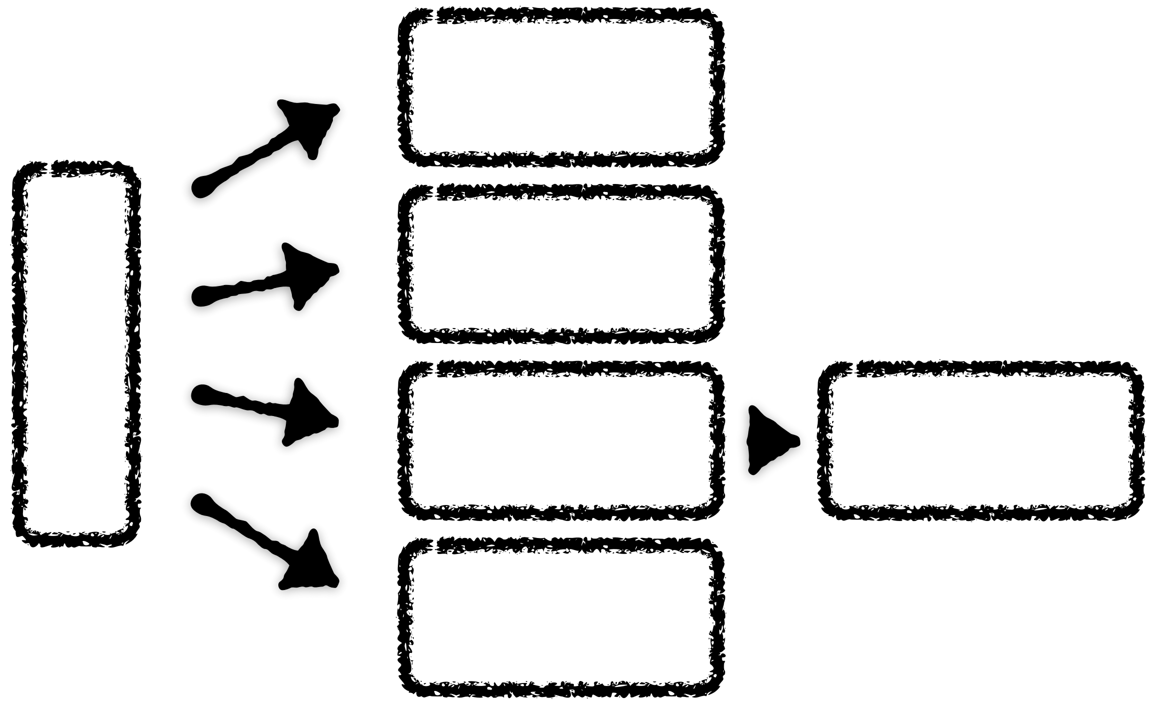
Circular Logs & Cuckoo Filters

CSC443H1 Database System Technology

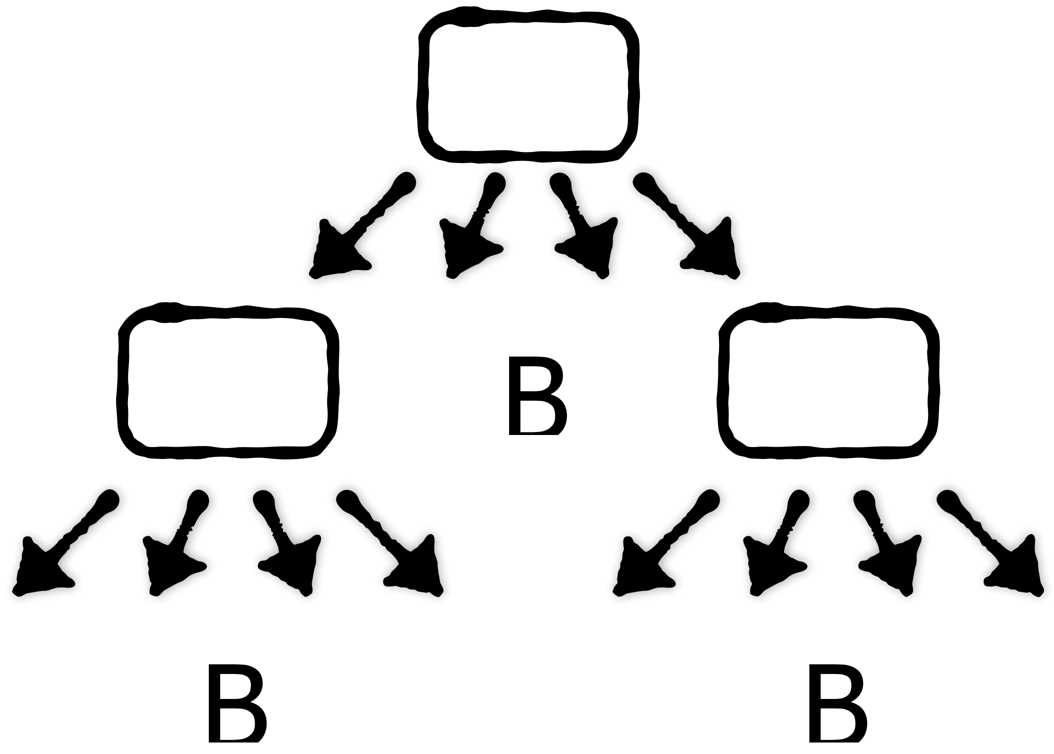
Niv Dayan

Extendible Hashing



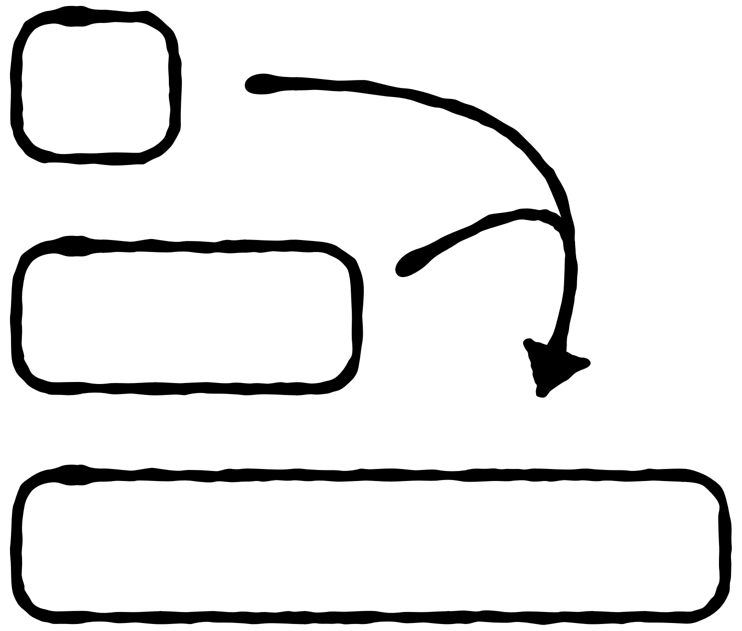
Cheapest gets
No scans

B-Trees



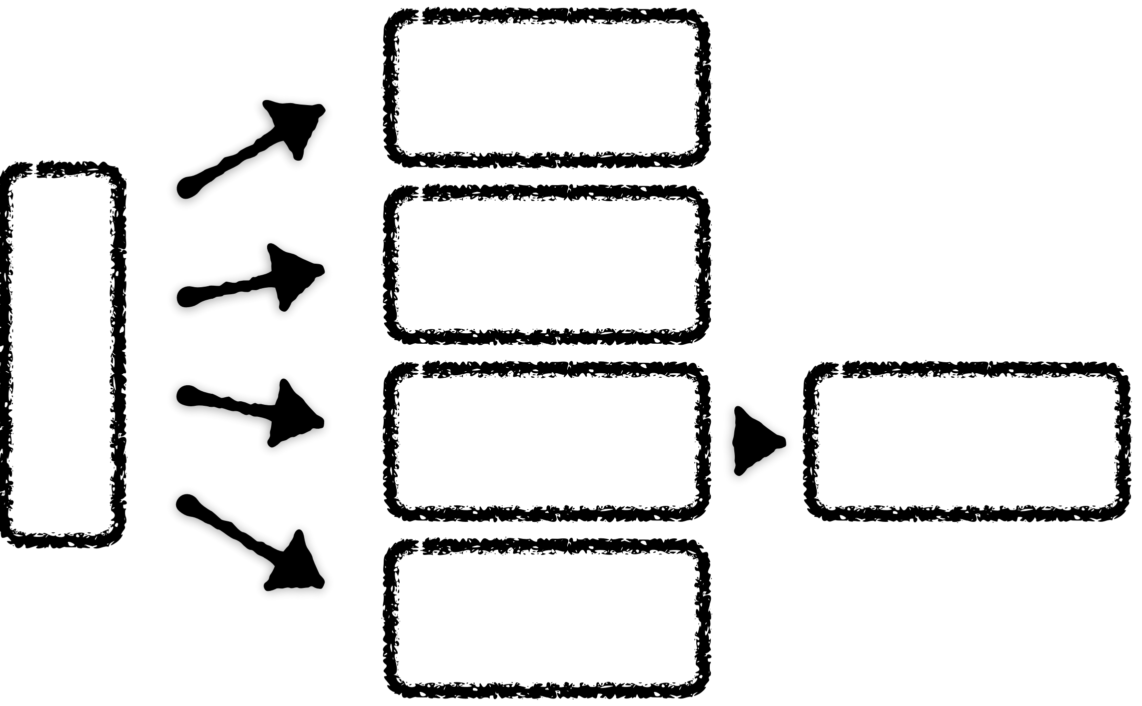
Cheap gets
& scans

LSM-tree



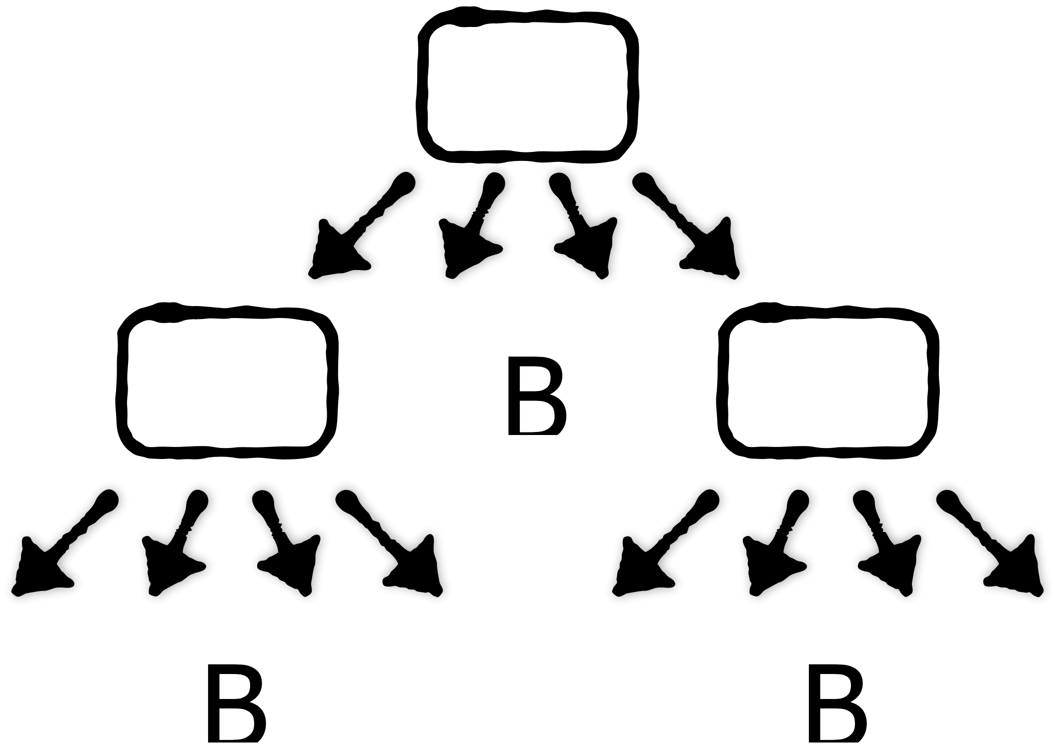
Cheaper writes
More memory

Extendible Hashing



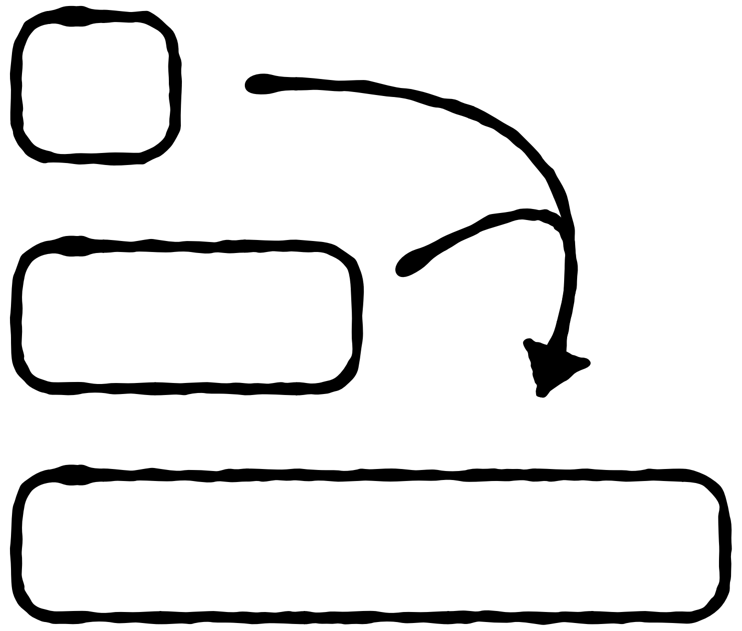
Cheapest gets
No scans

B-Trees



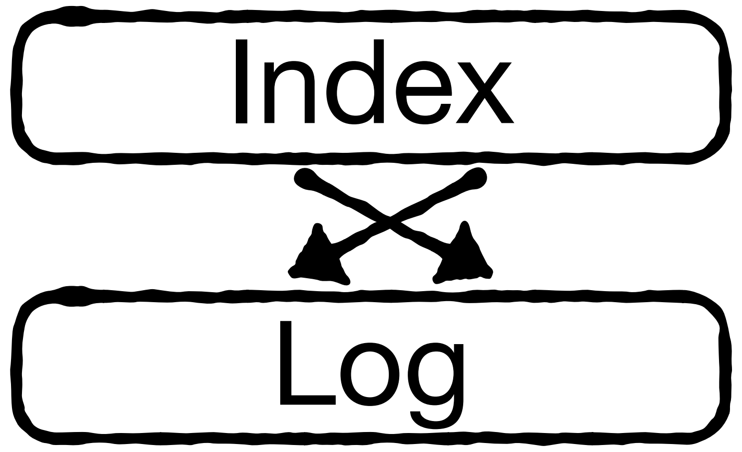
Cheap gets
& scans

LSM-tree



Cheaper writes
More memory

Circular Log



Cheapest writes
Most memory
No scans

Circular Log

Invented in 1992 as a “log structured file system”

Circular Log

Invented in 1992 as a “log structured file system”

File Systems

Flash Translation Layers

KV-stores

Various names, same data structure:

Index+Log

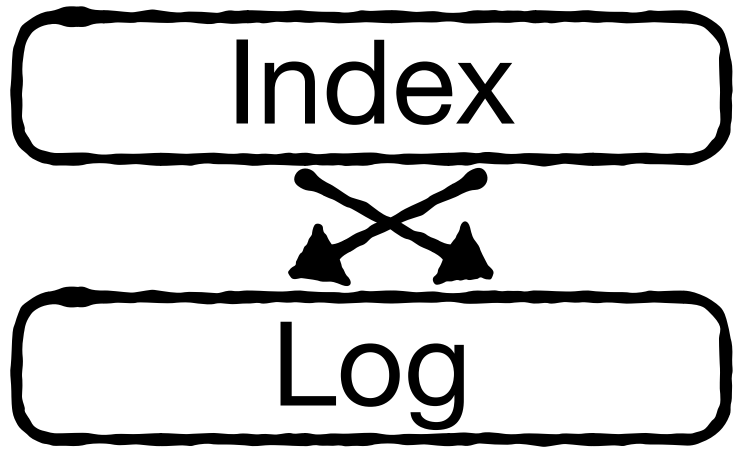
Circular Log

Log-structured Hash Table

log structured file system

Agenda

Mechanics



Hot/Cold separation

To reduce write-amp

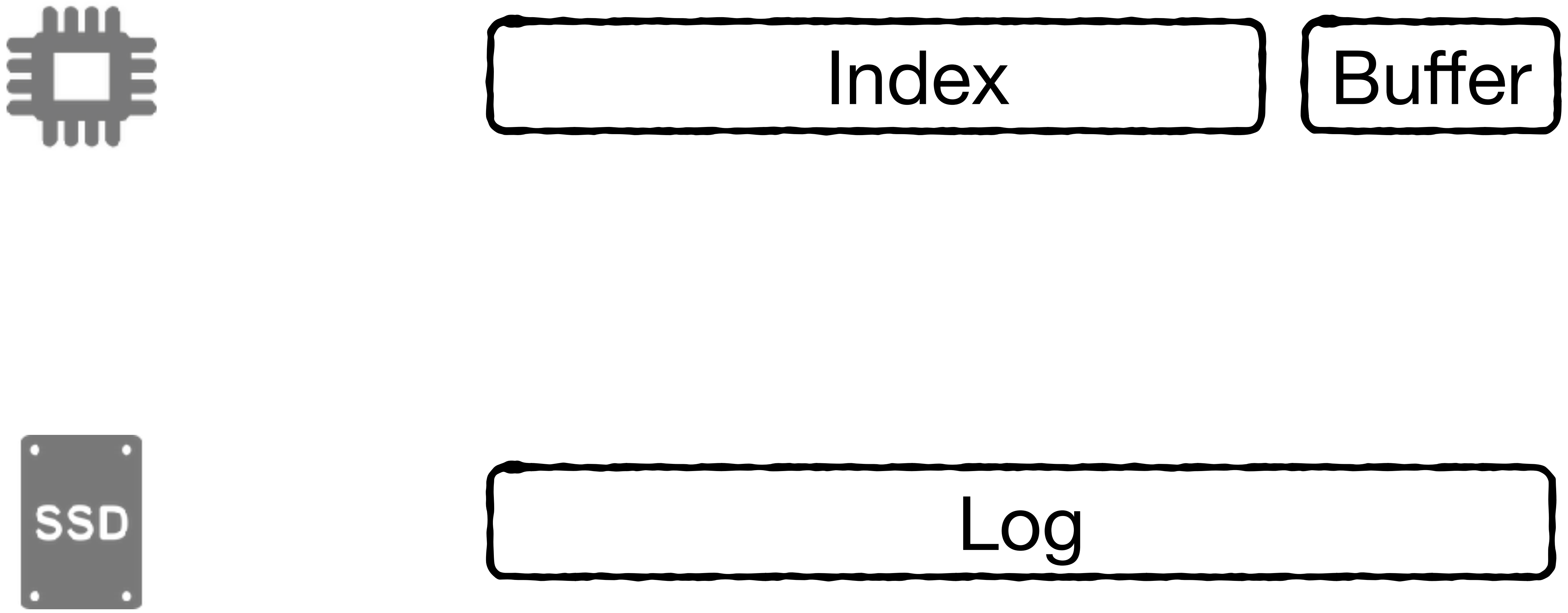
Checkpointing/ recovery

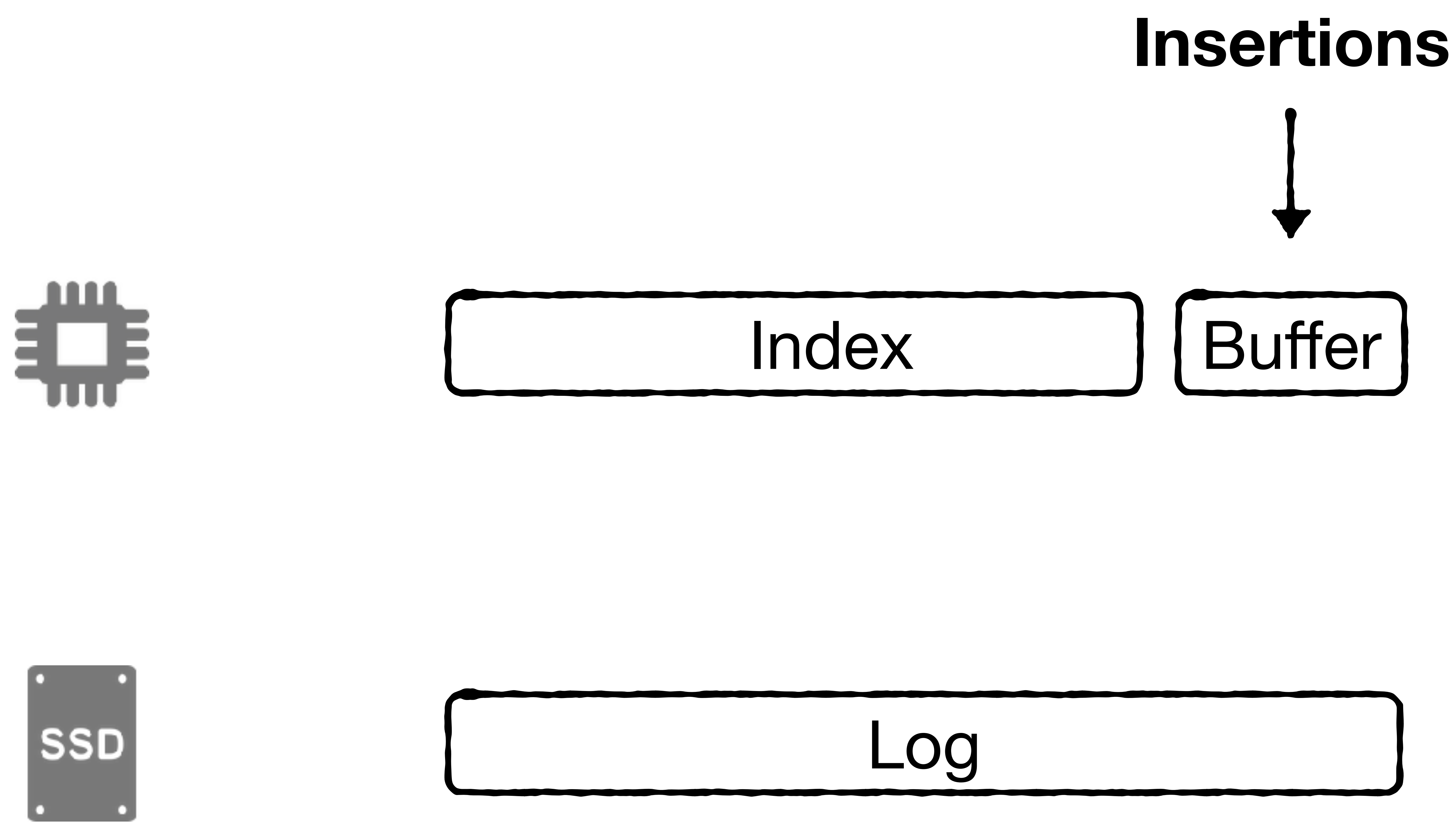
If power fails

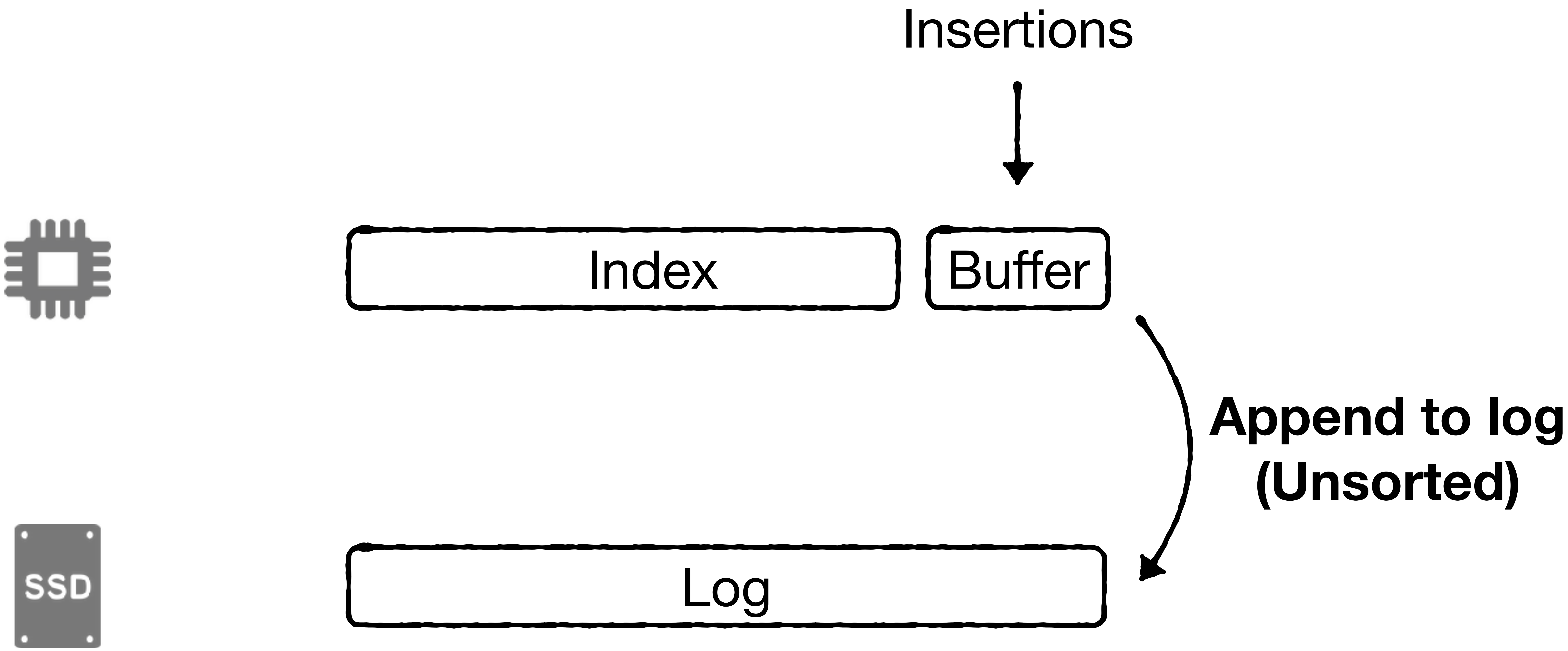
Cuckoo filtering

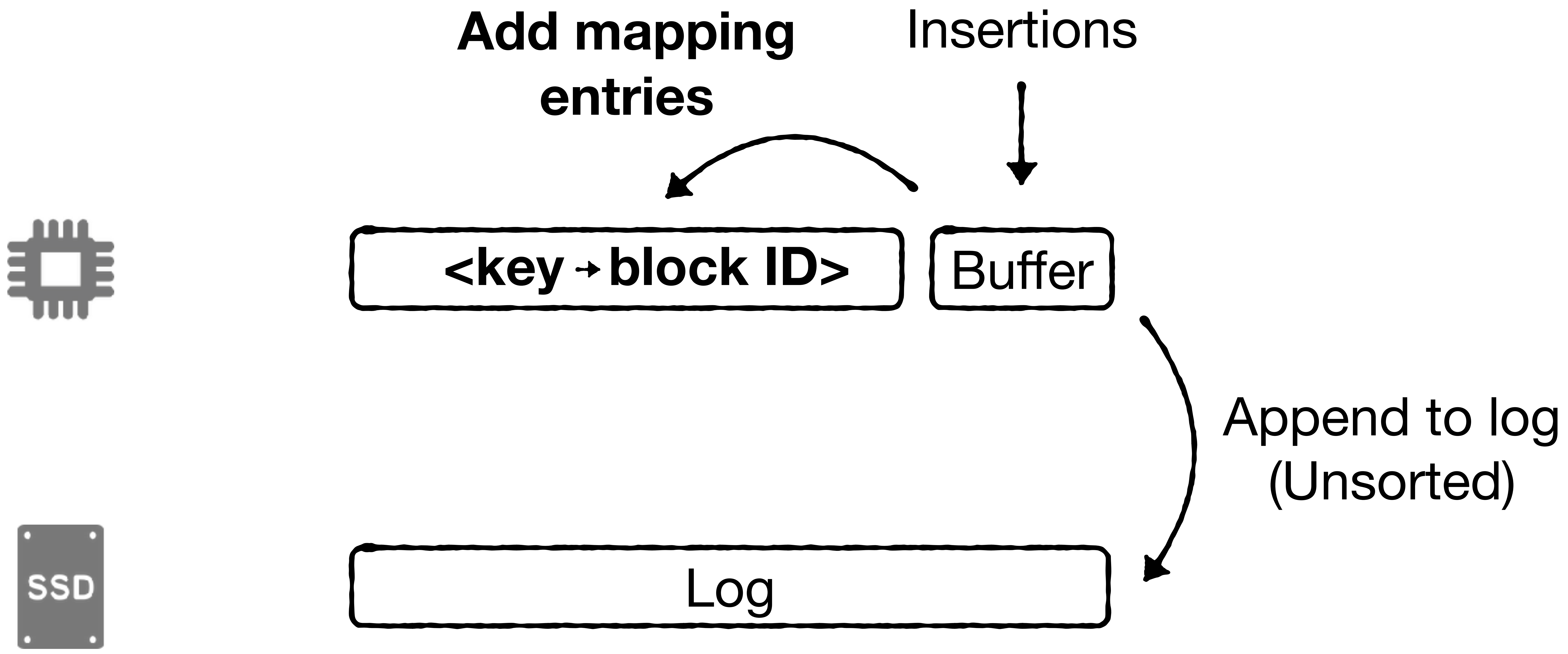
To reduce memory

Mechanics

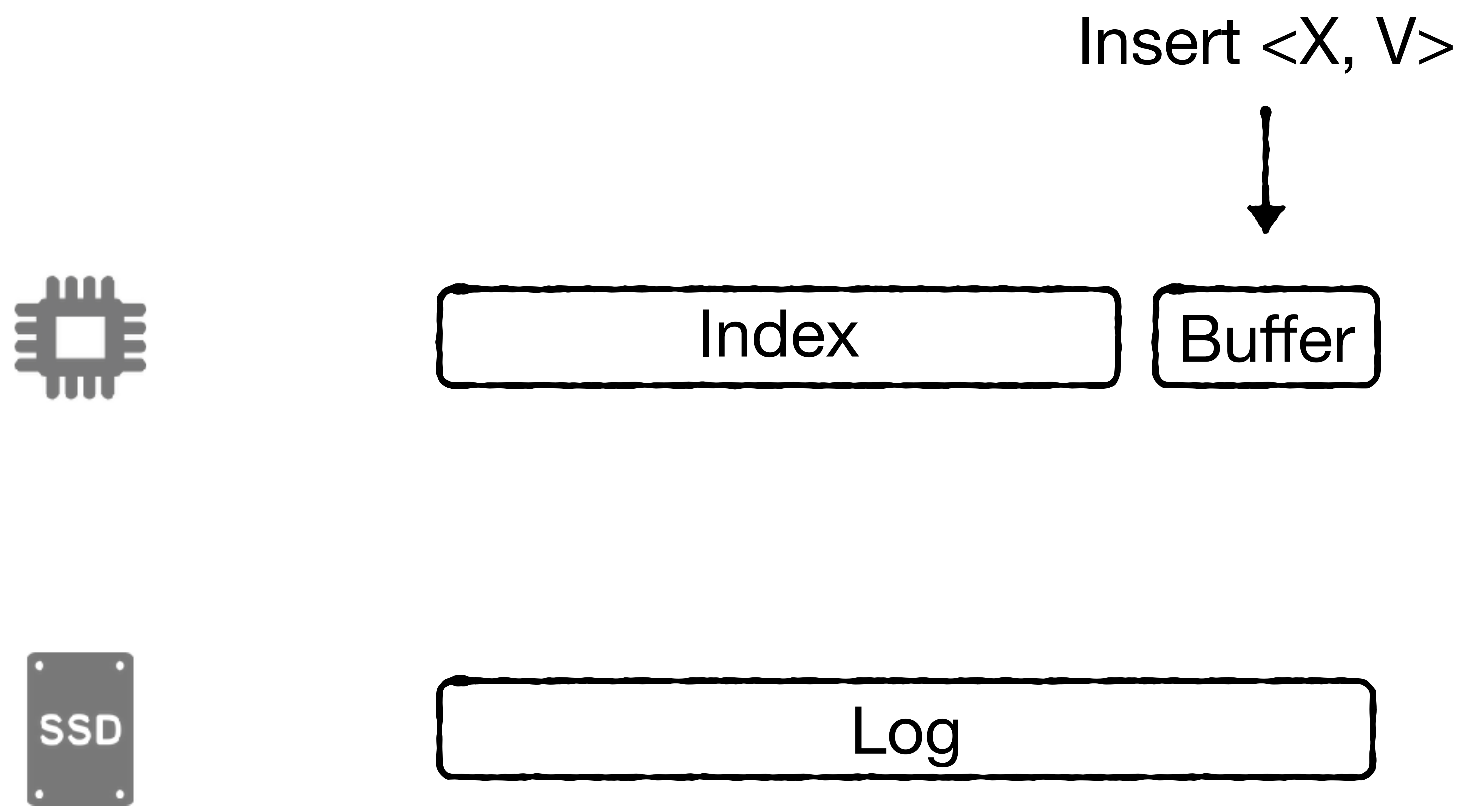


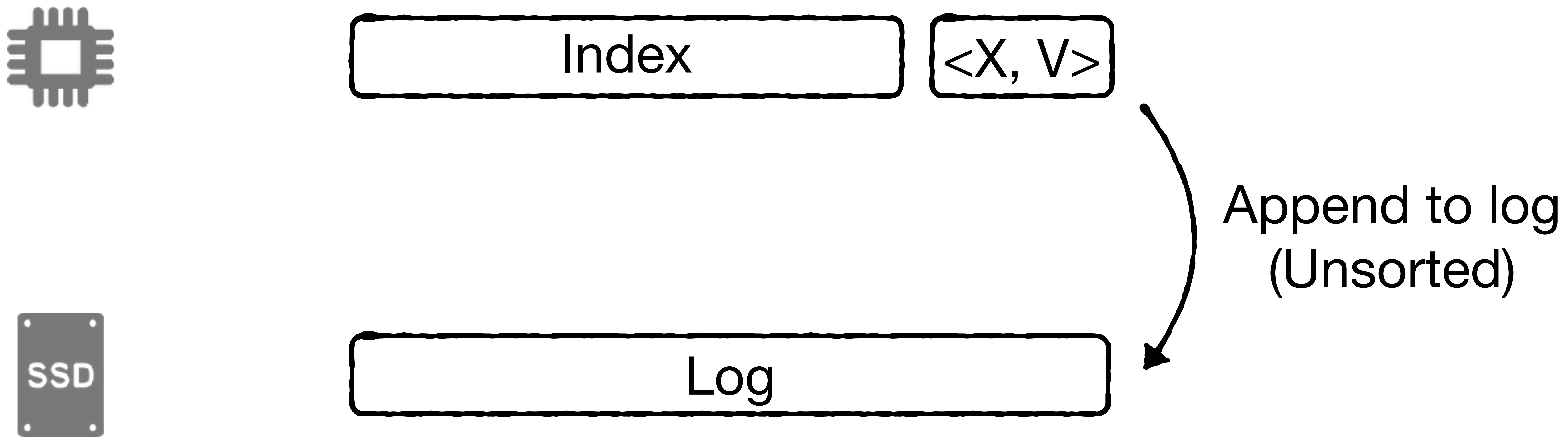

Insertions
Index
Buffer
SSD
Log

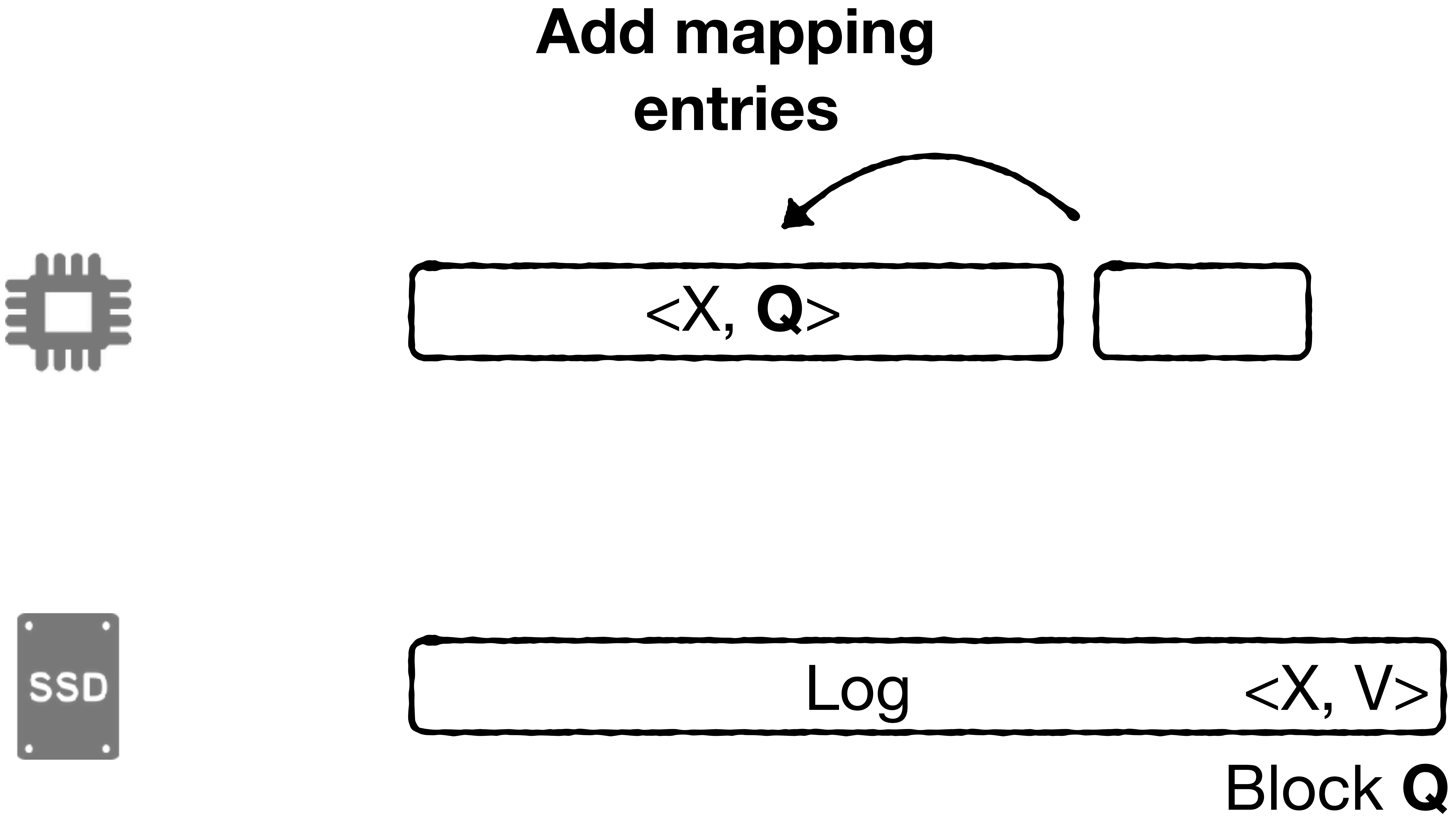

Insertions
Index
Buffer
Append to log
(Unsorted)
SSD
Log

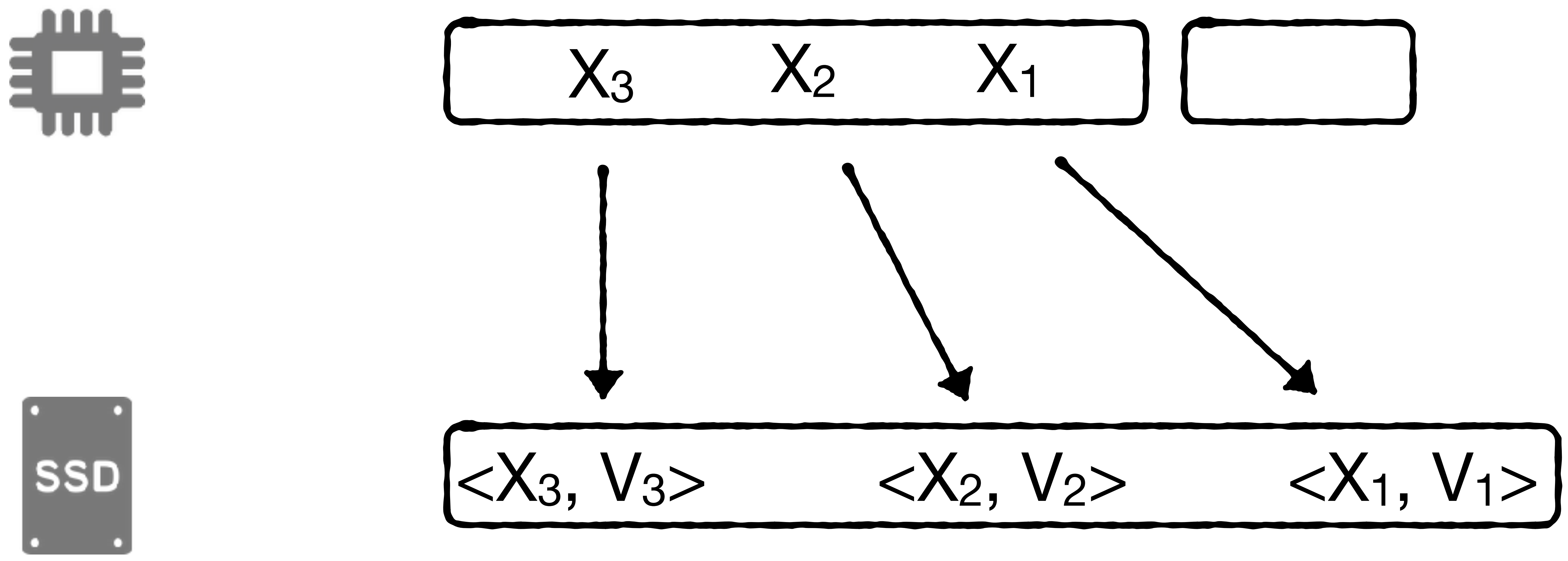


Add mapping entries
Insertions
<key→block ID>
Buffer
Append to log (Unsorted)
SSD
Log


Insert <X, V>
Index
Buffer
SSD
Log


Index
<X, V>
Append to log
(Unsorted)
SSD
Log

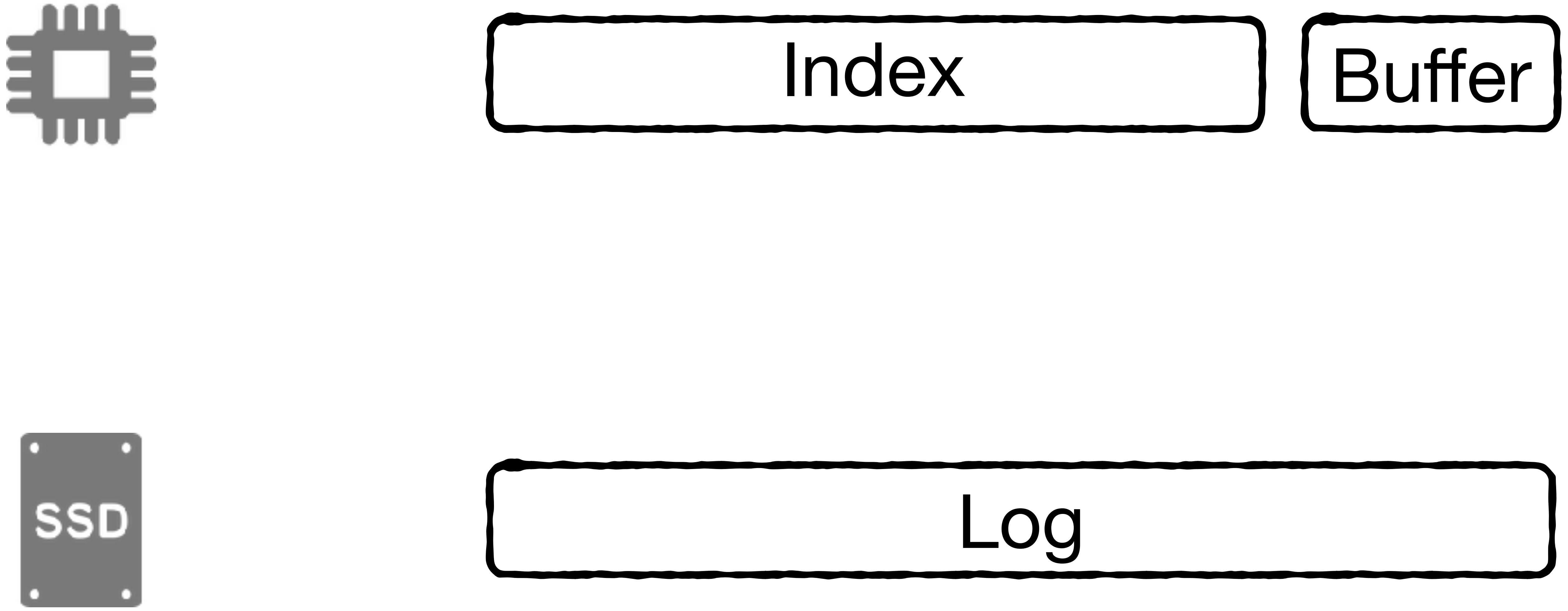

Add mapping entries
<X, **Q**>
SSD
Log
<X, V>
Block **Q**



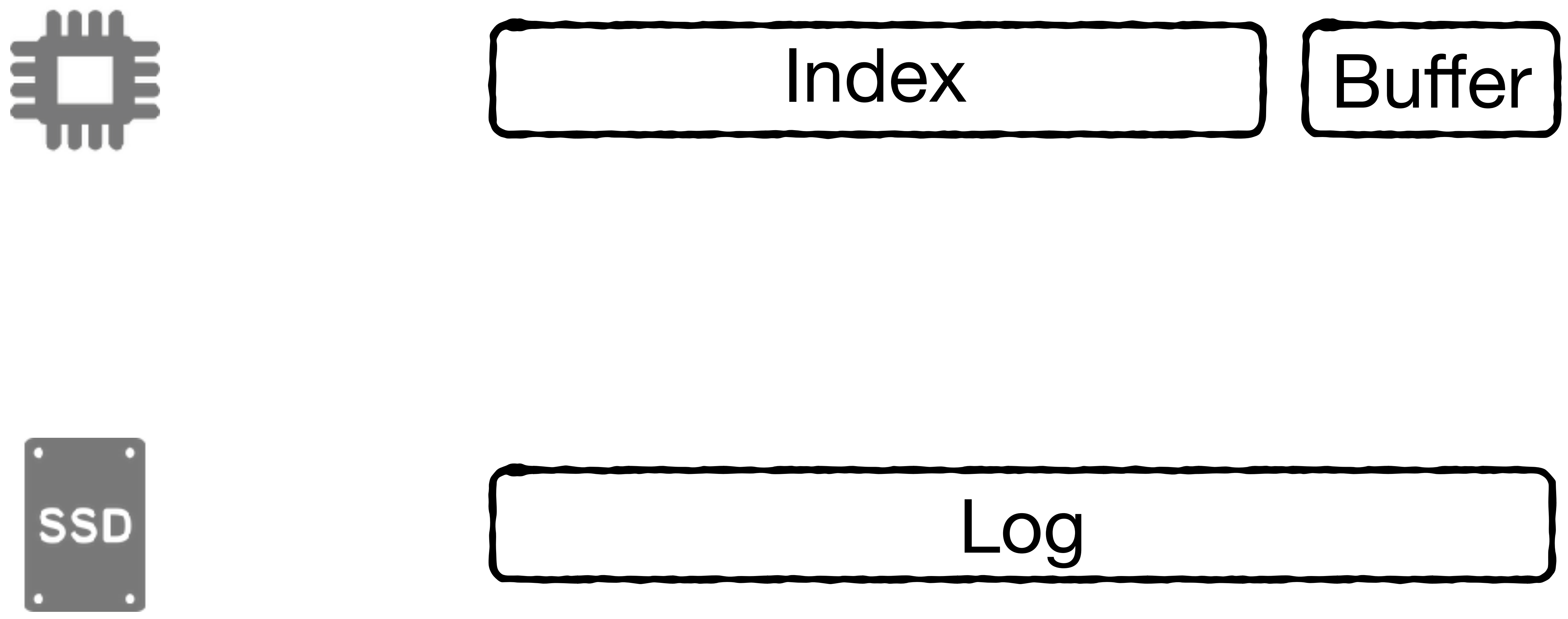
X_3 X_2 X_1
SSD
<X_3, V_3> <X_2, V_2> <X_1, V_1>

Write cost **(Assuming only insertions)**

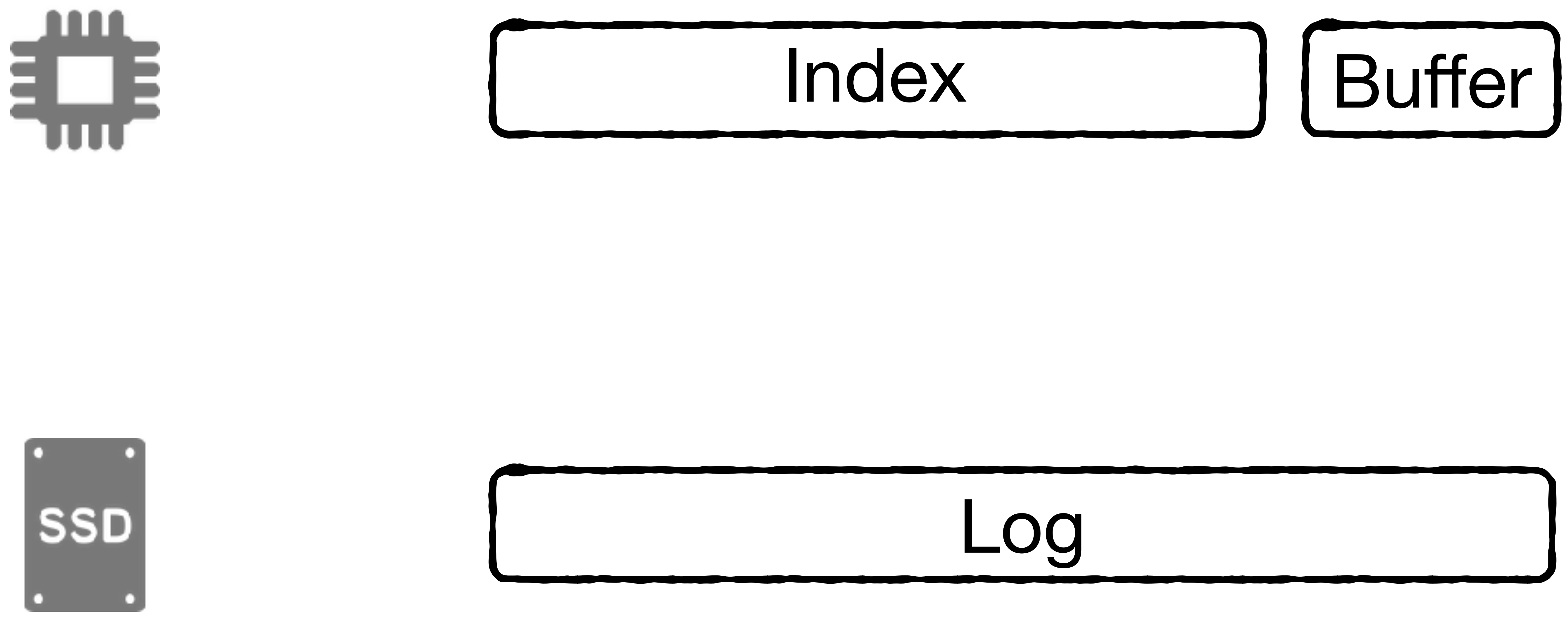
Get cost

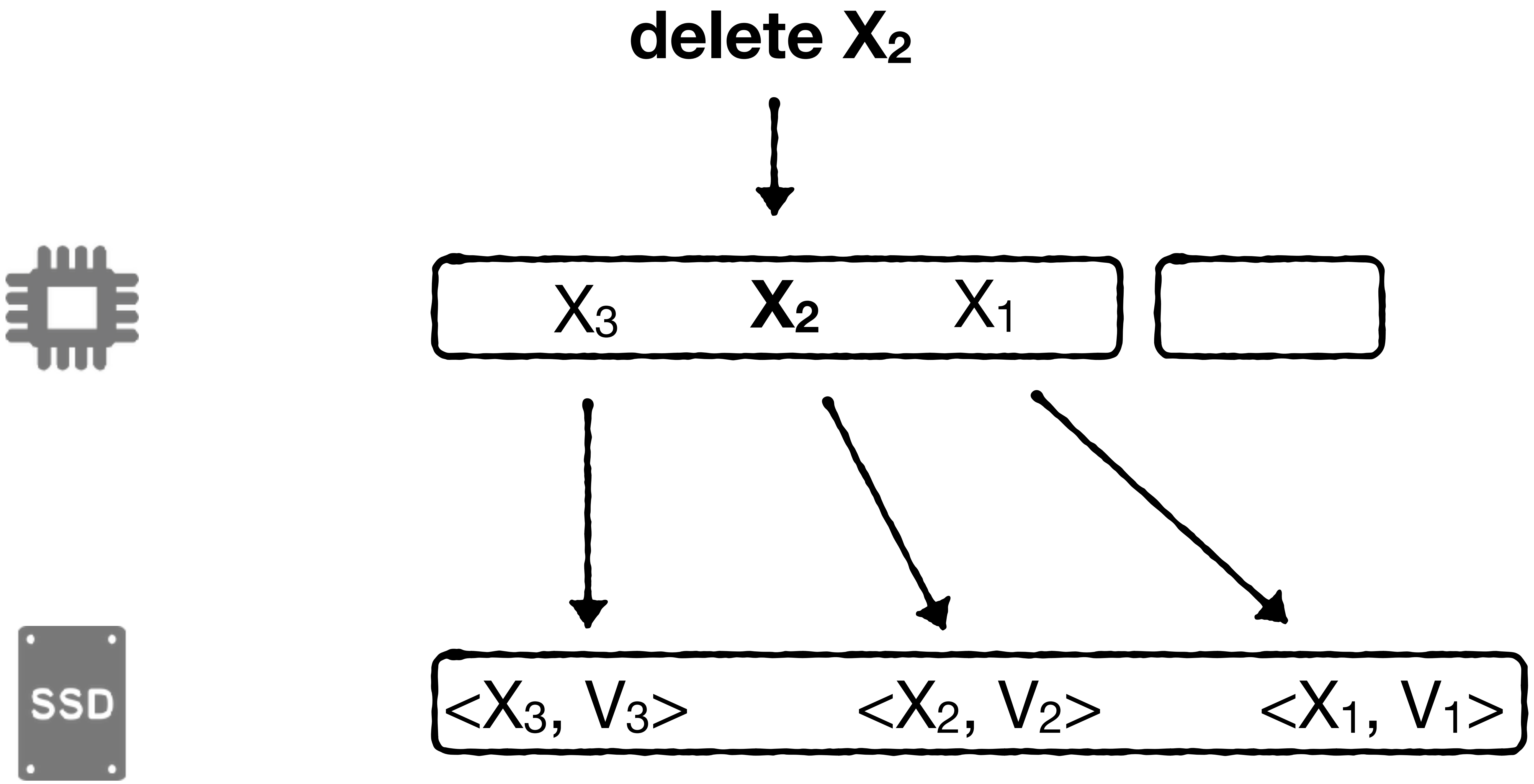


Write cost	**O(1/B)**	**(Assuming only insertions)**
Get cost	**O(1)**	

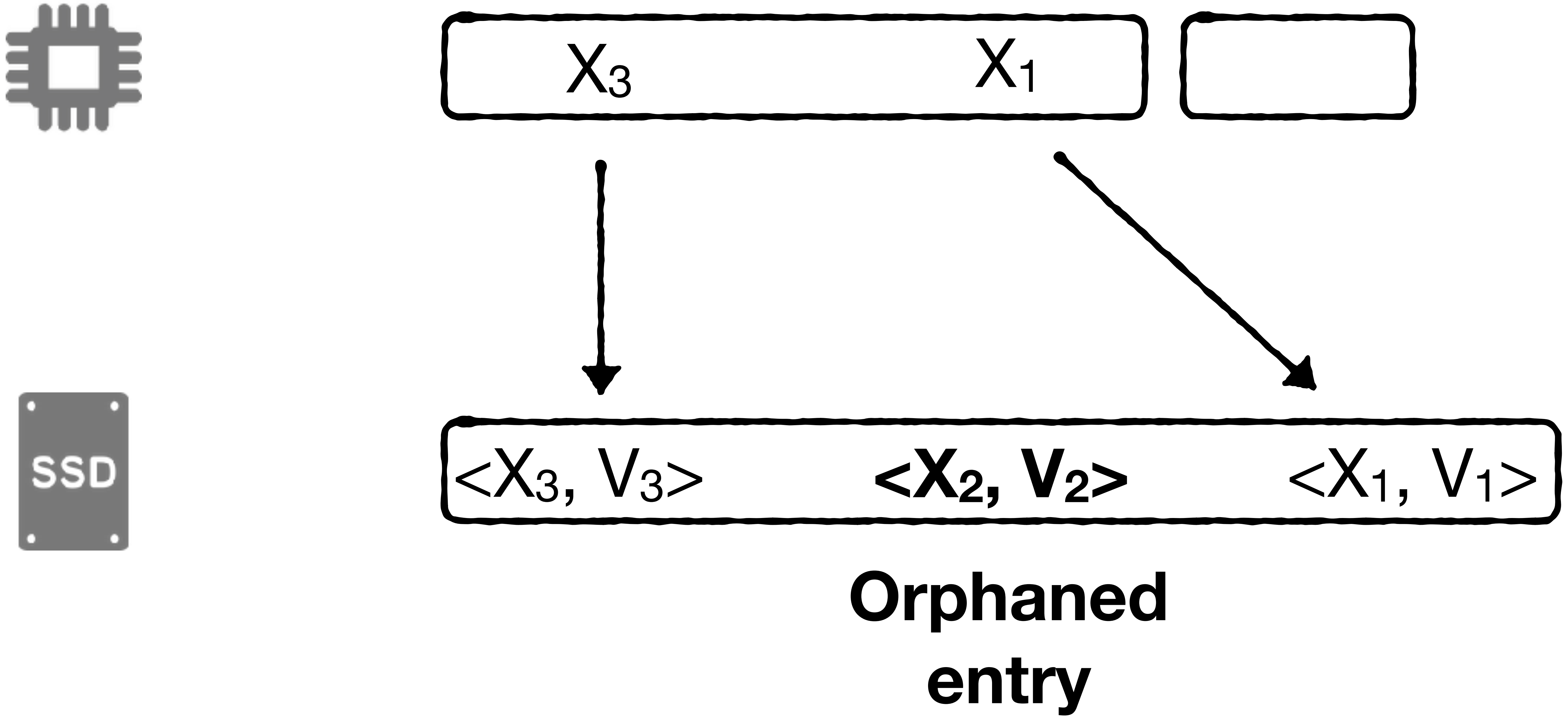


How to delete?



