COSI 167A
Advanced Data Systems

Class 18

Indexing + Modern Hardware Trends

Prof. Subhadeep Sarkar

Brandeis University

https://ssd-brandeis.github.io/COSI-167A/

lab

Class **logistics**

and administrivia

The **mid-semester project report** is due **today** (**11:59 PM**).

5 weeks remaining until the end of semester. **Use your time wisely**!

Final project report has **2 parts**.

- **Preliminary project report** due on **Dec 3**.
- Followed by **project presentation** (plan for a **15-min presentation**).
- **Final project report** due on **Dec 10**.

Today in COSI 167A

What's on the cards?

summarizing **indexing** techniques

modern **hardware trends**

What is an **index**?

The oracle of DBMSs!

Index

auxiliary data structure that helps find target data **quickly**

typically, **light-weight**, small enough to **fit in memory**

special form of < **key**, **value** >

indexed attribute

position/location/rowID/primary key/…

What are the possible **index designs**?

From B-trees to cracking

index	data organization	remark
B+-tree	Sorted & partitioned	Partition **k-ways** recursively
LSM-tree	Partially sorted	Optimize **inserts**
Radix tree	Radix-based	Partition using **key radix**
Hash index	Hash buckets	Partition by **hashing the key**
Bitmap index	None	Succinct **membership** represention
Zonemap	None	Use **metadata** to skip access
Cracking	Cracked & eventually sorted	**Query-driven** partitioning

What are the possible **index designs**?

From B-trees to cracking

index	data organization	remark
B+-tree	Sorted & partitioned	Partition **k-ways** recursively
LSM-tree	Partially sorted	Optimize **inserts**
Radix tree	Radix-based	Partition using **key radix**
Hash index	Hash buckets	Partition by **hashing the key**
Bitmap index	None	Succinct **membership** represention
Zonemap	None	Use **metadata** to skip access
Cracking	Cracked & eventually sorted	**Query-driven** partitioning

What are the possible **index designs**?

From B-trees to cracking

index	data organization	remark
B+-tree	Sorted & partitioned	Partition **k-ways** recursively
LSM-tree	Partially sorted	Optimize **inserts**
Radix tree	Radix-based	Partition using **key radix**
Hash index	Hash buckets	Partition by **hashing the key**
Bitmap index	None	Succinct **membership** represention
Zonemap	None	Use **metadata** to skip access
Cracking	Cracked & eventually sorted	**Query-driven** partitioning

What are the possible **index designs**?

From B-trees to cracking

index	data organization	remark
B+-tree	Sorted & partitioned	Partition **k-ways** recursively
LSM-tree	Partially sorted	Optimize **inserts**
Radix tree	Radix-based	Partition using **key radix**
Hash index	Hash buckets	Partition by **hashing the key**
Bitmap index	None	Succinct **membership** represention
Zonemap	None	Use **metadata** to skip access
Cracking	Cracked & eventually sorted	**Query-driven** partitioning

What are the possible **index designs**?

From B-trees to cracking

index	data organization	remark
B+-tree	Sorted & partitioned	Partition **k-ways** recursively
LSM-tree	Partially sorted	Optimize **inserts**
Radix tree	Radix-based	Partition using **key radix**
Hash index	Hash buckets	Partition by **hashing the key**
Bitmap index	None	Succinct **membership** represention
Zonemap	None	Use **metadata** to skip access
Cracking	Cracked & eventually sorted	**Query-driven** partitioning

What are the possible **index designs**?

From B-trees to cracking

index	data organization	remark
B+-tree	Sorted & partitioned	Partition **k-ways** recursively
LSM-tree	Partially sorted	Optimize **inserts**
Radix tree	Radix-based	Partition using **key radix**
Hash index	Hash buckets	Partition by **hashing the key**
Bitmap index	None	Succinct **membership** representation
Zonemap	None	Use **metadata** to skip access
Cracking	Cracked & eventually sorted	**Query-driven** partitioning

What are the possible **index designs**?

From B-trees to cracking

index	data organization	remark
B+-tree	Sorted & partitioned	Partition **k-ways** recursively
LSM-tree	Partially sorted	Optimize **inserts**
Radix tree	Radix-based	Partition using **key radix**
Hash index	Hash buckets	Partition by **hashing the key**
Bitmap index	None	Succinct **membership** represention
Zonemap	None	Use **metadata** to skip access
Cracking	Cracked & eventually sorted	**Query-driven** partitioning

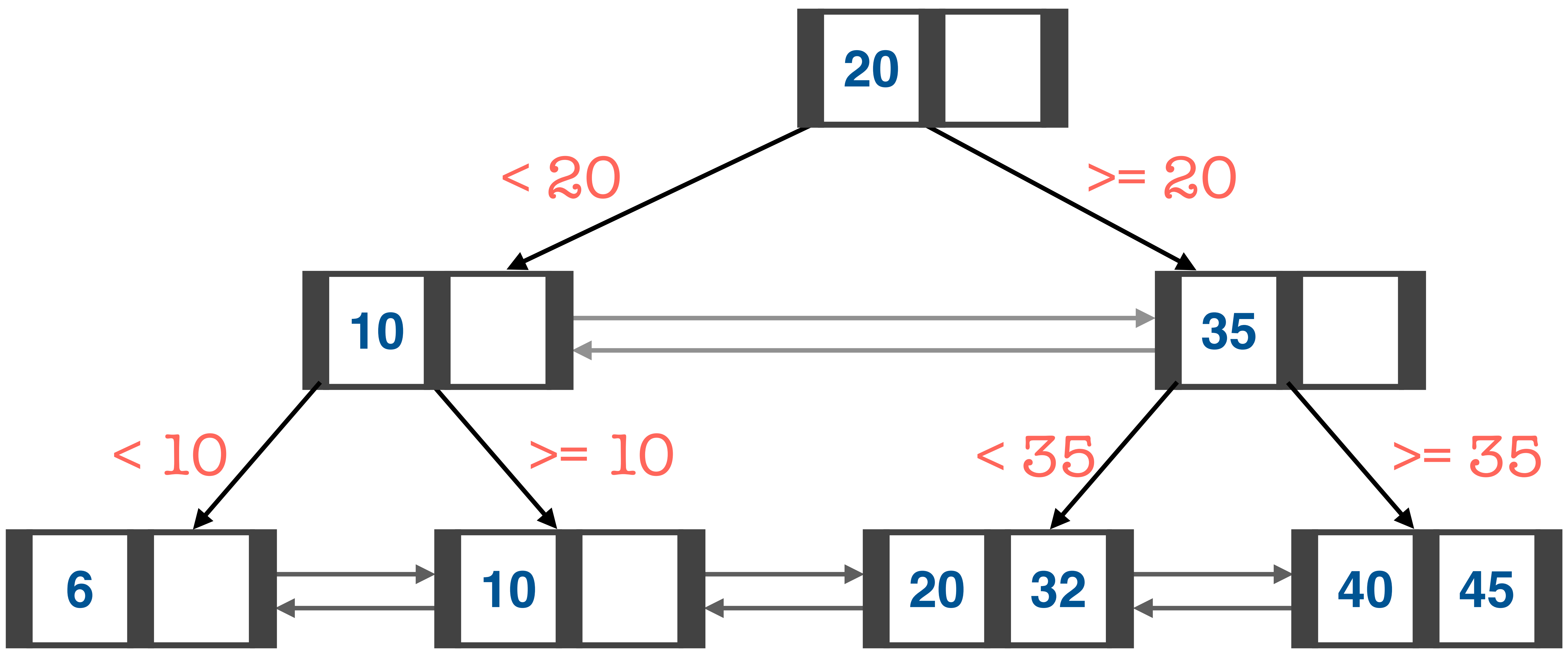
What are the possible **index designs**?

From B-trees to cracking

index	**point** queries	**short range** queries	**long range** queries	data **skew**	**updates**
B+-tree					
LSM-tree					
Radix tree					
Hash index					
Bitmap index					
Zonemap					
Cracking					

B⁺-tree

The most popular index data structure



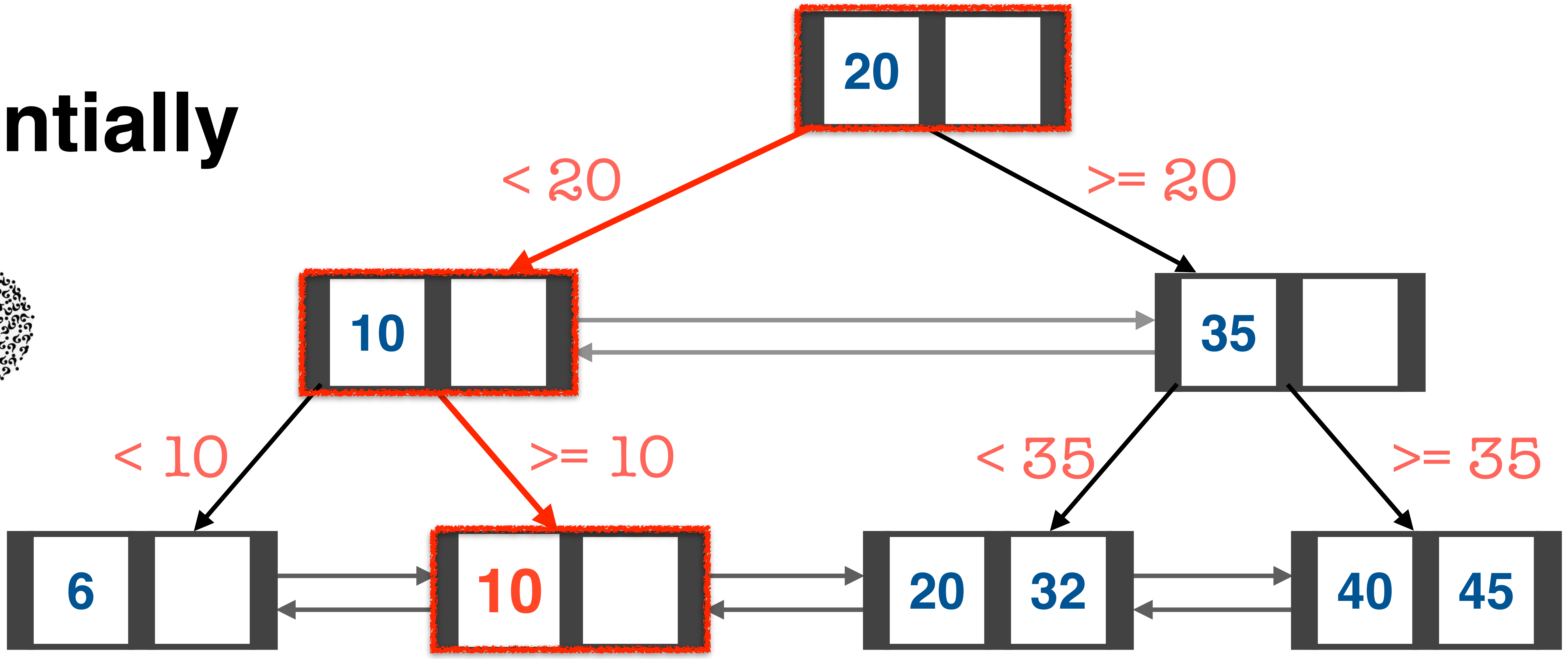
B^+-tree

The most popular index data structure

Search begins at root, and key comparisons **direct it to a leaf**

Point lookups are super-**efficient**

Range lookups can scan **sequentially**



Thought Experiment 1

What about **skewed data**?

It does well!

What are the possible **index designs**?

From B-trees to cracking

index	**point** queries	**short range** queries	**long range** queries	data **skew**	**updates**
B^+-tree	✓	✓	✓	✓	−
LSM-tree					
Radix tree					
Hash index					
Bitmap index					
Zonemap					
Cracking					

What are the possible **index designs**?

From B-trees to cracking

index	**point** queries	**short range** queries	**long range** queries	data **skew**	**updates**
B+-tree	✓	✓	✓	✓	–
LSM-tree	✓	✗	–	✓	✓
Radix tree					
Hash index					
Bitmap index					
Zonemap					
Cracking					

Thought Experiment 2

What about **growing data size**?

