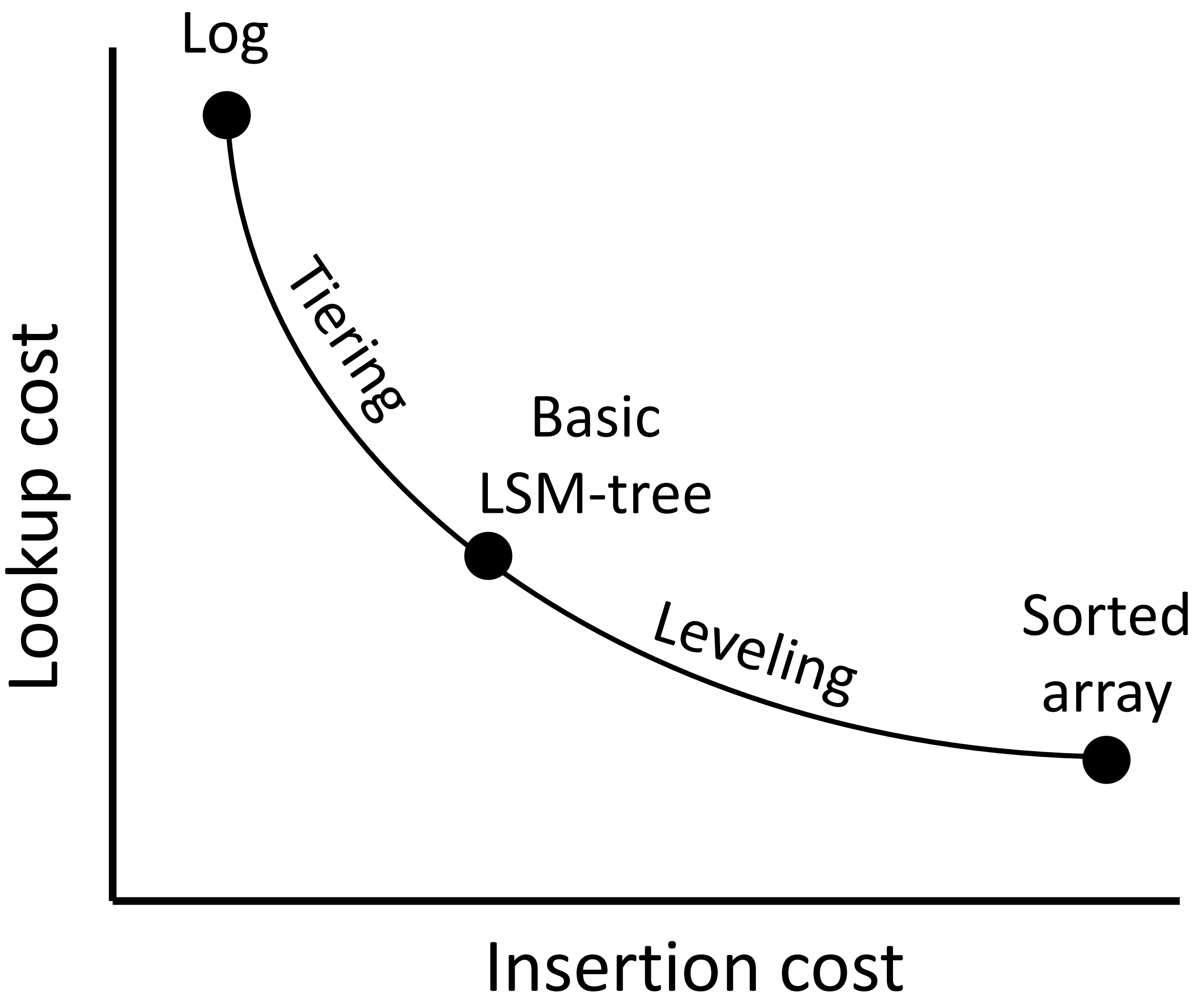
Lookup cost becomes:

O(N)

Insert cost becomes:

O(1/B)

The tiered LSM-tree becomes a log!


