LSM-Trees

Niv Dayan

Project Announcements

Feedback weeks

Oct 6-10 - Step 1

Nov 3-7 - Step 2

Deadline

Dec 1

Project Announcements

Min Expected Contribution: 20%

Struggling in your group? Ask for help.

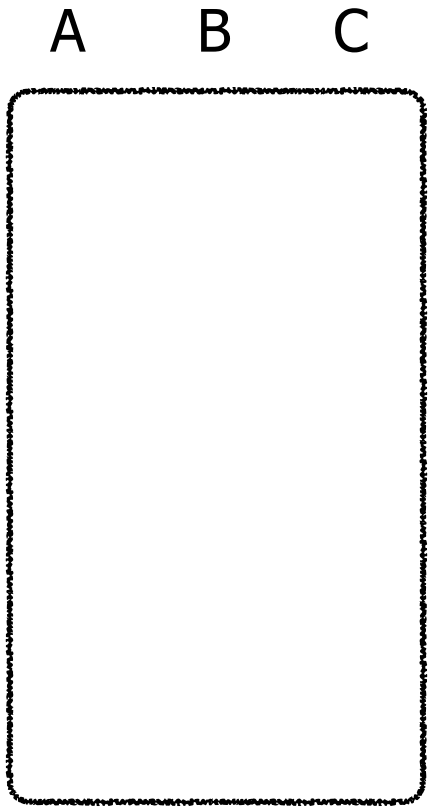
Project Announcements

Min Expected Contribution: 20%

Struggling in your group? Ask for help.

Last resort: splitting

We are trying to optimize the following kinds of queries



select * from table where A = “…”

select * from table where B > “…” and B < “…”

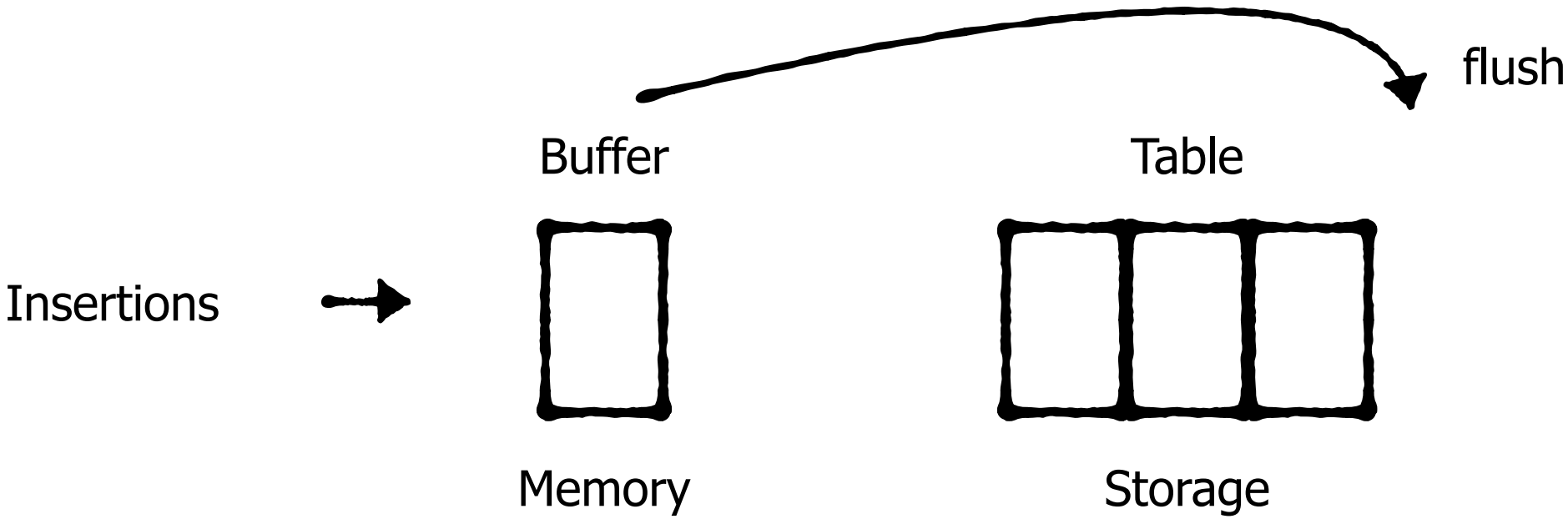
Recap

Append-Only Table | Sorted Table | Extendible Hashing | B-tree

Append-Only Table

O(N/B) for any selection query

O(1/B) for insertions



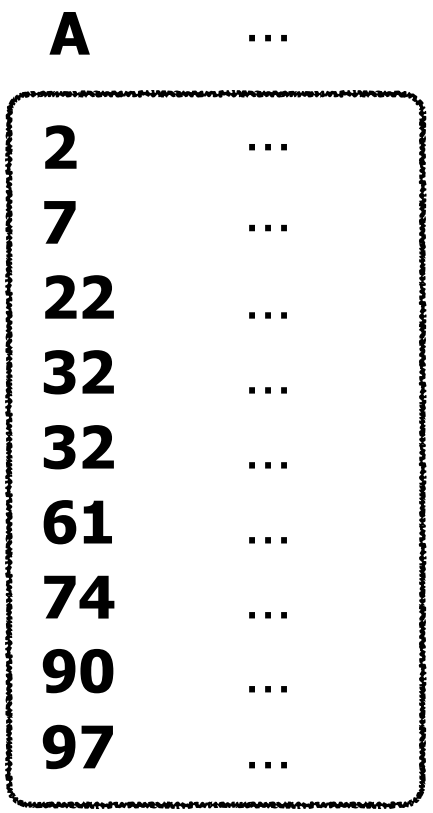
Append-Only Table

Sorted Table

Extendible Hashing

B-tree

Sorted Table



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

Update/insert/delete: O(N/B) read & write I/O

Append-Only Table

Sorted Table

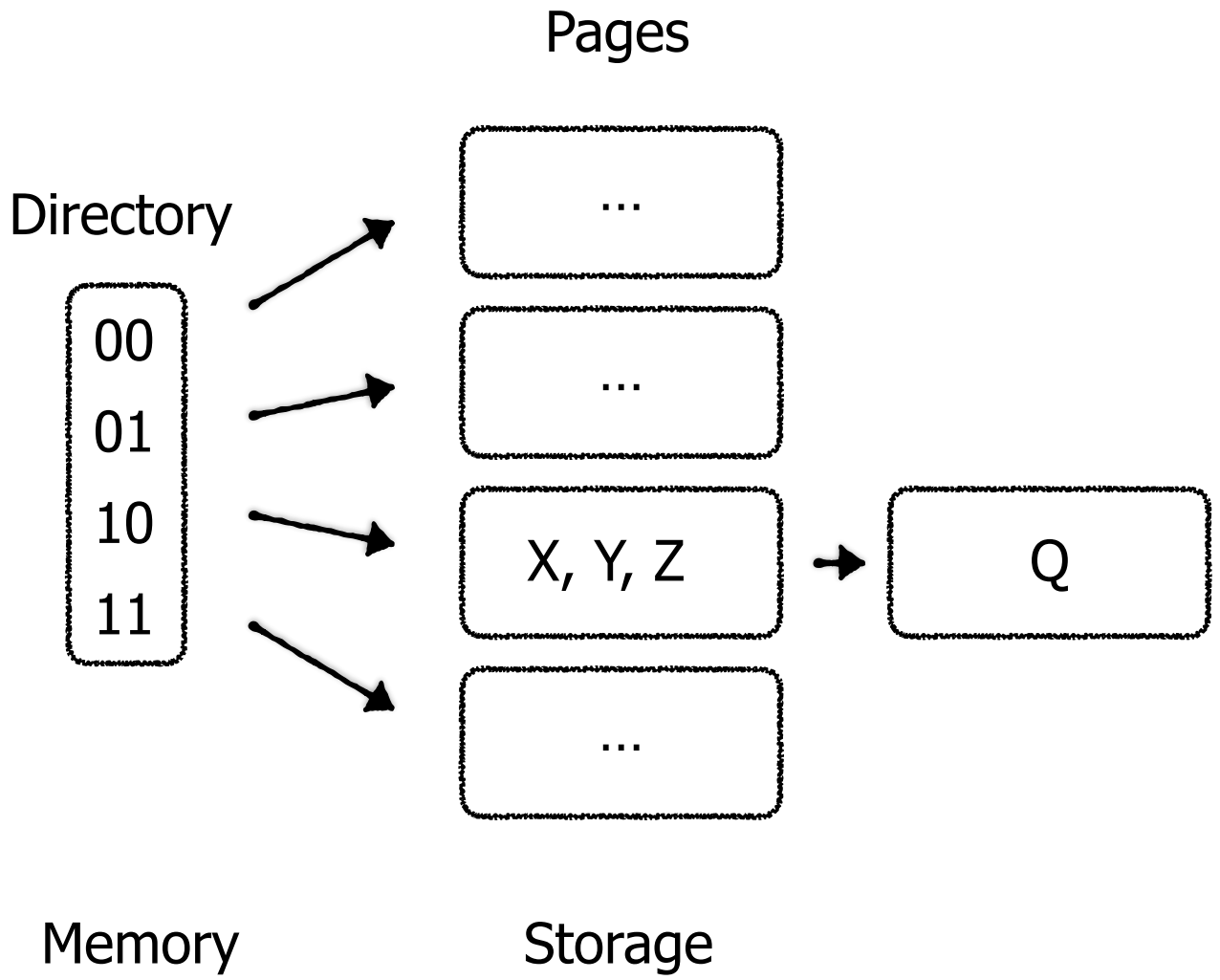
Extendible Hashing

B-tree

Extendible Hashing

A directory maps pages in storage with a given hash prefix

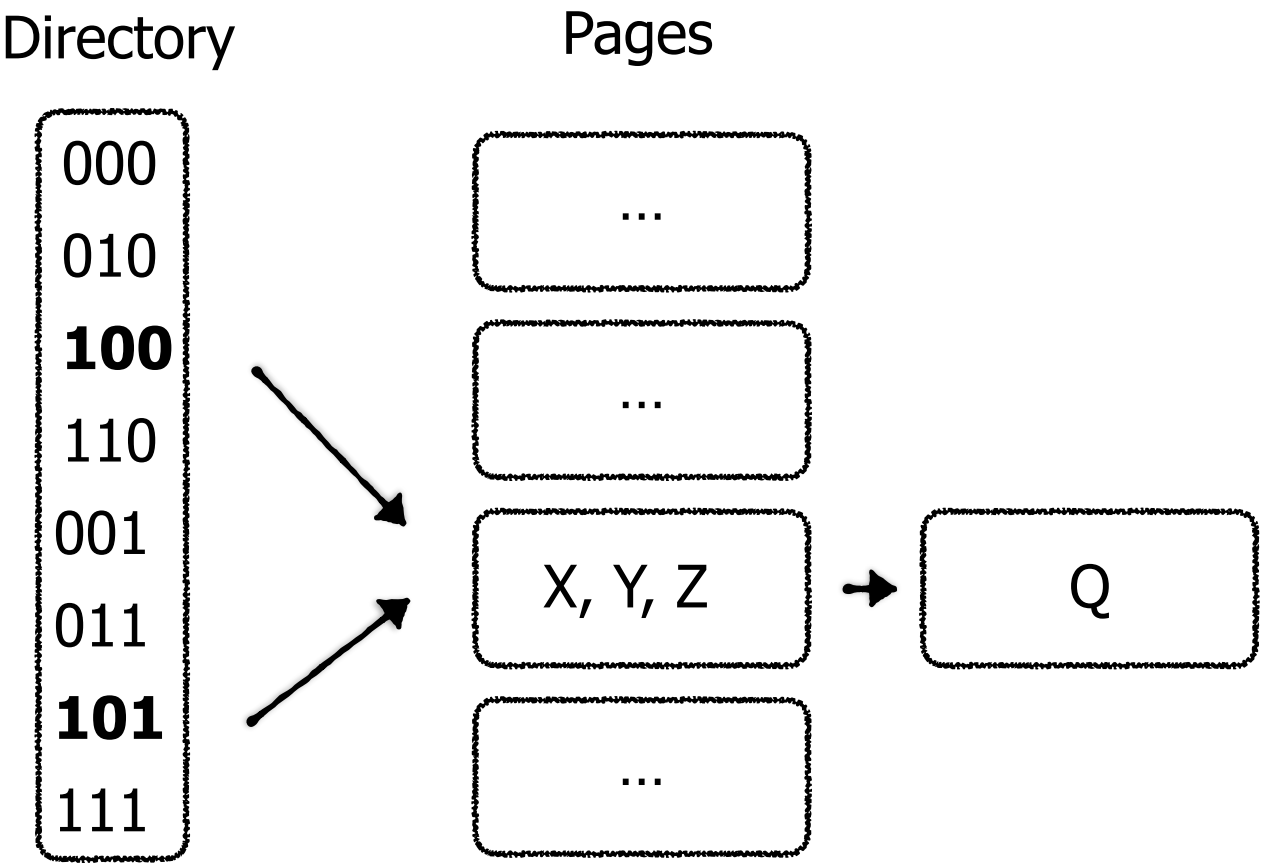
Handle overflows via chaining



Extendible Hashing

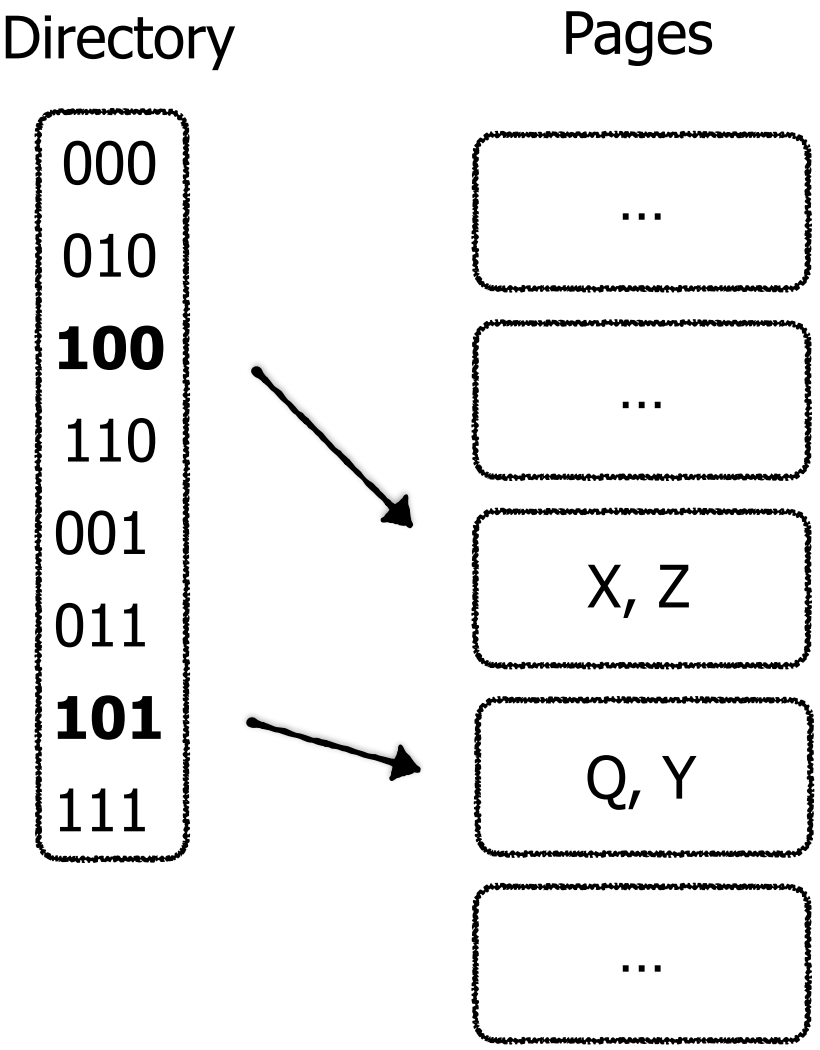
When we reach capacity, double directory size.

New directory slots still point to previous pages



Extendible Hashing

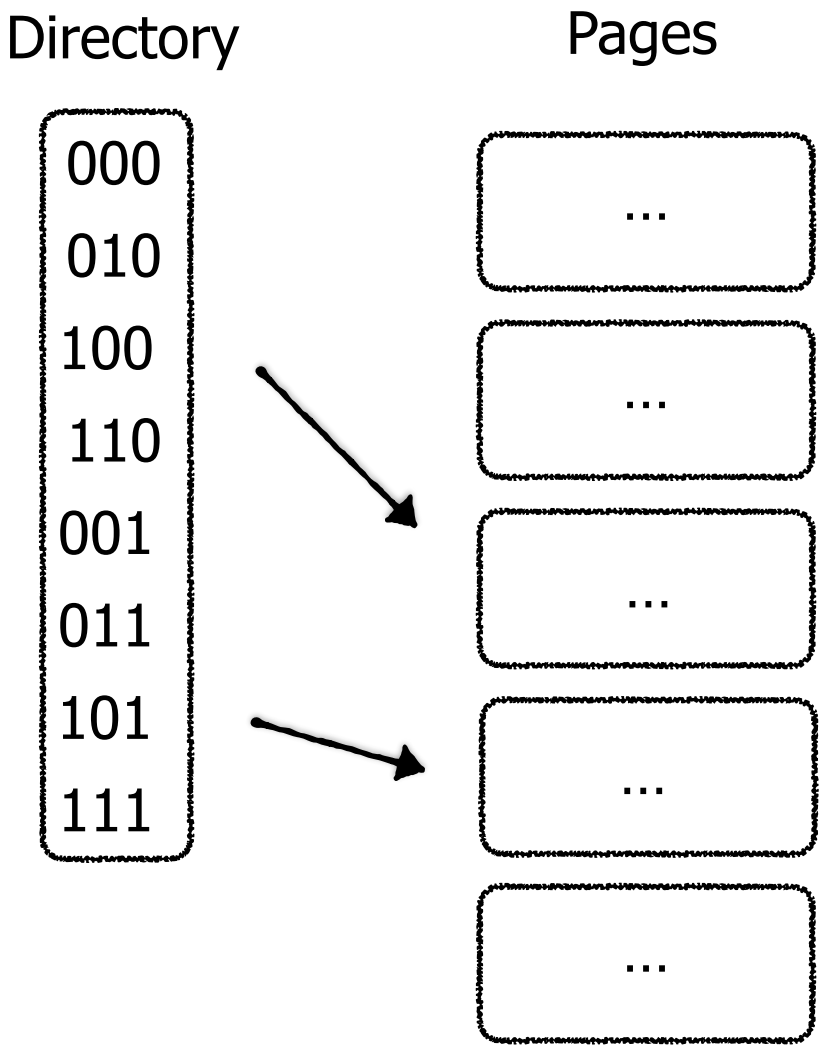
Split one overflowing bucket at a time by rehashing entries.



Extendible Hashing

Query cost: O(1) read I/O

Insertion cost:
- O(1) read I/O
- O(1) write I/O

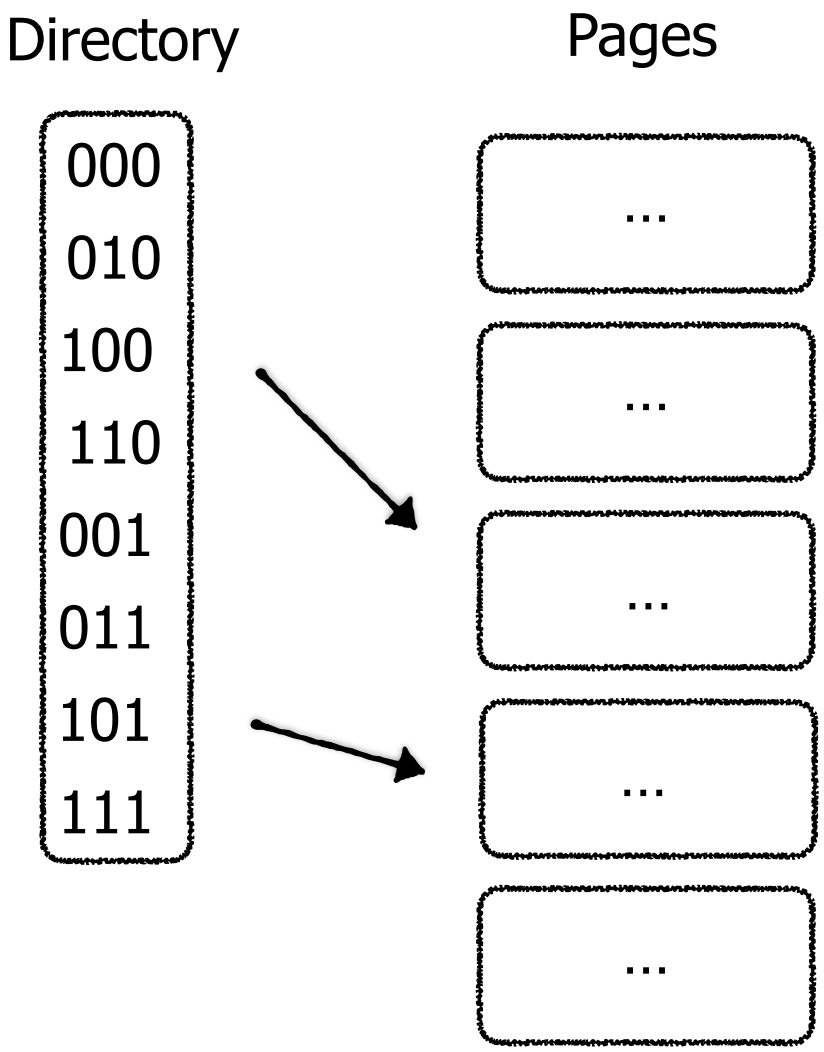


Extendible Hashing

Query cost: O(1) read I/O

Insertion cost:
- O(1) read I/O
- O(1) * **GC** write I/O

Random writes to storage entail SSD garbage-collection



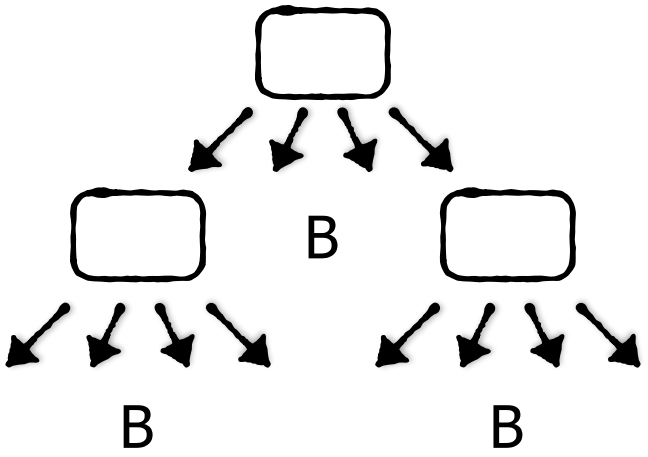
Append-Only Table

Sorted Table

Extendible Hashing

B-Tree

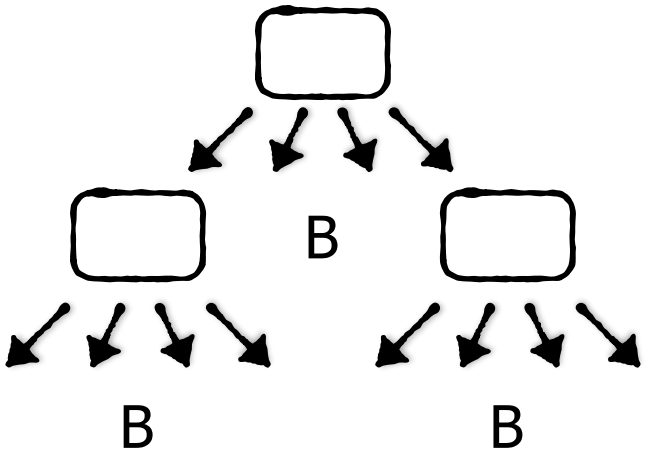
B-Tree



Each node has B children. This allows pruning by a factor of B in each level

Can issue random writes to storage leading to SSD garbage-collection

B-Tree



Each node has B children. This allows pruning by a factor of B in each level

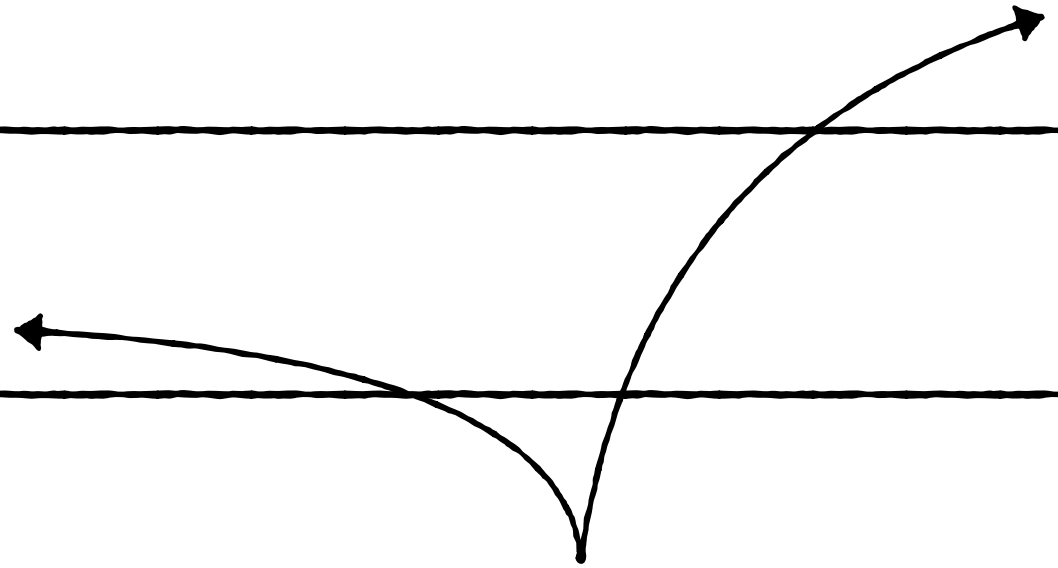
Can issue random writes to storage leading to SSD garbage-collection