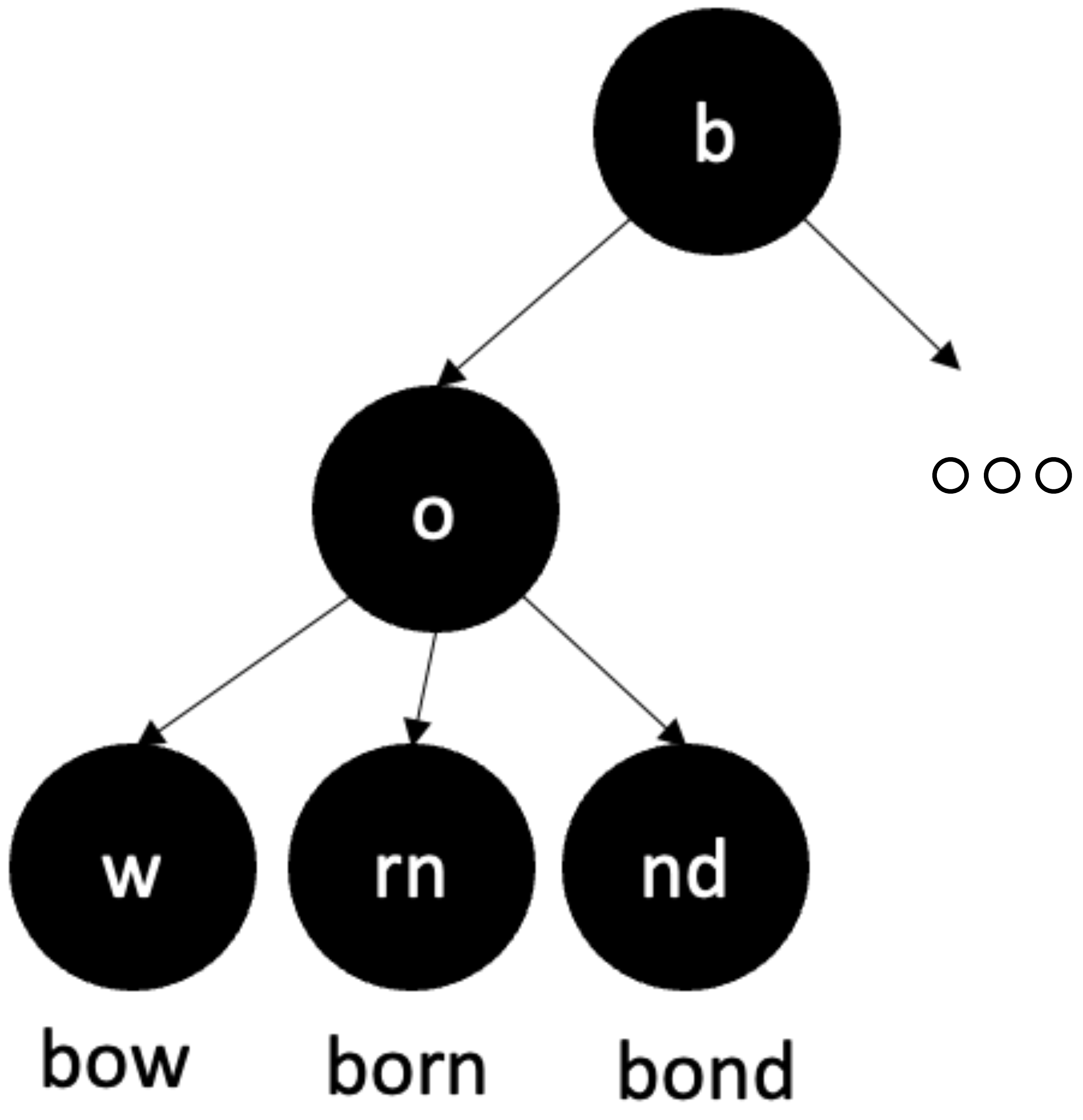
tree grows & so do costs!

Radix trees

A special case of tries and prefix B-trees

Idea: use **common prefixes** for internal nodes to **reduce size/height**!

max. tree hieght = **length of the longest key**

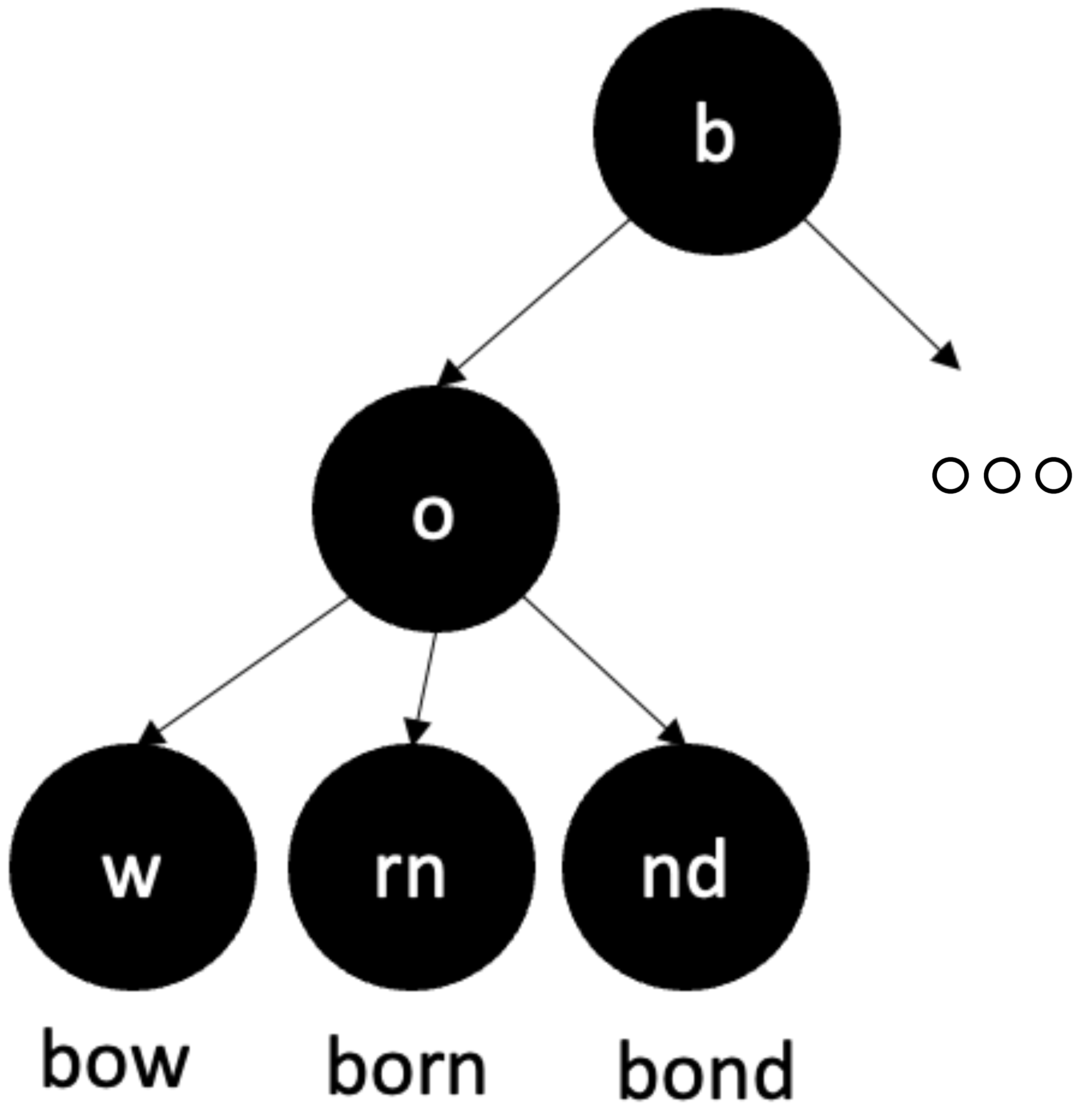


what about **integer keys**?

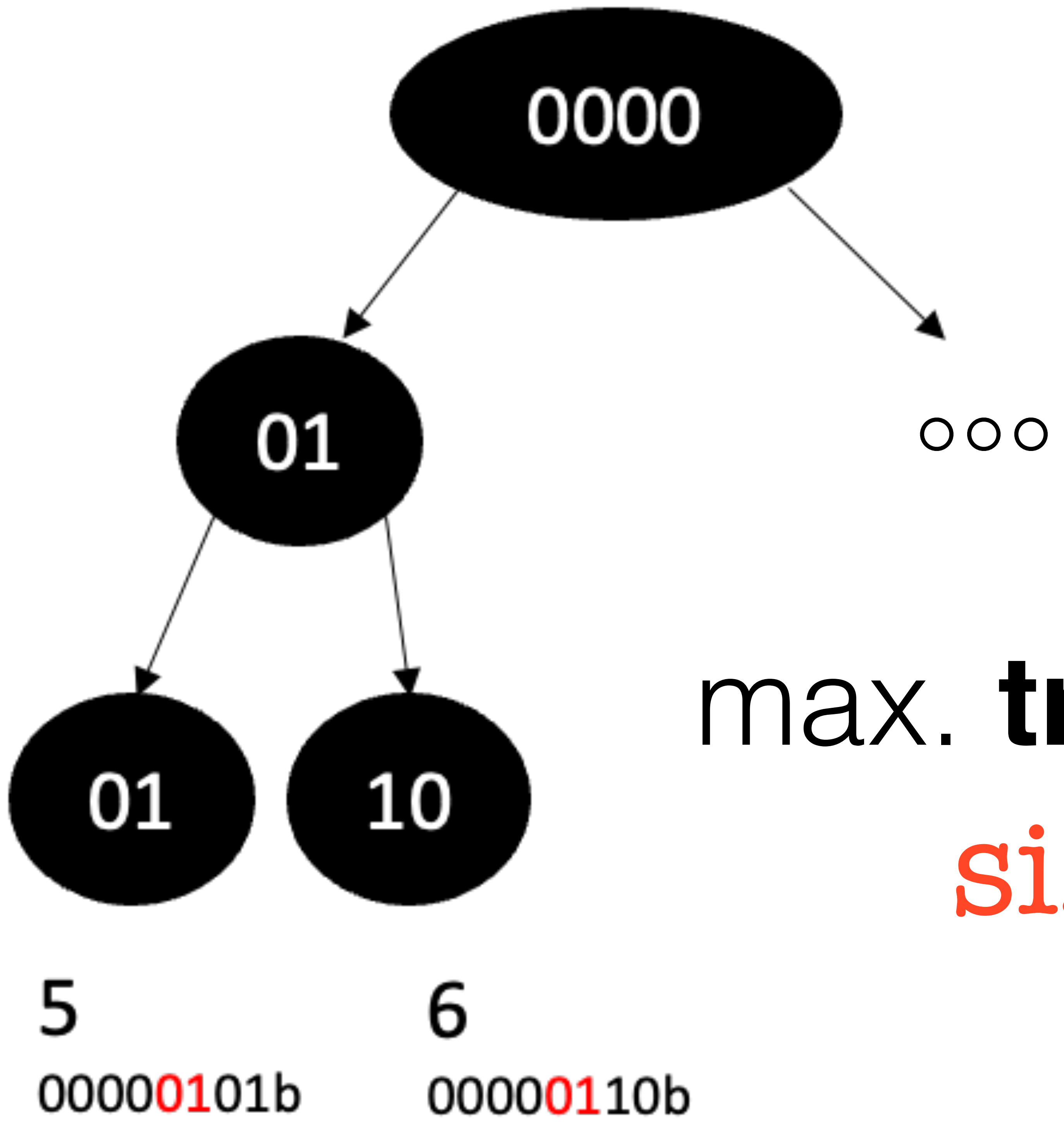
Radix trees

A special case of tries and prefix B-trees

Idea: use **common prefixes** for internal nodes to **reduce size/height**!



max. tree hieght = **length of the longest key**



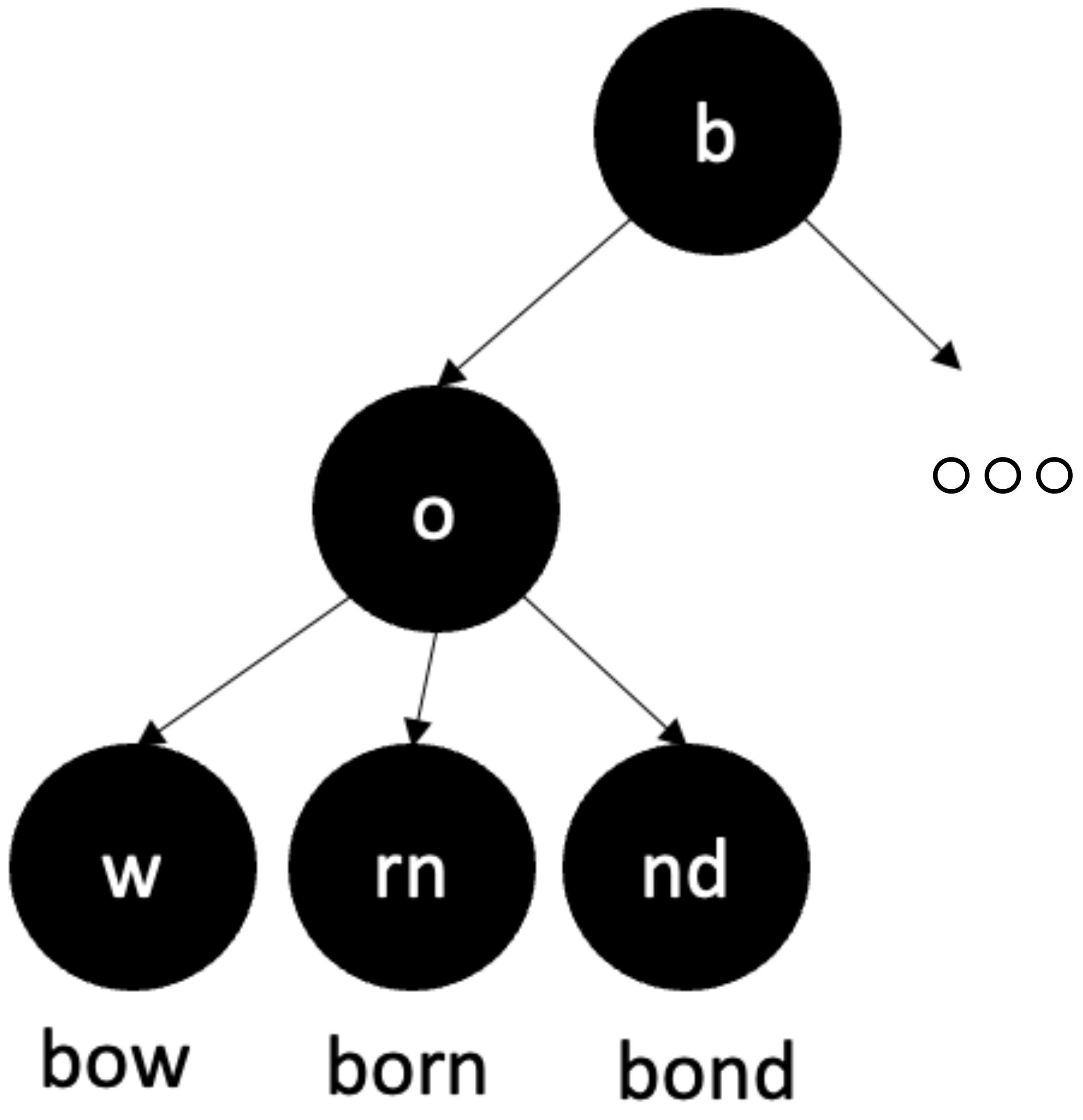
max. **tree hieght**?