Log
Tiering
Basic
LSM-tree
Leveling
Sorted
array
Lookup cost
Insertion cost

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

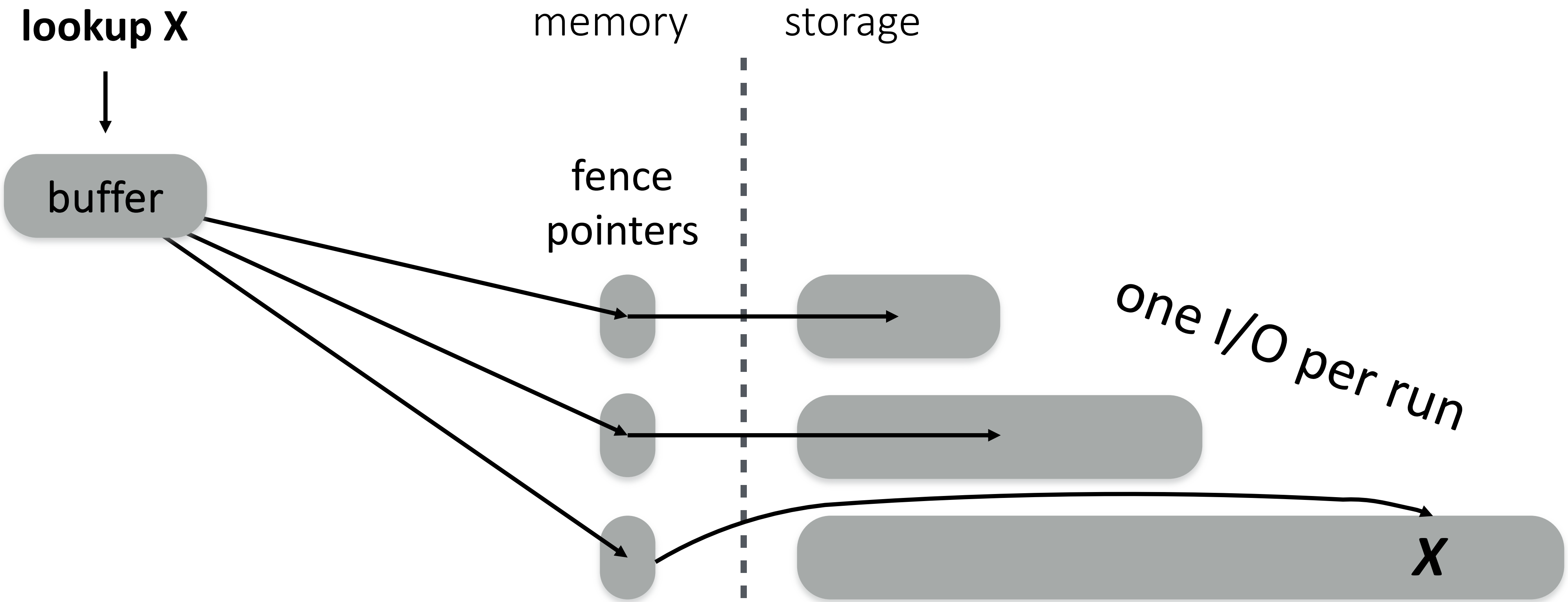
Quick sanity check: suppose $N = 2^{32}$
and $B = 2^{10}$
and $T = 2^{2}$

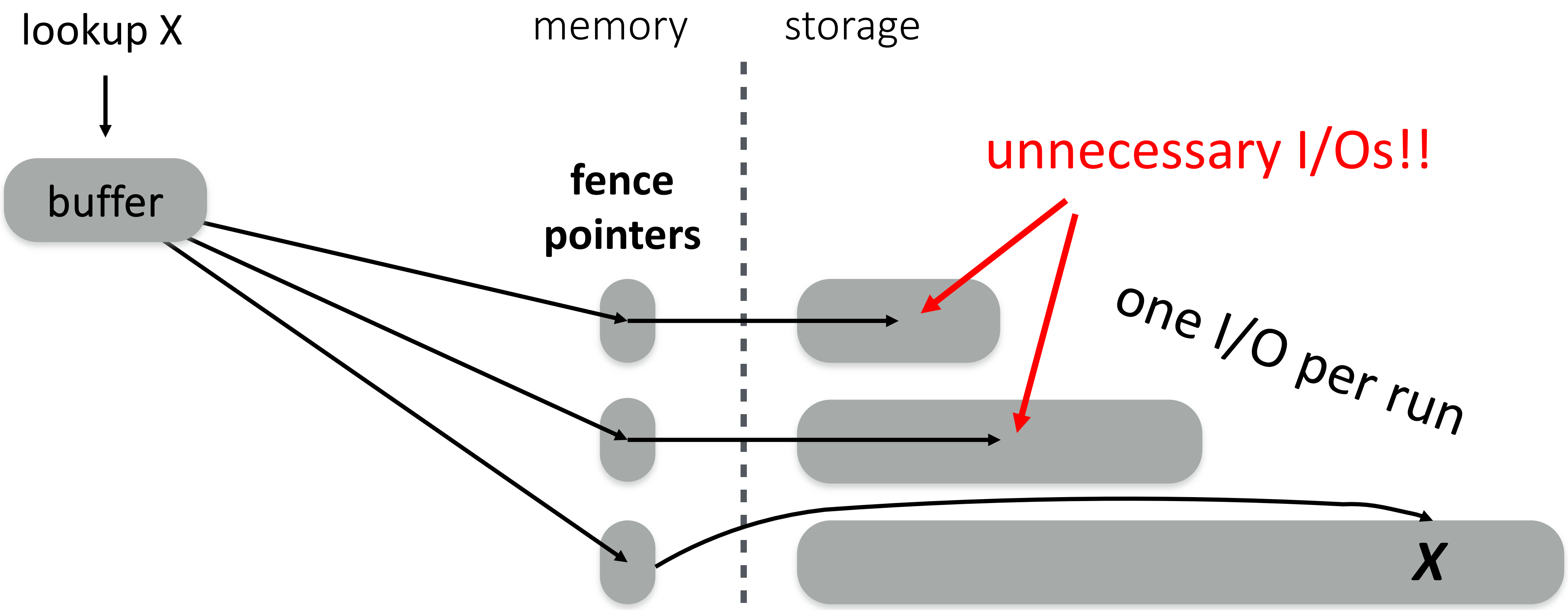
	Lookup cost	Insertion cost
Sorted array	$2^{0}=1$	$2^{31}=2B$
Log	$2^{32}=4B$	$2^{-10}=0.001$
B-tree	$2^{2}=4$	$2^{2}=4$
Basic LSM-tree	$2^{5}=32$	$2^{-5}=0.031$
Leveled LSM-tree	$2^{4}=16$	$2^{-4}=0.063$
Tiered LSM-tree	$2^{6}=64$	$2^{-6}=0.016$