Query: $O(\log_B N)$ read I/O

Update/insert/query: $O(\log_B N)$ read I/O
& $O(1)$ * GC write I/O

Cost Metric	Unit	append-only table	Sorted File	Extendible Hashing	B-Tree
Query	Read IOs	O(N/B)	$O(\log_2 N/B)$	O(1)	$O(\log_B N)$
Insert	Read IOs	0	O(N/B)	O(1)	$O(\log_B N)$
	Writes IOs	O(1/B)	O(N/B)	O(1) & GC	O(1) & GC

Cost Metric	Unit	append-only table	B-Tree
Query	Read IOs	O(N/B)	**O($\log_B$ N)**
Insert	Read IOs	0	O($\log_B$ N)
	Writes IOs	**O(1/B)**	O(1) & GC

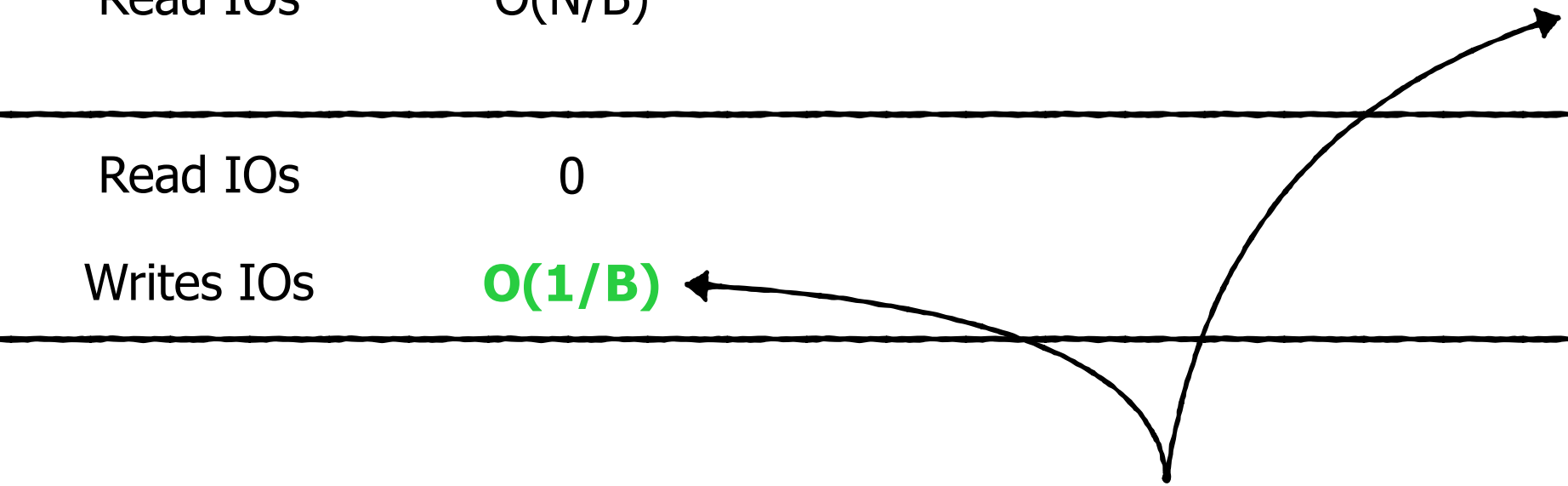


Can we get the best of both worlds?

Typical CS approach: **(1) Identify solutions with diff properties**

(2) Combine to get something in-between

Cost Metric	Unit	append-only table	B-Tree
Query	Read IOs	O(N/B)	**$O(\log_B N)$**
Insert	Read IOs	0	$O(\log_B N)$
	Writes IOs	**O(1/B)**	O(1) & GC



Can we get the best of both worlds?

The Log-Structured Merge-Tree

B-tree is invented — 1970

LSM-tree is invented — 1996

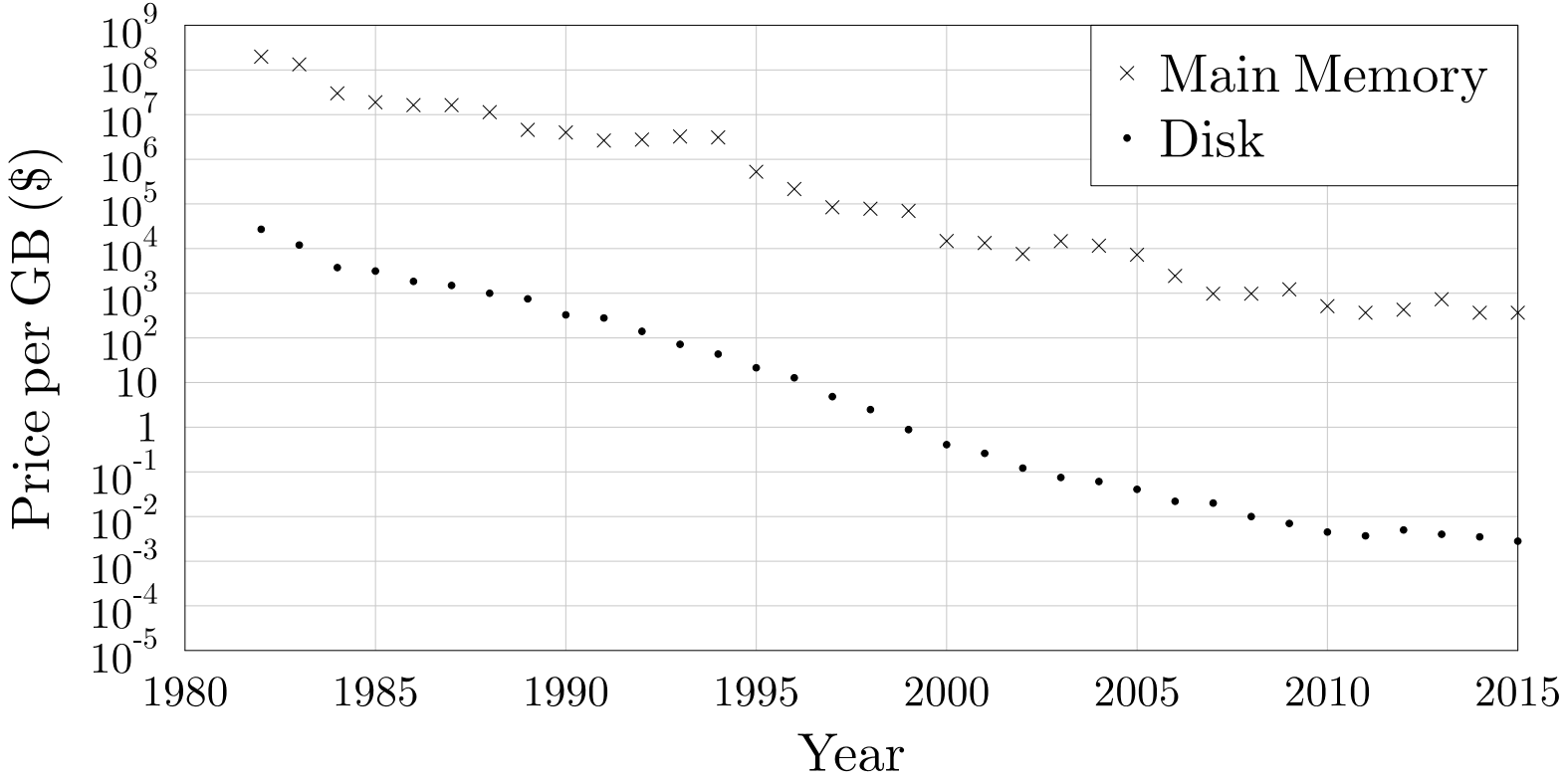
Google's BigTable adopts LSM-tree — 2005

LSM-tree is widely used — Today

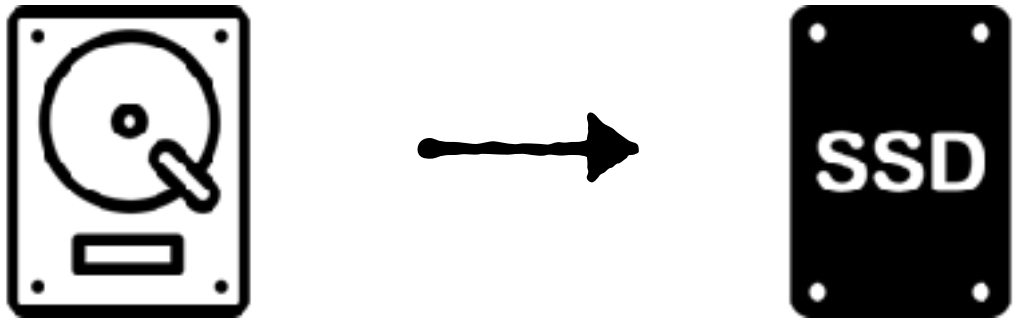
B-tree is invented	LSM-tree is invented	Google's BigTable adopts LSM-tree	LSM-tree is widely used
1970	1996	2005	Today

Why was it not invented and used sooner?

The declining costs of storage allow us to store more data cheaply.
Hence, application workloads are becoming more write-intensive.
This drives a need to optimize for data ingestion.



At the same, the advent of SSDs makes write I/Os more expensive than read I/Os. The drives a need for more write-optimized data structures.

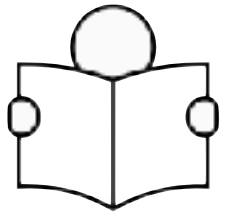


Necessity is the mother of invention

Plato

(Loose attribution)

Leveled LSM-tree



Basic LSM-tree

Tiered LSM-tree

Basic LSM-tree

Level

Buffer

0

Sorted arrays

1

2

3

Basic LSM-tree

Design principle #1: optimize for insertions by buffering

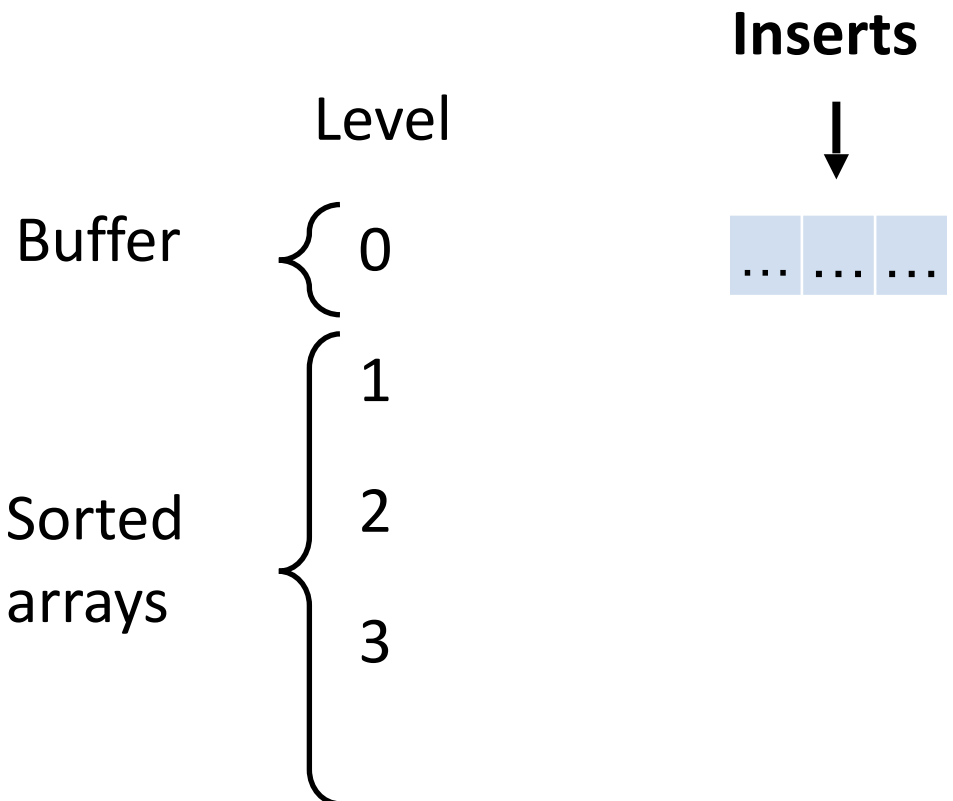
Level

Buffer 0

Sorted arrays 1 2 3

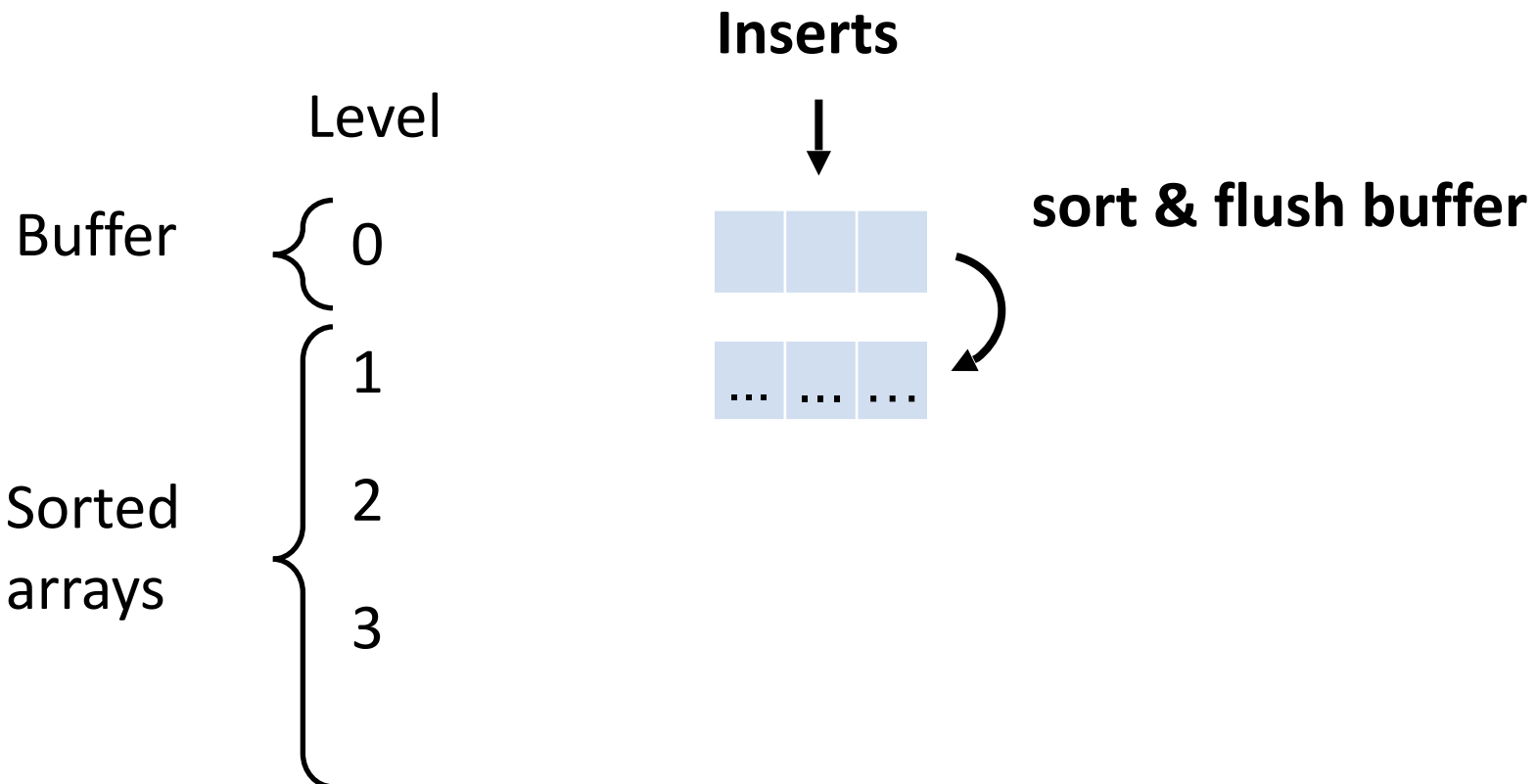
Basic LSM-tree

Design principle #1: optimize for insertions by buffering



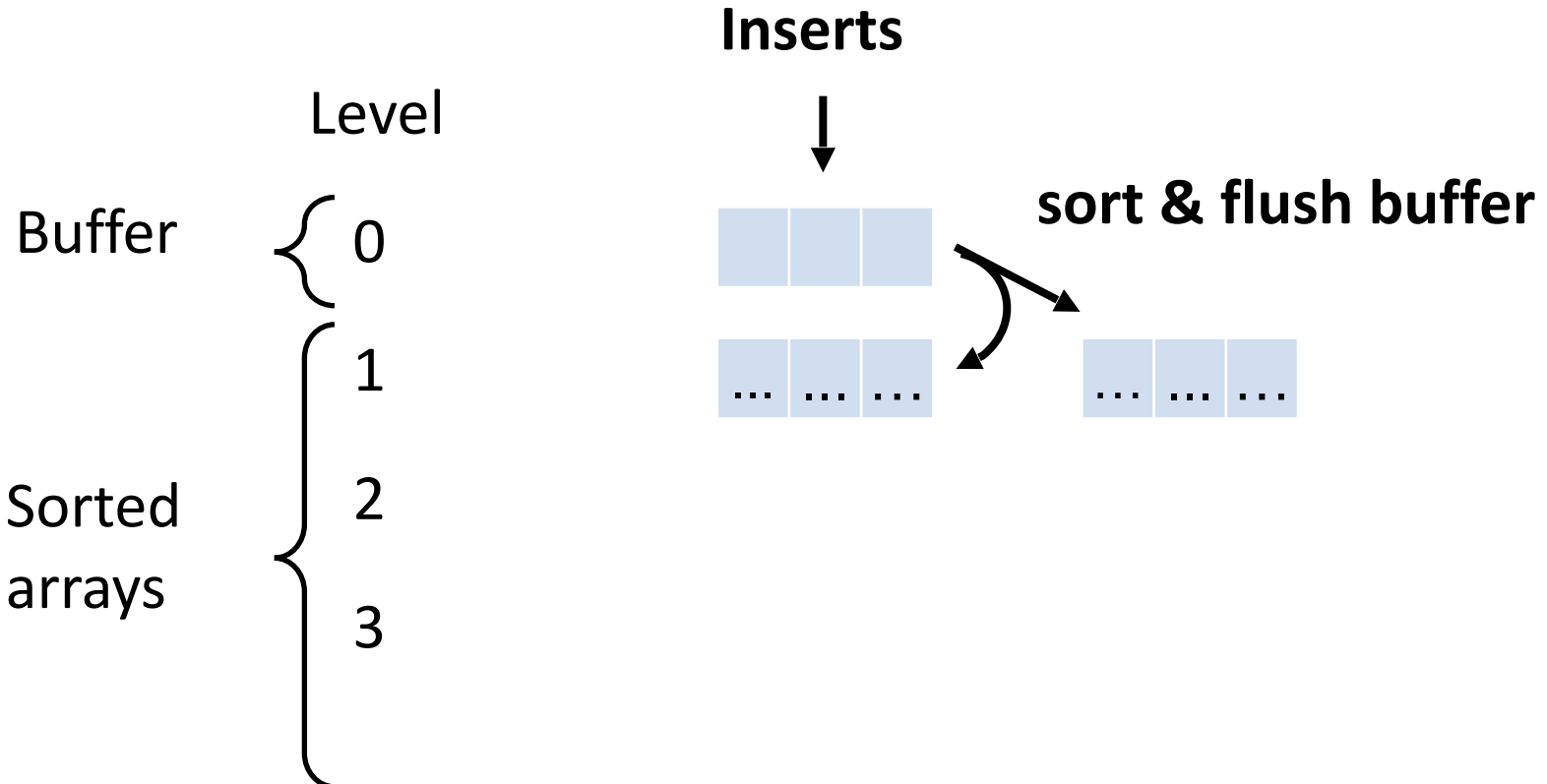
Basic LSM-tree

Design principle #1: optimize for insertions by buffering



Basic LSM-tree

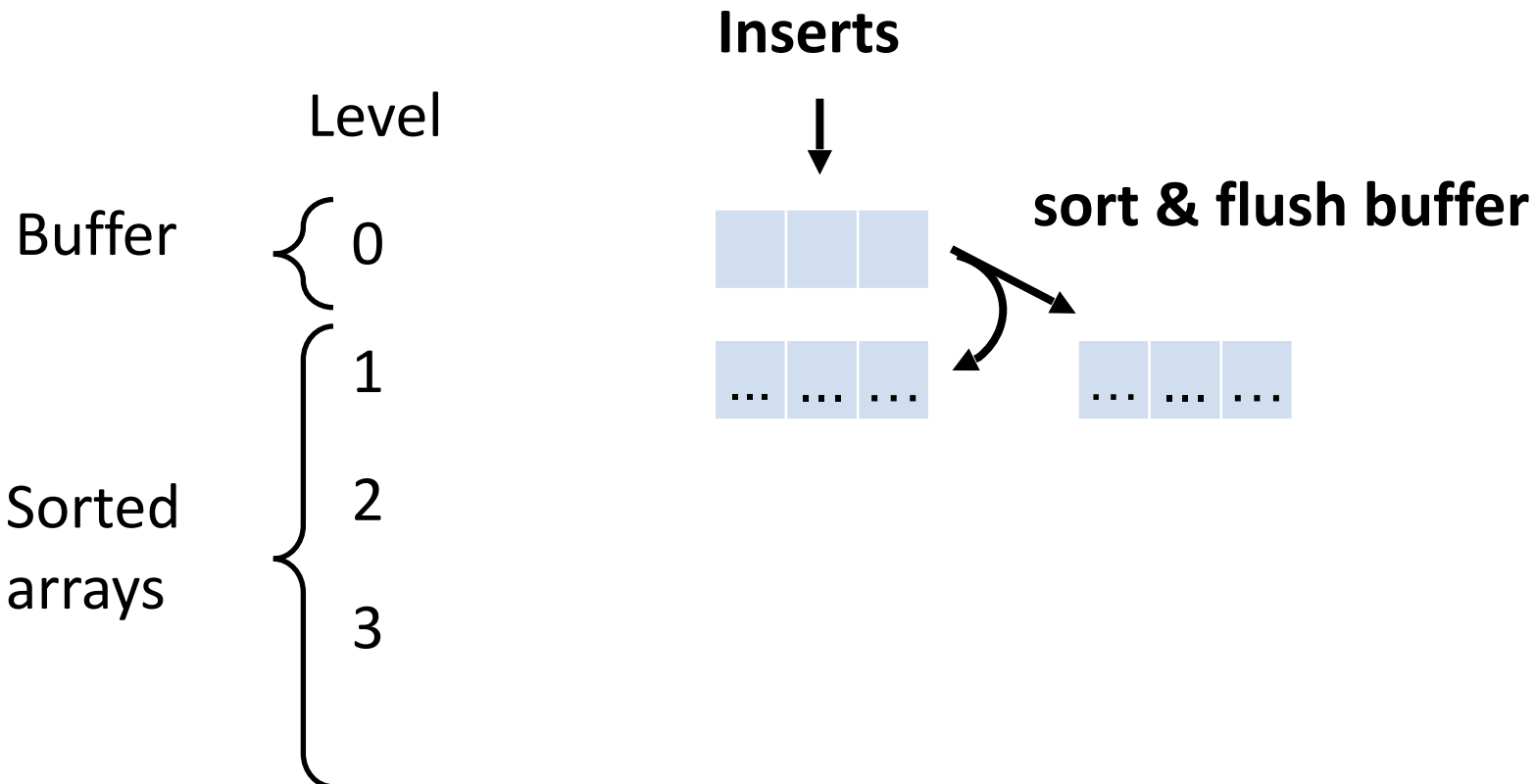
Design principle #1: optimize for insertions by buffering