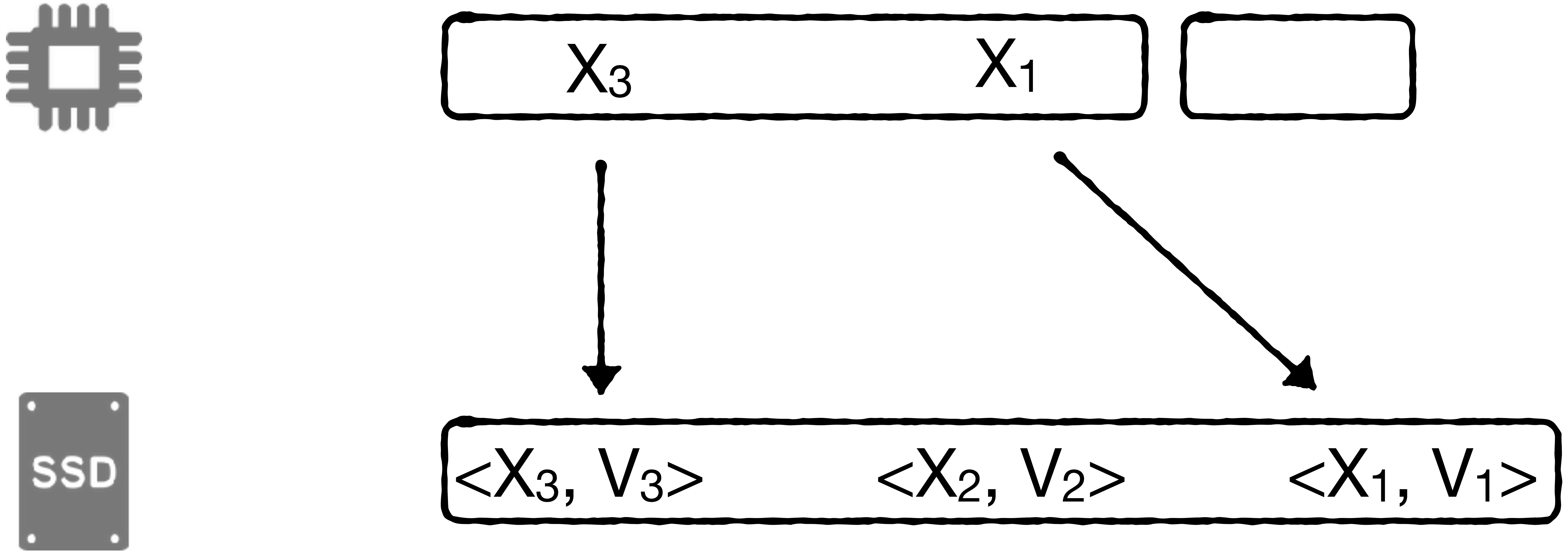


delete X_2
X_3 $\mathbf{X_2}$ X_1
SSD
<X_3, V_3>
<X_2, V_2>
<X_1, V_1>



SSD
X_3
X_1
$\langle X_3, V_3 \rangle$
$\langle \mathbf{X_2}, \mathbf{V_2} \rangle$
$\langle X_1, V_1 \rangle$
Orphaned entry

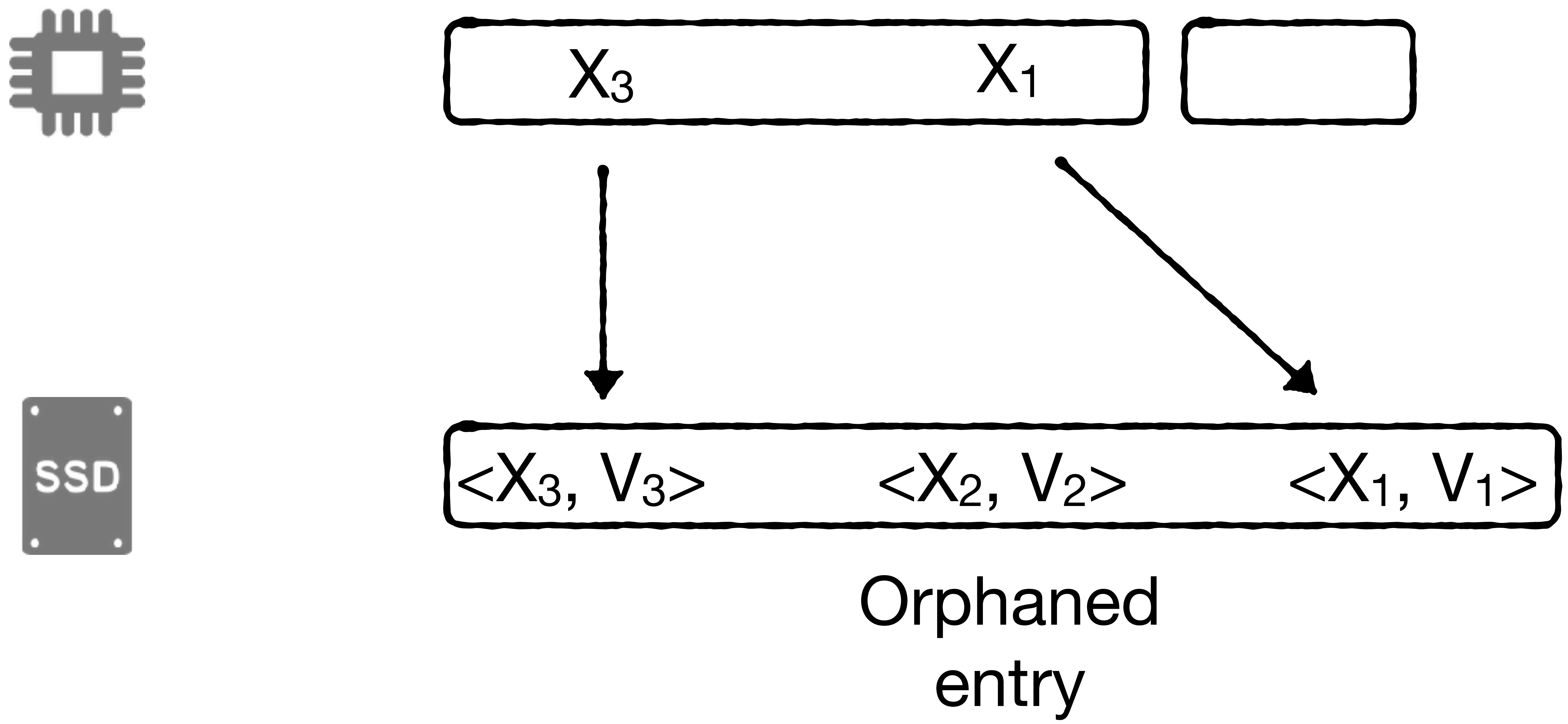
After many deletes, many orphaned entries accumulate



Orphaned
entry

After many deletes, many orphaned entries accumulate

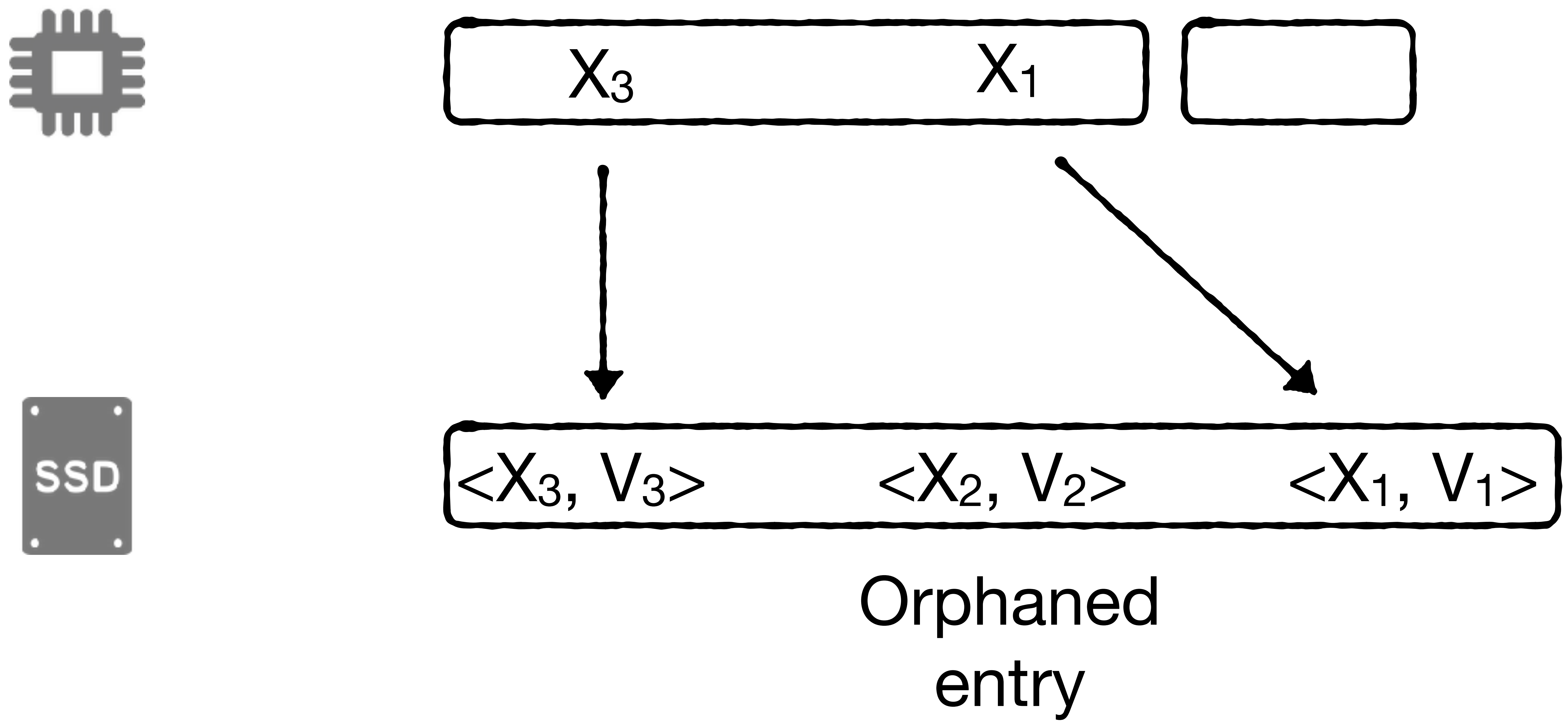
They take up space which we would prefer to use to store valid data



After many deletes, many orphaned entries accumulate

They take up space which we would prefer to use to store valid data

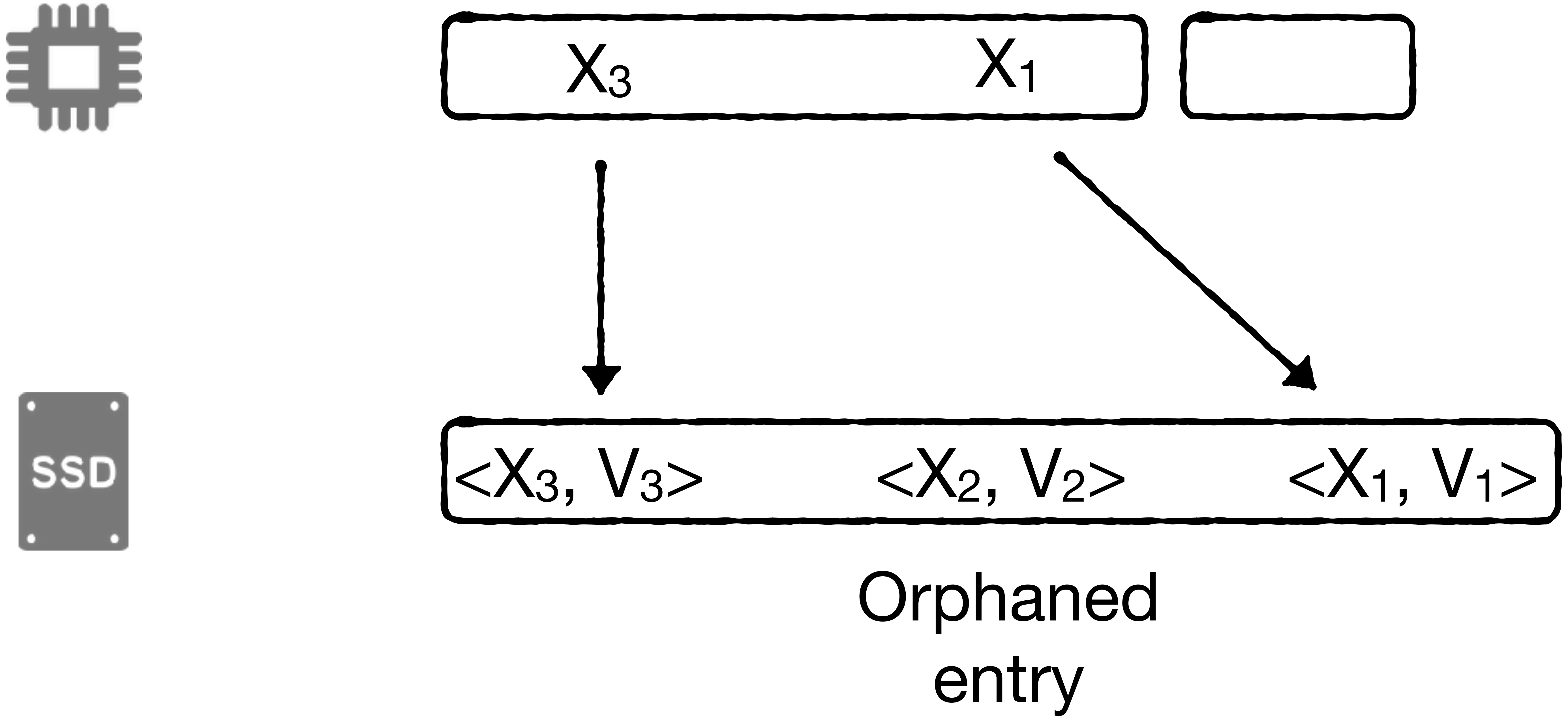
How to fix this?



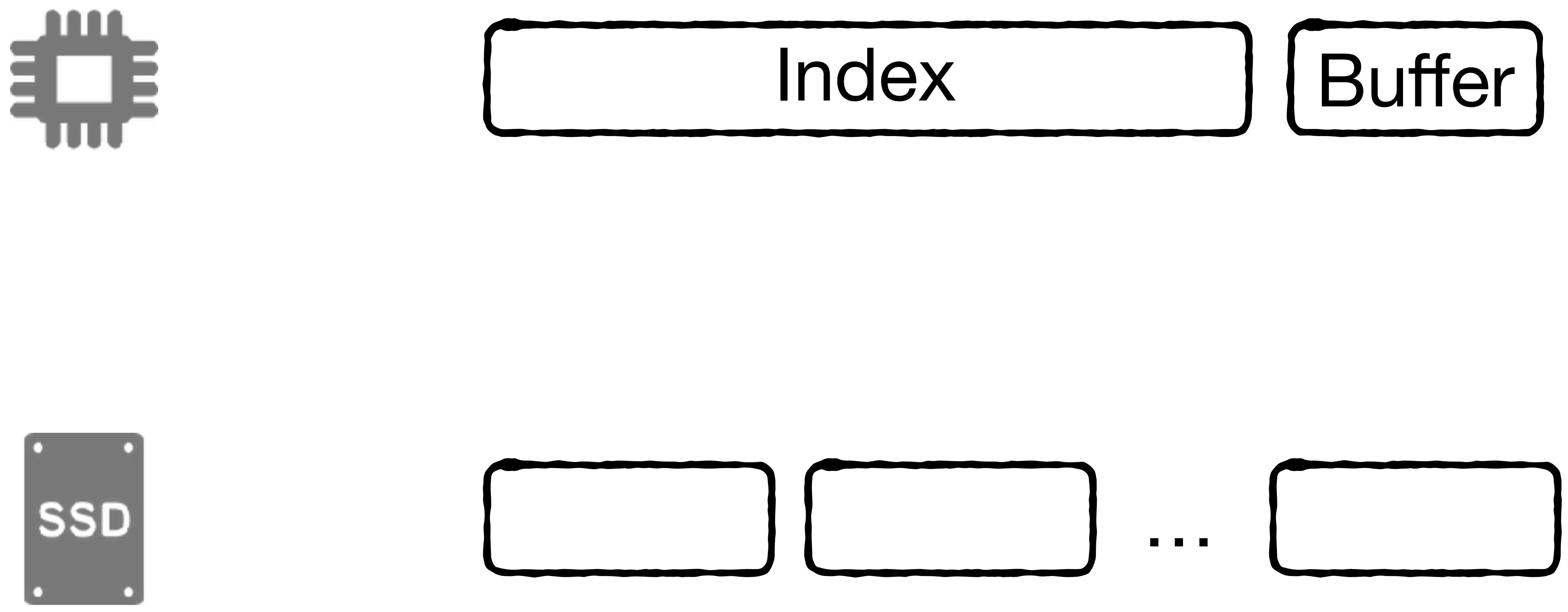
After many deletes, many orphaned entries accumulate

They take up space which we would prefer to use to store valid data

How to fix this? **Garbage Collection (as we saw in SSDs)**

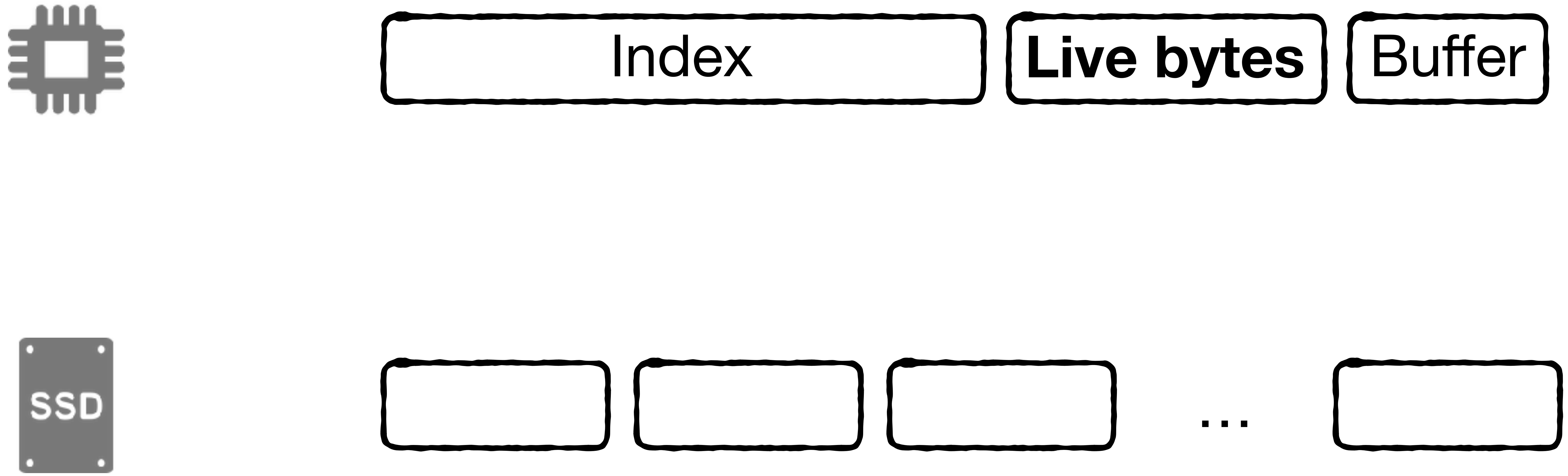


Conceptually divide log into equally-sized areas (can be in order of GBs)

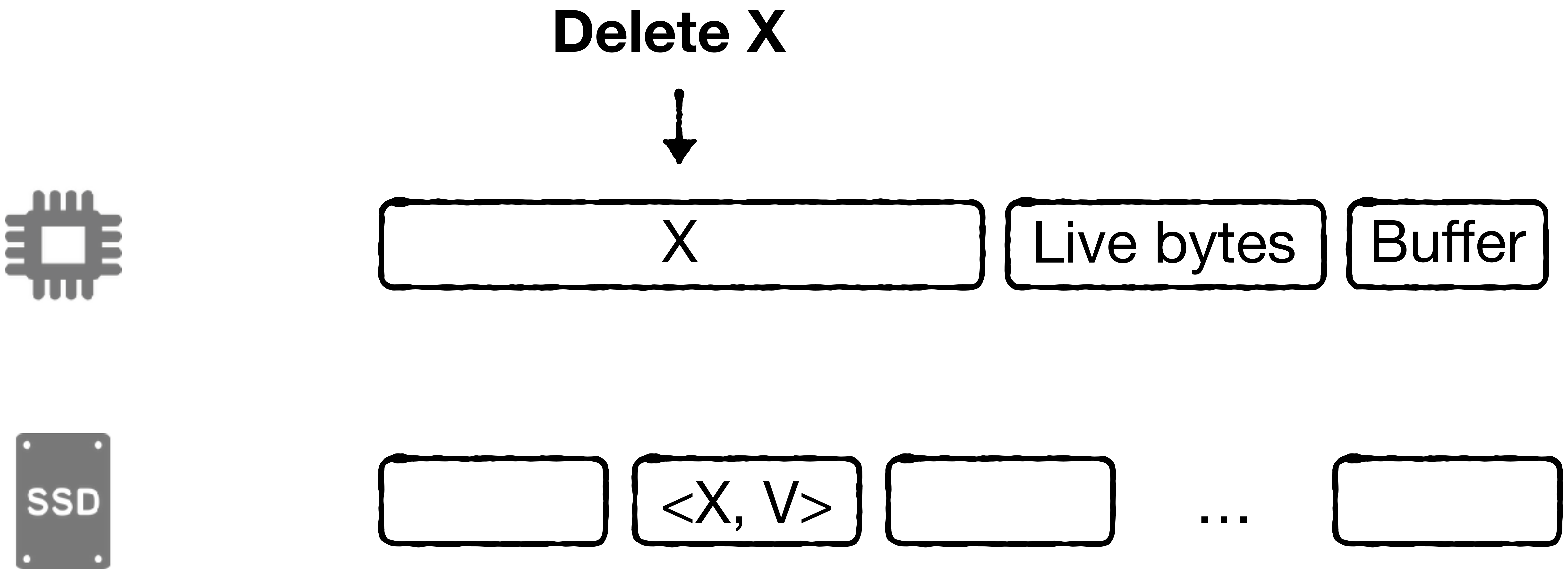


Conceptually divide log into equally-sized areas (can be in order of GBs)

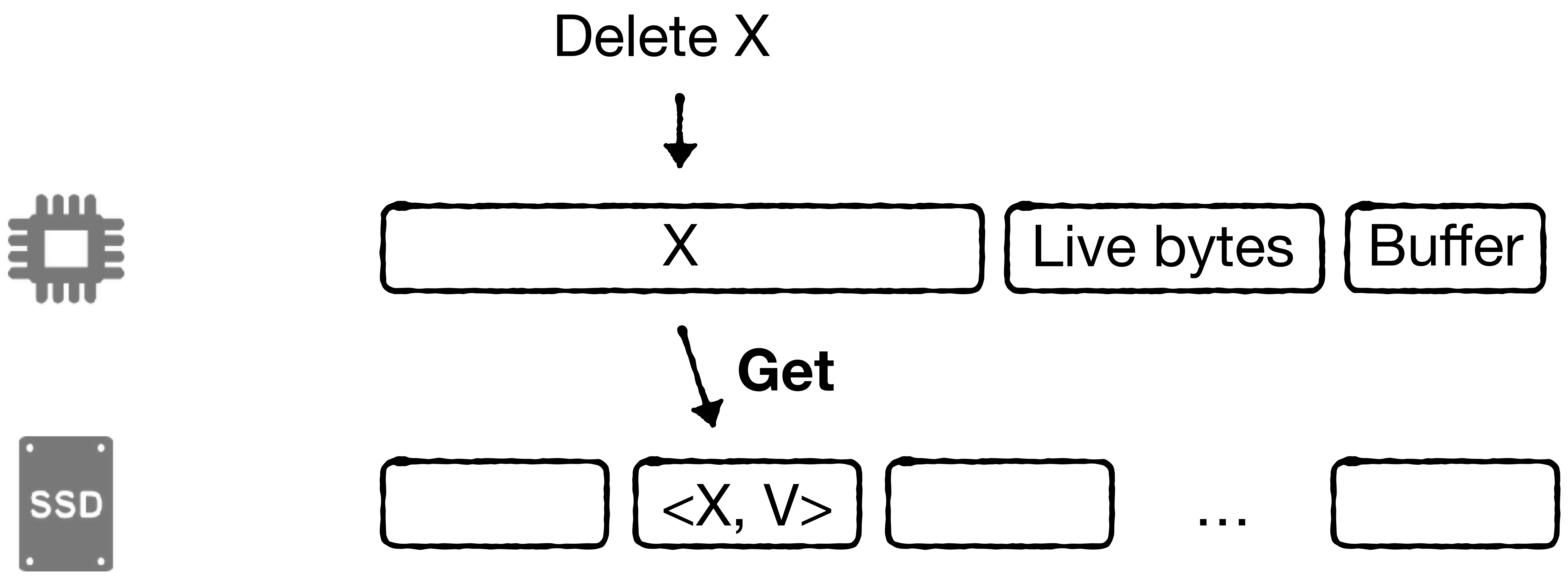
For each area, maintain a counter of the number of bytes representing live data



To delete:

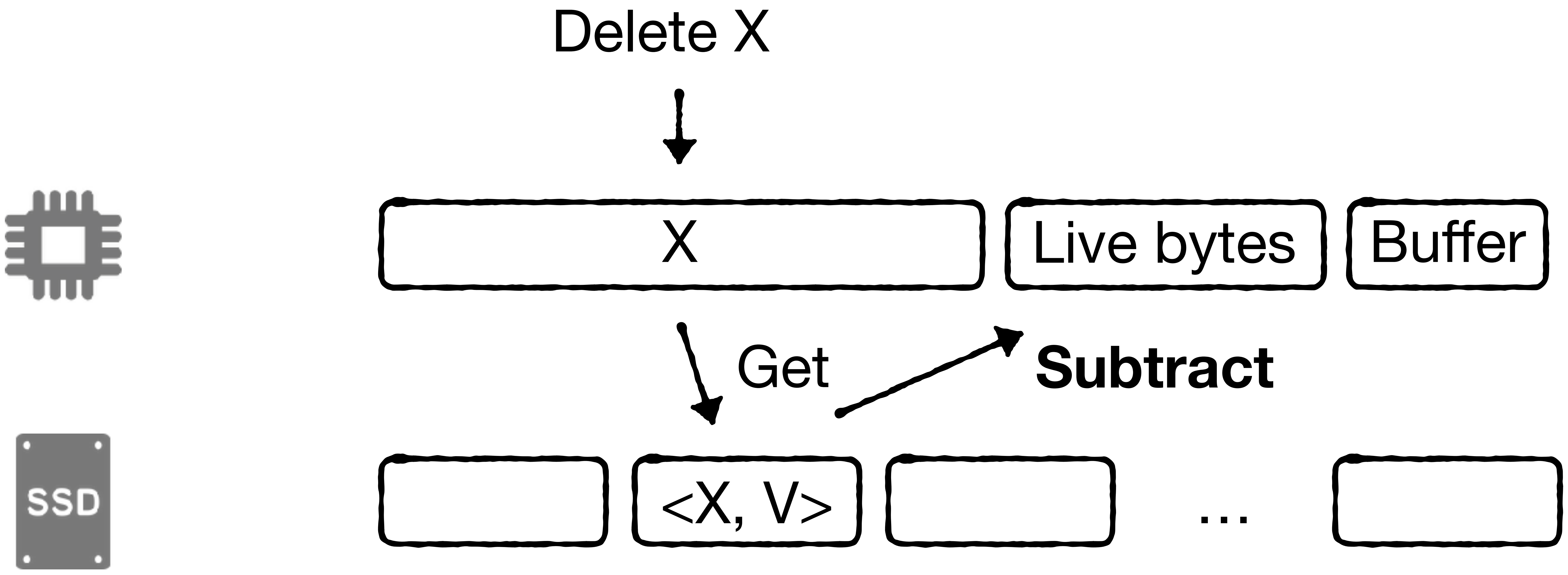


To delete: **(1) get entry from storage to check its size**



To delete: (1) get entry from storage to check its size

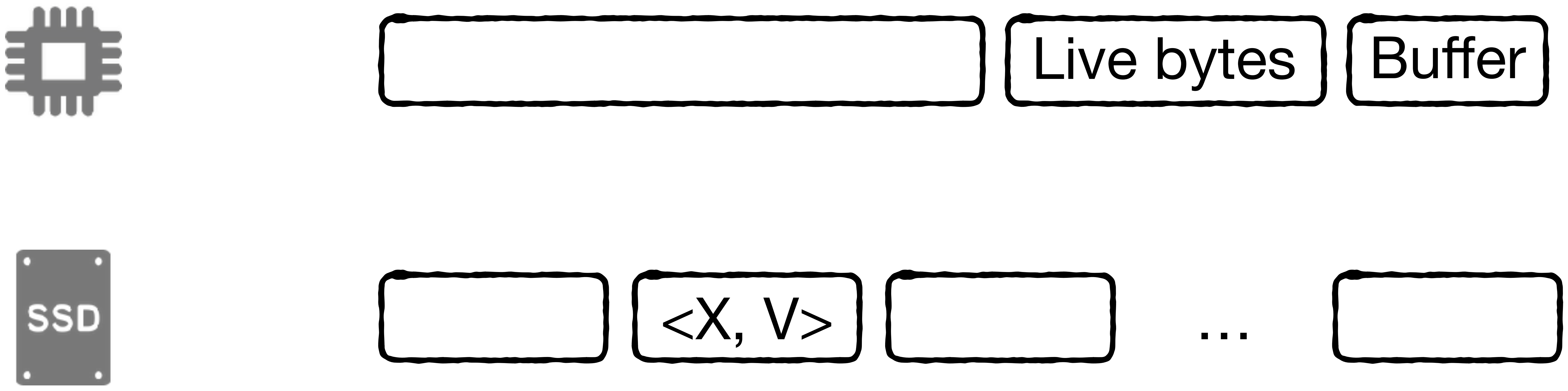
(2) subtract its size from area’s counter



To delete: (1) get entry from storage to check its size

(2) subtract its size from area’s counter

(3) remove entry from index

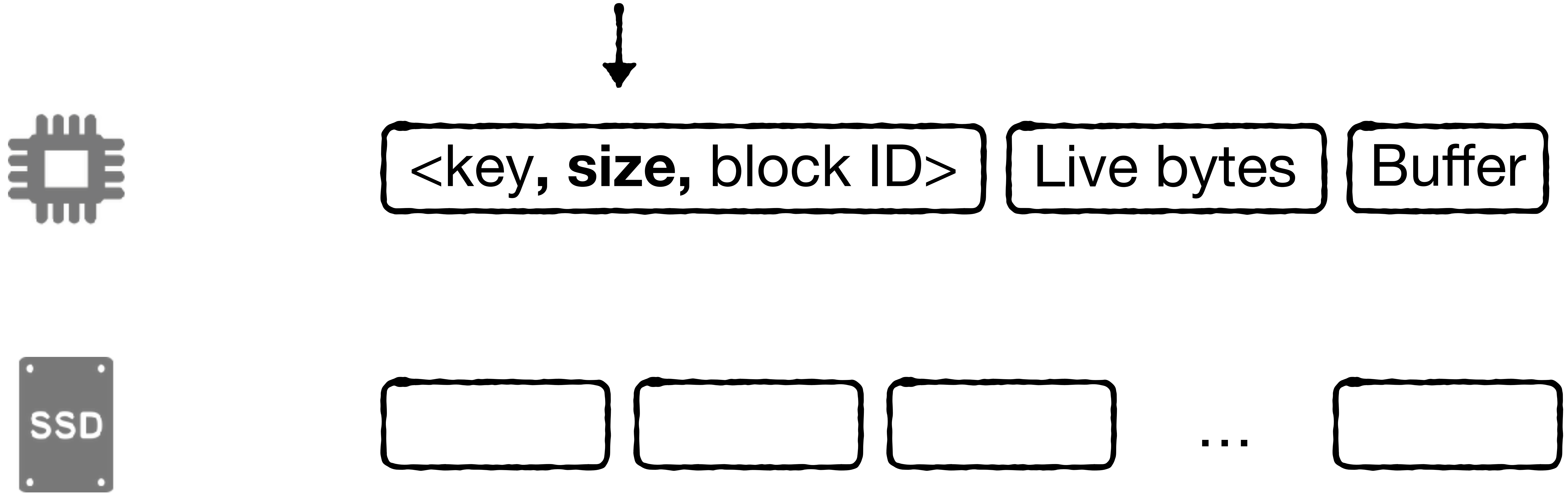


To delete: ~~(1) get entry from storage to check its size~~

(2) subtract its size from area’s counter

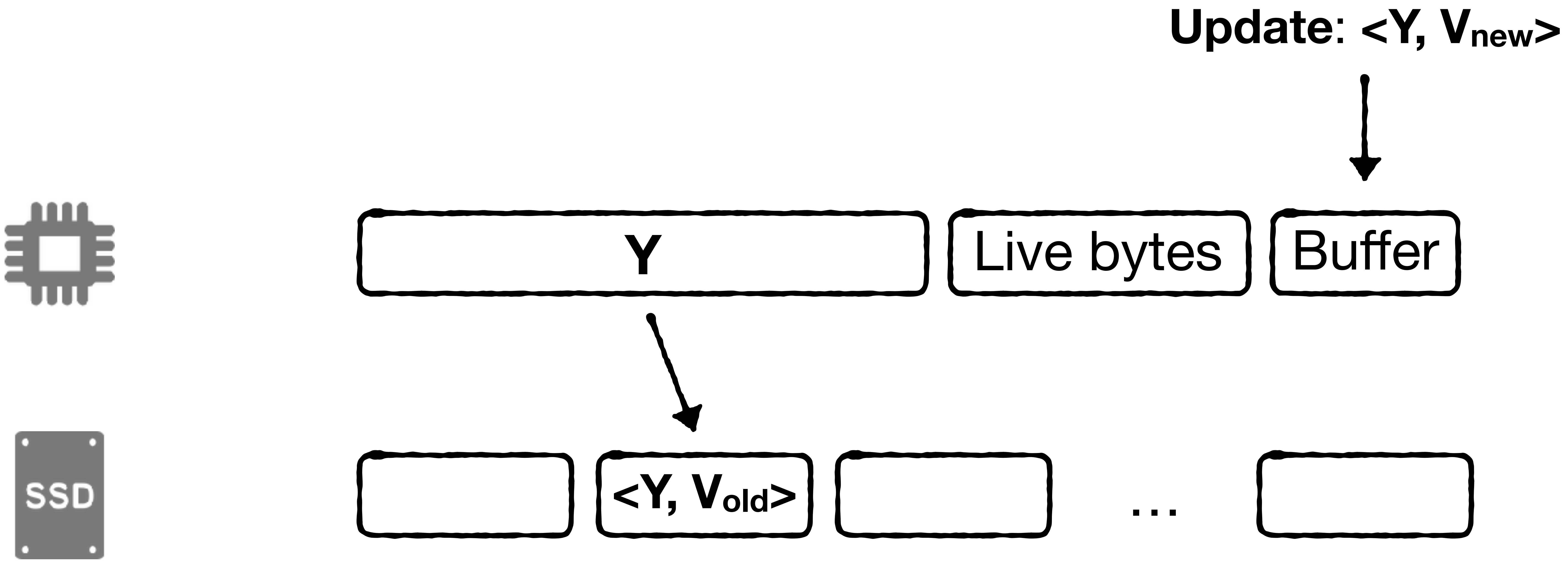
(3) remove entry from index

Can also store each entry’s size in the index, so we not have to get the entry from storage



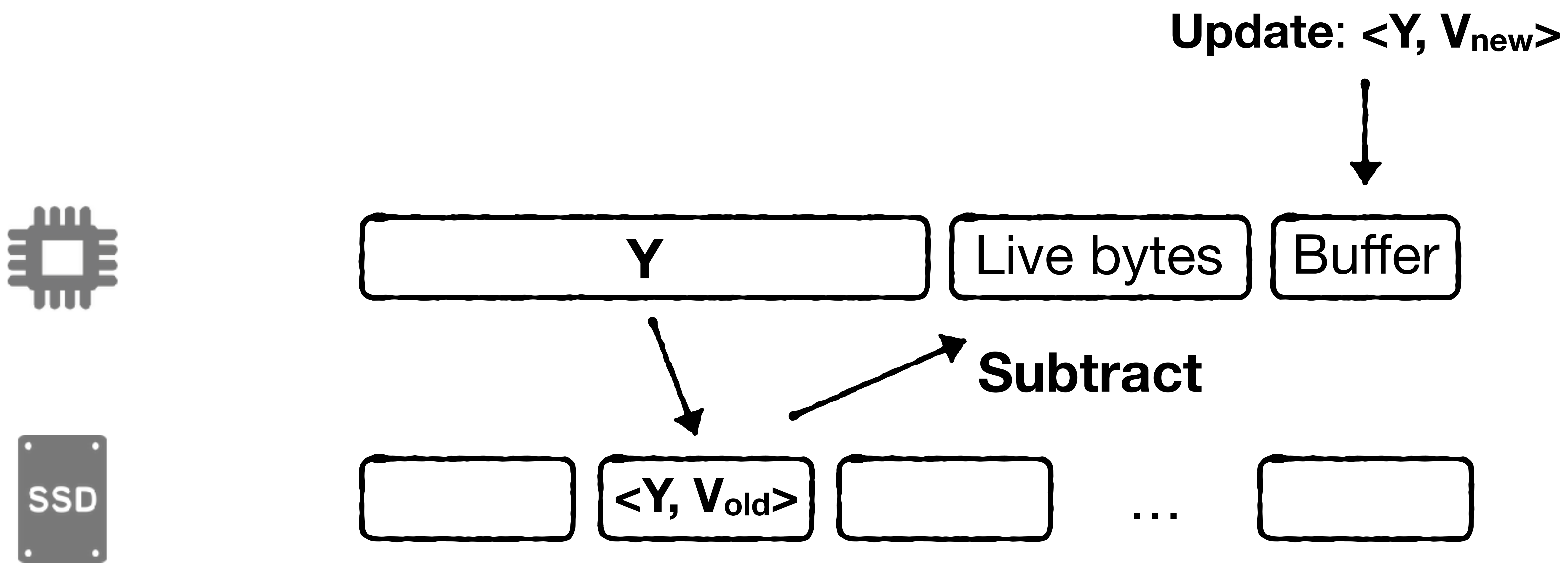
Updates

An update is a delete followed by an insertion of an entry with the same key



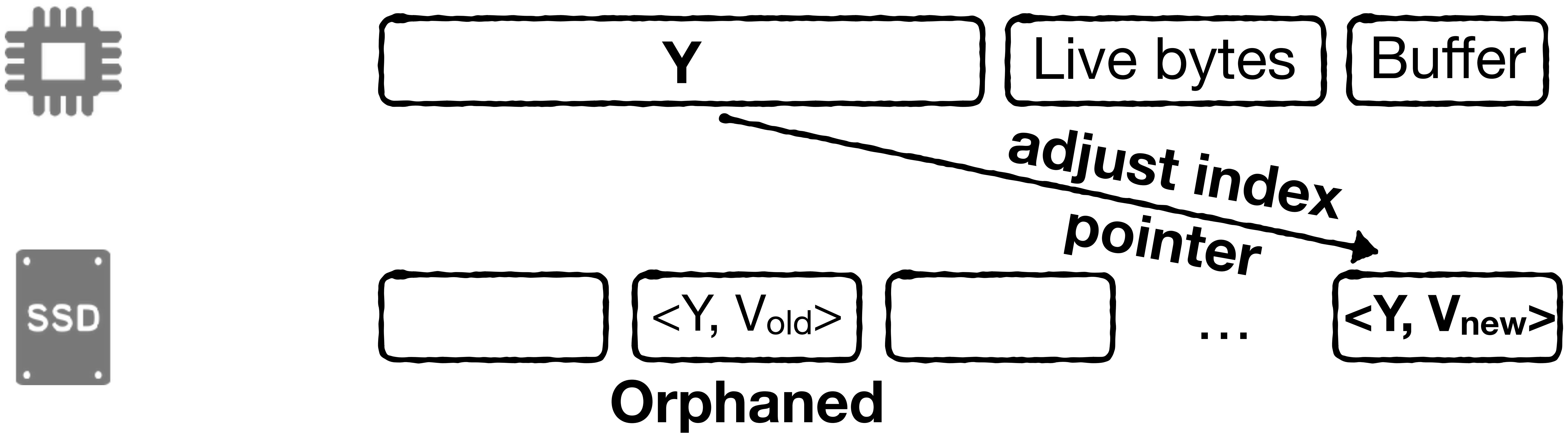
Updates

An update is a delete followed by an insertion of an entry with the same key



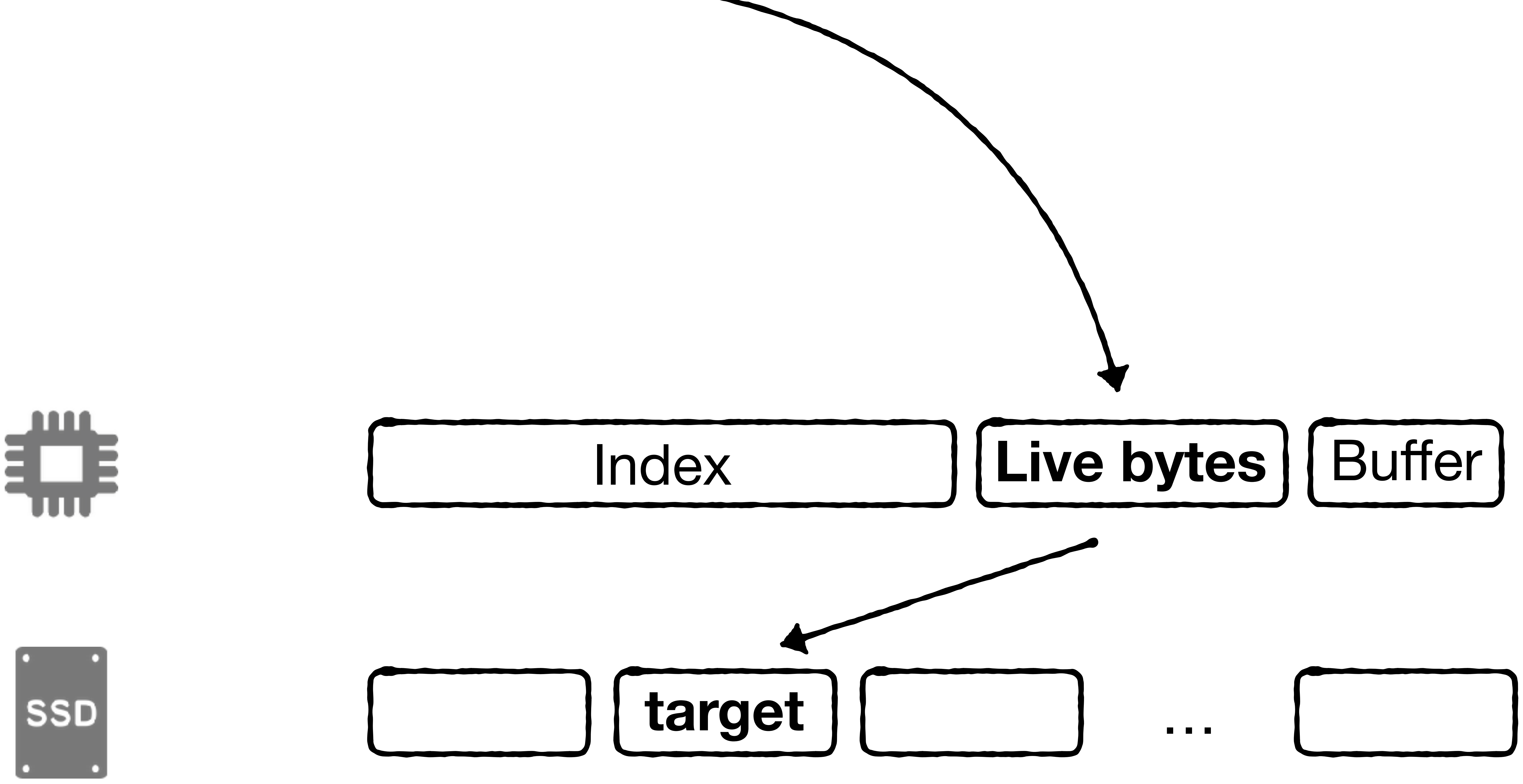
Updates

An update is a delete followed by an insertion of an entry with the same key



Garbage Collection

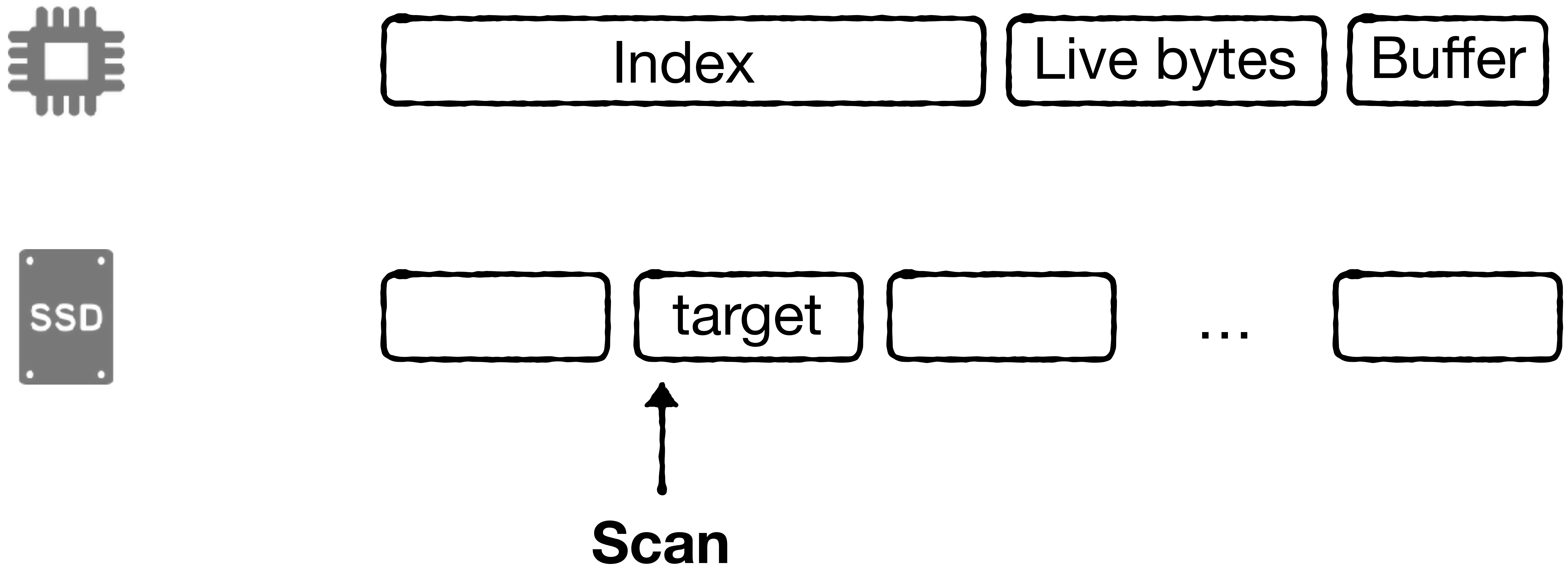
(1) pick area with least live data left



Garbage Collection

(1) pick area with least live data left

(2) Scan area and for each entry

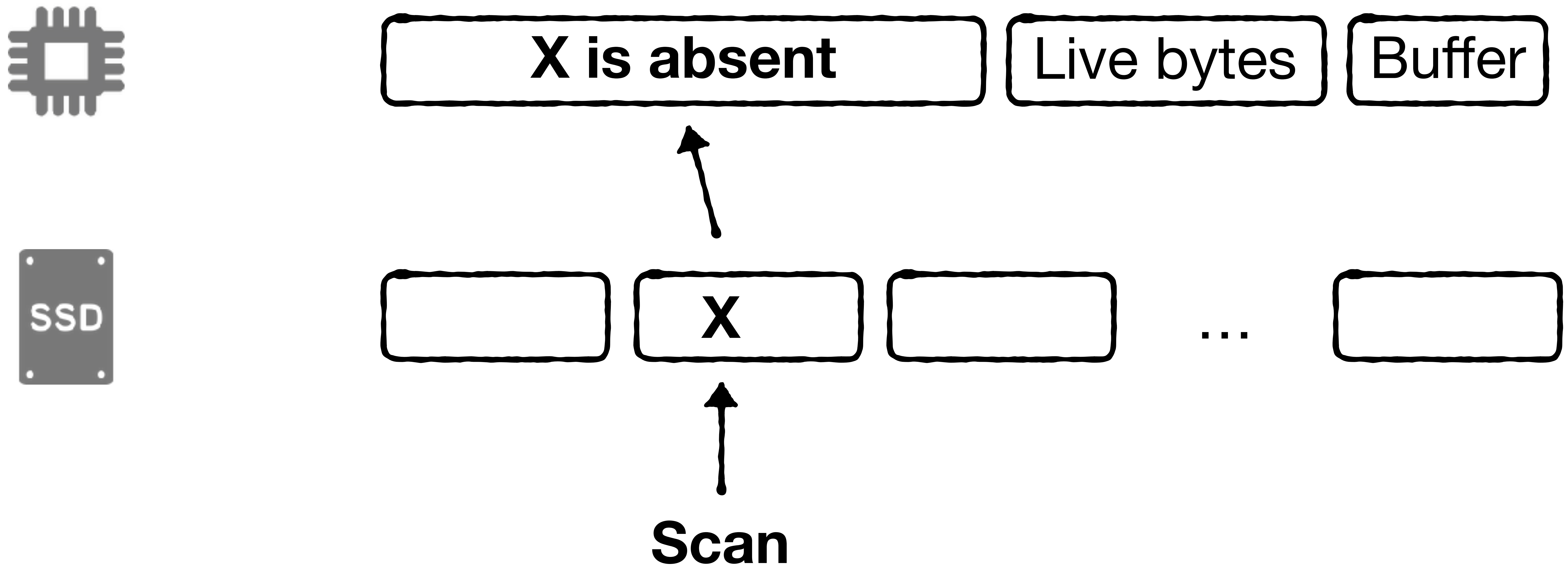


Garbage Collection

(1) pick area with least live data left

(2) Scan area and for each entry

(A) If the key is not indexed, the entry had been deleted, so move on



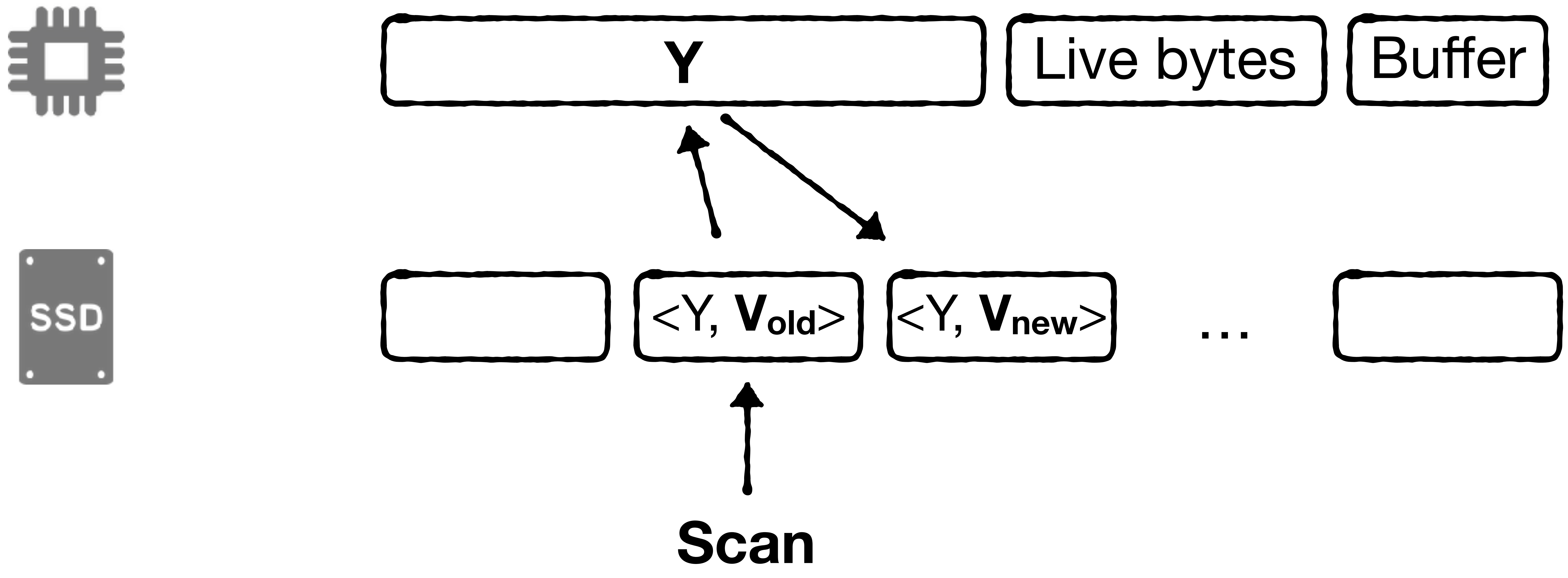
Garbage Collection

(1) pick area with least live data left

(2) Scan area and for each entry

(A) If the key is not indexed, the entry had been deleted, so move on

(B) If the key is indexed but pointing elsewhere, the entry is outdated, so move on



Garbage Collection

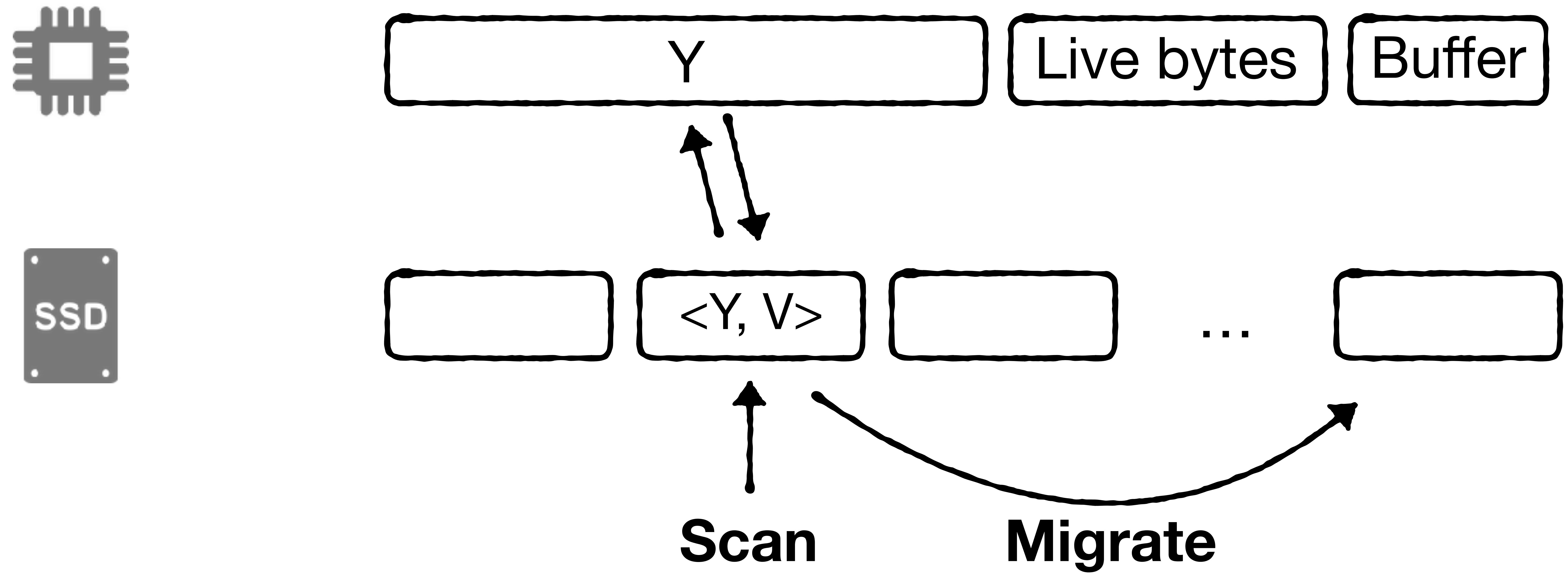
(1) pick area with least live data left

(2) Scan area and for each entry

(A) If the key is not indexed, the entry had been deleted, so move on

(B) If the key is indexed but pointing elsewhere, the entry is outdated, so move on

(C) If the key is indexed and pointing to this entry, it is valid, so migrate it



Garbage Collection

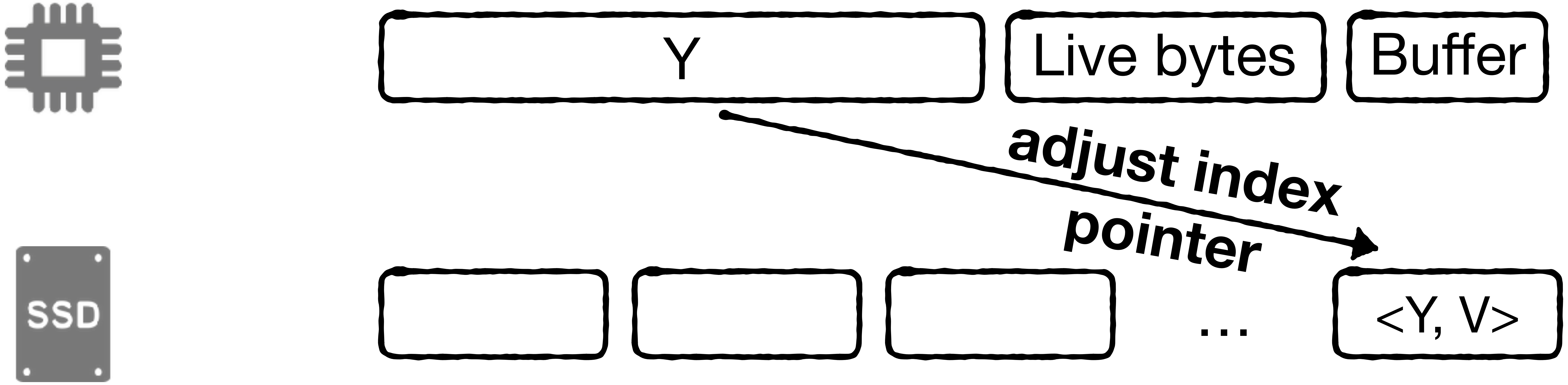
(1) pick area with least live data left

(2) Scan area and for each entry

(A) If the key is not indexed, the entry had been deleted, so move on

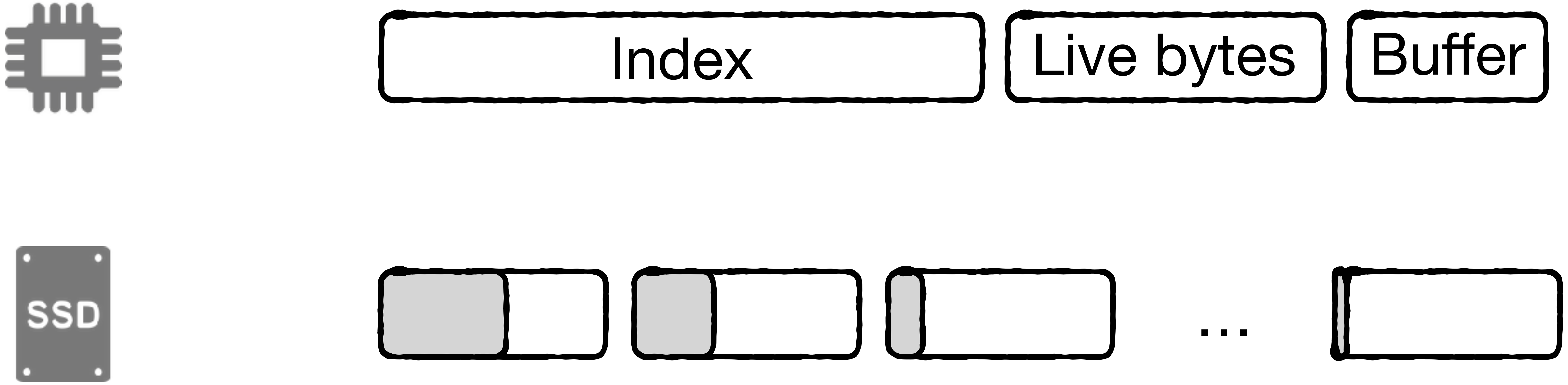
(B) If the key is indexed but pointing elsewhere, the entry is outdated, so move on

(C) If the key is indexed and pointing to this entry, it is valid, so migrate it



When to trigger garbage-collection?

Define a global threshold of (live data L / physical space P)

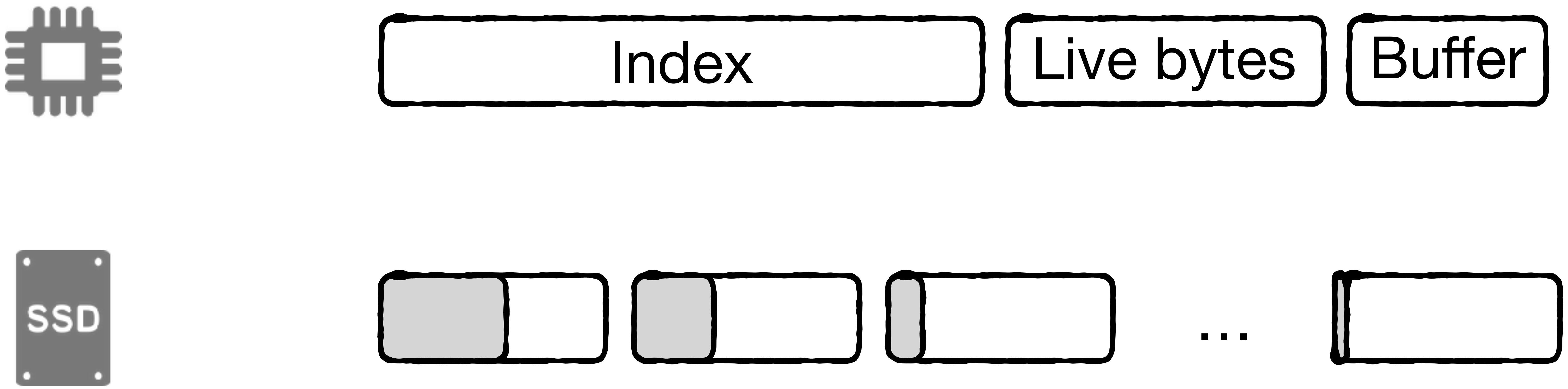


(live data / physical space) = white / (white + gray)

When to trigger garbage-collection?

Define a global threshold of (live data L / physical space P)

When this threshold is reached, trigger garbage-collection to free space

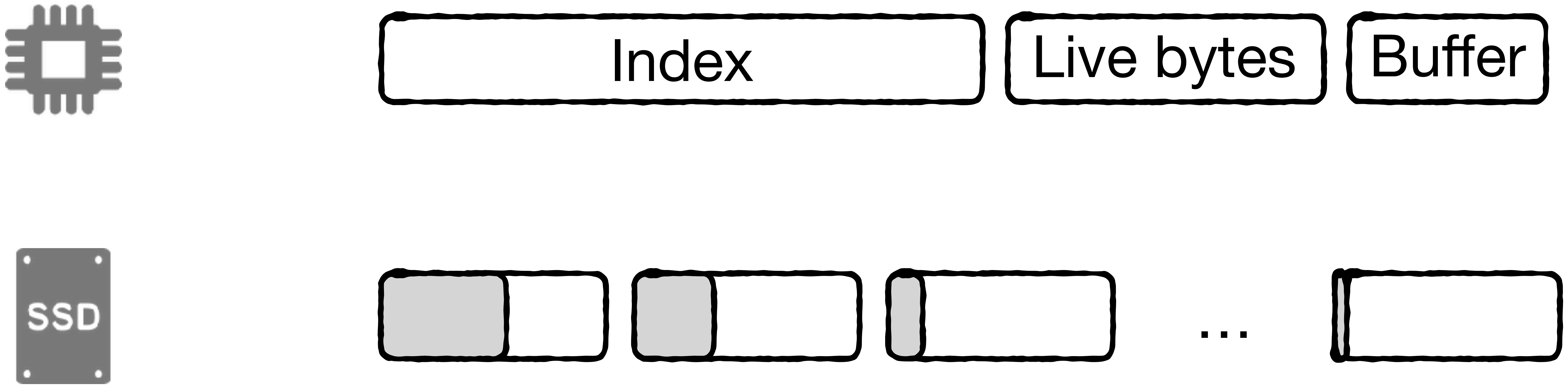


Garbage-Collection Write-Amplification

How to reason about this?

Let x = avg. % of valid pages in areas we pick to garbage-collect

WA = $1 + \dfrac{x}{1 - x}$



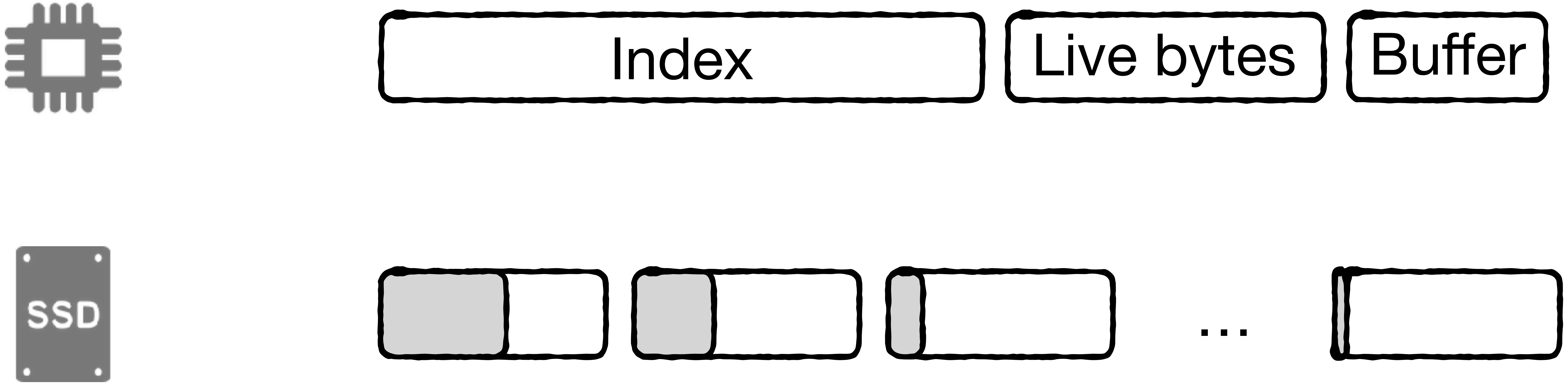
Garbage-Collection Write-Amplification

How to reason about this?