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	$2^{-10} \cdot \log_{10}(2^{32}) = 0.009$


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



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**

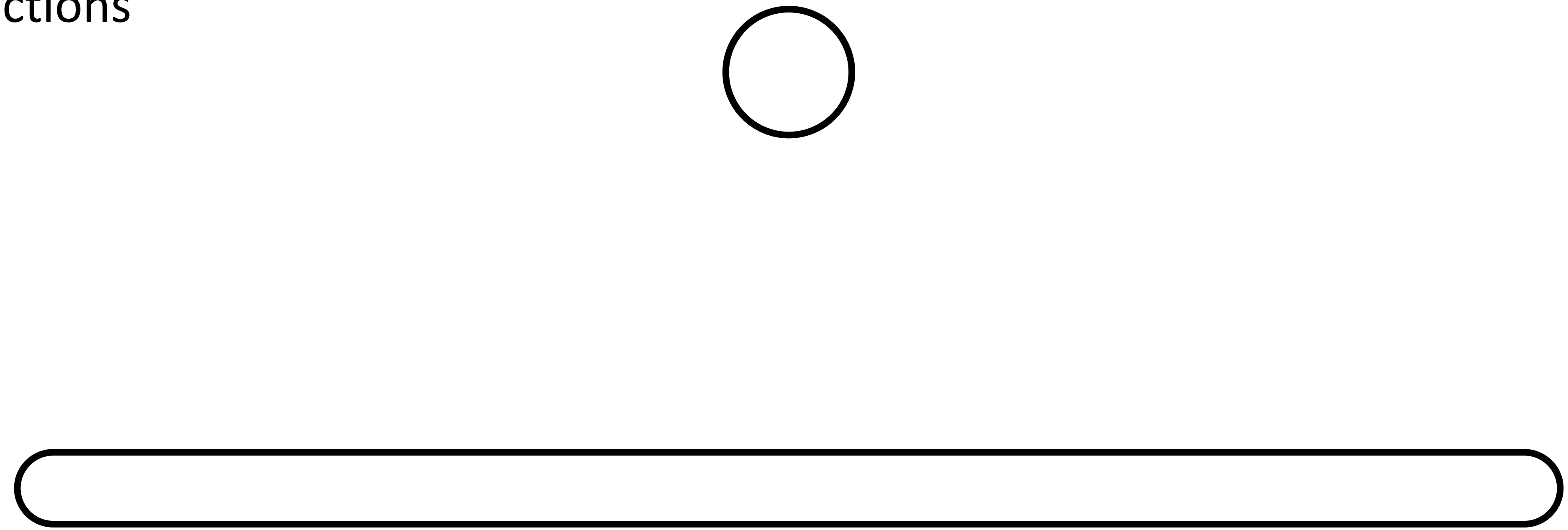
Bloom filters

k hash functions

$h_1(\blacksquare)$

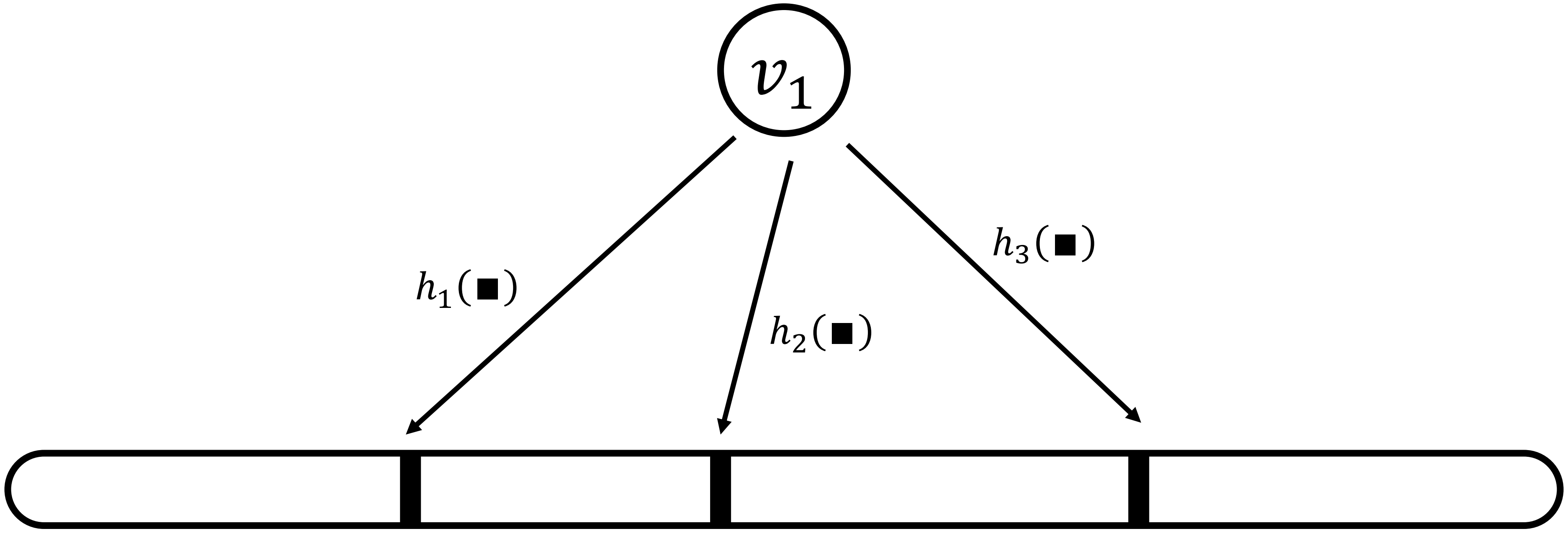
$h_2(\blacksquare)$

$h_3(\blacksquare)$