size of integer

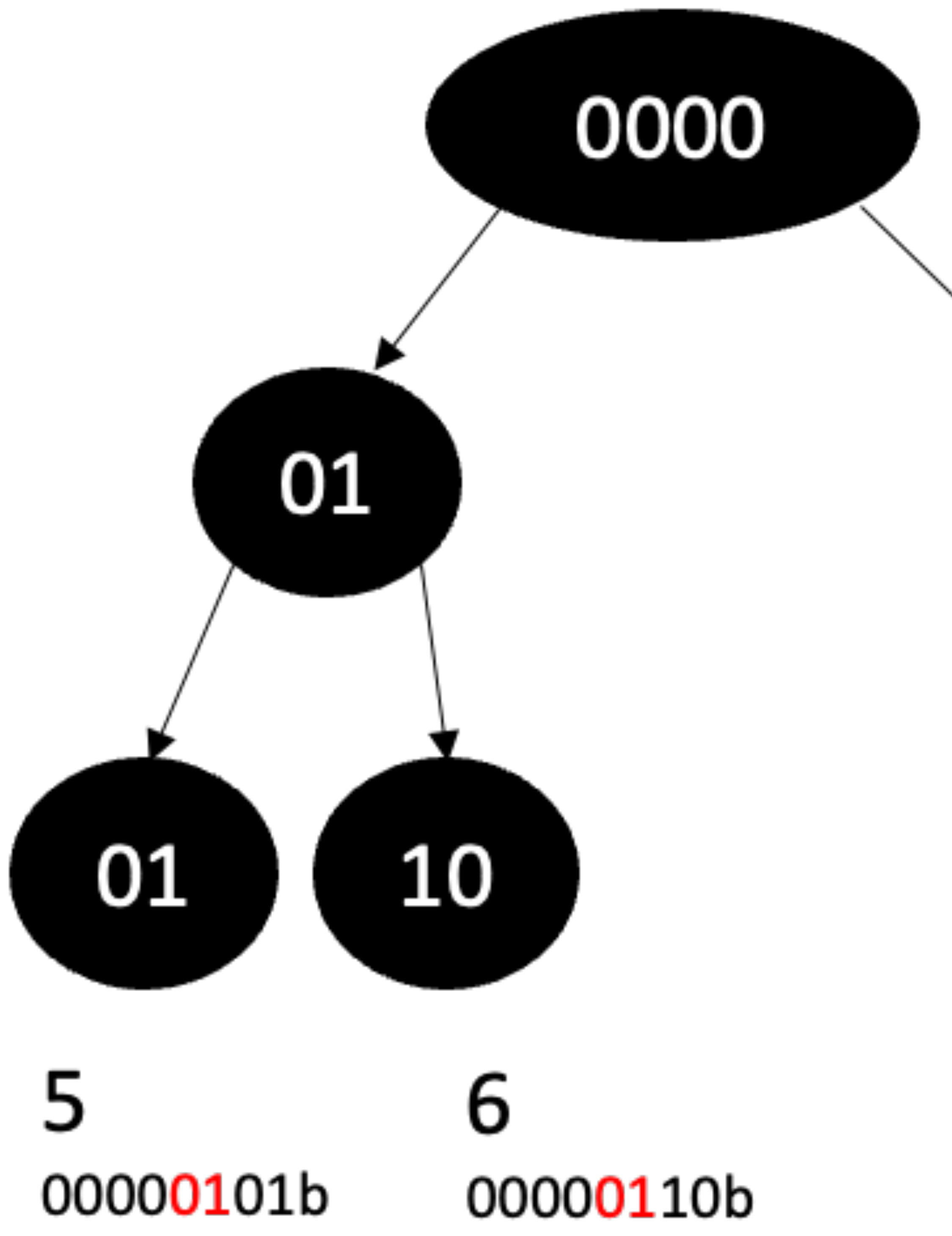
Radix trees

A special case of tries and prefix B-trees

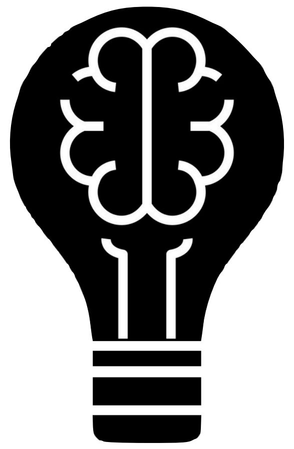
Idea: use **common prefixes** for internal nodes to **reduce size/height**!



max. tree hieght
=
length of the longest key



max. tree hieght
=
size of integer



Thought Experiment 3

What about **data skew**?

radix trees perform poorly!

What are the possible **index designs**?

From B-trees to cracking

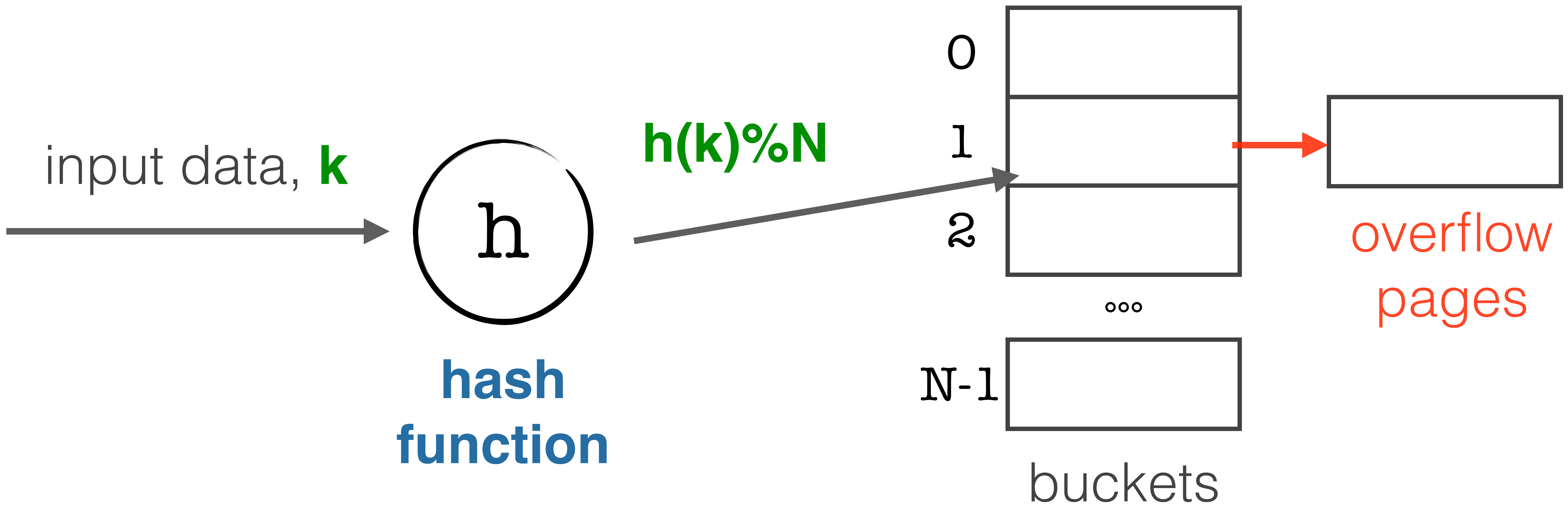
index	**point** queries	**short range** queries	**long range** queries	data **skew**	**updates**
B^+-tree	✓	✓	✓	✓	–
LSM-tree	✓	✗	–	✓	✓
Radix tree	✓	✓	✓	✗	–
Hash index					
Bitmap index					
Zonemap					
Cracking					

Hash indexes

Using fast CPU cycles to our advantage

Idea: a function to map **a larger (infinite) space to a smaller finite space**

an **ideal** hash function would **distribute keys uniformly**



What are the possible **index designs**?

From B-trees to cracking

index	**point** queries	**short range** queries	**long range** queries	data **skew**	**updates**
B^+-tree	✓	✓	✓	✓	−
LSM-tree	✓	✗	−	✓	✓
Radix tree	✓	✓	✓	✗	−
Hash index	✓	−	✗	✗	✓
Bitmap index					
Zonemap					
Cracking					

Bitmap index

Fast, light-weight but with limited applicability

Use case: **few distinct values repeating severally**

Column A
30
20
30
10
20
10
30
20

Bitmap index

Fast, light-weight but with limited applicability

Use case: **few distinct values repeating severally**

Column A	A=10	A=20	A=30
30	0	0	1
20	0	1	0
30	0	0	0
10	1	0	0
20	0	1	0
10	1	0	0
30	0	0	1
20	0	1	0

Bitmap index

Fast, light-weight but with limited applicability

Use case: **few distinct values** **repeating severally**

Column A	A=10	A=20	A=30
30	0	0	1
20	0	1	0
30	0	0	0
10	1	0	0
20	0	1	0
10	1	0	0
30	0	0	1
20	0	1	0

Advantages:

Bitmap index

Fast, light-weight but with limited applicability

Use case: **few distinct values** **repeating severally**

Column A	A=10	A=20	A=30
30	0	0	1
20	0	1	0
30	0	0	0
10	1	0	0
20	0	1	0
10	1	0	0
30	0	0	1
20	0	1	0

Advantages:

speed & size

- **compact representation** of query result
- query result is **readily available**

bitvectors

- fast **Boolean operators** (AND/OR/NOT)
- bitwise ops faster than looping over metadata

Limitations:

Bitmap index

Fast, light-weight but with limited applicability

Use case: **few distinct values repeating severally**

Column A	A=10	A=20	A=30
30	0	0	1
20	0	1	0
30	0	0	0
10	1	0	0
20	0	1	0
10	1	0	0
30	0	0	1
20	0	1	0

Limitations:

index size — **space-inefficient** for domains with **large cardinality**

imagine column A has **100M** entries

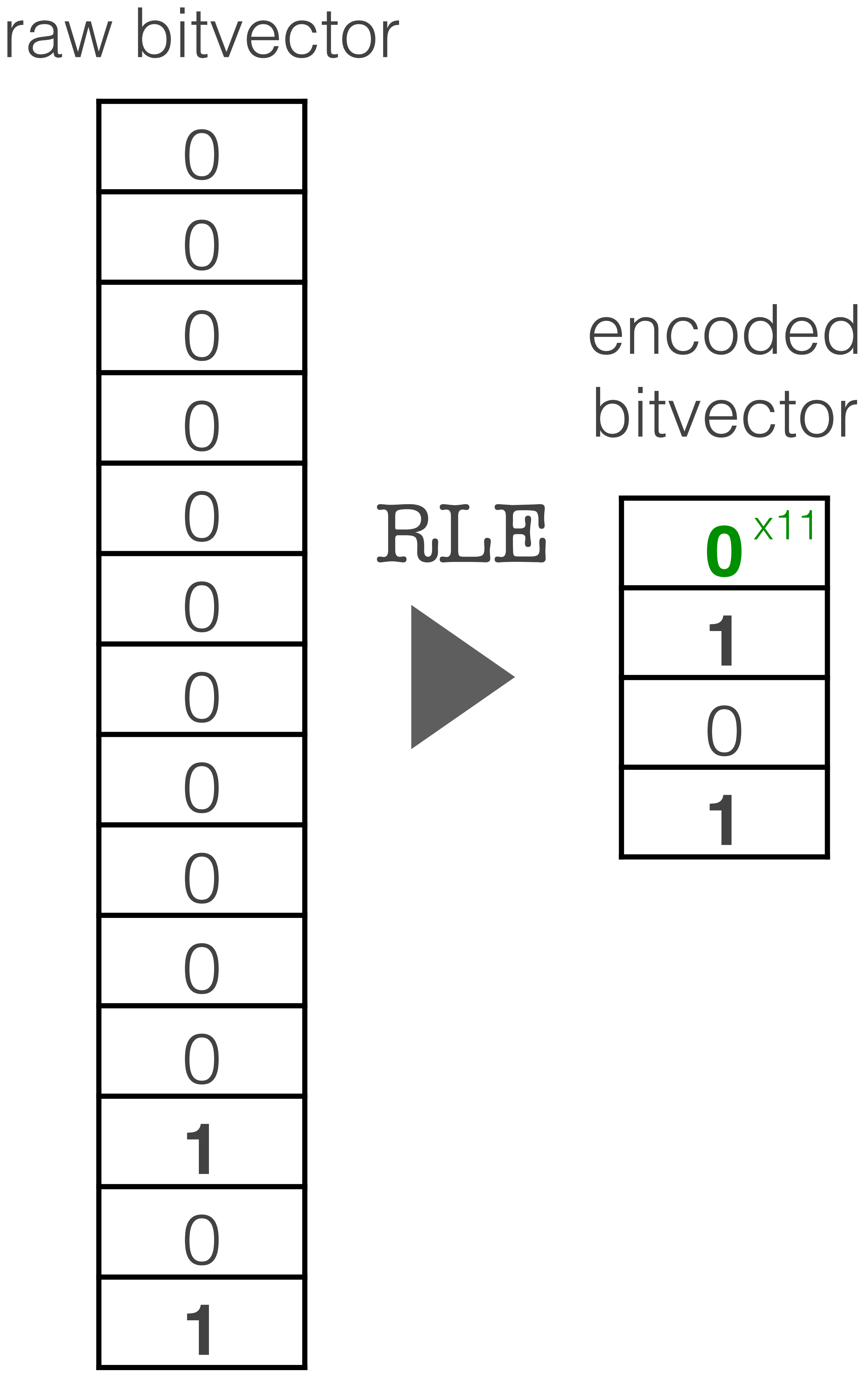
index size = **12.5 MB per distinct value**

solution?