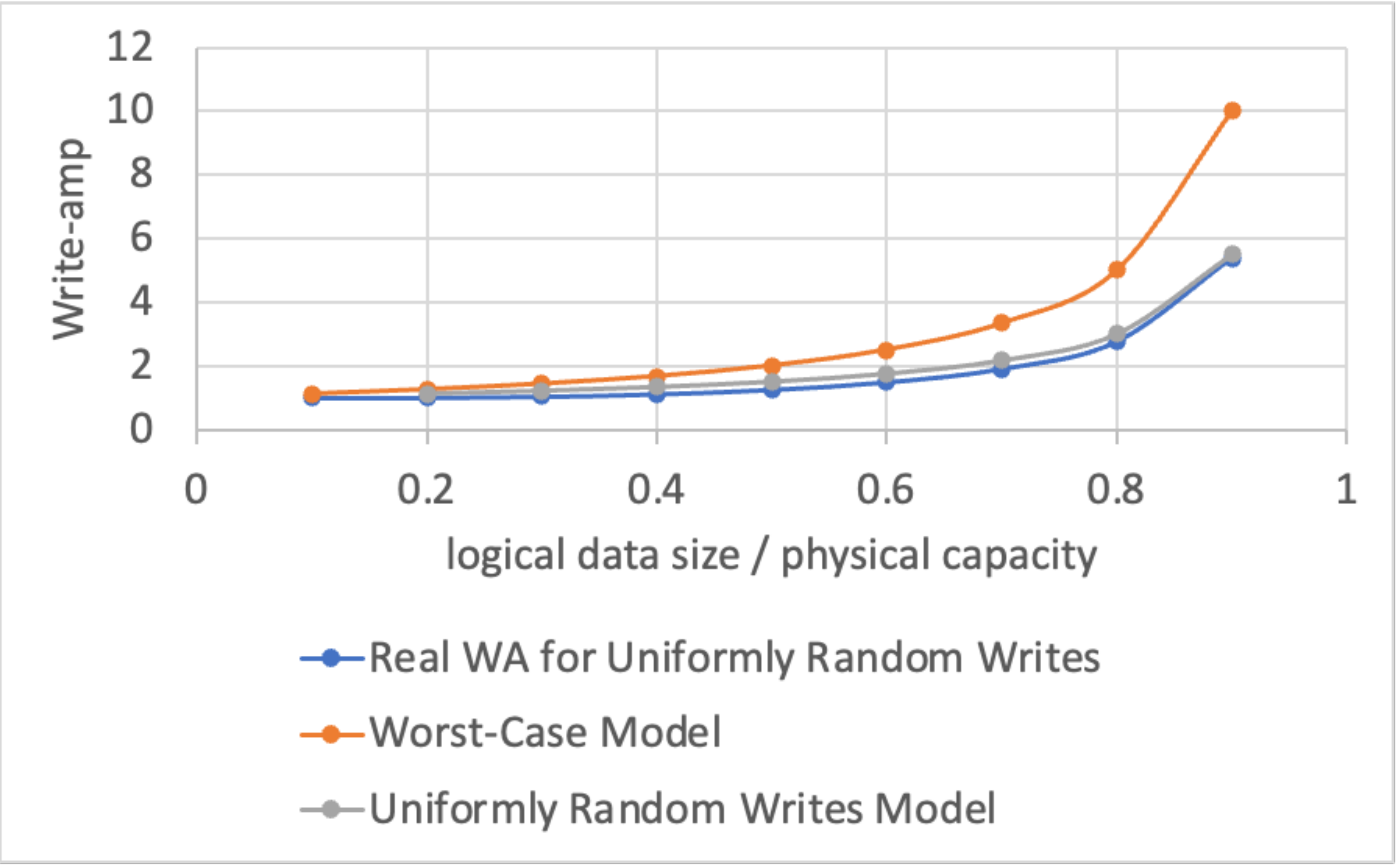
Let x = avg. % of valid pages in areas we pick to garbage-collect

WA = $1 + \dfrac{x}{1 - x}$

Same analysis as for garbage-collection in SSDs, so refer to that :)



Garbage-Collection Write-Amplification



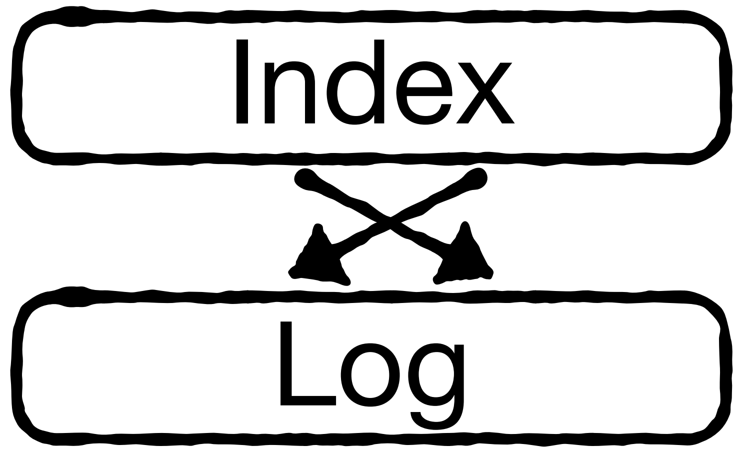
$$1 + \frac{L/P}{1 - L/P}$$ **Worst case**

$$1 + \frac{1}{2} \cdot \frac{L/P}{1 - L/P}$$ **Uniformly random**

L = logical data size

P = physical data size

Mechanics



Hot/Cold Separation

To reduce write-amp

Checkpointing/ recovery

If power fails

Cuckoo filtering

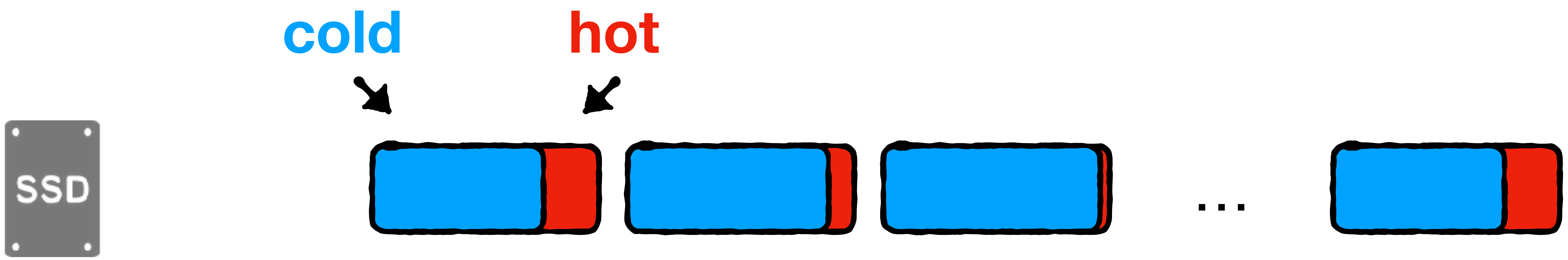
To reduce memory

Hot/Cold Separation

Normal workloads are neither worst-case nor randomly distributed

Normal workloads are neither worst-case nor randomly distributed

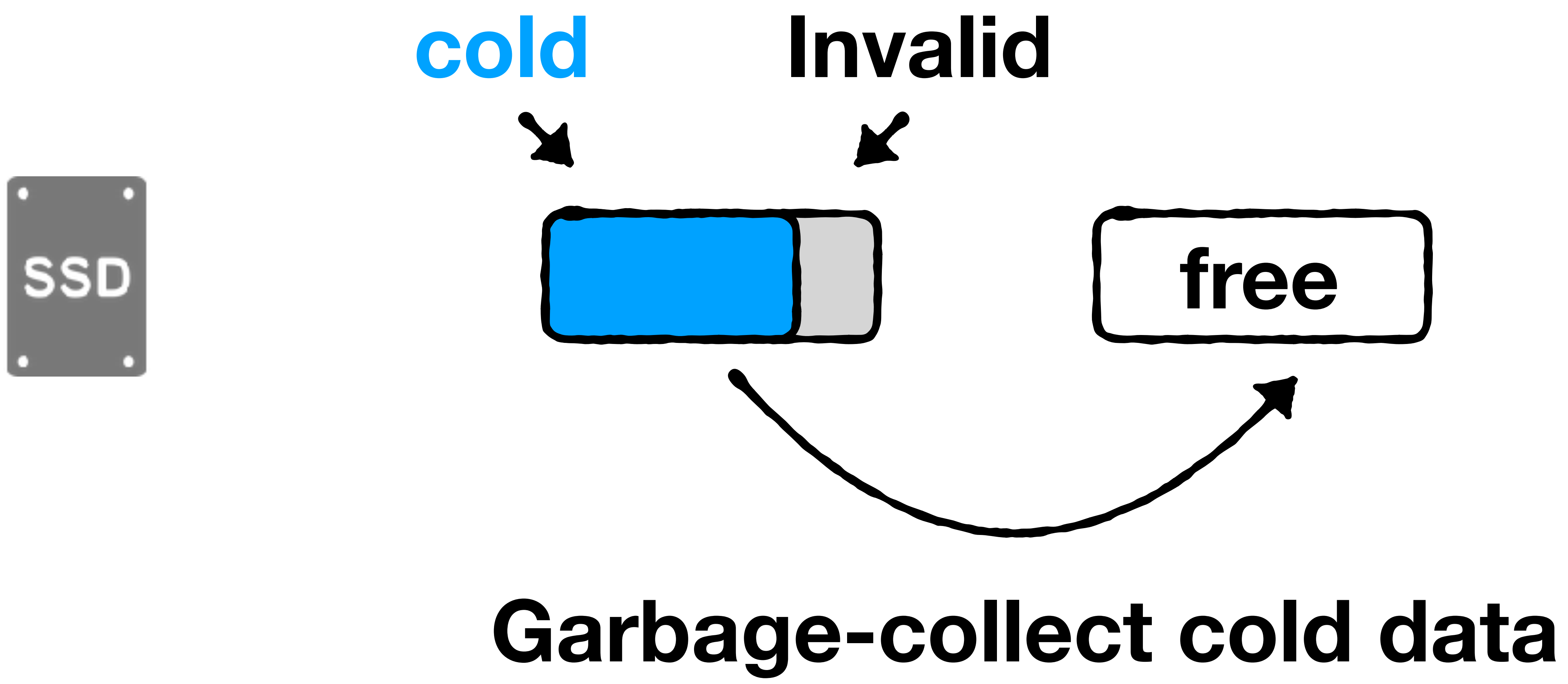
Typically, few entries are frequently updated while most are seldom updated



Normal workloads are neither worst-case nor randomly distributed

Typically, few entries are frequently updated while most are seldom updated

Hot entries are invalidated quickly, so by the time we garbage-collect, there is usually only cold data left

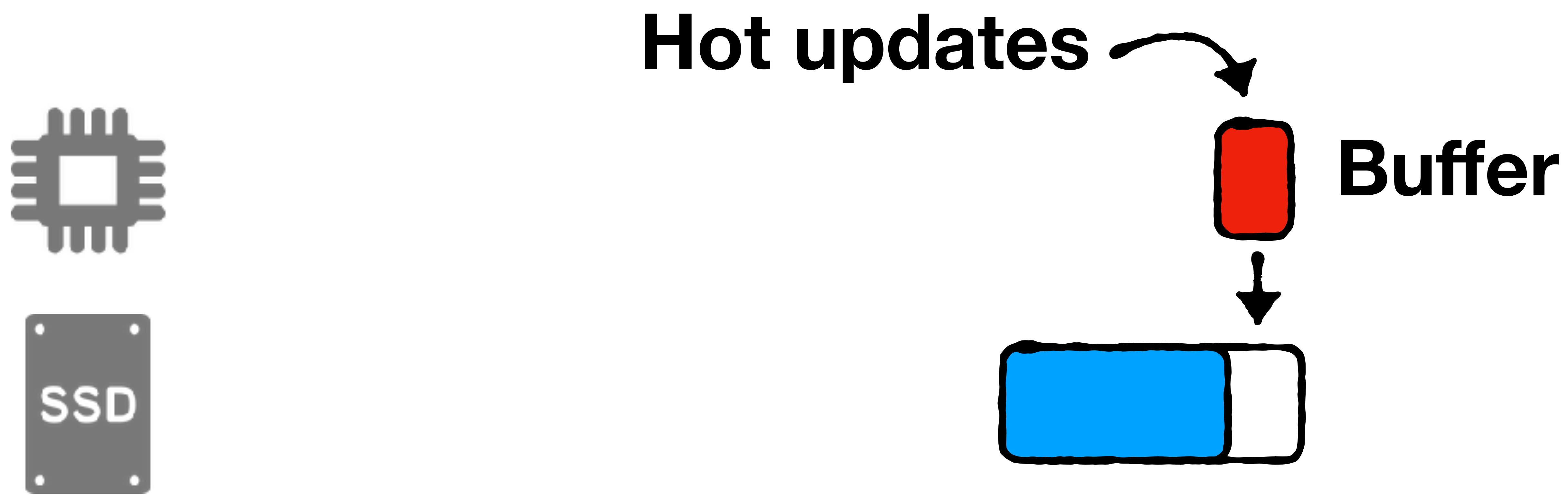


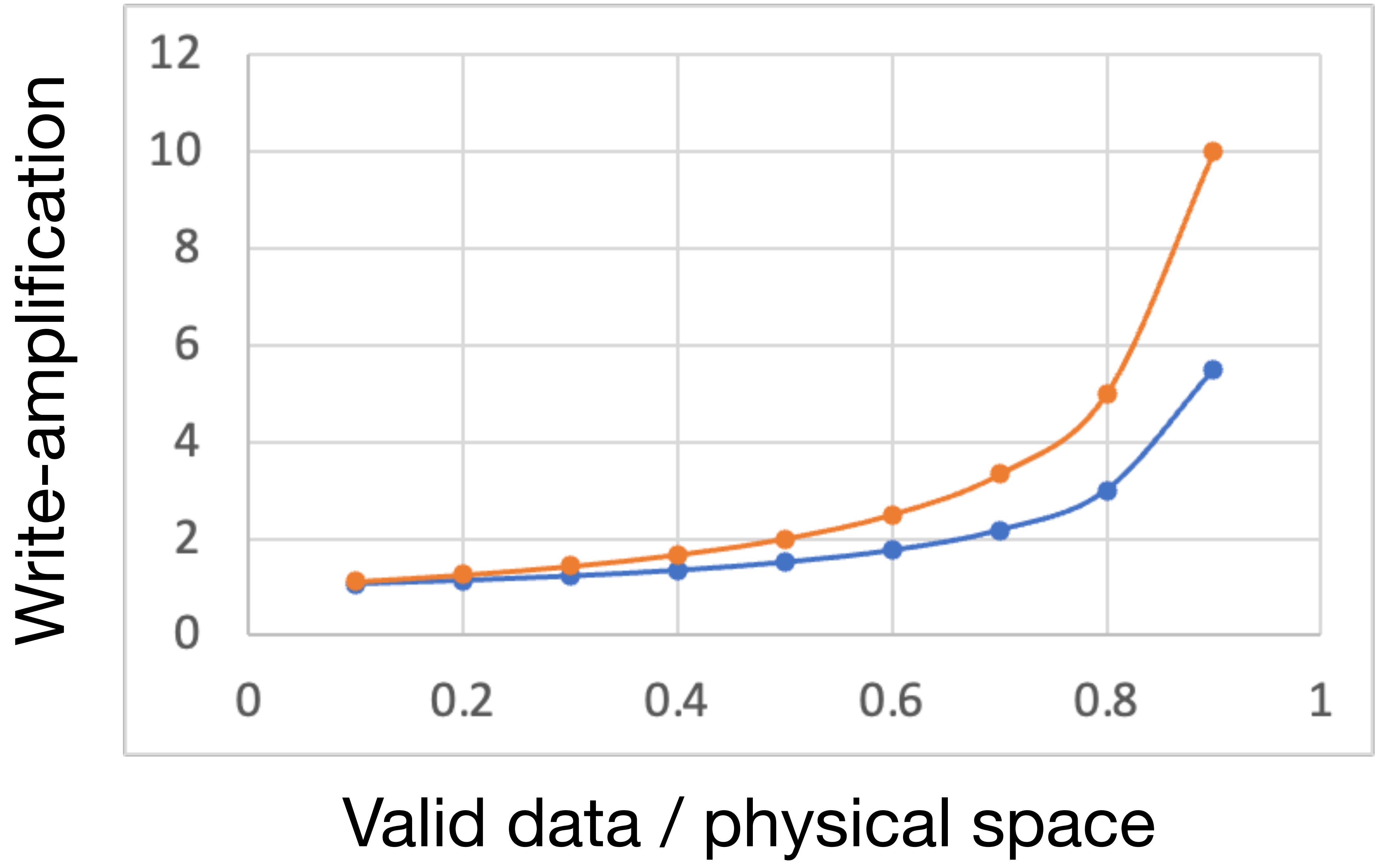
Normal workloads are neither worst-case nor randomly distributed

Typically, few entries are frequently updated while most are seldom updated

Hot entries are invalidated quickly, so by the time we garbage-collect, there is usually only cold data left

This migrated cold data gets mixed with more hot data, and the cycle repeats.





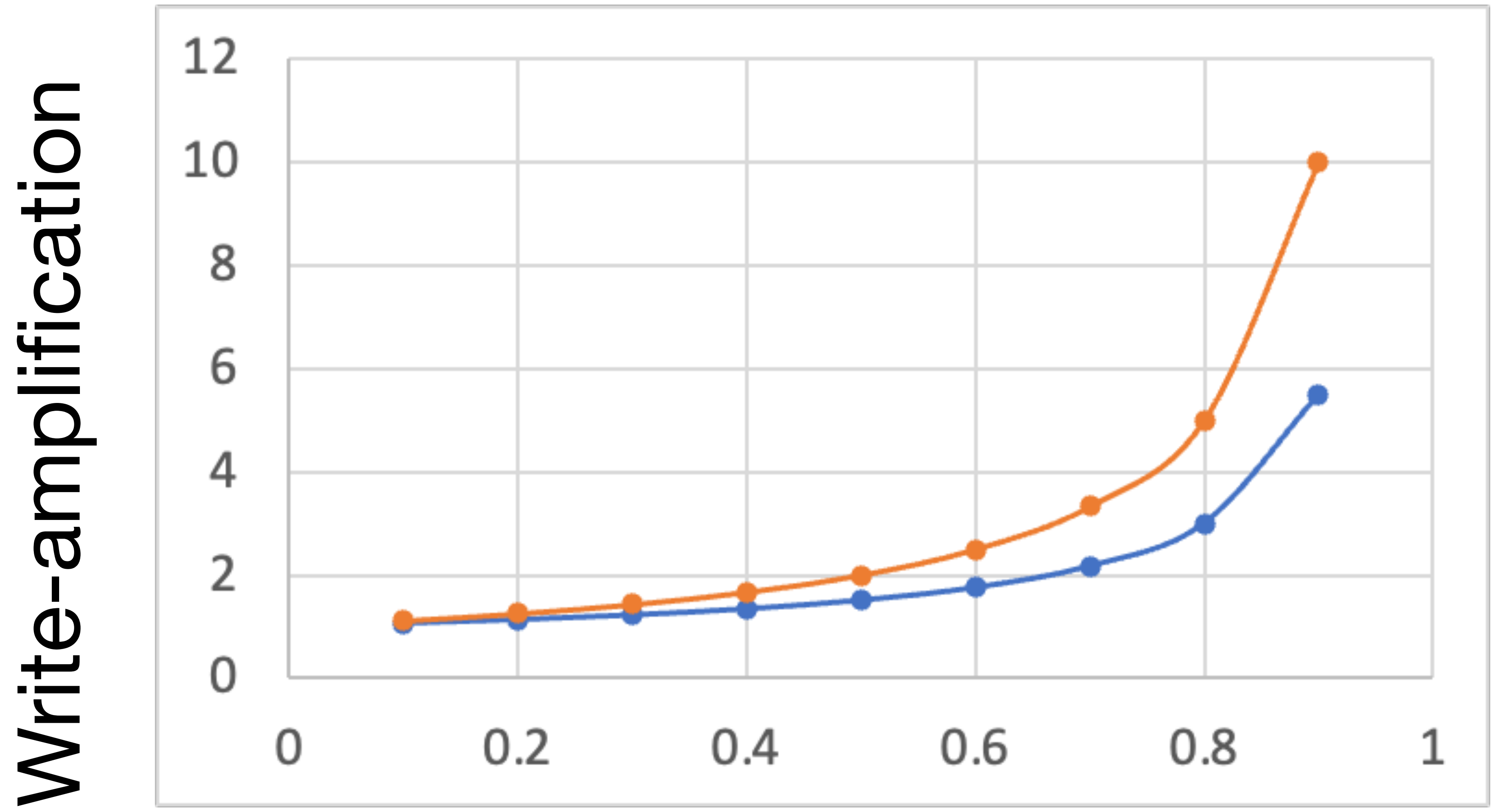
$$1 + \frac{L/P}{1 - L/P}$$

Mixing hot/cold data brings us towards worst-case

$$1 + \frac{L/P}{2 \cdot (1 - L/P)}$$

Uniformly random workloads

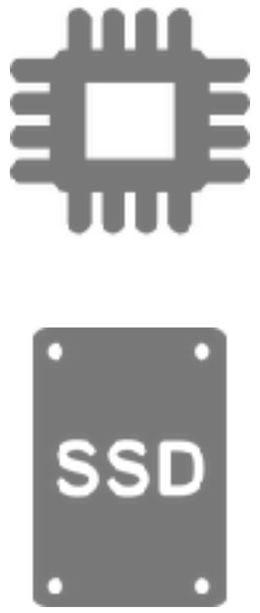
Hot vs. Cold Data

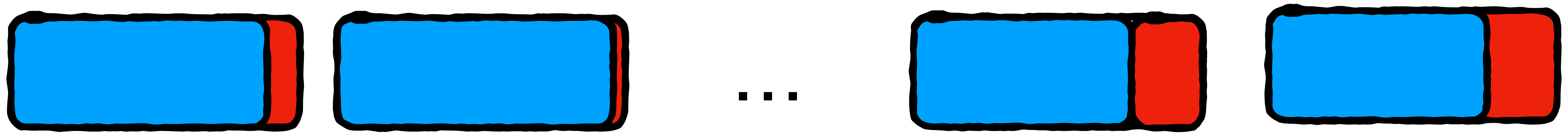


$$1 + \frac{L/P}{1 - L/P}$$

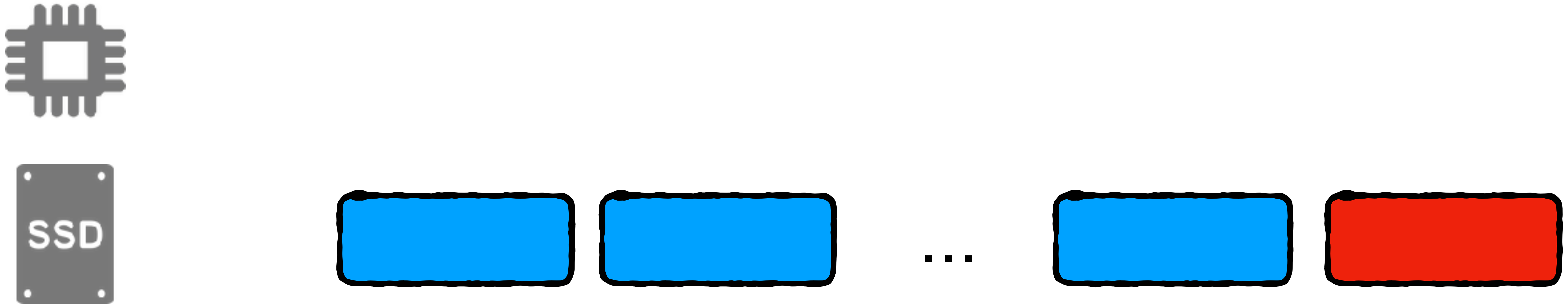
Mixing hot/cold data brings us towards worst-case

How can we avoid garbage-collecting cold data all the time?



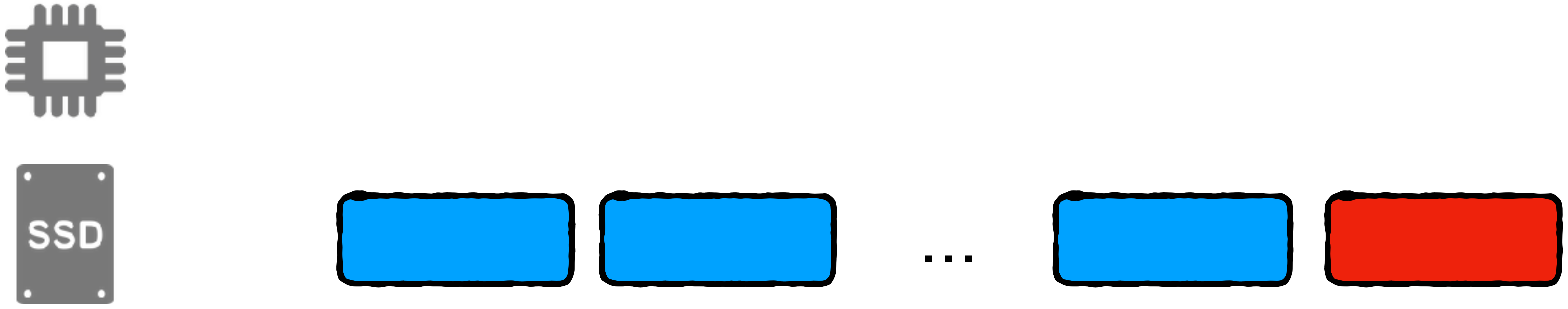


Hot vs. Cold Data Separation



Hot vs. Cold Data Separation

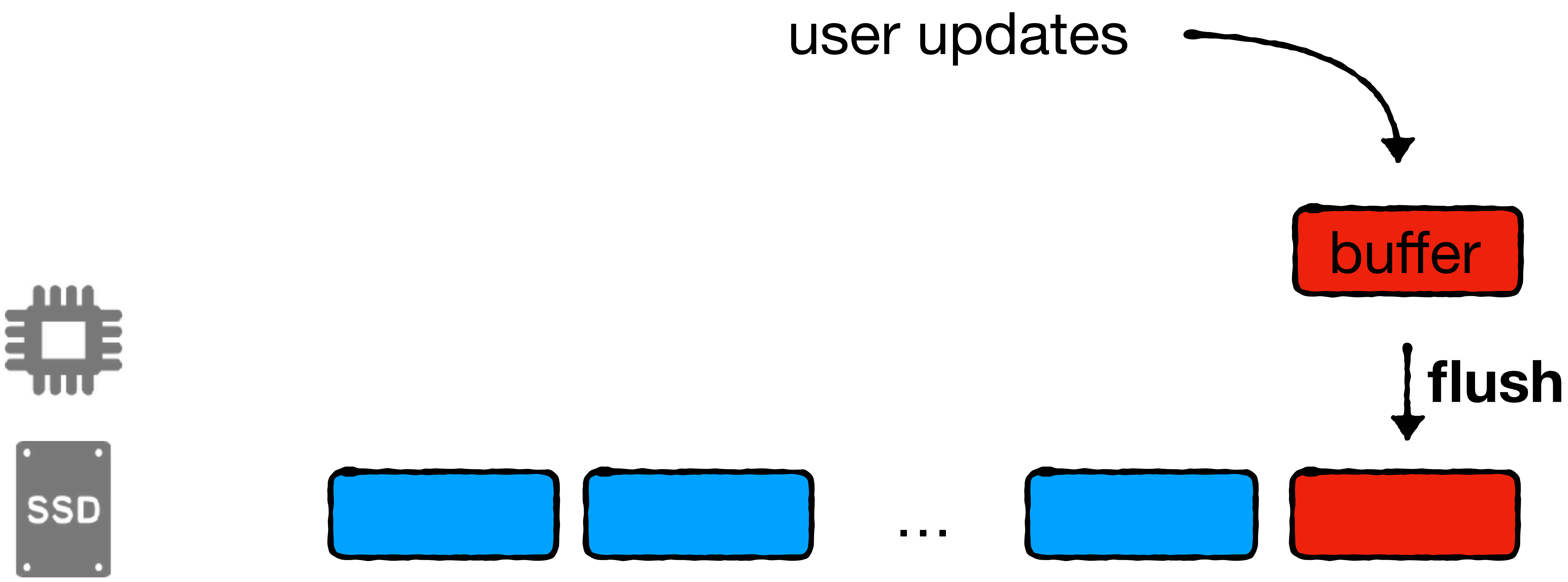
We can separate hot vs. cold data into different areas



Hot vs. Cold Data Separation

We can separate hot vs. cold data into different areas

Insight 1: user updates are generally hot
(i.e., data recently written is likely to be written again)

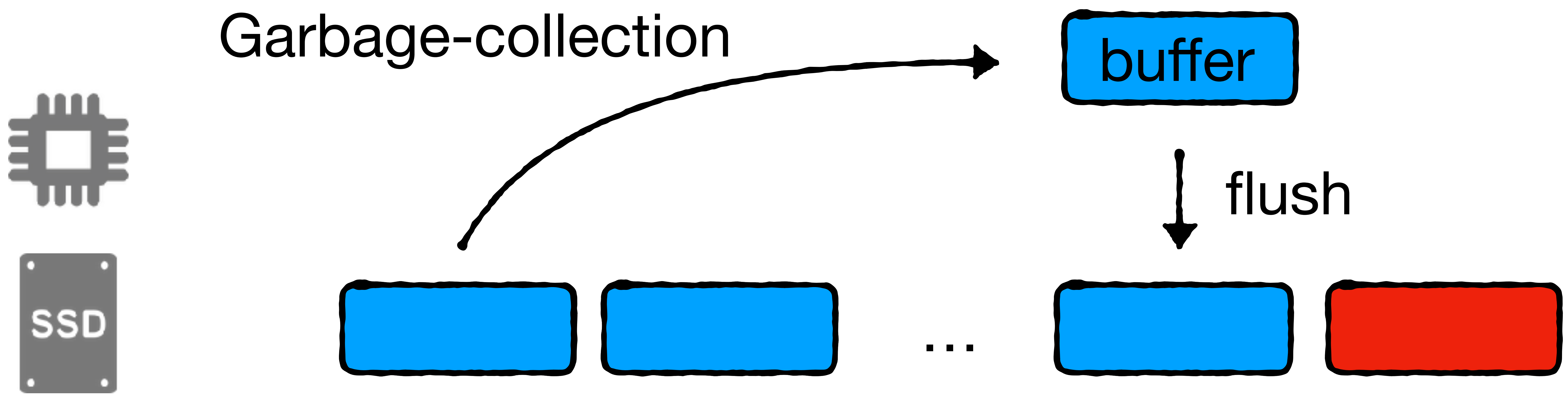


Hot vs. Cold Data Separation

We can separate hot vs. cold data into different areas

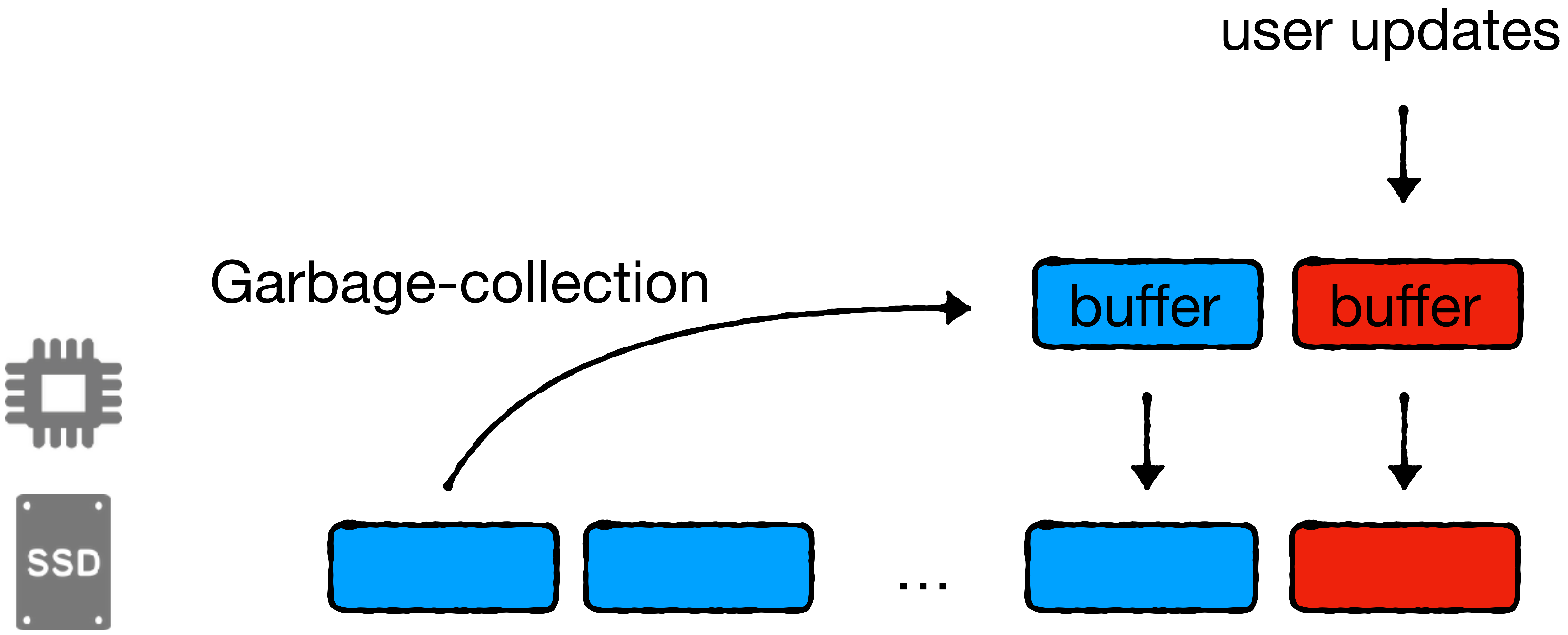
Insight 1: user updates are generally hot
(i.e., data recently written is likely to be written again)

Insight 2: garbage-collected data is generally cold
(i.e., it had already existed for a long while without getting updated)



Hot vs. Cold Data Separation

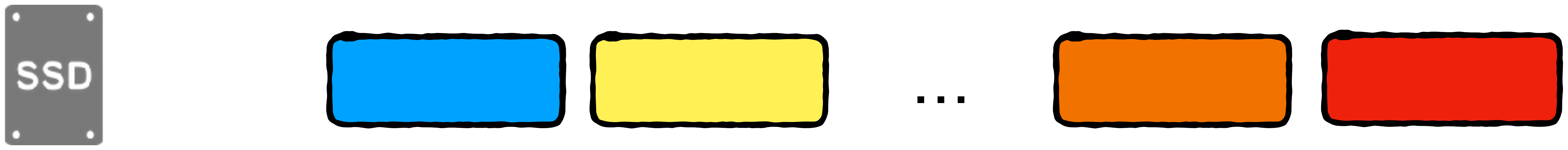
Simplest solution: separate user updates and cold data using different buffers into different areas



Hot vs. Cold Data Separation

Simplest solution: separate user updates and cold data using different buffers into different areas

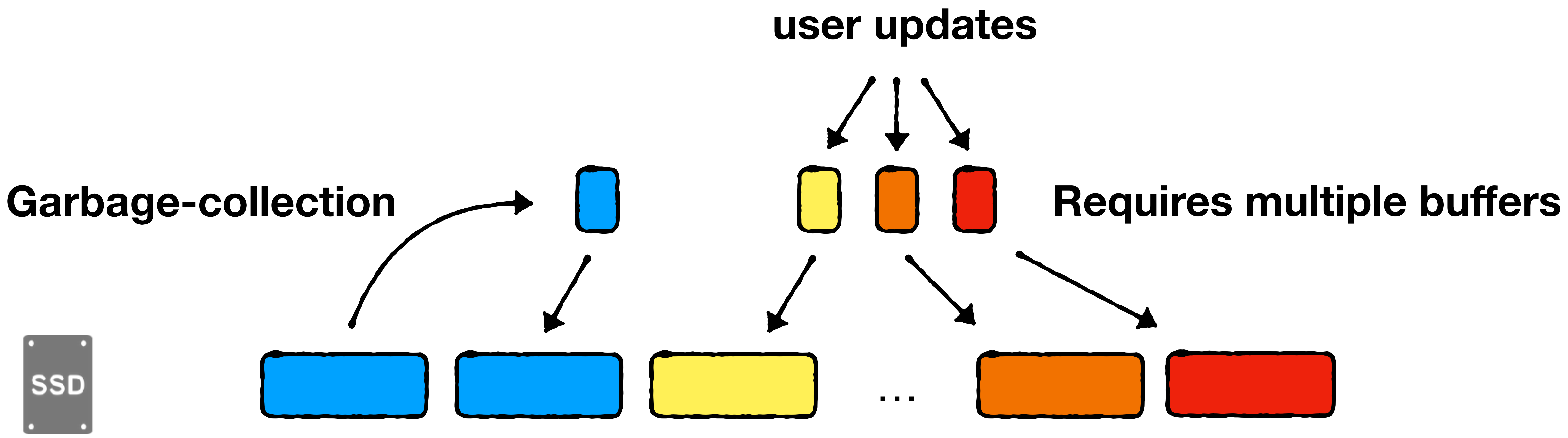
More advanced: separate data with different temperatures into different areas



Hot vs. Cold Data Separation

Simplest solution: separate user updates and cold data using different buffers into different areas

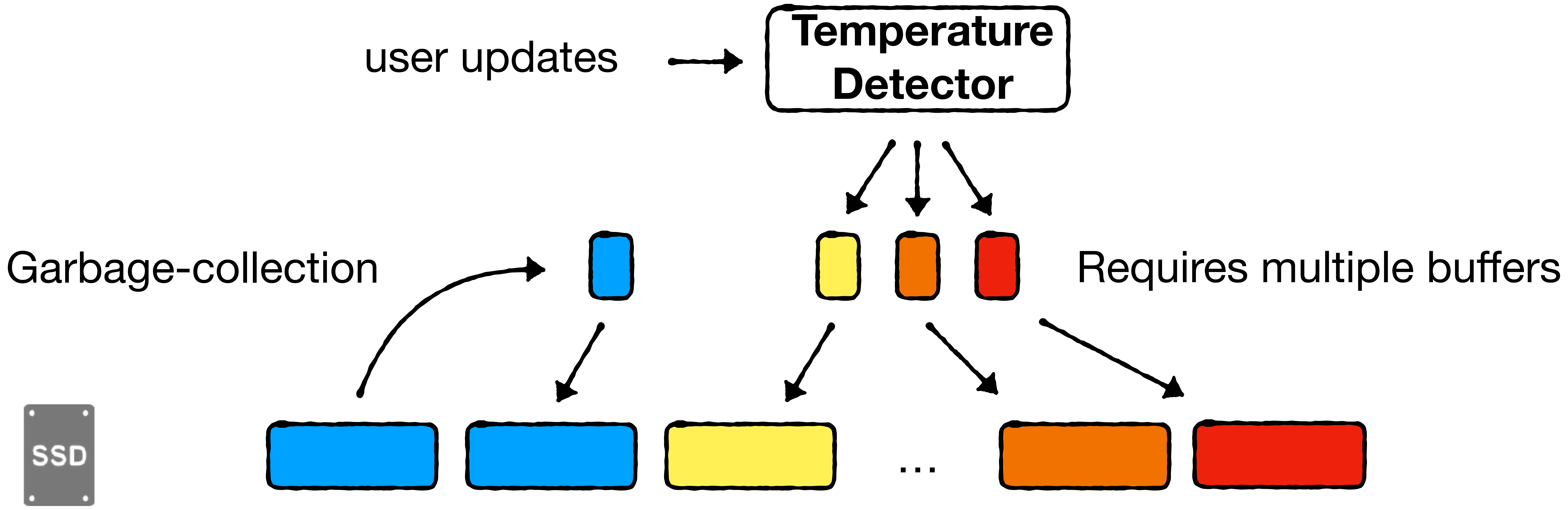
More advanced: separate data with different temperatures into different areas



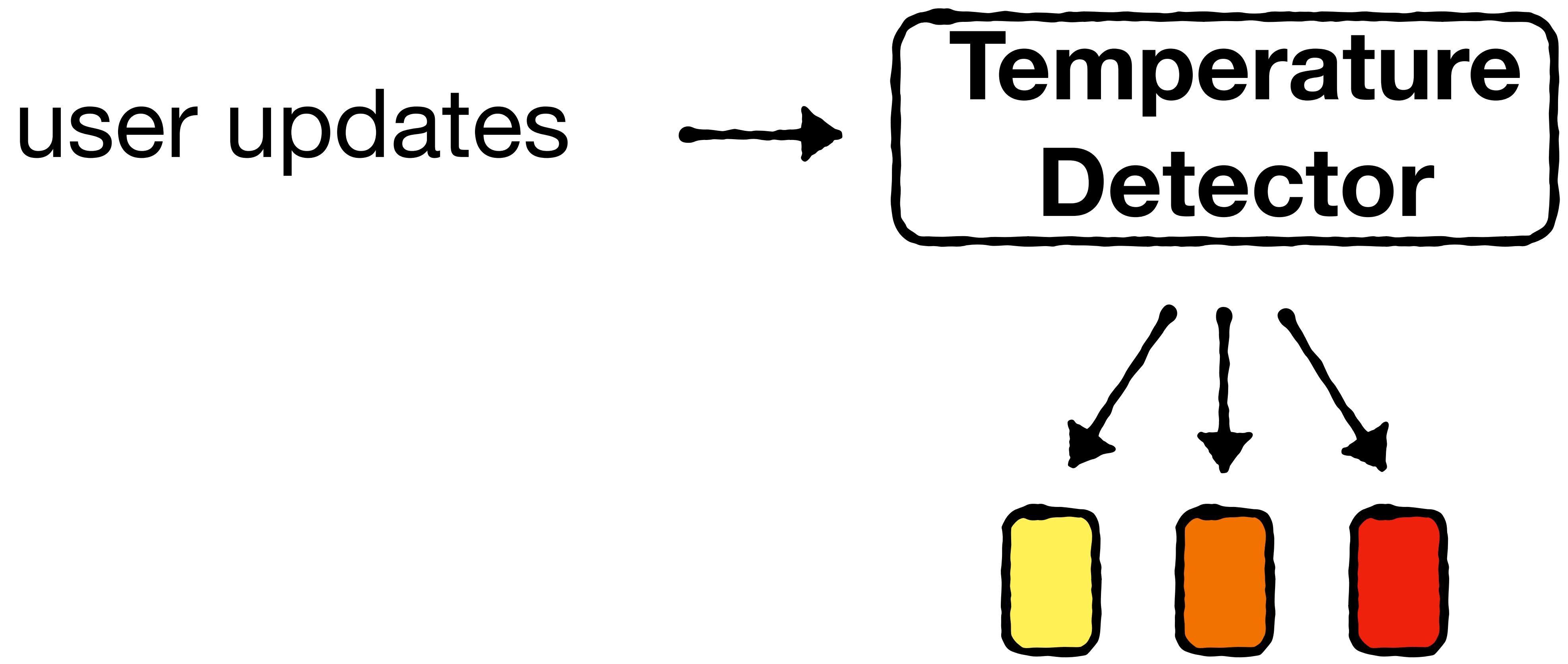
Hot vs. Cold Data Separation

Simplest solution: separate user updates and cold data using different buffers into different areas

More advanced: separate data with different temperatures into different areas

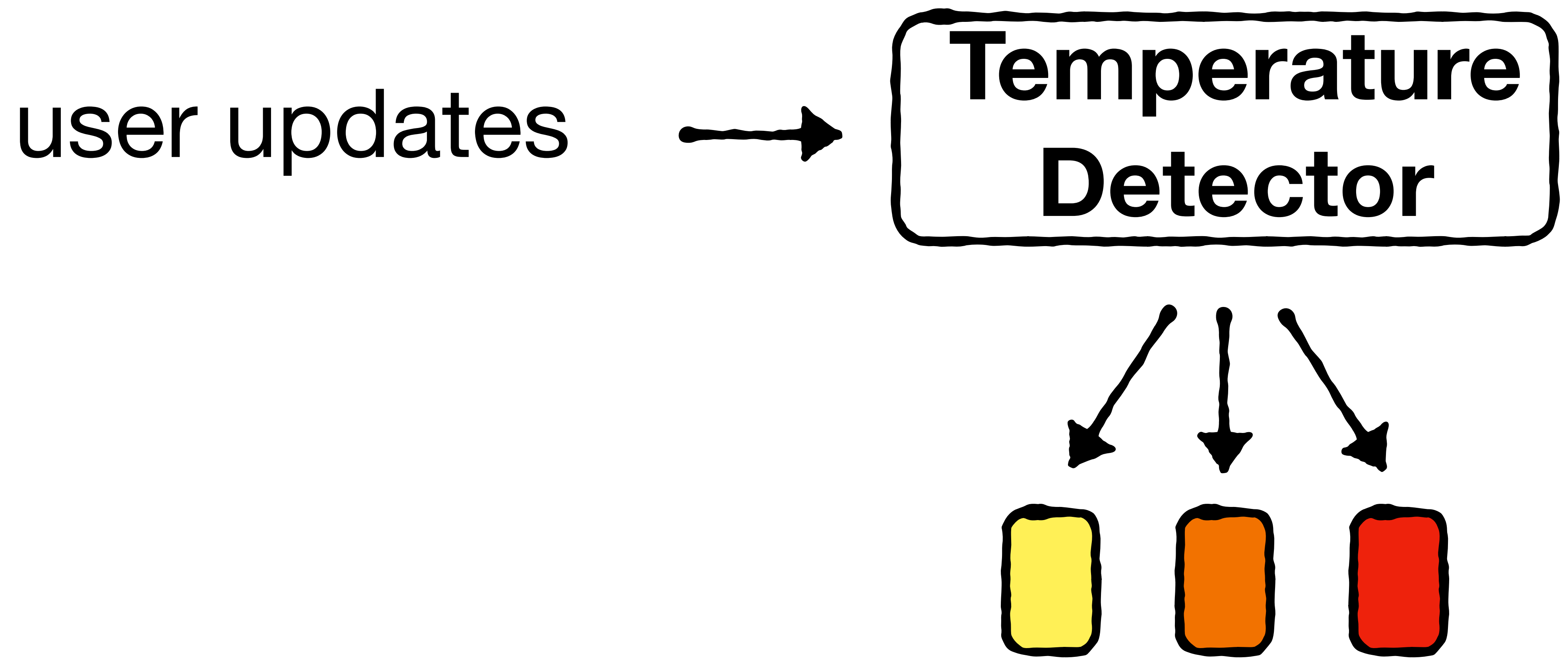


Estimates how likely a page is to be updated again



Estimates how likely a page is to be updated again

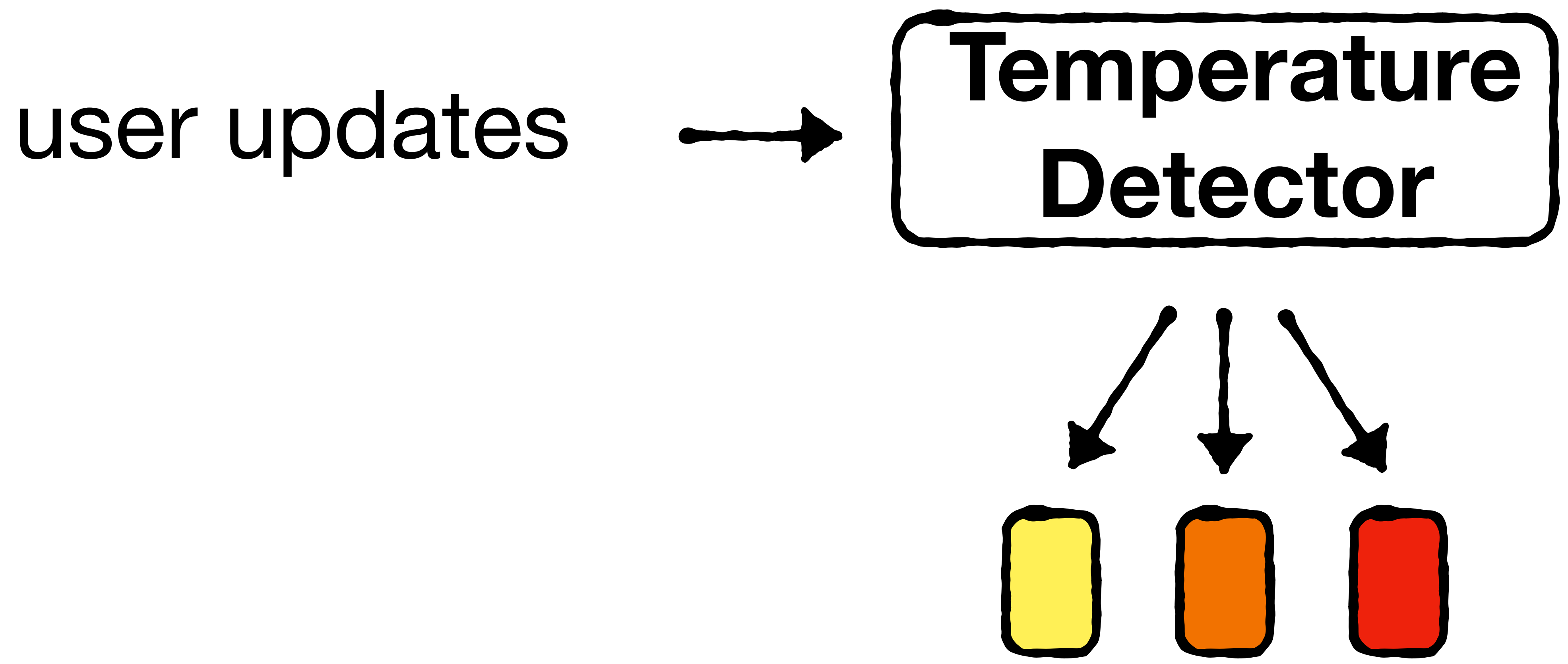
Should be a light-weight data structure that can fit in memory



Estimates how likely a page is to be updated again

Should be a light-weight data structure that can fit in memory

There is a lot of research on this, but we'll explore just one solution relying on a cool data structure called count-min



Count-Min

A data structure that reports that frequency of elements in a data stream

0 0 0 0 0 0 0 0 0 0 0 0

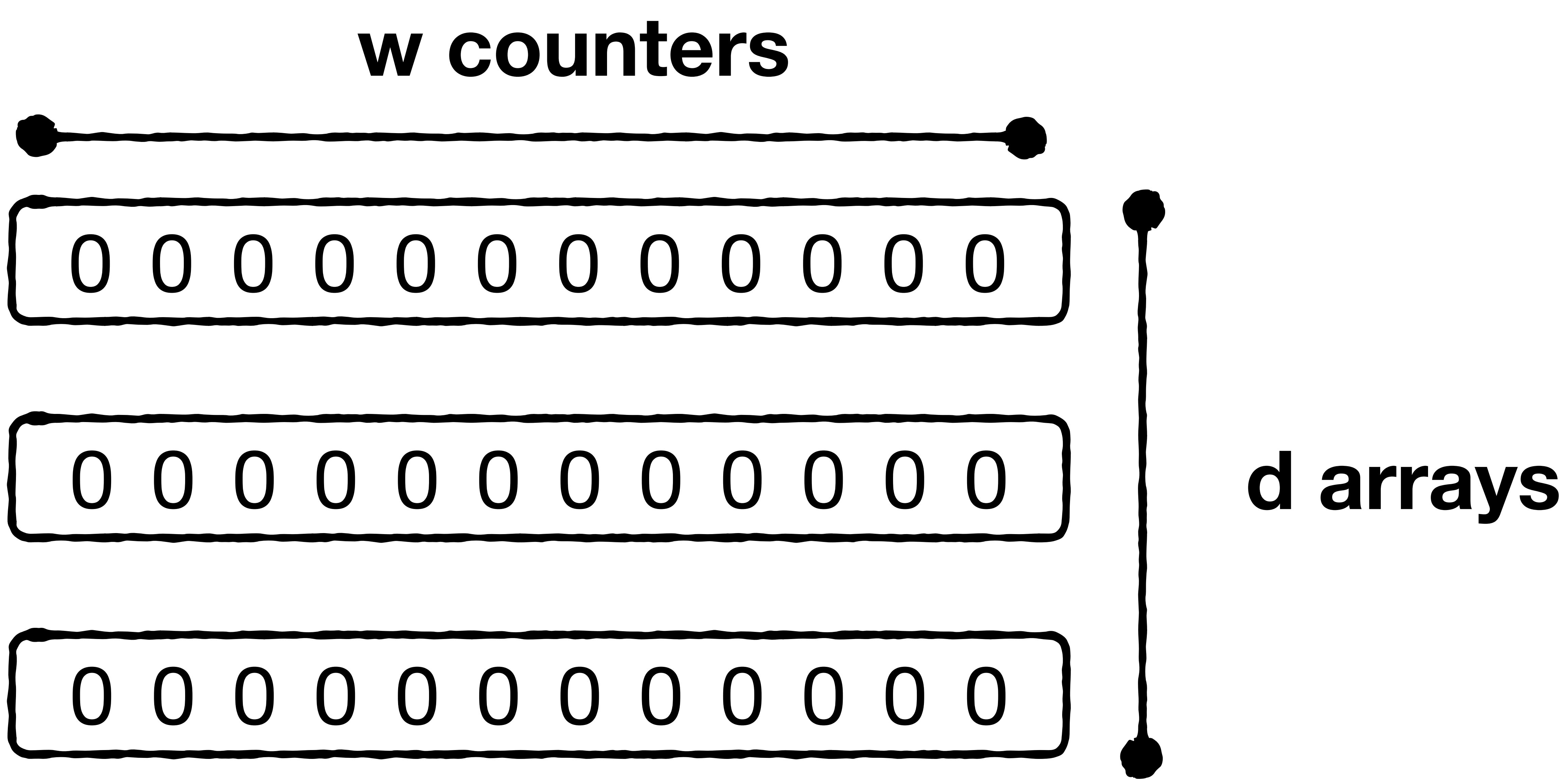
0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

Count-Min

A data structure that reports that frequency of elements in a data stream

Consists of d arrays of w counters each

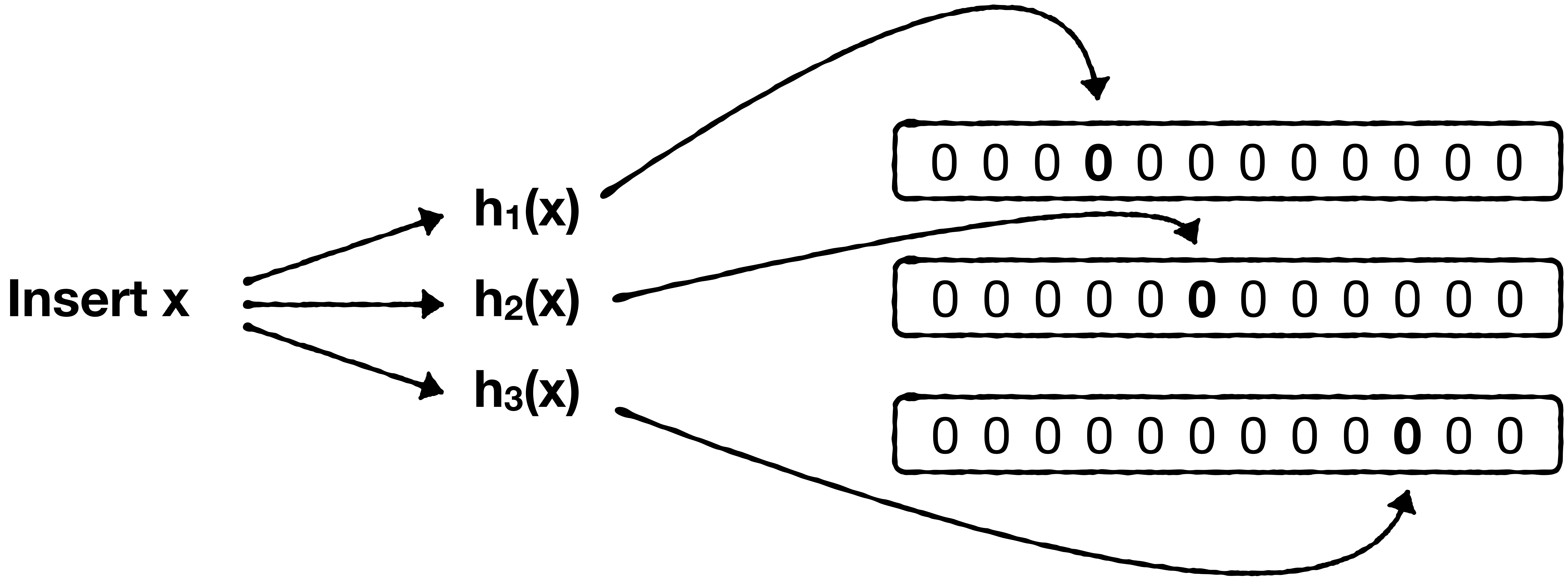


Count-Min

A data structure that reports that frequency of elements in a data stream

Consists of d arrays of w counters each

Insert an entry by hashing it to one counter in each array using different hash function



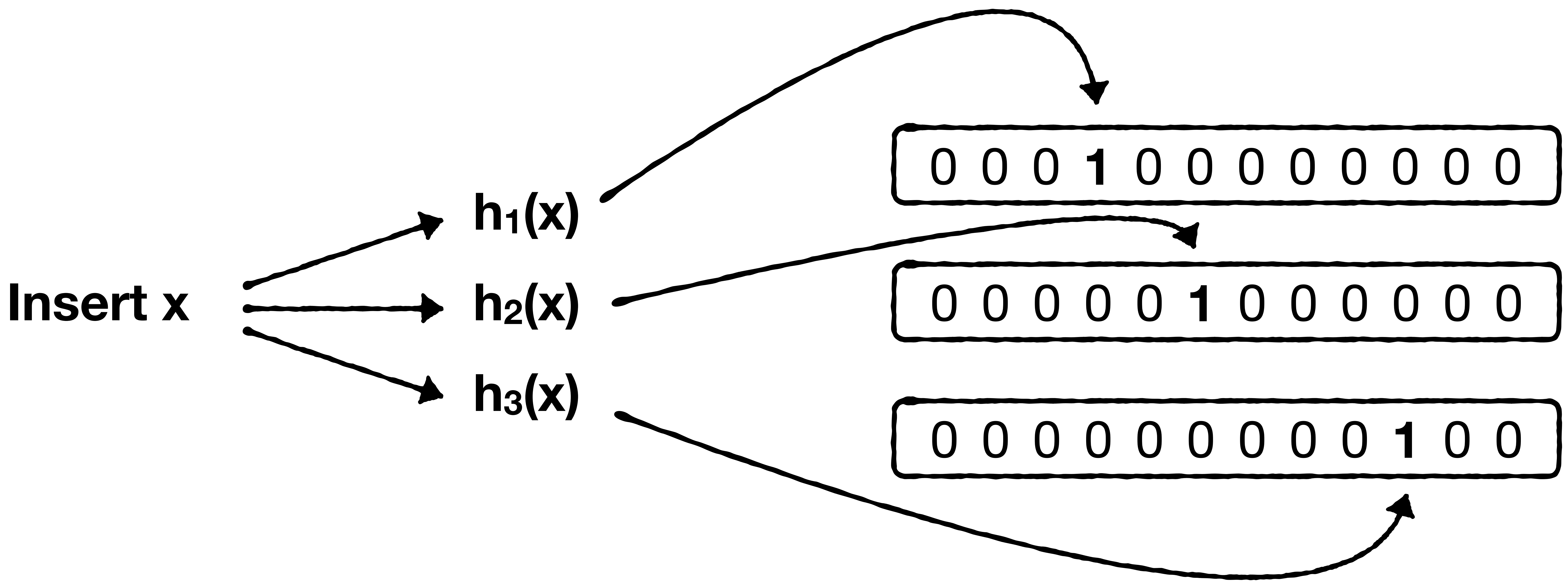
Count-Min

A data structure that reports that frequency of elements in a data stream

Consists of d arrays of w counters each

Insert an entry by hashing it to one counter in each array using different hash function