Basic LSM-tree

Design principle #1: optimize for insertions by buffering

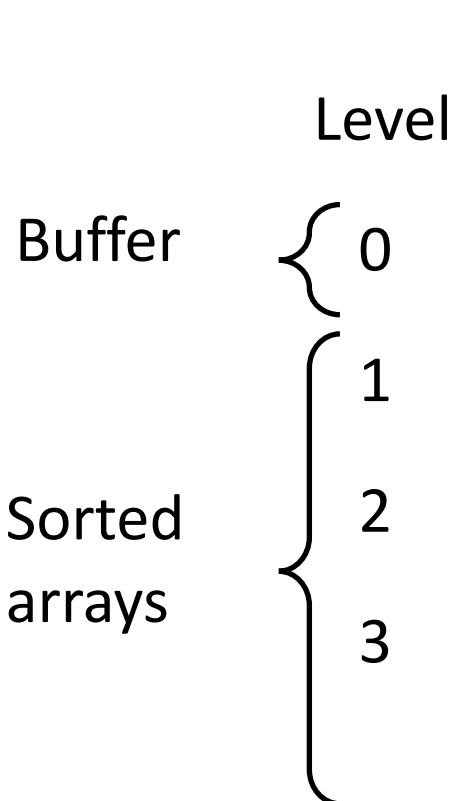
Design principle #2: optimize for lookups by sort-merging SSTs

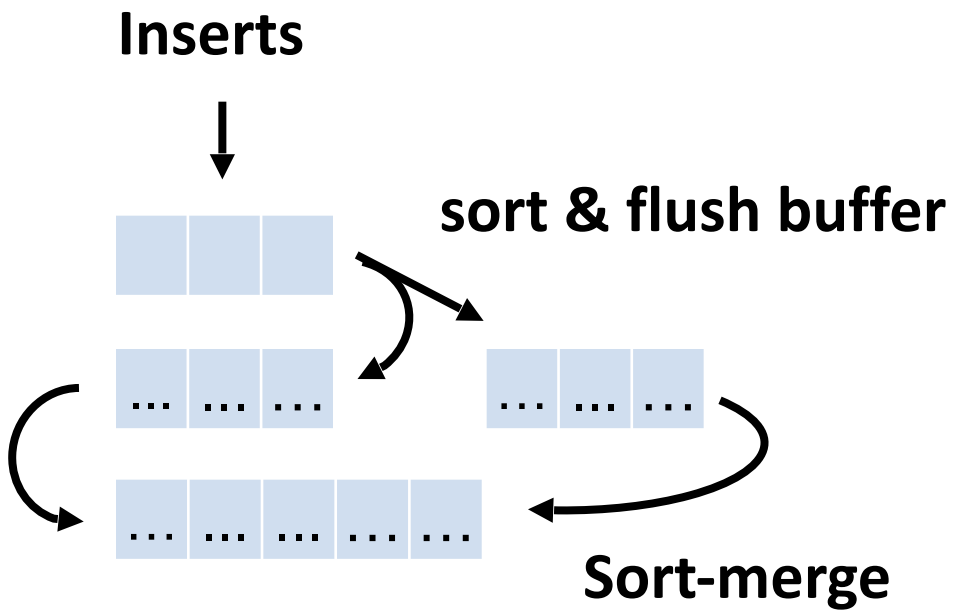


Basic LSM-tree

Design principle #1: optimize for insertions by buffering

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

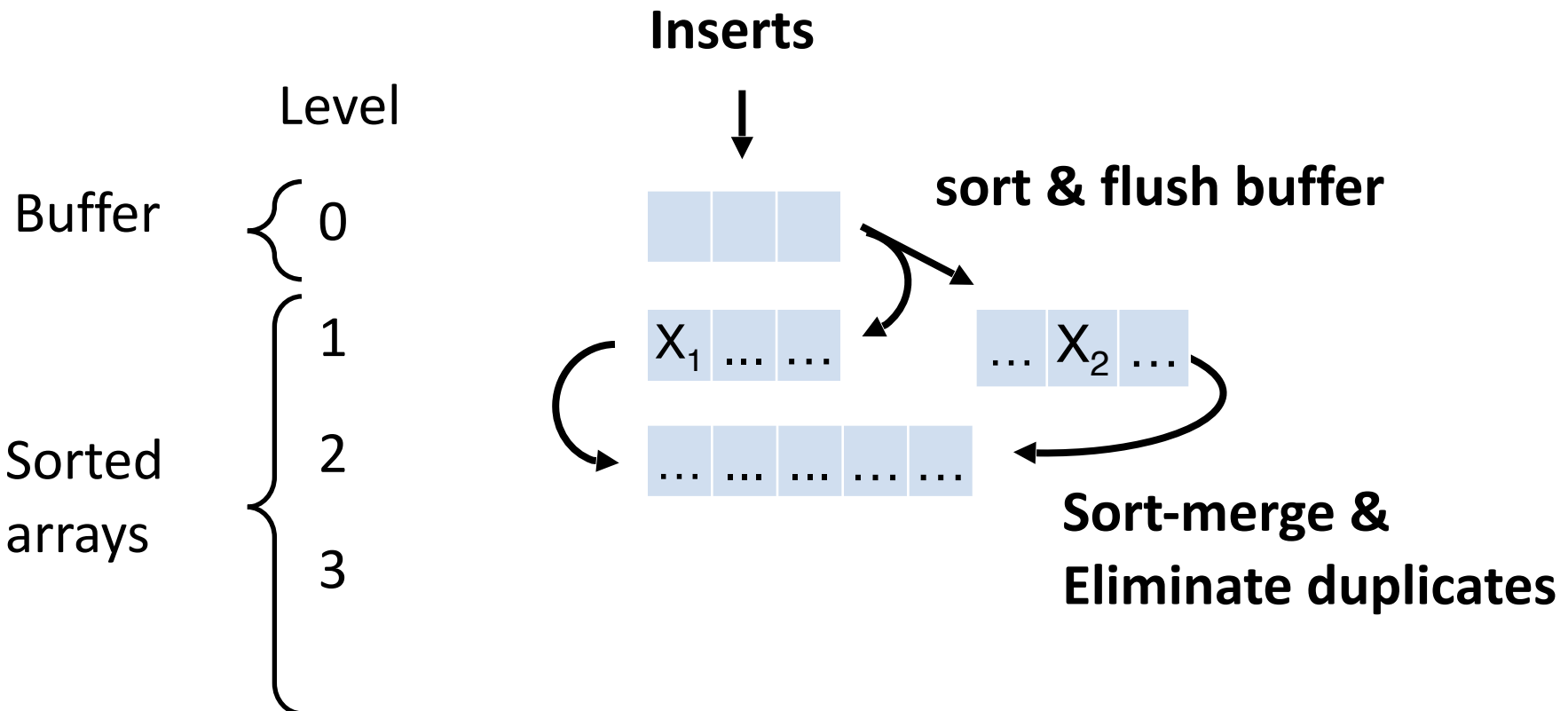




Basic LSM-tree

Design principle #1: optimize for insertions by buffering

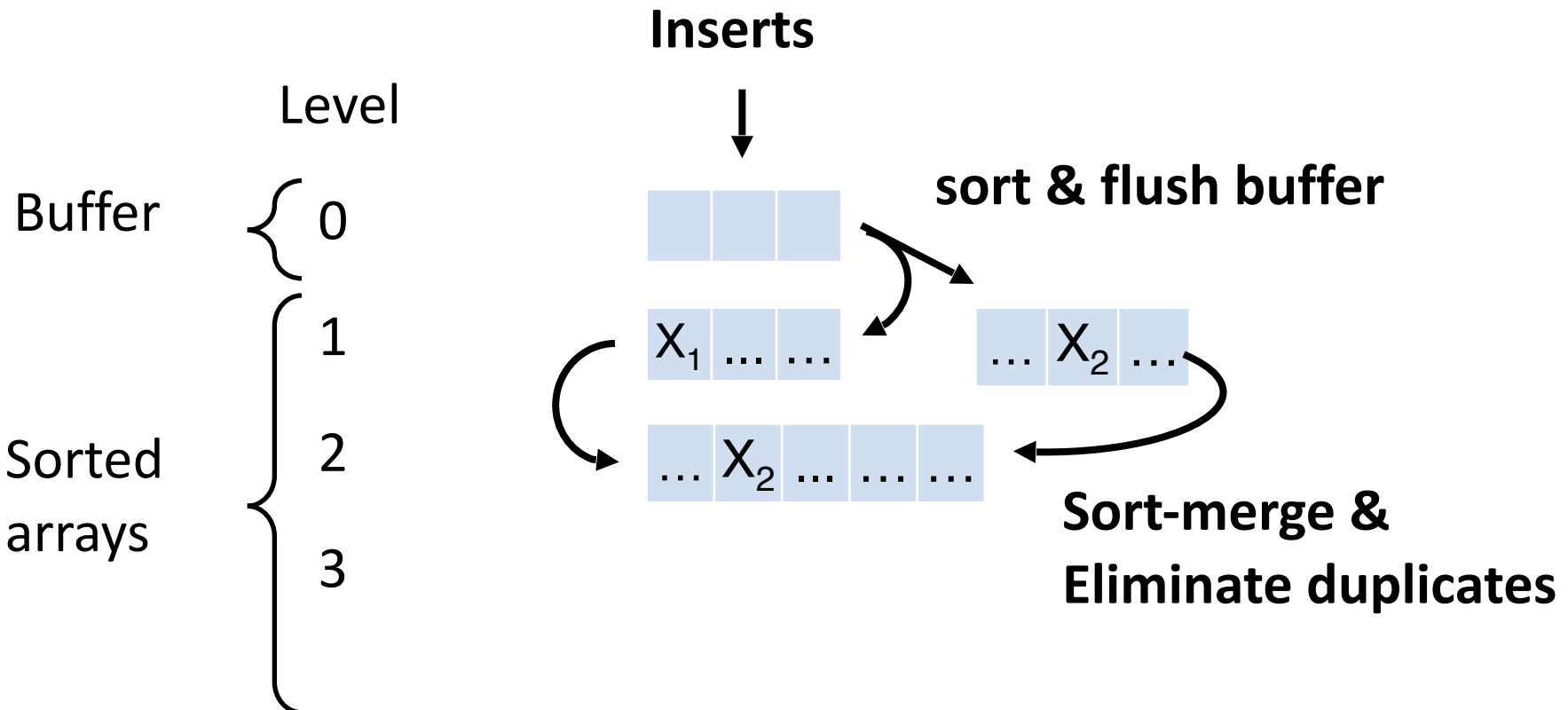
Design principle #2: optimize for lookups by sort-merging SSTs



Basic LSM-tree

Design principle #1: optimize for insertions by buffering

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



Basic LSM-tree

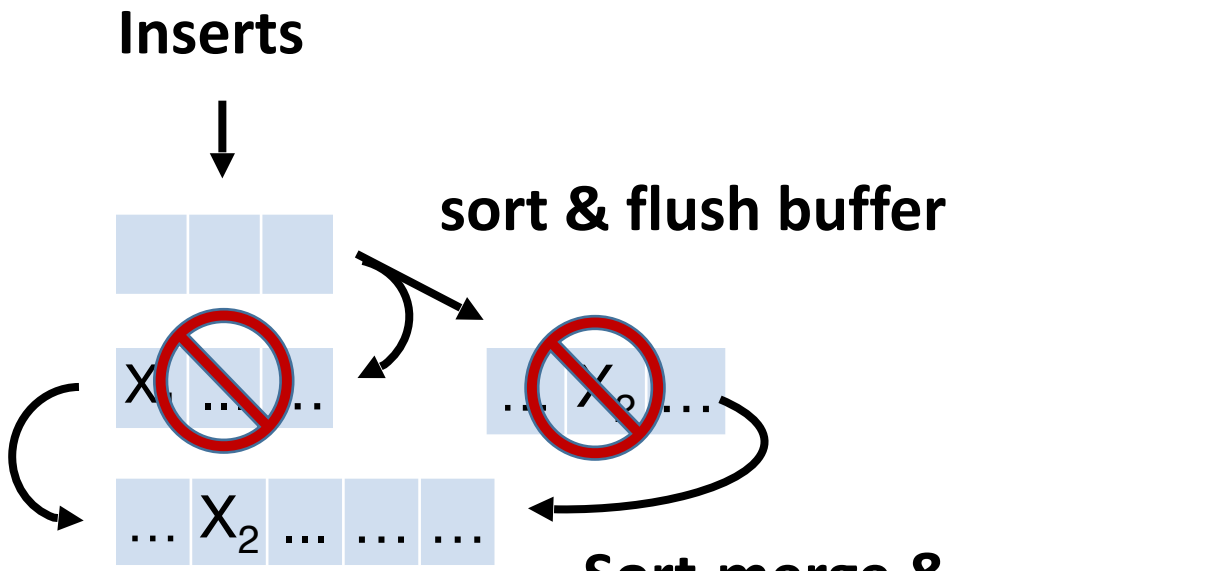
Design principle #1: optimize for insertions by buffering

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

Level

Buffer 0

Sorted arrays 1 2 3



Basic LSM-tree – Example

Buffer

Sorted arrays

Level

0

1

2

3

Basic LSM-tree – Example

Level

Buffer

0

Sorted arrays

1

2

3

inserts

4	6	9

Basic LSM-tree – Example

Buffer

Sorted arrays

Level

0

1

2

3

inserts

4 6 9

sort & flush buffer

Basic LSM-tree – Example

Buffer

Sorted arrays

Level

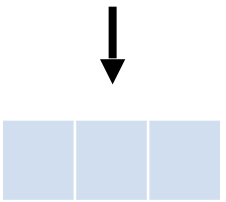
0

1

2

3

inserts



Basic LSM-tree – Example

Level

Buffer

0

Sorted arrays

1

2

3

inserts

3	4	8

4	6	9

Basic LSM-tree – Example

Buffer

Sorted arrays

Level

0

1

2

3

inserts

sort & flush buffer

4 6 9

3 4 8

Basic LSM-tree – Example

Level

Buffer

0

1

2

3

Sorted arrays

inserts

Basic LSM-tree – Example

Buffer

Sorted arrays

Level

0

1

2

3

inserts

4 6 9

3 4 8

3 4 6 8 9

Sort-merge

Basic LSM-tree – Example

Buffer

Sorted arrays

Level

0

1

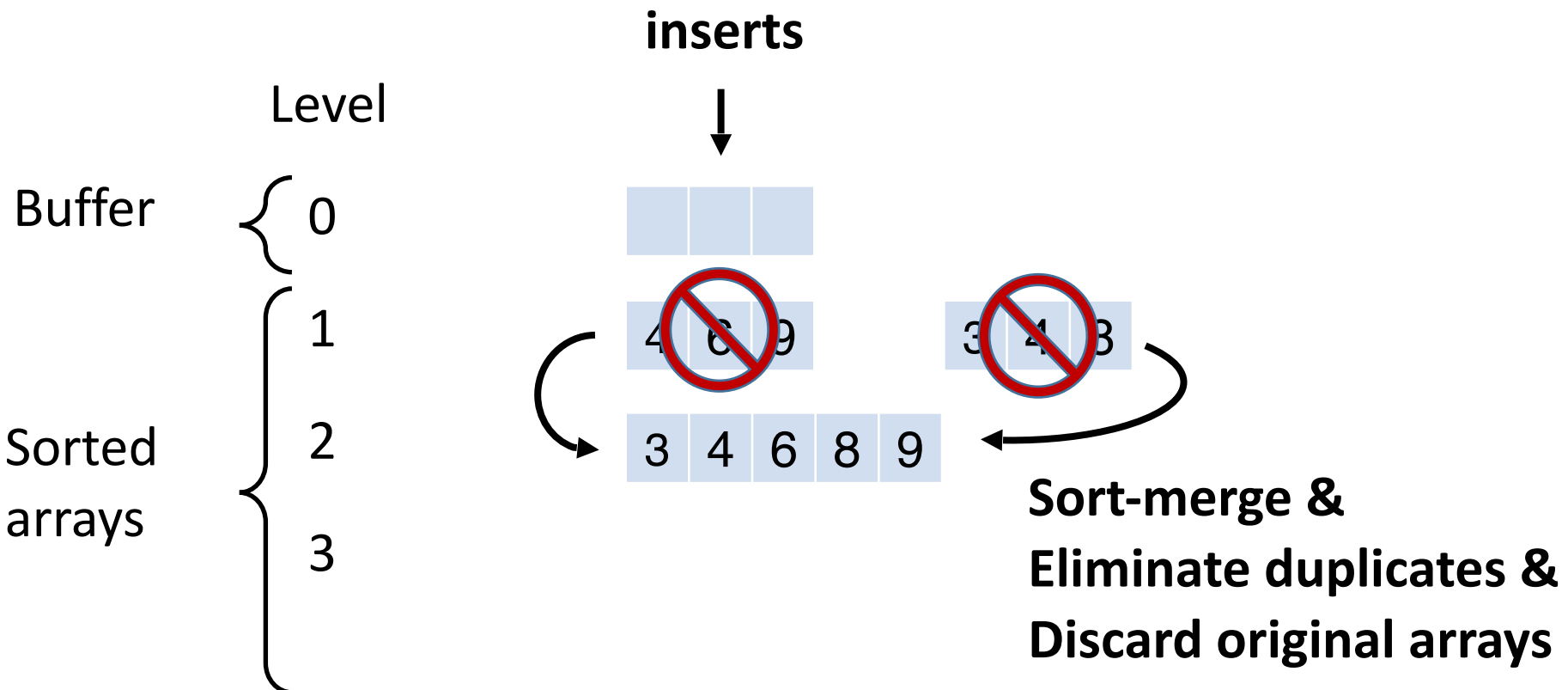
2

3

inserts

Sort-merge &
Eliminate duplicates

Basic LSM-tree – Example



Basic LSM-tree – Example

Level

Buffer

0

Sorted arrays

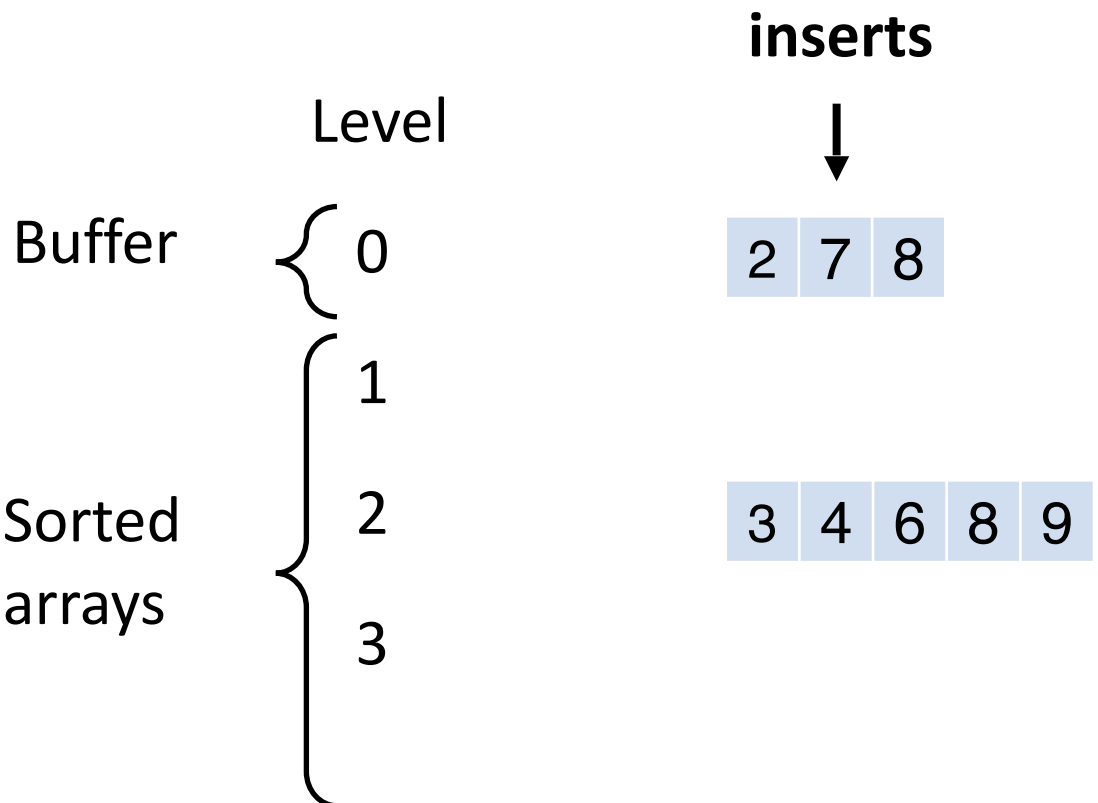
1

2

3

inserts

Basic LSM-tree – Example



Basic LSM-tree – Example

Buffer

Sorted arrays

Level

0

1

2

3

inserts

sort & flush buffer

2 7 8

3 4 6 8 9

Basic LSM-tree – Example

Buffer

Sorted arrays

Level

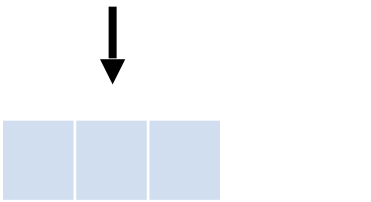
0

1

2

3

inserts



Levels have exponentially increasing capacities.

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

An updates is made out-of-place through an insertion into the buffer

An updates is made out-of-place through an insertion into the buffer

Only an entry's most recent version should be returned to the user during a query

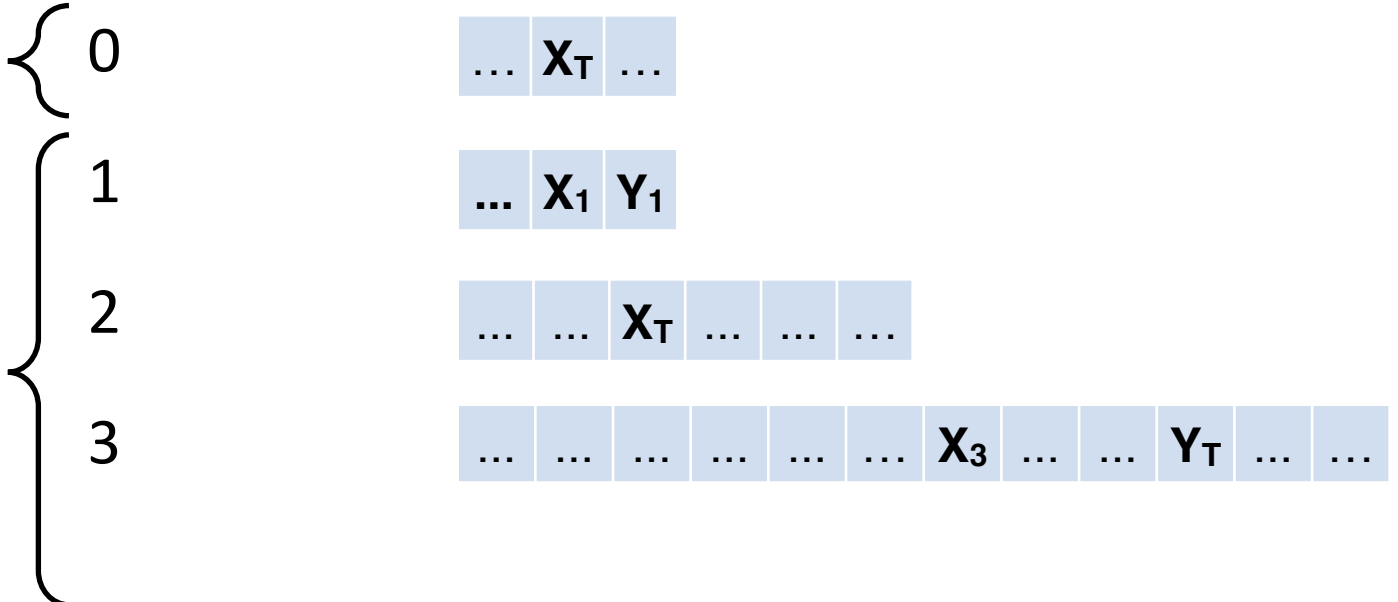
How to handle deletes?