run length encoding

Bitmap index

Fast, light-weight but with limited applicability



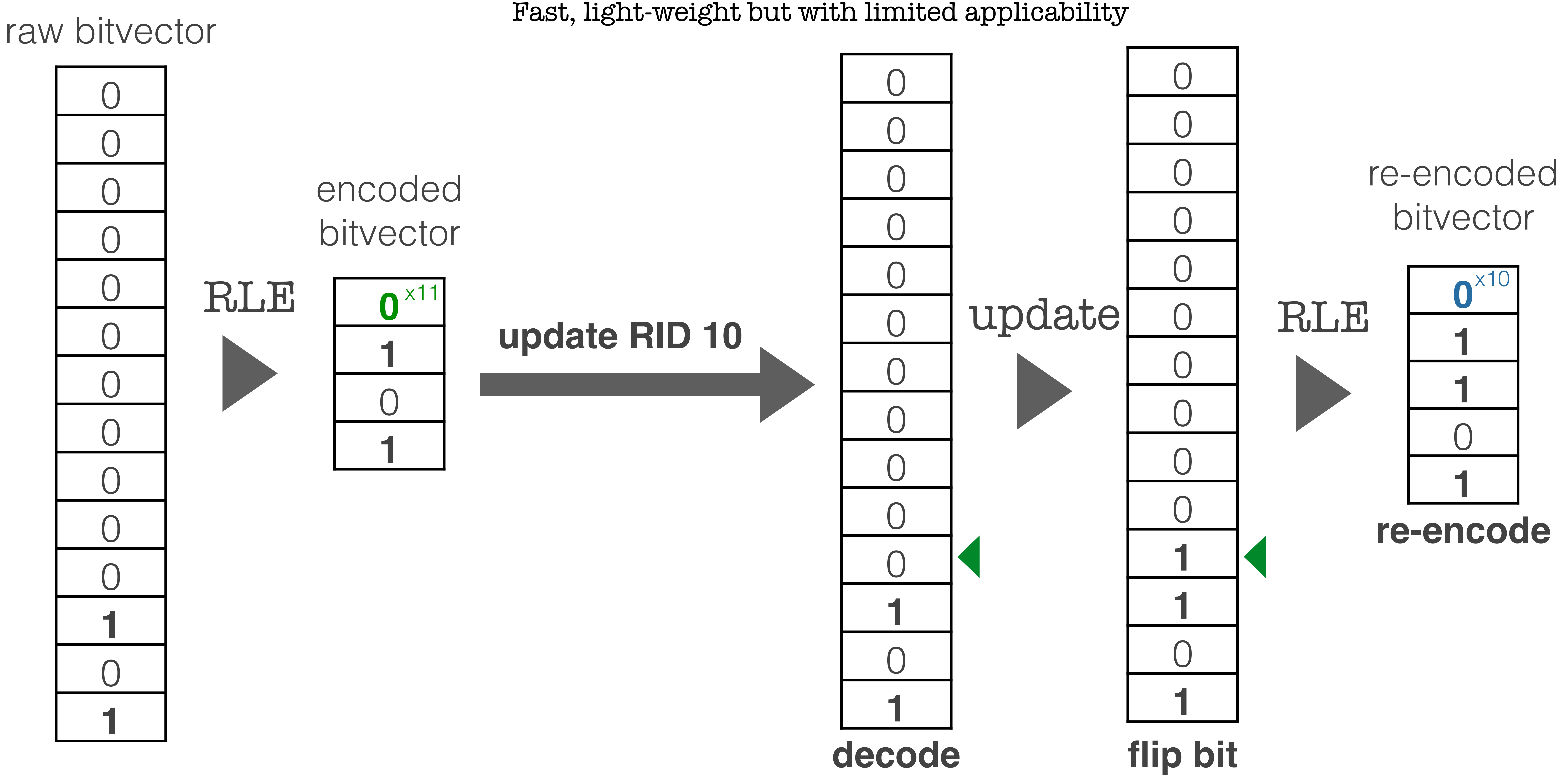
Thought Experiment 4

What about **updates**?

decompressing and re-compressing

Bitmap index

Fast, light-weight but with limited applicability



What are the possible **index designs**?

From B-trees to cracking

index	**point** queries	**short range** queries	**long range** queries	data **skew**	**updates**
B^+-tree	✓	✓	✓	✓	–
LSM-tree	✓	✗	–	✓	✓
Radix tree	✓	✓	✓	✗	–
Hash index	✓	–	✗	✗	✓
Bitmap index	✓	–	–	✓	✗
Zonemap					
Cracking					

What are the possible **index designs**?

From B-trees to cracking

index	**point** queries	**short range** queries	**long range** queries	data **skew**	**updates**
B^+-tree	✓	✓	✓	✓	–
LSM-tree	✓	✗	–	✓	✓
Radix tree	✓	✓	✓	✗	–
Hash index	✓	–	✗	✗	✓
Bitmap index	✓	–	–	–	✗
Zonemap	–	–	✓	✓	–
Cracking					

Cracking

Indexing on the fly

Idea: take **hints from queries** to **create partitions**

gradually moving toward a **sorted layout**

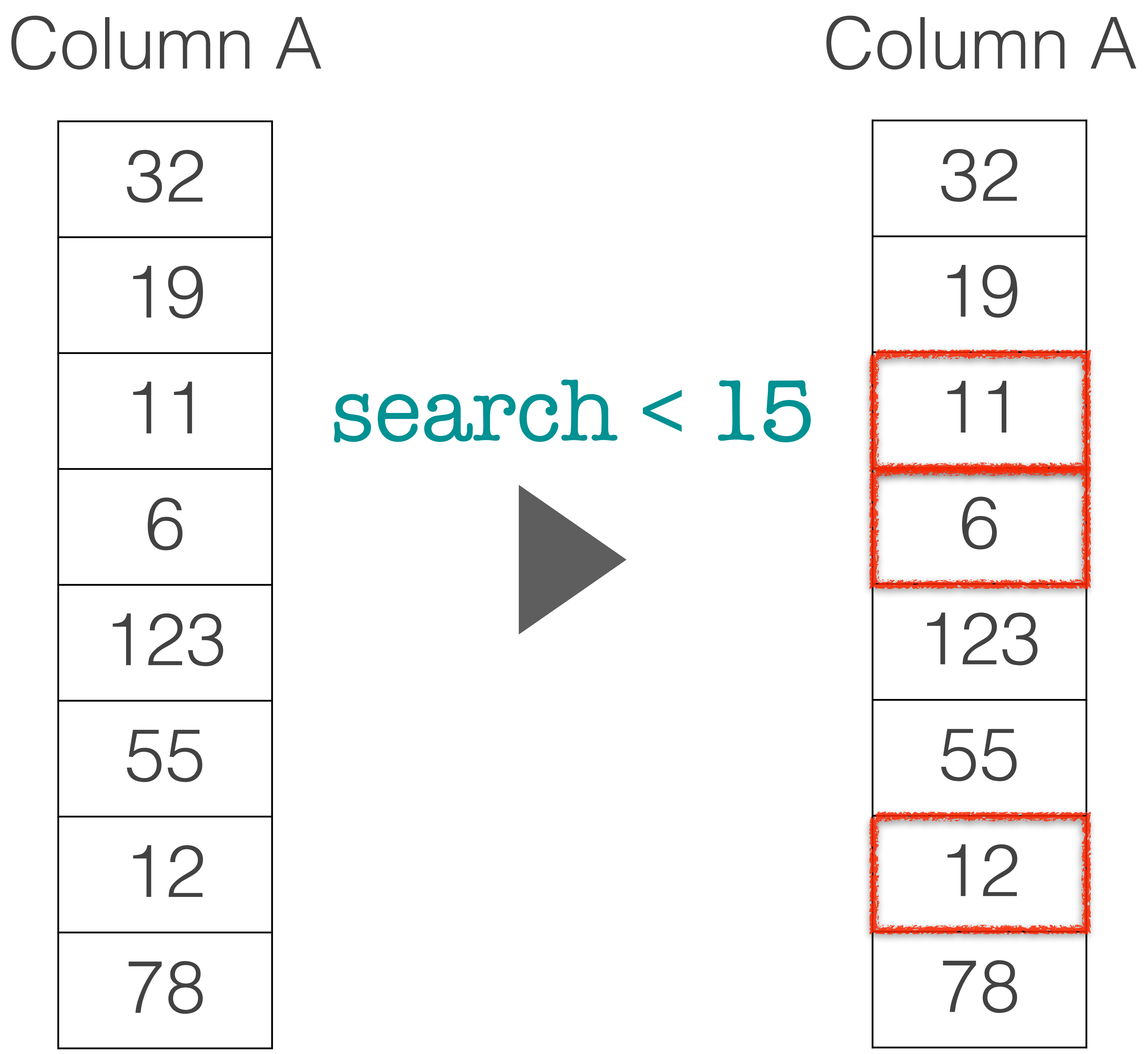
Column A

32
19
11
6
123
55
12
78

Cracking

Indexing on the fly

Idea: take **hints from queries** to **create partitions**
gradually moving toward a **sorted layout**