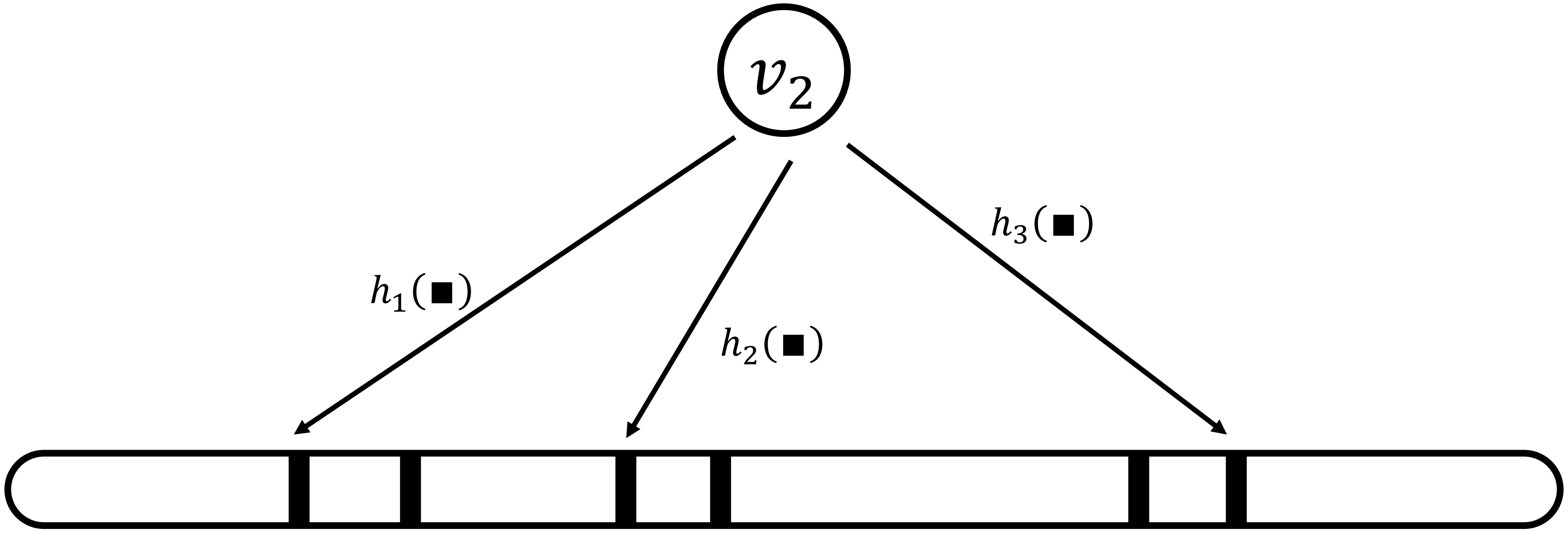
an array of m bits

Bloom filters – insert v_1



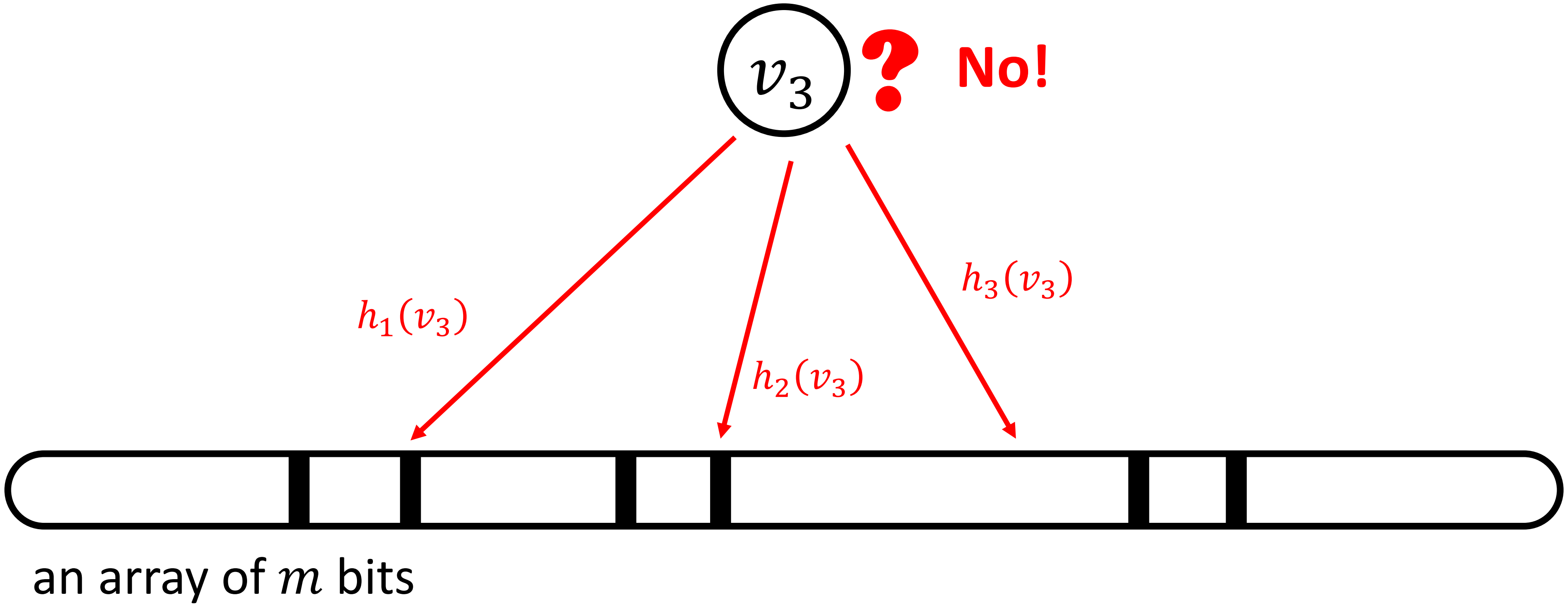
an array of m bits

Bloom filters – insert v_2

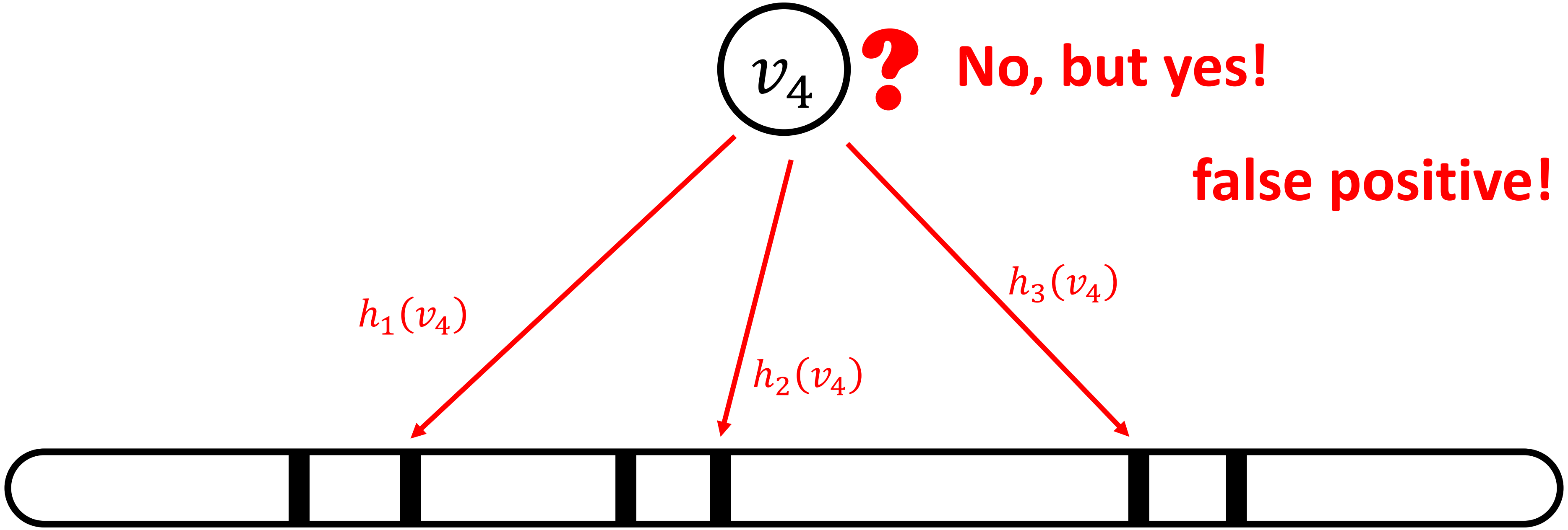