	Level		Capacity
Buffer	0	… **x_0** …	1
Sorted arrays	1	… x_1 …	2
	2	… … x_2 … … …	4
	3	… … … … … … x_3 … … … … …	8

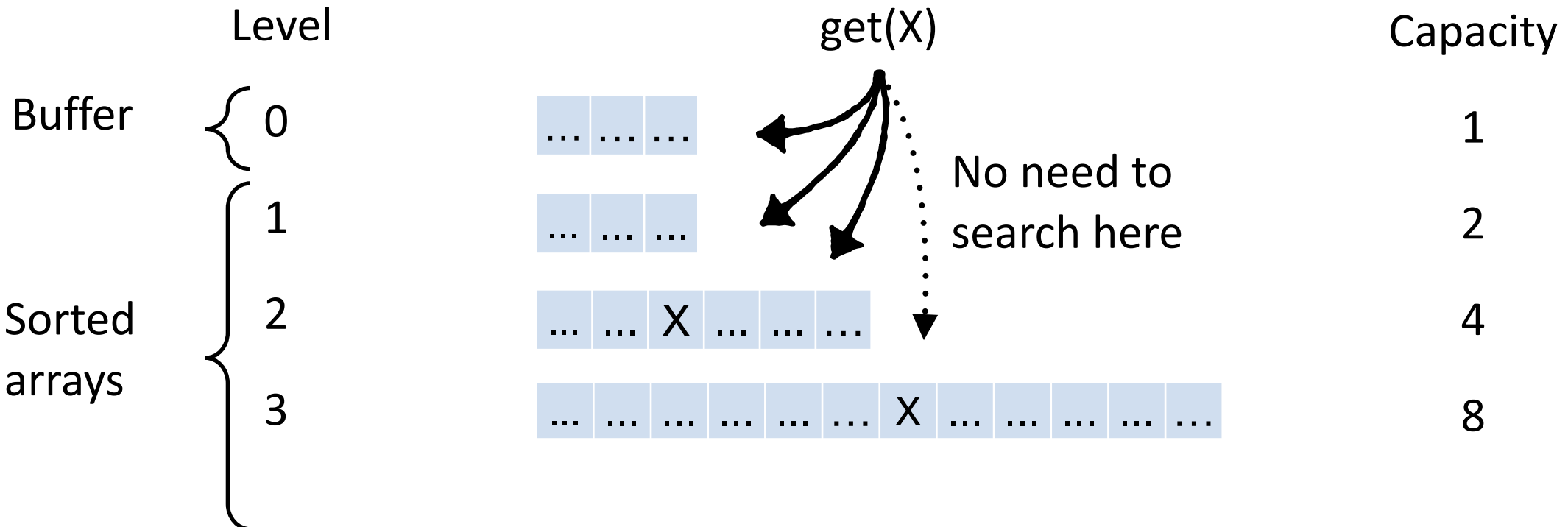
Delete an entry out-of-place by inserting a tombstone

Delete an entry out-of-place by inserting a tombstone

If the most recent version of an entry is a tombstone, it's considered deleted. (X is deleted but Y isn't in this example)



A point query searches the LSM-tree from smaller to larger levels. It stops when it finds the first entry with a matching key. Entries with a matching key at larger levels are older and thus superseded.



How many levels to search?

Buffer

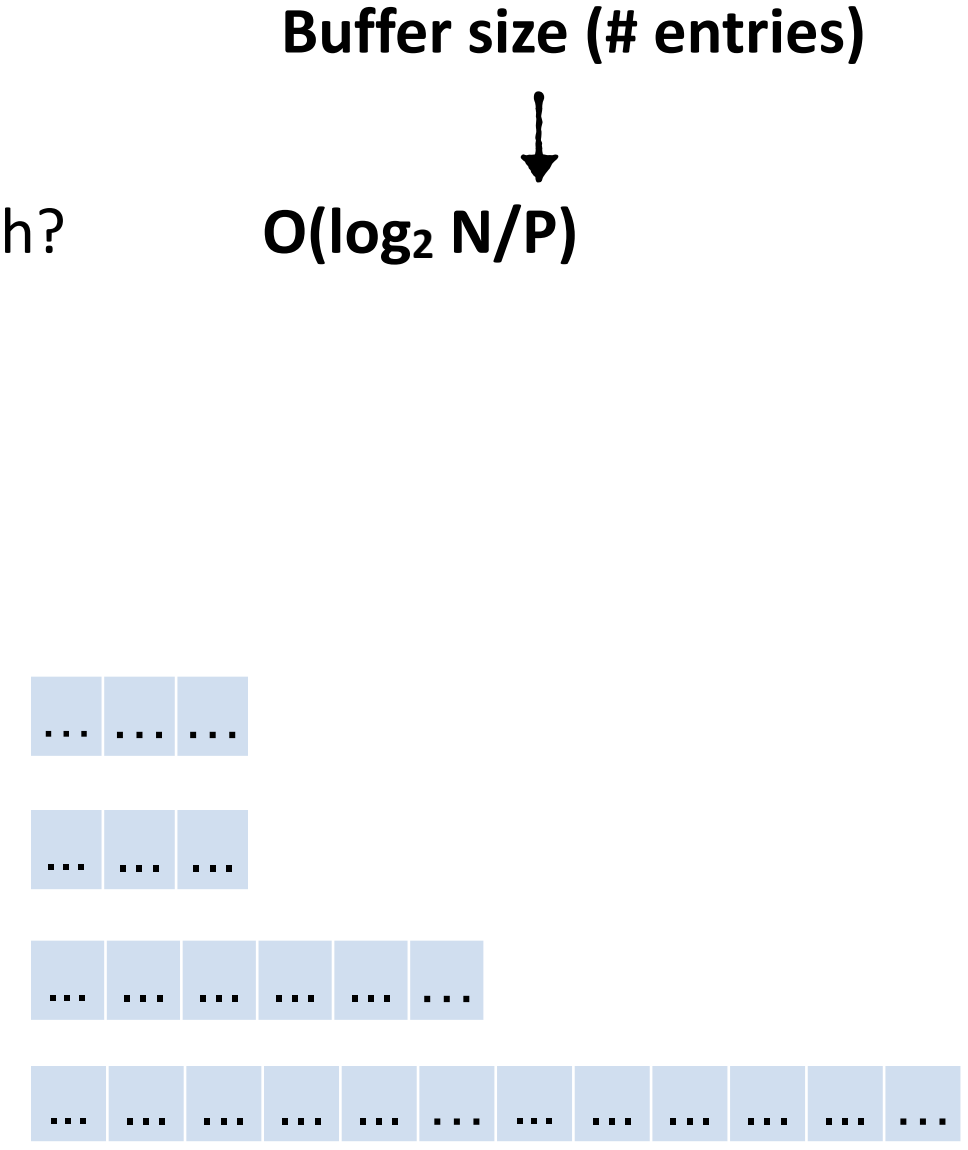
Sorted arrays

Level		Capacity
0	… … …	1
1	… … …	2
2	… … … … … …	4
3	… … … … … … … … … … … …	8

Buffer size (# entries)

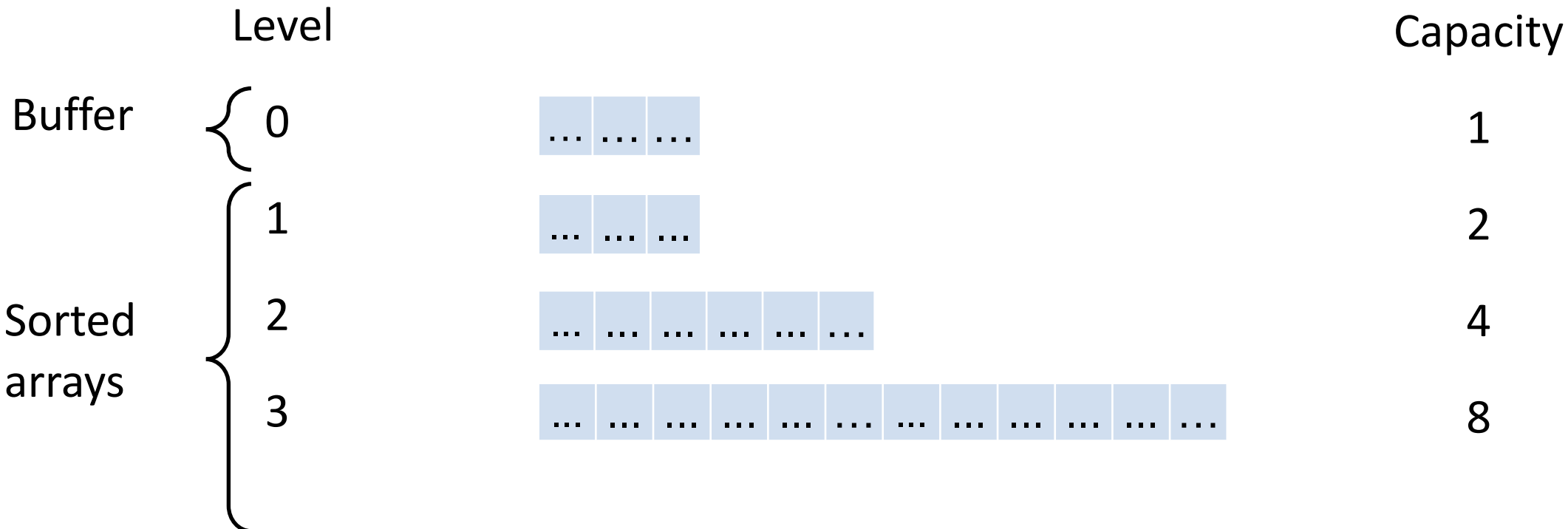
↓

How many levels to search? **$O(\log_2 N/P)$**



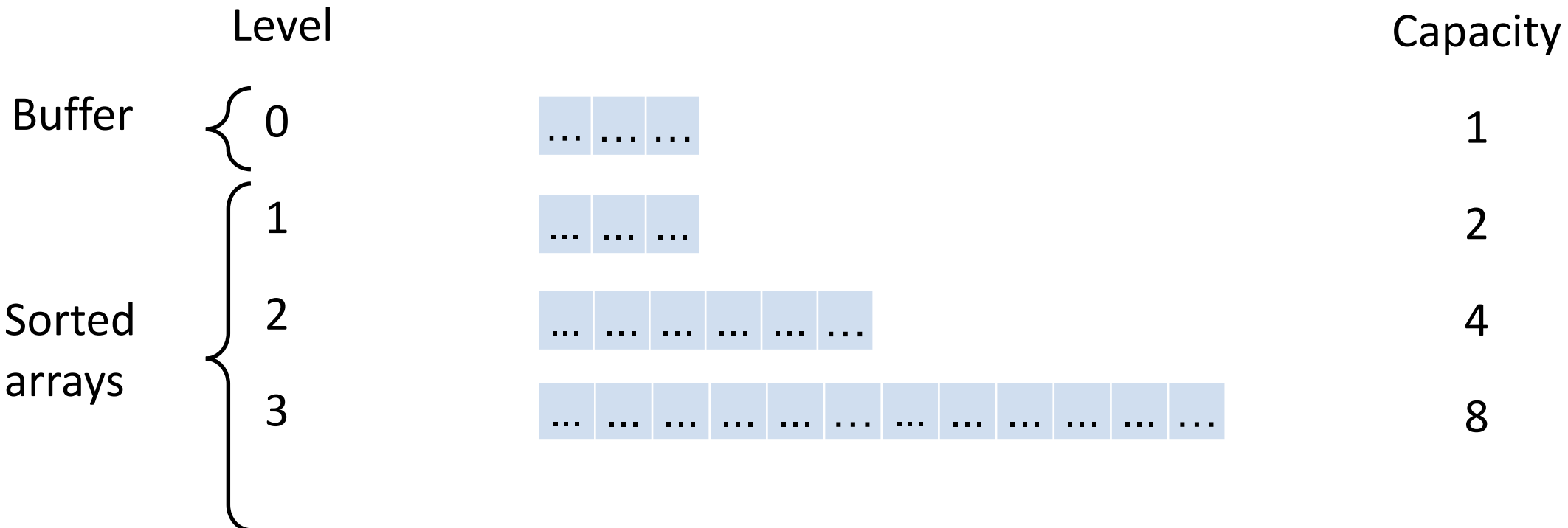
How many levels to search? $O(\log_2 N/P)$

Cost per searching each level?



How many levels to search? $O(\log_2 N/P)$

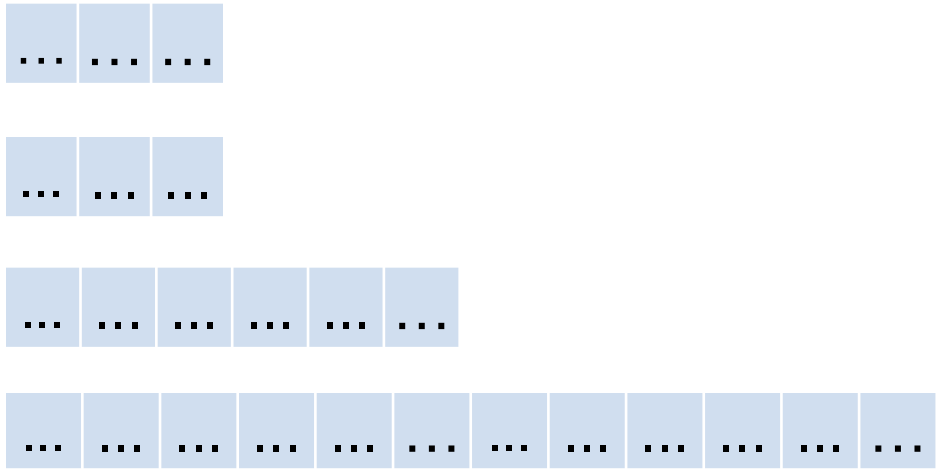
Cost per searching each level? $O(\log_2 N/B)$ w. binary search



How many levels to search? $O(\log_2 N/P)$

Cost per searching each level? $O(\log_2 N/B)$

Total search cost: **$O(\log_2(N/P) * \log_2(N/B))$**

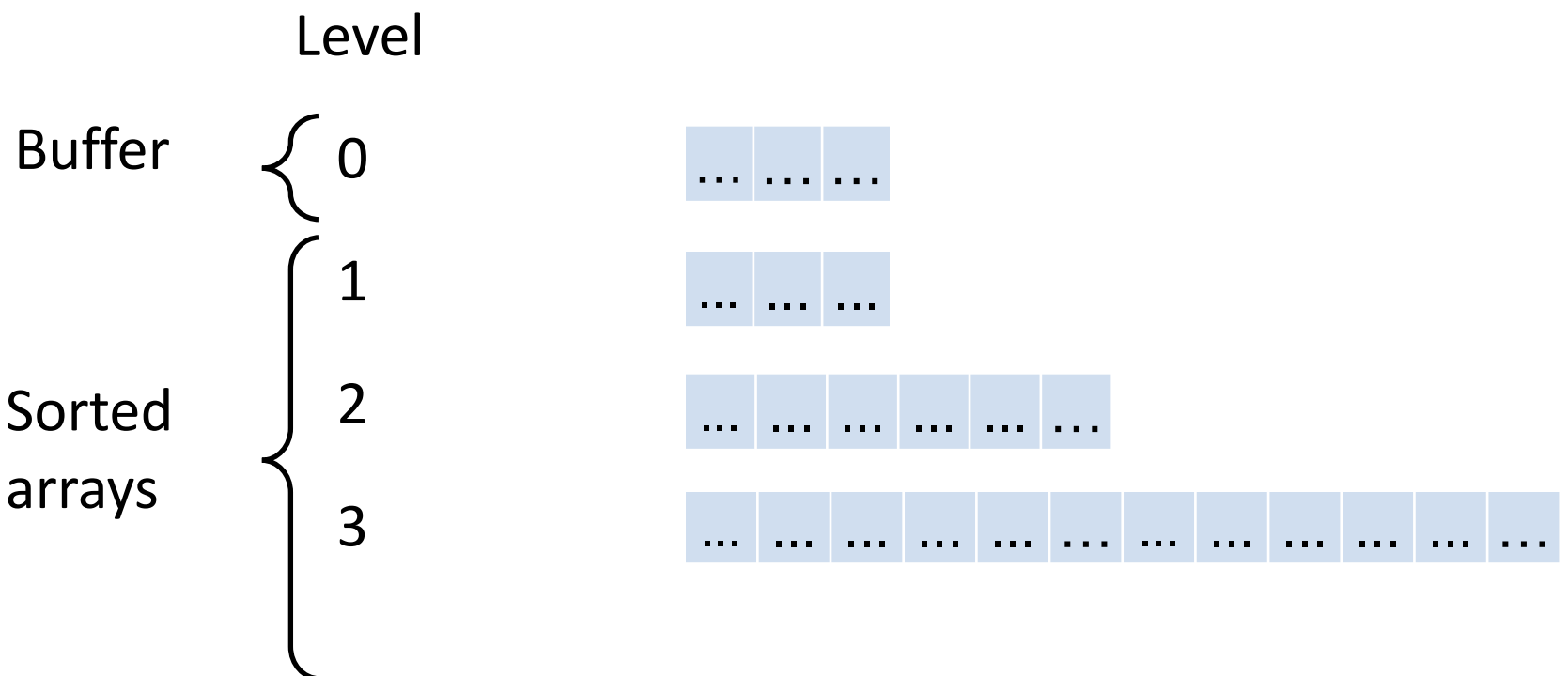


How many levels to search? $O(\log_2 N/P)$

Cost per searching each level? $O(\log_2 N/B)$

Total search cost: **$O(\log_2(N/P) * \log_2(N/B))$**

If P≈B then: **$O(\log_2(N/B)^2)$**



Basic LSM-tree – Get Queries Cost

We can do slightly better by structuring each file (SST) as a static B-tree.
How would this impact search cost?

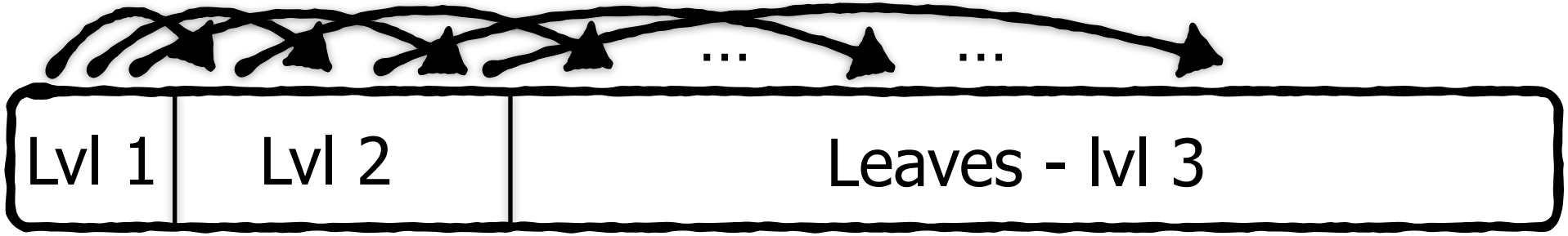
Total search cost: **$O(\log_2(N/P) * \log_2(N/B))$**

SST static B-tree structure

Basic LSM-tree – Get Queries Cost

We can do slightly better by structuring each file (SST) as a static B-tree.
How would this impact search cost?

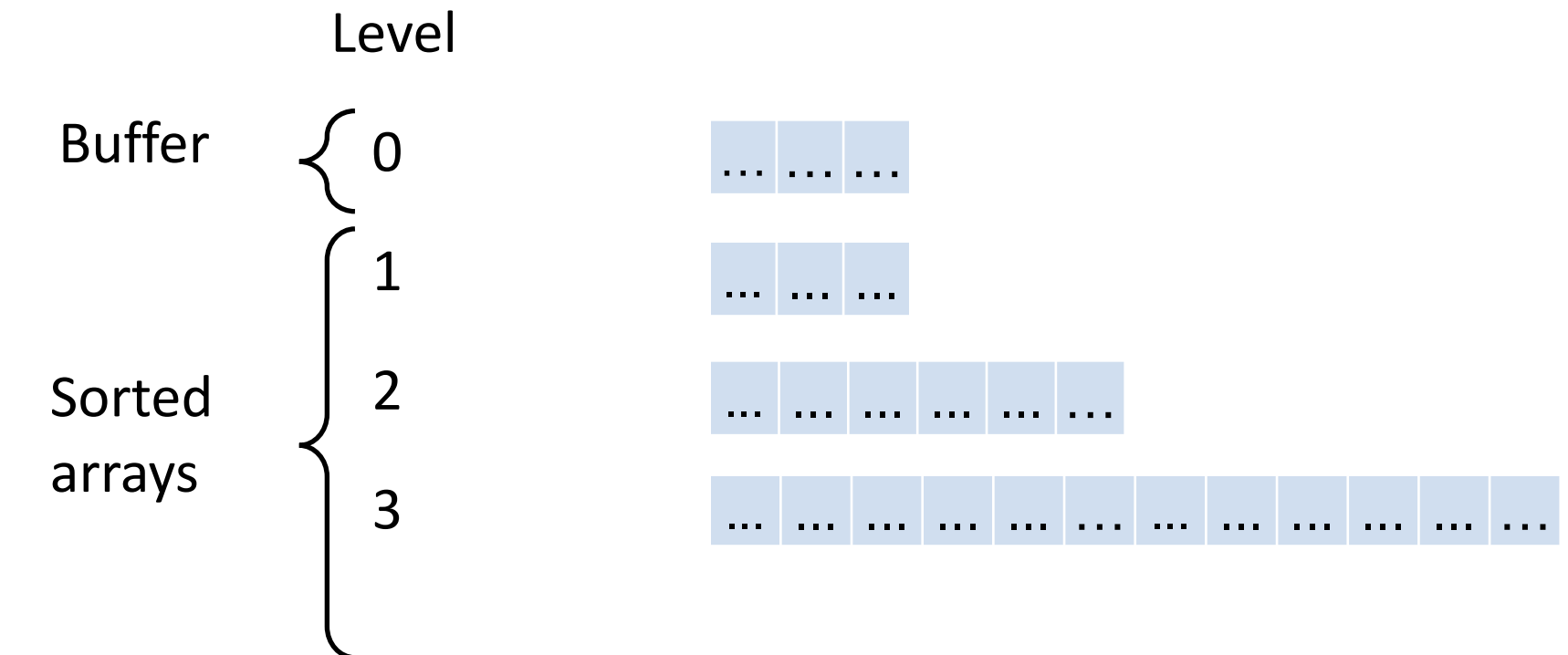
Total search cost: $O(\log_2(N/P) * \mathbf{\log_B N})$



SST static B-tree structure

Basic LSM-tree – Scan Queries

Return most recent version of each entry in the range across entire tree.

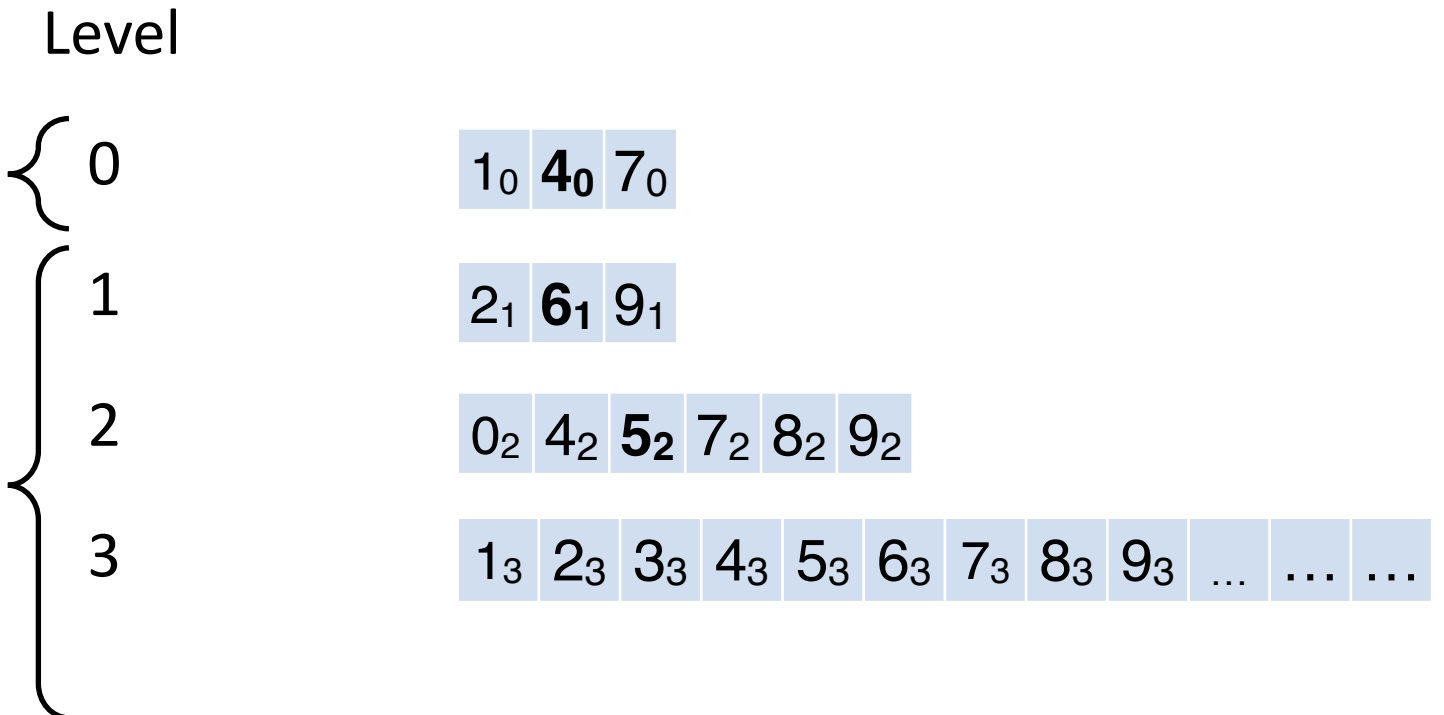


Basic LSM-tree – Scan Queries

Return most recent version of each entry in the range across entire tree.

e.g., Scan(4, 6) - range inclusive

Expected output?