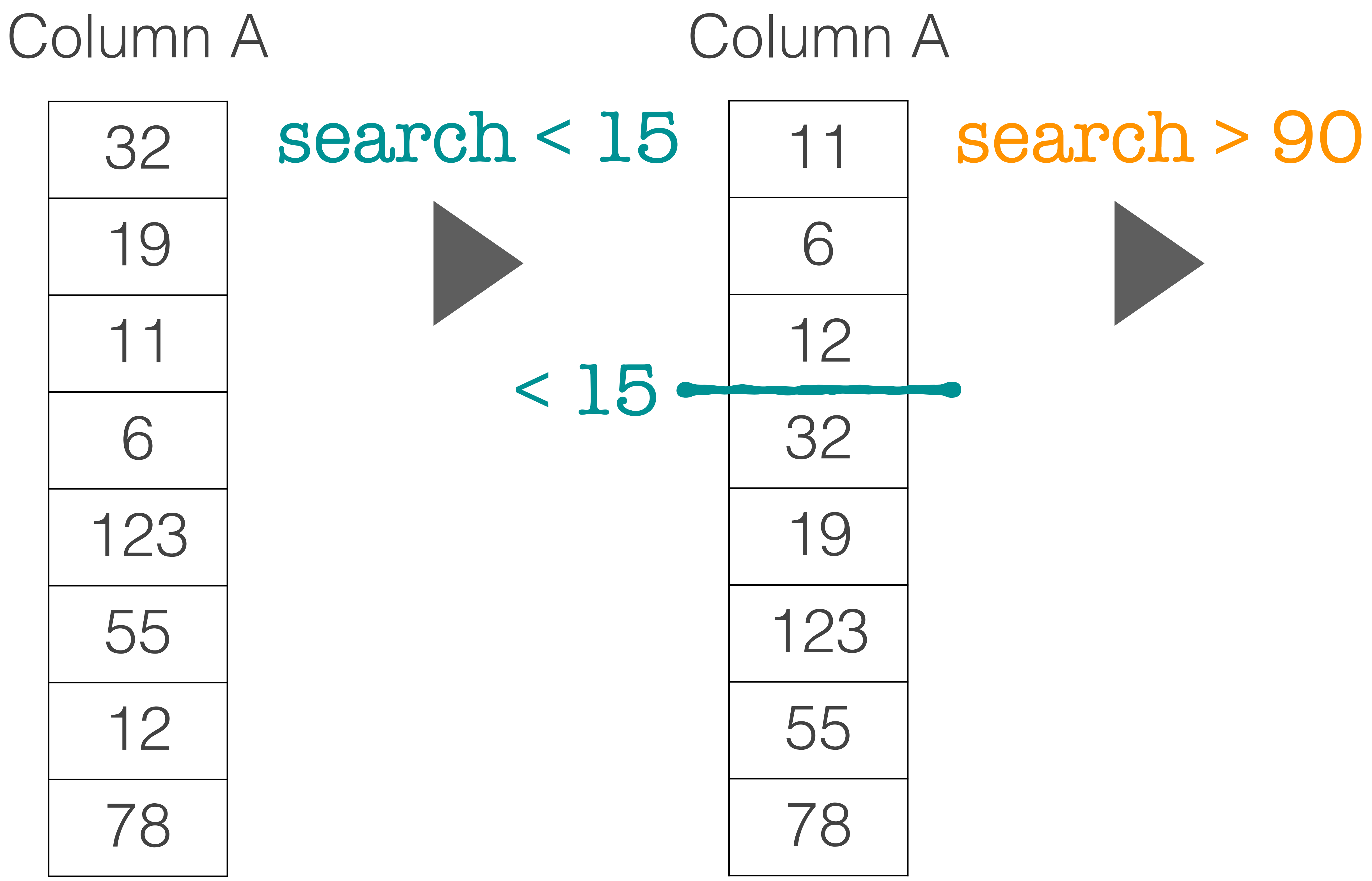


Cracking

Indexing on the fly

Idea: take **hints from queries** to **create partitions**

gradually moving toward a **sorted layout**

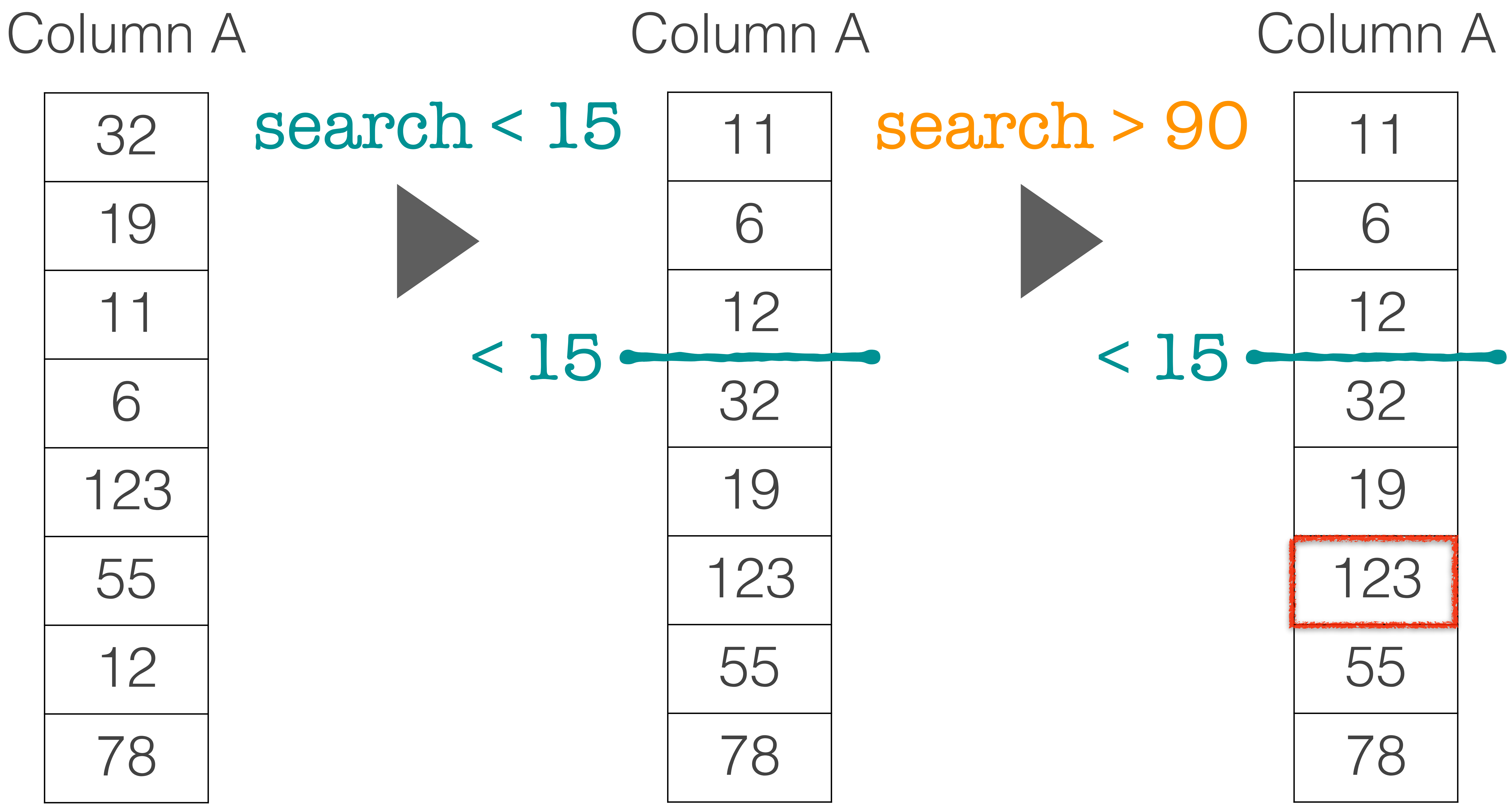


Cracking

Indexing on the fly

Idea: take **hints from queries** to **create partitions**

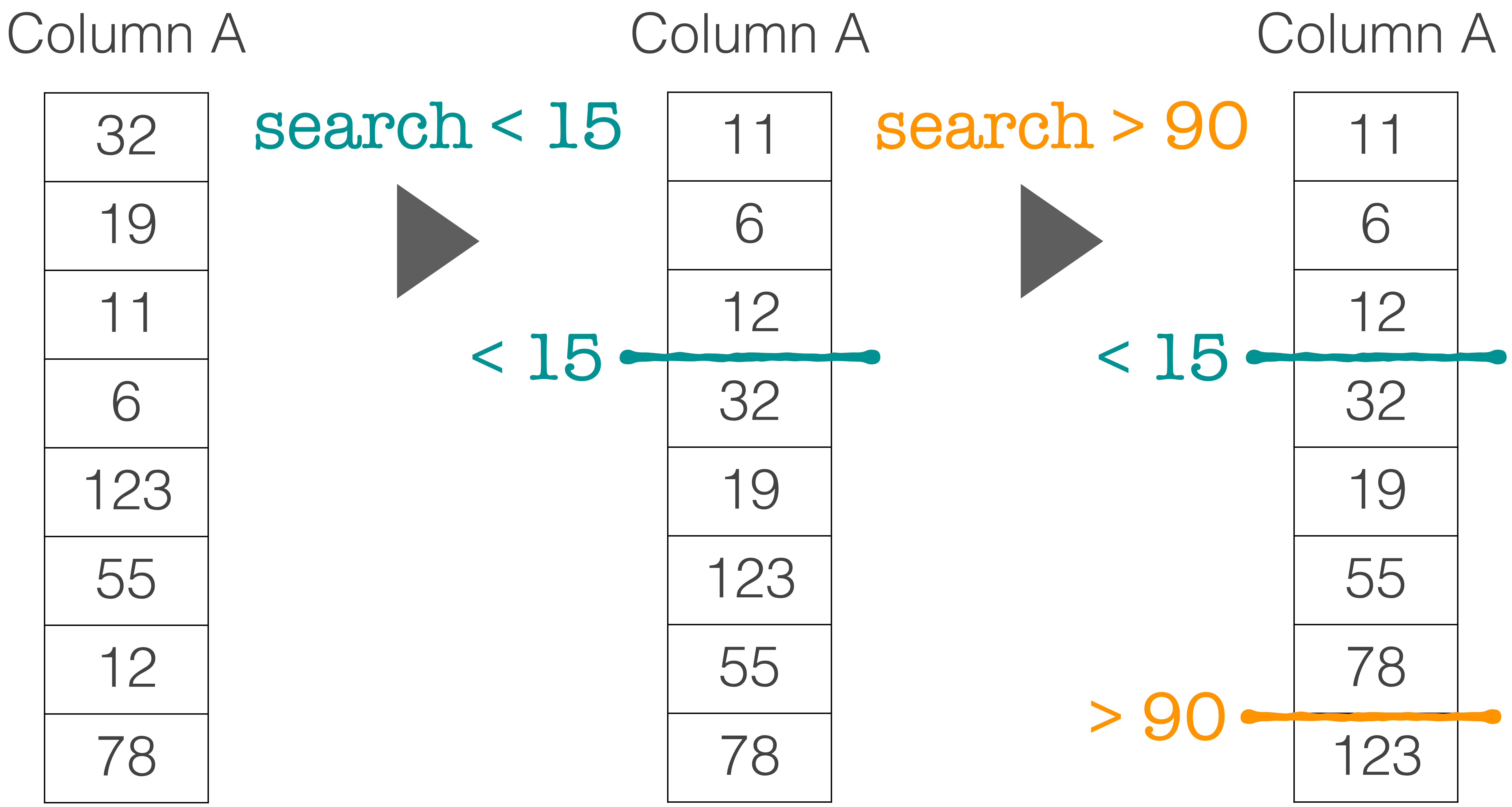
gradually moving toward a **sorted layout**



Cracking

Indexing on the fly

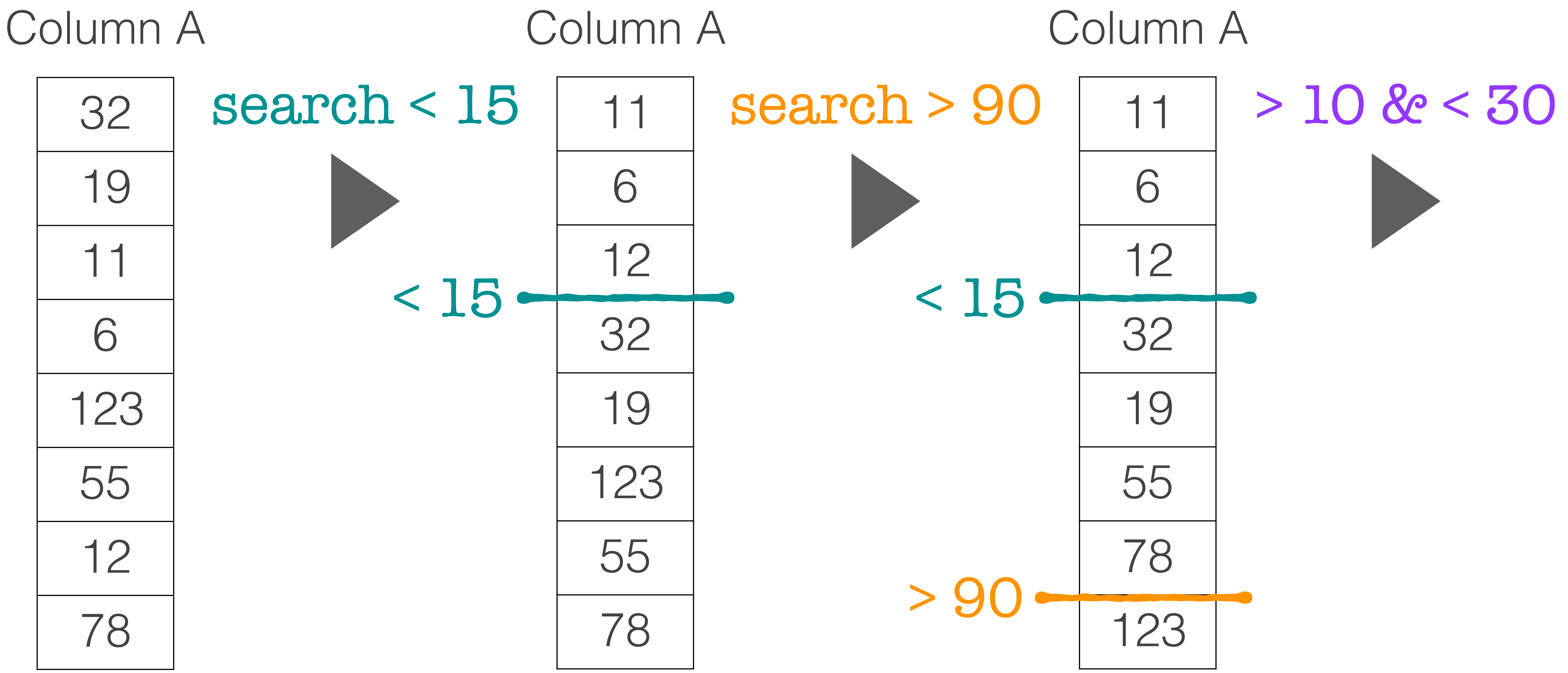
Idea: take **hints from queries** to **create partitions** gradually moving toward a **sorted layout**



Cracking

Indexing on the fly

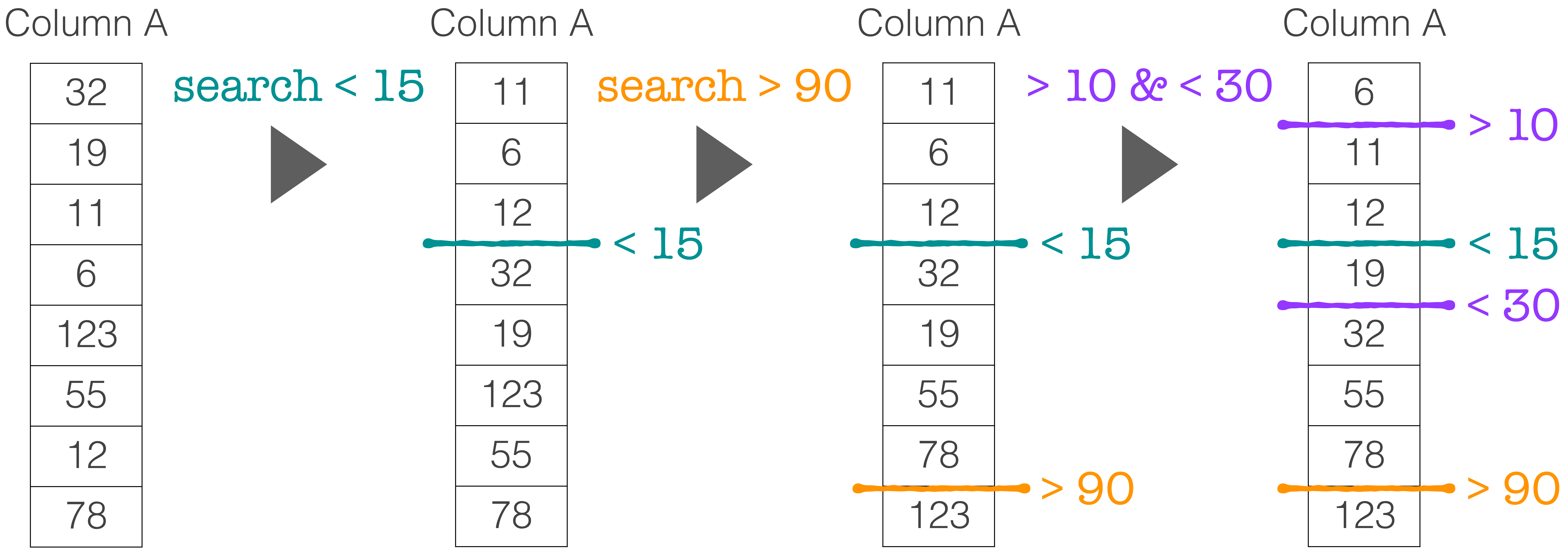
Idea: take **hints from queries** to **create partitions**
gradually moving toward a **sorted layout**