And increment that counter



Count-Min

After a while counters have a wide distribution of values

3	5	1	9	0	4	0	1	0	0	6	0

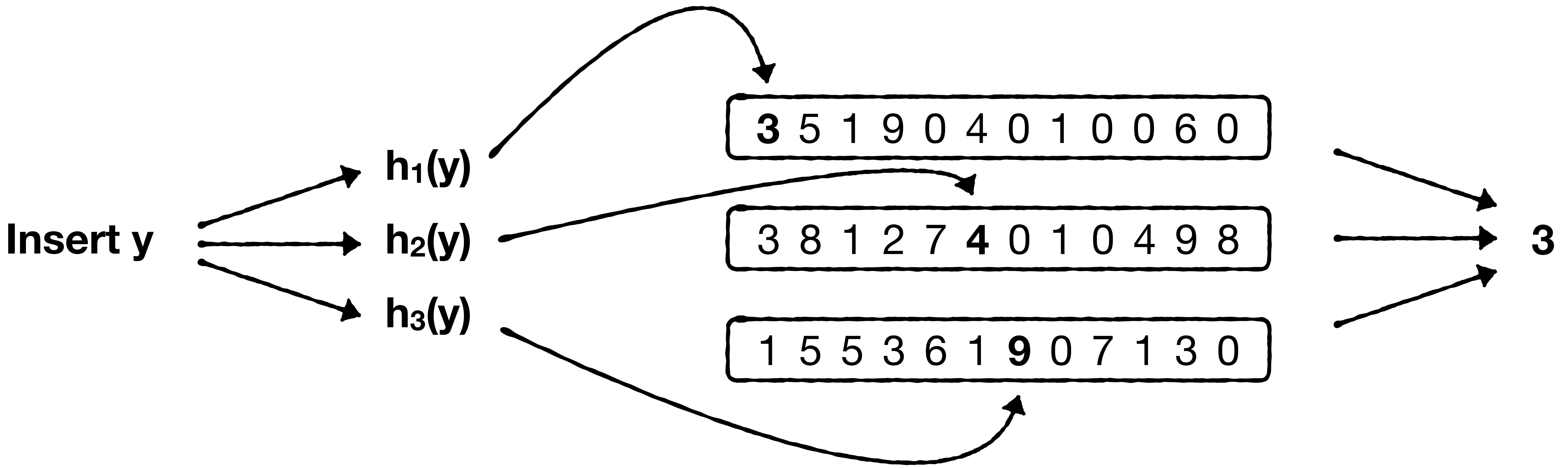
3	8	1	2	7	4	0	1	0	4	9	8

1	5	5	3	6	1	9	0	7	1	3	0

Count-Min

After a while counters have a wide distribution of values

To query for the frequency of a key, hash it to each array and return minimum

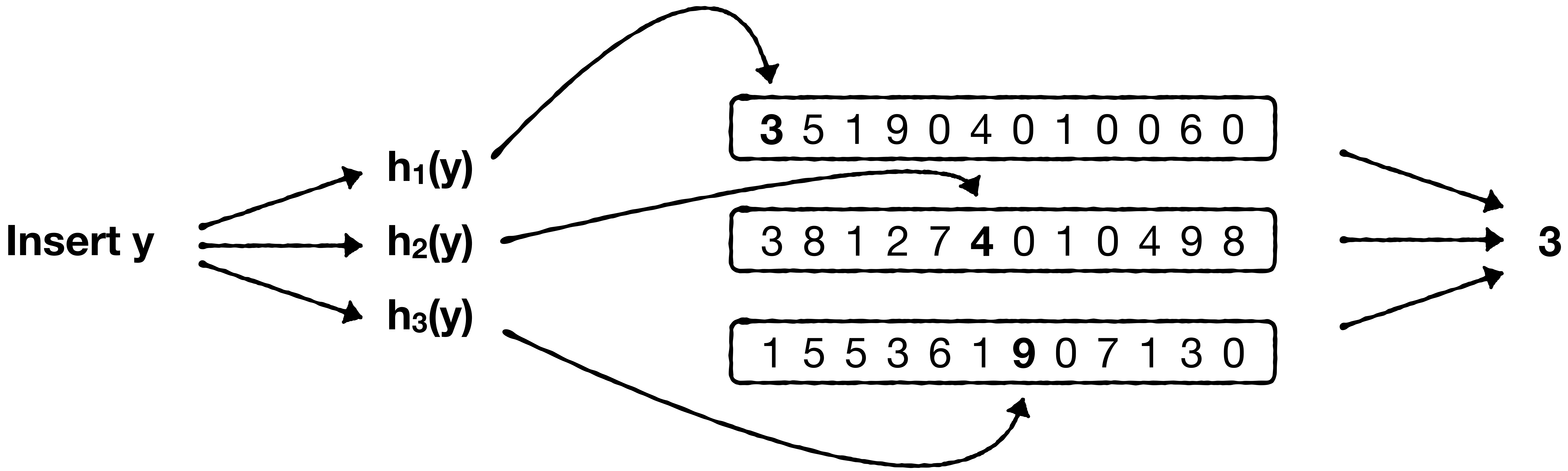


Count-Min

After a while counters have a wide distribution of values

To query for the frequency of a key, hash it to each array and return minimum

This result is a guaranteed upper bound of the real count



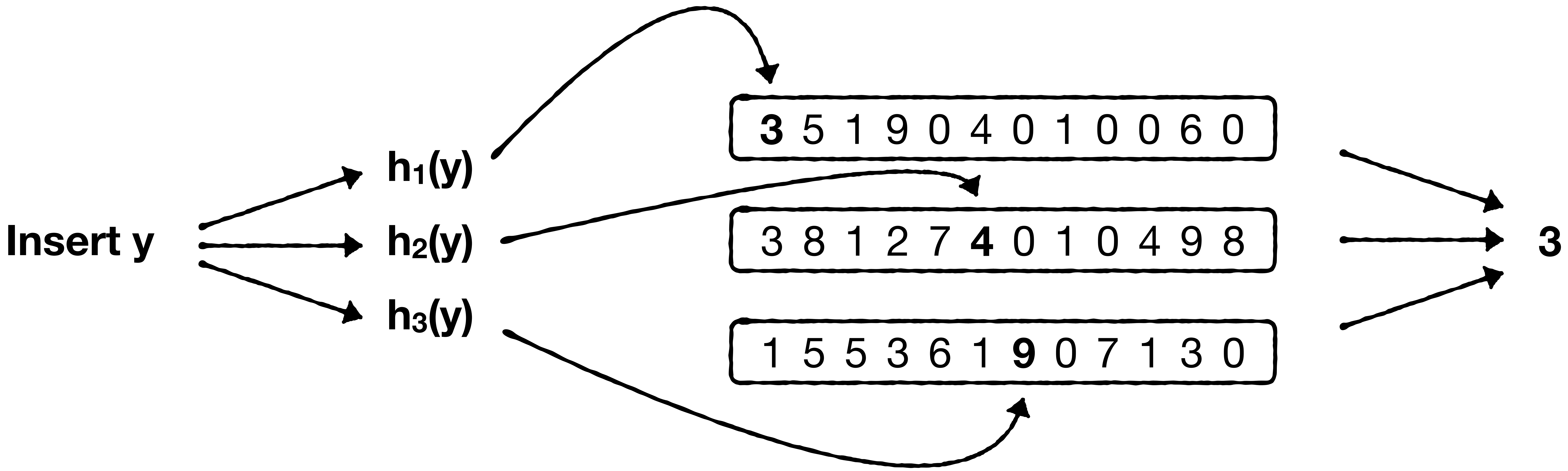
Count-Min

After a while counters have a wide distribution of values

To query for the frequency of a key, hash it to each array and return minimum

This result is a guaranteed upper bound of the real count

Result might overestimate the true answer due to hash collisions



Count-Min

Estimated frequency ≤ true frequency + ε · num insertions **with prob. 1- δ**

3	5	1	9	0	4	0	1	0	0	6	0

3	8	1	2	7	**4**	0	1	0	4	9	8

1	5	5	3	6	1	**9**	0	7	1	3	0

Count-Min

Estimated frequency ≤ true frequency + $\boldsymbol{\varepsilon}$ · num insertions   with prob. 1- $\boldsymbol{\delta}$

Parameters

3 5 1 9 0 4 0 1 0 0 6 0

3 8 1 2 7 **4** 0 1 0 4 9 8

1 5 5 3 6 1 **9** 0 7 1 3 0

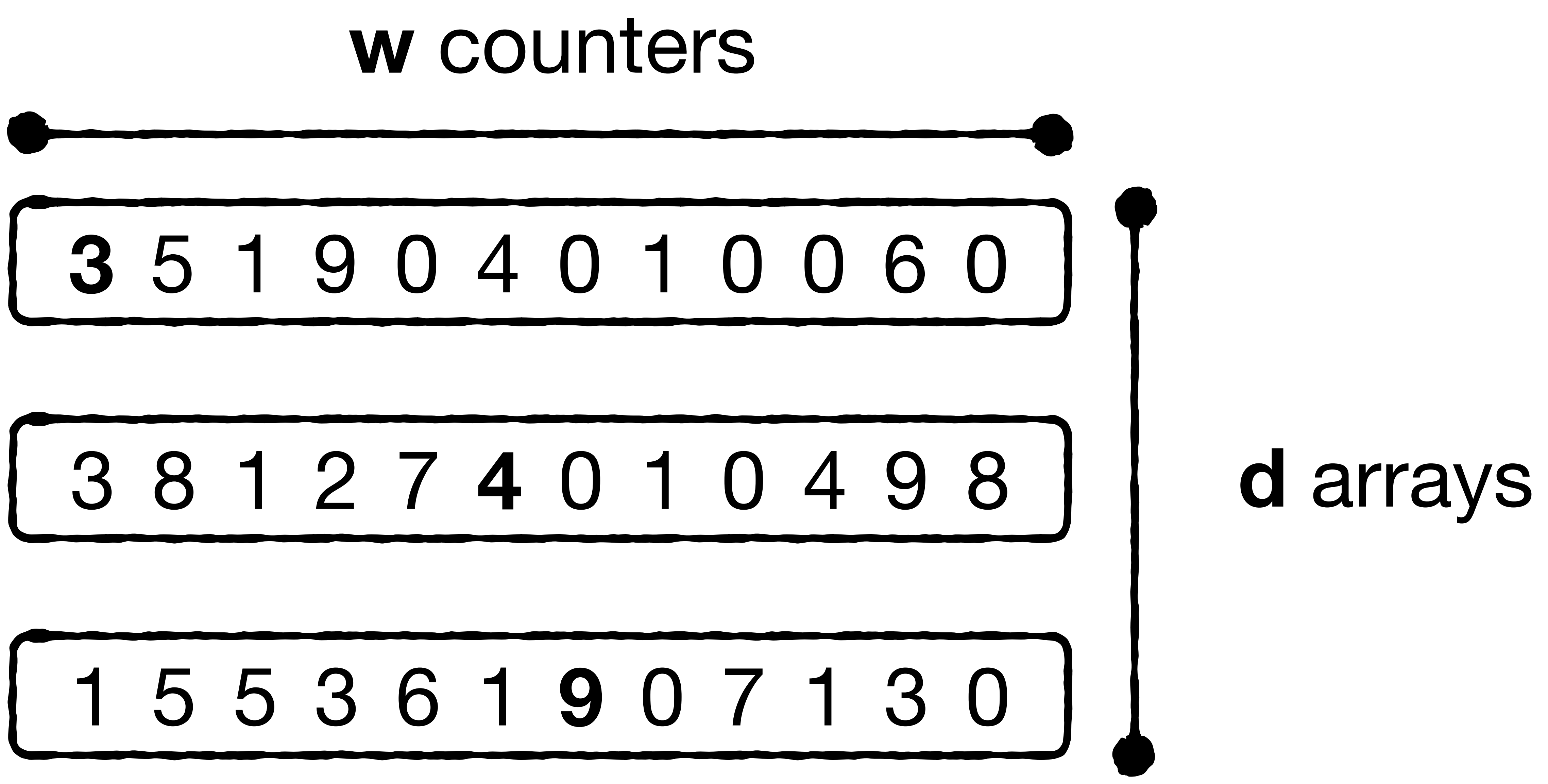
Count-Min

Estimated frequency ≤ true frequency + **ε** · num insertions with prob. 1- **δ**

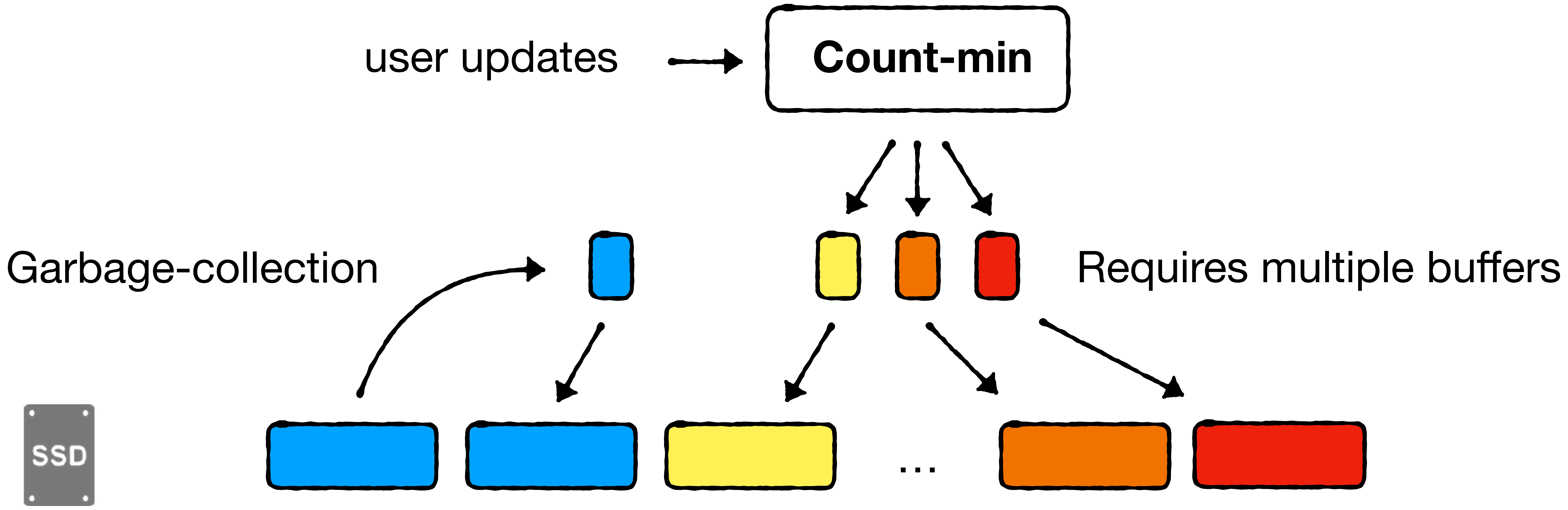
$w = \lceil e/\varepsilon \rceil$

$d = \lceil \ln 1/\delta \rceil$

The parameters determine the values of w and d

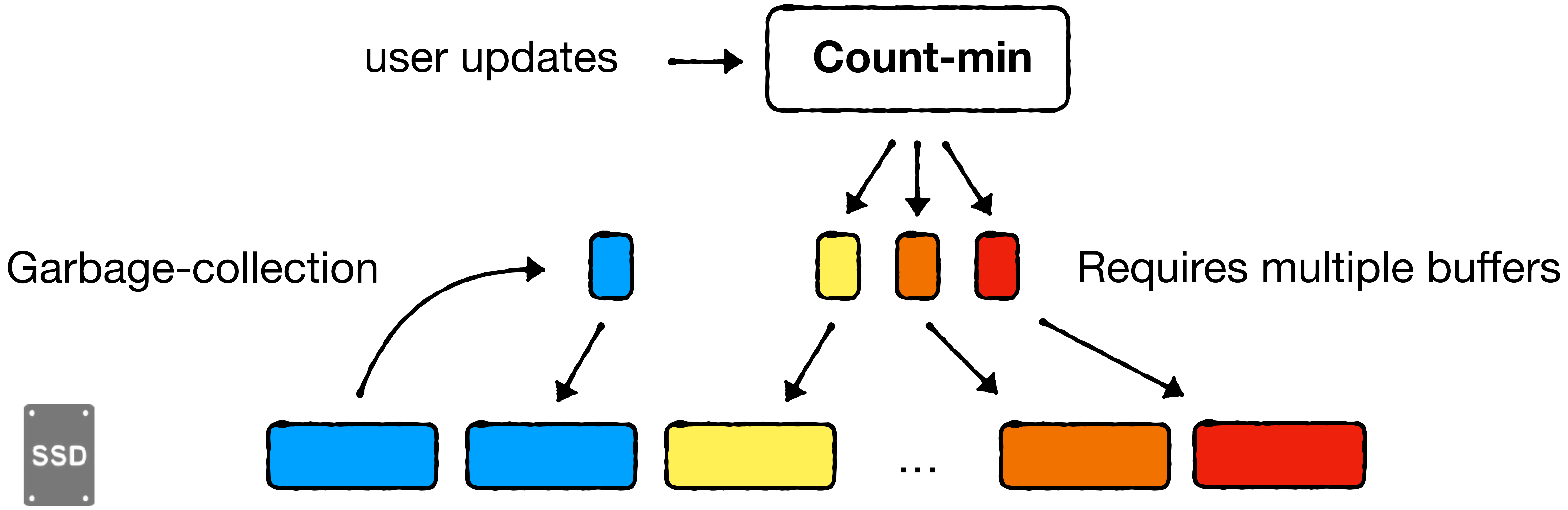


We can employ count-min to estimate frequency



We can employ count-min to estimate frequency

Pitfall: a value was very hot in the past but became cold. Count-min would still tell us its hot. Solutions?



Decay: every x insertions, we can divide all counters by 2.

3	5	1	9	0	4	0	1	0	0	6	0

3	8	1	2	7	4	0	1	0	4	9	8

1	5	5	3	6	1	9	0	7	1	3	0

Decay: every x insertions, we can divide all counters by 2.

1	2	1	4	0	2	0	0	0	0	3	0

1	4	0	1	3	2	0	0	0	2	4	4

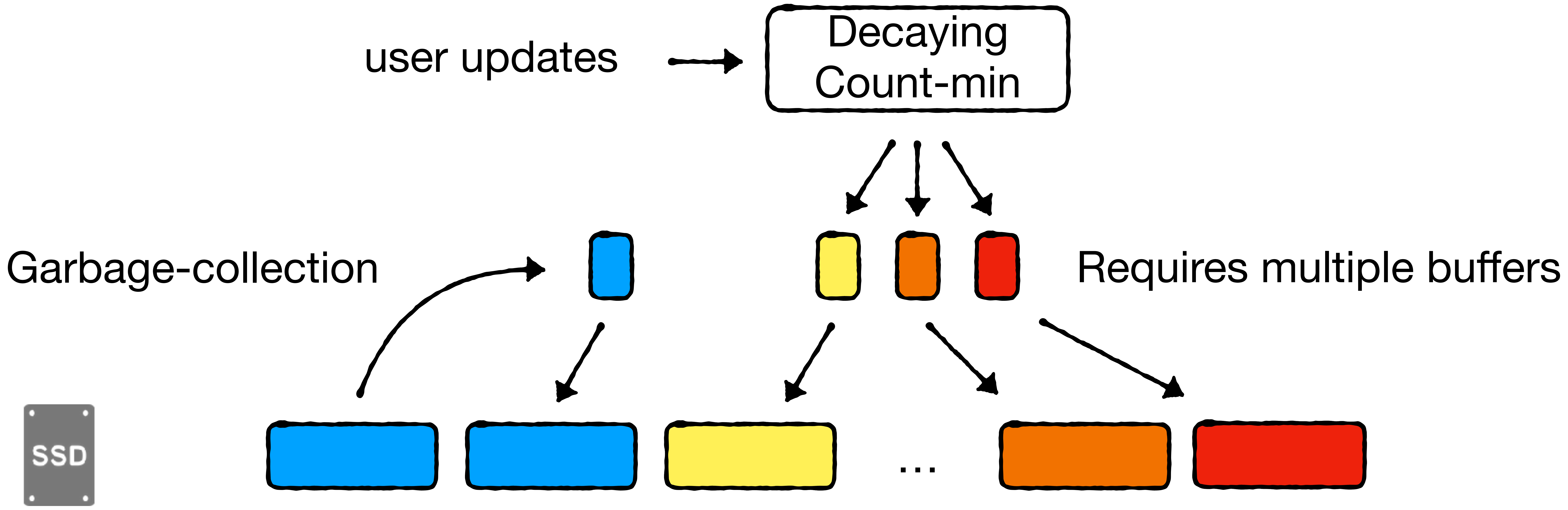
0	2	2	1	3	0	4	0	3	0	1	0

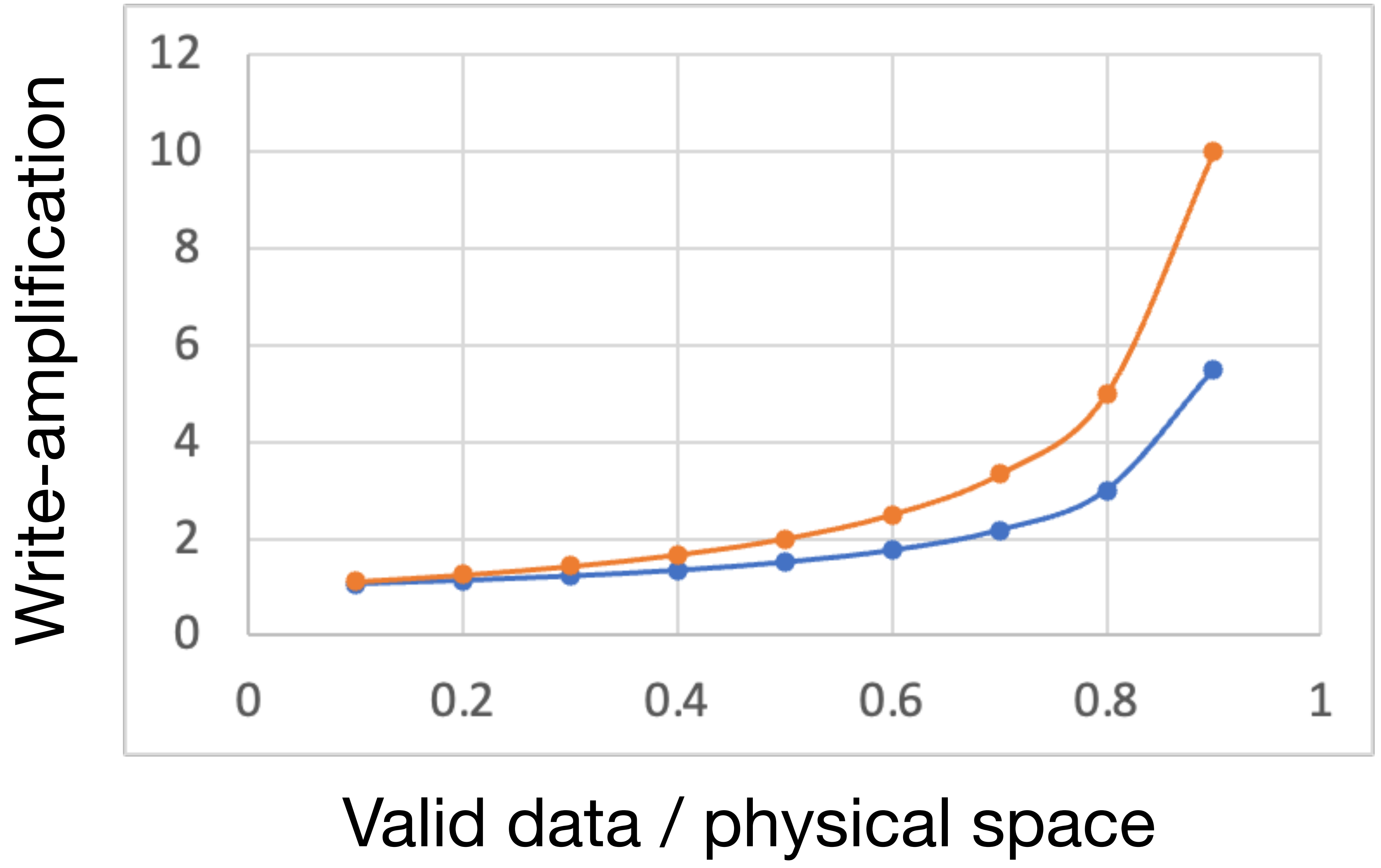
In addition, we can safeguard against counter overflows by not incrementing counters that have reached their maximum value.

1 2 1 4 0 2 0 0 0 0 3 0

1 4 0 1 3 2 0 0 0 2 4 4

0 2 2 1 3 0 4 0 3 0 1 0


user updates
Decaying
Count-min
Garbage-collection
Requires multiple buffers
SSD
...



$$1 + \frac{L/P}{1 - L/P}$$ ← Mixing hot/cold data brings us towards worst-case

$$1 + \frac{L/P}{2 \cdot (1 - L/P)}$$ ← Uniformly random workloads

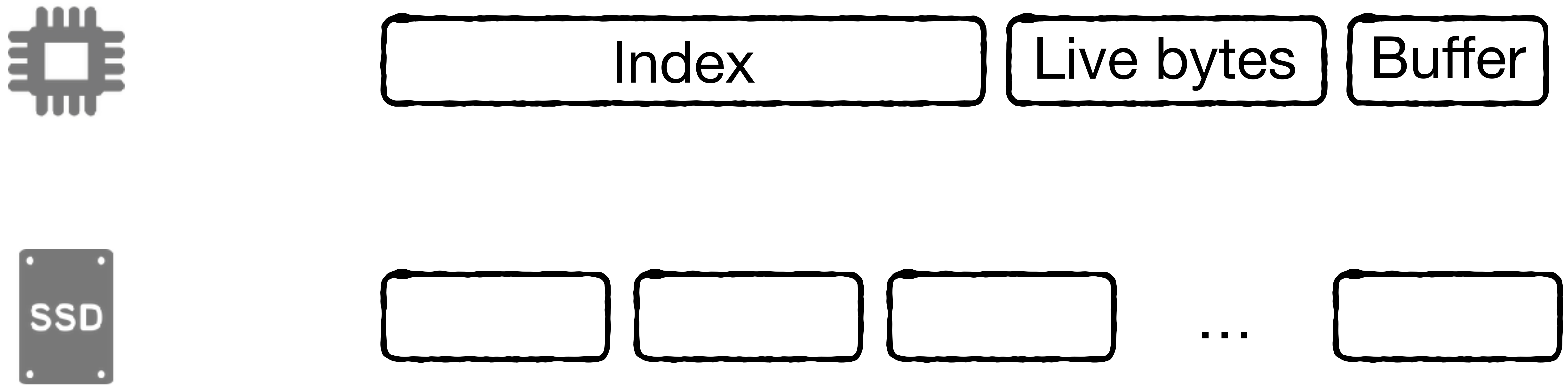
... ← **Hot/cold separation improves WA beyond uniform case**

$$\text{GC overheads} = \begin{cases} 1 + \dfrac{L/P}{1 - L/P} & \leftarrow \text{Mixing hot/cold data brings us towards worst-case} \\ 1 + \dfrac{L/P}{2 \cdot (1 - L/P)} & \leftarrow \text{Uniformly random workloads} \\ \ldots & \leftarrow \textbf{Hot/cold separation improves WA beyond uniform case} \end{cases}$$

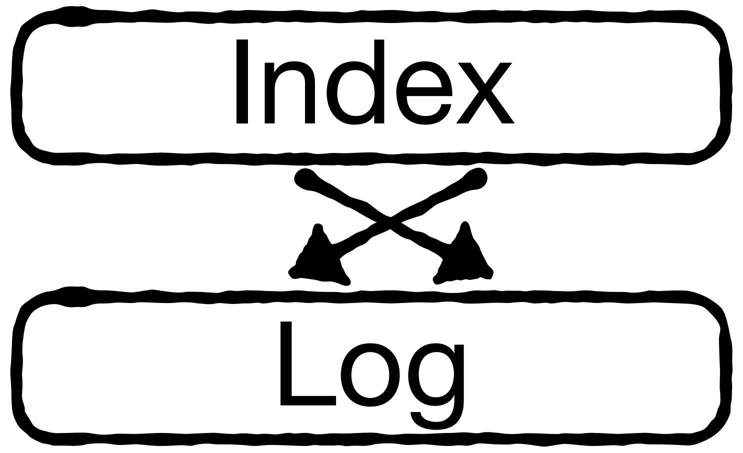
Overall Cost Analysis

Write cost: O(GC / B) I/O

Reads: 1 read I/O



Mechanics



Hot/Cold separation

To reduce write-amp

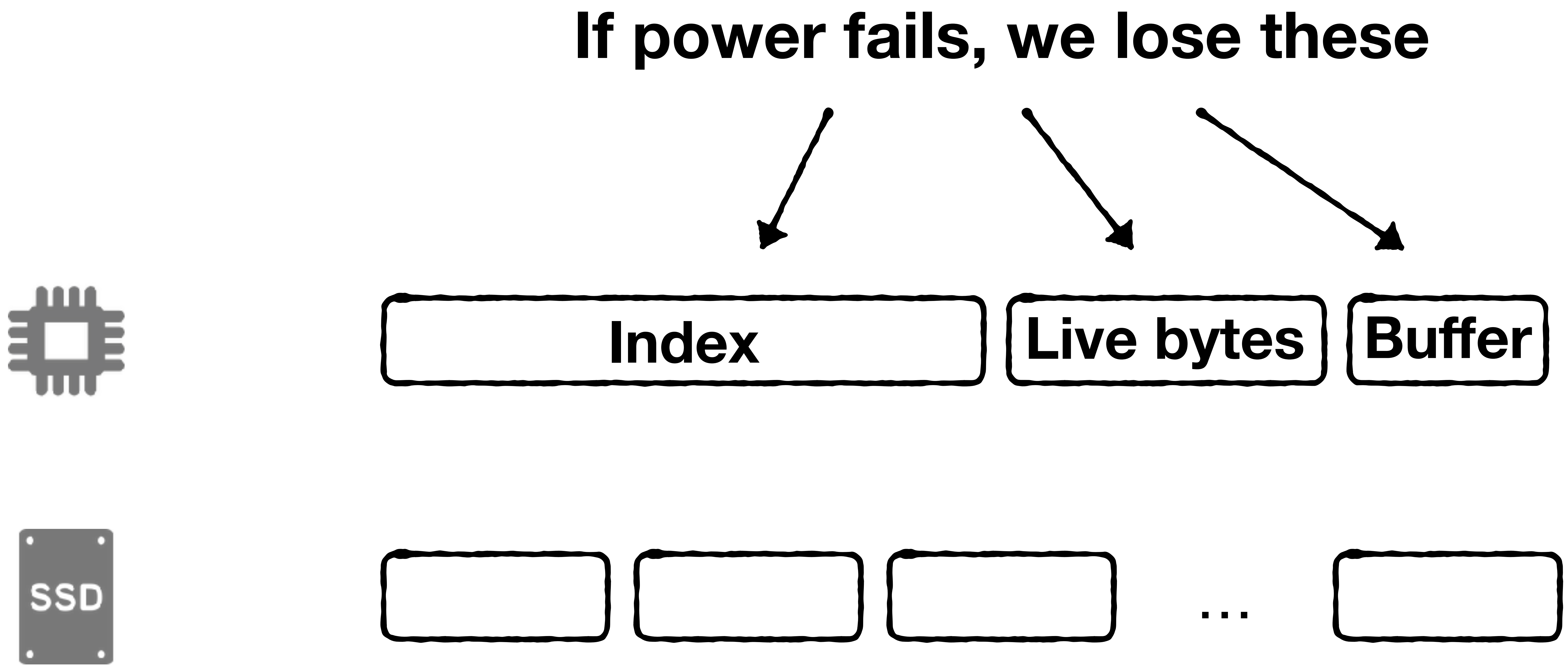
Checkpointing & Recovery

If power fails

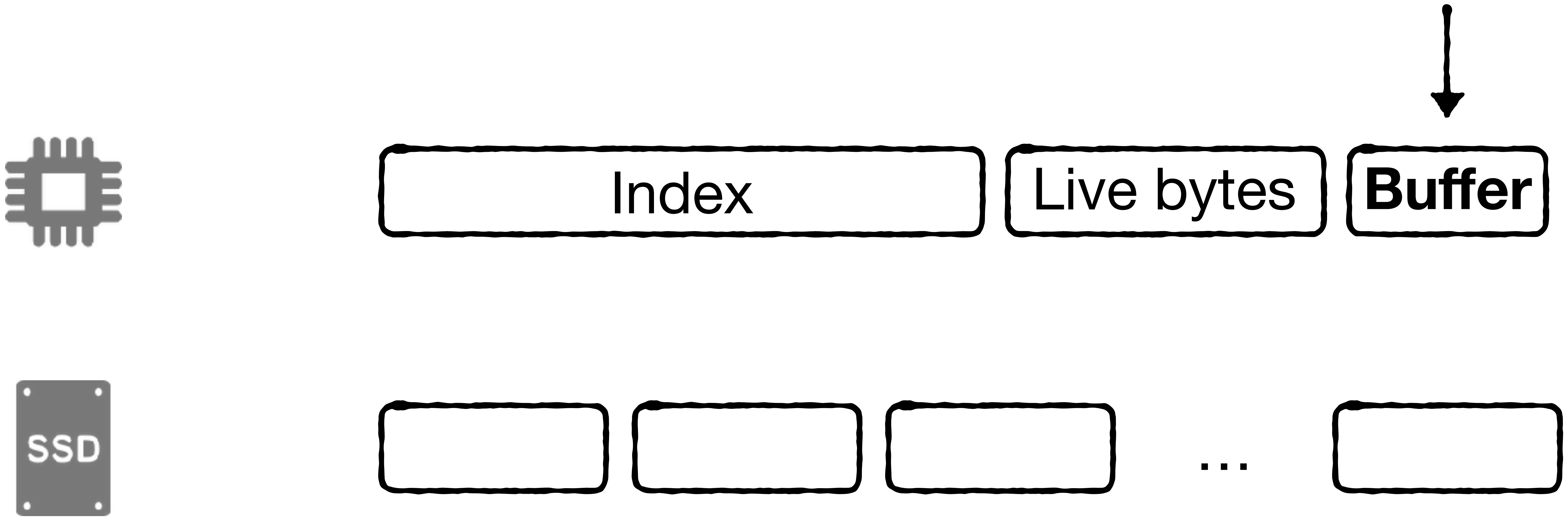
Cuckoo filtering

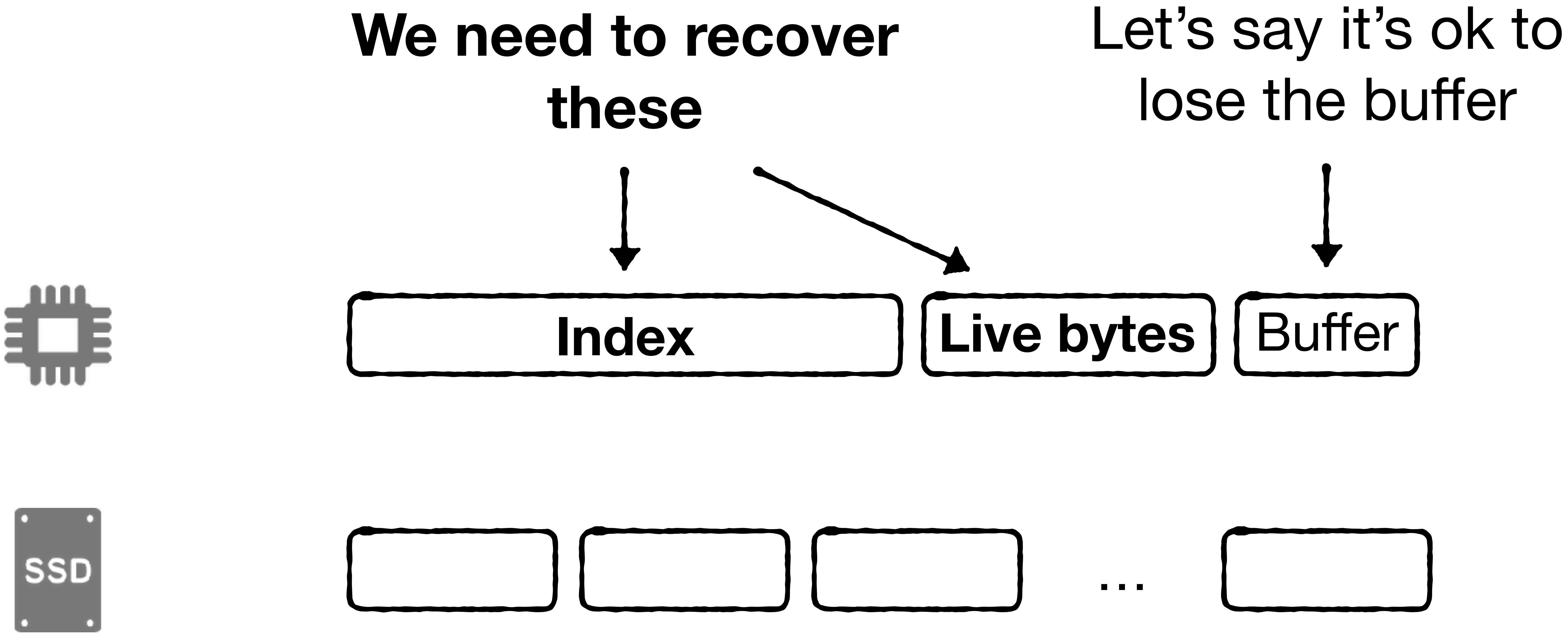
To reduce memory

Checkpointing & Recovery

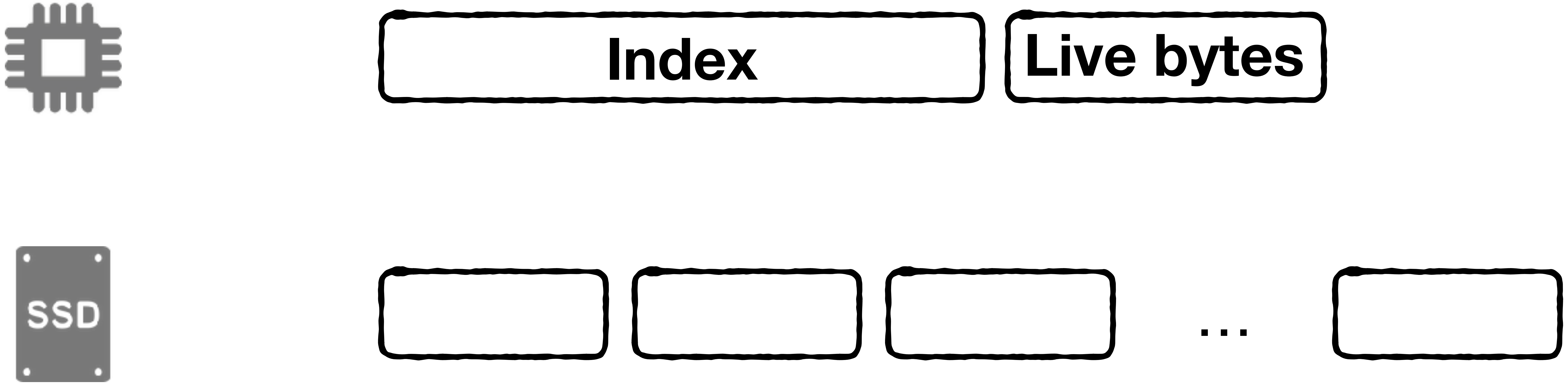


Let's say it's ok to lose the buffer

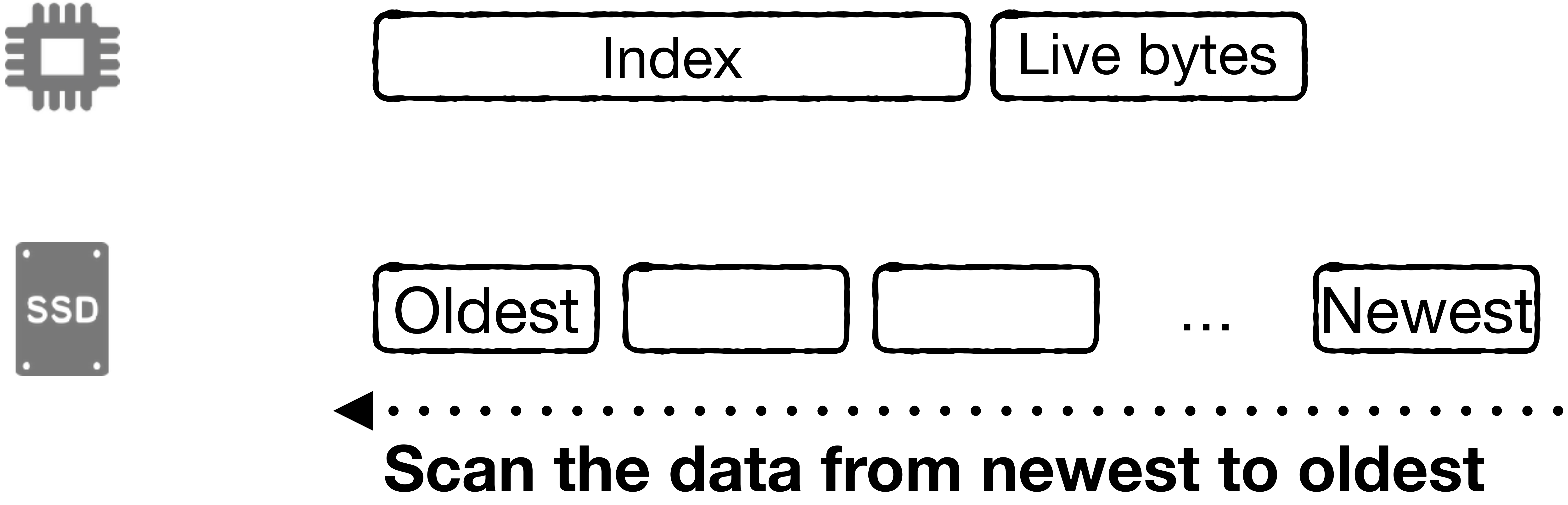



We need to recover these
Let's say it's ok to lose the buffer
Index
Live bytes
Buffer
SSD
...

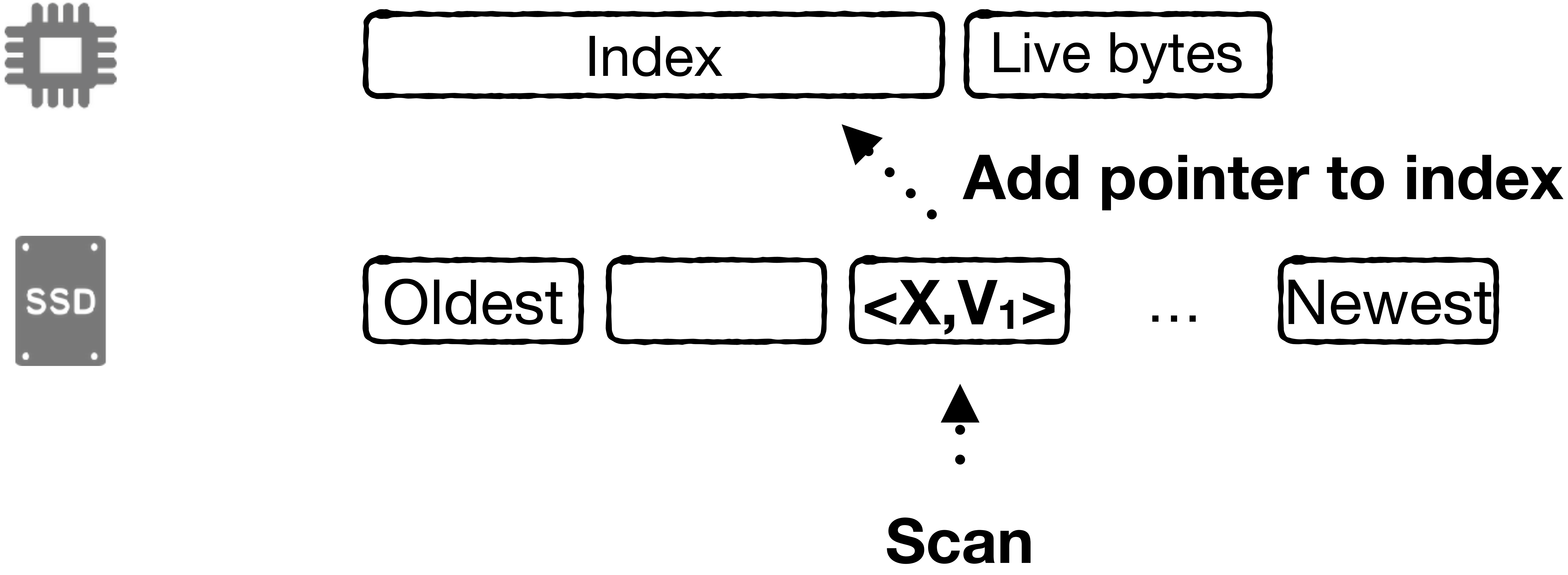
Simple recovery algorithm?



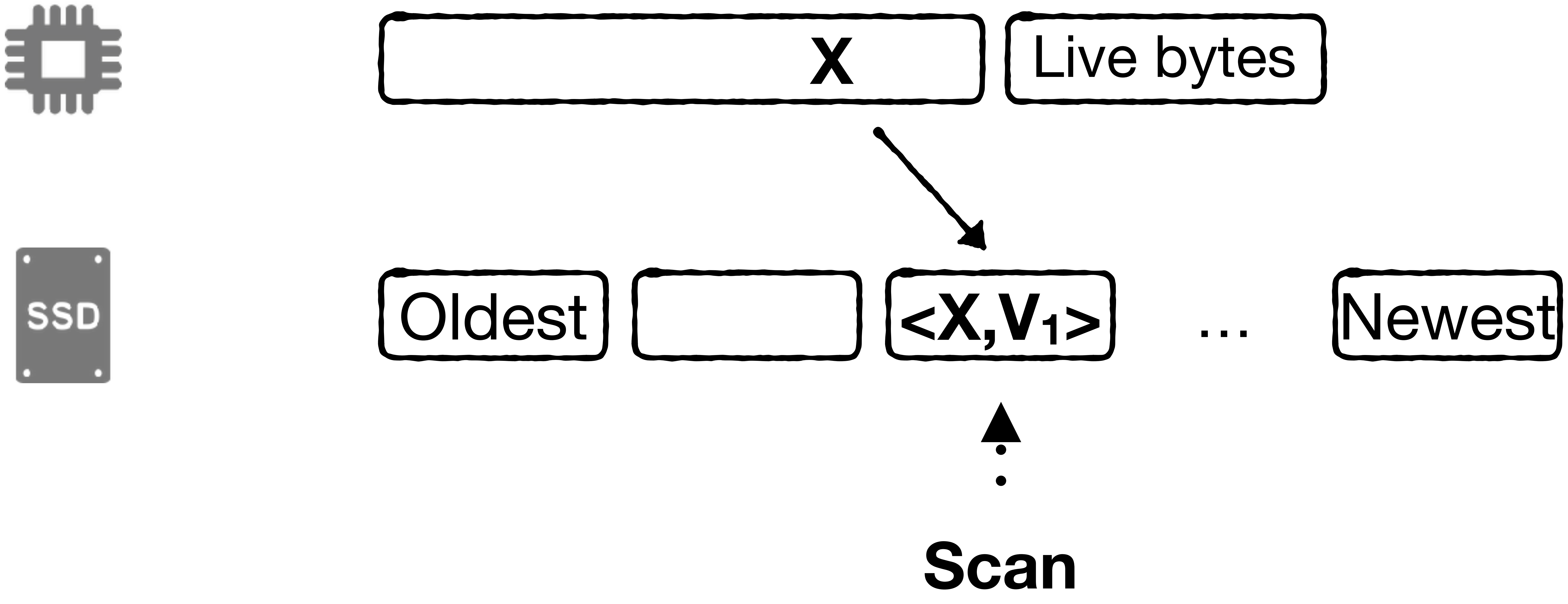
Simple recovery algorithm?



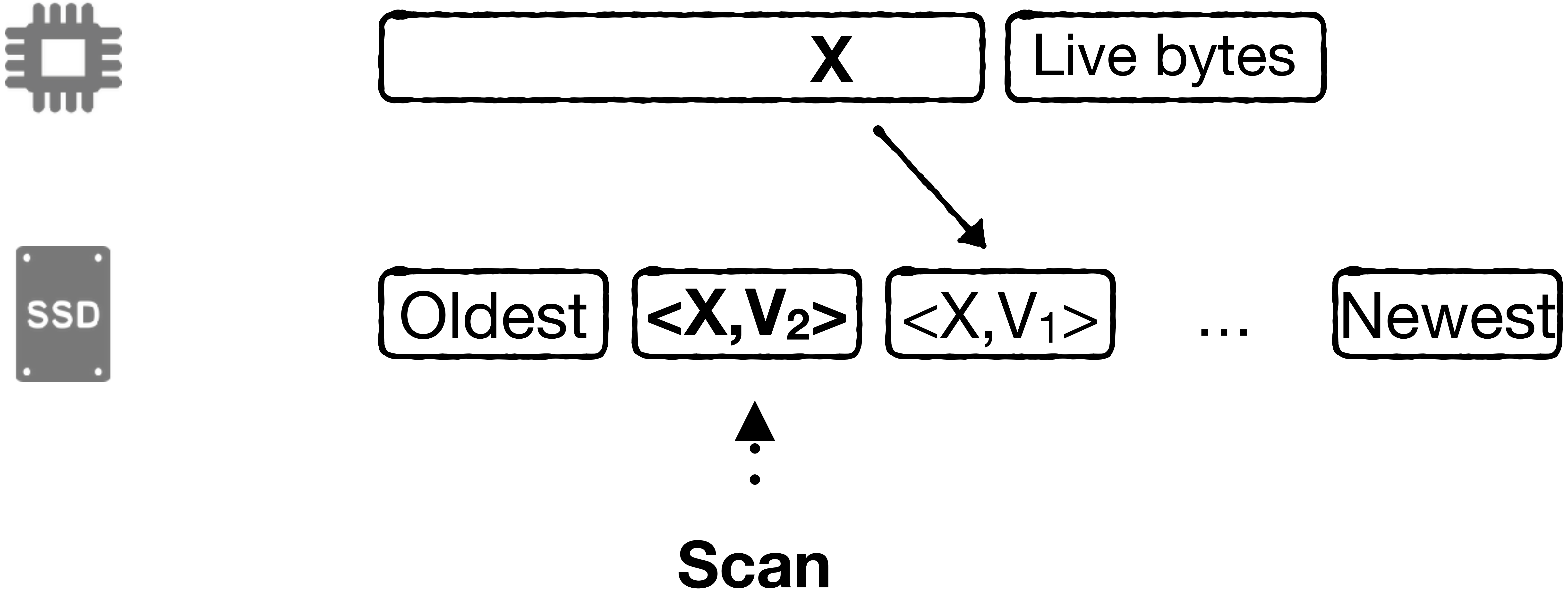
Simple recovery algorithm?



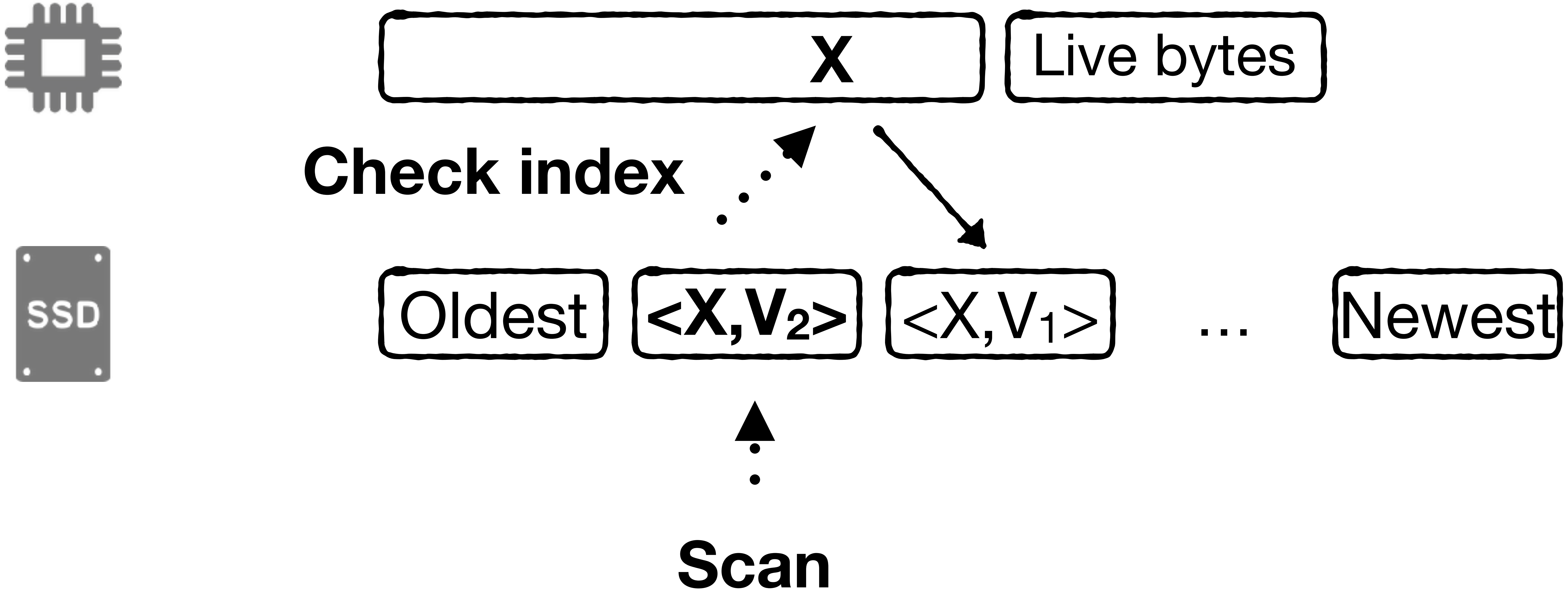
Simple recovery algorithm?



Simple recovery algorithm?

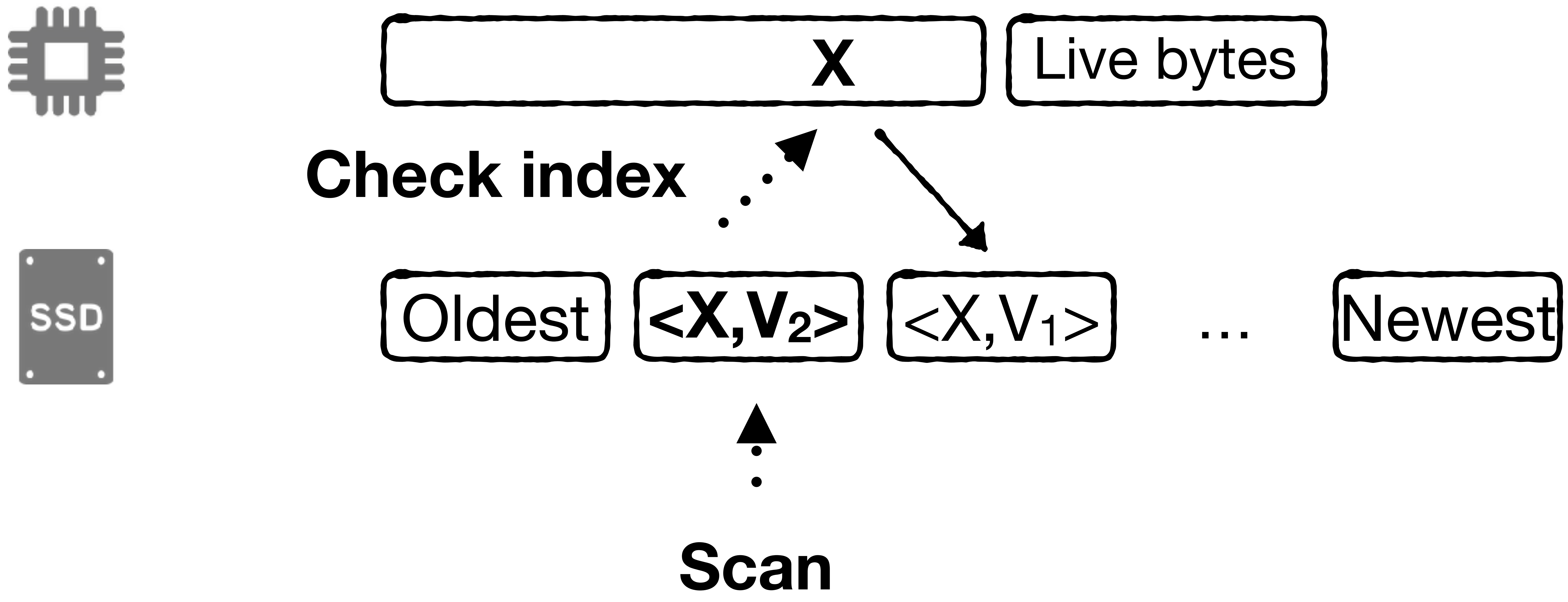


Simple recovery algorithm?



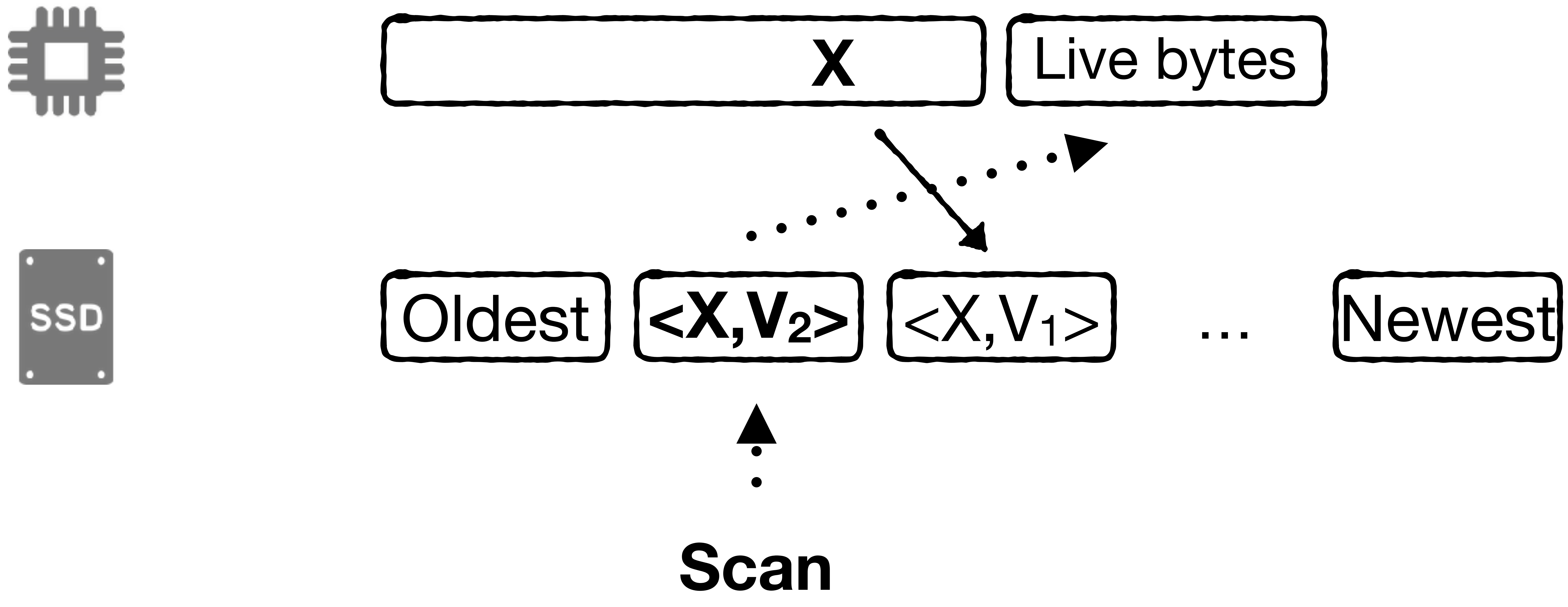
Simple recovery algorithm?

Since a mapping entry with the same key already exists, the current entry is obsolete.

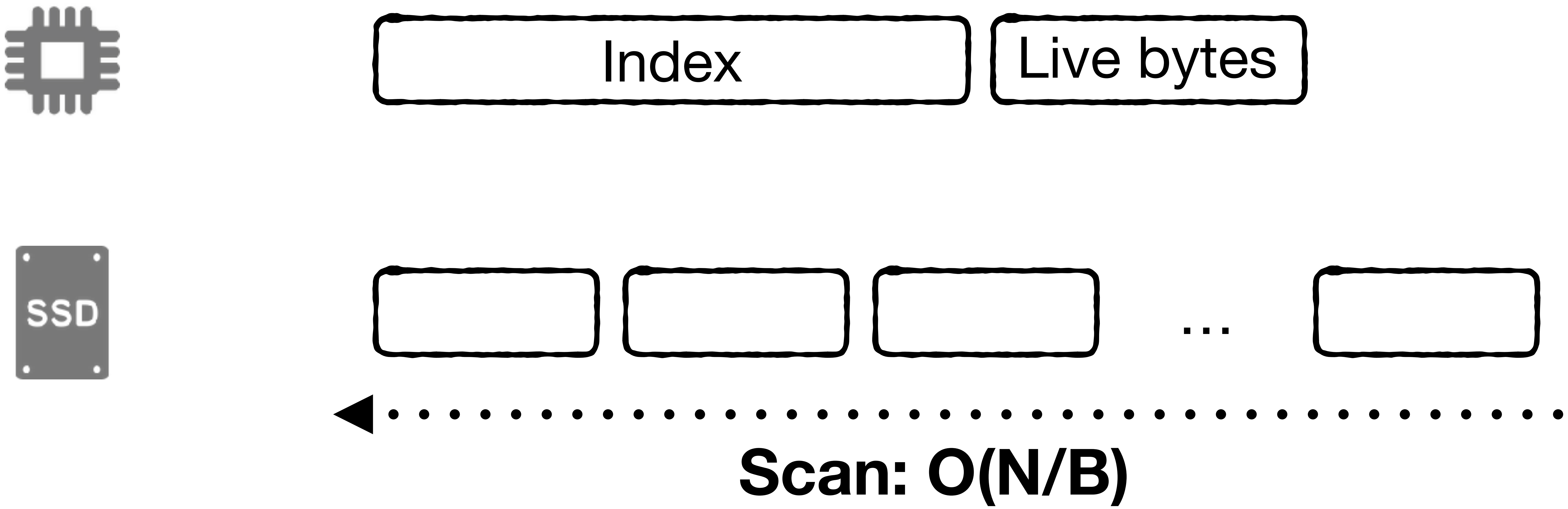


Simple recovery algorithm?

So we instead subtract size of entry from live bytes.