an array of m bits

Bloom filters – query v_3



Bloom filters – query v_4



an array of m bits

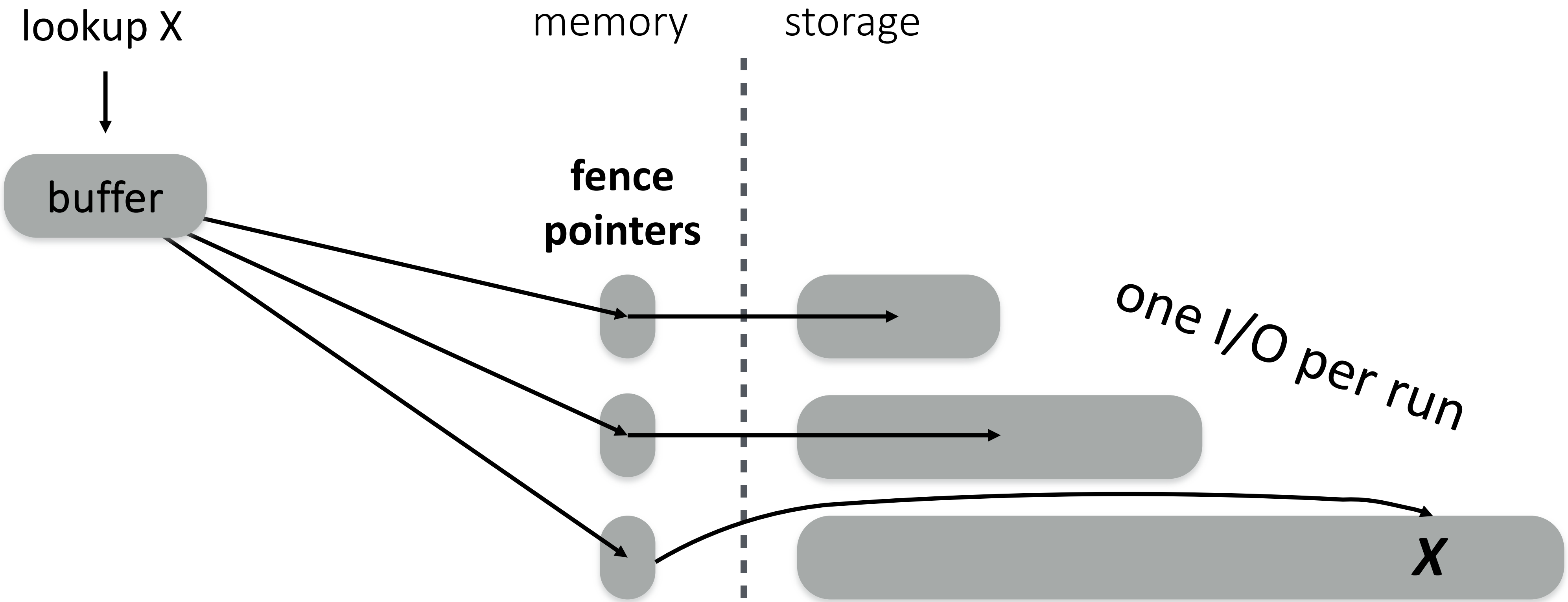
false positive rate: $f = e^{-\frac{m}{n}\cdot(ln(2))^2}$