Cracking

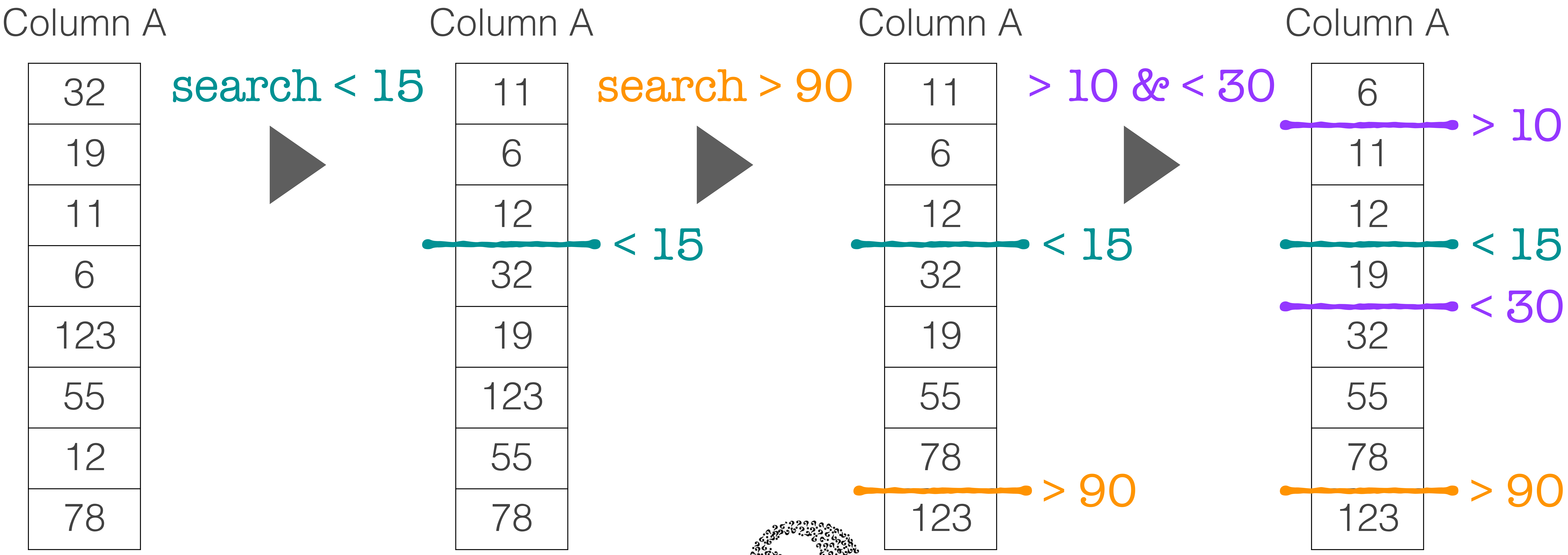
Indexing on the fly

Idea: take **hints from queries** to **create partitions**
gradually moving toward a **sorted layout**



Cracking

Indexing on the fly



Thought Experiment 5

What about **updates**?

Lazy merging

What are the possible **index designs**?

From B-trees to cracking

index	**point** queries	**short range** queries	**long range** queries	data **skew**	**updates**
B+-tree	✓	✓	✓	✓	–
LSM-tree	✓	✗	–	✓	✓
Radix tree	✓	✓	✓	✗	–
Hash index	✓	–	✗	✗	✓
Bitmap index	✓	–	–	–	✗
Zonemap	–	–	✓	✓	–

How to decide which index to use?

How to decide **which index to use**?

The million dollar question

Break it down to
design primitives!

Index design primitives

Asking the fundamental design questions

How to **physically organize** the data?

How to **search** through the data?

Can we **accelerate** search **using metadata**?

How to **update** or **add** new **data**?

Index design primitives

Asking the fundamental design questions

Global data organization
- sorted
- unsorted
- logging

How to **search** through the data?

Can we **accelerate** search **using metadata**?

How to **update** or **add** new **data**?

Index design primitives

Asking the fundamental design questions

Global data organization { sorted / unsorted / logging

Global search algorithm { scan / tight-loop search / direct addressing

Can we **accelerate** search **using metadata**?

How to **update** or **add** new **data**?

Index design primitives

Asking the fundamental design questions

Global data organization
- sorted
- unsorted
- logging

Global search algorithm
- scan
- tight-loop search
- direct addressing

Indexing technique
- zonemaps/imprints
- trees (radix/B+)
- Hash-based

How to **update** or **add** new **data**?

Index design primitives

Asking the fundamental design questions

Global data organization
- sorted
- unsorted
- logging

Global search algorithm
- scan
- tight-loop search
- direct addressing

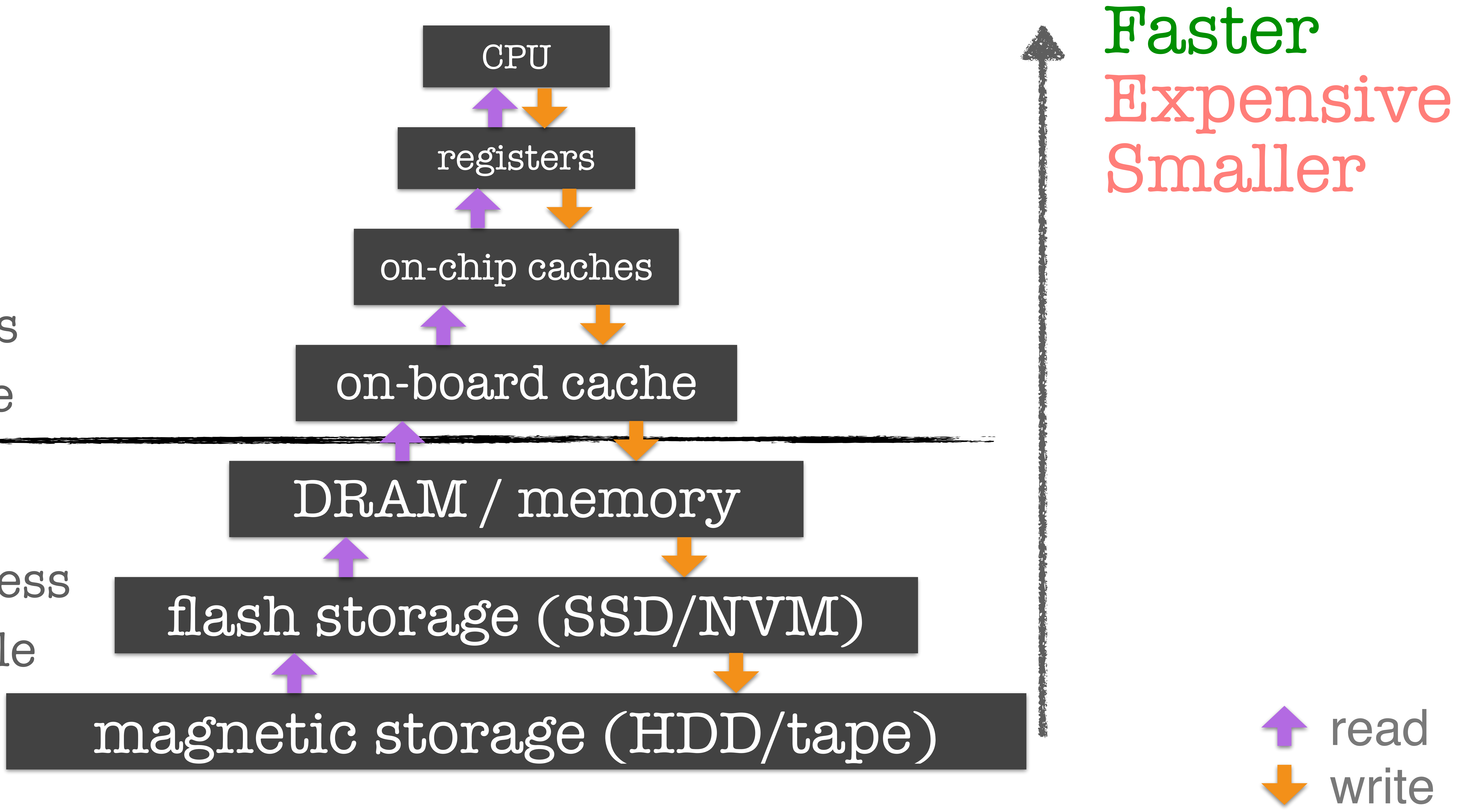
Indexing technique
- zonemaps/imprints
- trees (radix/B+)
- Hash-based

Data modification policy
- in-place
- out-of-place
- deffered in-place

Modern Hardware

Recap: **Storage hierarchy**

How data moves!