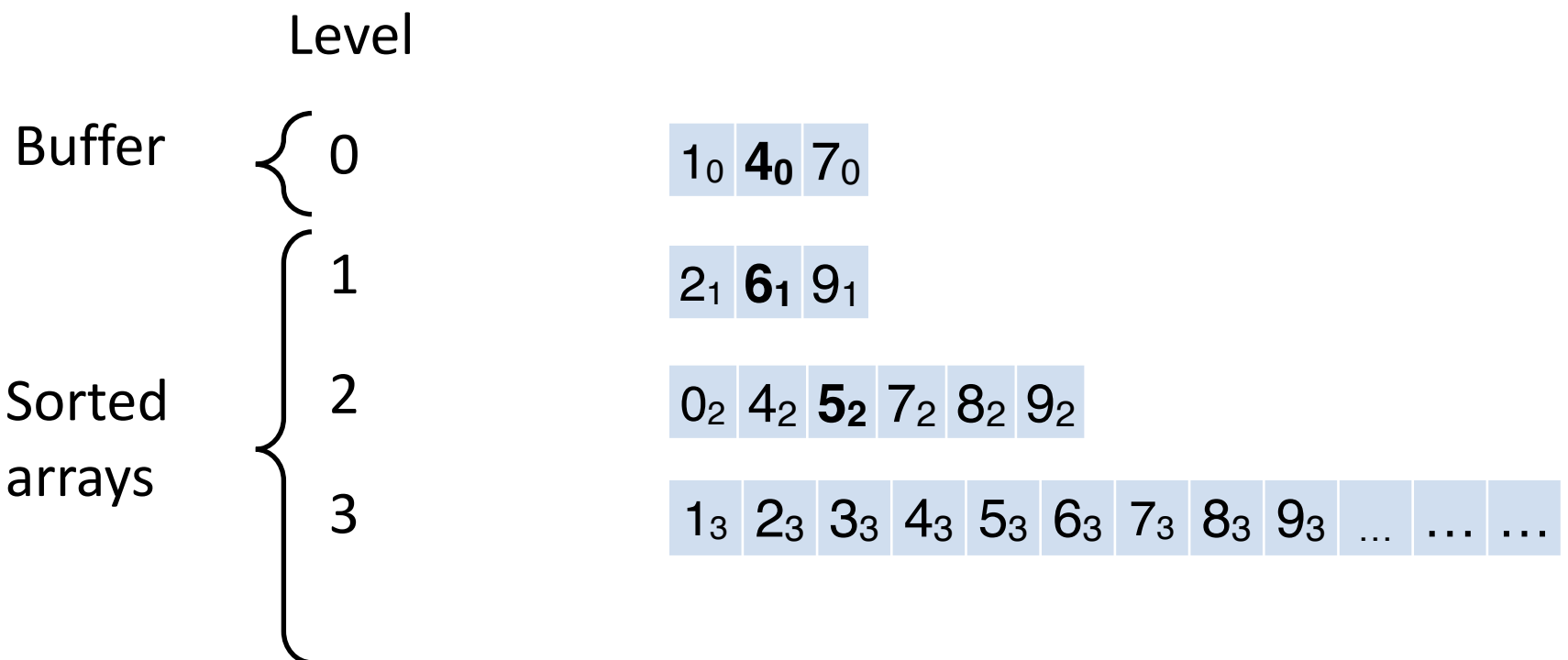


Basic LSM-tree – Scan Queries

Return most recent version of each entry in the range across entire tree.

e.g., Scan(4, 6)

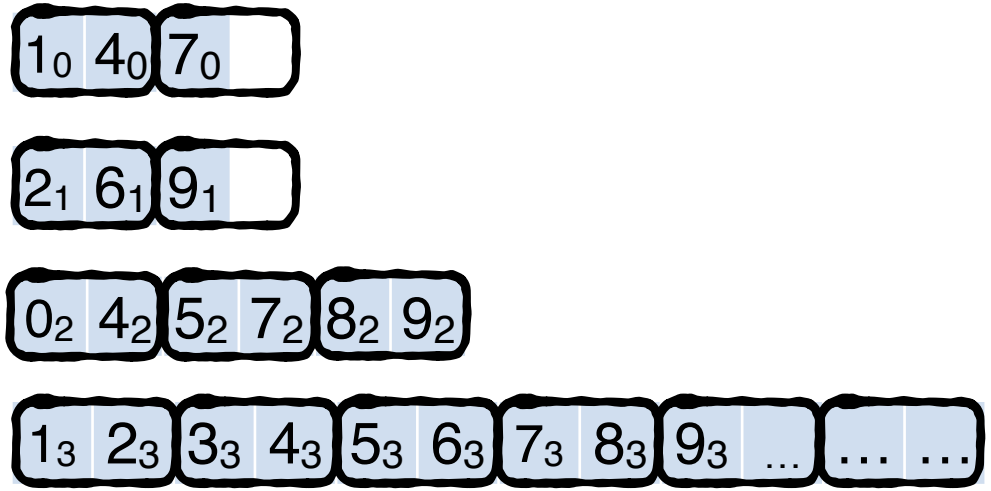
Expected output: $4_0\ 5_2\ 6_1$



Scan(4, 6)

1. **Allocate an in-memory buffer (≥1 page) for each level**

e.g., 1 page buffer and a page Contains 2 entries



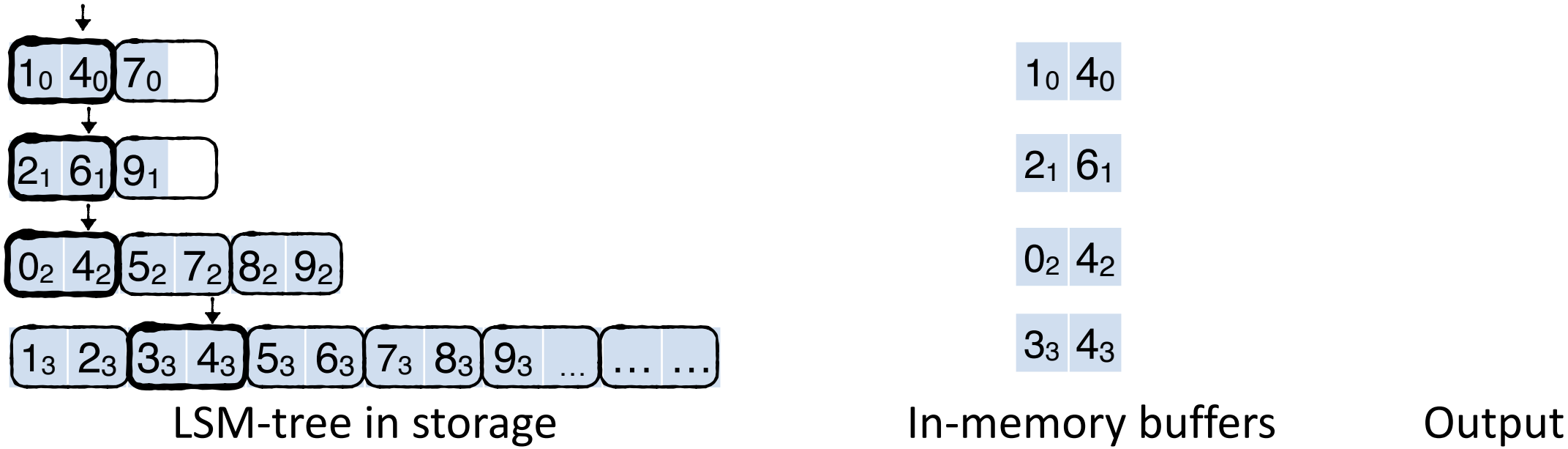
LSM-tree in storage

In-memory buffers

Output

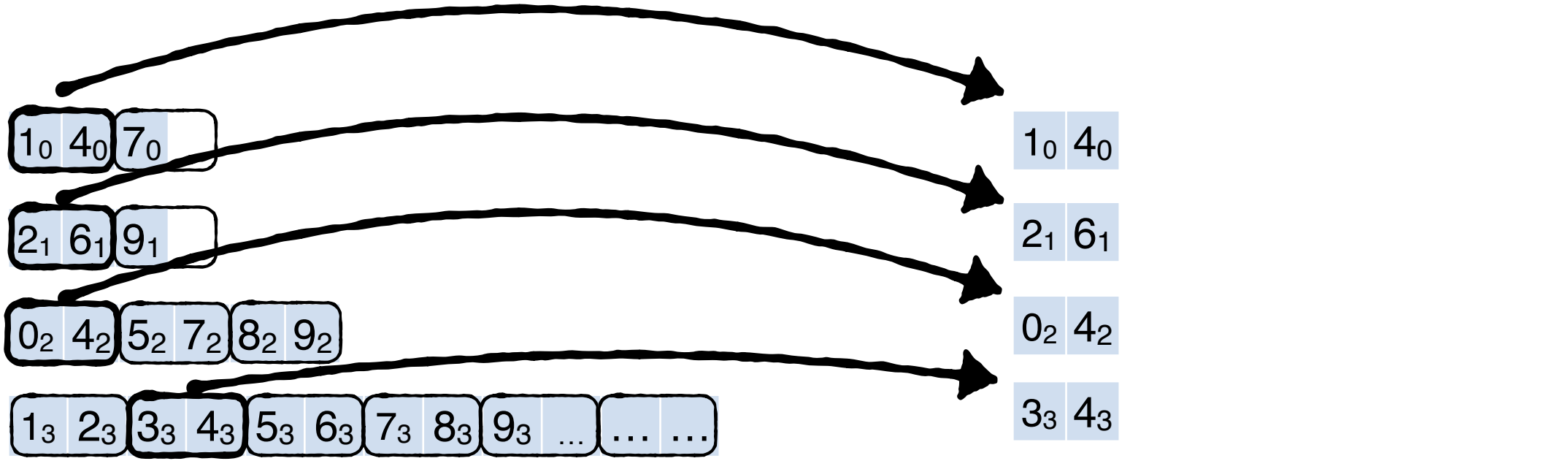
Scan(4, 6)

1. Allocate an in-memory buffer (≥1 page) for each level
2. Search for start of key range at each level



Scan(4, 6)

1. Allocate an in-memory buffer (≥1 page) for each level
2. Search for start of key range at each level

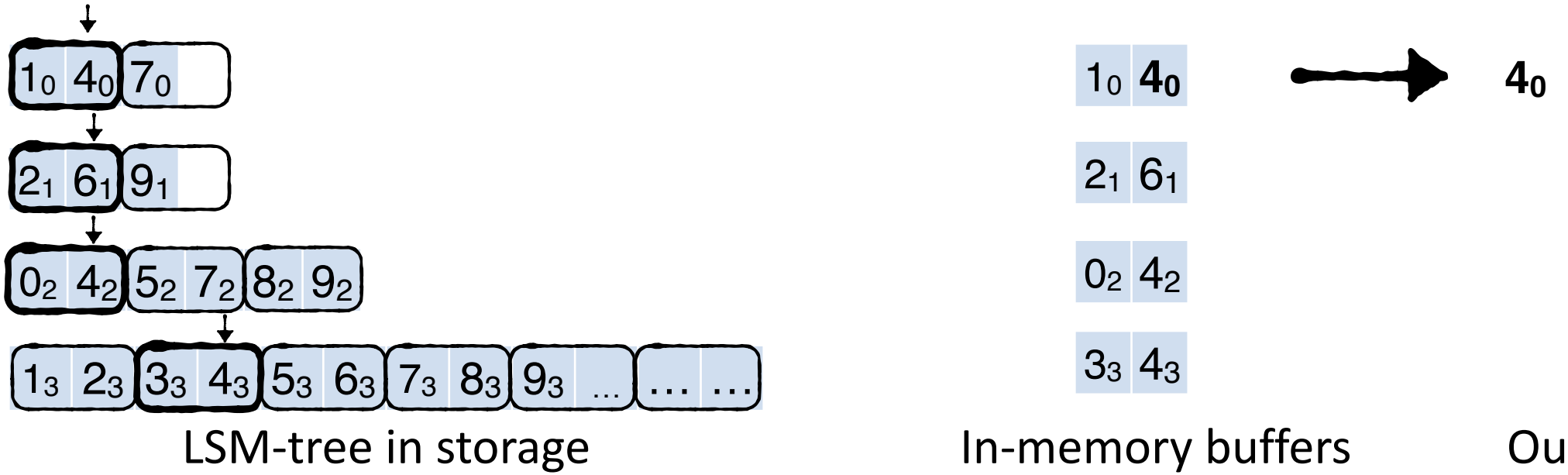


Scan(4, 6)

1. Allocate an in-memory buffer (≥1 page) for each level
2. Search for start of key range at each level

Loop until reaching end of range

3. Output youngest version of entry with next smallest key

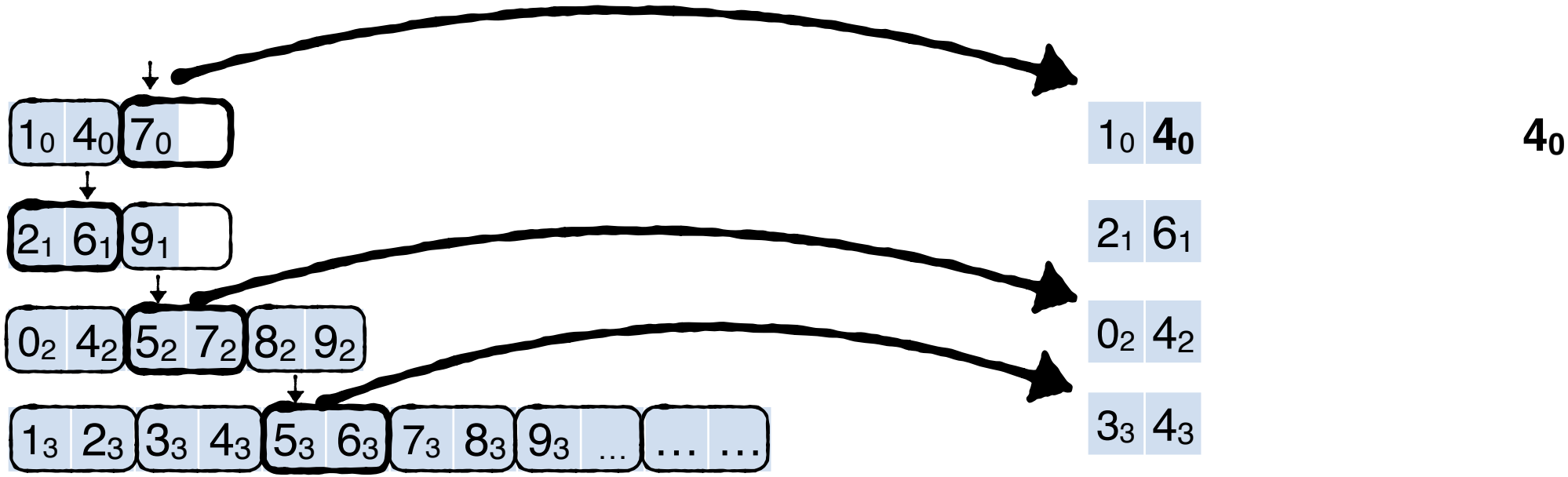


Scan(4, 6)

1. Allocate an in-memory buffer (≥1 page) for each level
2. Search for start of key range at each level

Loop until reaching end of range

3. Output youngest version of entry with next smallest key
4. If we traverse last entry in a given buffer, read next page from run

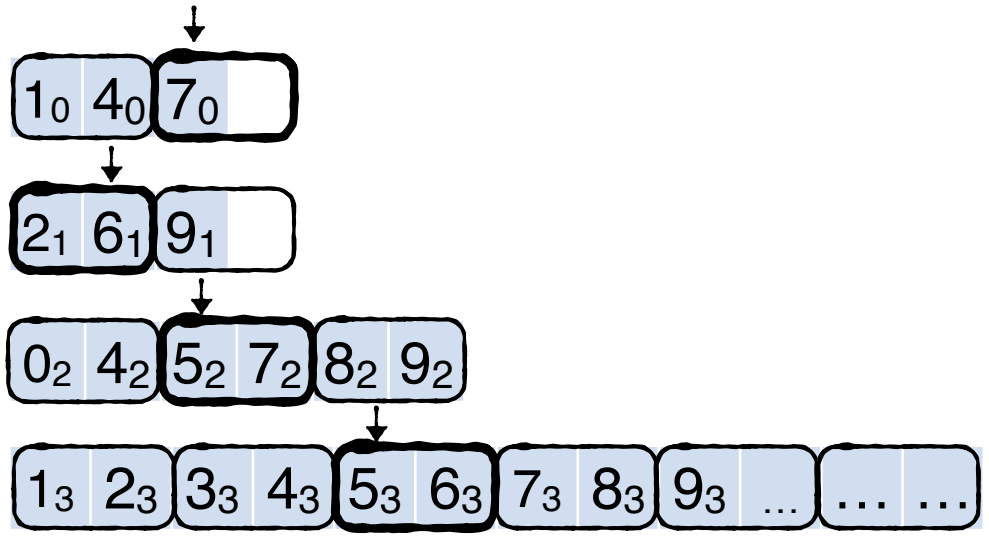


Scan(4, 6)

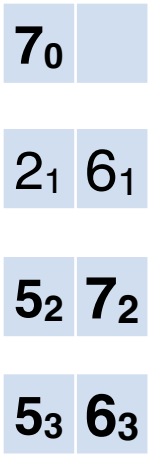
1. Allocate an in-memory buffer (≥1 page) for each level
2. Search for start of key range at each level

Loop until reaching end of range

3. Output youngest version of entry with next smallest key
4. If we traverse last entry in a given buffer, read next page from run



LSM-tree in storage



In-memory buffers

4₀

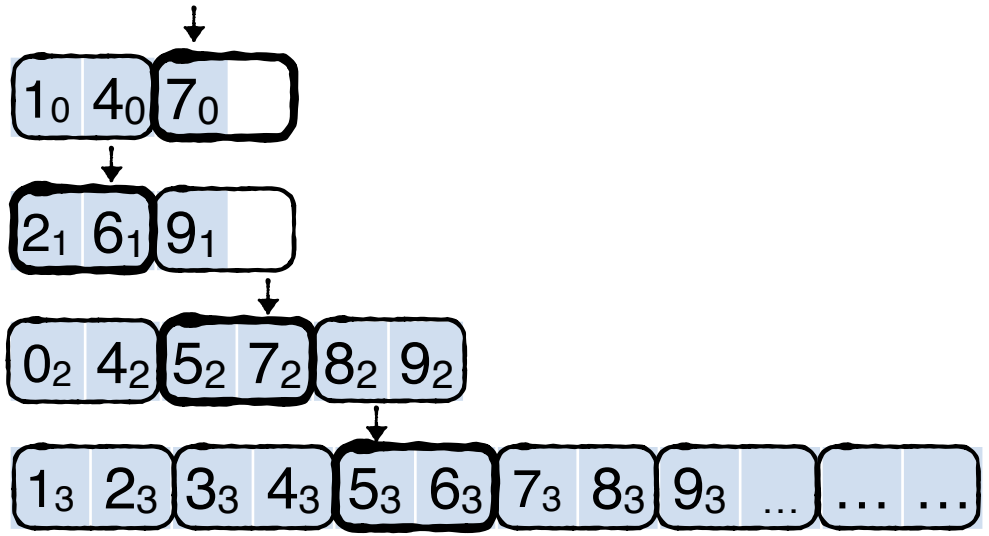
Output

Scan(4, 6)

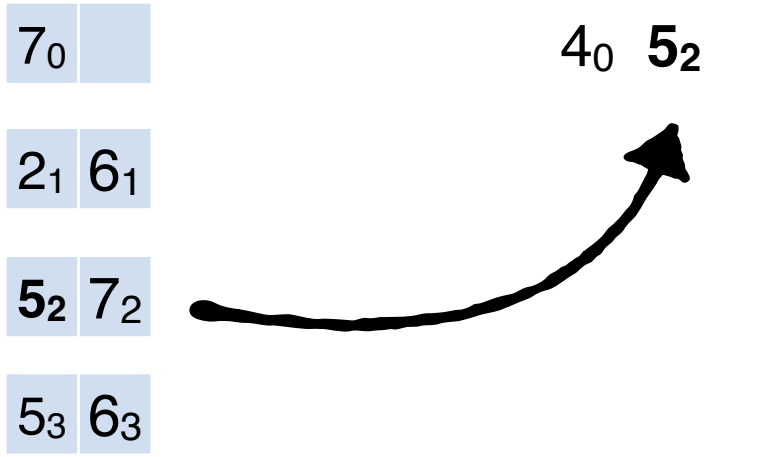
1. Allocate an in-memory buffer (≥1 page) for each level
2. Search for start of key range at each level

Loop until reaching end of range

3. Output youngest version of entry with next smallest key
4. If we traverse last entry in a given buffer, read next page from run



LSM-tree in storage



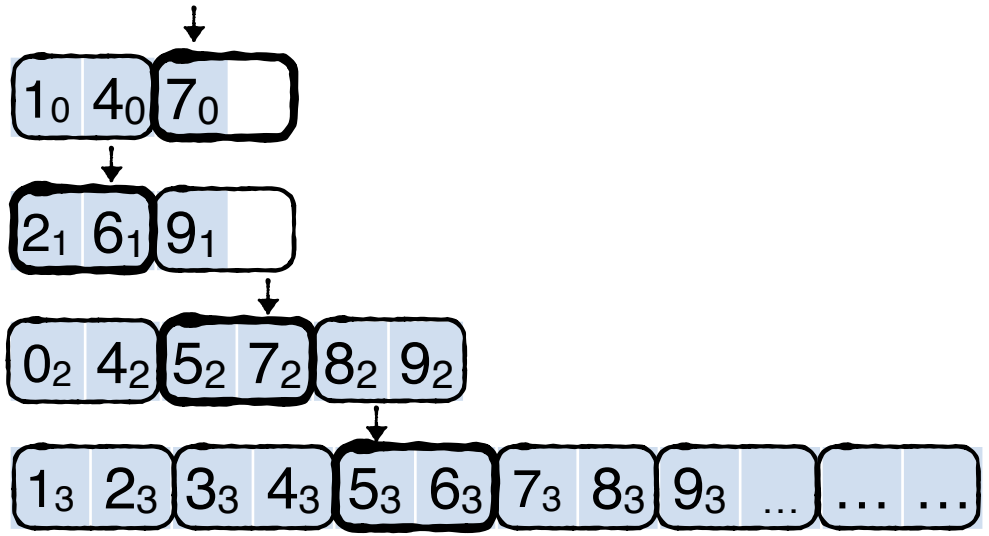
In-memory buffers Output

Scan(4, 6)

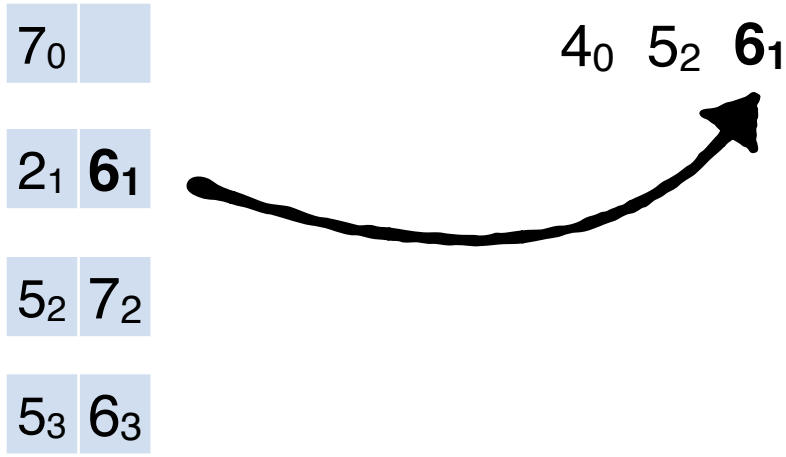
1. Allocate an in-memory buffer (≥1 page) for each level
2. Search for start of key range at each level

Loop until reaching end of range

3. Output youngest version of entry with next smallest key
4. **If we traverse last entry in a given buffer, read next page from run**



LSM-tree in storage



In-memory buffers Output

Basic LSM-tree – Scan Queries I/O Cost?