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

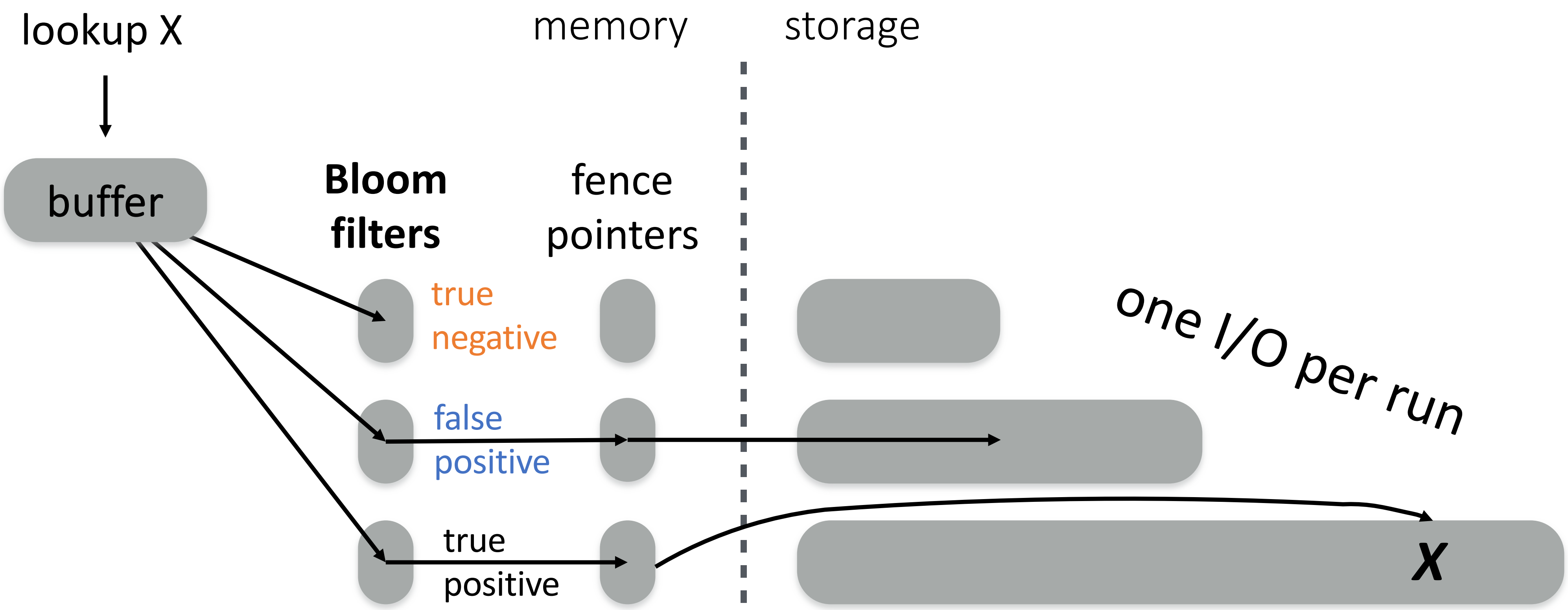
after inserting n elements

→ we have m/n **bits per key**

Augmenting the LSM design with Bloom filters



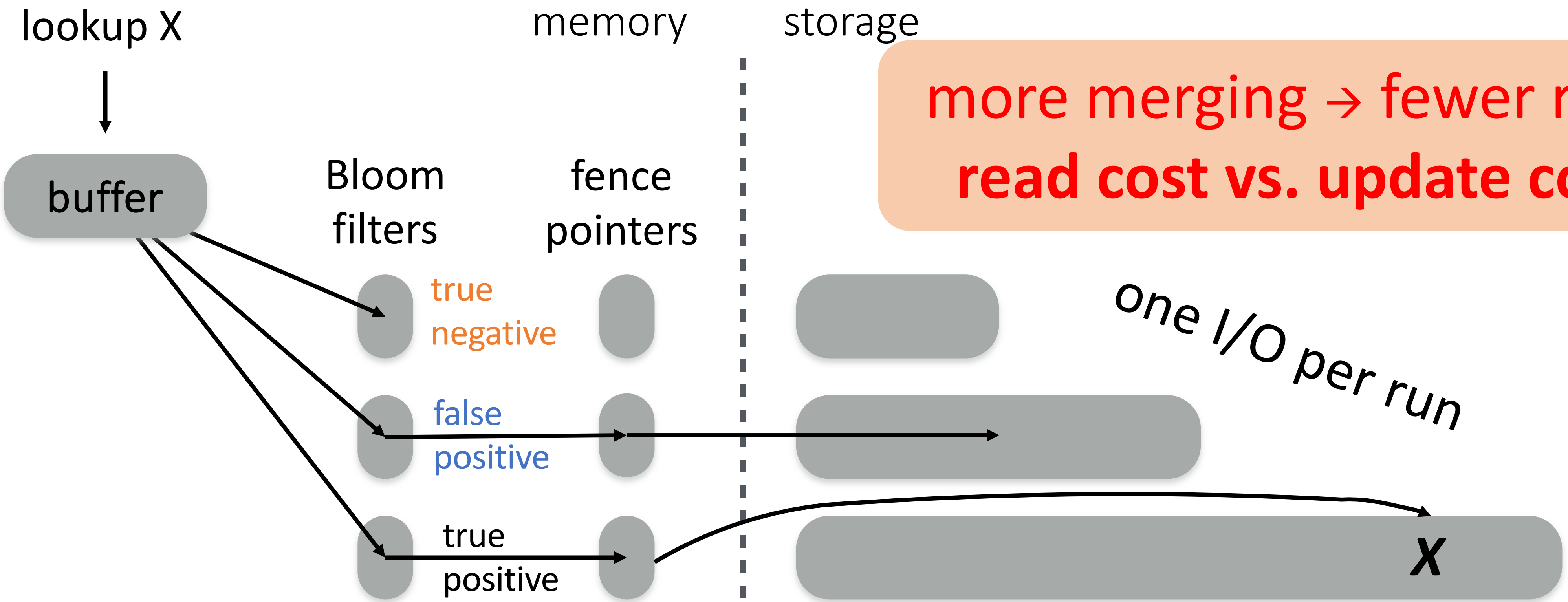
Augmenting the LSM design with Bloom filters