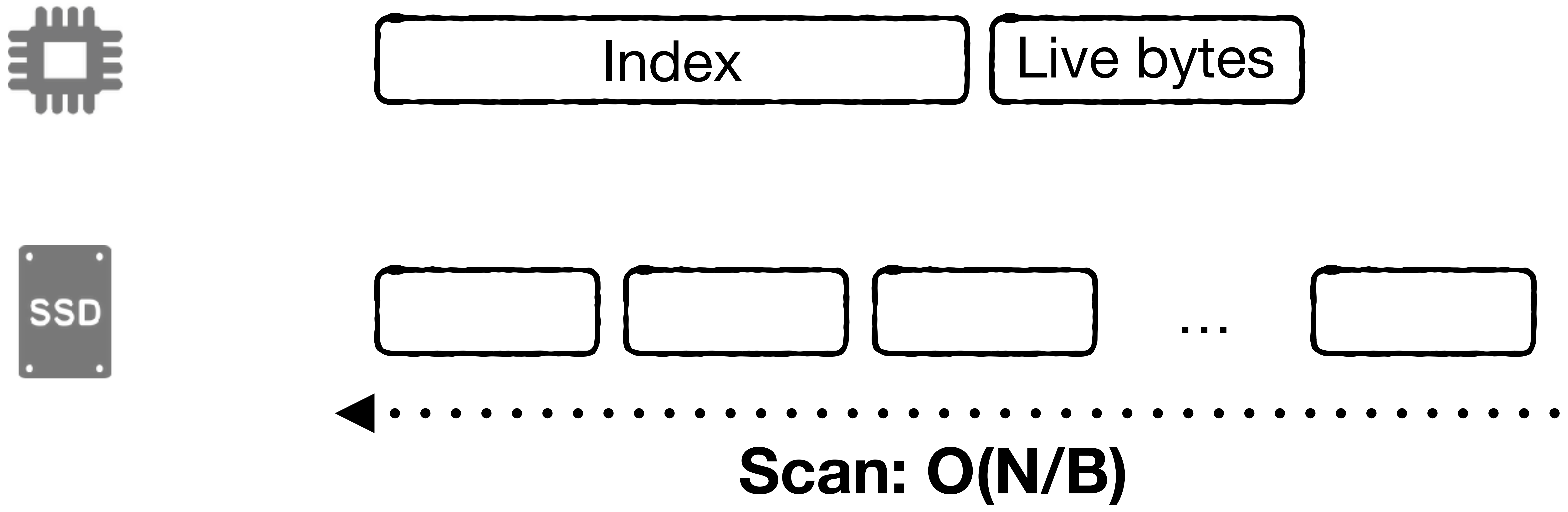


We can recover with one pass over the data



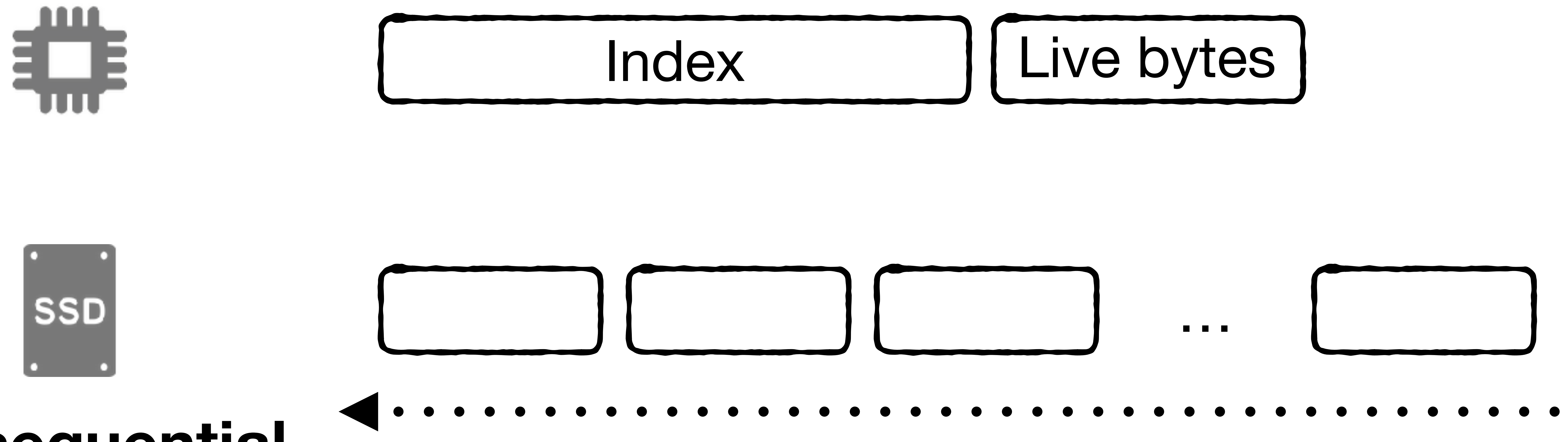
We can recover with one pass over the data

But if the data is huge, this can take a while.



We can recover with one pass over the data

But if the data is huge, this can take a while.



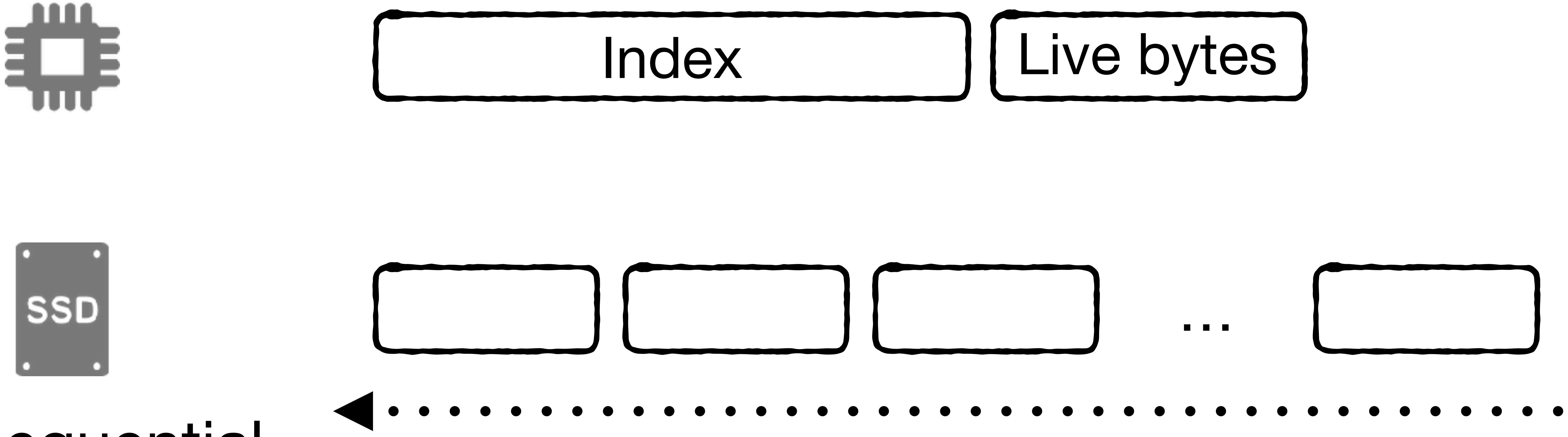
e.g., 200 MB/s sequential throughput

1 TB of data takes 1 hour to recover

We can recover with one pass over the data

But if the data is huge, this can take a while.

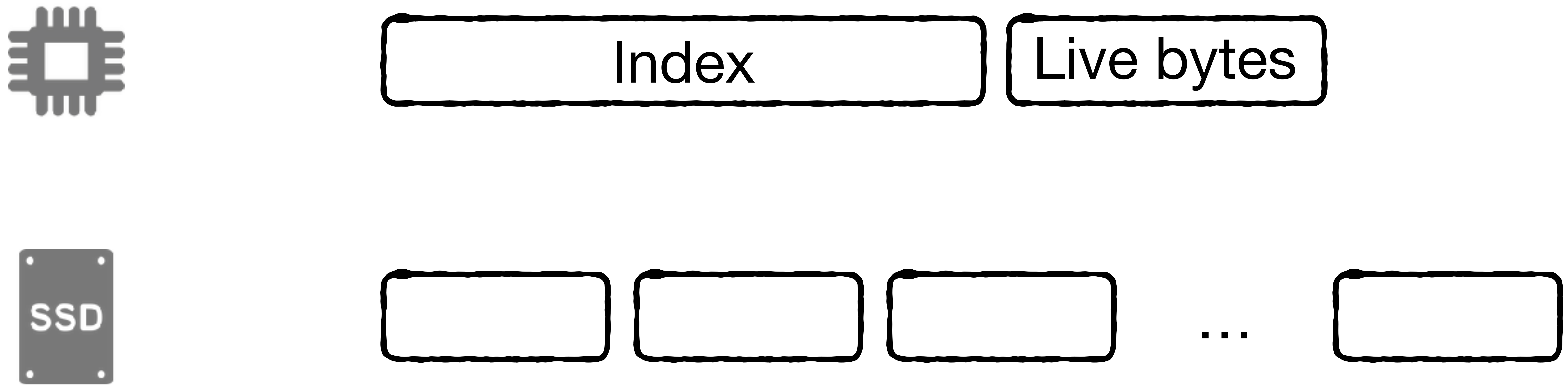
Any good ideas?



e.g., 200 MB/s sequential throughput

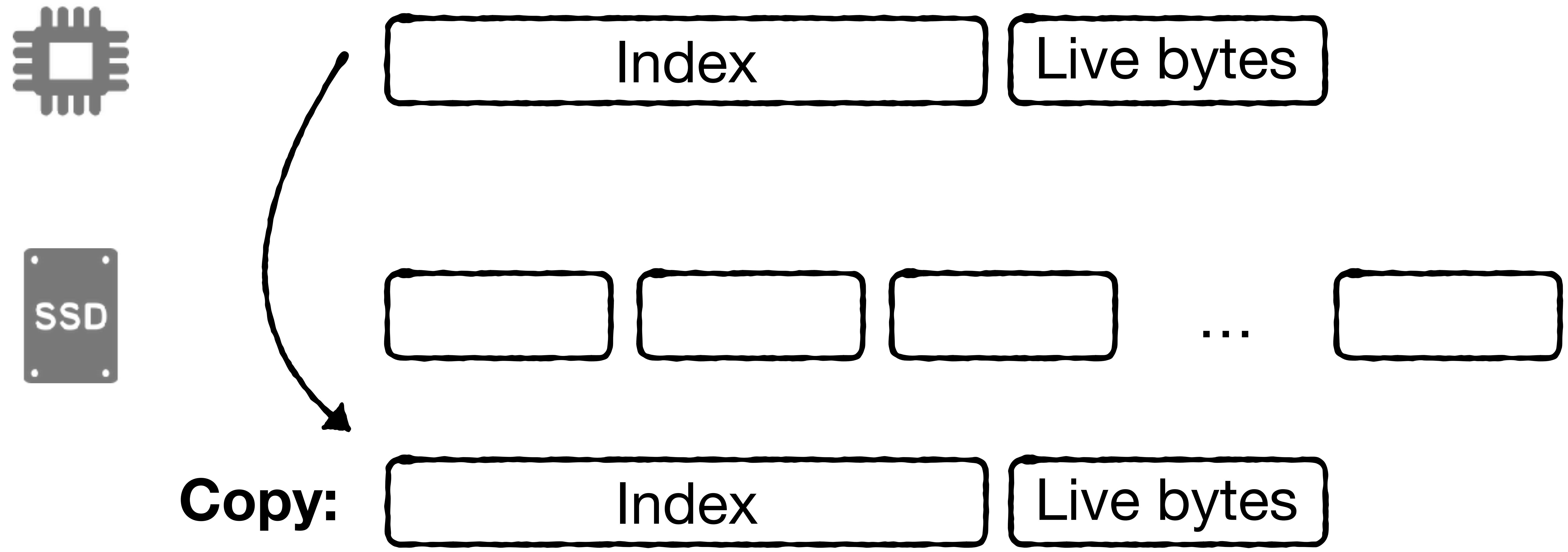
1 TB of data takes 1 hour to recover

Checkpointing



Checkpointing

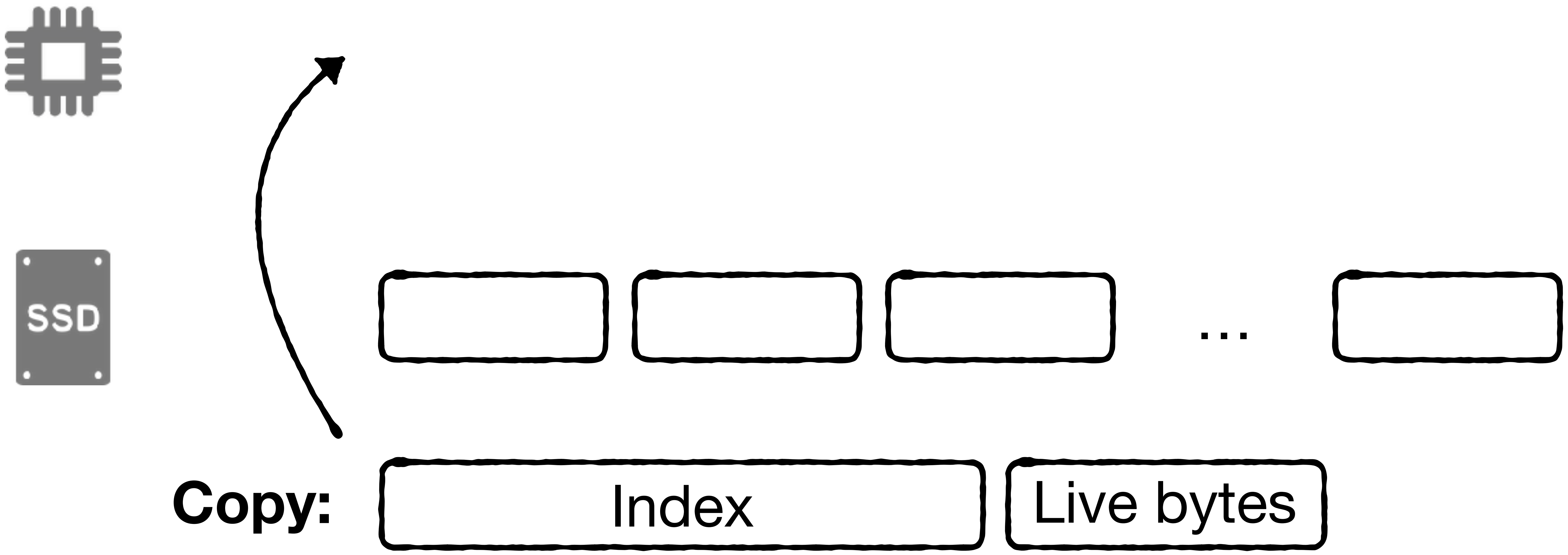
Every X updates/insertions, store copy of index & live bytes



Checkpointing

Every X updates/insertions, store copy of index & live bytes

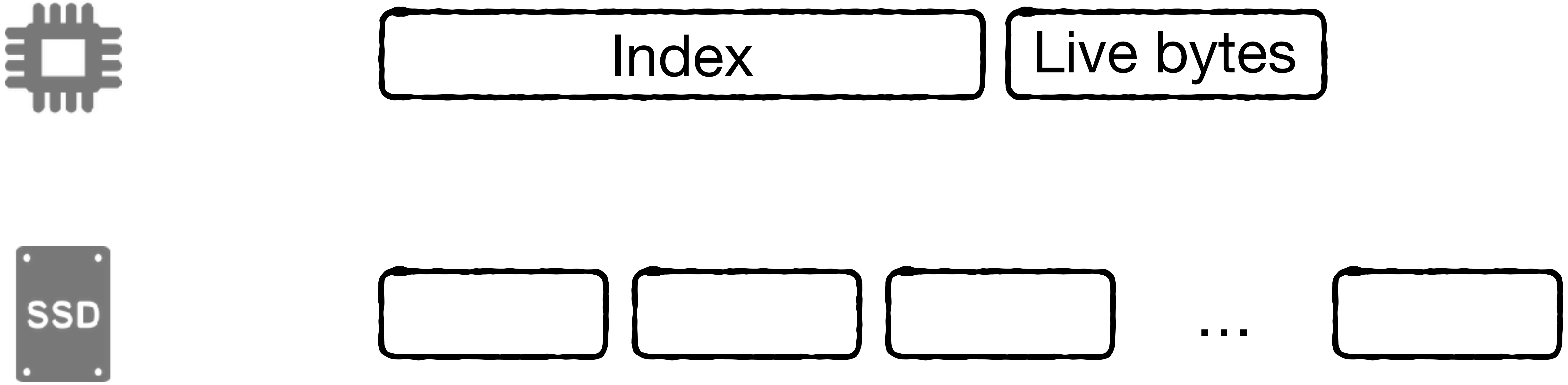
After recovery, load copy



Checkpointing

Every X updates/insertions, store copy of index & live bytes

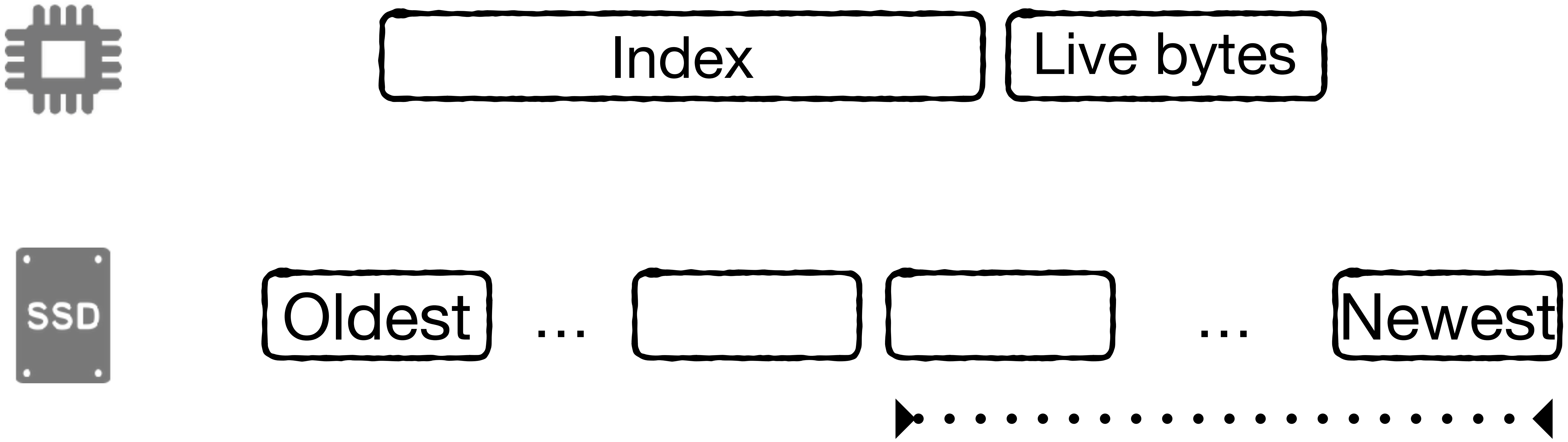
After recovery, load copy



Checkpointing

Every X updates/insertions, store copy of index & live bytes

After recovery, load copy

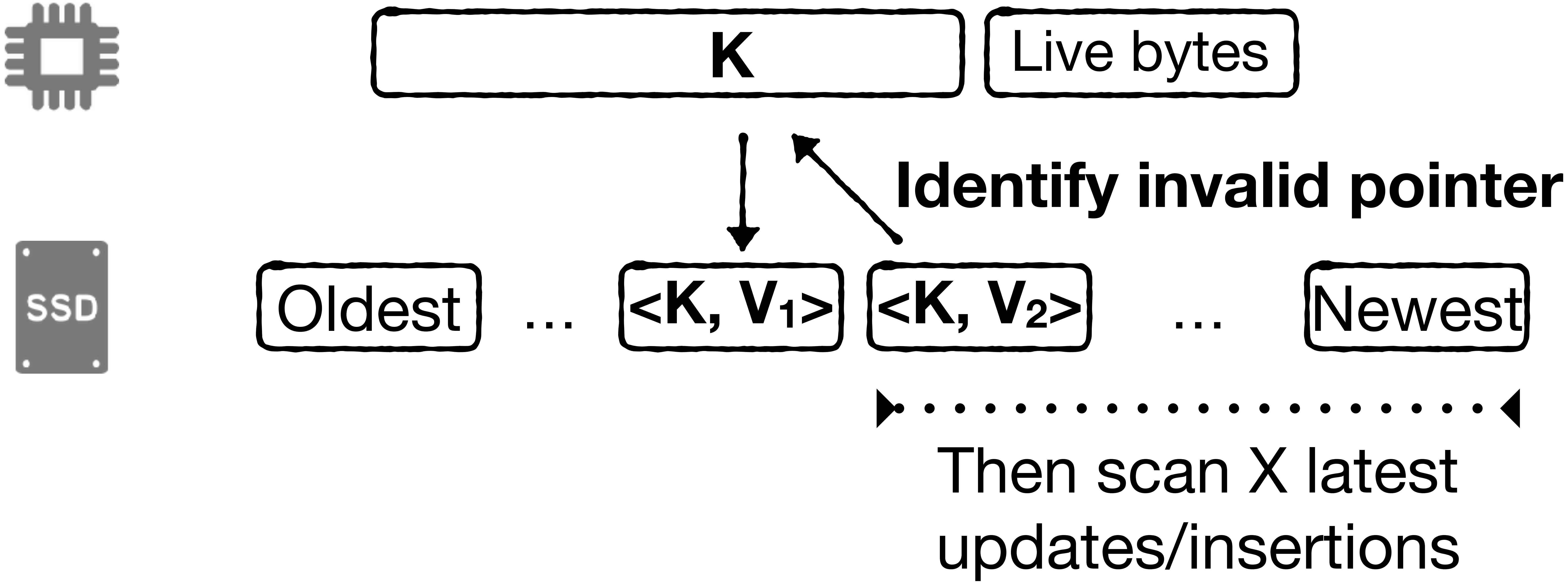


Then scan X latest updates/insertions

Checkpointing

Every X updates/insertions, store copy of index & live bytes

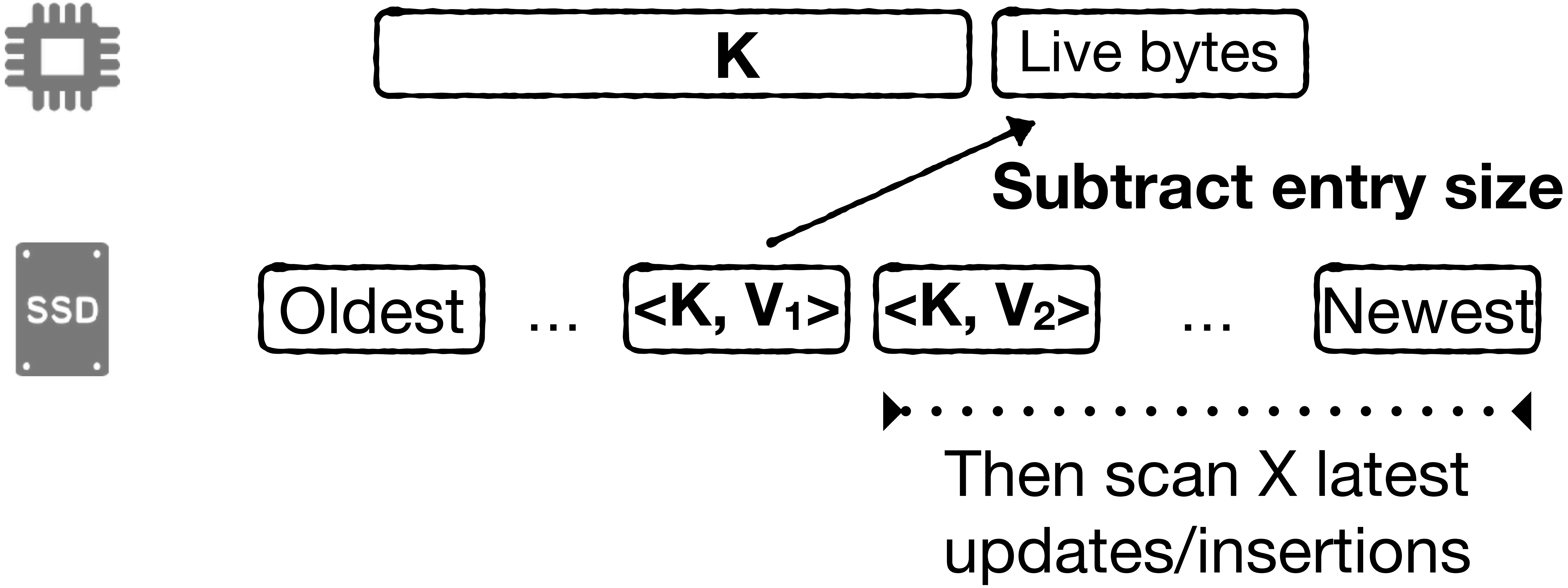
After recovery, load copy



Checkpointing

Every X updates/insertions, store copy of index & live bytes

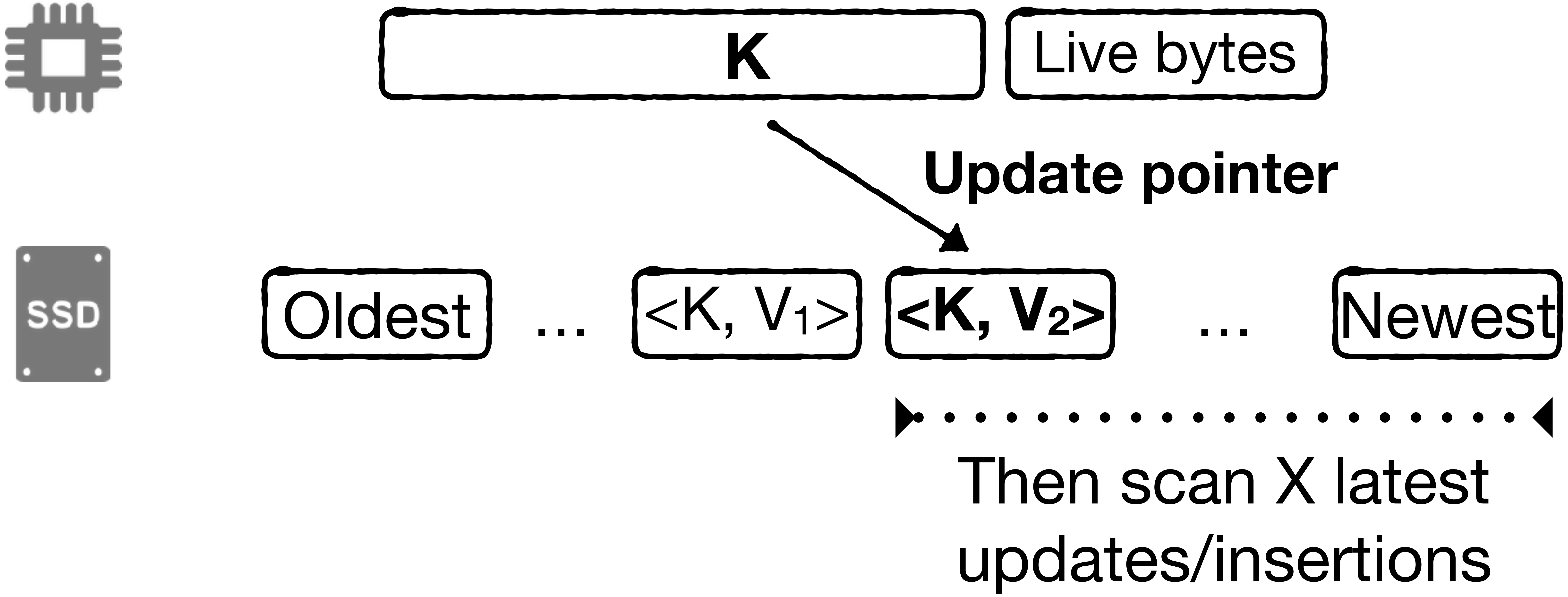
After recovery, load copy



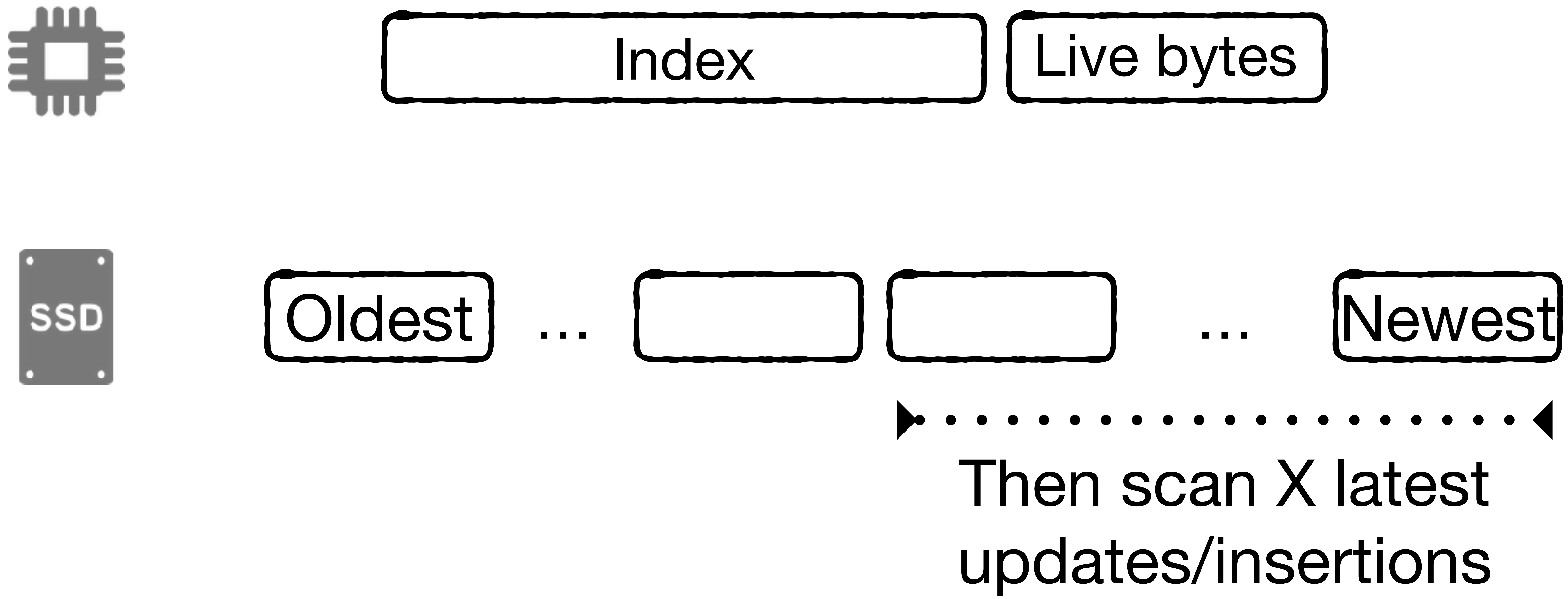
Checkpointing

Every X updates/insertions, store copy of index & live bytes

After recovery, load copy

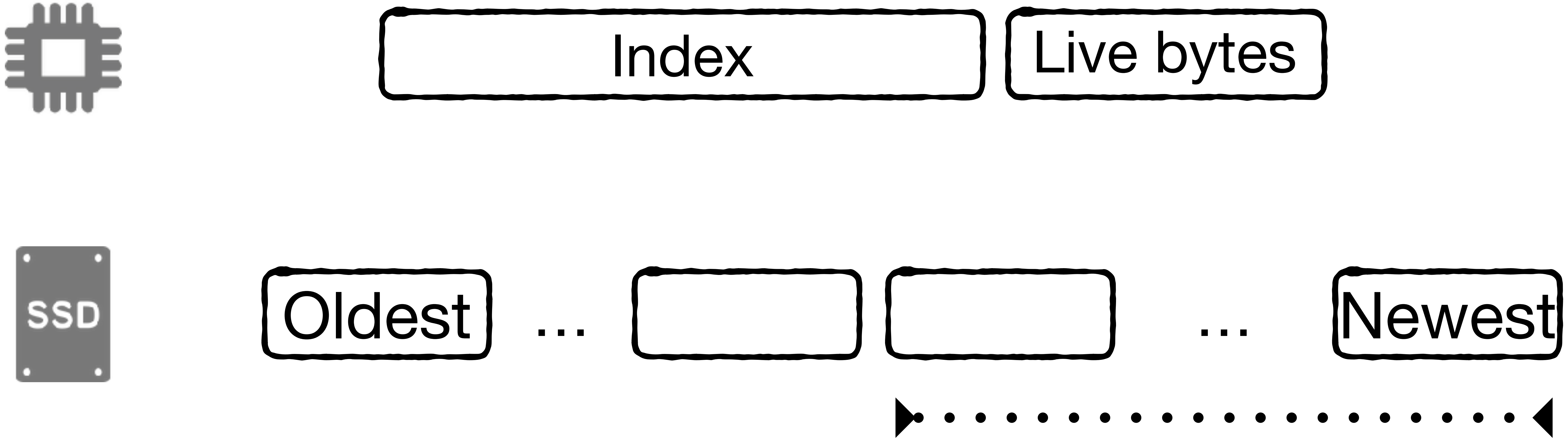


The frequency of checkpointing controls a trade-off between write cost and recovery time.



Additional write cost: O(index size / X)

Recovery time: O(X/B) reads for backwards scan



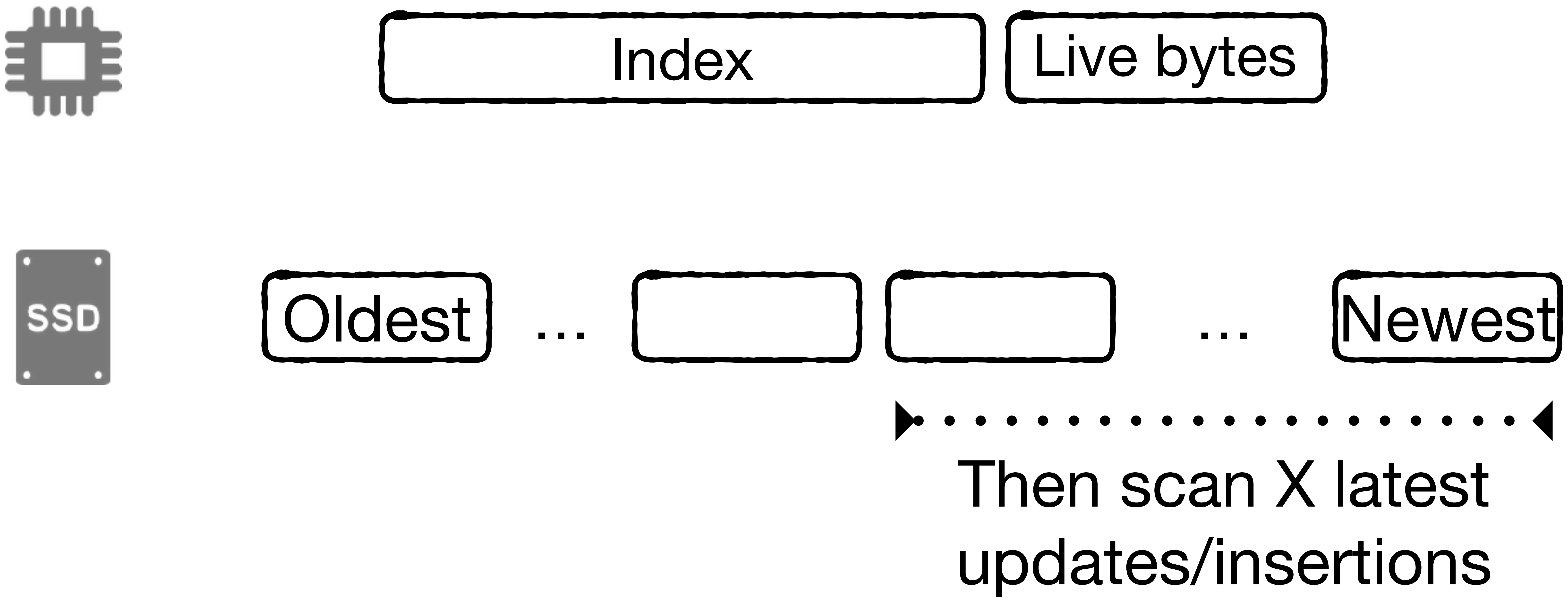
Then scan X latest updates/insertions

Costs Summary

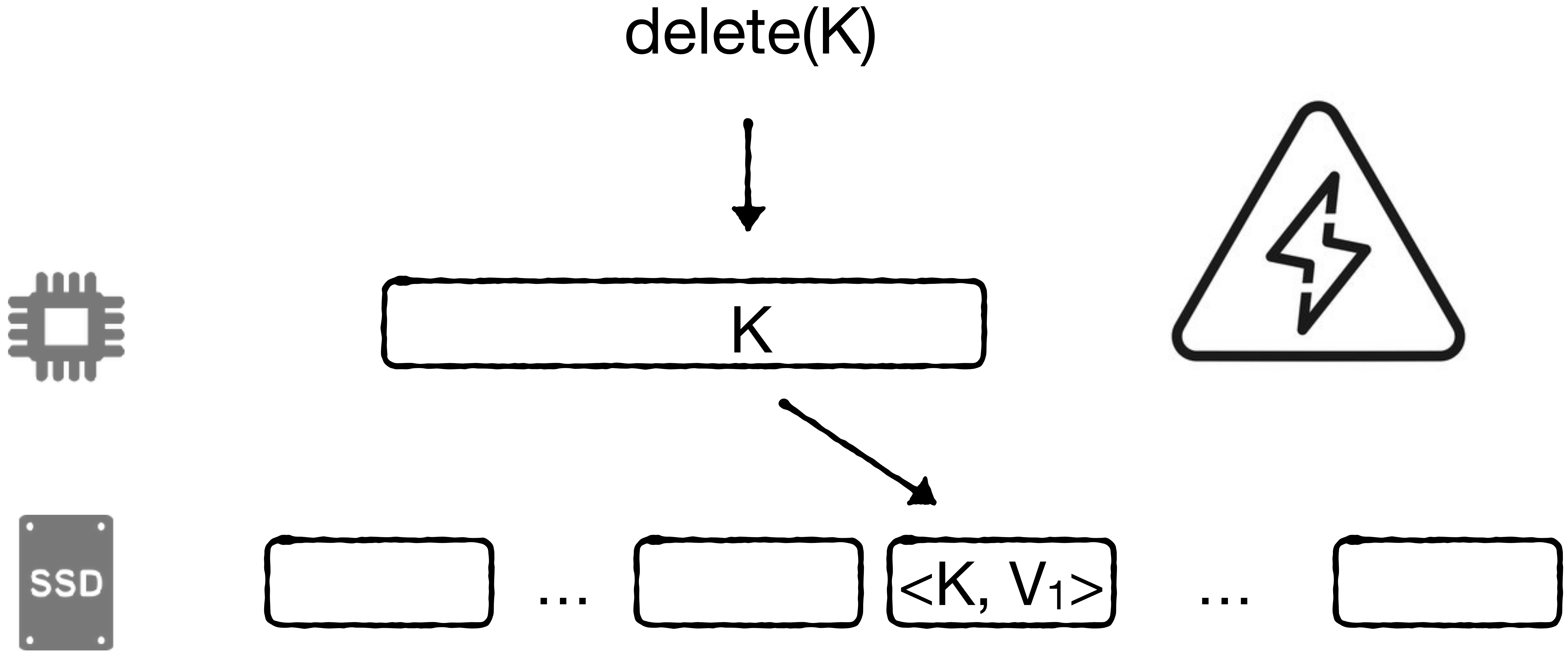
write cost: O(GC/B + index size / X)

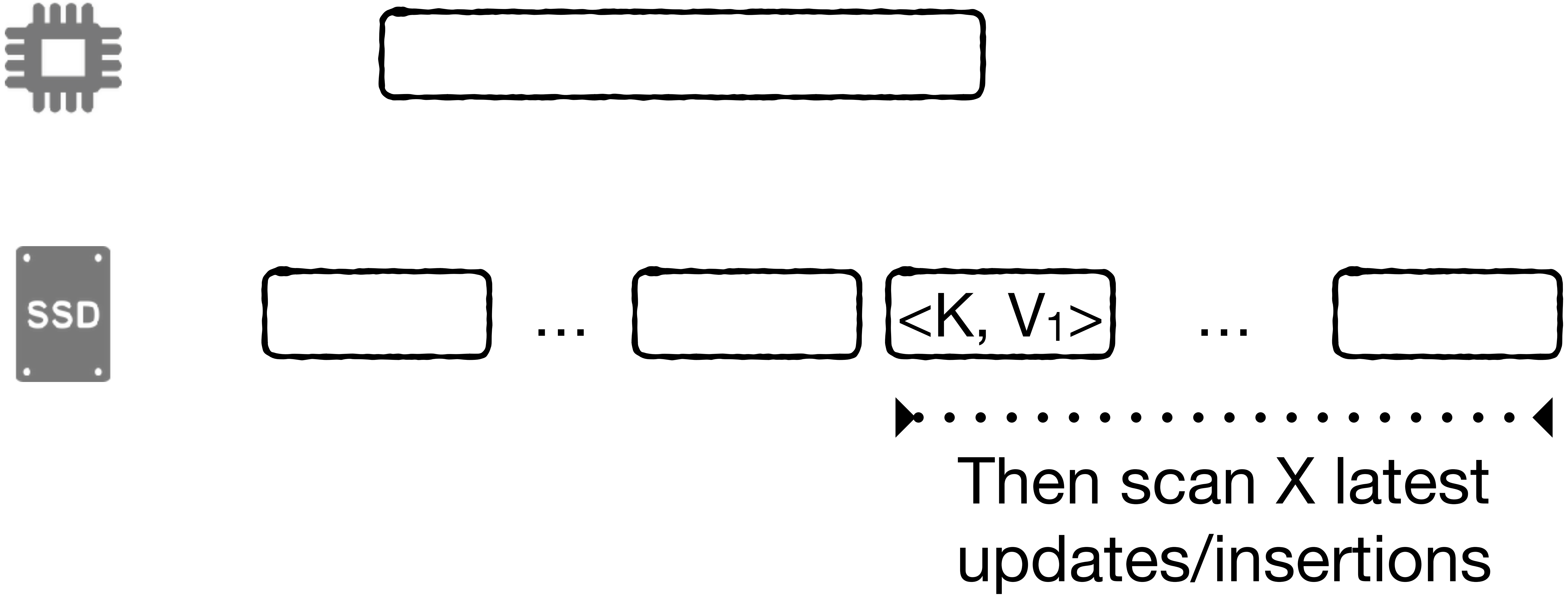
Recovery time: O(X/B) reads for backwards scan

Get cost: O(1)

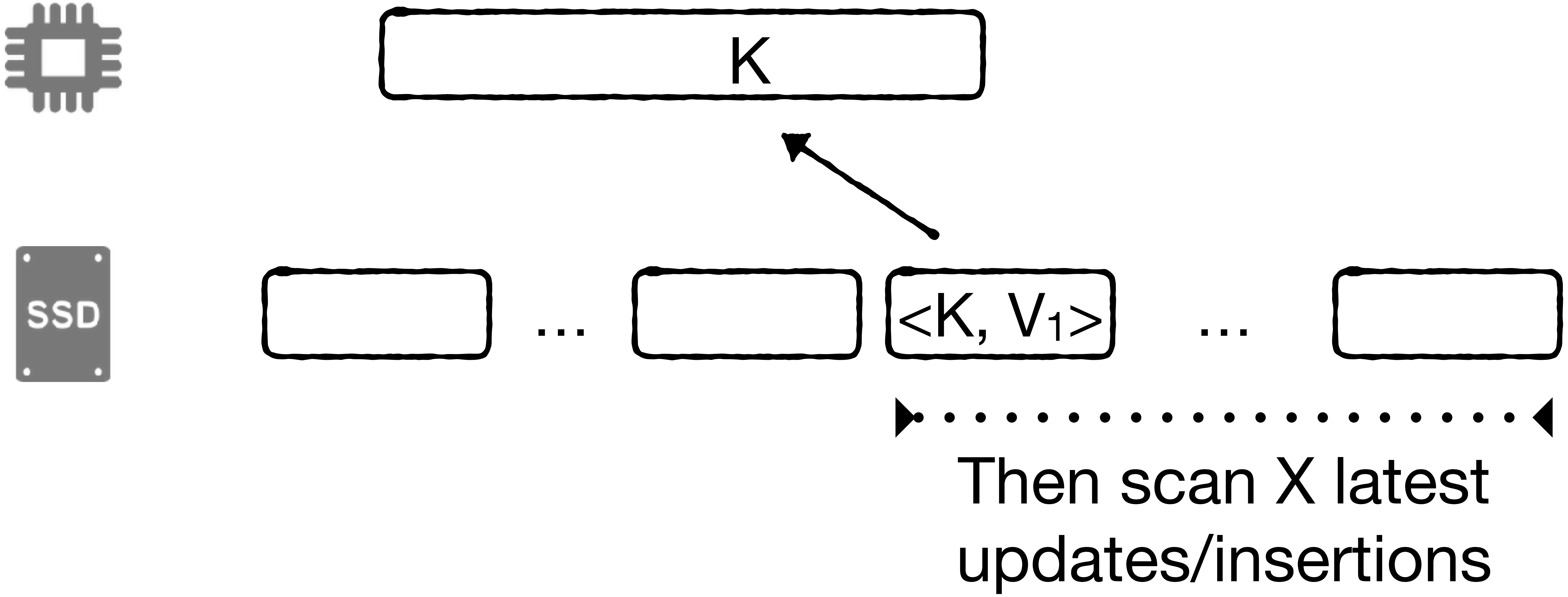


Suppose we delete an entry and then power fails. Can you spot a problem?



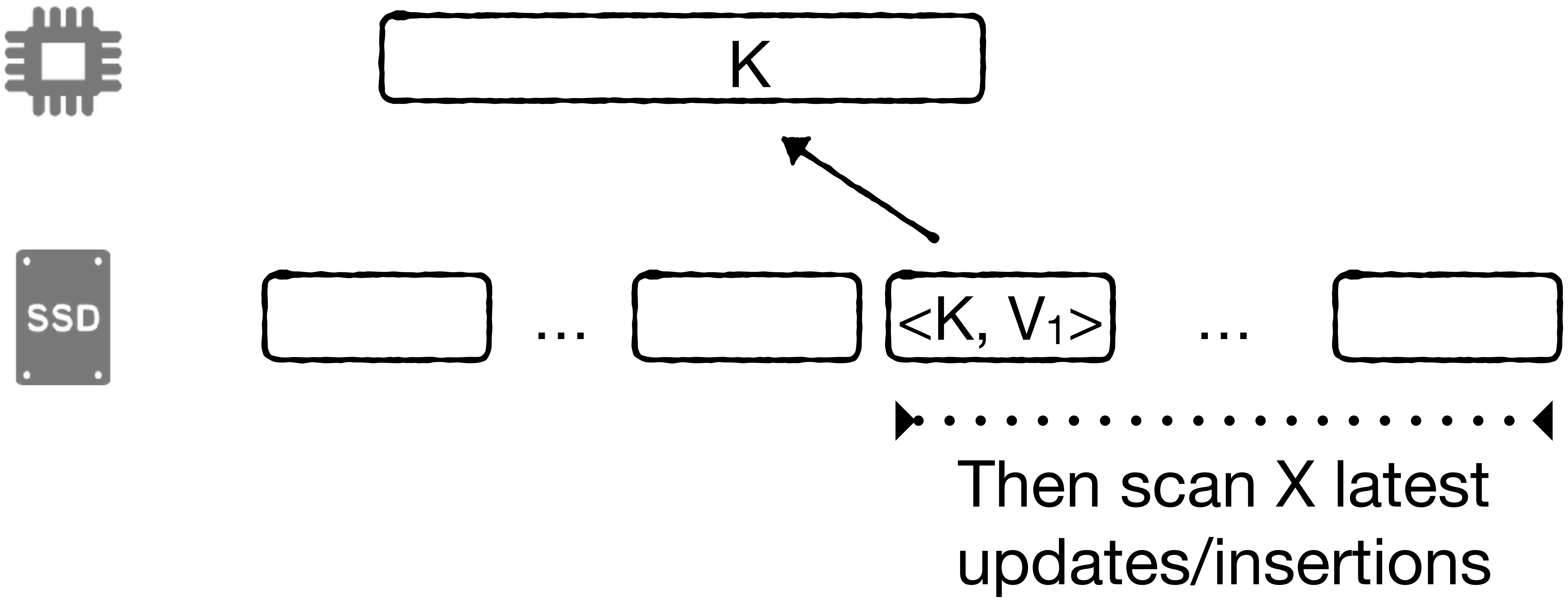

SSD
...
<K, V1>
...
Then scan X latest
updates/insertions

May load deleted entry back to index during recovery

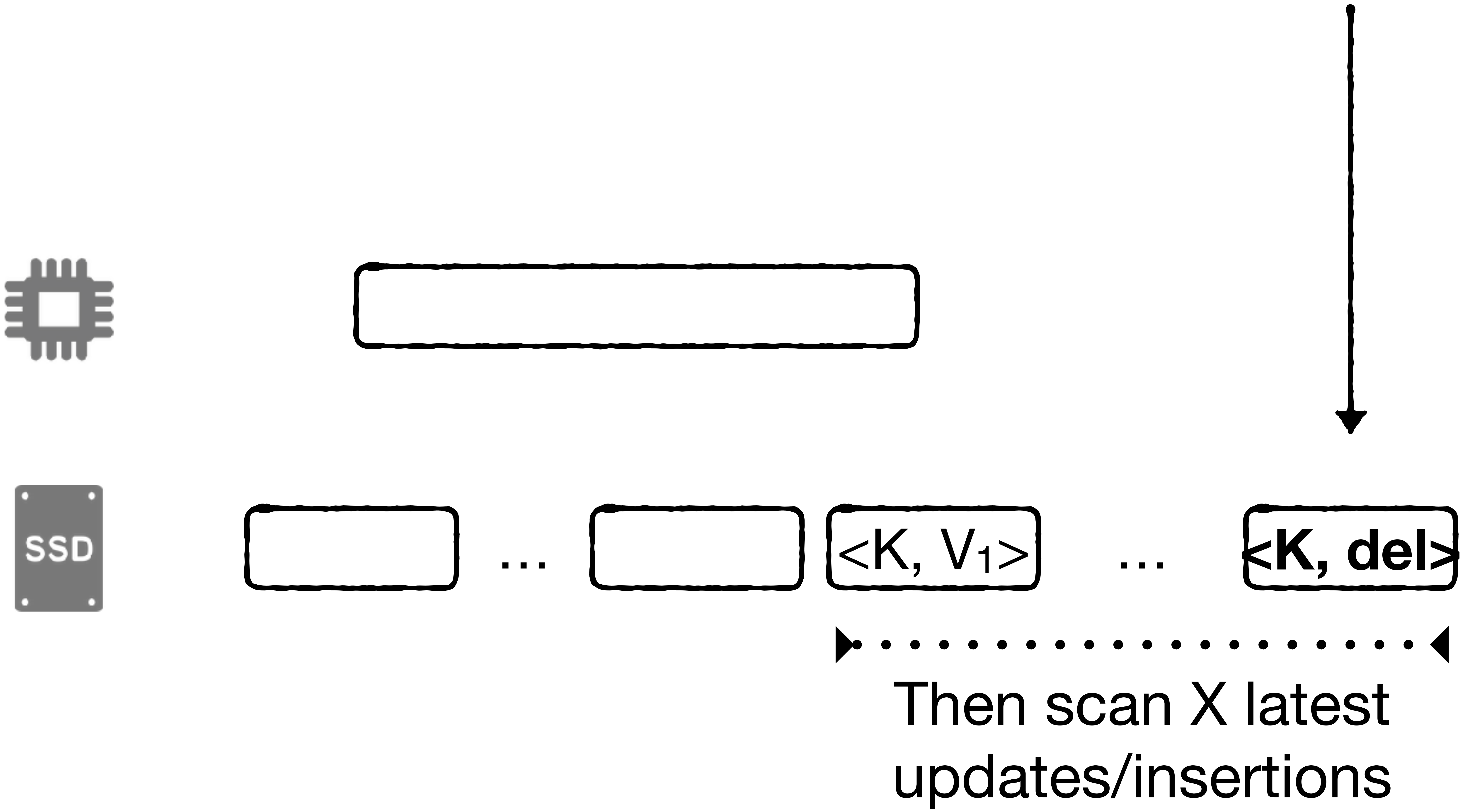


May load deleted entry back to index during recovery

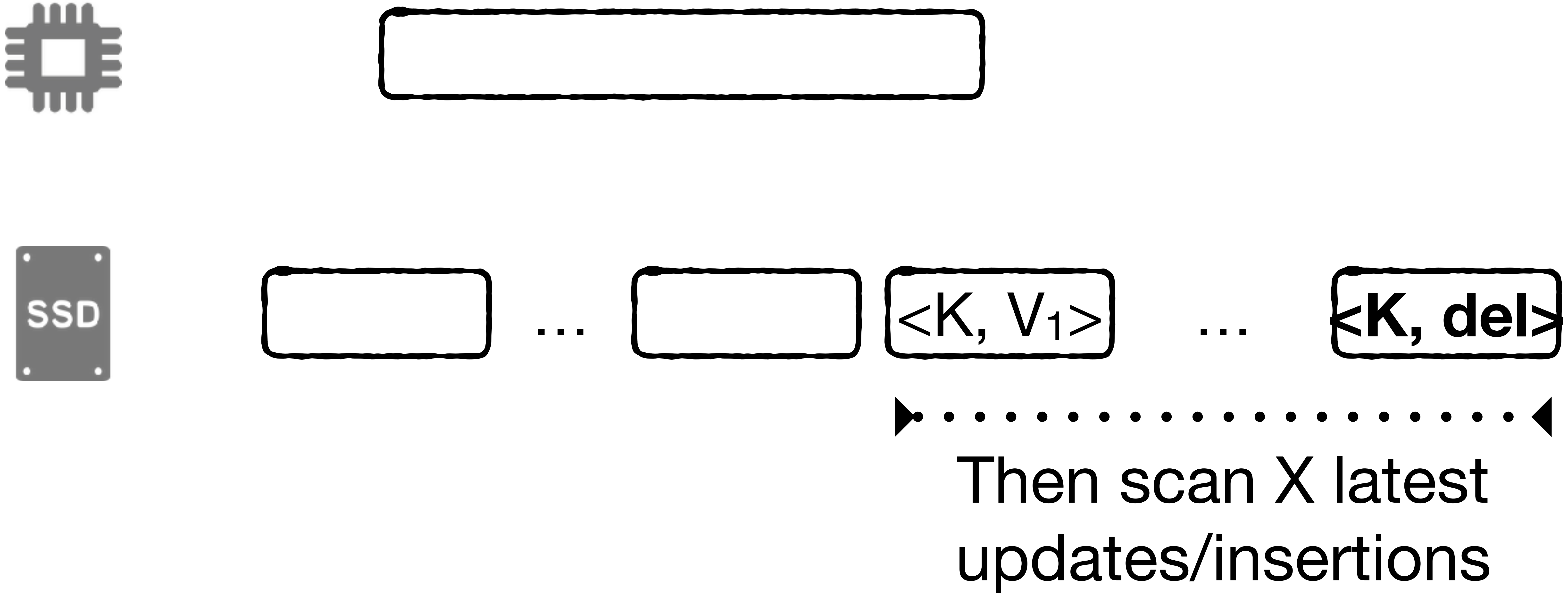
Solution?



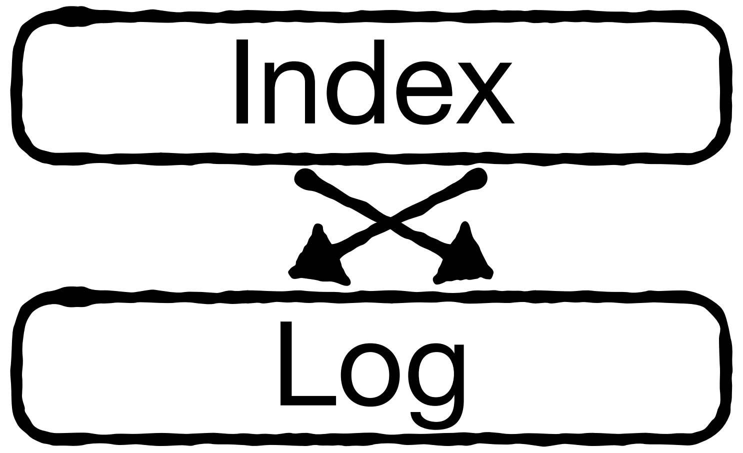
Insert tombstone when deleting



While recovering, if the first instance of a key we see is a tombstone, we ignore all subsequent entries with this key.



Mechanics



Hot/Cold separation

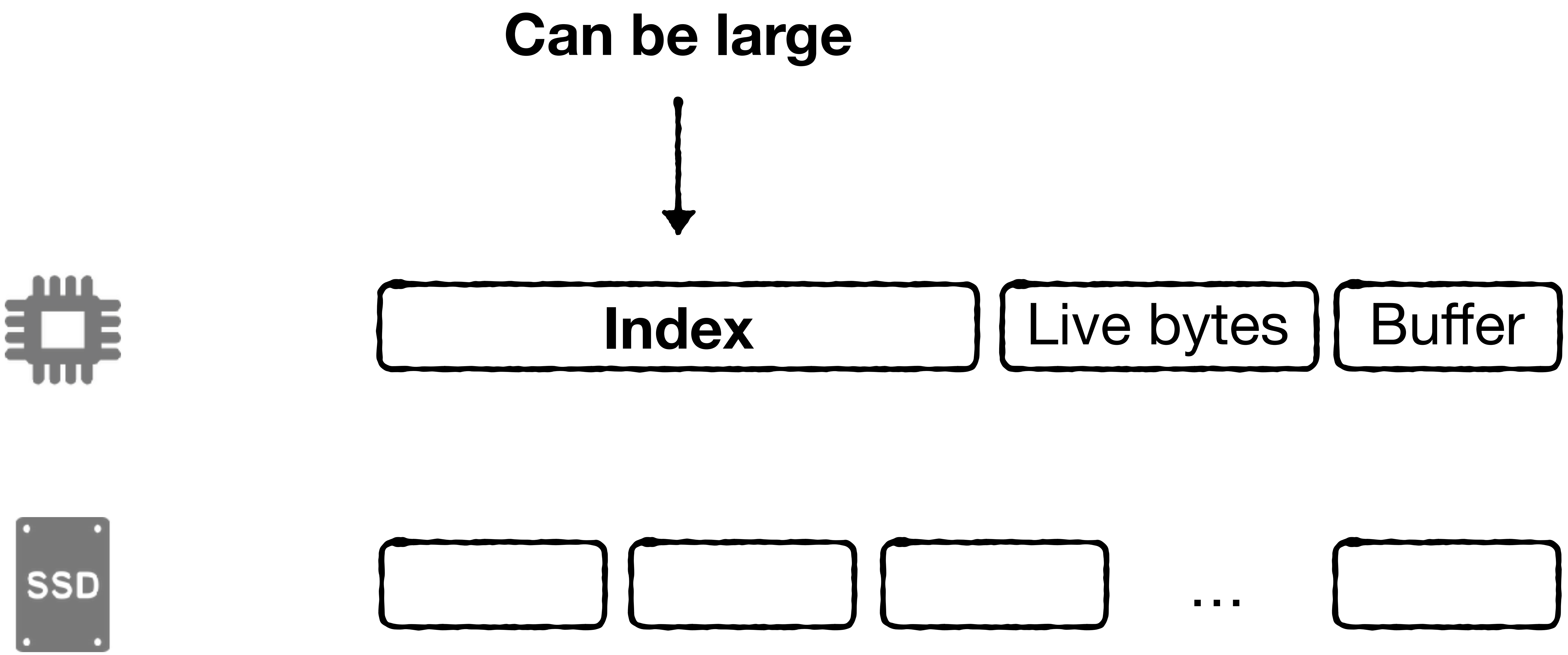
To reduce write-amp

Checkpointing & Recovery

If power fails

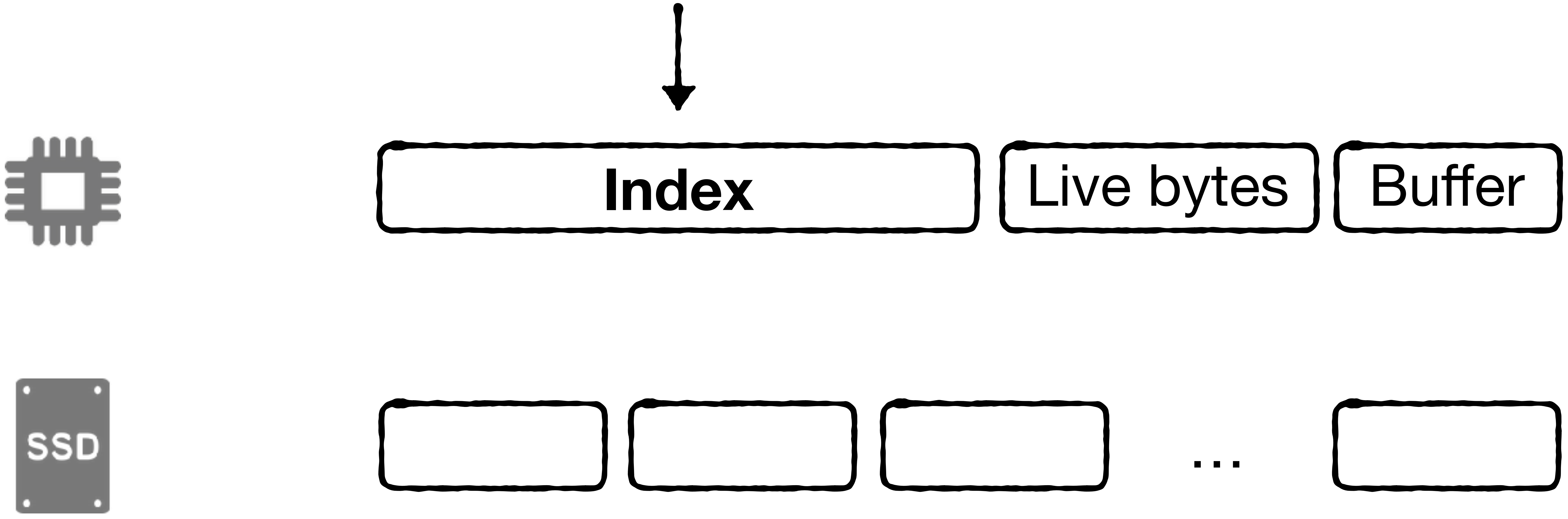
Cuckoo filtering

To reduce memory


Can be large
Index
Live bytes
Buffer
SSD
...