LSM-tree in storage

In-memory buffers

Basic LSM-tree – Scan Queries I/O Cost

$$O(\log_2(N/P) * \log_B(N) + S/B)$$

LSM-tree in storage

In-memory buffers

Basic LSM-tree – Scan Queries I/O Cost

 # lvls

$O(\mathbf{\log_2(N/P)} * \log_B(N) + S/B)$

LSM-tree in storage

In-memory buffers

Basic LSM-tree – Scan Queries I/O Cost

lvls

$O(\log_2(N/P) * \mathbf{\log_B(N)} + S/B)$

B-tree search cost

LSM-tree in storage

In-memory buffers

Basic LSM-tree – Scan Queries I/O Cost

lvls **# output size**

$O(\log_2(N/P) * \log_B(N) + \mathbf{S/B})$

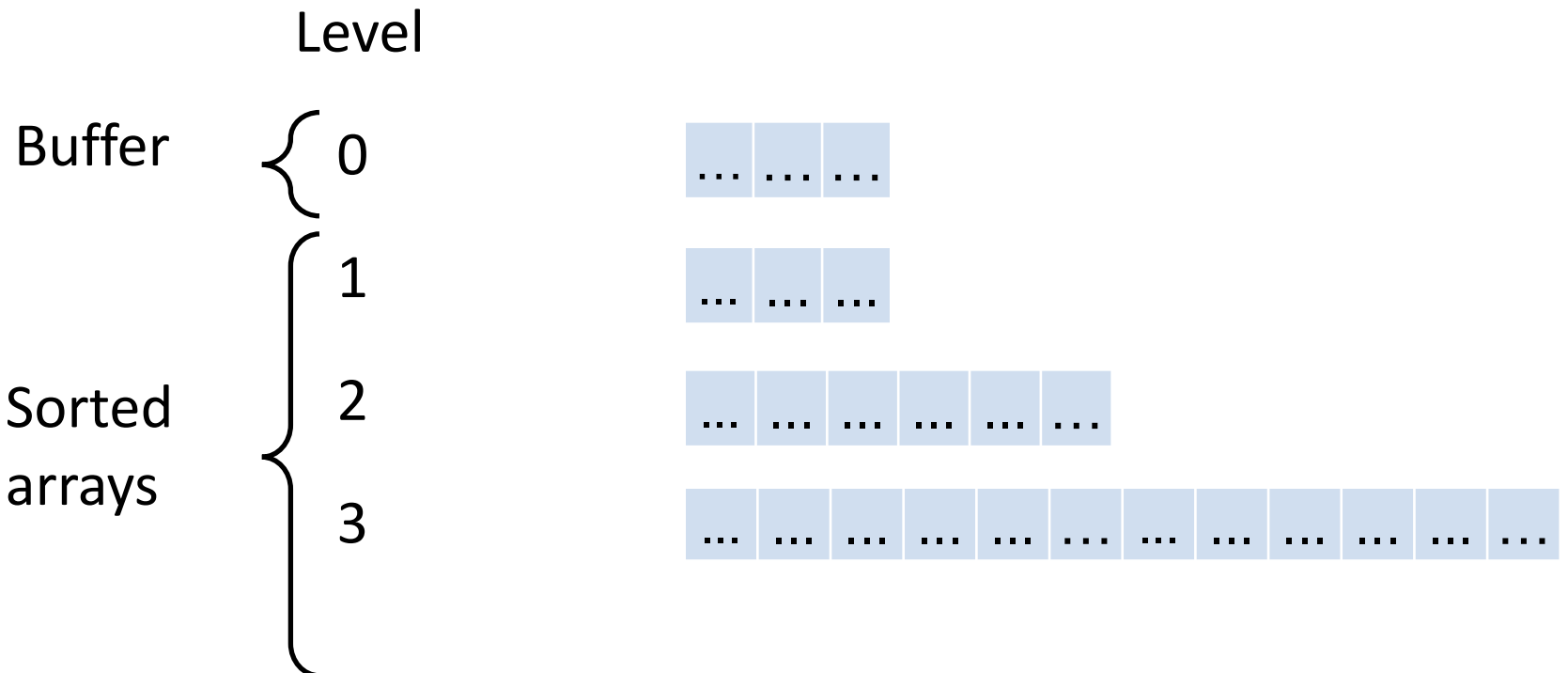
B-tree search cost

LSM-tree in storage

In-memory buffers

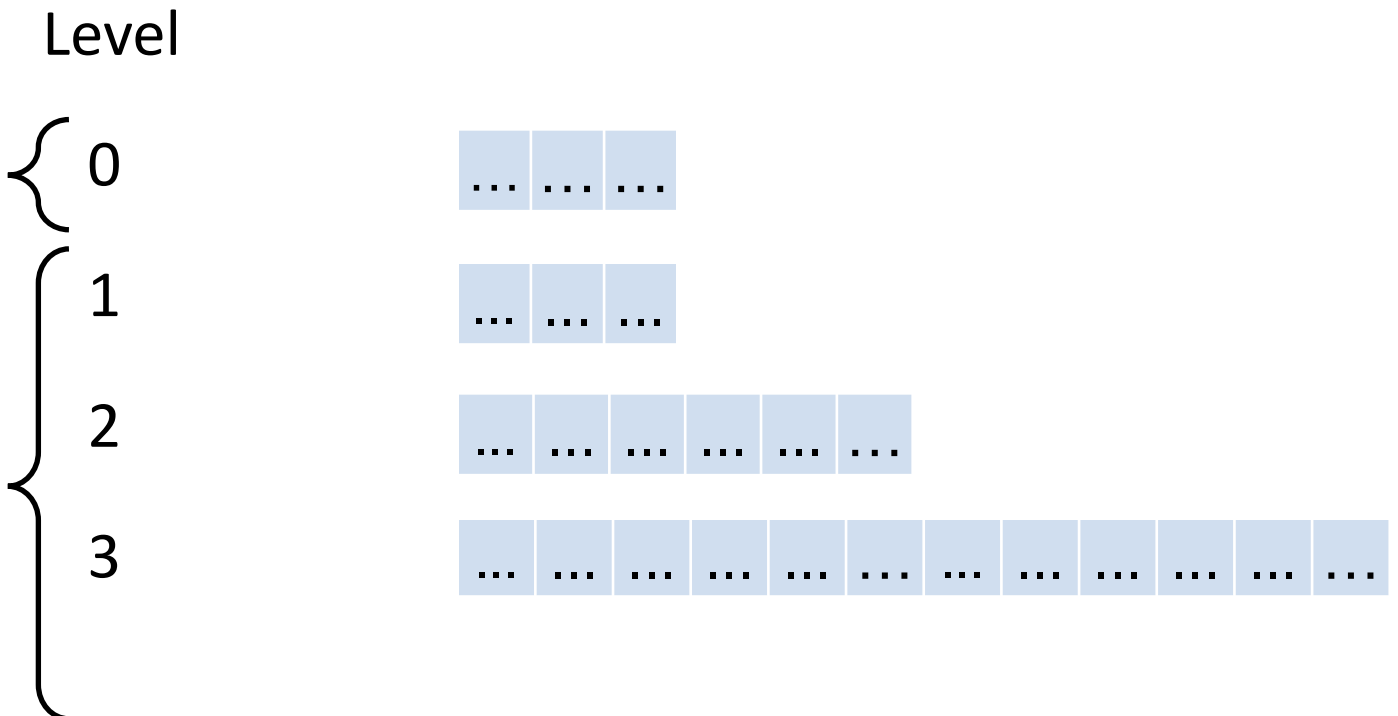
Basic LSM-tree – Insertion/Update/Delete cost

How many times is each entry copied?



Basic LSM-tree – Insertion/Update/Delete cost

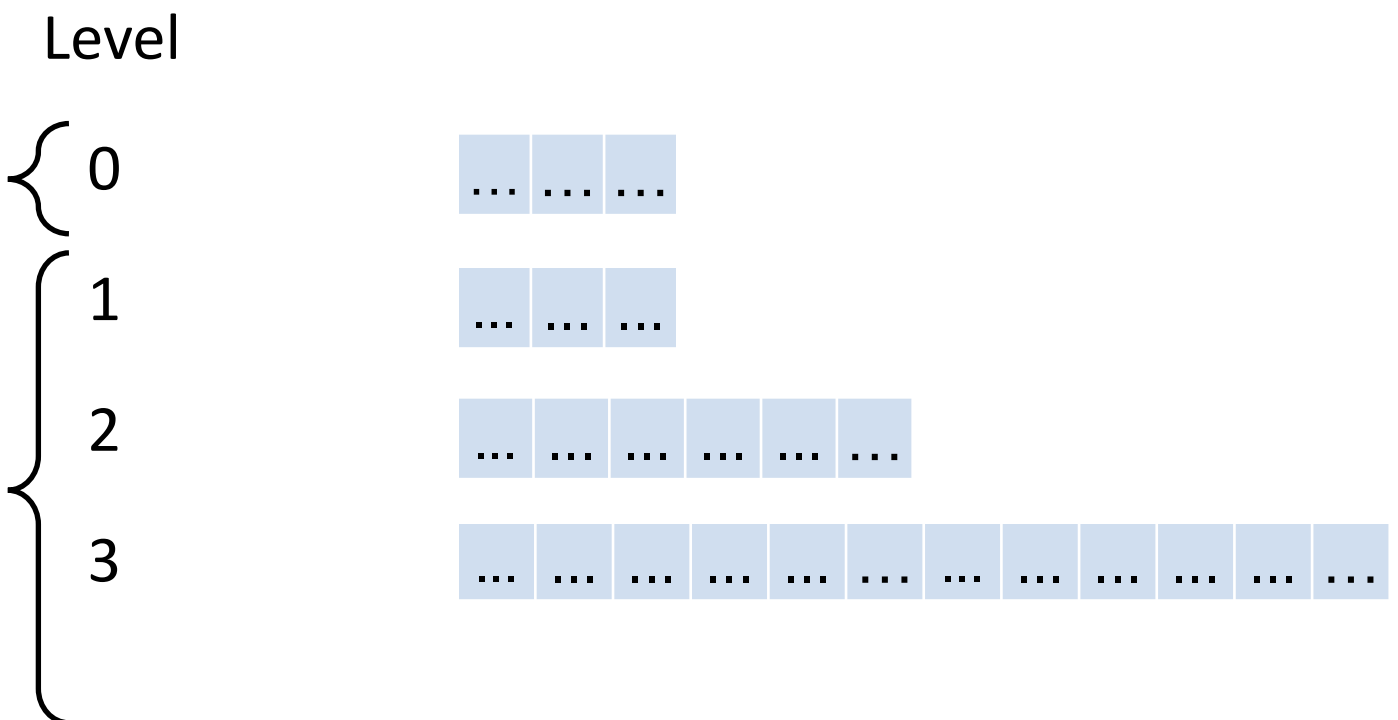
How many times is each entry copied? **$O(\log_2 N/P)$**



Basic LSM-tree – Insertion/Update/Delete cost

How many times is each entry copied? $O(\log_2 N/P)$

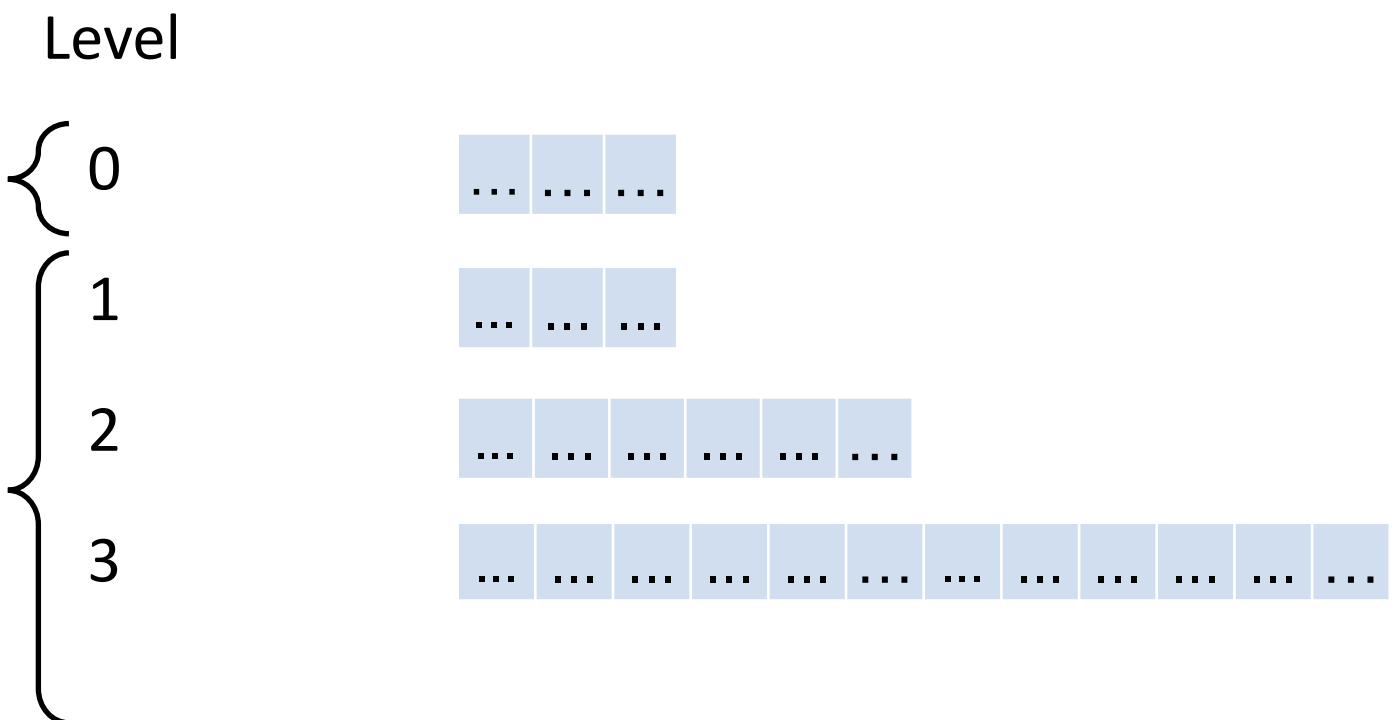
Price of each copy?



Basic LSM-tree – Insertion/Update/Delete cost

How many times is each entry copied? $O(\log_2 N/P)$

Price of each copy? $O(1/B)$ reads & writes

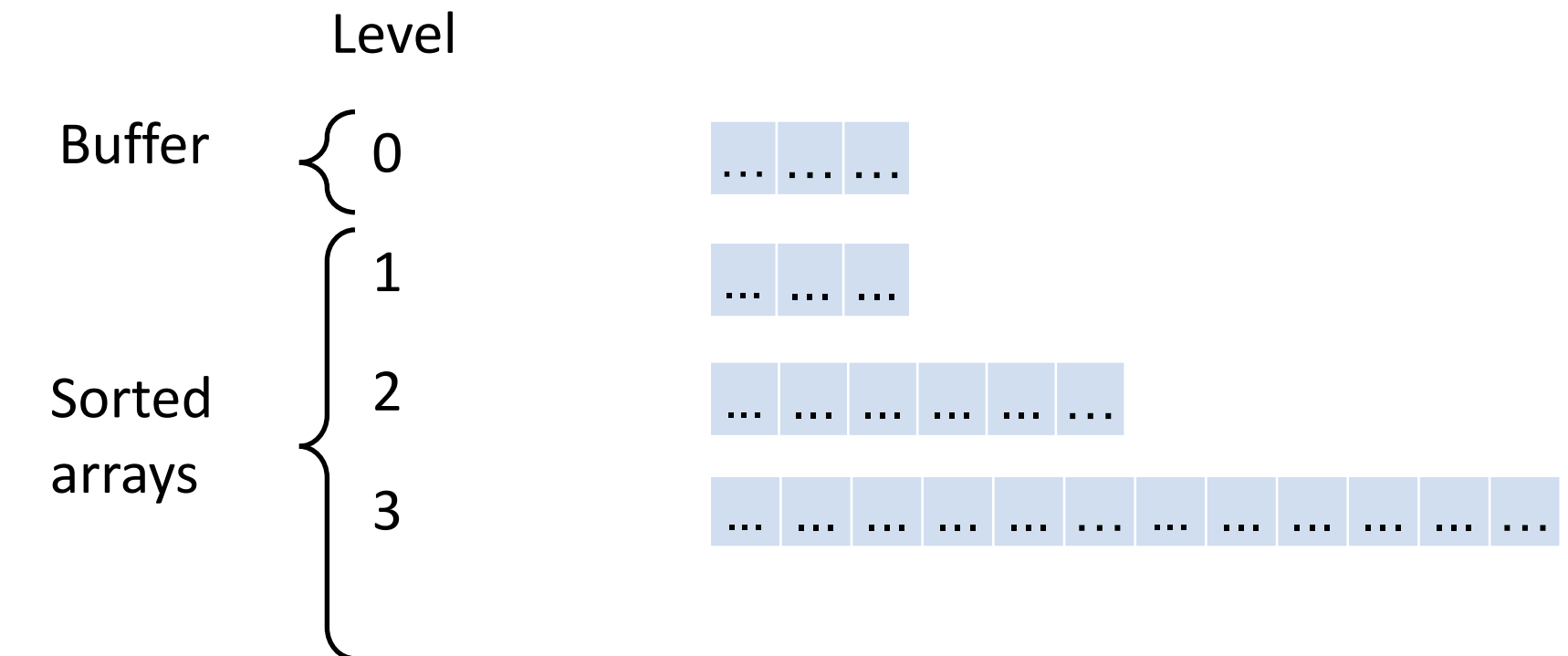


Basic LSM-tree – Insertion/Update/Delete cost

How many times is each entry copied? $O(\log_2 N/P)$

Price of each copy? $O(1/B)$ reads & writes

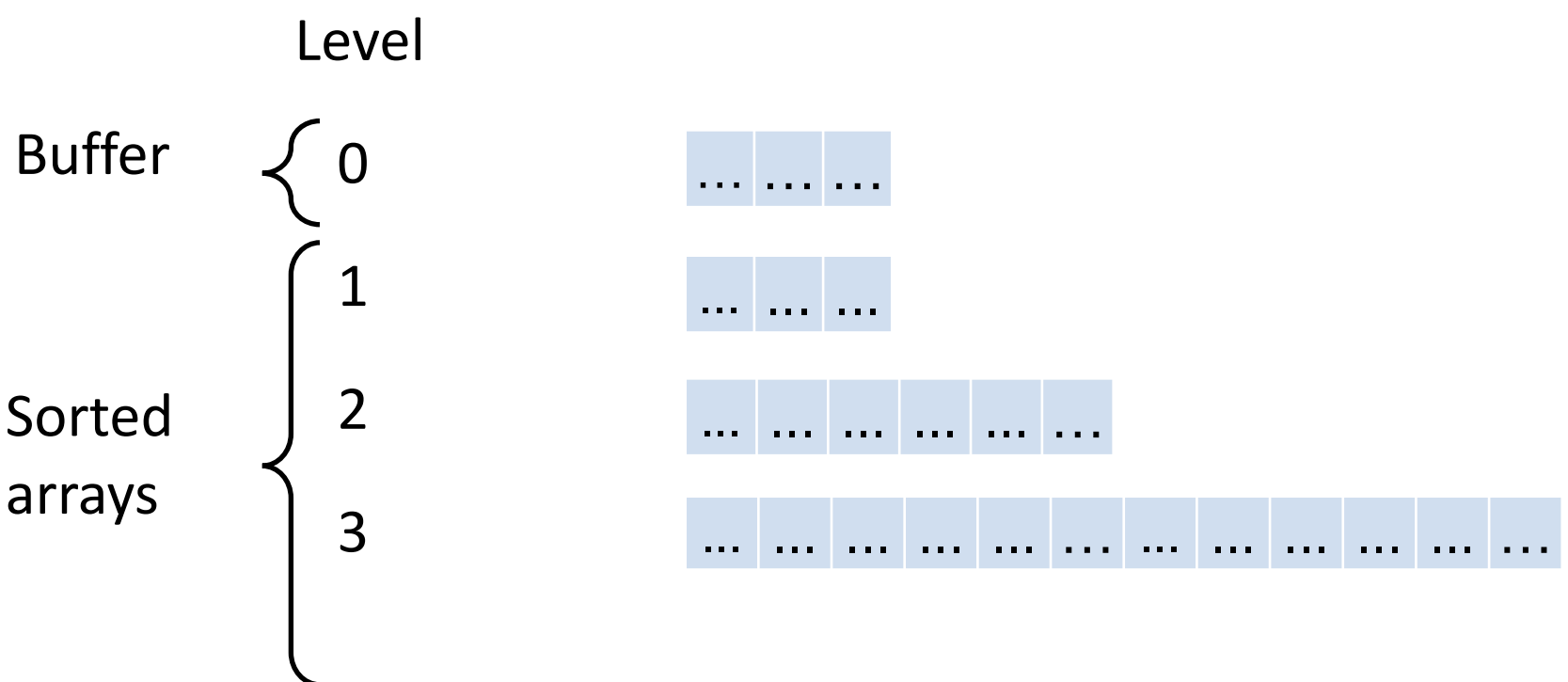
Total cost: $O((\log_2 N/P)/B)$ read & write I/Os



Basic LSM-tree – Insertion/Update/Delete cost

Total cost: $O((\log_2 N/P)/B)$ read & write I/Os

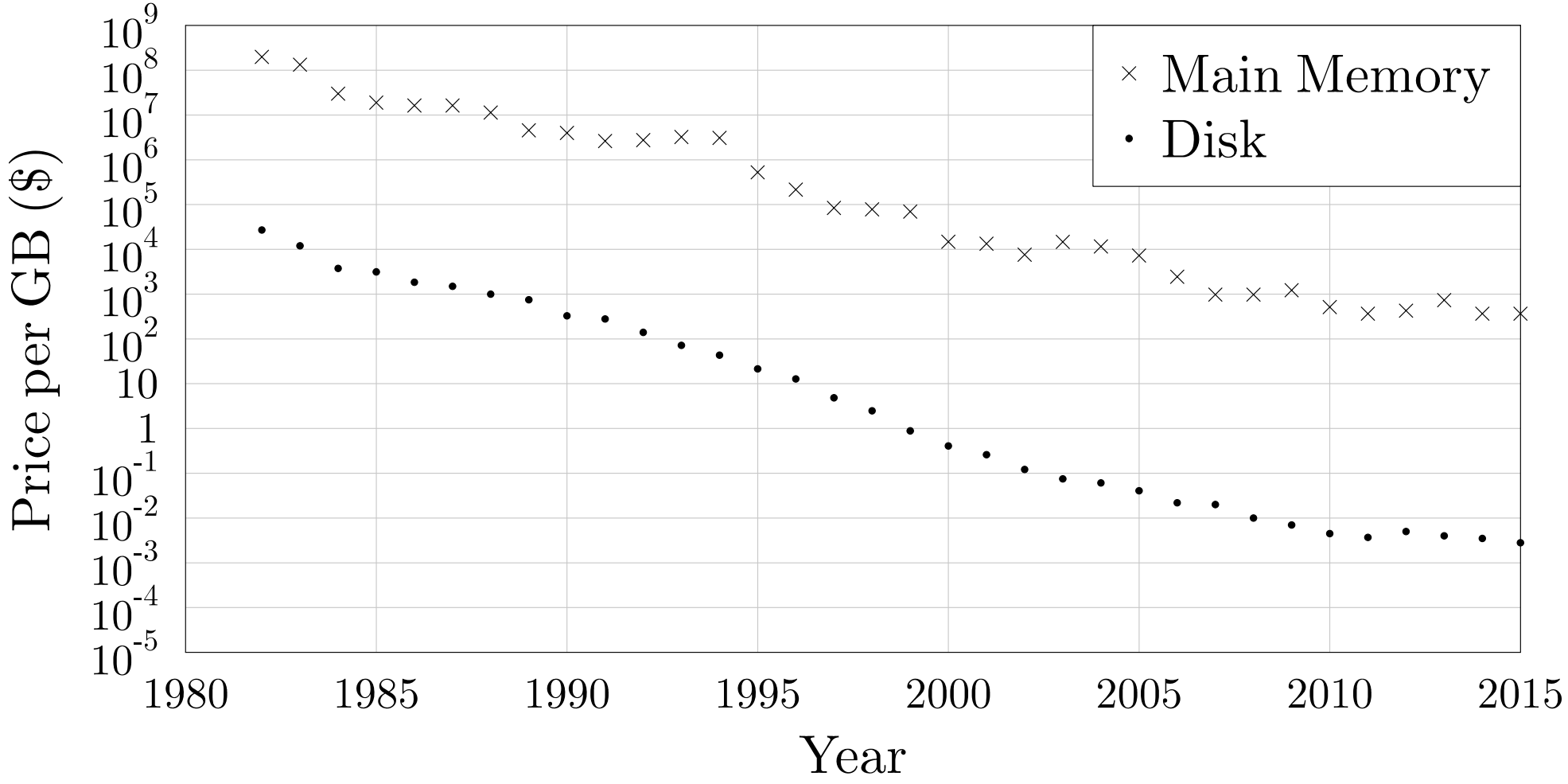
As all writes are large & sequential (rather than random), there is less SSD garbage-collection



Operation	I/O	append-only table	Basic LSM-tree table	B-Tree
Query	Reads	O(N/B)	**$O(\log_2(N/P) * \log_B N)$**	$O(\log_B N)$
Insert	Reads	0	**$O((\log_2 N/P)/B)$**	$O(\log_B N)$
	Writes	O(1/B)	**$O((\log_2 N/P)/B)$**	O(1) & GC

Break :)

Declining Main Memory Cost

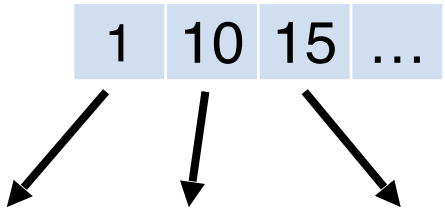


Declining Main Memory Cost

It’s viable to pin the internal nodes of B-trees in memory

Fence pointers

array



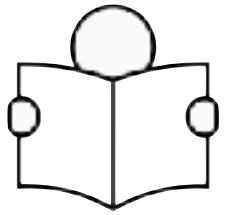
Block 1	Block 2	Block 3	…
1	10	15	…
3	11	16	…
6	13	18	…

Cost Metric	Unit	append-only table	**Basic LSM-tree table**	**B-Tree**
Query	Reads IOs	O(N/B)	O($\log_2(N/P) * \log_B N$)	O($\log_B N$)
Insert	Reads IOs	0	O(($\log_2 N/P$)/B)	O($\log_B N$)
	Write IOs	O(1/B)	O(($\log_2 N/P$)/B)	O(1) & GC