Volatile

Random access

Byte accessible

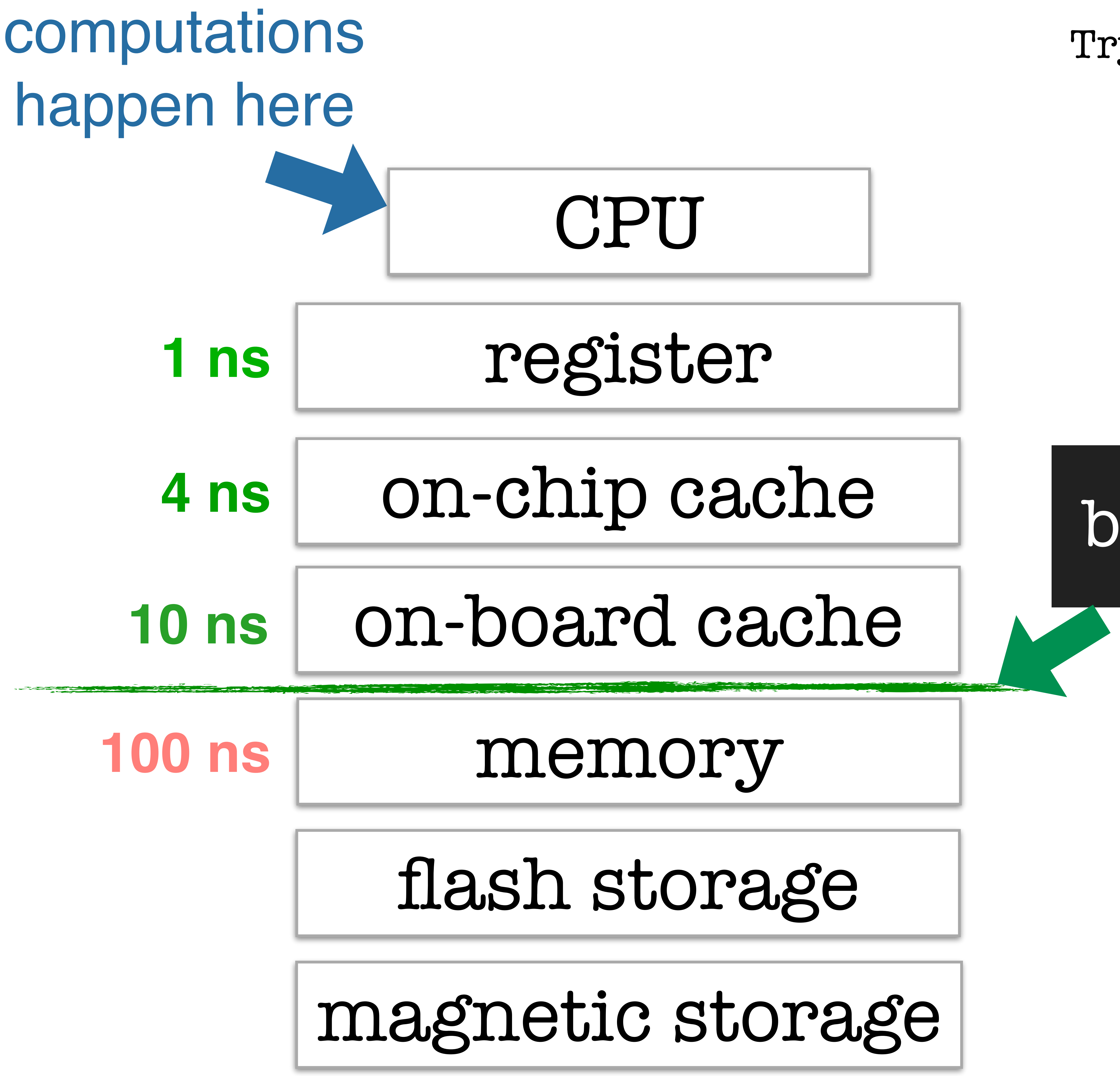
Non-volatile

Sequential access

Block accessible

Memory wall

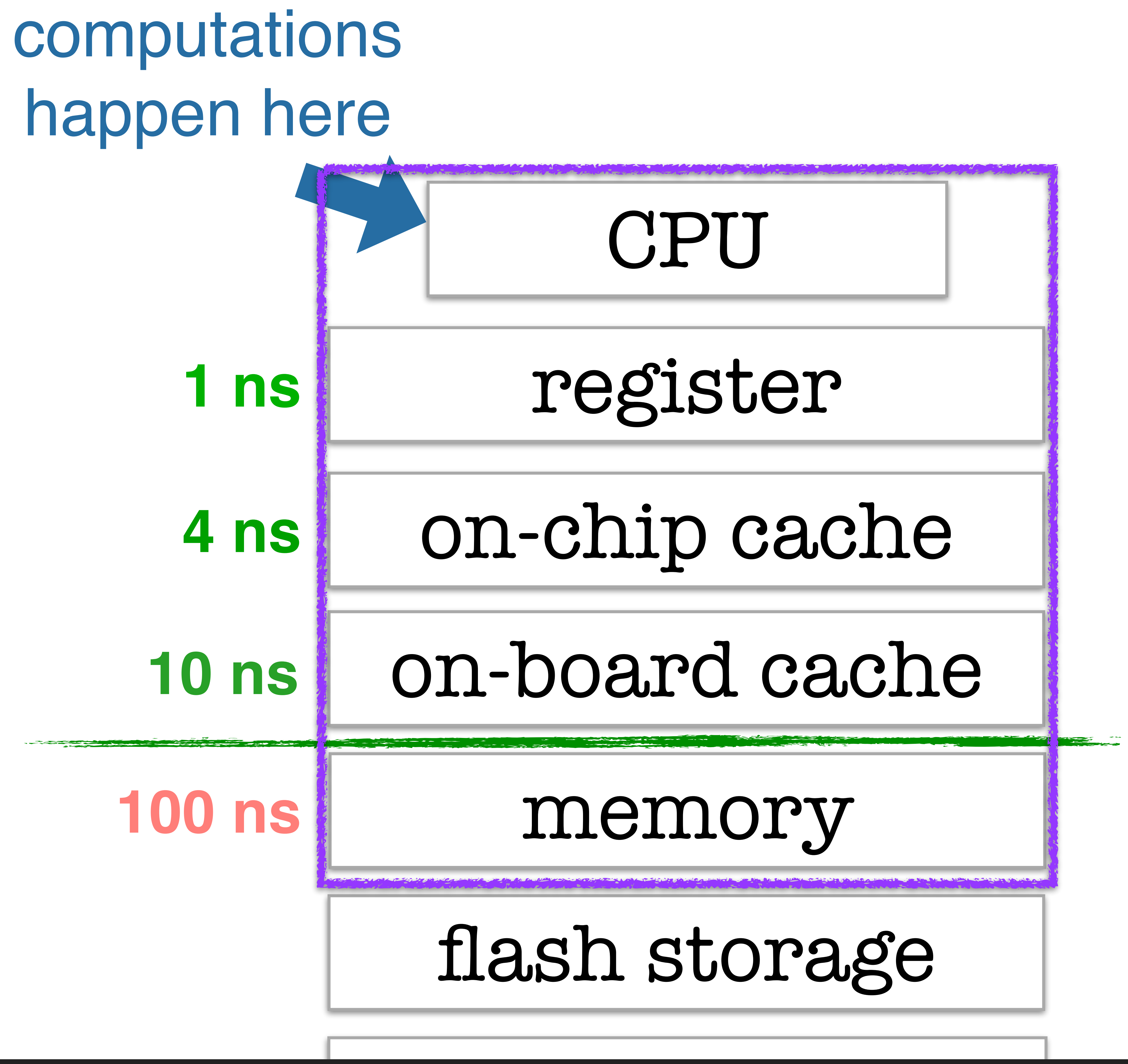
Try not to jump the wall



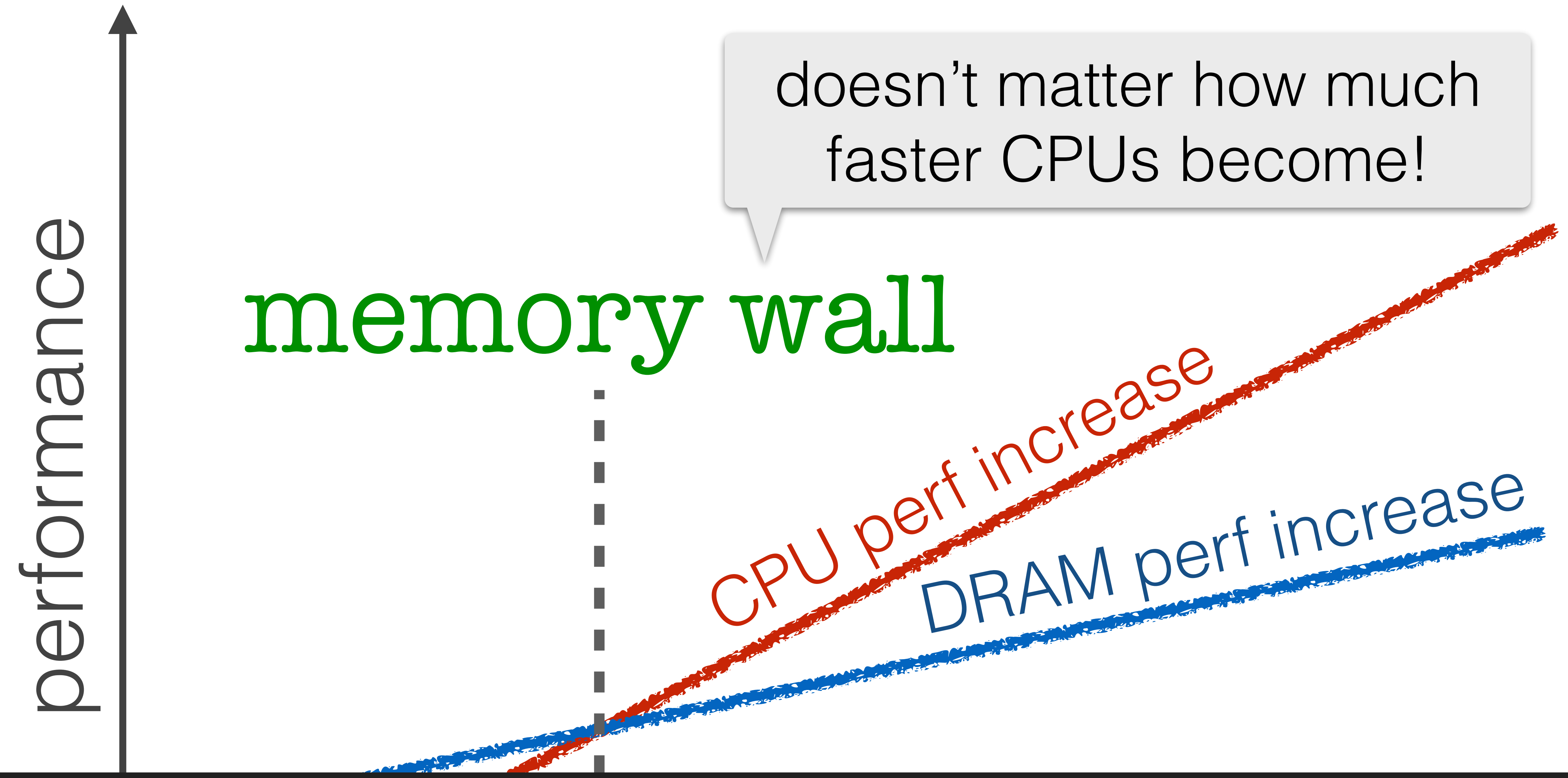
be careful when you go below the green line

Memory wall

Try not to jump the wall



be careful when you go below the green line



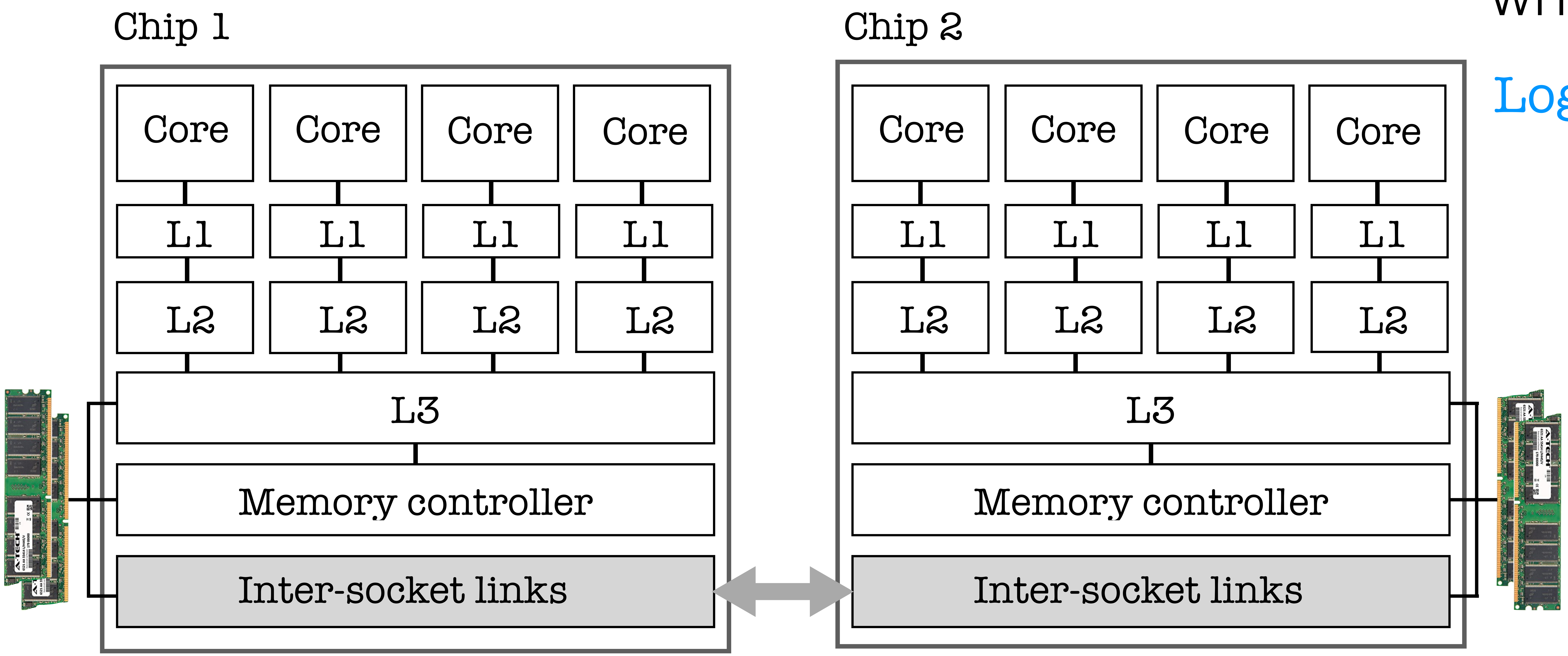
Can we optimize futher if **data fit in memory**?

Cache hierarchy

Optimizing data access

what is a **chip**?
what is a **socket**?
what is a **core**?

Logical vs. Physical core

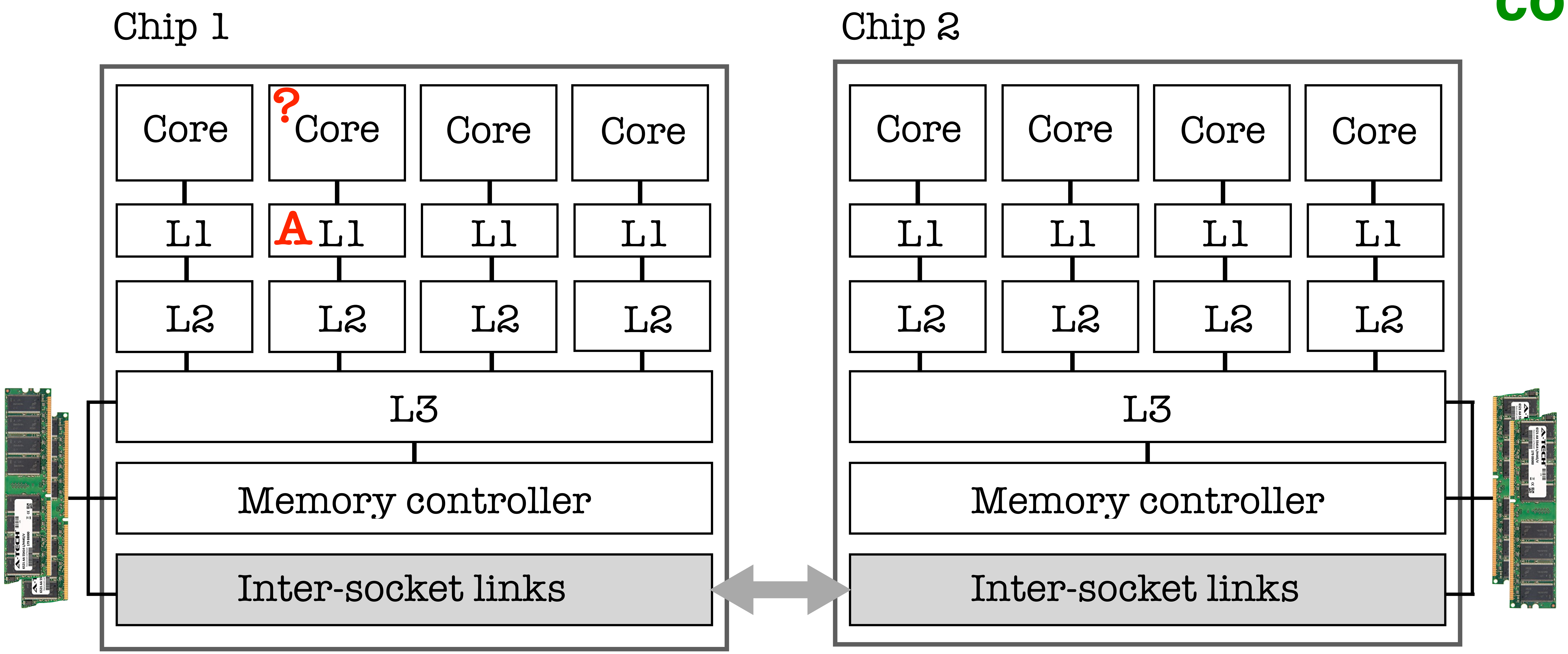


internals of a **multisocket multicore server**

Cache hierarchy

Optimizing data access

what if the target data is in the **core's private L1**?