Index size = $N \cdot (P + K) / \alpha$

N = data size
P = pointer size
K = key size
α = collision resolution overheads



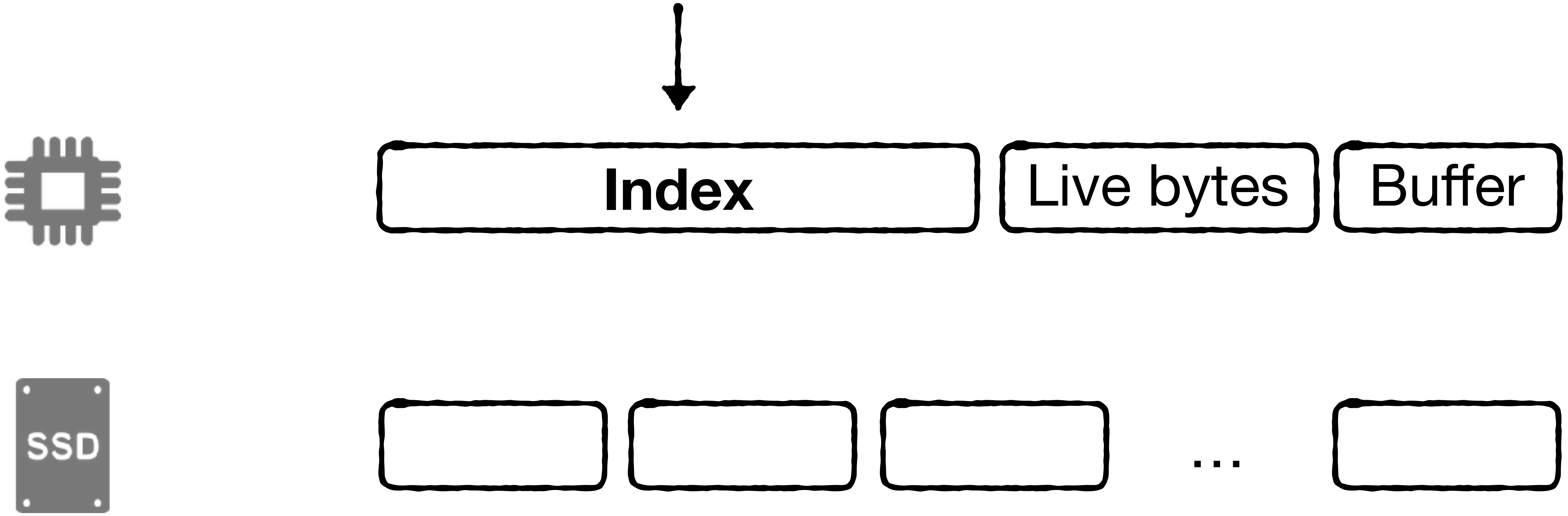
Index size = $N \cdot (P + K) / \alpha$

N = data size

P = pointer size **= $O(\log_2 N/B)$**

K = key size **= $\Omega(\log_2 N)$**

α = collision resolution overheads **≈ 0.8**



Index size = $N \cdot (P + K) / \alpha$

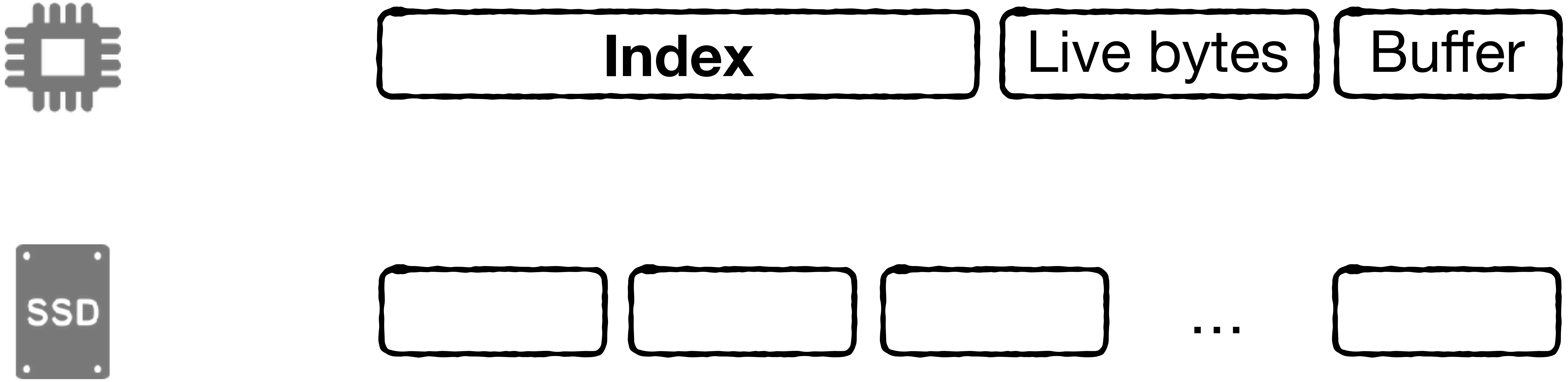
N = data size

P = pointer size = $O(\log_2 N/B)$

K = key size = $\Omega(\log_2 N)$

α = collision resolution overheads ≈ 0.8

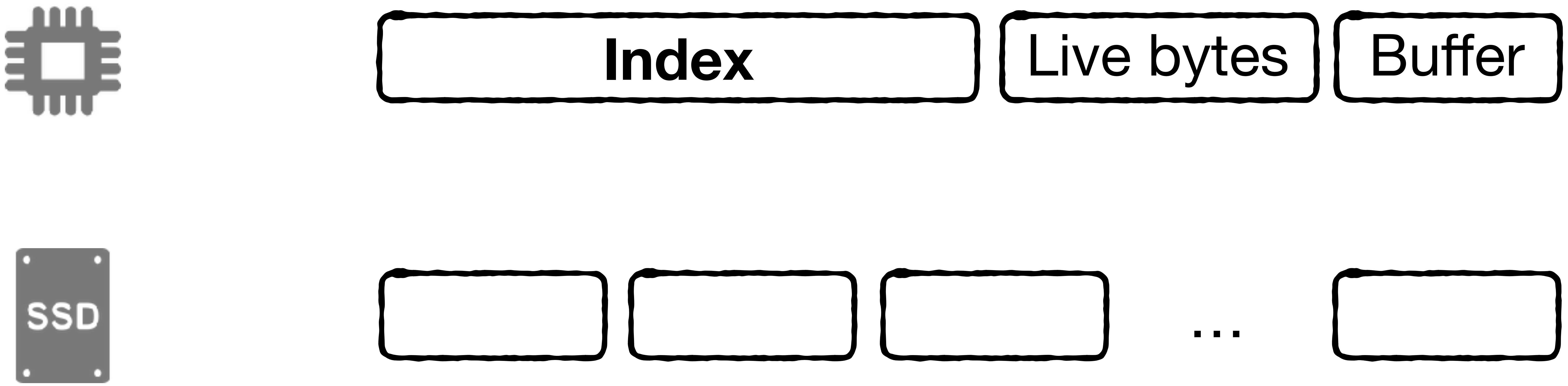
Ultimately attack this with cuckoo filters →



Index size = $N \cdot (P + K) / \alpha$

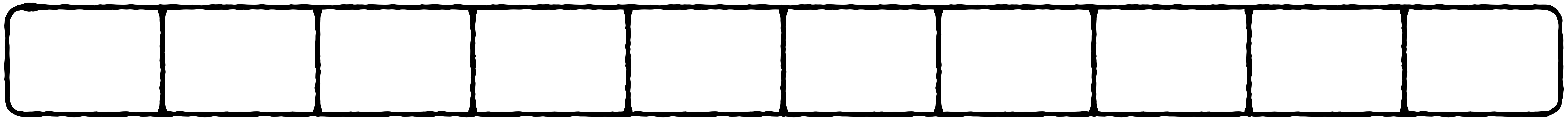
N = data size
P = pointer size = $O(\log_2 N/B)$
K = key size = $\Omega(\log_2 N)$
α = collision resolution overheads ≈ 0.8

But first attack this with cuckoo hashing →



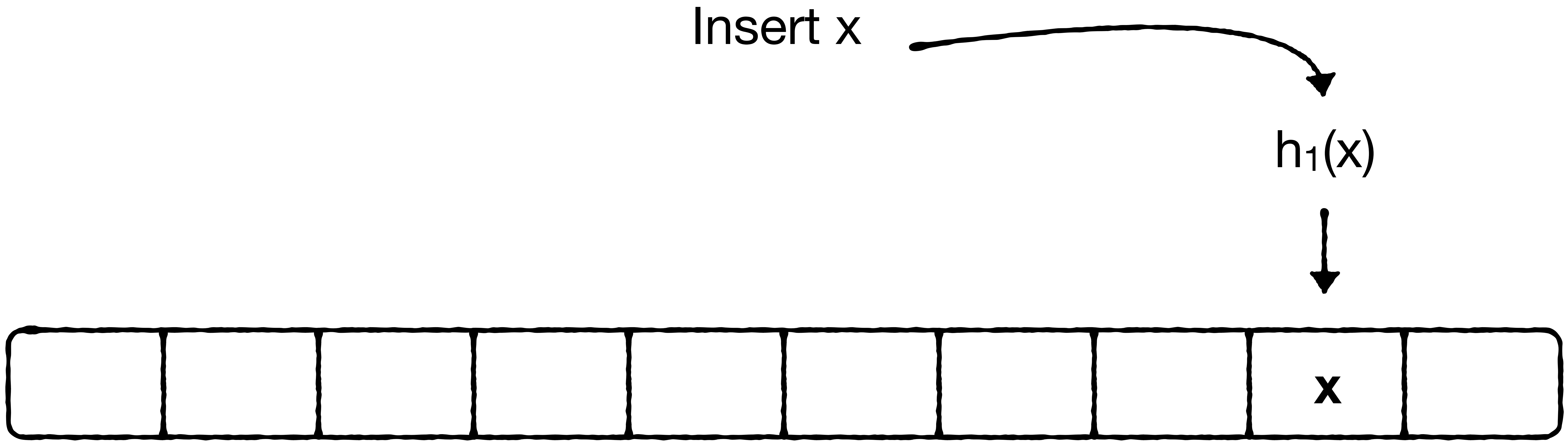
Cuckoo Hashing

Table of buckets, each containing 1 entry
Two hash functions



Cuckoo Hashing

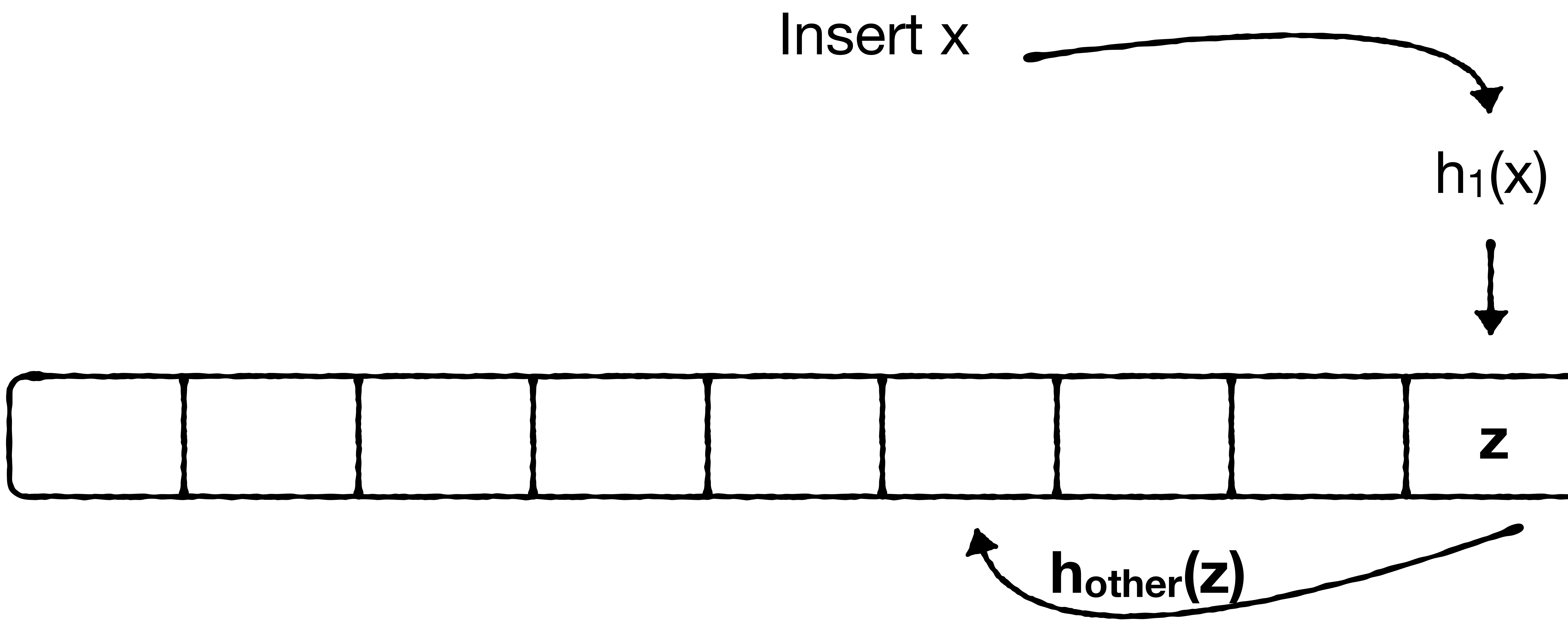
Table of buckets, each containing 1 entry
Two hash functions



Insert if there is free space

Cuckoo Hashing

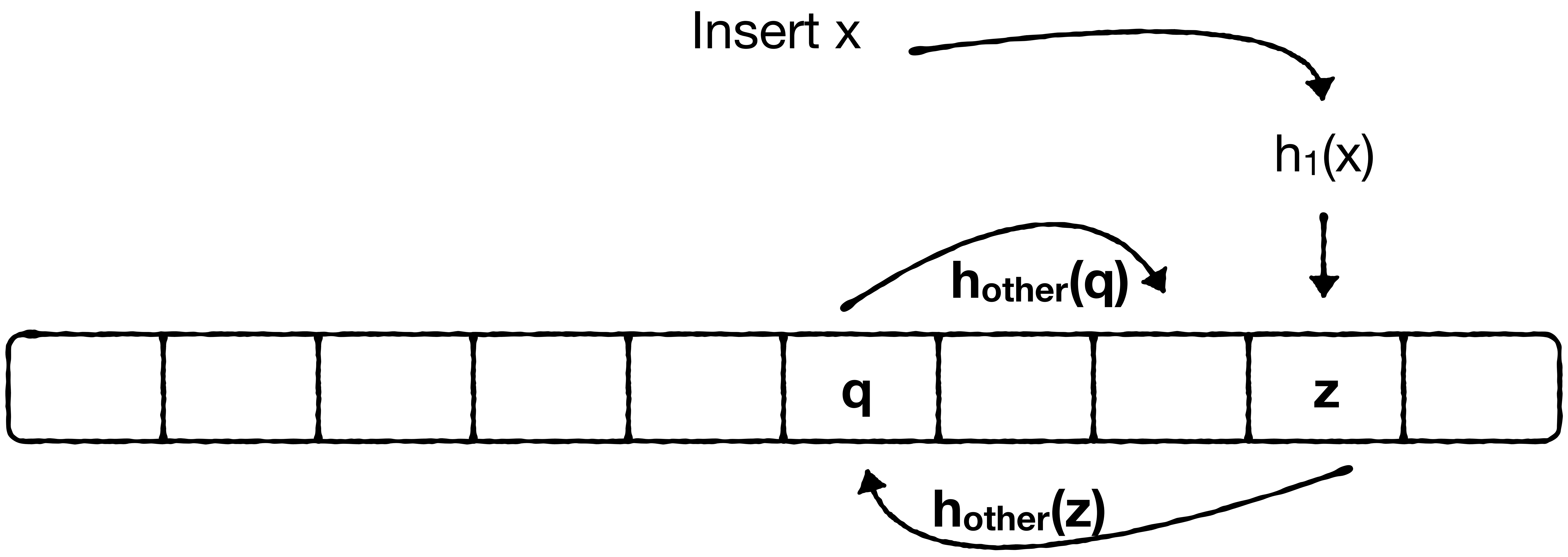
Table of buckets, each containing 1 entry
Two hash functions



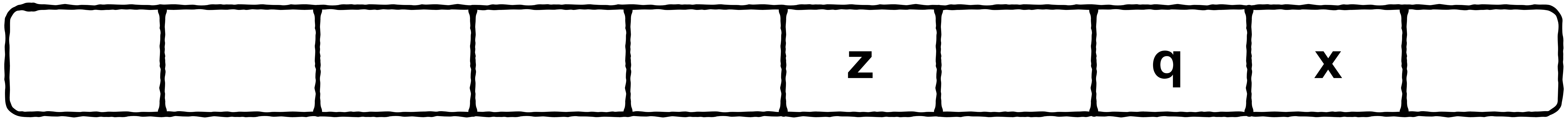
Otherwise, evict existing entry using its alternative hash function to an alternative slot

Cuckoo Hashing

Table of buckets, each containing 1 entry
Two hash functions

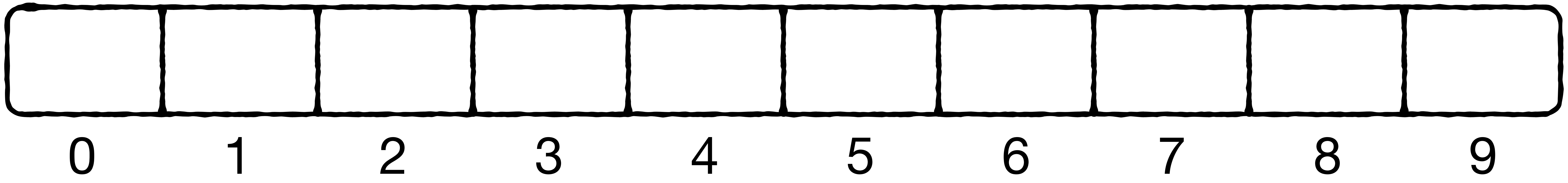


Continue doing this recursively until all entries are mapped to an empty bucket



Continue doing this recursively until all entries are mapped to an empty bucket

z	$h_1(z) = 7$	$h_2(z) = 2$
q	$h_1(q) = 7$	$h_2(q) = 5$
x	$h_1(x) = 2$	$h_2(x) = 9$



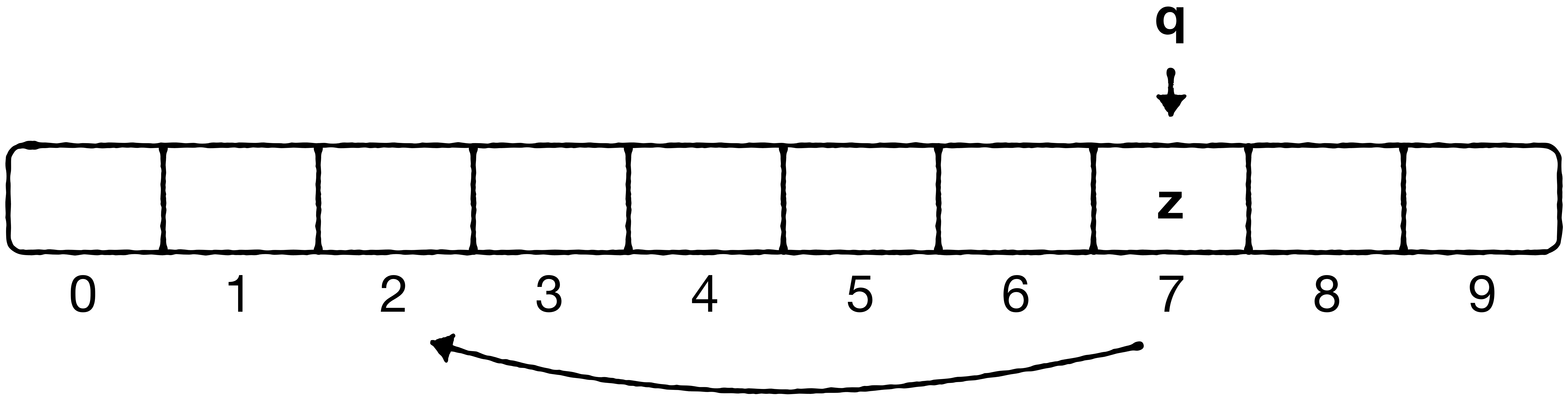
	$h_1(z) = 7$	$h_2(z) = 2$
q	$h_1(q) = 7$	$h_2(q) = 5$
x	$h_1(x) = 2$	$h_2(x) = 9$

							z		
0	1	2	3	4	5	6	7	8	9

$h_1(z) = 7$ $h_2(z) = 2$

$h_1(q) = 7$ $h_2(q) = 5$

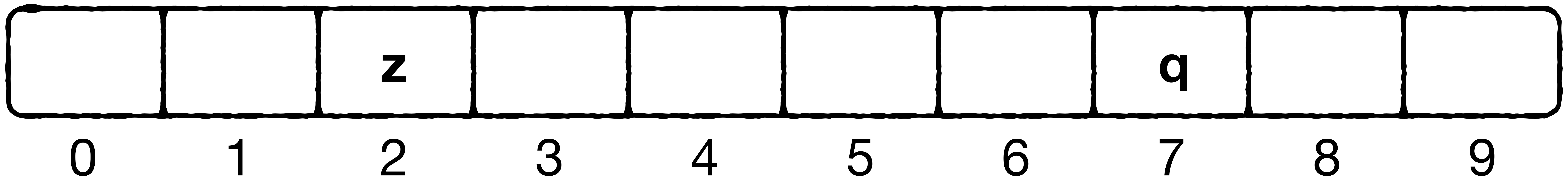
x $h_1(x) = 2$ $h_2(x) = 9$



$h_1(z) = 7$ $\quad$ $h_2(z) = 2$

$h_1(q) = 7$ $\quad$ $h_2(q) = 5$

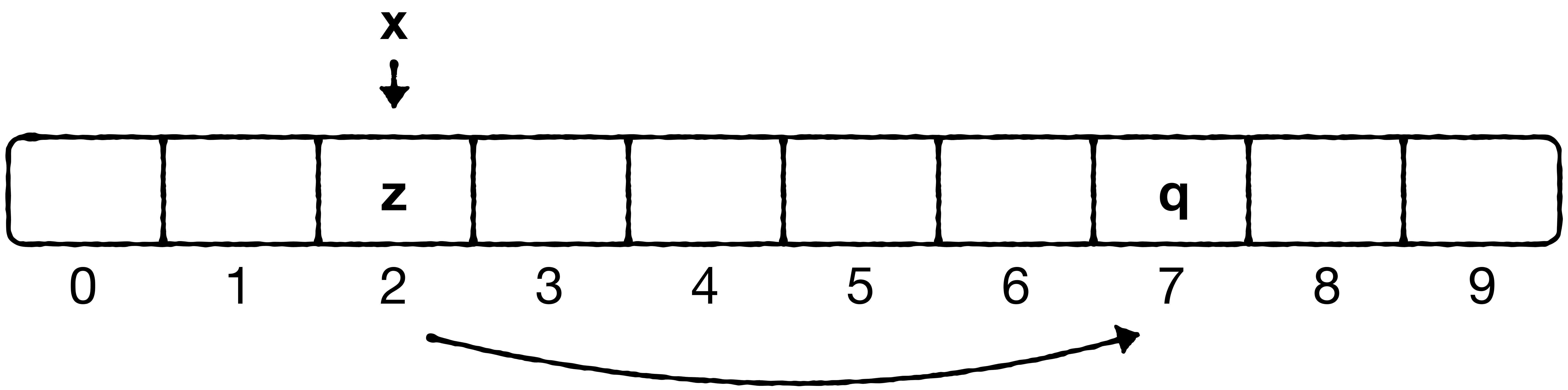
x $\quad$ $h_1(x) = 2$ $\quad$ $h_2(x) = 9$



$h_1(z) = 7$ $h_2(z) = 2$

$h_1(q) = 7$ $h_2(q) = 5$

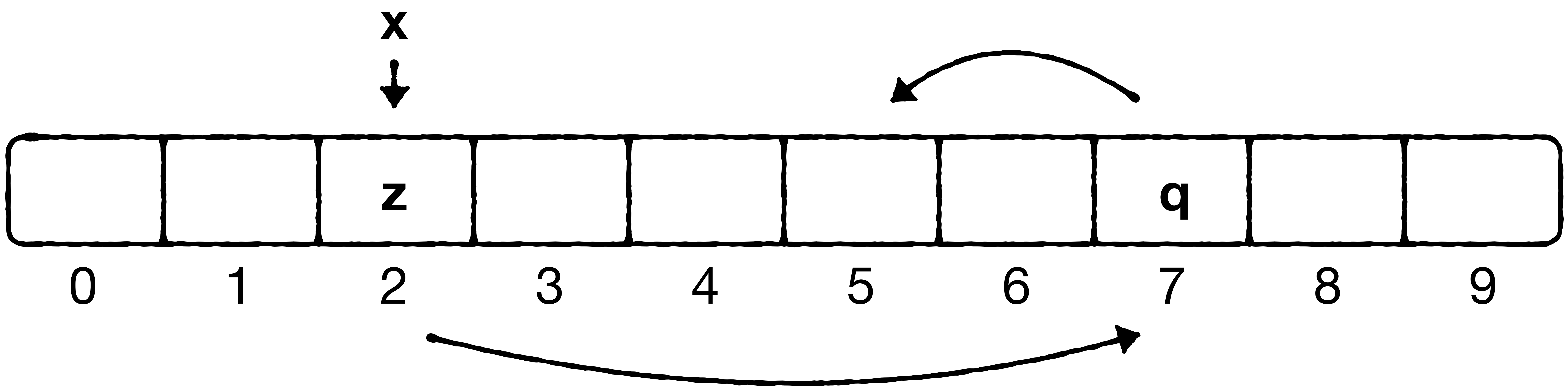
$h_1(x) = 2$ $h_2(x) = 9$



$h_1(z) = 7$ $\quad$ $h_2(z) = 2$

$h_1(q) = 7$ $\quad$ $h_2(q) = 5$

$h_1(x) = 2$ $\quad$ $h_2(x) = 9$



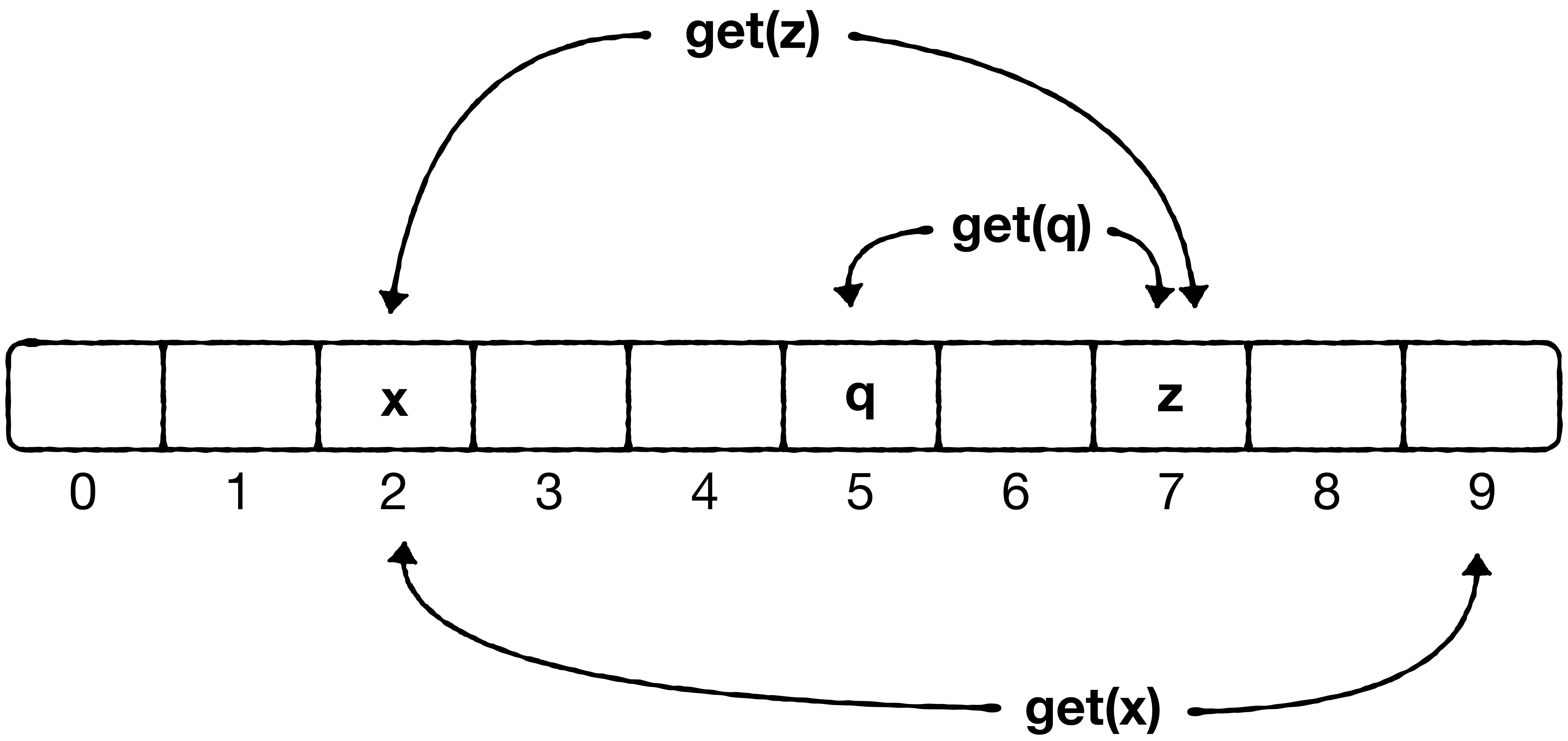
$h_1(z) = 7$ $h_2(z) = 2$

$h_1(q) = 7$ $h_2(q) = 5$

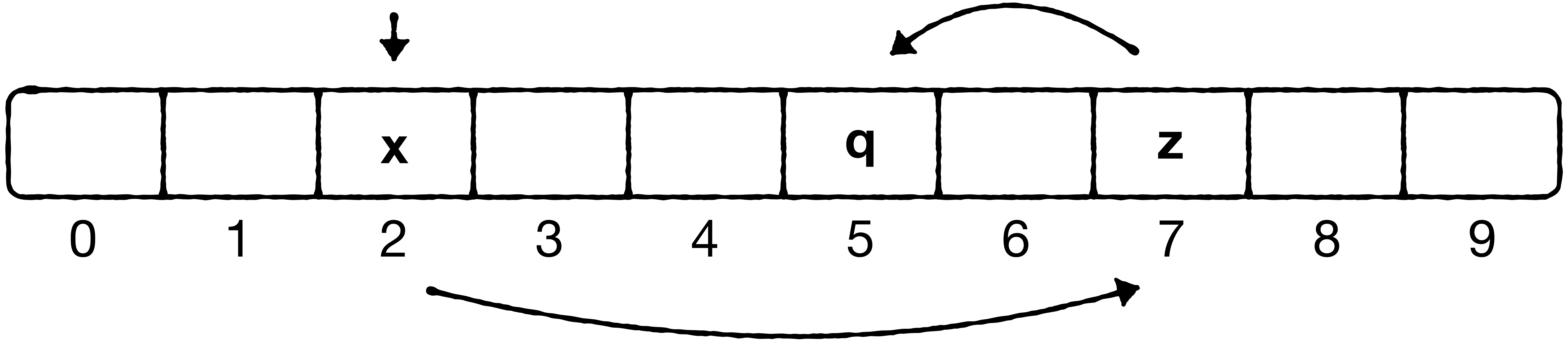
$h_1(x) = 2$ $h_2(x) = 9$

		x			**q**		**z**		
0	1	2	3	4	5	6	7	8	9

A query has to check at most two locations to find a key



Insertions may endure multiple evictions and swapping of keys across buckets



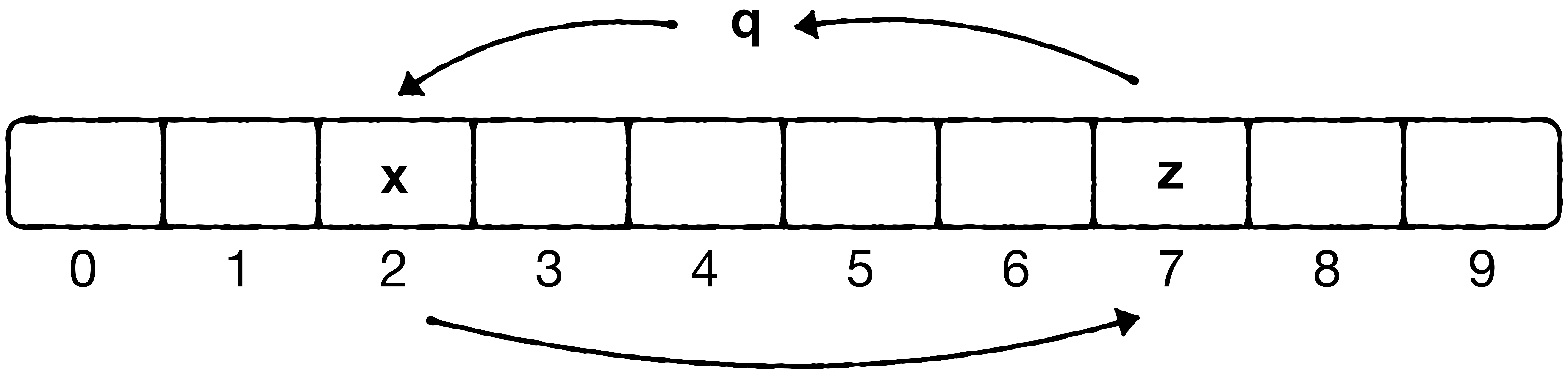
Insertions may endure multiple evictions and swapping of keys across buckets

Worse: an infinite loop is technically possible

$h_1(z) = 7$ $h_2(z) = 2$

$h_1(q) = 2$ $h_2(q) = 7$

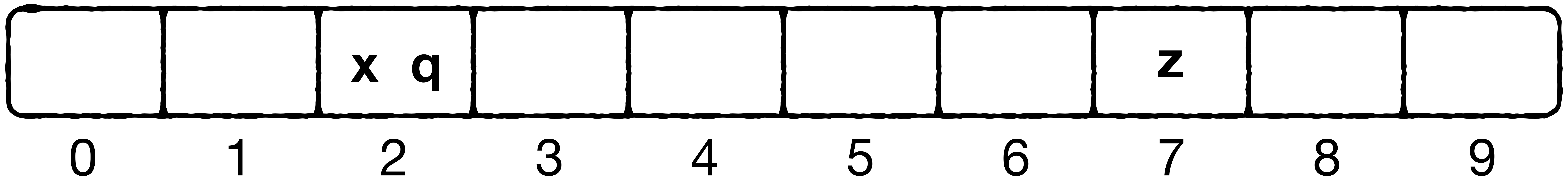
$h_1(x) = 7$ $h_2(x) = 2$



Insertions may endure multiple evictions and swapping of keys across buckets

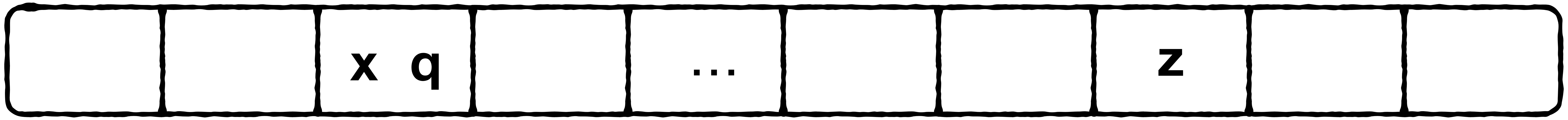
Worse: an infinite loop is technically possible

alleviate by allowing multiple keys per bucket



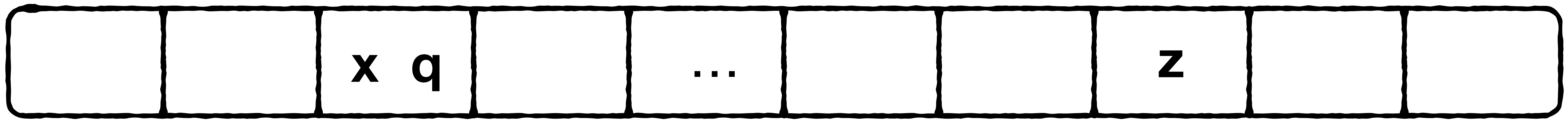
Theory: insertions succeed in O(1) expected time with high probability

Assuming…



Theory: insertions succeed in O(1) expected time with high probability

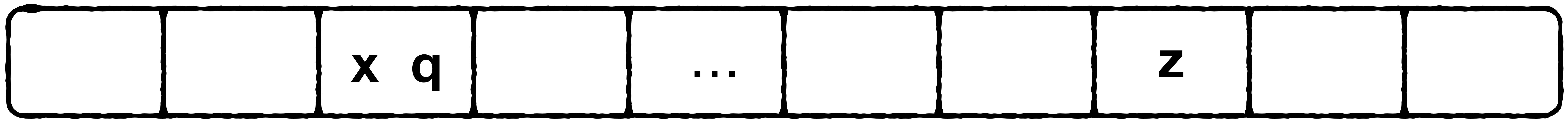
Assuming **load factor < 50% for bucket size 1**



Theory: insertions succeed in O(1) expected time with high probability

Assuming **load factor < 50% for bucket size 1**

load factor < 84% for bucket size 2

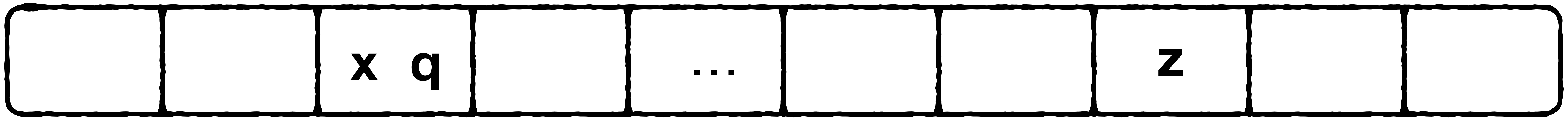


Theory: insertions succeed in O(1) expected time with high probability

Assuming **load factor < 50% for bucket size 1**

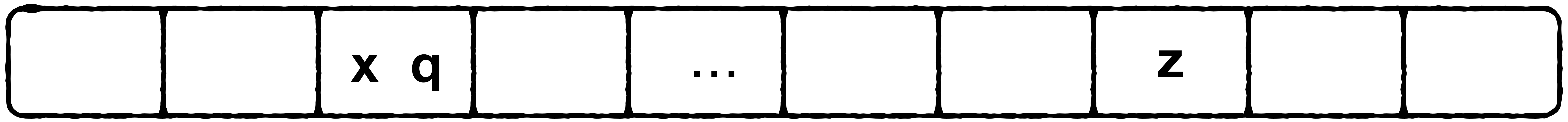
load factor < 84% for bucket size 2

load factor < 95% for bucket size 4

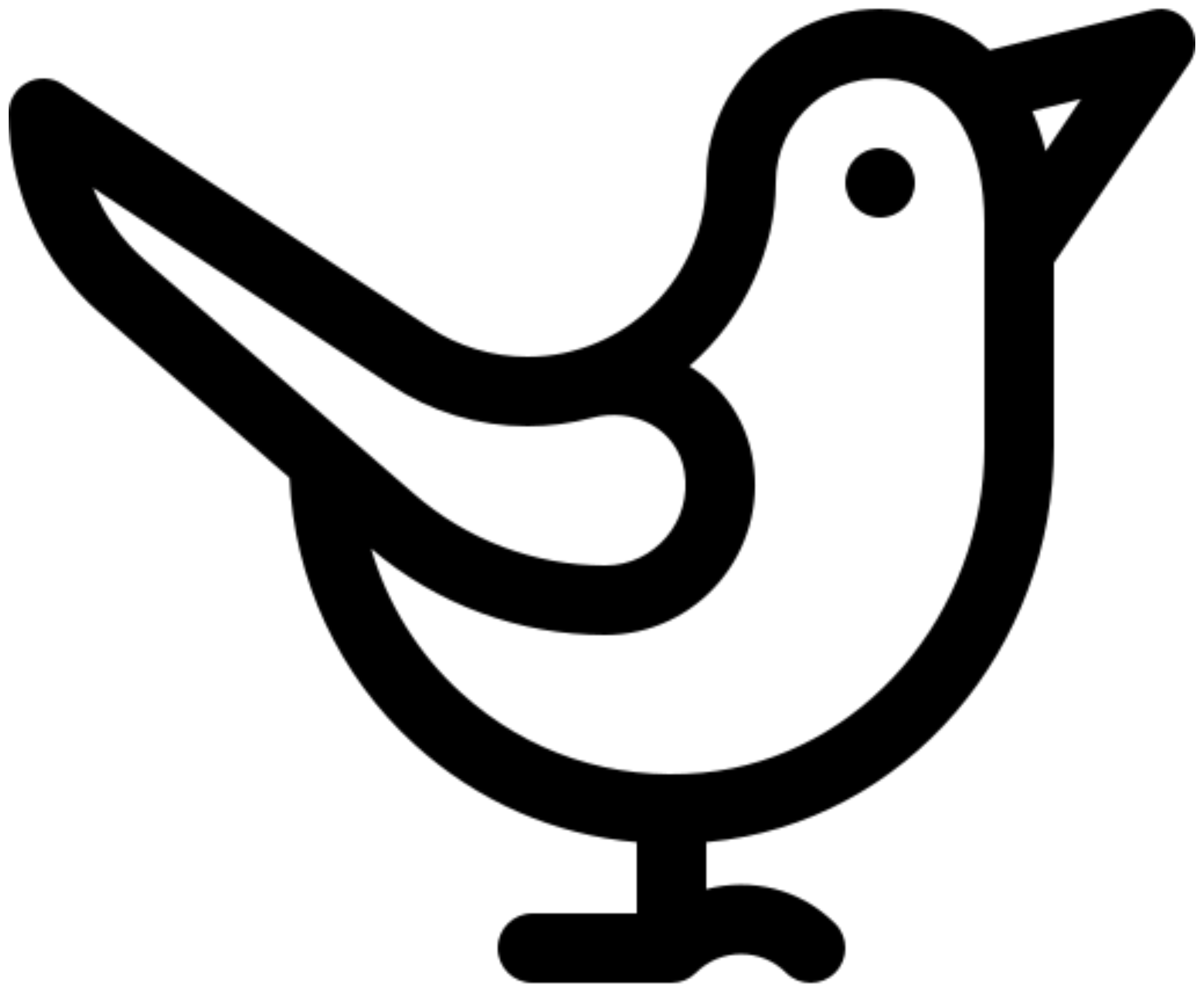


Theory: insertions succeed in O(1) expected time with high probability

We'll use this → load factor < 95% for bucket size 4



Why is it called Cuckoo hashing?



Lay eggs in other birds’ nests, and the hatchlings “evict” the other birds’ eggs.

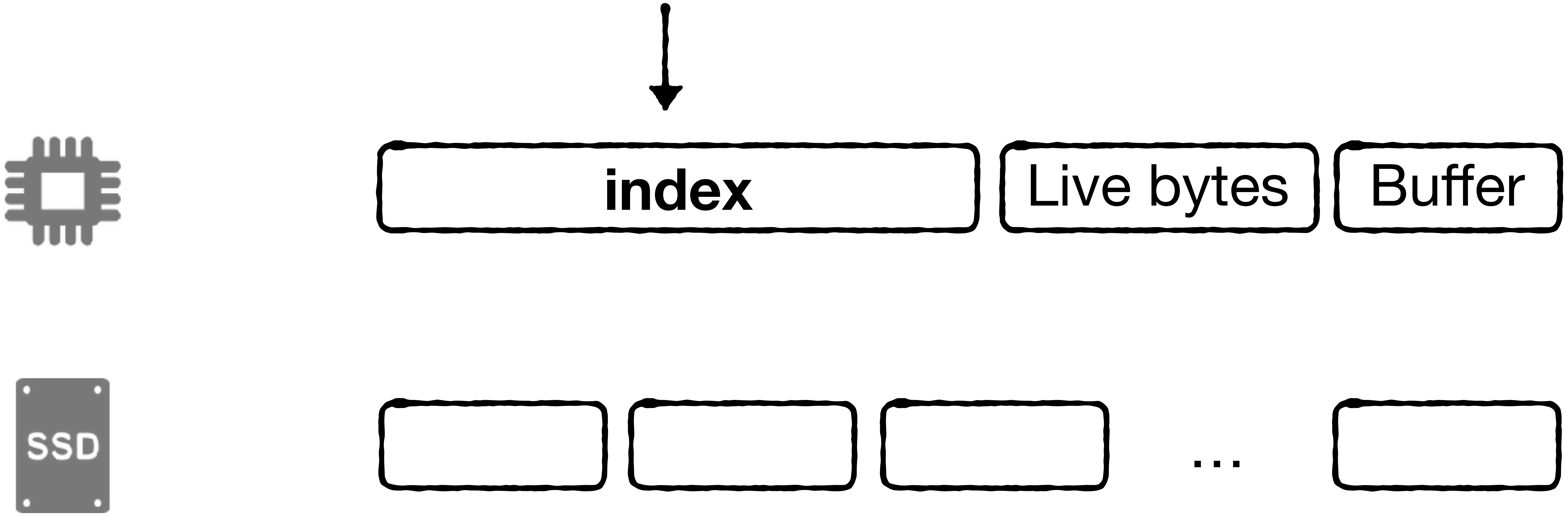
Index size = N * (P + K) / α

N = data size

P = pointer size

K = key size

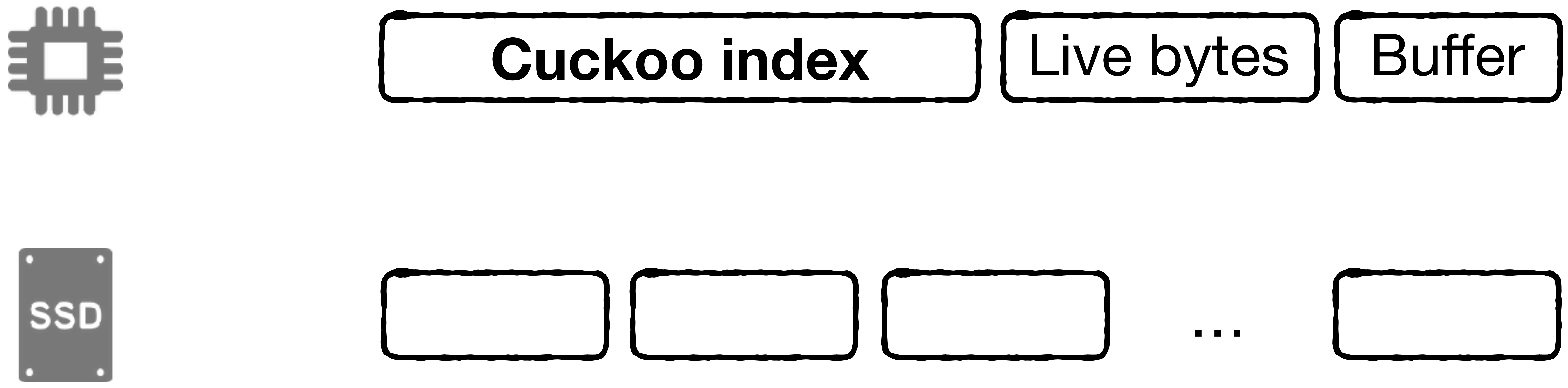
α = collision resolution overheads



Index size = N * (P + K) / α

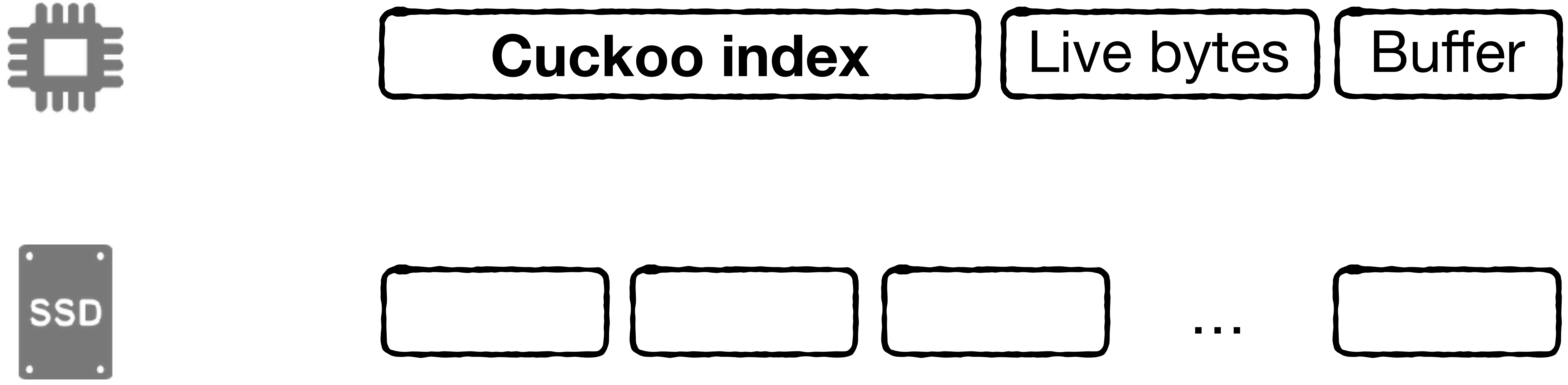
N = data size
P = pointer size
K = key size

Addressed with cuckoo hashing → **α = collision resolution overheads**



Index size = N * (P + K) / α

N = data size
P = pointer size
K = key size
Addressed with cuckoo hashing → **α = collision resolution overheads ≈0.8 → ≈0.95**



Index size = N * (P + K) / α

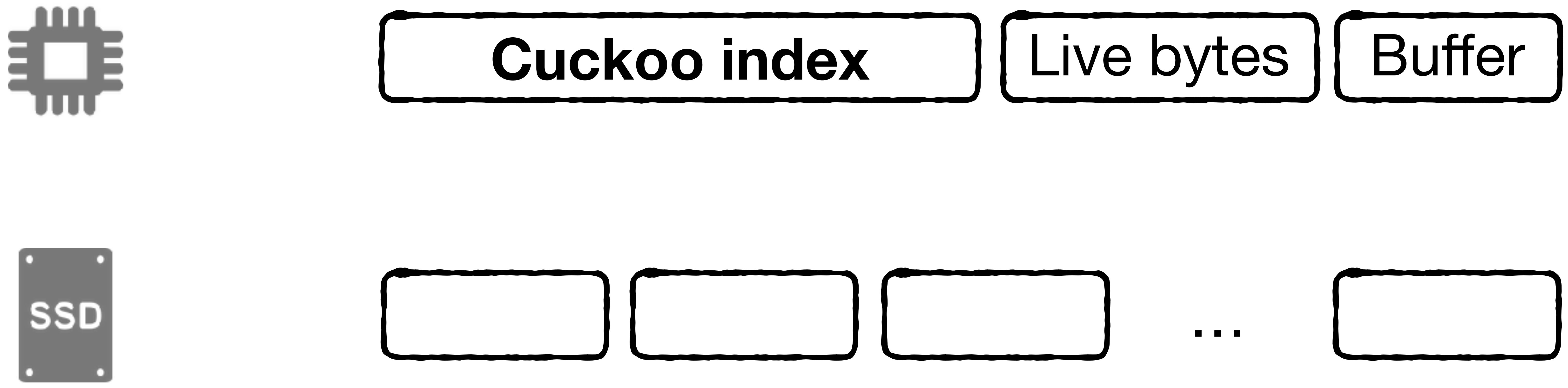
Now lets attack this With a Cuckoo Filter → **K** = **key size** = **$\Omega(\log_2 N)$**

N = data size

P = pointer size

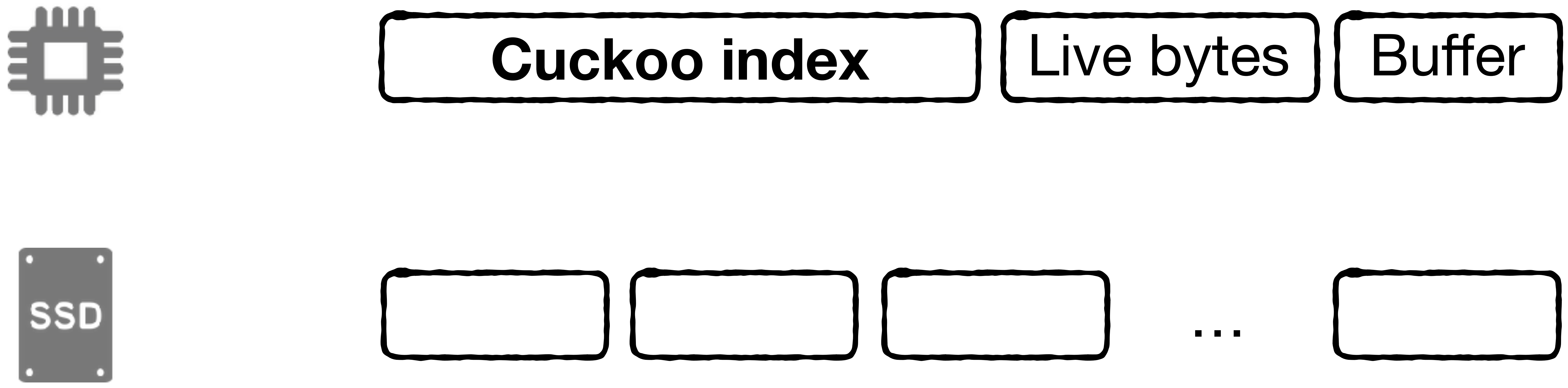
K = **key size** = **$\Omega(\log_2 N)$**

α = collision resolution overheads ≈0.8 → ≈0.95



Problems with storing full keys in a hash table

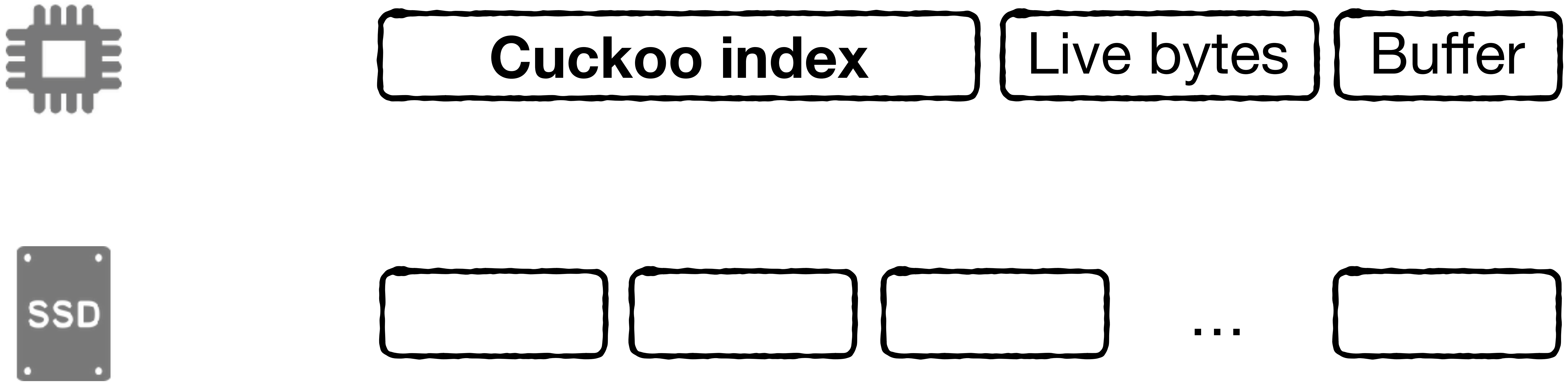
(1) Keys may be arbitrarily large



Problems with storing full keys in a hash table

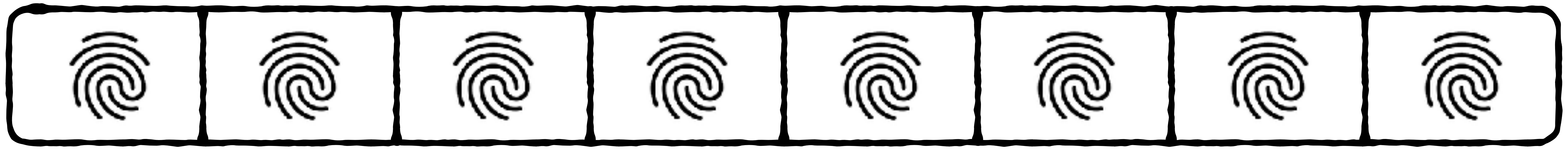
(1) Keys may be arbitrarily large

(2) Keys can be variable-length (requires additional metadata and CPU cycles to encode)



Cuckoo Filter

Same as Cuckoo hash tables, but store fingerprints instead of keys

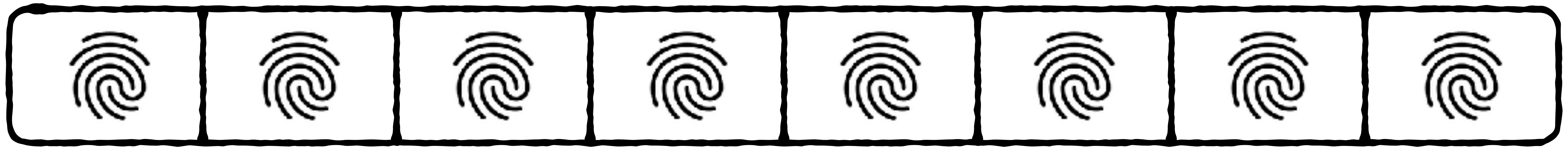


Cuckoo Filter

Same as Cuckoo hash tables, but store fingerprints instead of keys

A fingerprint is a hash digest derived by hashing a key

FP( **) =** 

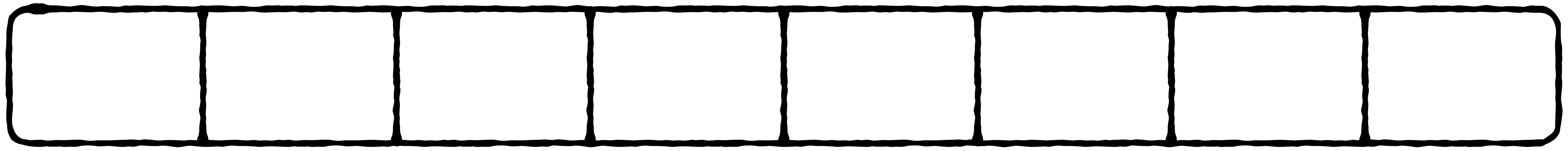


Cuckoo Filter

Same as Cuckoo hash tables, but store fingerprints instead of keys

A fingerprint is a hash digest derived by hashing a key

Example: $\mathbf{FP(\ X\ ) =}\ \overbrace{0100}^{\textbf{M bits}}$

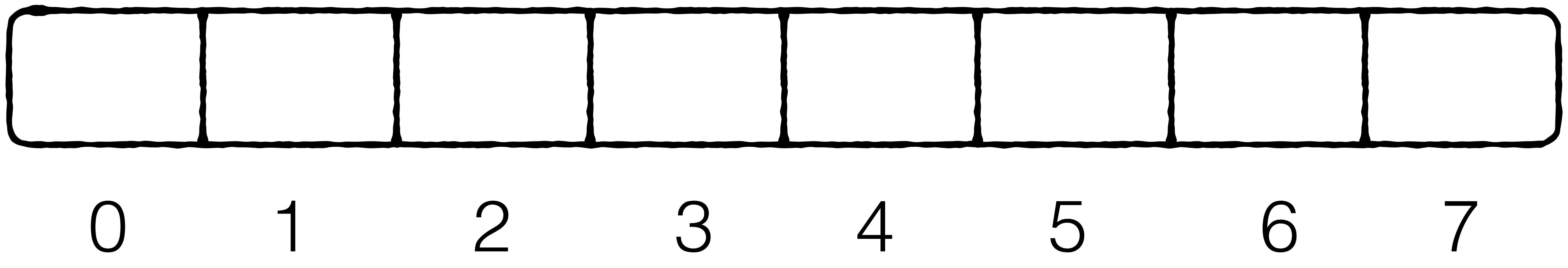


Cuckoo Filter

Same as Cuckoo hash tables, but store fingerprints instead of keys

A fingerprint is a hash digest derived by hashing a key

Example: $\textbf{FP(X) =}\ \overbrace{0100}^{\textbf{M bits}}$ $\qquad$ $\mathbf{h_1(X) =}$ Bucket address

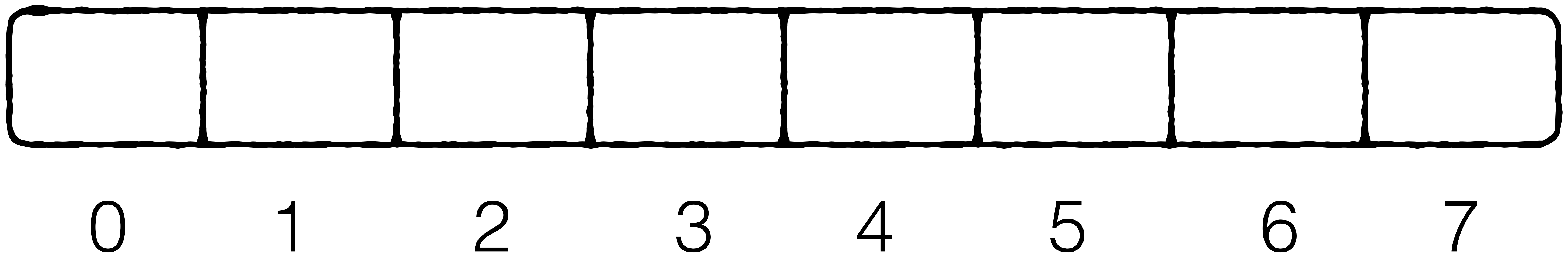


Cuckoo Filter

Same as Cuckoo hash tables, but store fingerprints instead of keys

A fingerprint is a hash digest derived by hashing a key

Example: $\mathbf{FP(X) =} \overbrace{0100}^{\textbf{M bits}}$ $\mathbf{h_1(X) =} 3$



Cuckoo Filter

Same as Cuckoo hash tables, but store fingerprints instead of keys

A fingerprint is a hash digest derived by hashing a key

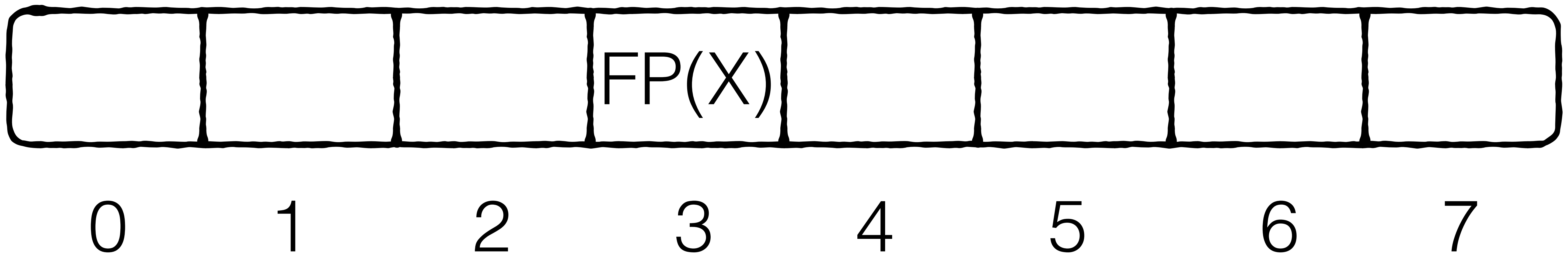
Example: $FP(X) = \overbrace{\qquad\qquad}^{\text{M bits}}$ $\quad h_1(X) = \mathbf{3}$

			0100				
0	1	2	3	4	5	6	7

Cuckoo Filter

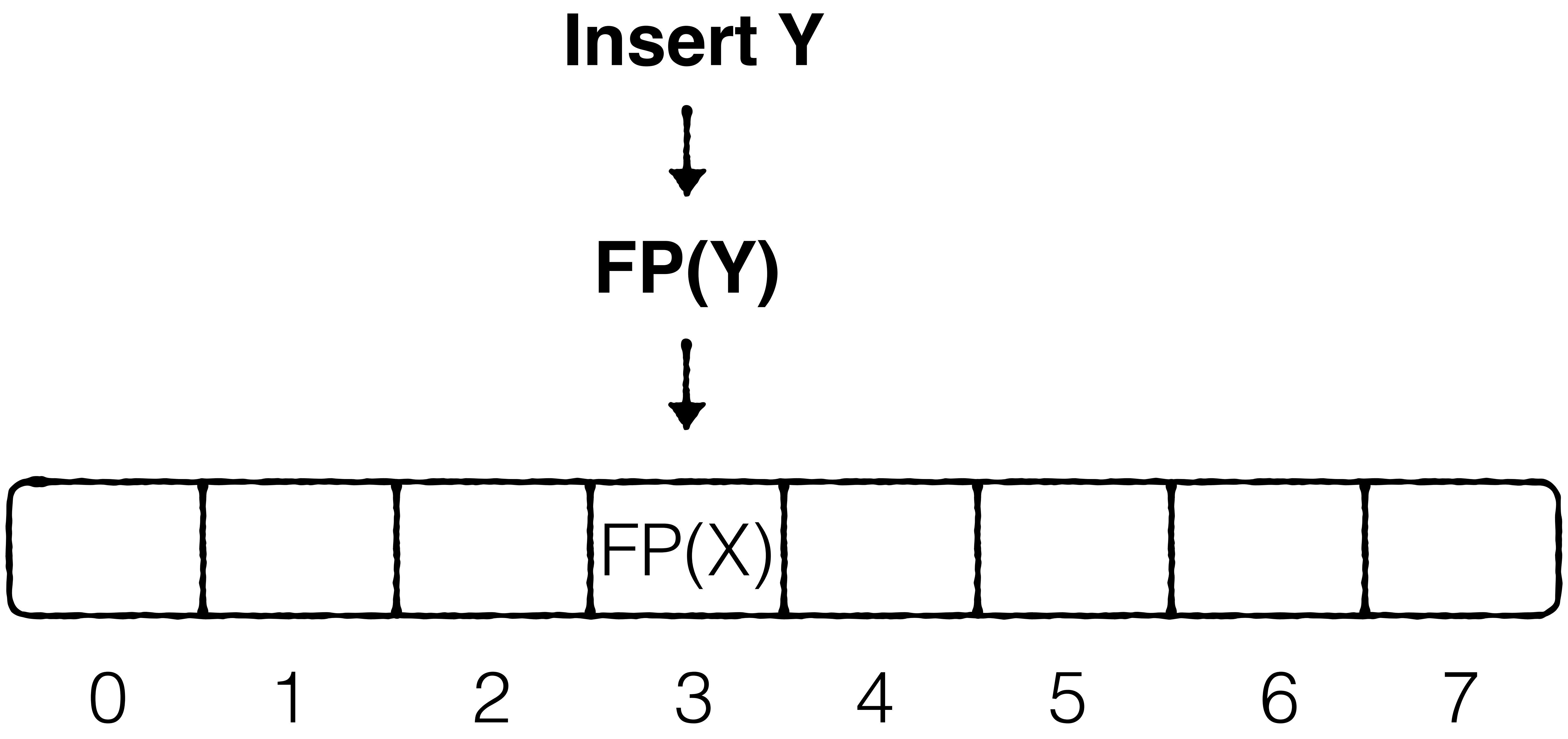
Same as Cuckoo hash tables, but store fingerprints instead of keys

A fingerprint is a hash digest derived by hashing a key



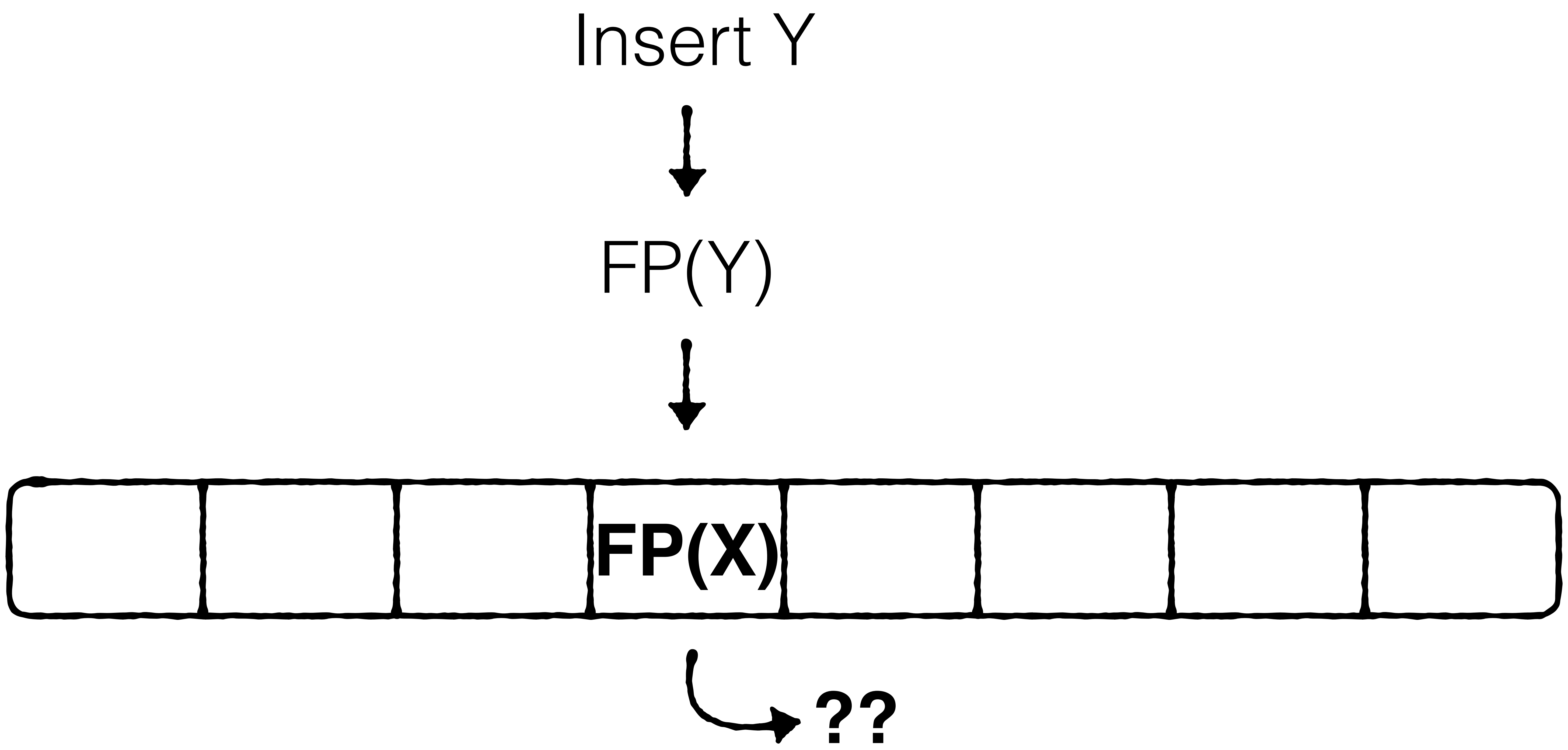
Cuckoo Filter

Suppose we then insert another fingerprint to the same bucket



Cuckoo Filter

Suppose we then insert another fingerprint to the same bucket



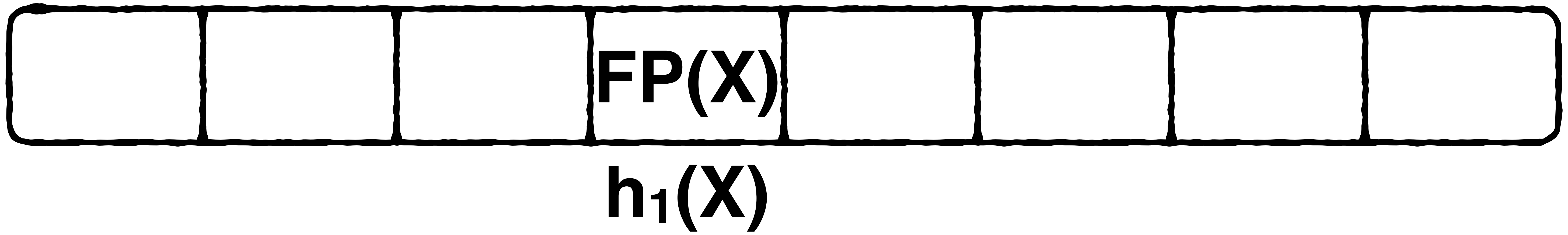
Where to evict X? We no longer have its key!
How to derive an alternative bucket?