Cost Metric	Unit	append-only table	**Basic LSM-tree table**	**B-Tree**
Query	Reads IOs	O(N/B)	O($\log_2(N/P)$)	O(1)
Insert	Reads IOs	0	O(($\log_2$ N/P)/B)	O(1)
	Write IOs	O(1/B)	O(($\log_2$ N/P)/B)	O(1) & GC
Memory	#entries	**O(B)**	**O(N/B)**	**O(N/B)**

Memory cost is measured here as # of entries stored in memory

Leveled LSM-tree



Basic LSM-tree

Tiered LSM-tree

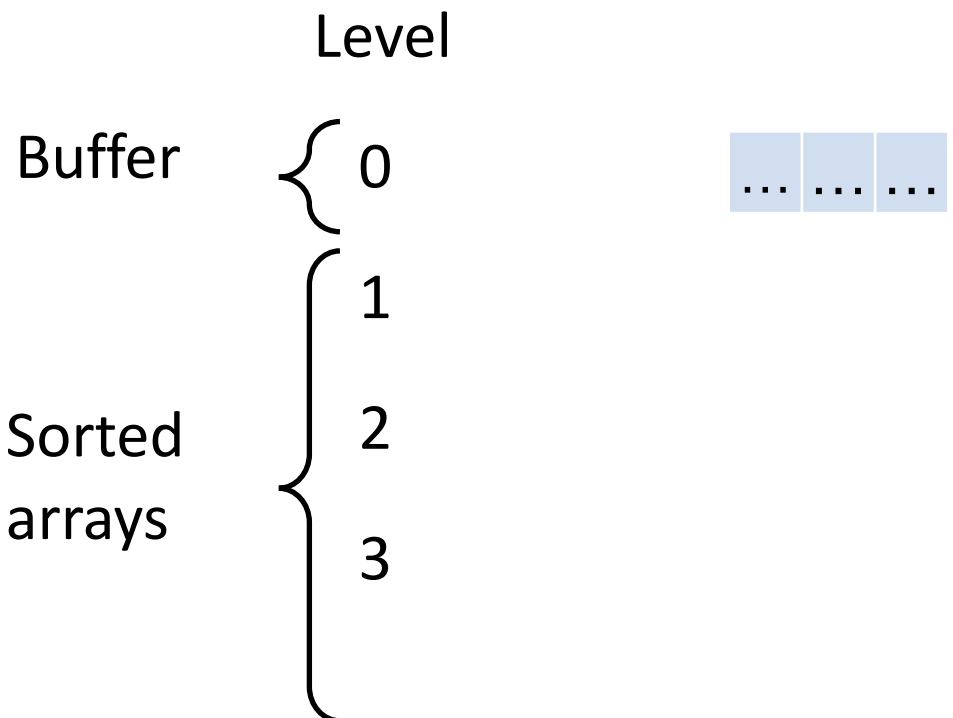
Leveled LSM-tree

 Lookup cost

 Update cost

Leveled LSM-tree

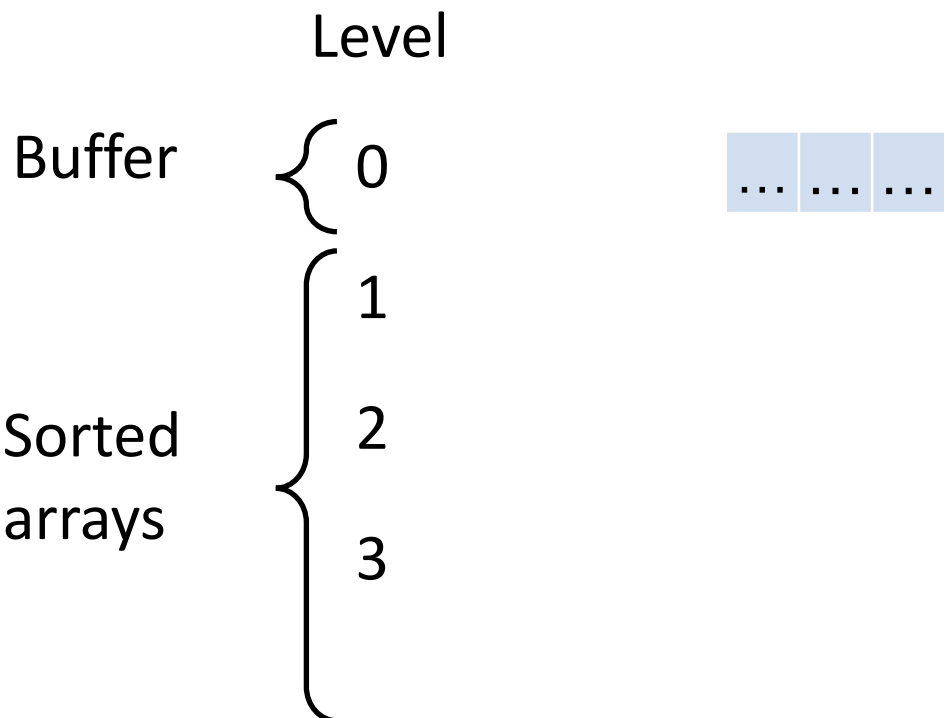
Lookup cost depends on number of levels



Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

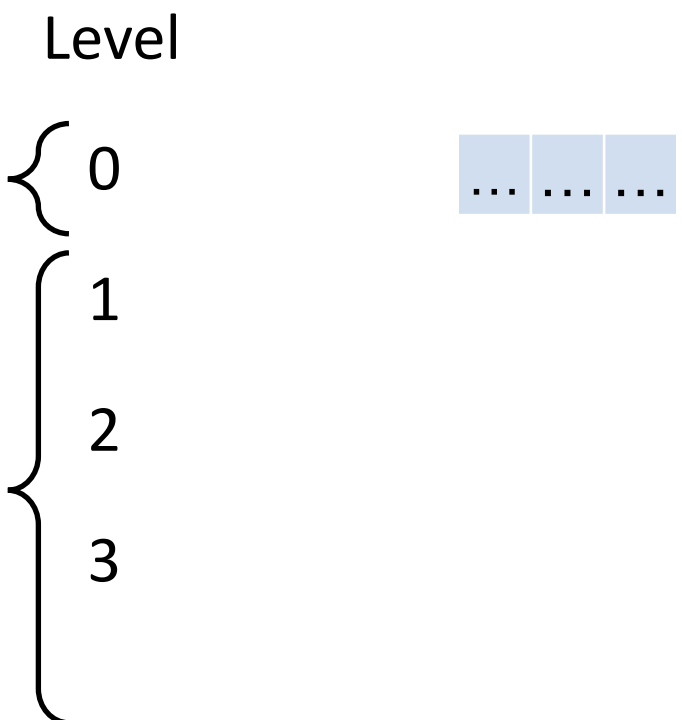


Leveled LSM-tree

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? 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

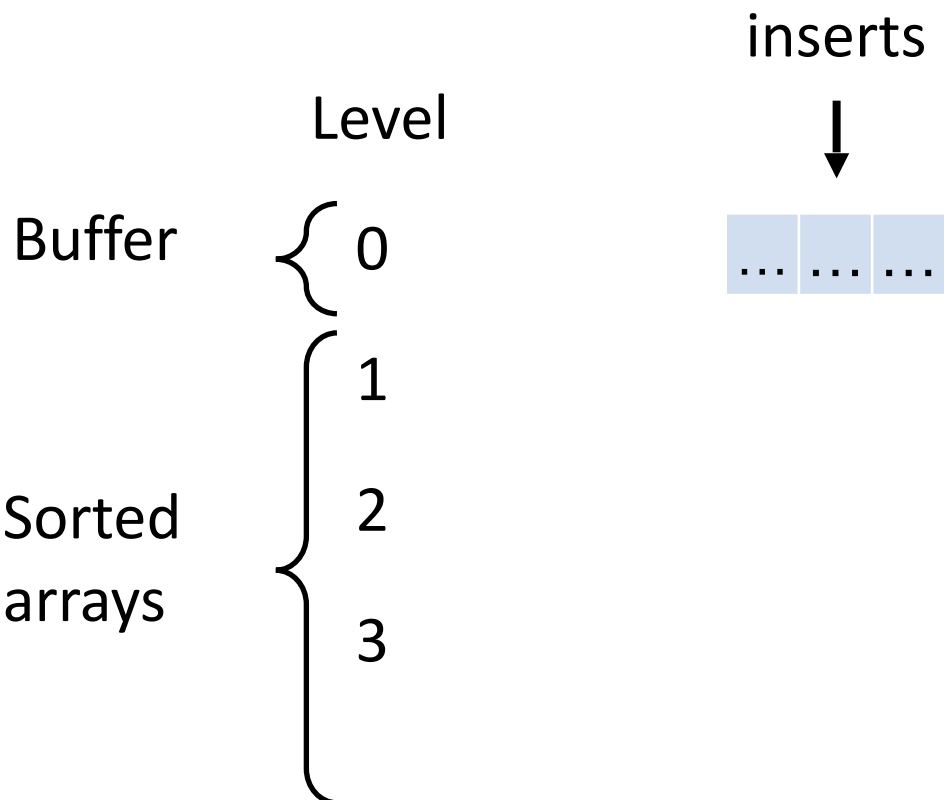
Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

Increase size ratio T

E.g. size ratio of 4



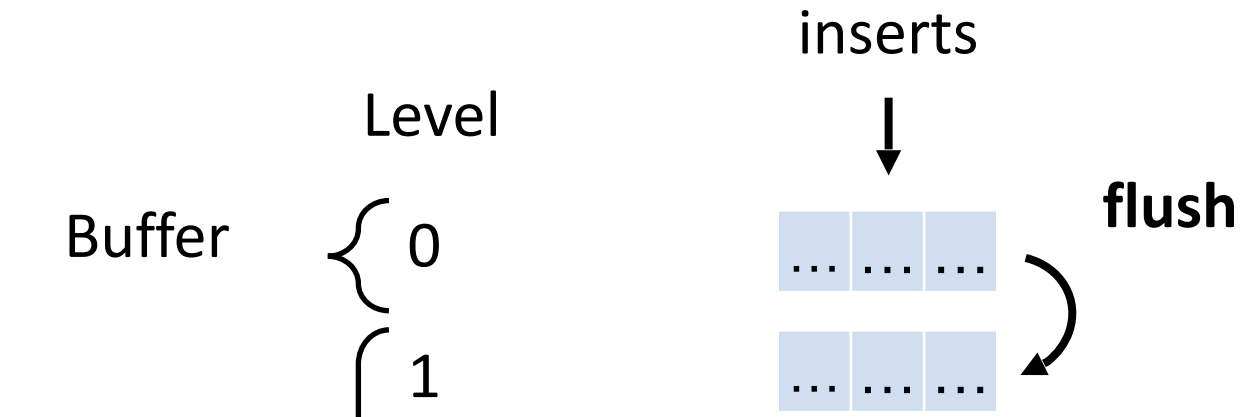
Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

Increase size ratio T

E.g. size ratio of 4



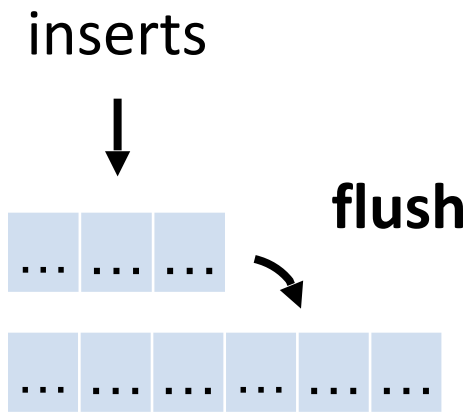
Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

Increase size ratio T

E.g. size ratio of 4



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

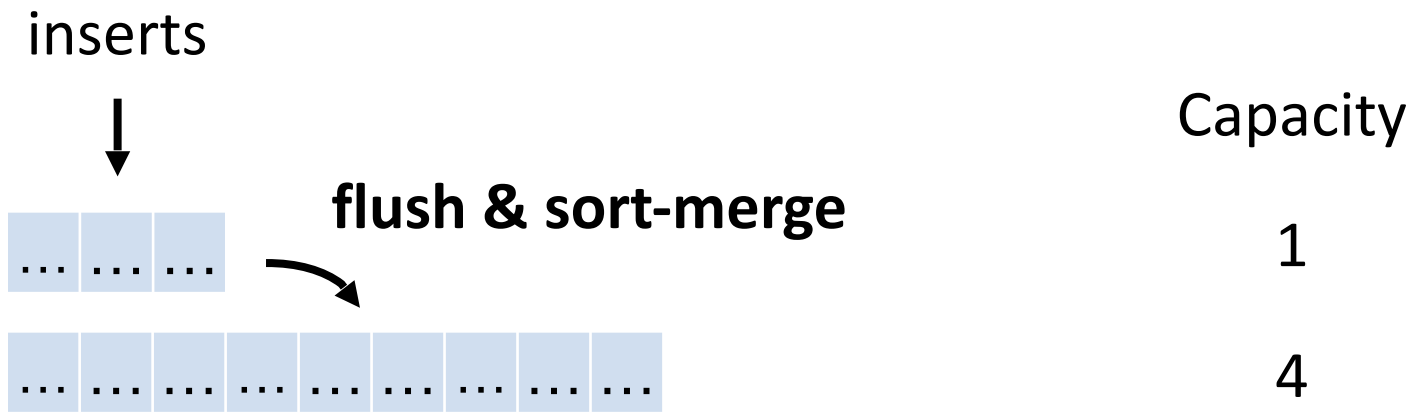
Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it?

Increase size ratio T

E.g. size ratio of 4

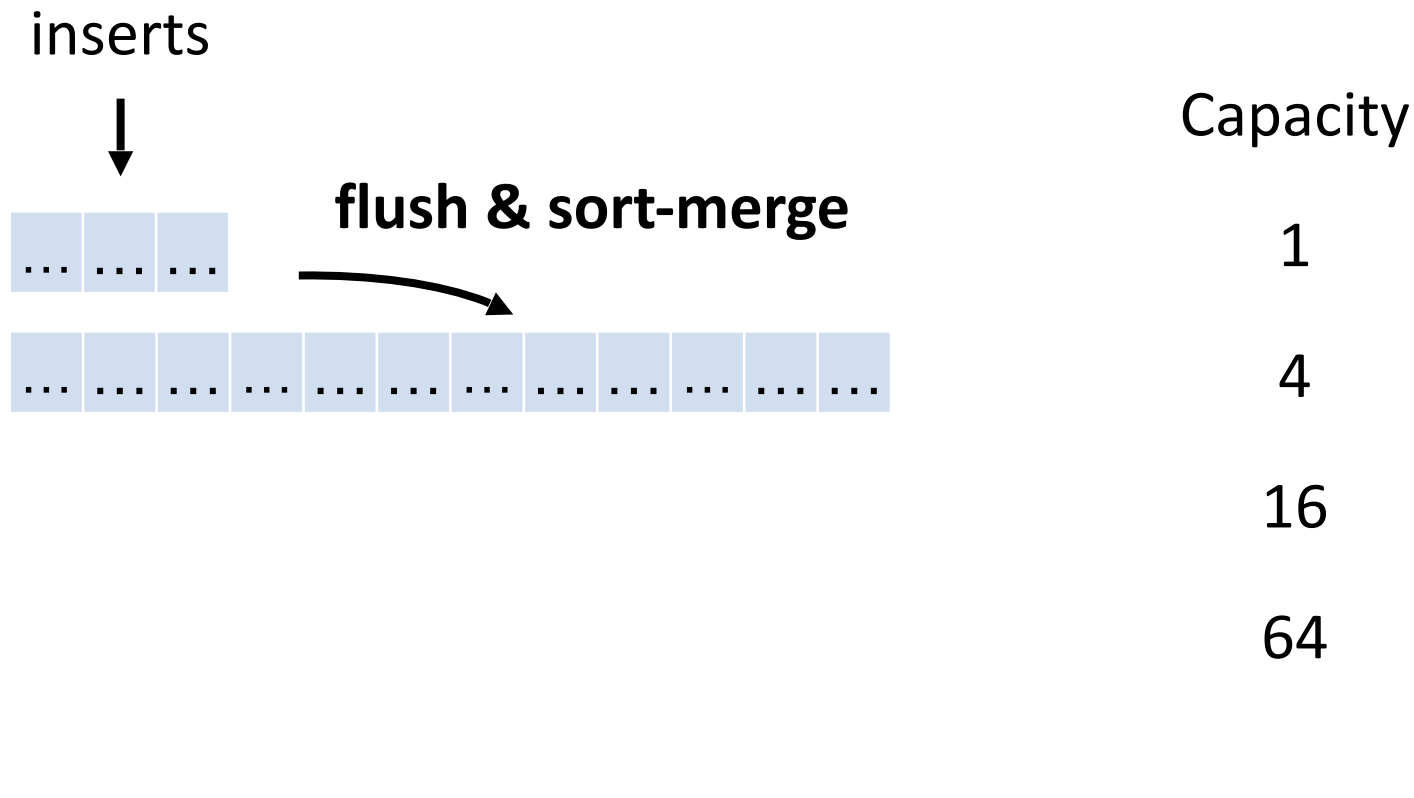


Leveled LSM-tree

Lookup cost depends on number of levels

How to reduce it? Increase size ratio T

E.g. size ratio of 4



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

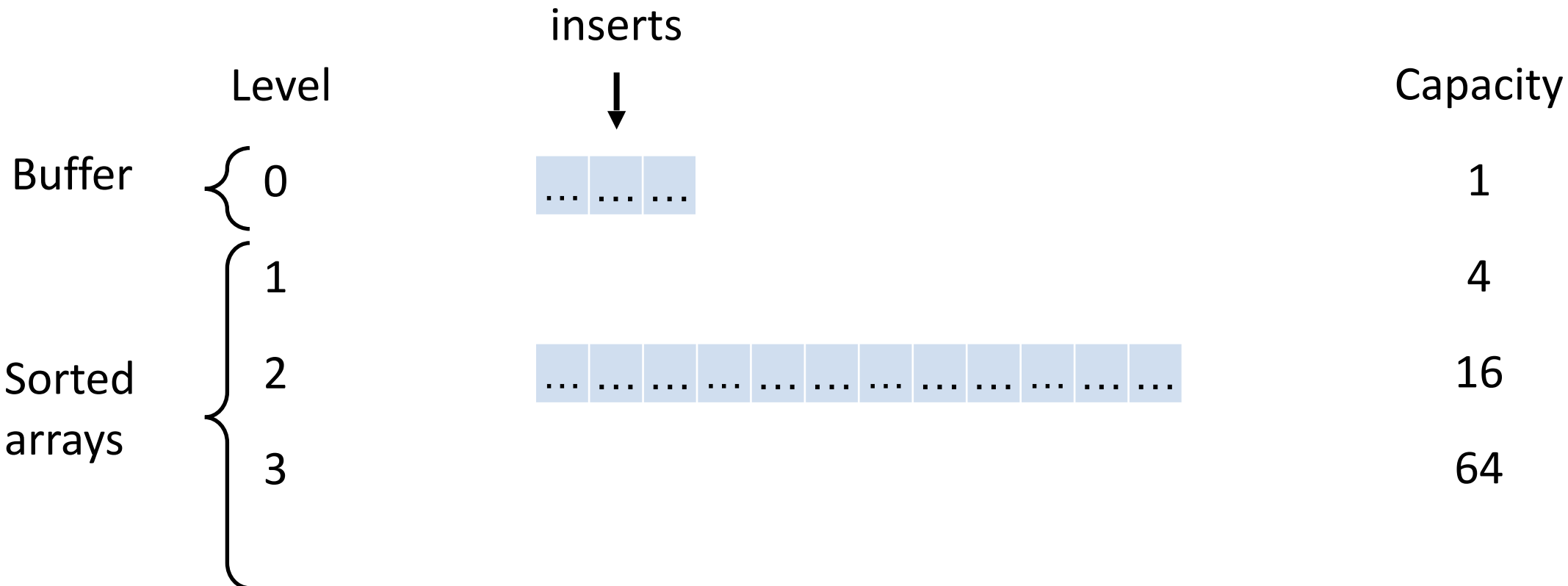
How to reduce it?

Increase size ratio T

E.g. size ratio of 4

Leveled LSM-tree

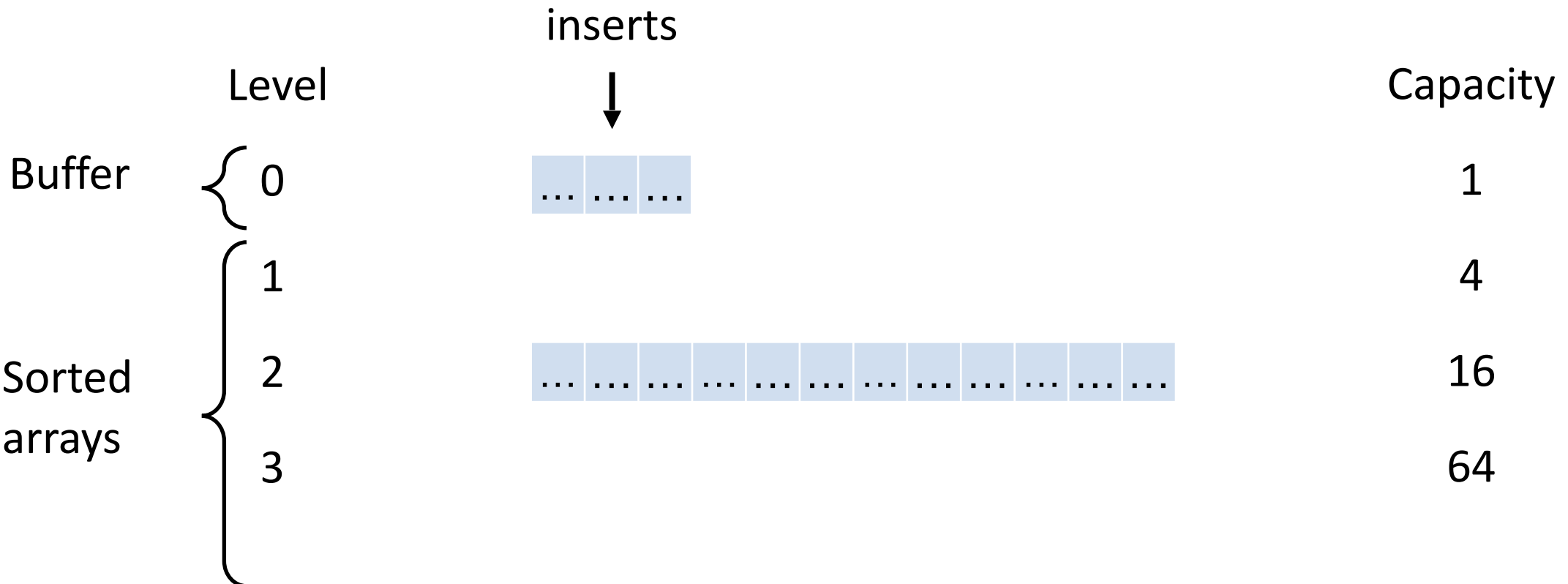
Lookup cost?



Leveled LSM-tree

Lookup cost?

$O(\log_T(N/P))$

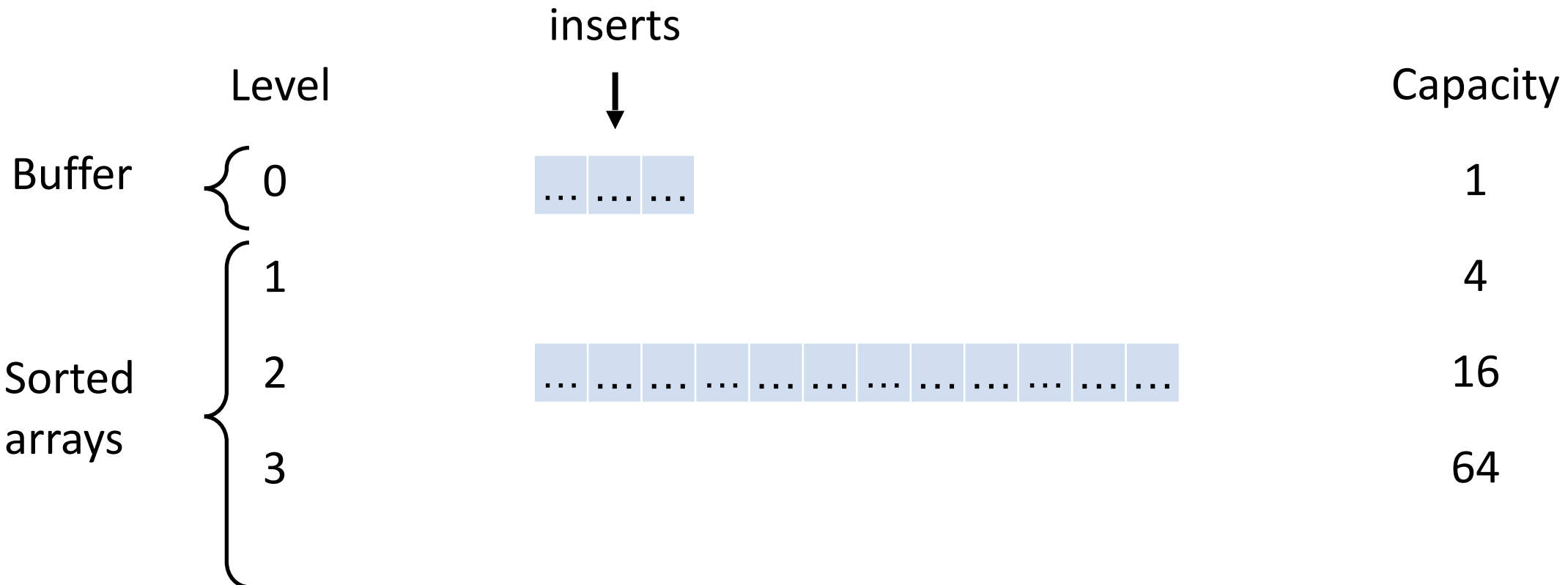


Leveled LSM-tree

Lookup cost?