Empty Queries: **only FPs**

Non-Empty Queries: **FPs and one I/O**

performance & cost trade-offs



more merging → fewer runs
read cost vs. update cost

bigger filters → fewer false positives
memory space vs. read cost

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 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	$\mathbf{O(1 + f \cdot \log_T(N))}$	$O(T/B \cdot \log_T(N))$
Tiered LSM-tree	$\mathbf{O(1 + f \cdot T \cdot \log_T(N))}$	$O(1/B \cdot \log_T(N))$

Results Catalogue – with fence pointers & BFs

Quick sanity check: suppose $N = 2^{32}$

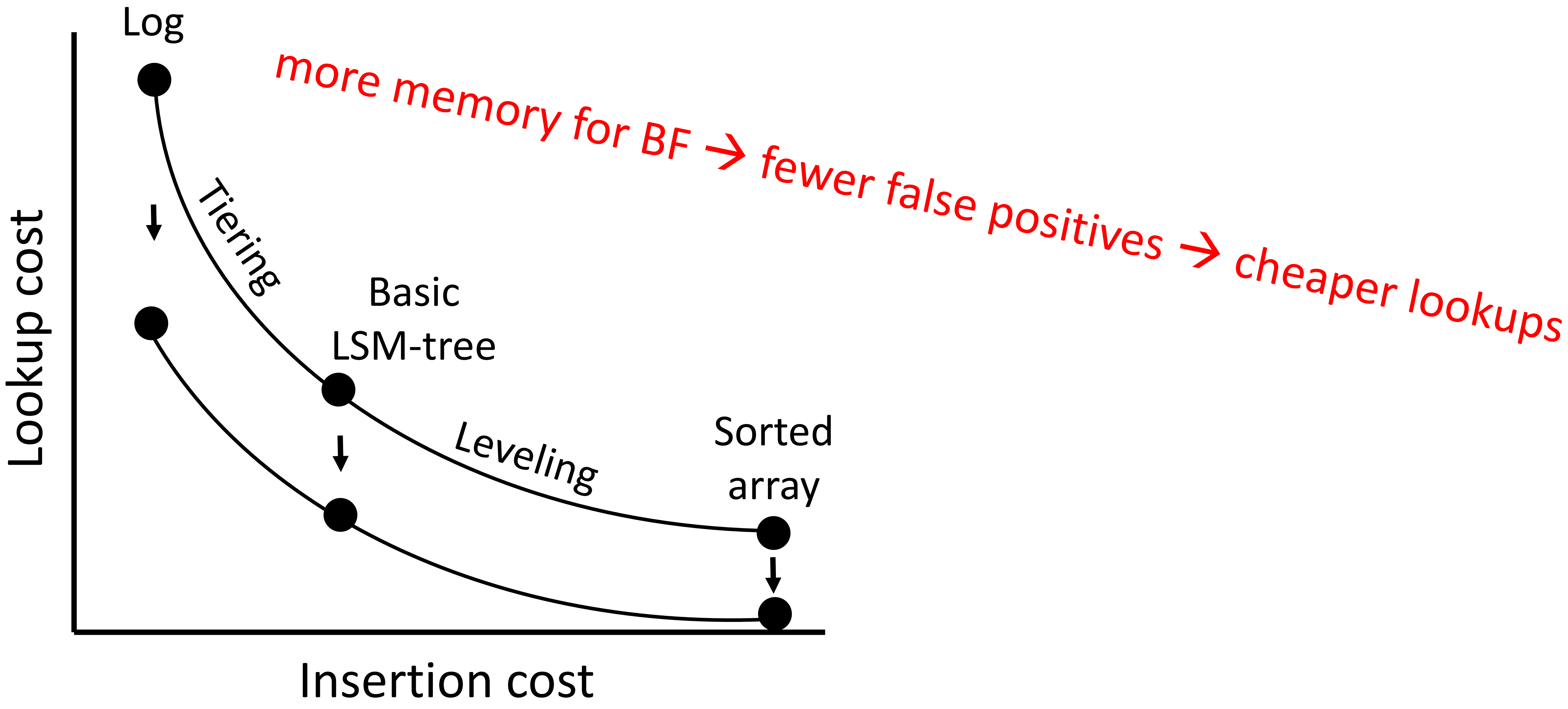
and $B = 2^{10}$

and $T = 10$ 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!