How to derive an alternative bucket?

We have two pieces of information about X

(1) Bucket address: $h_1(X)$

(2) fingerprint FP(X)



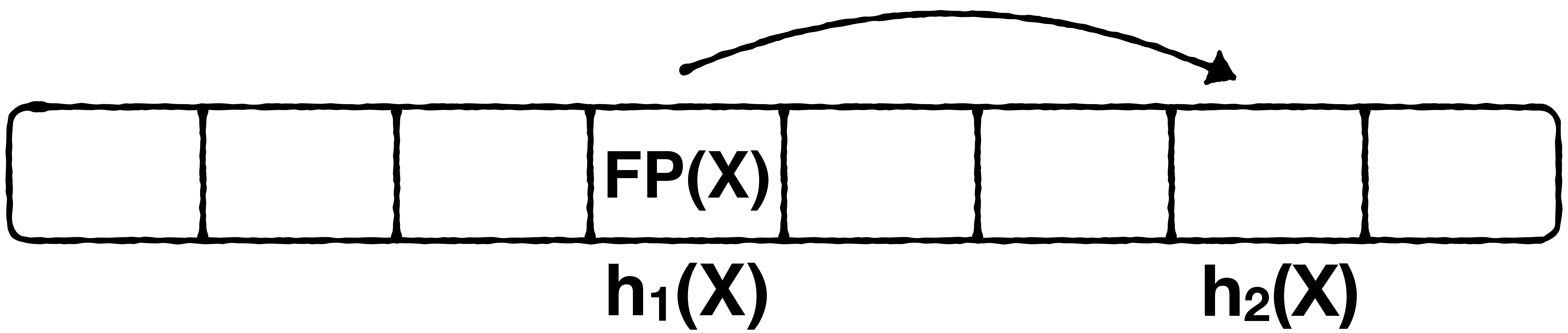
How to derive an alternative bucket?

We have two pieces of information about X

(1) Bucket address: $h_1(X)$

(2) fingerprint FP(X)

We want to combine them to give alternative bucket address $h_2(X)$



How to derive an alternative bucket?

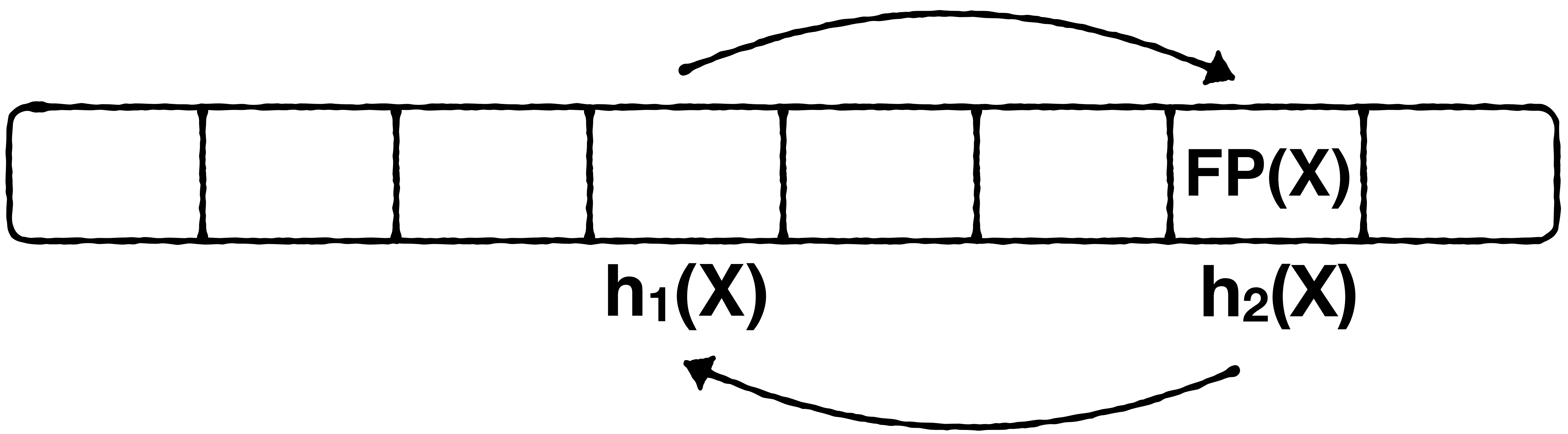
We have two pieces of information about X

(1) Bucket address: $h_1(X)$

(2) fingerprint FP(X)

We want to combine them to give alternative bucket address $h_2(X)$

This mapping must be reversible, so we can derive $h_1(X)$ from $h_2(X)$ and FP(X)



How to derive an alternative bucket?

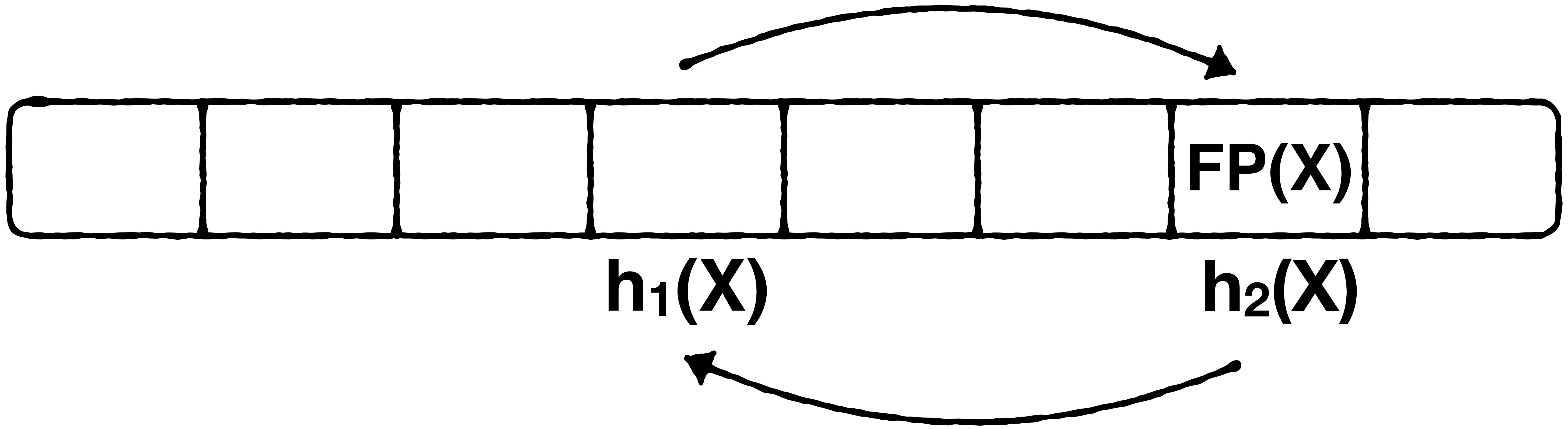
We have two pieces of information about X

(1) Bucket address: $h_1(X)$

(2) fingerprint FP(X)

We want to combine them to give alternative bucket address $h_2(X)$

This mapping must be reversible, so we can derive $h_1(X)$ from $h_2(X)$ and FP(X)



Any ideas?

XOR Operator

Input 1	0	0	1	1
		$\oplus$		
Input 2	0	1	0	1
		=		
Parity	0	1	1	0

XOR Operator

Input 1	0	0	1	1

$$\oplus$$

Input 2	0	1	0	1

$$=$$

Parity	0	1	1	0

If number of 1s in an input column is even, the result is 0. If odd, it is 1.

Parity can help recover any input

Suppose we lost input 2

Input 1	0	0	1	1
Parity	0	1	1	0

Parity can help recover any input

Suppose we lost input 2

Input 1	0	0	1	1
		$\oplus$		
Parity	0	1	1	0
		=		
Input 2	0	1	0	1

Recovered

Parity can help recover any input

Or suppose we lost input 1

Input 2	0	1	0	1
Parity	0	1	1	0

Parity can help recover any input

Or suppose we lost input 1

Input 2	0	1	0	1

$$\oplus$$

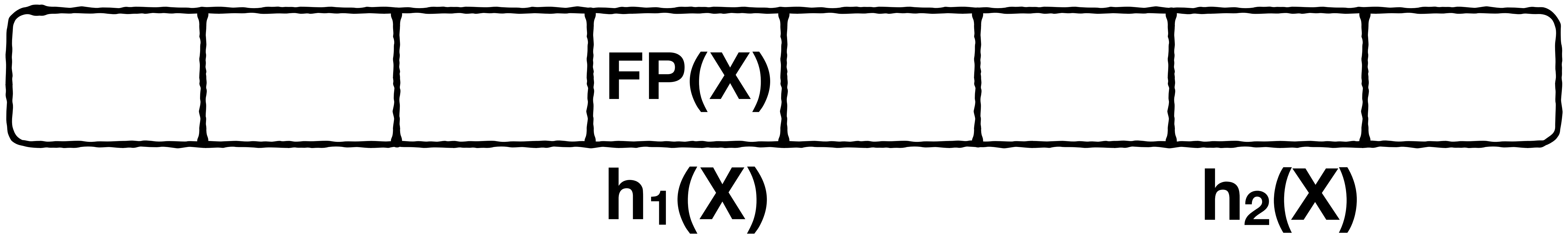
Parity	0	1	1	0

$$=$$

Input 1	0	0	1	1	**Recovered**

Using XOR in Cuckoo filters

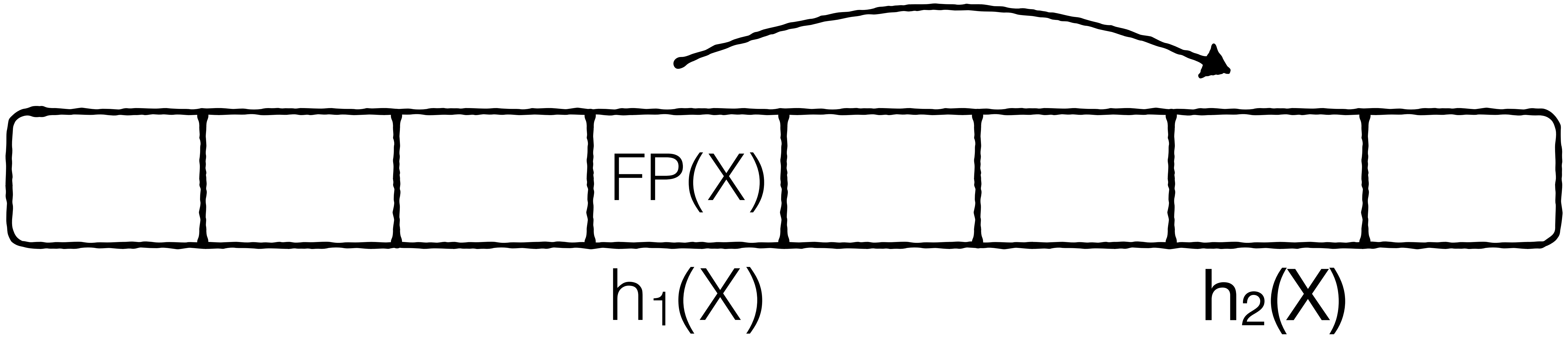
Let's XOR the bucket address and a hash of the fingerprint:



Using XOR in Cuckoo filters

Let’s XOR the bucket address and a hash of the fingerprint:

$$h_2(X) = h_1(X) \text{ xor } hash(FP(X))$$

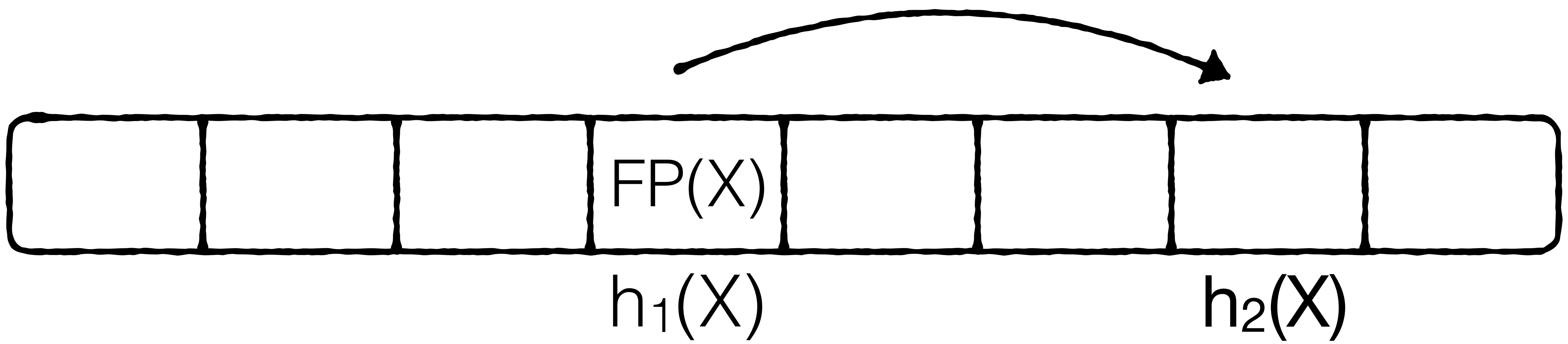


Using XOR in Cuckoo filters

Let's XOR the bucket address and a hash of the fingerprint:

(We hash the fingerprint to map it to the same address space size as the buckets)

$h_2(X) = h_1(X)$ xor $\mathbf{hash(FP(X))}$



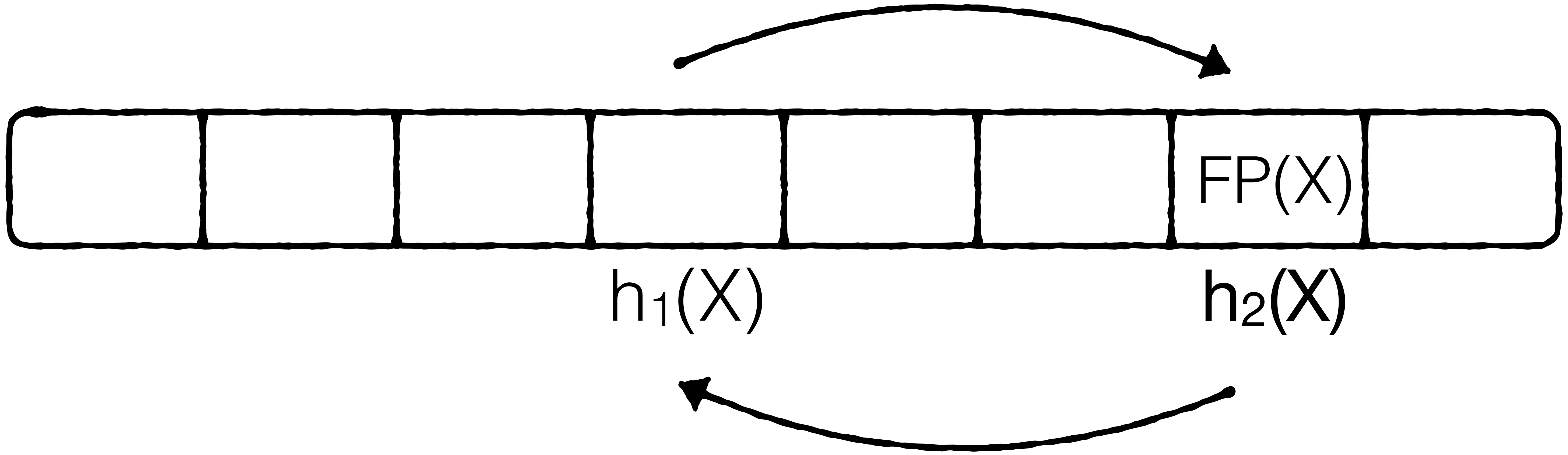
Using XOR in Cuckoo filters

Let's XOR the bucket address and a hash of the fingerprint:

(We hash the fingerprint to map it to the same address space size as the buckets)

The resulting mapping is reversible

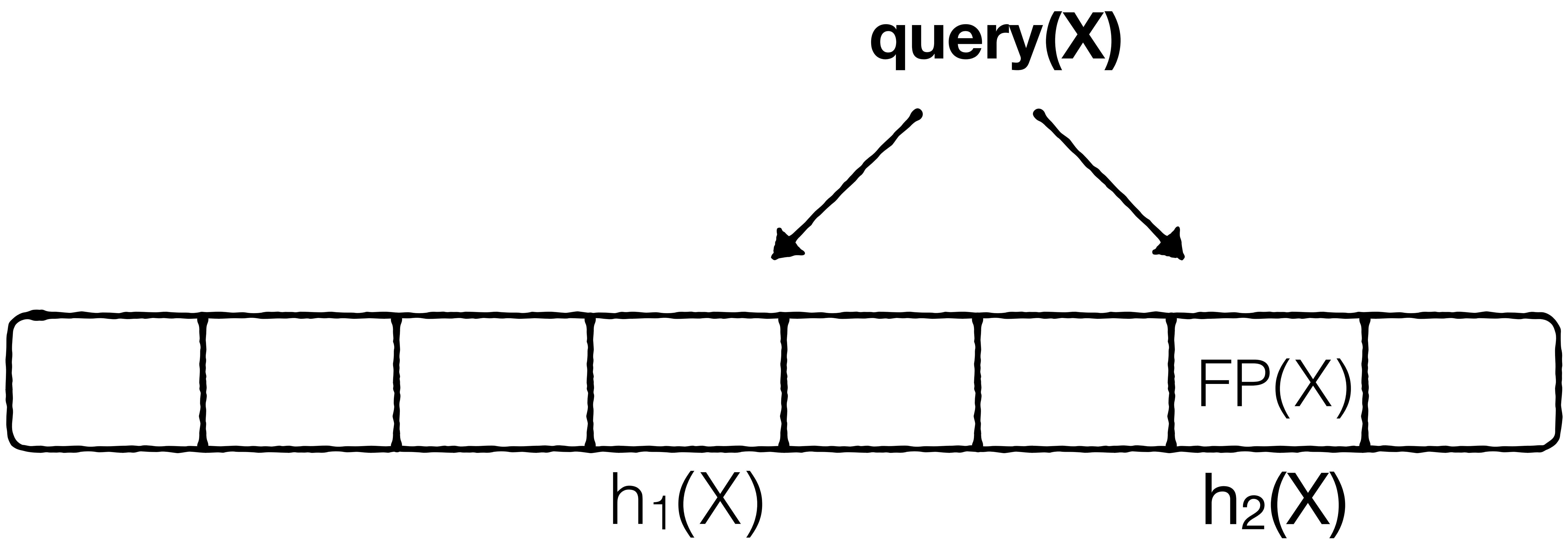
$h_2(X) = h_1(X)$ xor hash(FP(X))



$h_1(X) = h_2(X)$ xor hash(FP(X))

Using XOR in Cuckoo filters

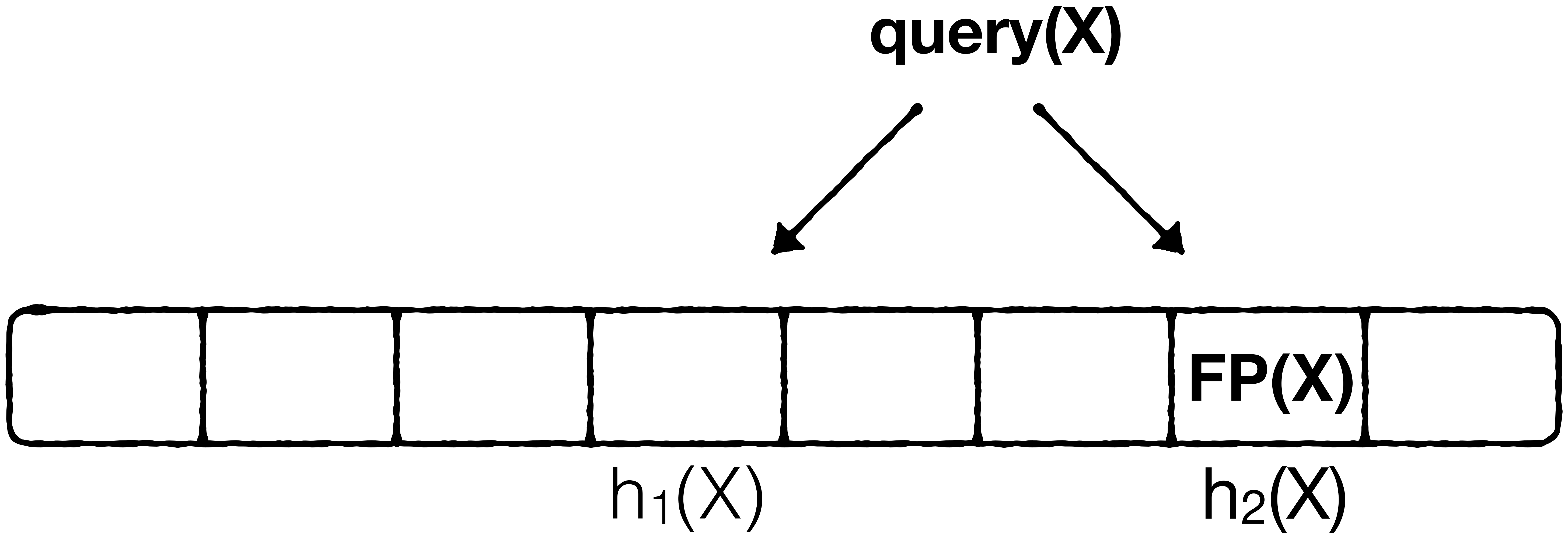
Thus, we must search only two buckets to find an entry's fingerprint



Using XOR in Cuckoo filters

Thus, we must search only two buckets to find an entry's fingerprint

If we find a matching fingerprint, we report a positive.

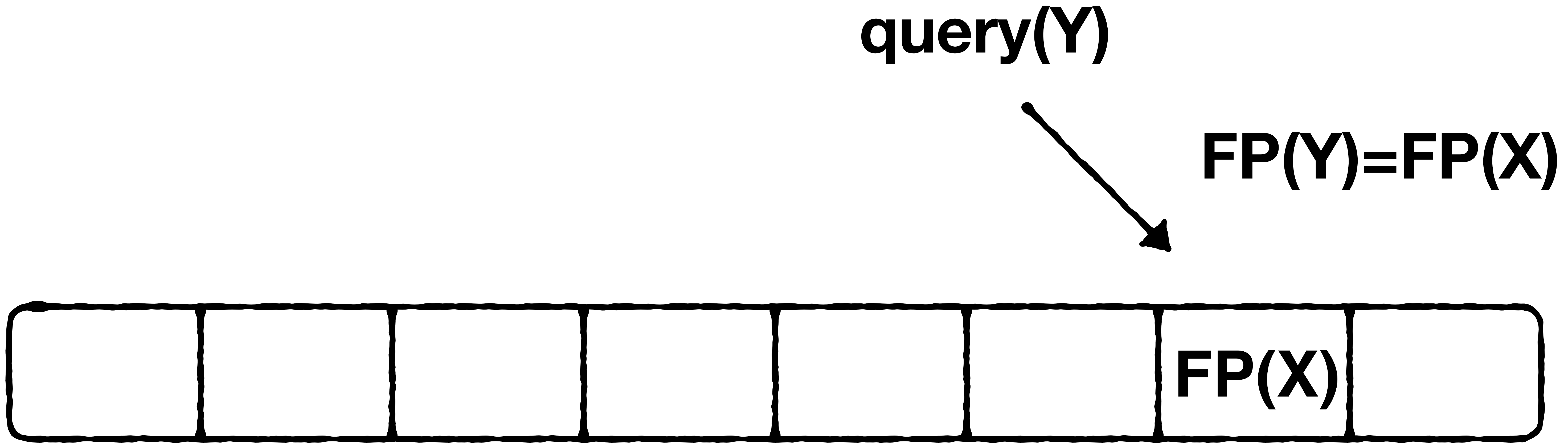


Using XOR in Cuckoo filters

Thus, we must search only two buckets to find an entry’s fingerprint

If we find a matching fingerprint, we report a positive.

However, we can have false positives.

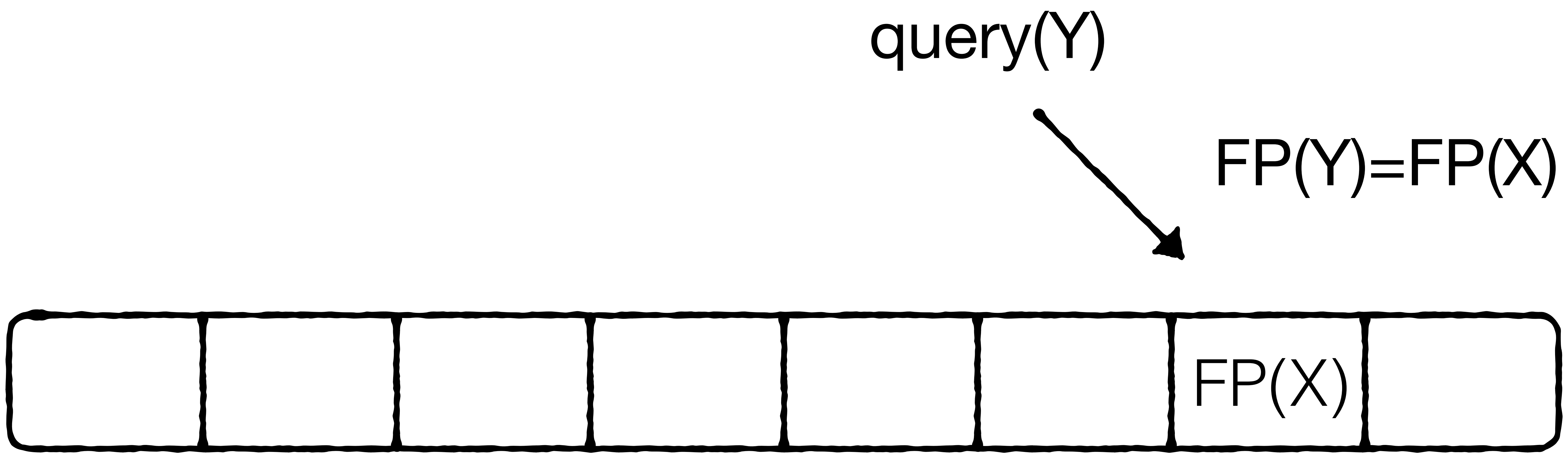


Using XOR in Cuckoo filters

Thus, we must search only two buckets to find an entry's fingerprint

If we find a matching fingerprint, we report a positive.

However, we can have false positives.



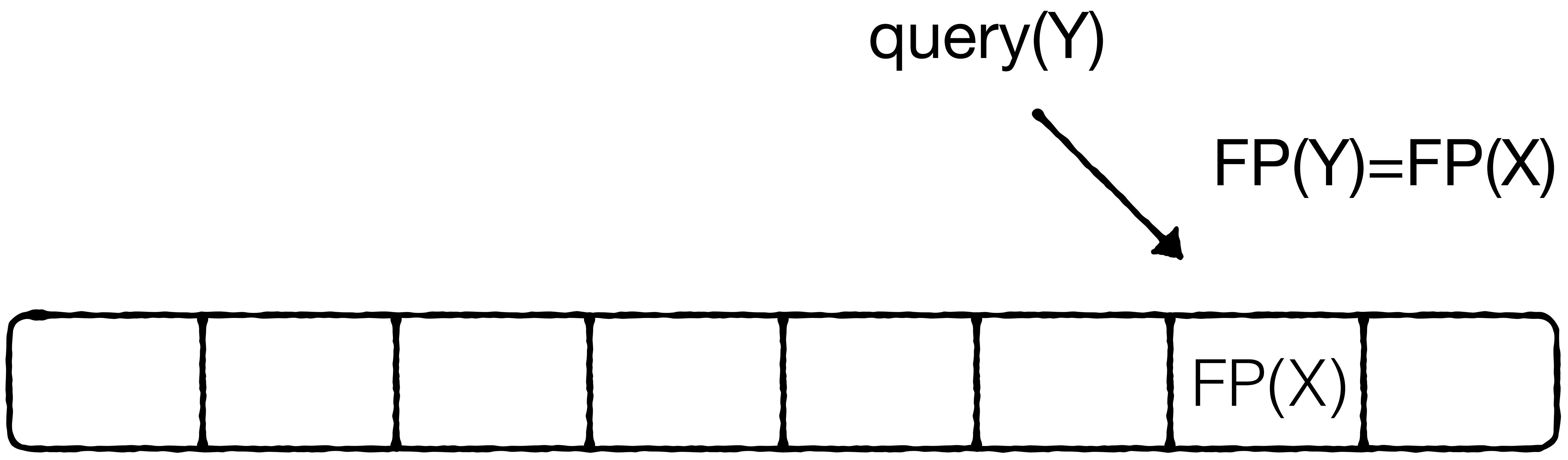
Can we have false negatives?

Using XOR in Cuckoo filters

Thus, we must search only two buckets to find an entry’s fingerprint

If we find a matching fingerprint, we report a positive.

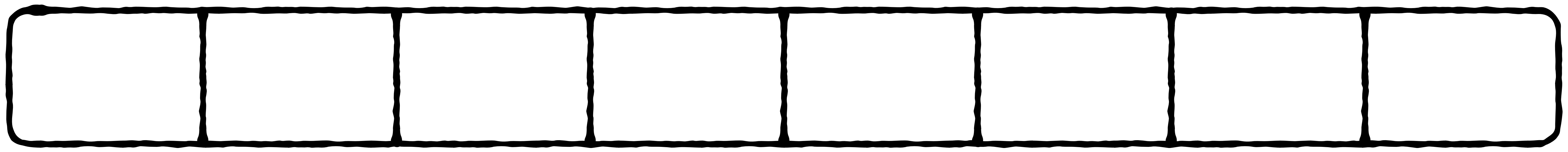
However, we can have false positives.



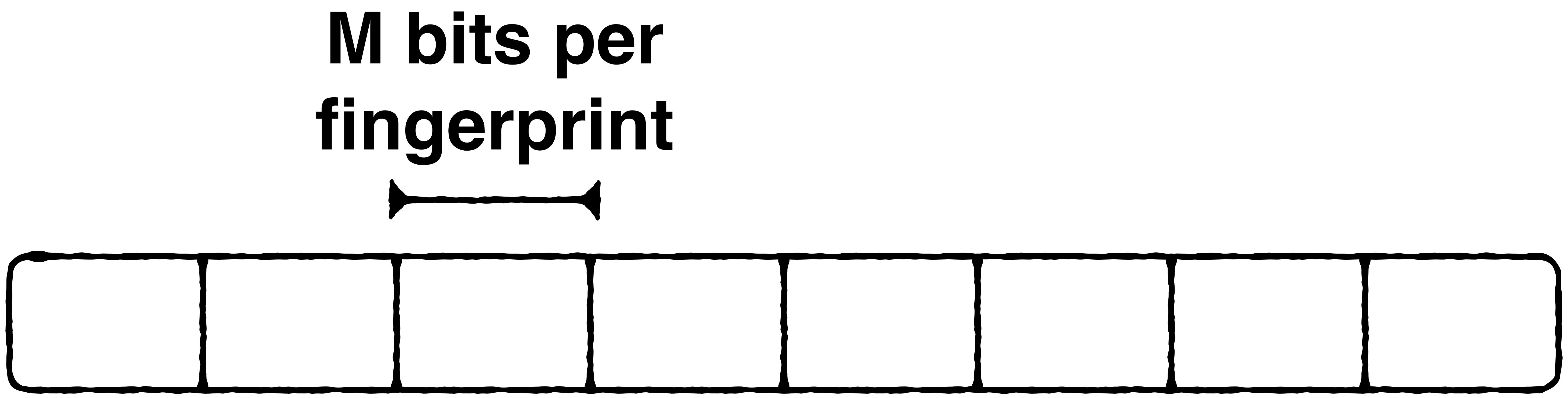
Can we have false negatives?

No, because a fingerprint for a key that had been inserted is in one of only two possible buckets, both of which we search.

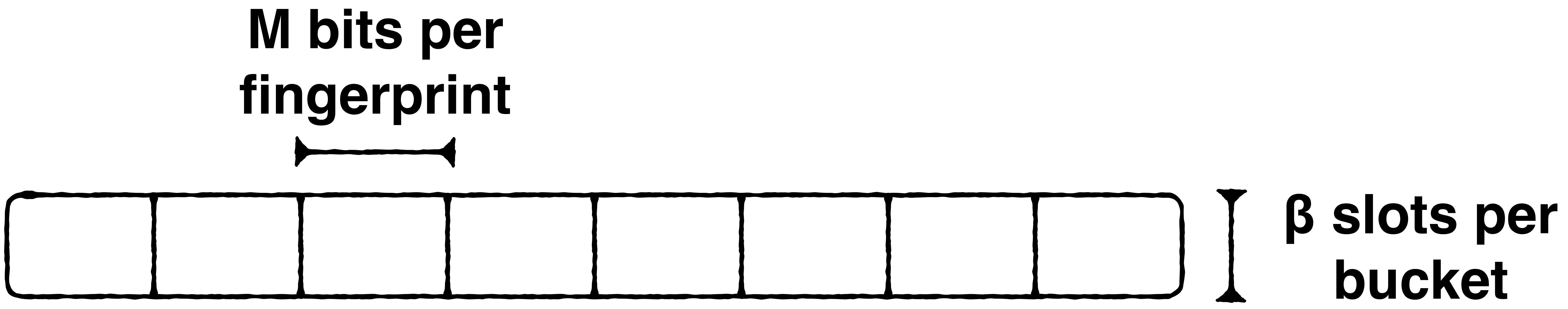
What’s the false positive rate?



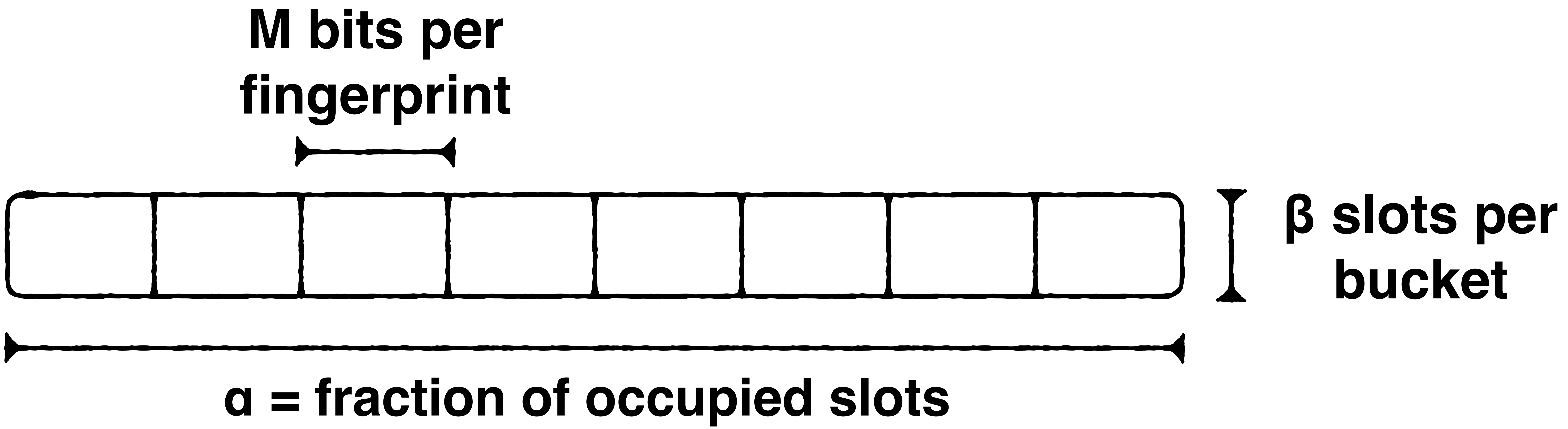
What’s the false positive rate?



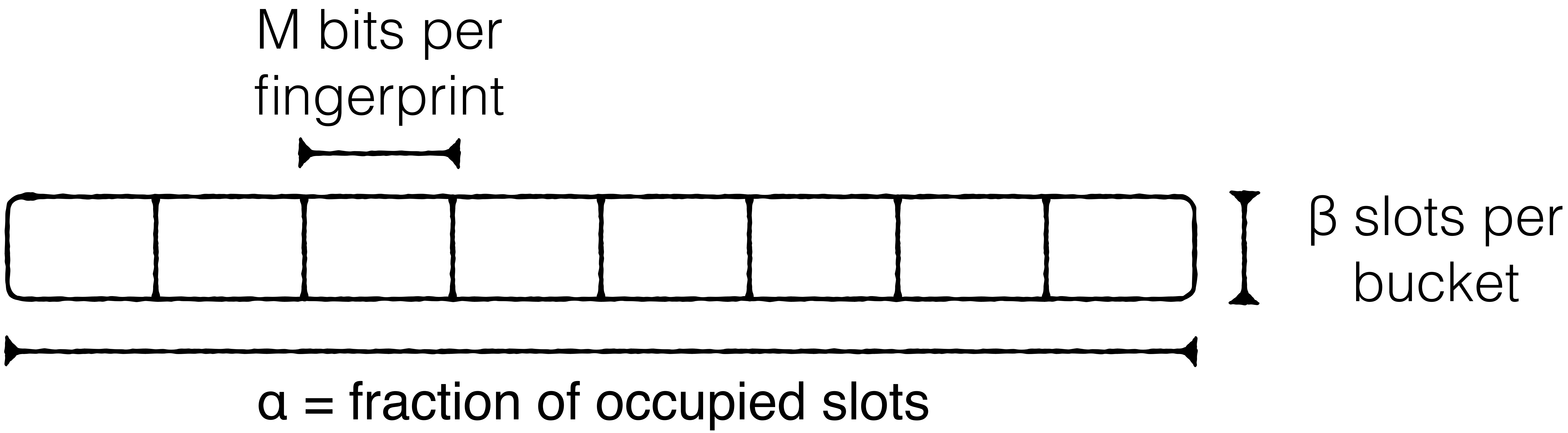
What's the false positive rate?



What’s the false positive rate?

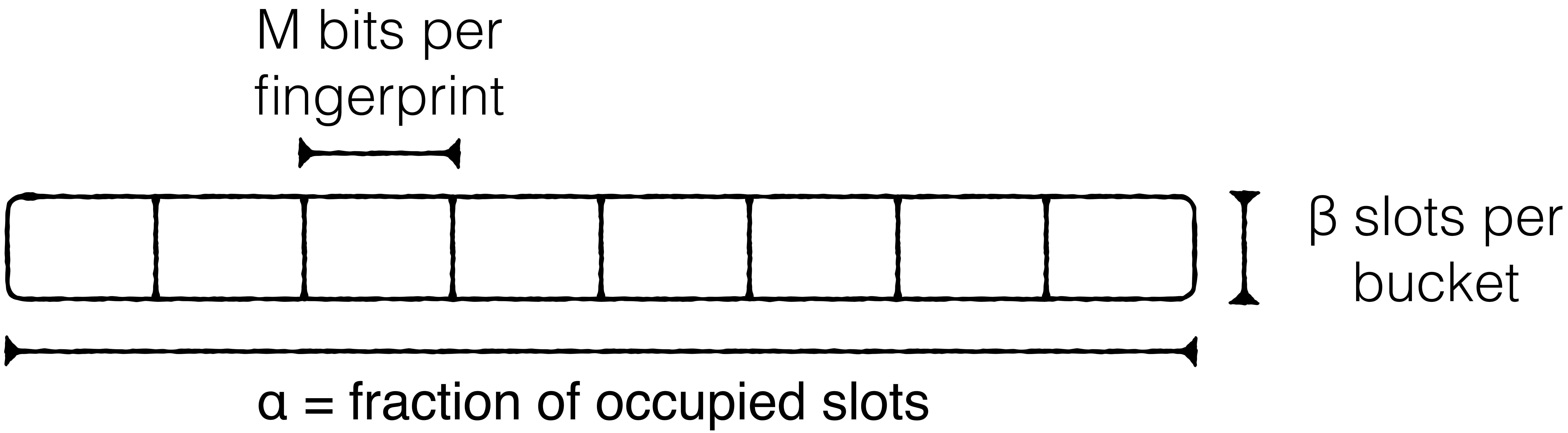


$$\textbf{false positive rate} \approx 2 \cdot 2^{-M} \cdot \beta \cdot \alpha$$



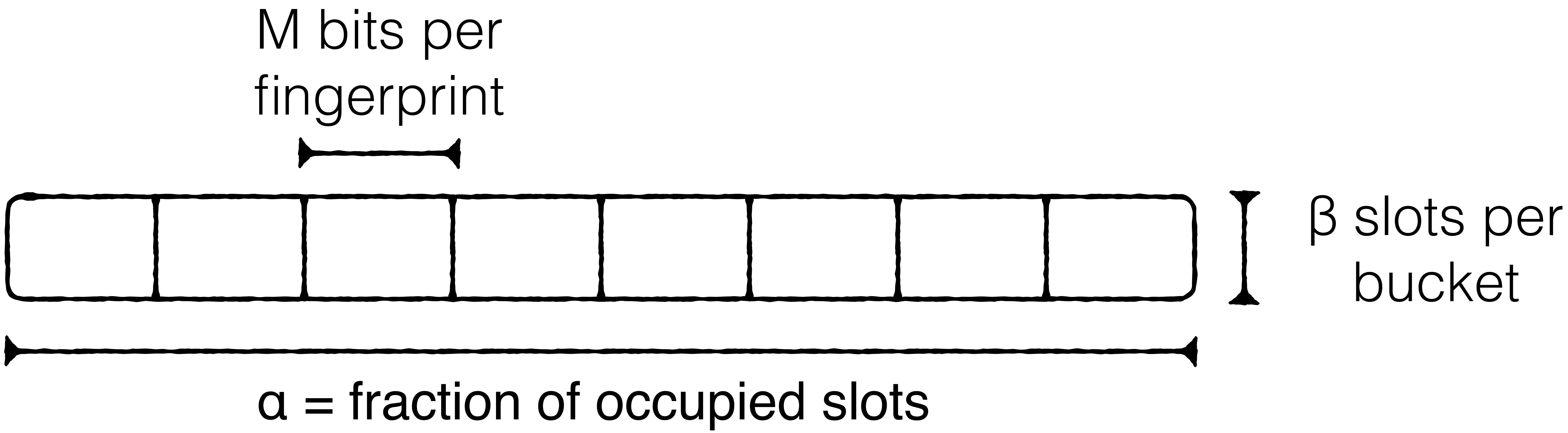
Since a query searches two buckets

false positive rate $\approx 2 \cdot 2^{-M} \cdot \beta \cdot \alpha$

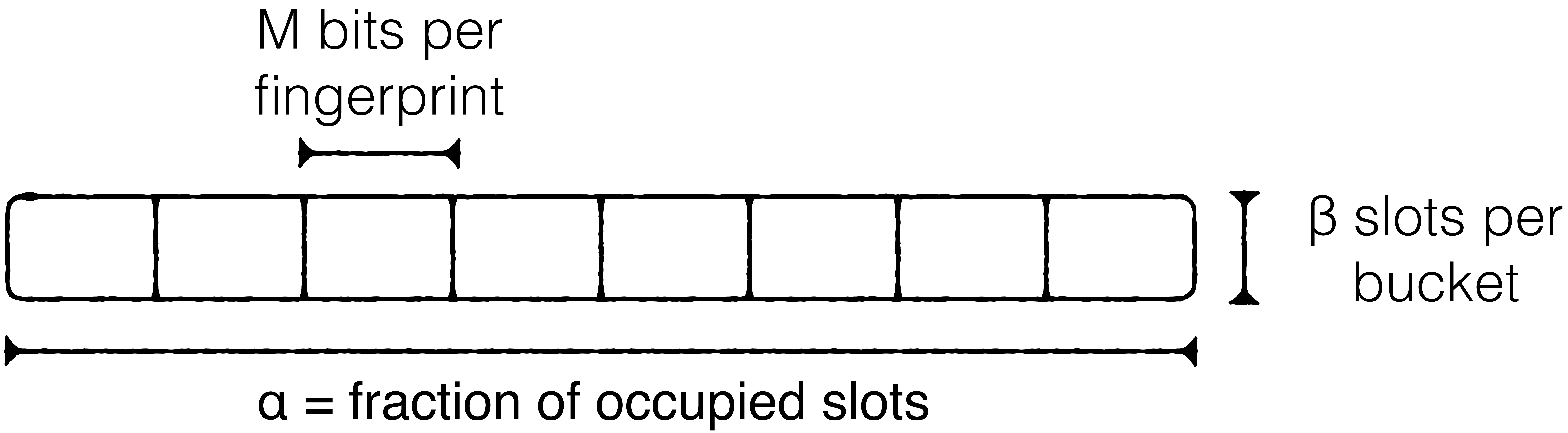


When at capacity

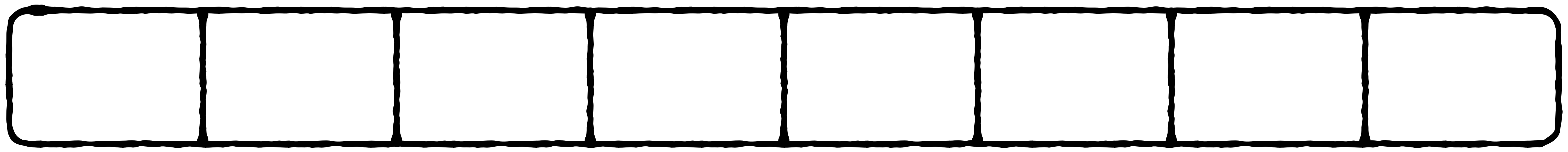
false positive rate $\approx 2 \cdot 2^{-M} \cdot 4 \cdot \mathbf{0.95}$



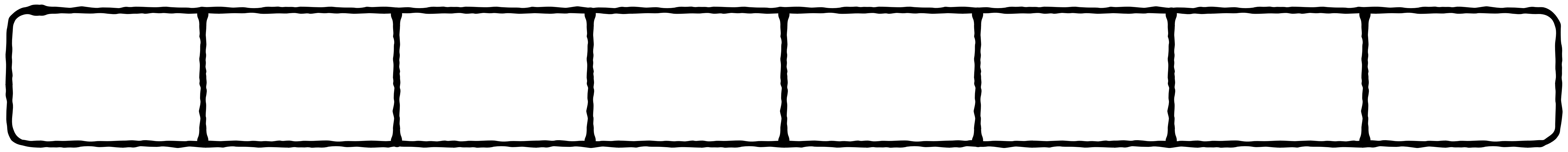
false positive rate $\approx \mathbf{2^{-M+3}}$



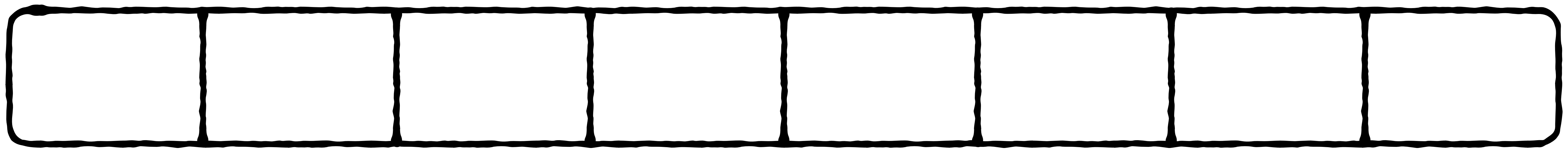
false positive rate $\approx 2^{-M+3}$ **$< 2^{-M \cdot \ln(2)}$** **For Bloom filter when $M \geq 10$**



	Cuckoo Filter	**Bloom filter**
false positive rate:	$\approx 2^{-M+3}$	$2^{-M \cdot \ln(2)}$
Memory accesses for positive query?		



	Cuckoo Filter	**Bloom filter**
false positive rate:	$\approx 2^{-M+3}$	$2^{-M \cdot \ln(2)}$
Memory accesses for positive query	**1.5**	$\mathbf{M \cdot \ln(2)}$



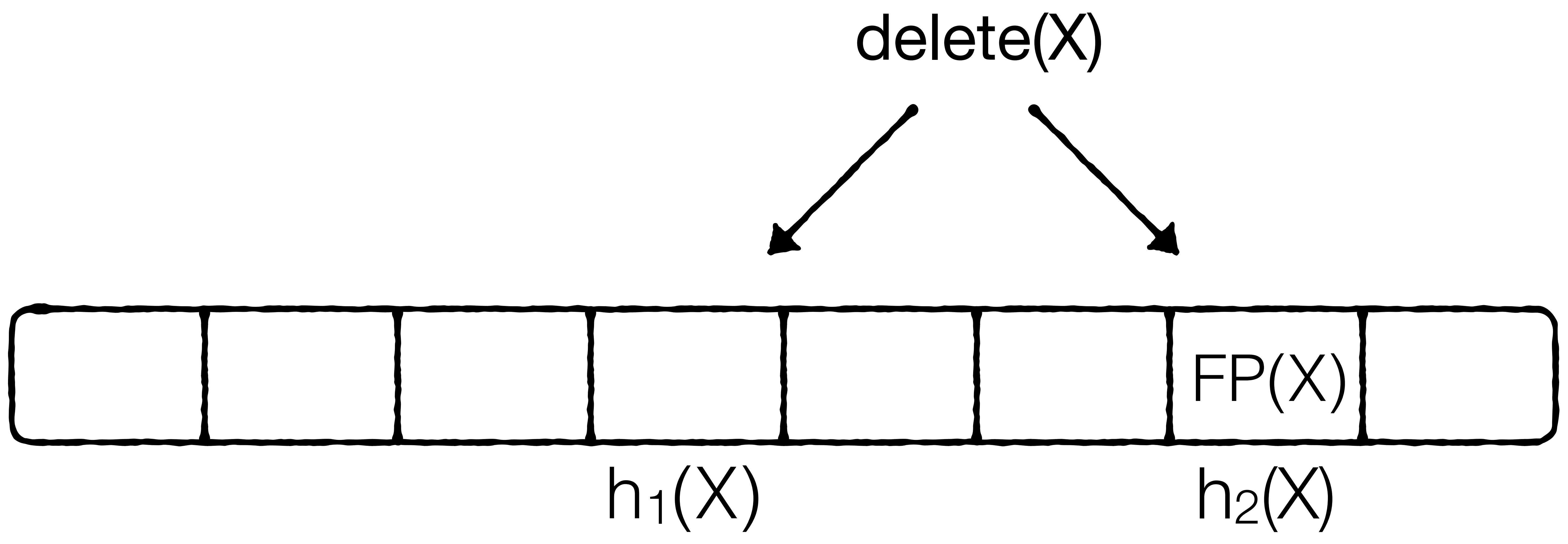
	Cuckoo Filter	**Bloom filter**
false positive rate:	$\approx 2^{-M+3}$	$2^{-M \cdot \ln(2)}$
Memory accesses for positive query	1.5	$M \cdot \ln(2)$
Memory accesses for negative query?		

	Cuckoo Filter	**Bloom filter**
false positive rate:	$\approx 2^{-M+3}$	$2^{-M \cdot \ln(2)}$
Memory accesses for positive query	1.5	$M \cdot \ln(2)$
Memory accesses for negative query	**2**	**≈2**

	Cuckoo Filter	**Bloom filter**
false positive rate:	$\approx 2^{-M+3}$	$2^{-M \cdot \ln(2)}$
Memory accesses for positive query	1.5	$M \cdot \ln(2)$
Memory accesses for negative query	2	≈ 2
Insertion cost	Exp. O(1)	$M \cdot \ln(2)$

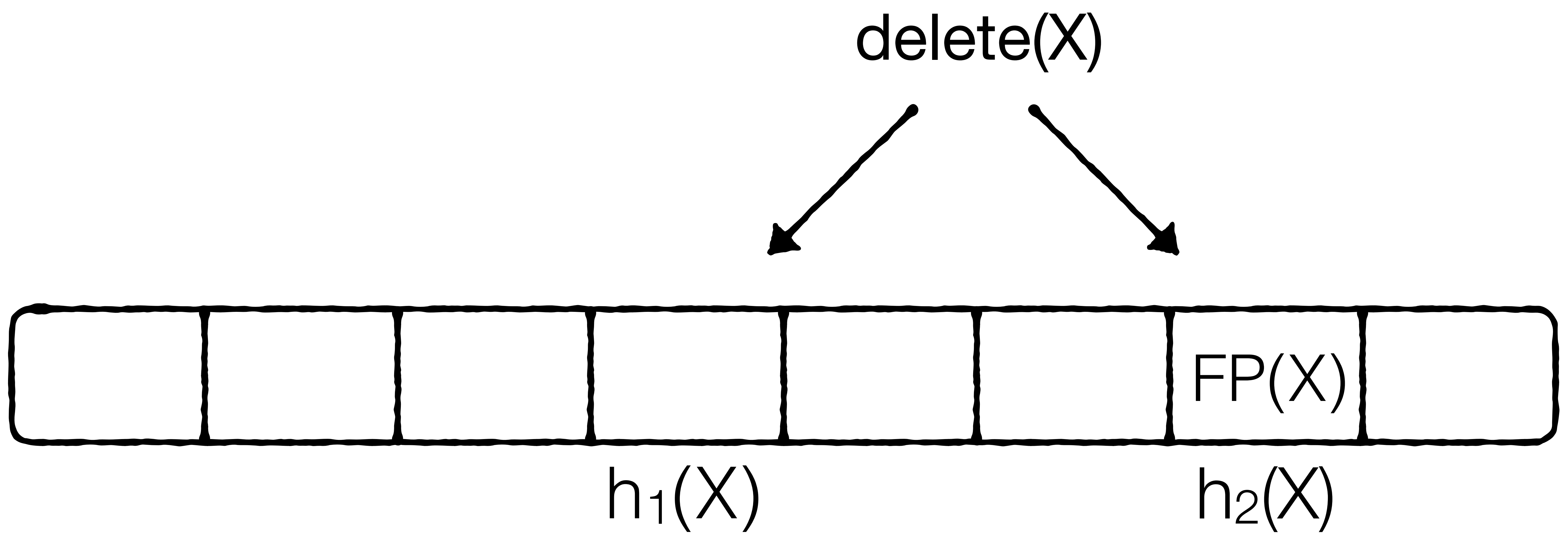
	Cuckoo Filter	Bloom filter
false positive rate:	$\approx 2^{-M+3}$	$2^{-M \cdot \ln(2)}$
Memory accesses for positive query	1.5	$M \cdot \ln(2)$
Memory accesses for negative query	2	≈ 2
Insertion cost	Exp. $O(1)$	$M \cdot \ln(2)$
Deletes?		**N/A**
Storing payloads?		**N/A**

Deletes in a Cuckoo Filter



Deletes in a Cuckoo Filter

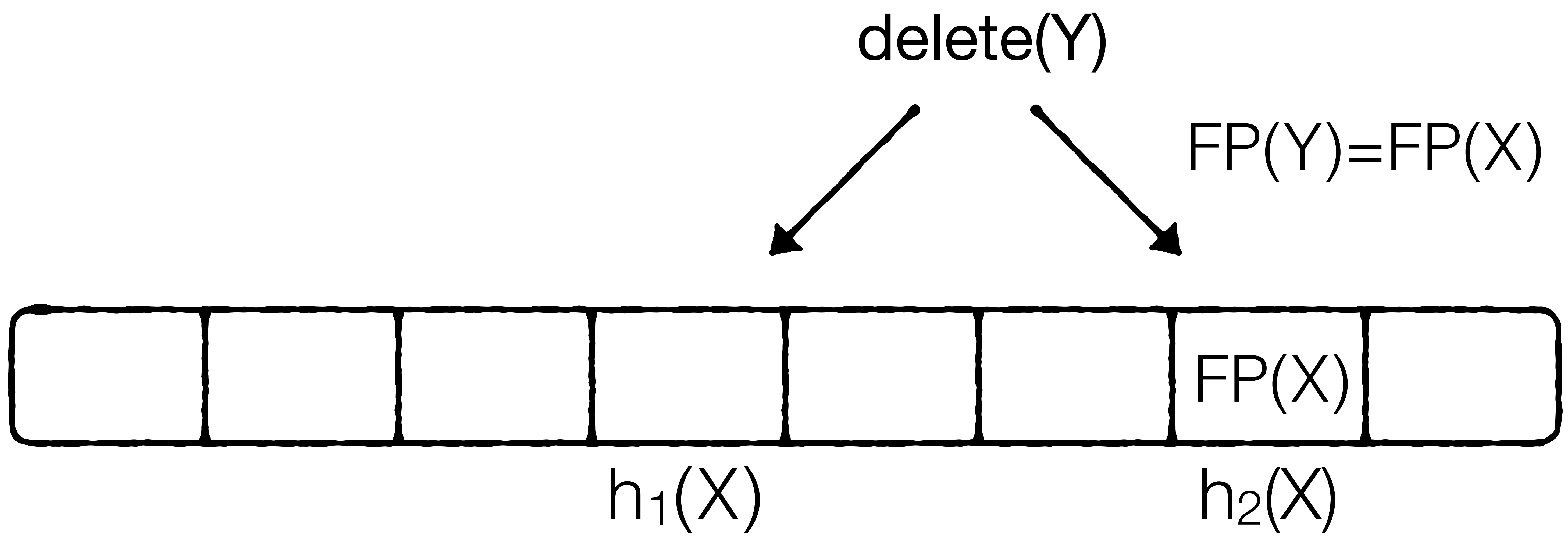
A delete searches both buckets for a key and removes a matching fingerprints.