$O(\log_T(N/P))$

Insertion cost?



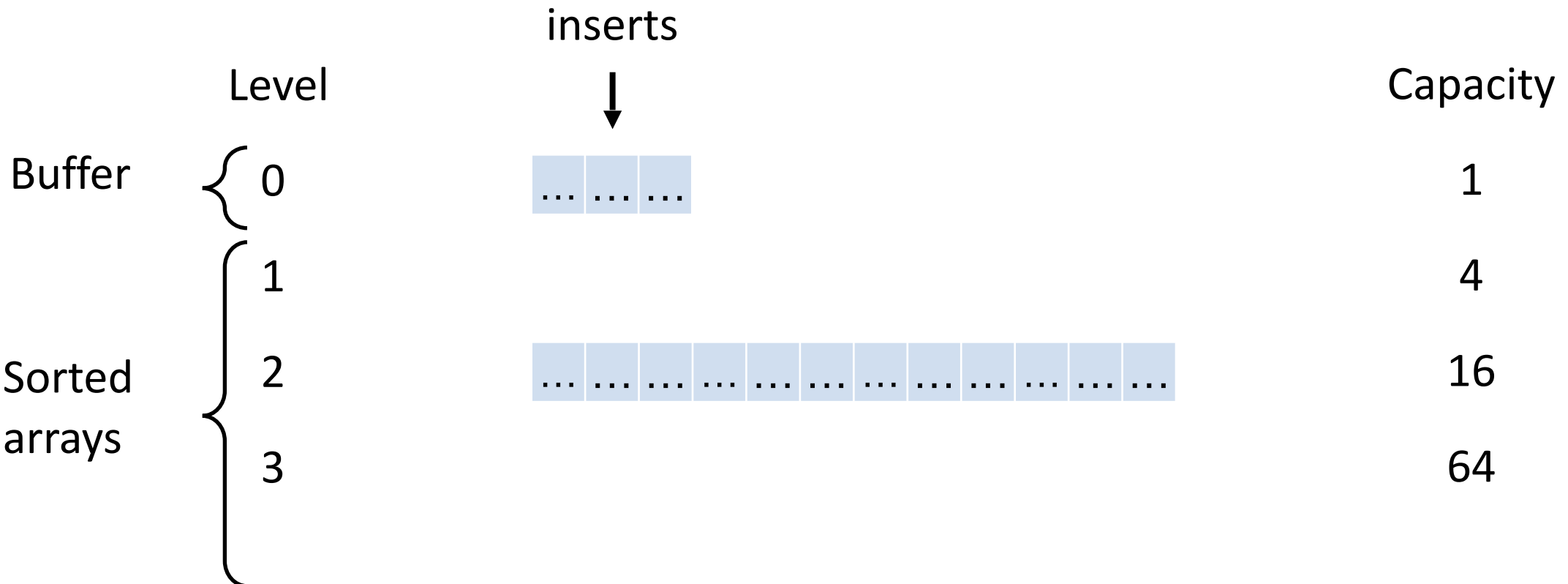
Leveled LSM-tree

Lookup cost?

$O(\log_T(N/P))$

Insertion cost?

$O(T/B \bullet \log_T(N/P))$



Leveled LSM-tree

Lookup cost?

$O(\log_T(N/P))$

Insertion cost?

$O(T/B \bullet \log_T(N/P))$

What happens as we increase the size ratio T?

Leveled LSM-tree

Lookup cost?

$O(\log_T(N/P))$

Insertion cost?

$O(T/B \bullet \log_T(N/P))$

What happens as we increase the size ratio T?

Leveled LSM-tree

Lookup cost?

$O(\log_T(N/P))$

Insertion cost?

$O(T/B \bullet \log_T(N/P))$

What happens as we increase the size ratio T?

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

Leveled LSM-tree

Lookup cost?

$O(\log_T(N/P))$

Insertion cost?

$O(T/B \bullet \log_T(N/P))$

What happens as we increase the size ratio T?

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

Lookup cost becomes:

$O(1)$

Insert cost becomes:

$O(N/(B \bullet P))$

Leveled LSM-tree

Lookup cost?

$O(\log_T(N/P))$

Insertion cost?

$O(T/B \bullet \log_T(N/P))$

What happens as we increase the size ratio T?

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

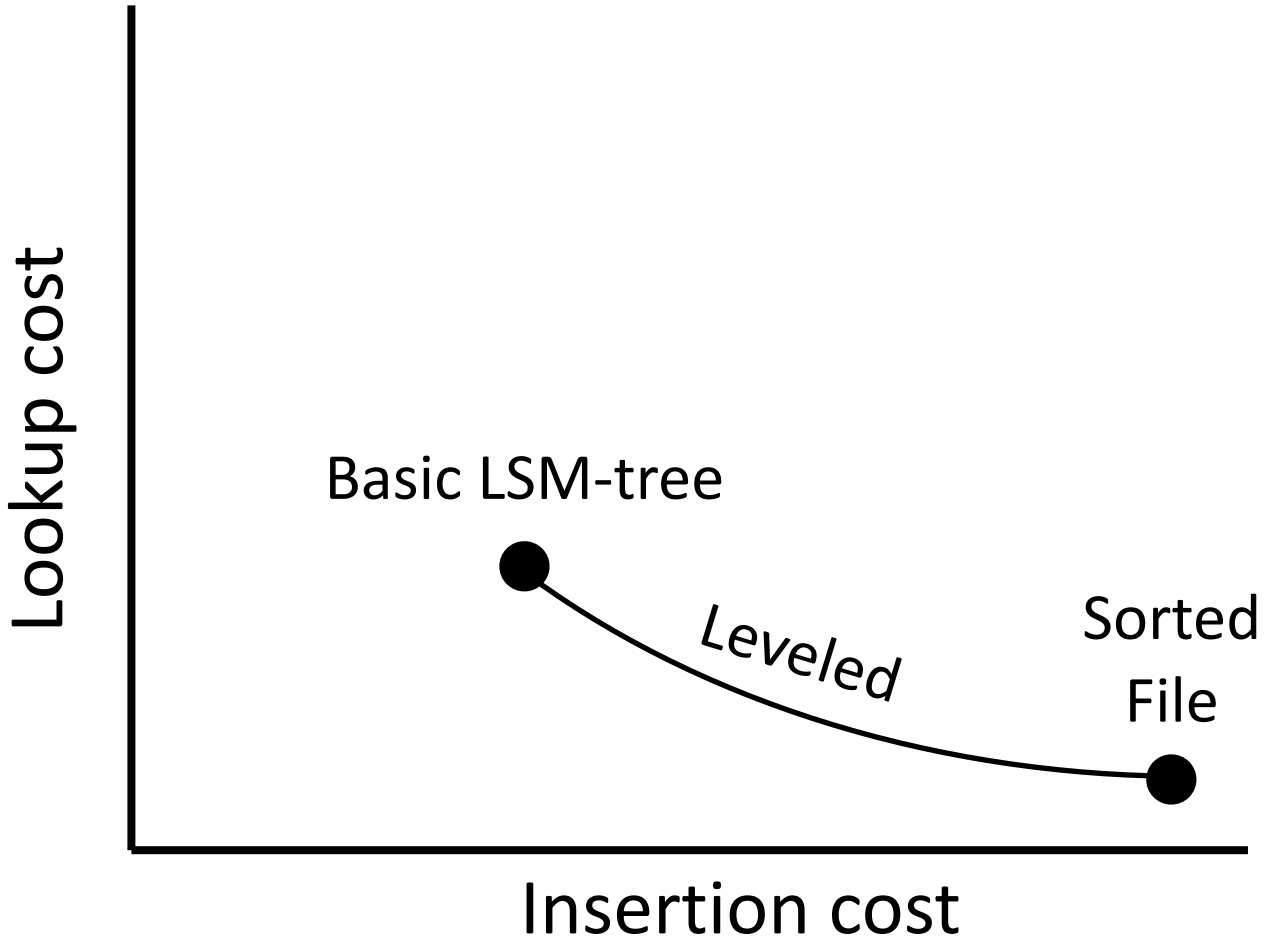
Lookup cost becomes:

$O(1)$

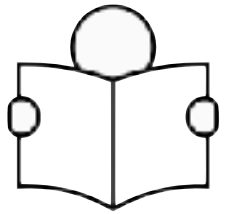
Insert cost becomes:

$O(N/(B \bullet P))$

The LSM-tree becomes a sorted file!


Lookup cost
Basic LSM-tree
Leveled
Sorted
File
Insertion cost

Leveled LSM-tree



Basic LSM-tree

Tiered LSM-tree

Tiered LSM-tree

 Lookup cost

 Insertion cost

Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.

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.

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

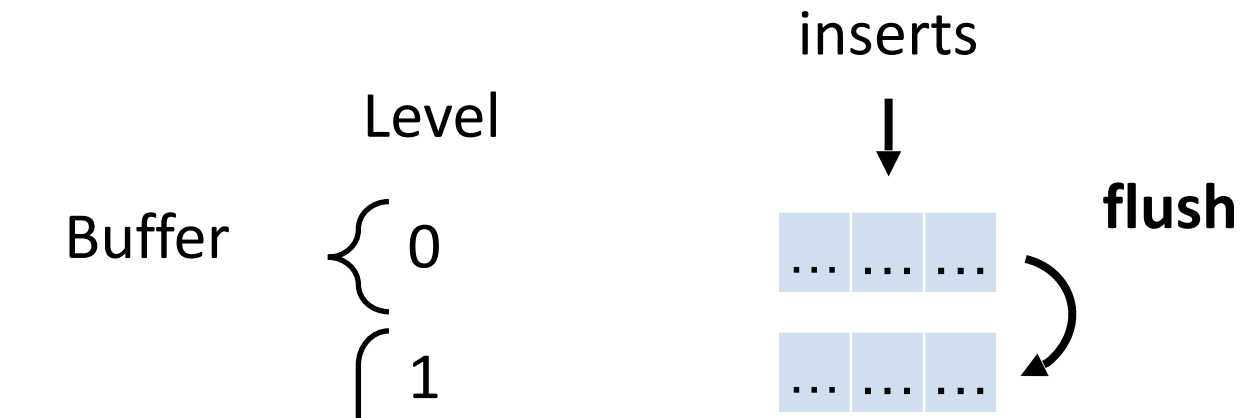
	Level		Capacity
Buffer	0	… … …	1
Sorted arrays	1		4
	2		16
	3		64

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

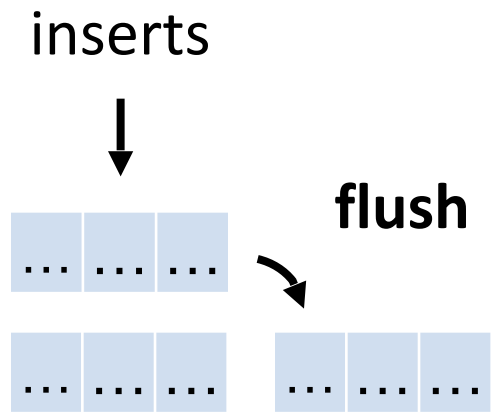
Tiered LSM-tree

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



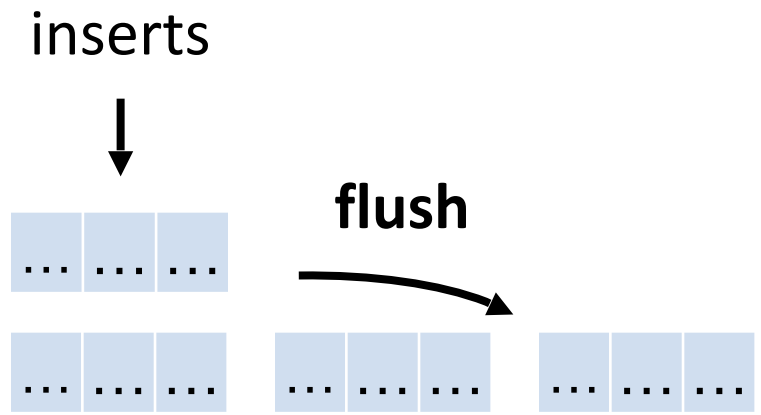
Tiered LSM-tree

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



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

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

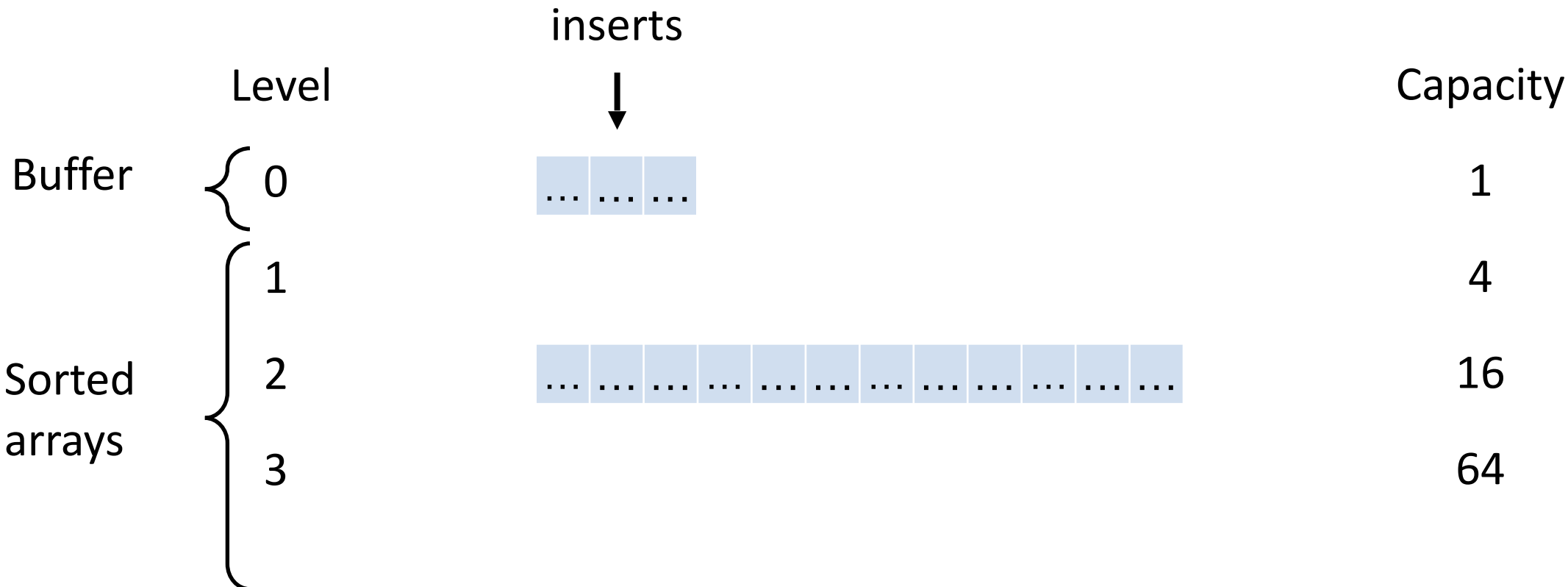
Tiered LSM-tree

Lookup cost?

Tiered LSM-tree

Lookup cost?

$O(T \bullet \log_T(N/P))$

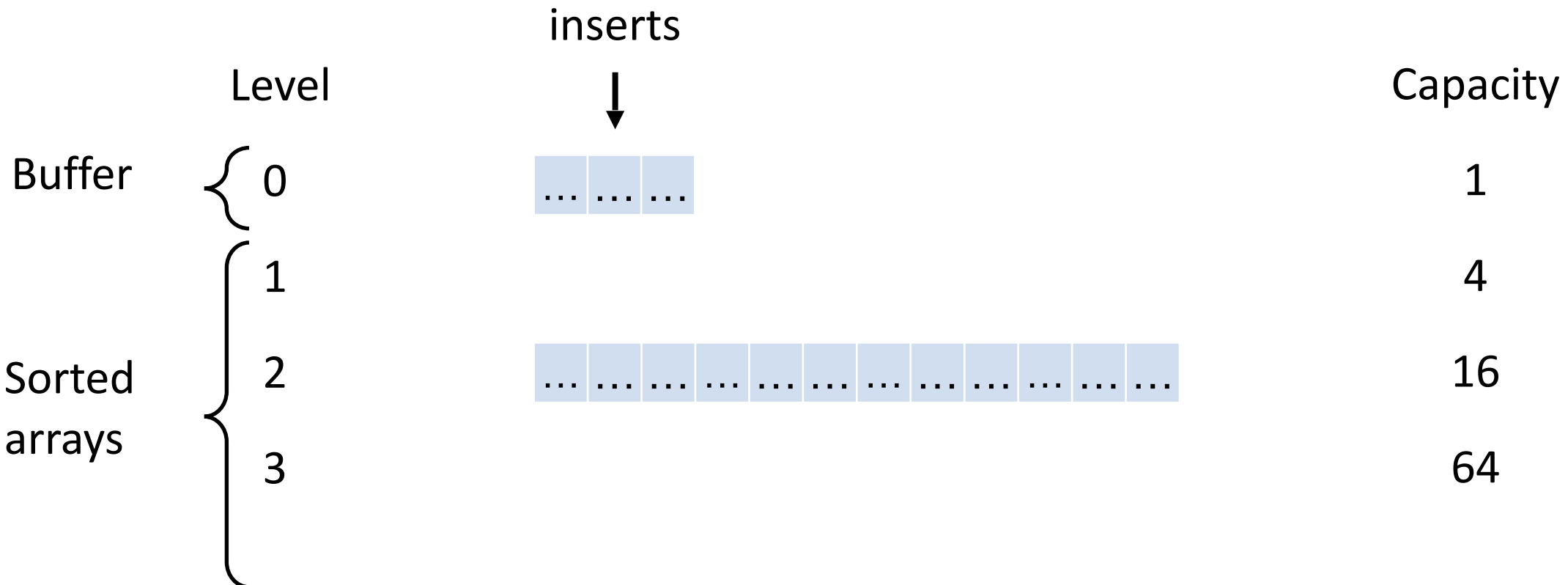


Tiered LSM-tree

Lookup cost?

$O(T \bullet \log_T(N/P))$

Insertion cost?



Tiered LSM-tree

Lookup cost?

$O(T \bullet \log_T(N/P))$

Insertion cost?

$O(1/B \bullet \log_T(N/P))$

Tiered LSM-tree

Lookup cost?

$O(T \bullet \log_T(N/P))$

Insertion cost?

$O(1/B \bullet \log_T(N/P))$

What happens as we increase the size ratio T?

Tiered LSM-tree

Lookup cost?

$O(T \bullet \log_T(N/P))$

Insertion cost?

$O(1/B \bullet \log_T(N/P))$

What happens as we increase the size ratio T?

Tiered LSM-tree

Lookup cost?

$O(T \bullet \log_T(N/P))$

Insertion cost?

$O(1/B \bullet \log_T(N/P))$

What happens as we increase the size ratio T?

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

Tiered LSM-tree

Lookup cost?

$O(T \bullet \log_T(N/P))$

Insertion cost?

$O(1/B \bullet \log_T(N/P))$

What happens as we increase the size ratio T?

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

Lookup cost becomes:

$O(N/P)$

Insert cost becomes:

$O(1/B)$

Tiered LSM-tree

Lookup cost?

$O(T \bullet \log_T(N/P))$

Insertion cost?

$O(1/B \bullet \log_T(N/P))$

What happens as we increase the size ratio T?

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

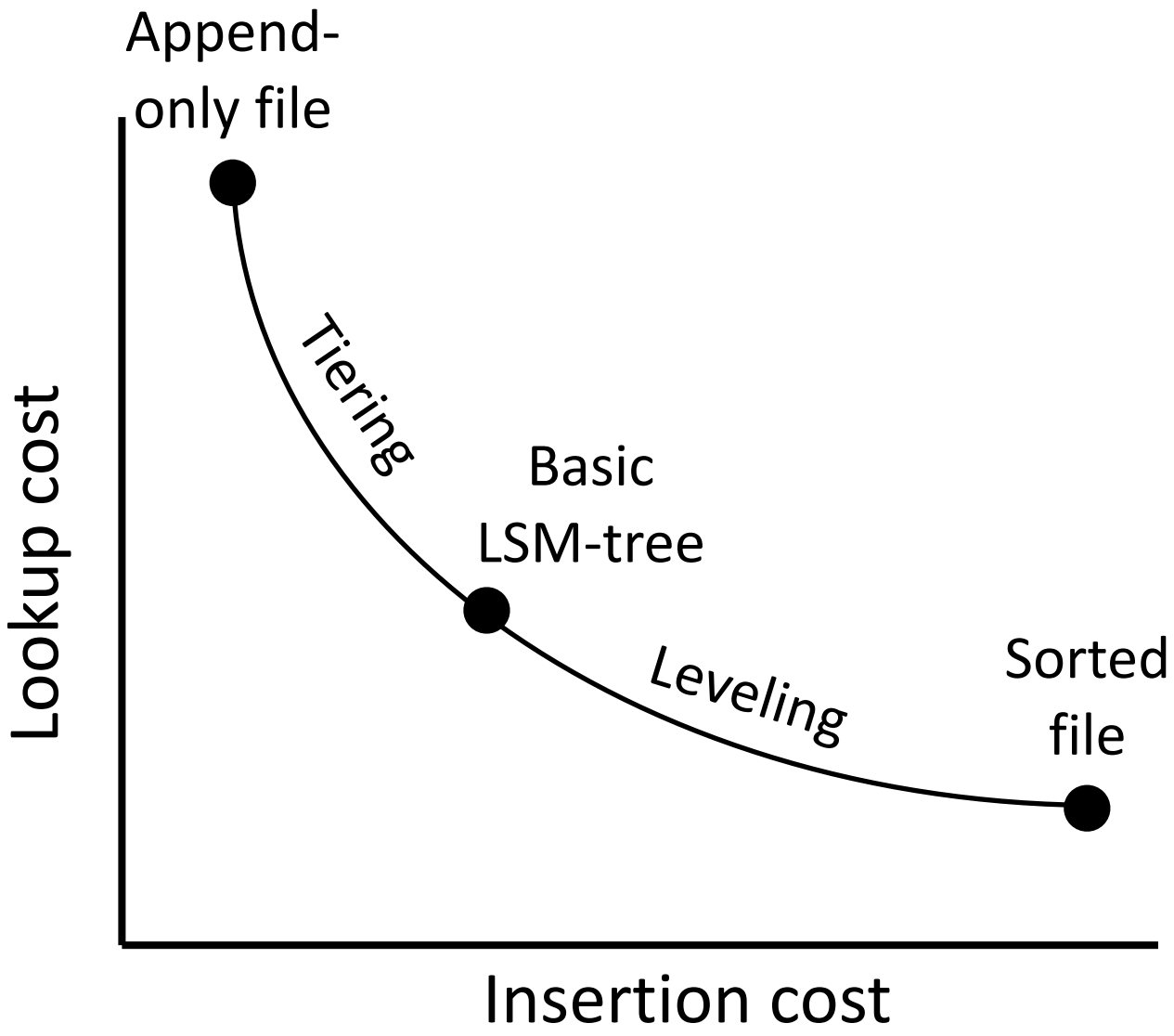
Lookup cost becomes:

$O(N/P)$

Insert cost becomes:

$O(1/B)$

The tiered LSM-tree becomes an append-only file!


Append-
only file
Tiering
Basic
LSM-tree
Leveling
Sorted
file
Lookup cost
Insertion cost

Cost Metric	Unit	**Unordered File**	**Tiered LSM-tree**	**Leveled LSM-tree**	**B-Tree**
Query	Reads IOs	O(N/B)	O(L * T)	O(L)	O(1)
Insert	Reads IOs	0	O(L/B)	O((L * T)/B)	O(1)
	Write IOs	O(1/B)	O(L/B)	O((L * T)/B)	O(1) & GC
Memory	#entries	O(B)	O(N/B)	O(N/B)	O(N/B)

Let $L = \log_T(N/P)$

This table assumes internal nodes fit in memory.

Conclusions - LSM-trees are:

Write-optimized

Conclusions - LSM-trees are:

Write-optimized

Highly tunable

Conclusions - LSM-trees are:

Write-optimized

Highly tunable

Backbone of many modern systems

Conclusions - LSM-trees are:

Write-optimized

Highly tunable

Backbone of many modern systems

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

Conclusions - LSM-trees are:

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)