Deletes in a Cuckoo Filter

A delete searches both buckets for a key and removes a matching fingerprints.

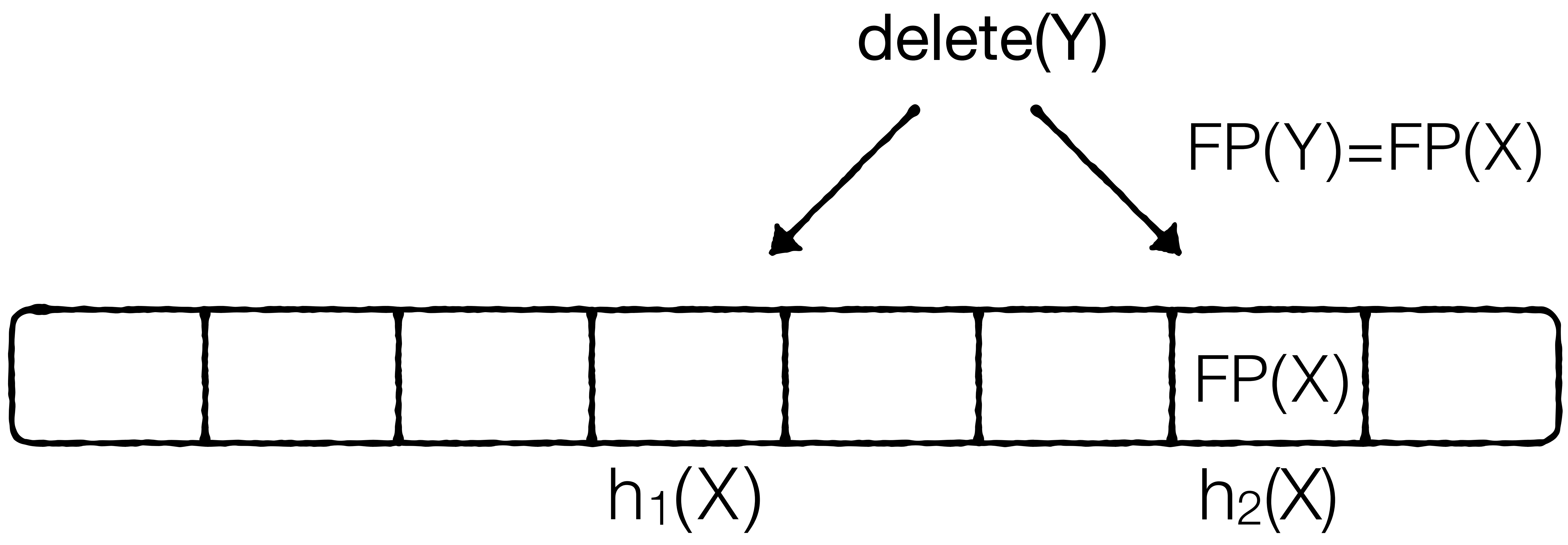
What if we delete a fingerprint for a key that was never inserted?



Deletes in a Cuckoo Filter

A delete searches both buckets for a key and removes a matching fingerprints.

What if we delete a fingerprint for a key that was never inserted? **False negatives…**

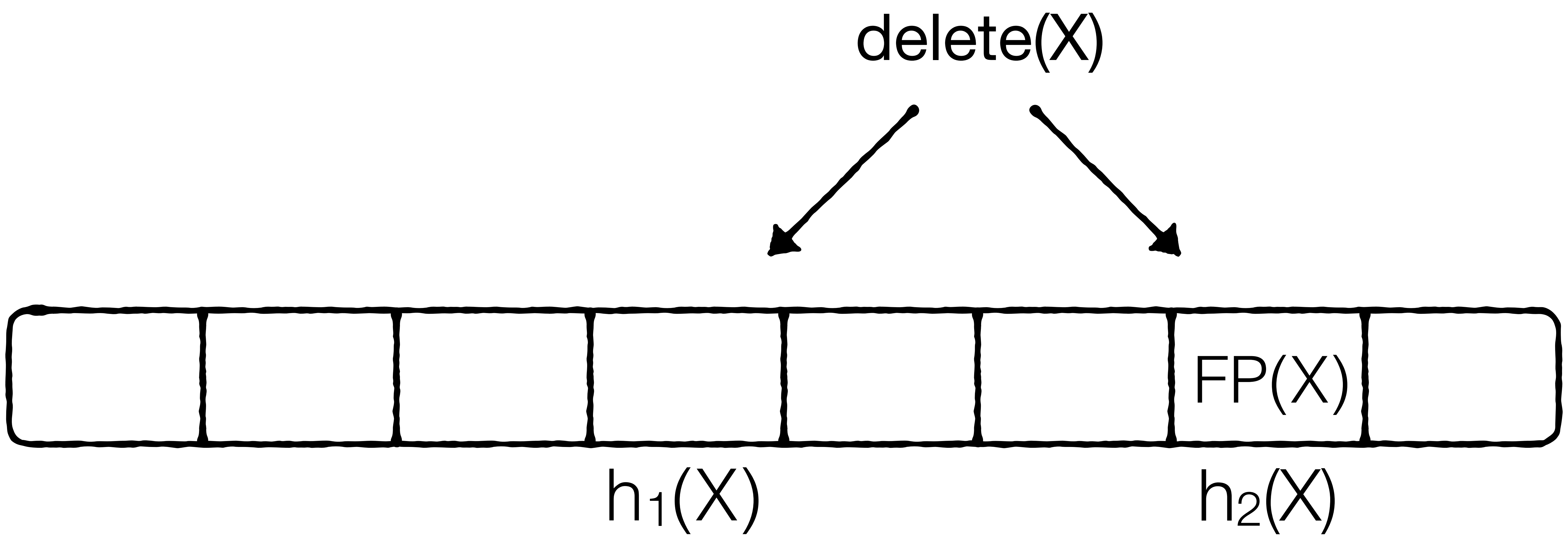


Deletes in a Cuckoo Filter

A delete searches both buckets for a key and removes a matching fingerprints.

What if we delete a fingerprint for a key that was never inserted? False negatives…

As long as we delete keys we know for sure have been inserted, no false negatives can occur



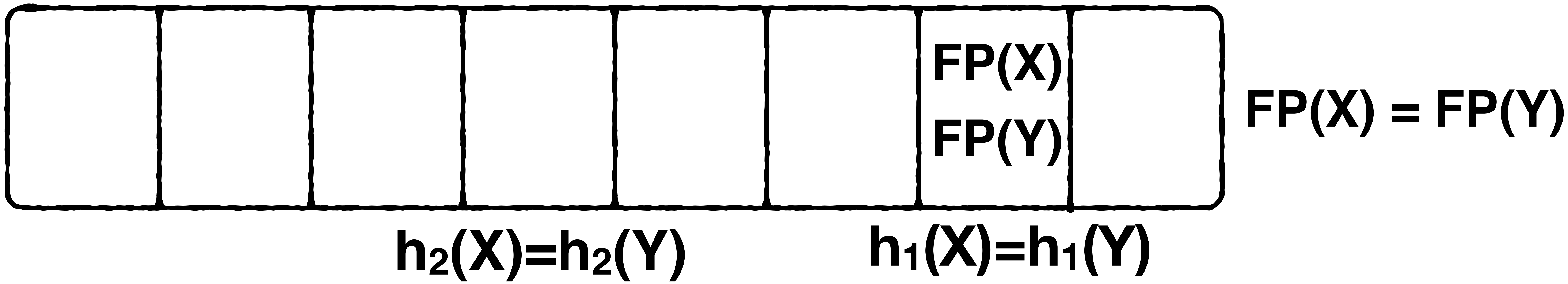
Deletes in a Cuckoo Filter

A delete searches both buckets for a key and removes a matching fingerprints.

What if we delete a fingerprint for a key that was never inserted?   False negatives…

As long as we delete keys we know for sure have been inserted, no false negatives can occur

Works even if we have matching fingerprints for different keys in the same bucket



Deletes in a Cuckoo Filter

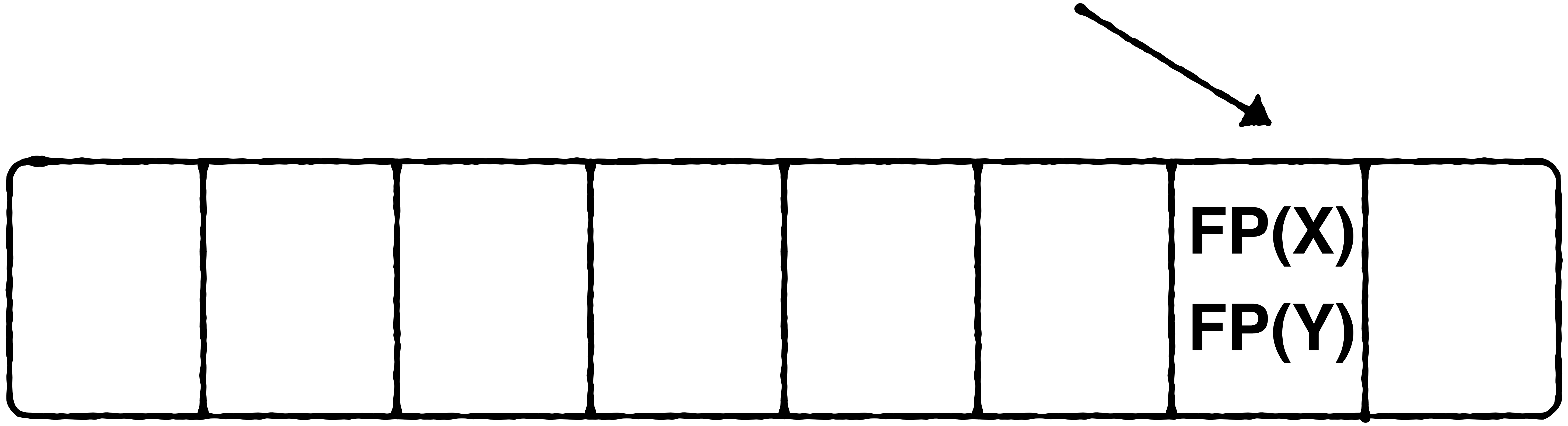
A delete searches both buckets for a key and removes a matching fingerprints.

What if we delete a fingerprint for a key that was never inserted?   False negatives…

As long as we delete keys we know for sure have been inserted, no false negatives can occur

Works even if we have matching fingerprints for different keys in the same bucket

After delete(X), get(Y) will still succeed, whichever fingerprint we delete.



FP(X) = FP(Y)

$h_2(X)=h_2(Y)$ $h_1(X)=h_1(Y)$

	Cuckoo Filter	**Bloom filter**
false positive rate:	$\approx 2^{-M+3}$	$2^{-M \cdot \ln(2)}$
Memory accesses for positive query	1.5	$M \cdot \ln(2)$
Memory accesses for negative query	2	2
Insertion cost	O(1)	$M \cdot \ln(2)$
Deletes?	**1.5**	**N/A**
Storing Payloads?		N/A

Storing Payloads in a Filter

Can we store a payload associated with each key and retrieve it on positive query?

Storing Payloads in a Filter

Can we store a payload associated with each key and retrieve it on positive query?

Bloom filter:

Insert(x)

Map to random bits

0 1 0 0 1 1 0 1 0 0 0 1 0 0 1 1 1

Storing Payloads in a Filter

Can we store a payload associated with each key and retrieve it on positive query?

Insert(x)

Bloom filter:

Map to random bits

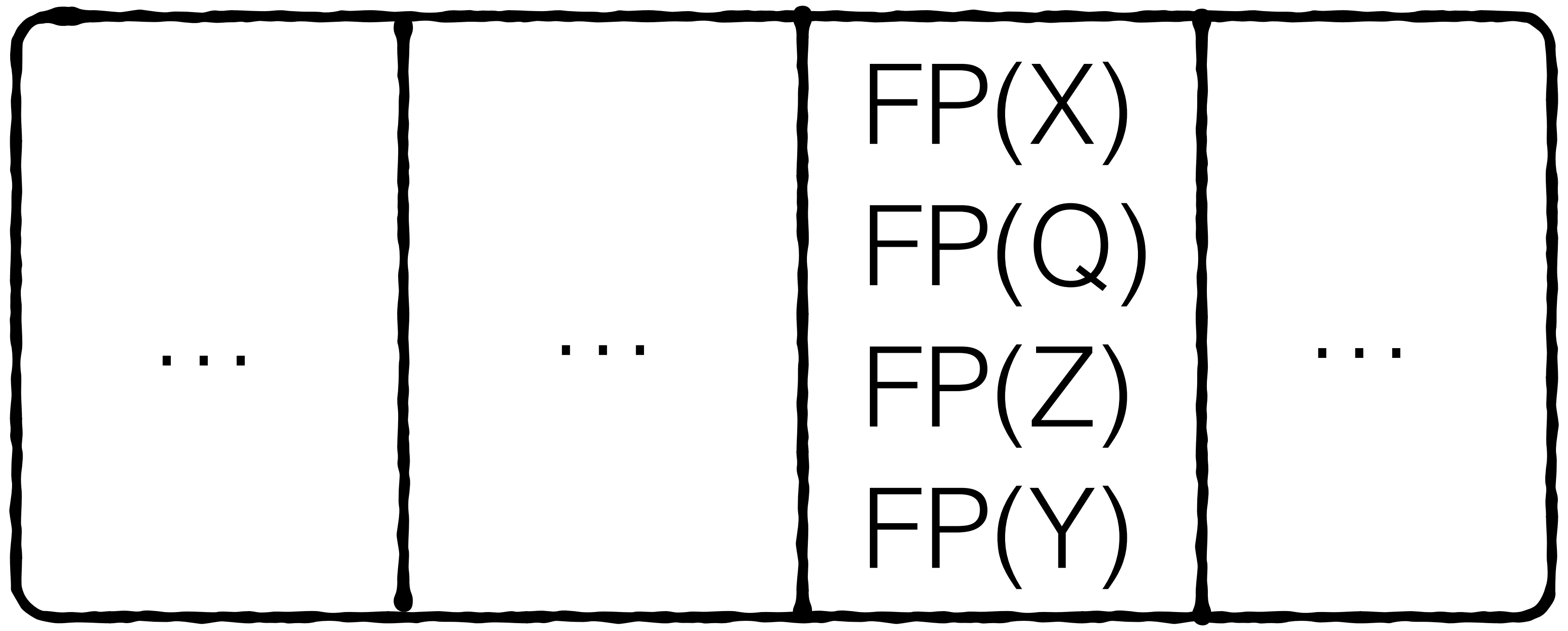
0 1 0 0 1 1 0 1 0 0 0 1 0 0 1 1 1

There is nowhere obvious to associate a payload with each key

Storing Payloads in a Filter

Can we store a payload associated with each key and retrieve it on positive query?

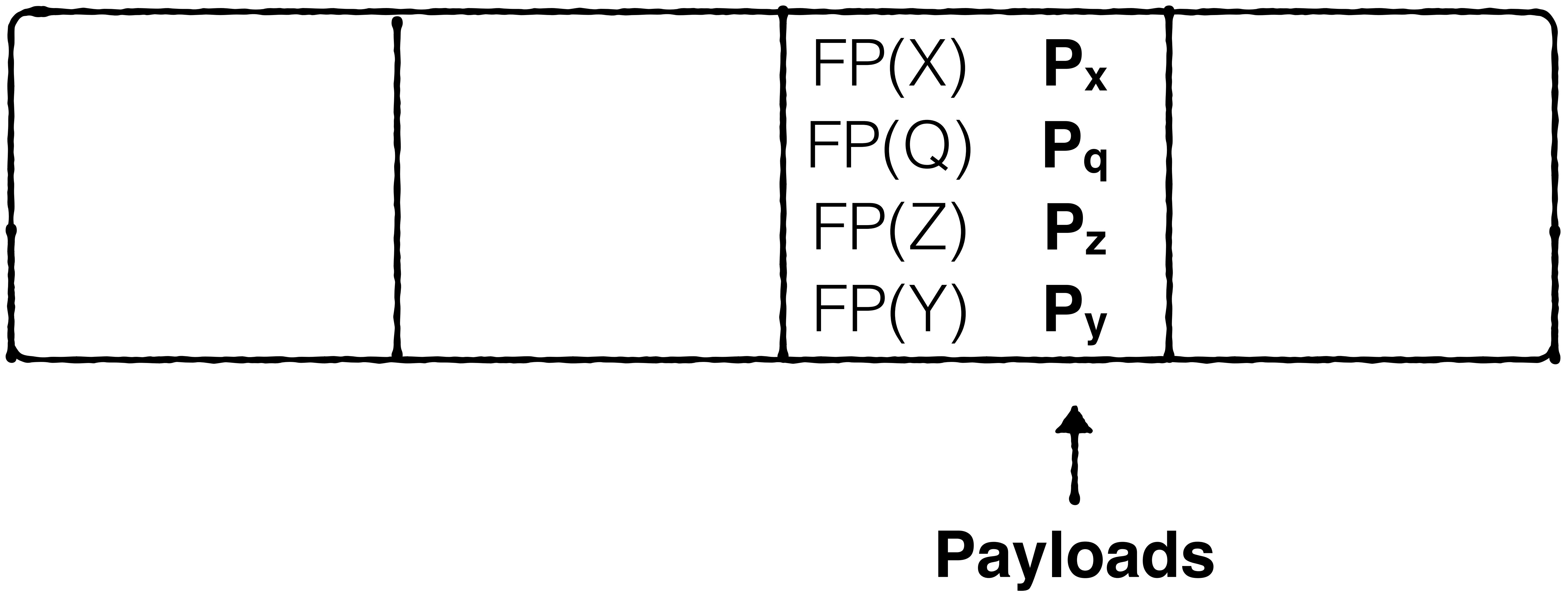
Cuckoo filter:



Storing Payloads in a Filter

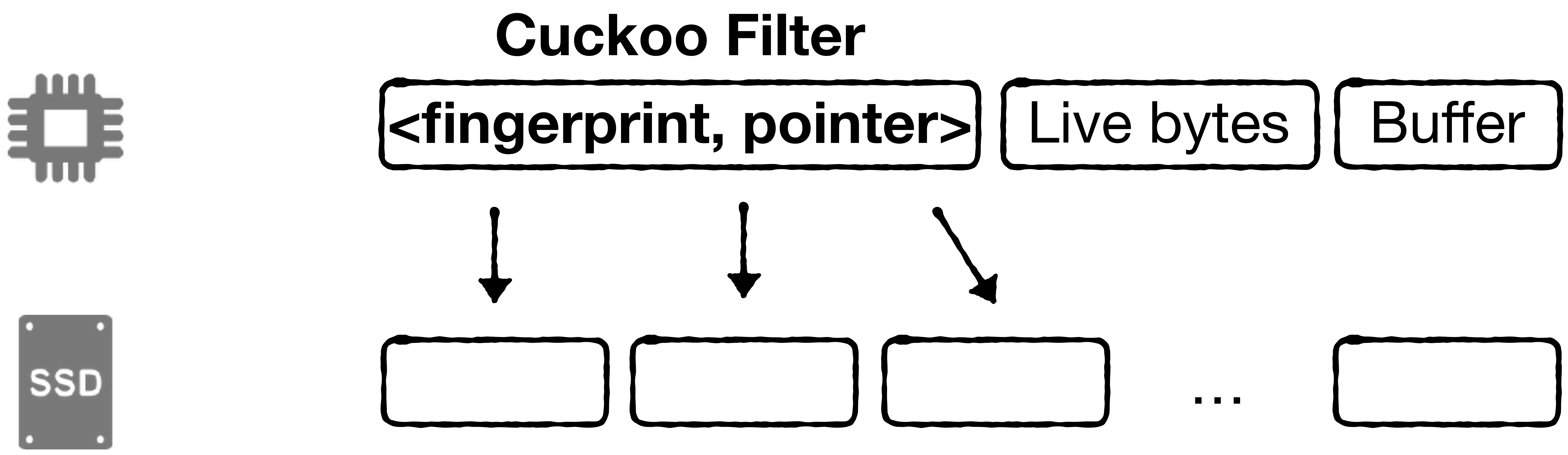
Can we store a payload associated with each key and retrieve it on positive query?

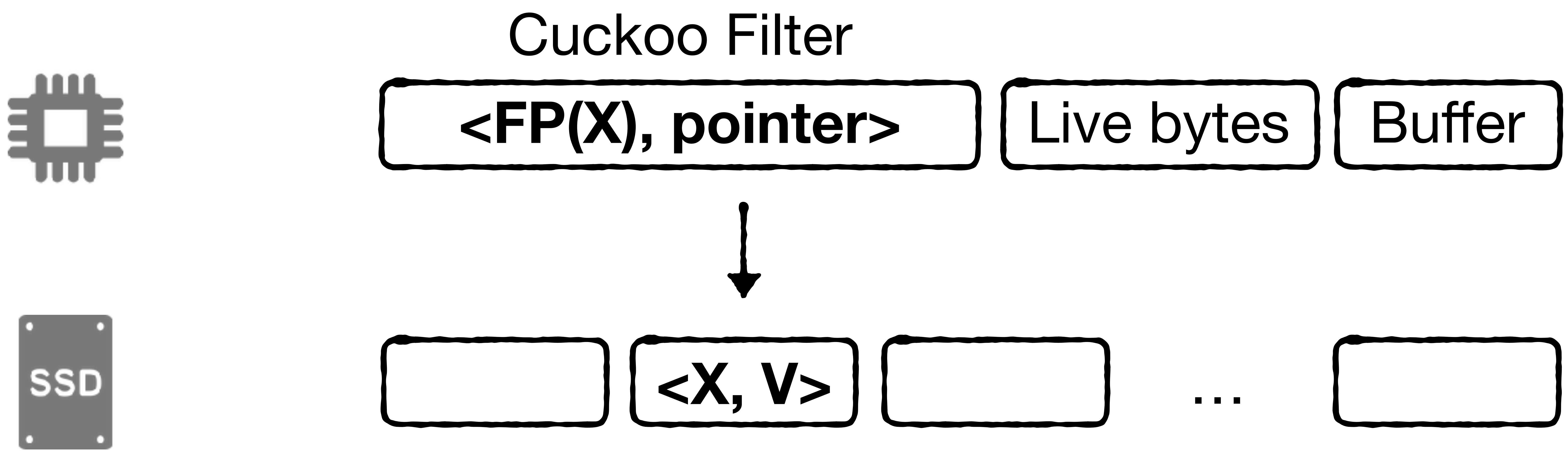
Cuckoo filter:



Just store a payload alongside each fingerprint

	Cuckoo Filter	**Bloom filter**
false positive rate:	$\approx 2^{-M+3}$	$2^{-M \cdot \ln(2)}$
Memory accesses for positive query	1.5	$M \cdot \ln(2)$
Memory accesses for negative query	2	2
Insertion cost	O(1)	$M \cdot \ln(2)$
Deletes?	1.5	N/A
Storing Payloads?	**Yes**	**N/A**


Cuckoo Filter
<fingerprint, pointer>
Live bytes
Buffer
SSD
...



Cuckoo Filter
<FP(X), pointer>
Live bytes
Buffer
SSD
<X, V>
...

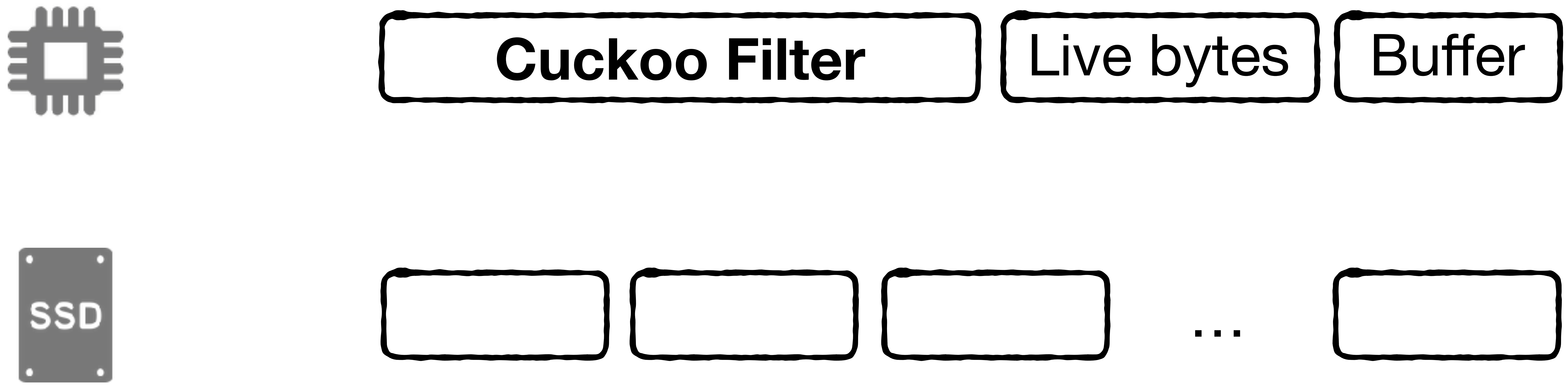
Index size = $N * (P + \mathbf{K}) * \alpha$

N = data size

P = pointer size = $O(\log_2 N/B)$

Our goal was reducing this → $\mathbf{K}$ **= key size =** $\mathbf{\Omega(\log_2 N)}$

α = collision resolution overheads $\approx 0.8 \rightarrow \approx 0.95$



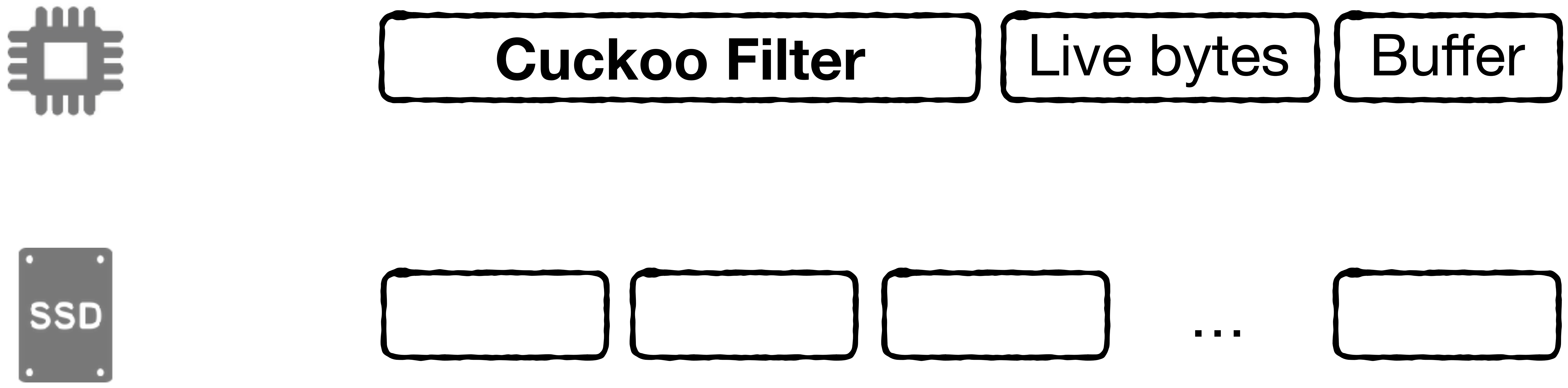
Index size = N * (P + **M**) / α

N = data size

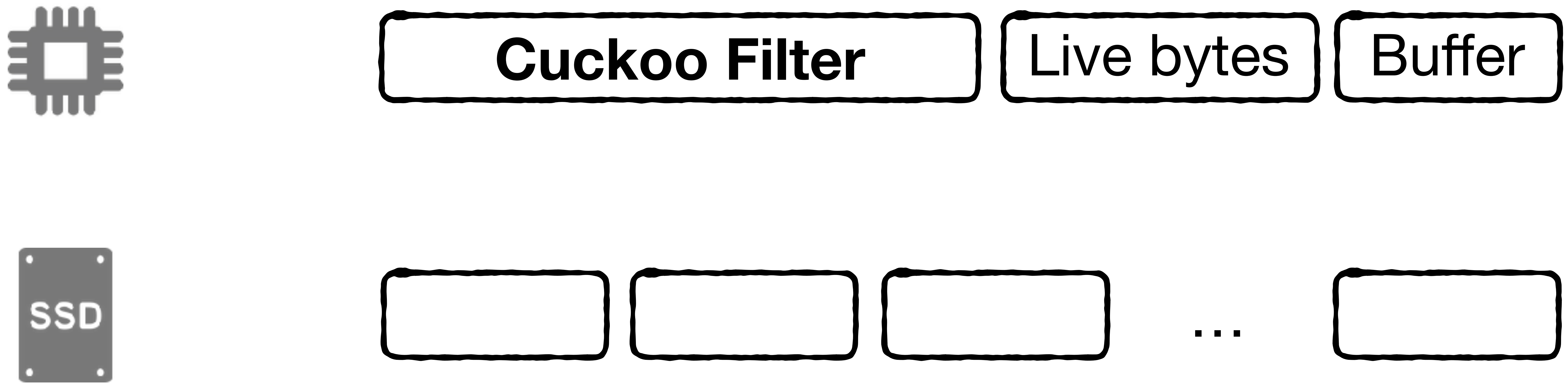
P = pointer size = $O(\log_2 N/B)$

Our goal was reducing this → **K = key size = $\Omega(\log_2 N)$ → M bits / entry**

α = collision resolution overheads ≈0.8 → ≈0.95

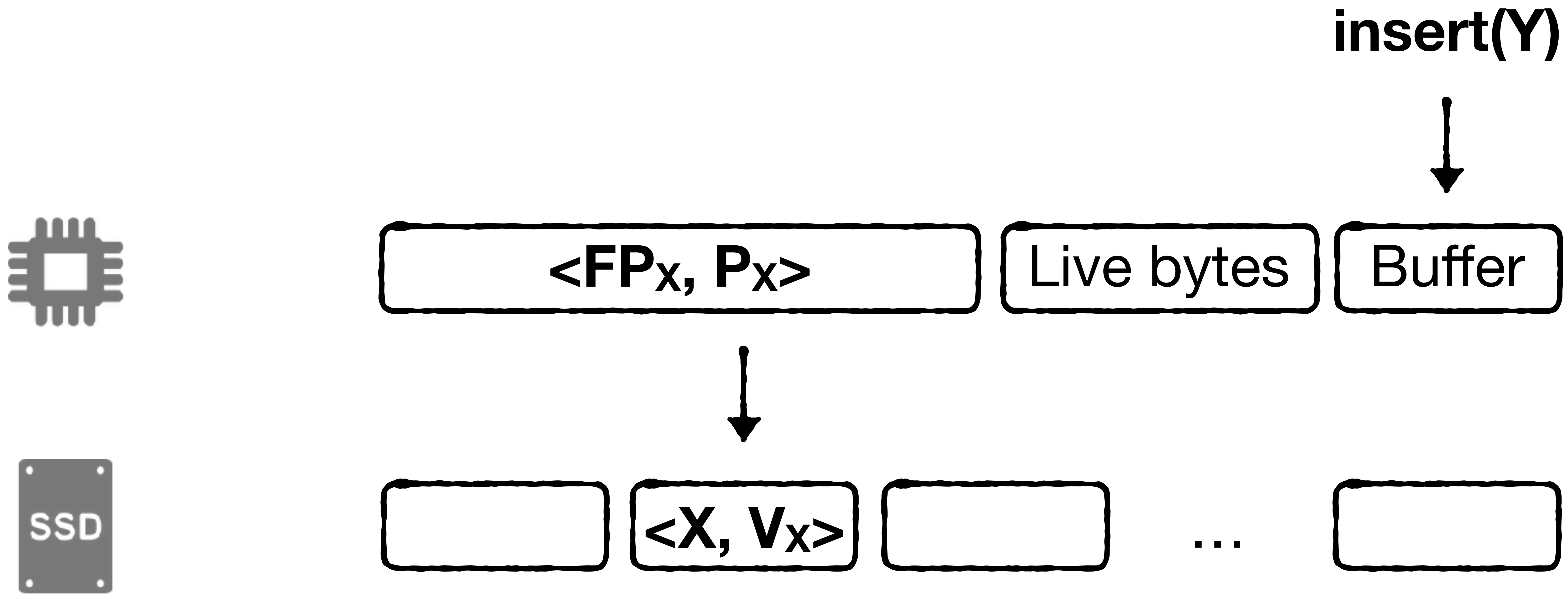


Implication of Using a Filter on Insertions/Deletes/Updates



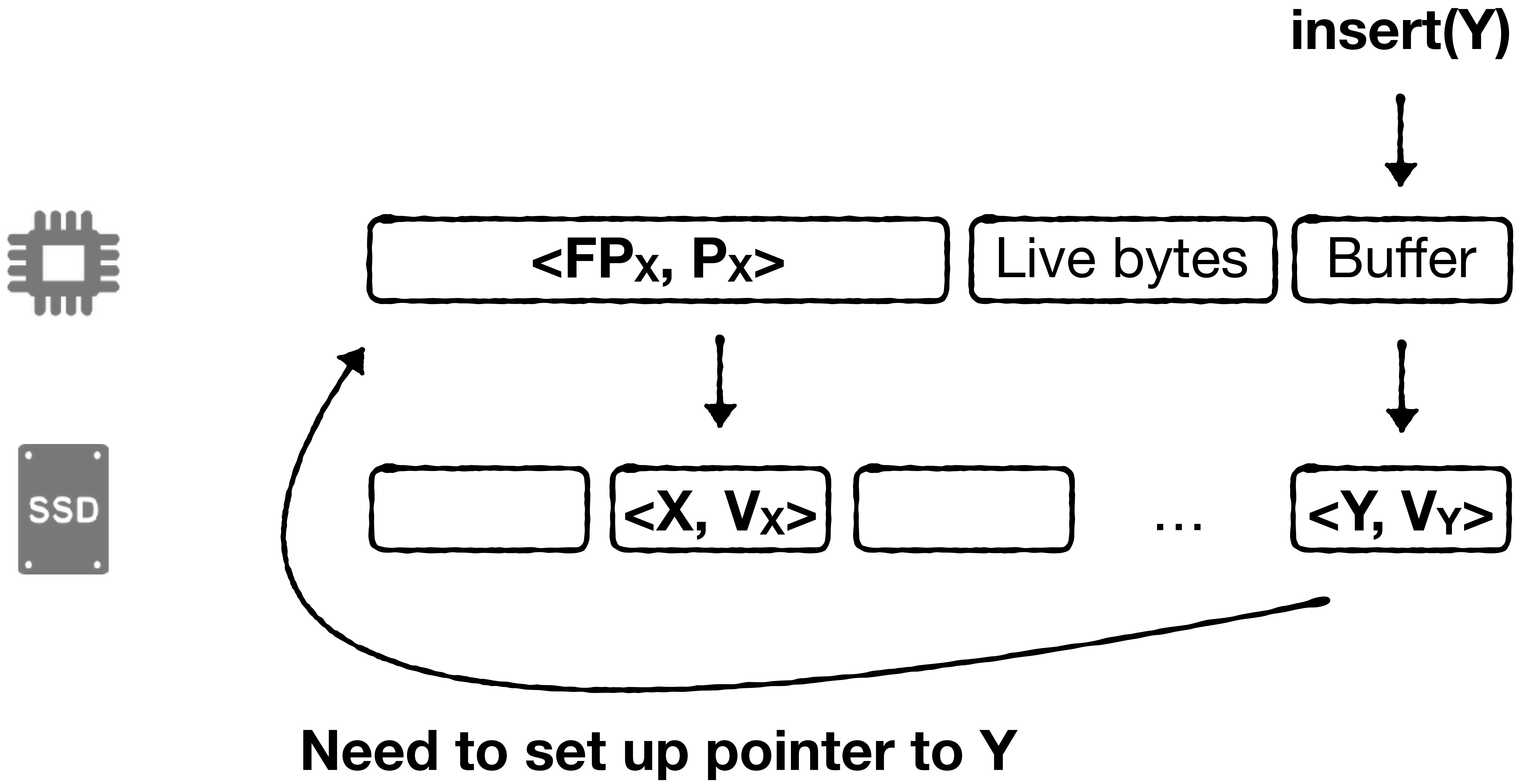
Implication of Using a Filter on Insertions/Deletes/Updates

Suppose we insert key Y that has a matching fingerprint to existing key X (FP_{X}=FP_{y})



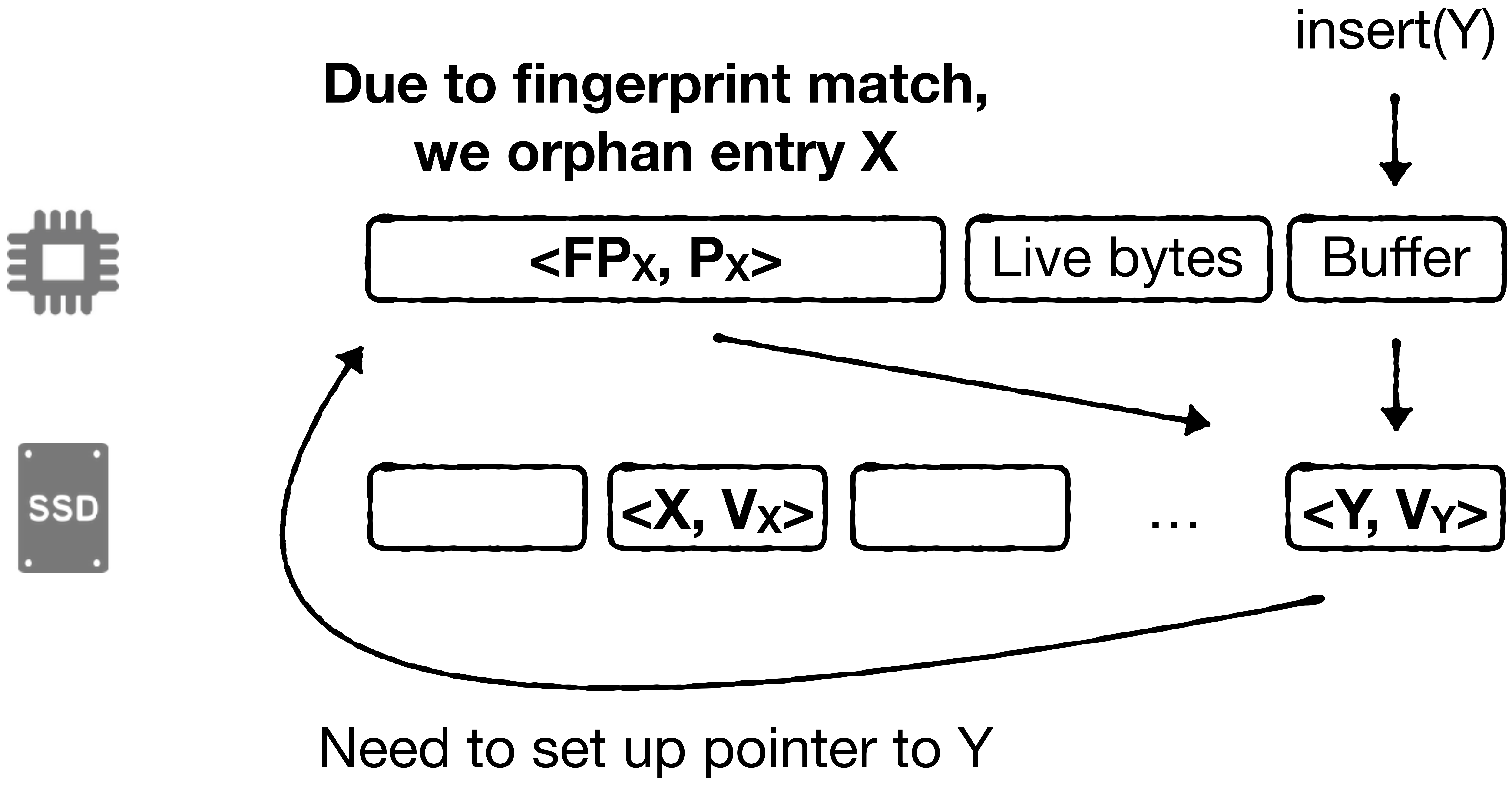
Implication of Using a Filter on Insertions/Deletes/Updates

Suppose we insert key Y that has a matching fingerprint to existing key X



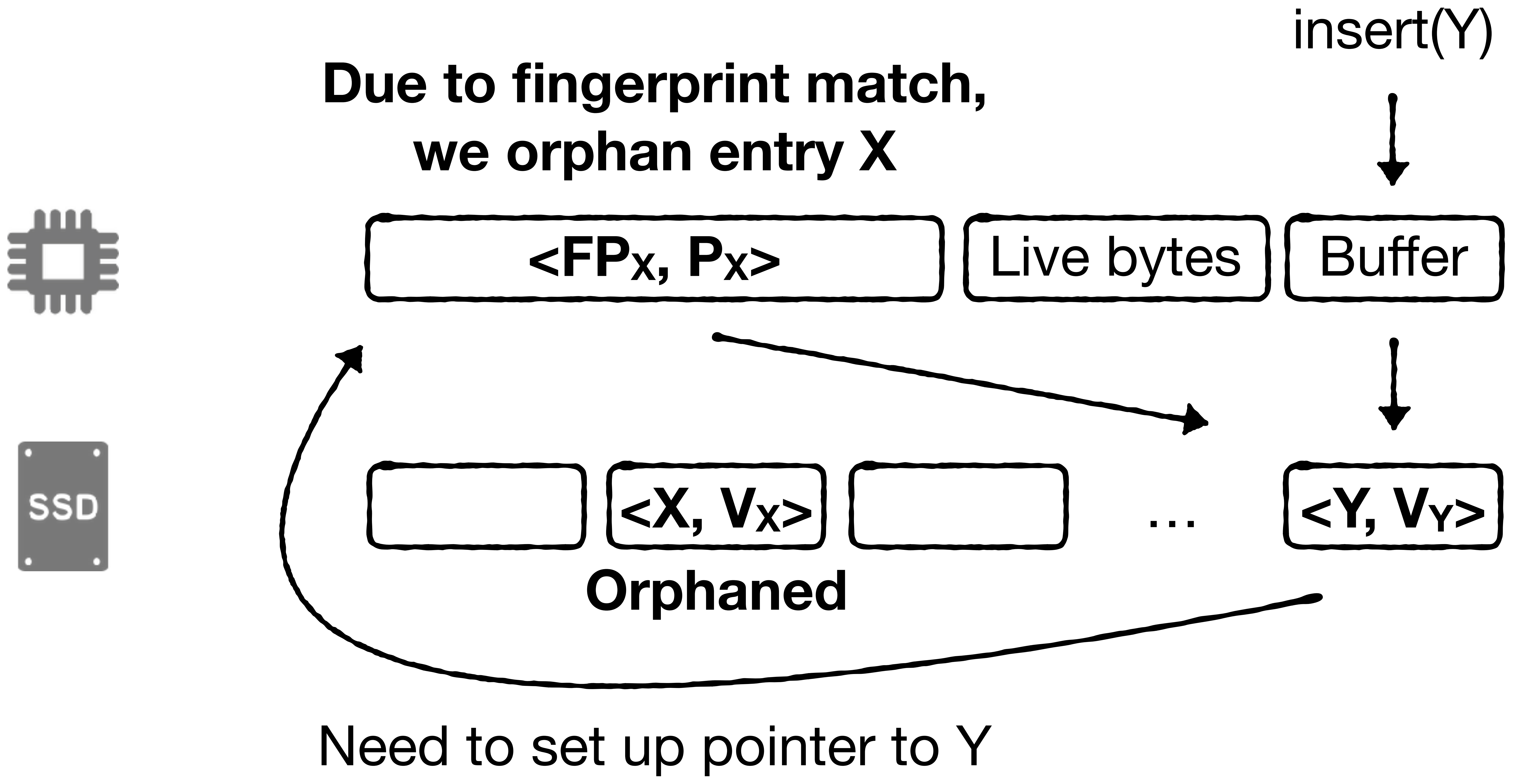
Implication of Using a Filter on Insertions/Deletes/Updates

Suppose we insert key Y that has a matching fingerprint to existing key X



Implication of Using a Filter on Insertions/Deletes/Updates

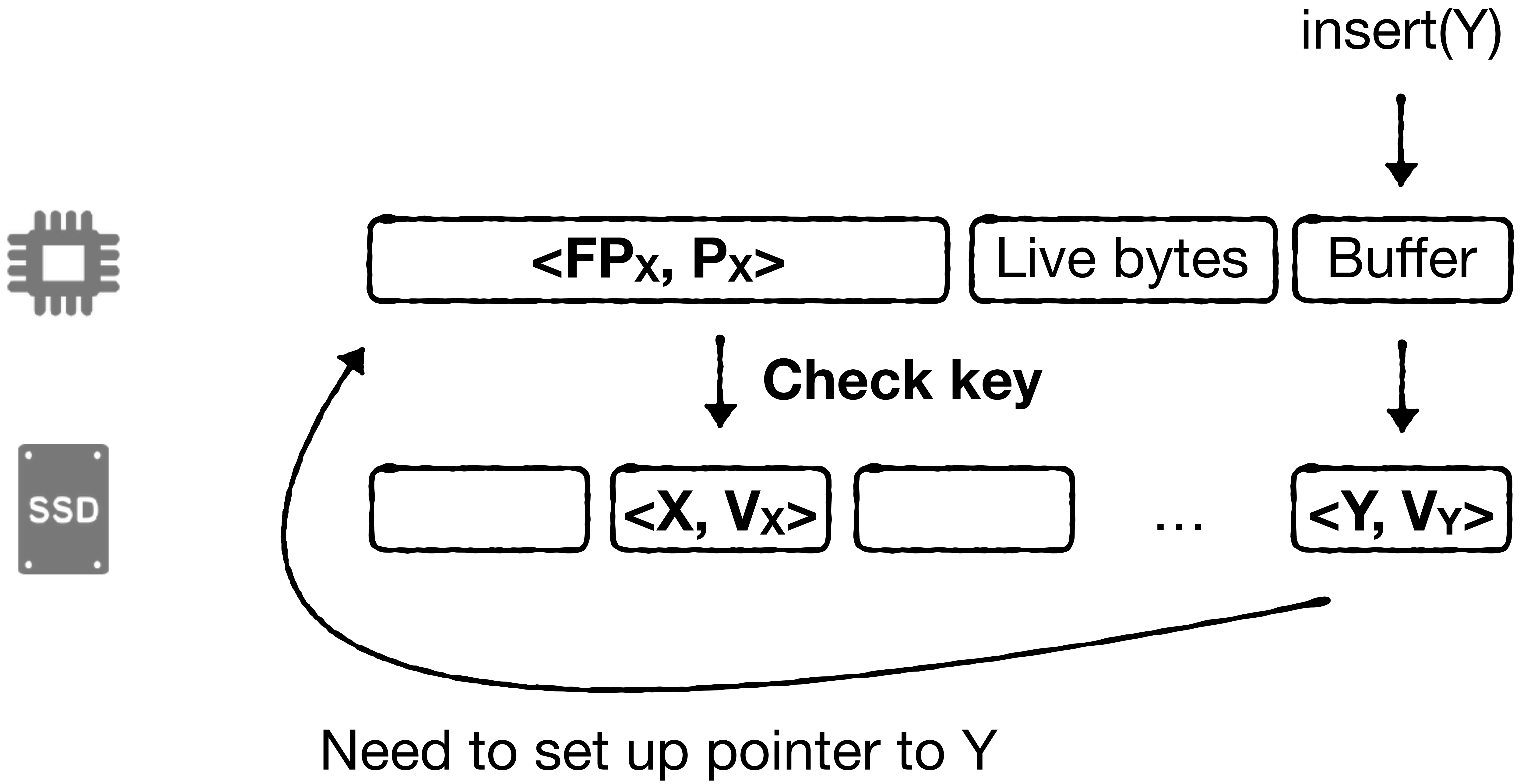
Suppose we insert key Y that has a matching fingerprint to existing key X



Implication of Using a Filter on Insertions/Deletes/Updates

Suppose we insert key Y that has a matching fingerprint to existing key X

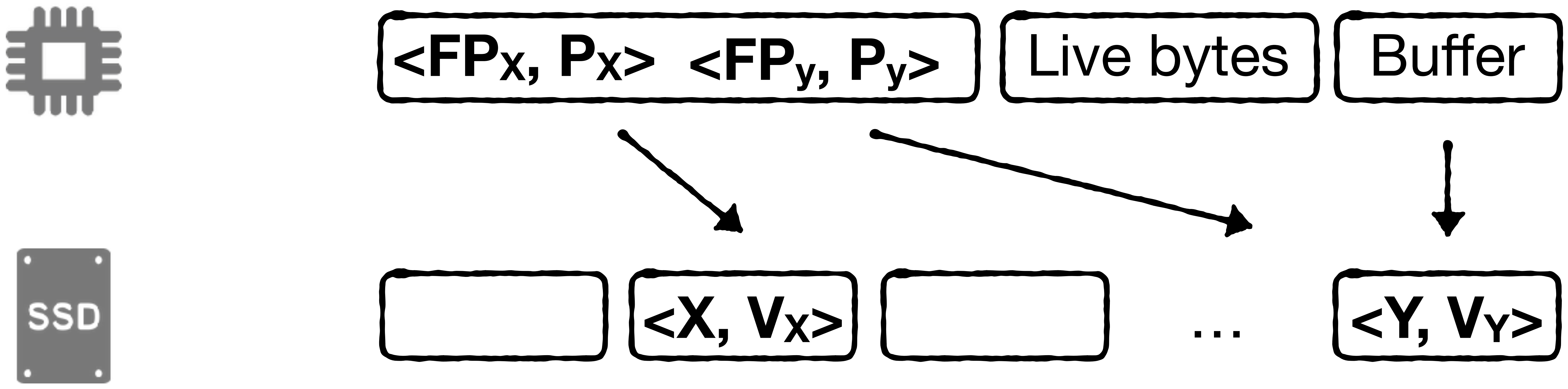
To safeguard against orphaning, we must issue read-before-write



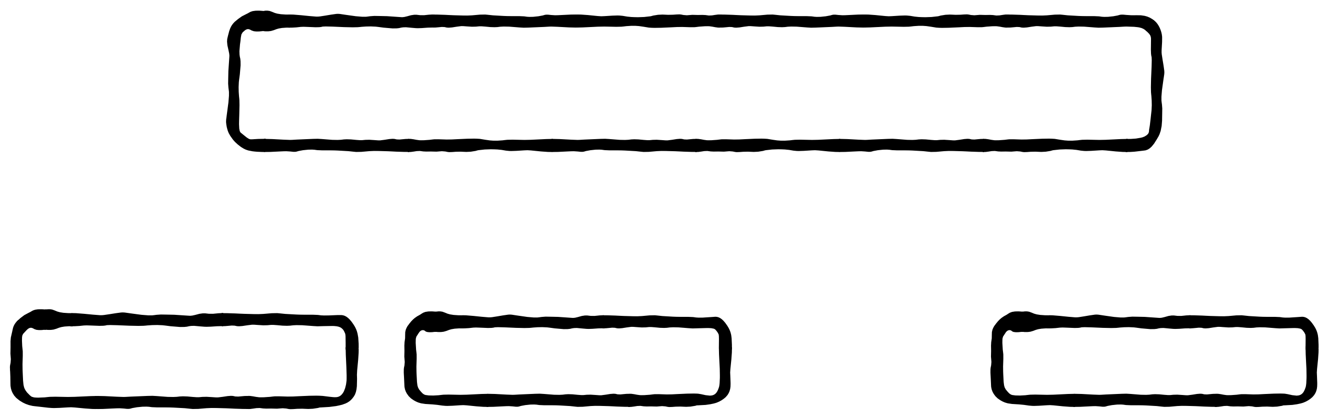
Implication of Using a Filter on Insertions/Deletes/Updates

Suppose we insert key Y that has a matching fingerprint to existing key X

To safeguard against orphaning, we must issue read-before-write



Circular Log w.
Cuckoo Filter



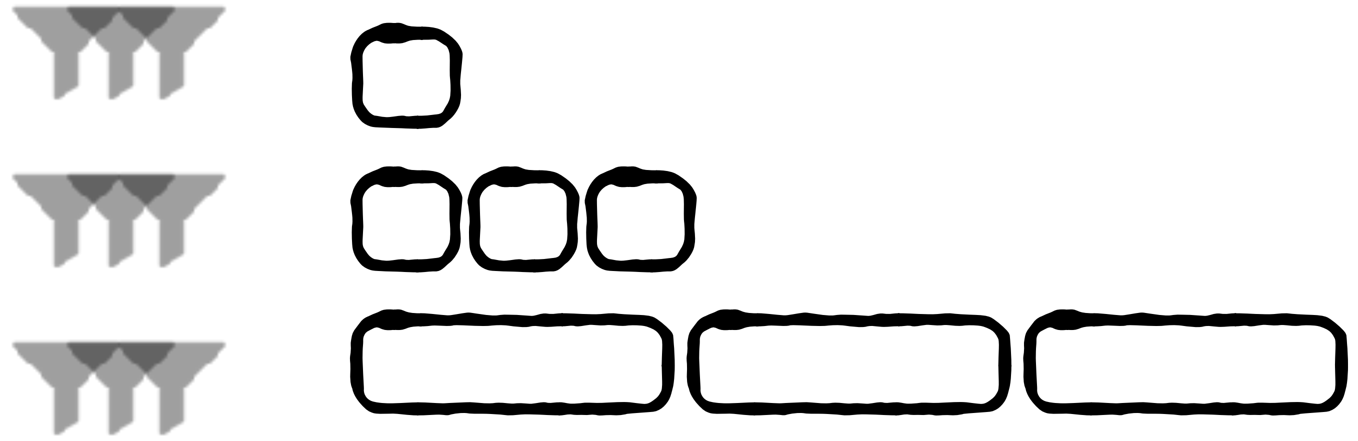
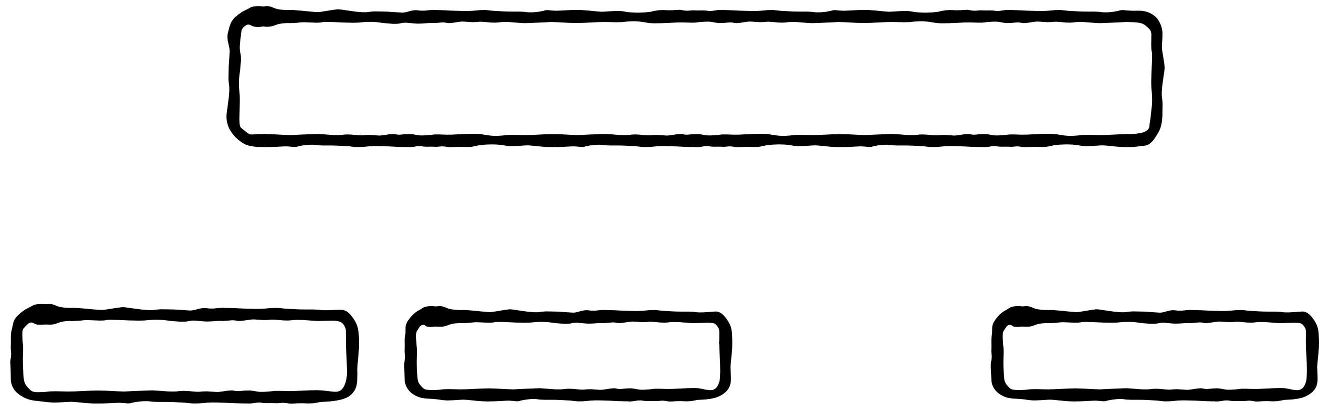
Updates/ Deletes: $O(1+2^{-M+3})$ reads & $O(GC/B)$ writes (excluding checkpointing)

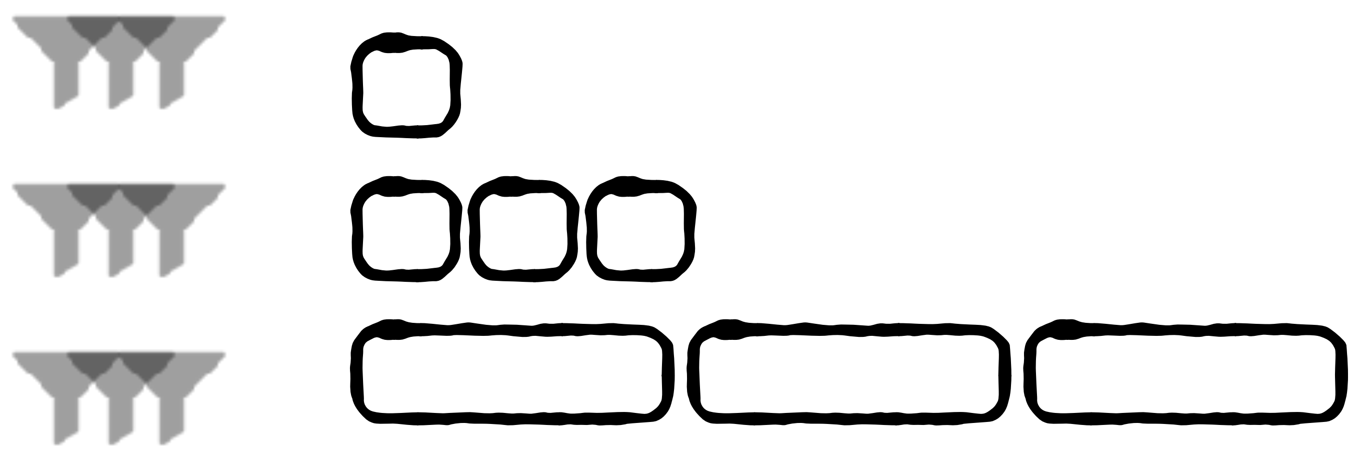
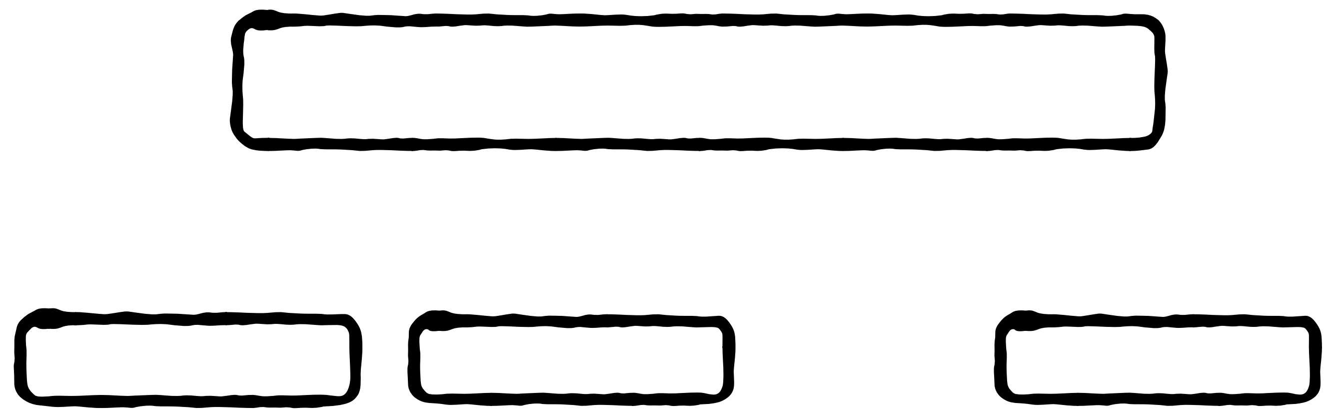
Insert: $O(2^{-M+3})$ read & $O(GC/B)$ writes

Gets: $O(1+2^{-M+3})$

Scan: $O(N/B)$

Memory (bits / entry): $O(\log_2(N/B) + M))$

	Basic LSM-tree w. Monkey	Circular Log w. Cuckoo Filter
		
Updates/ Deletes:	$O(\log_2(N/P) / B)$ reads & writes	$O(1+2^{-M+3})$ reads & $O(GC/B)$ writes
Insert:	$O(\log_2(N/P) / B)$ reads & writes	$O(2^{-M+3})$ read & $O(GC/B)$ writes
Gets:	$O(1+2^{-M * \ln(2)})$	$O(1+2^{-M+3})$
Scan:	$O(\log_2 N/P + S/B)$	$O(N/B)$
Memory (bits / entry)	$O(K/B + M)$	$O(\log_2(N/B) + M))$

	Basic LSM-tree w. Monkey	Circular Log w. Cuckoo Filter
		
Updates/ Deletes:	$O(\log_2(N/P) / B)$ reads & writes	$O(1+2^{-M+3})$ reads & $O(GC/B)$ writes
Insert:	$O(\log_2(N/P) / B)$ reads & writes	$O(2^{-M+3})$ read & $O(GC/B)$ writes
Gets:	$O(1+2^{-M * \ln(2)})$	$O(1+2^{-M+3})$
Scan:	$O(\log_2 N/P + S/B)$	$O(N/B)$
Memory (bits / entry)	$O(K/B + M)$	$O(\log_2(N/B) + M))$
Recovery	**Swift**	**Long**

And now: office hours