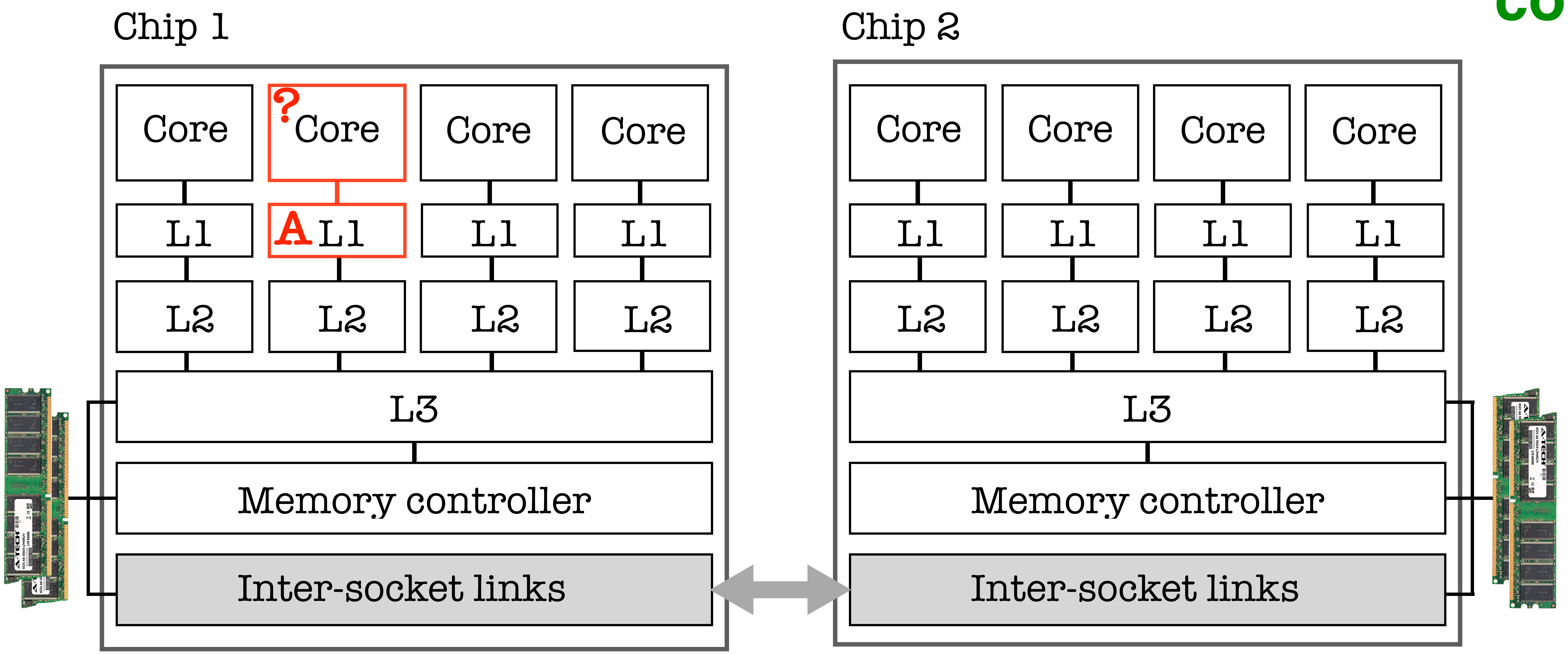


internals of a **multisocket multicore server**

Cache hierarchy

Optimizing data access

what if the target data is in the **core's private L1**?

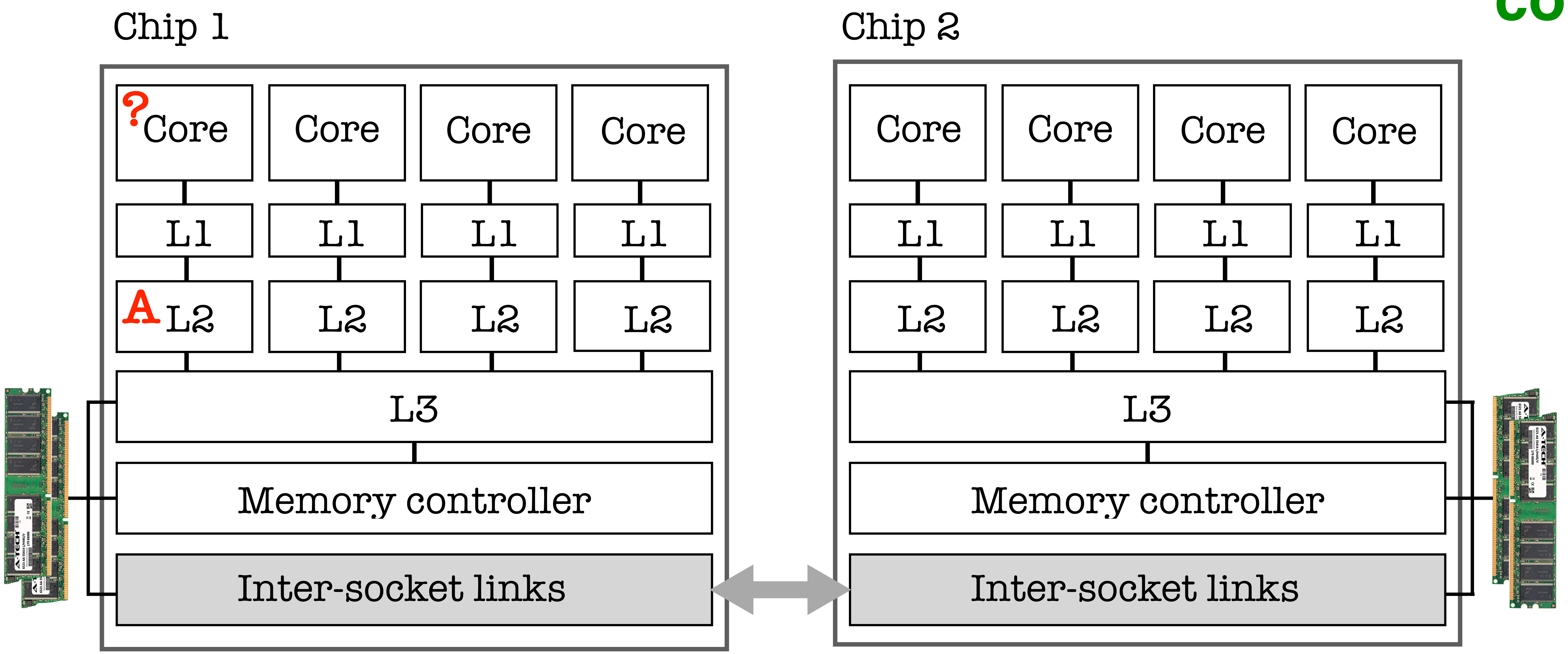


internals of a **multisocket multicore server**

Cache hierarchy

Optimizing data access

what if the target data is in the **core's private L2**?

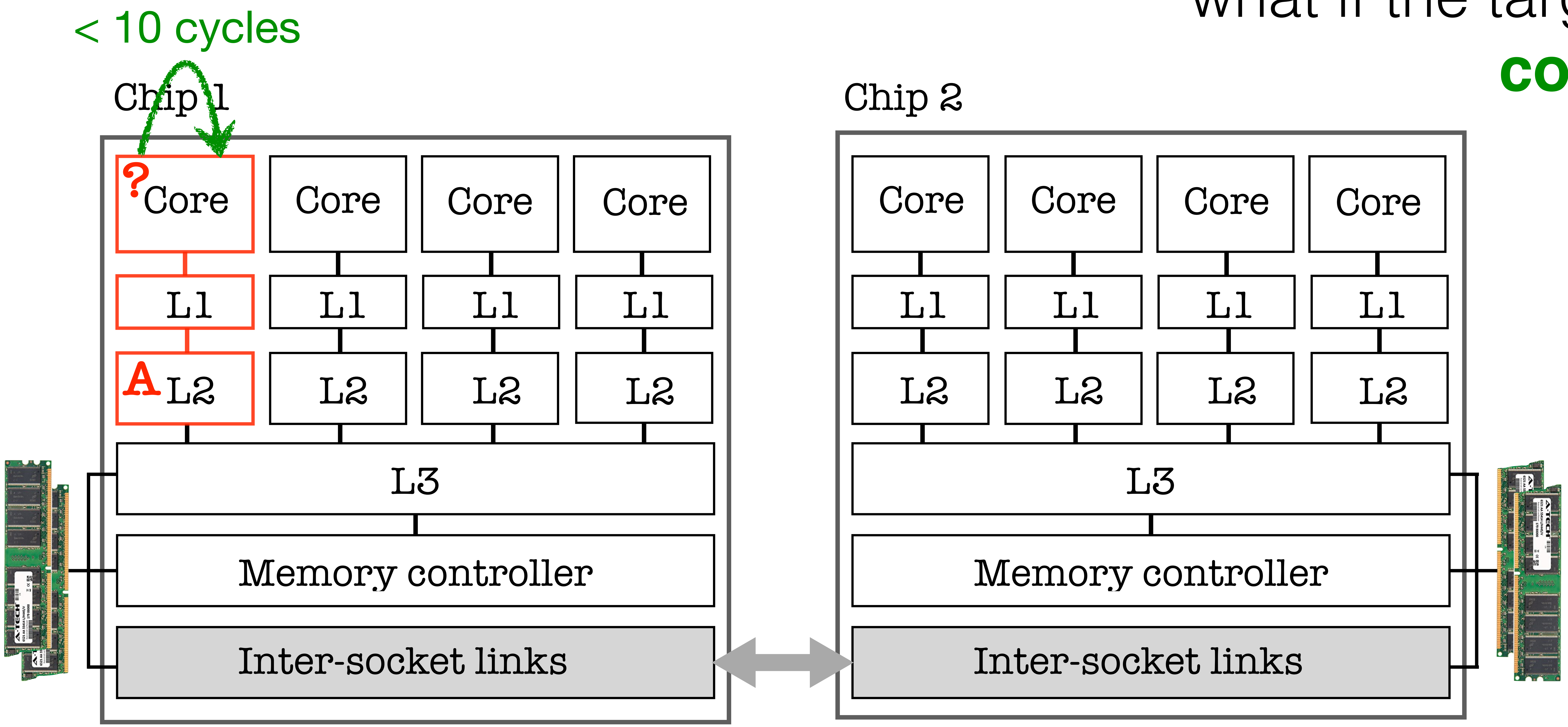


internals of a **multisocket multicore server**

Cache hierarchy

Optimizing data access

what if the target data is in the **core's private L2**?

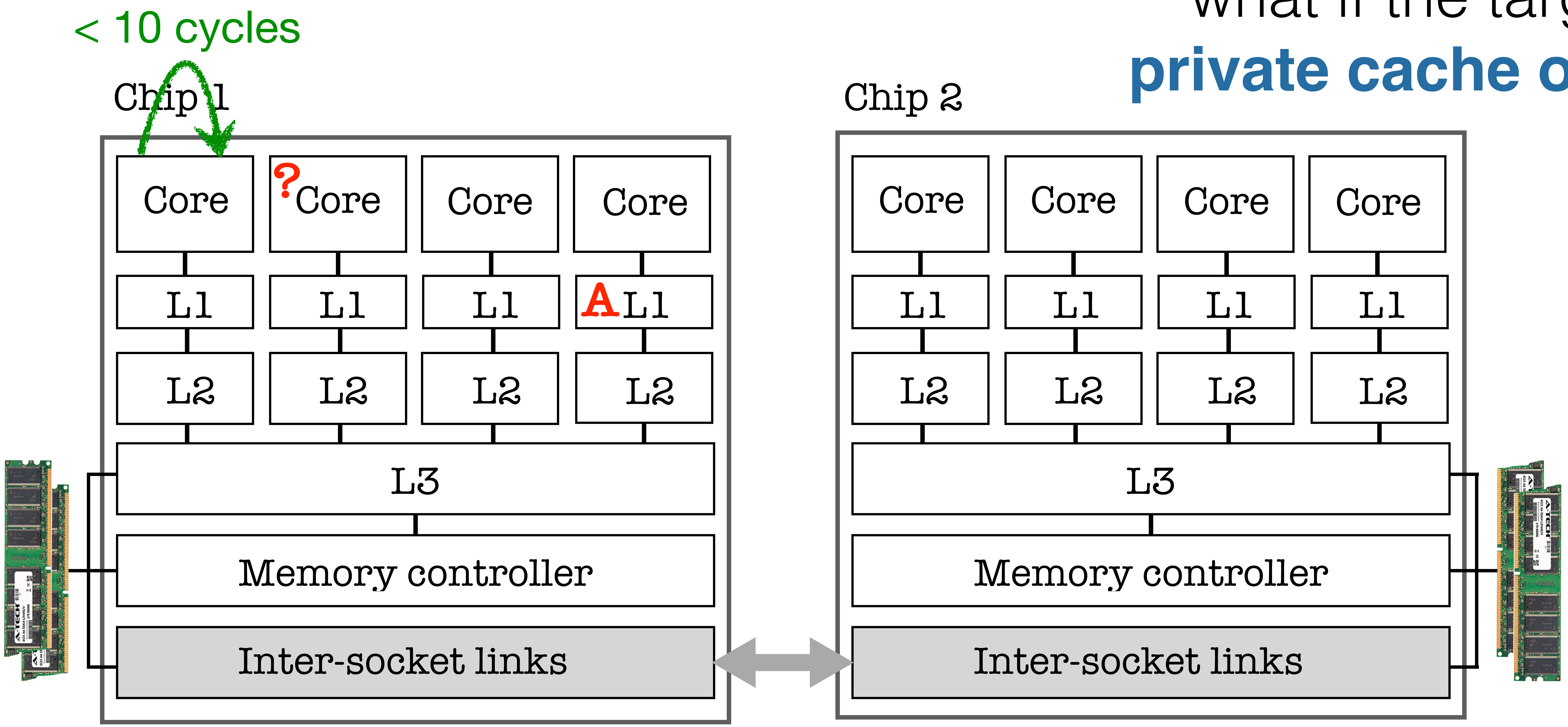


internals of a **multisocket multicore server**

Cache hierarchy

Optimizing data access

what if the target data is in the **private cache of another core**?

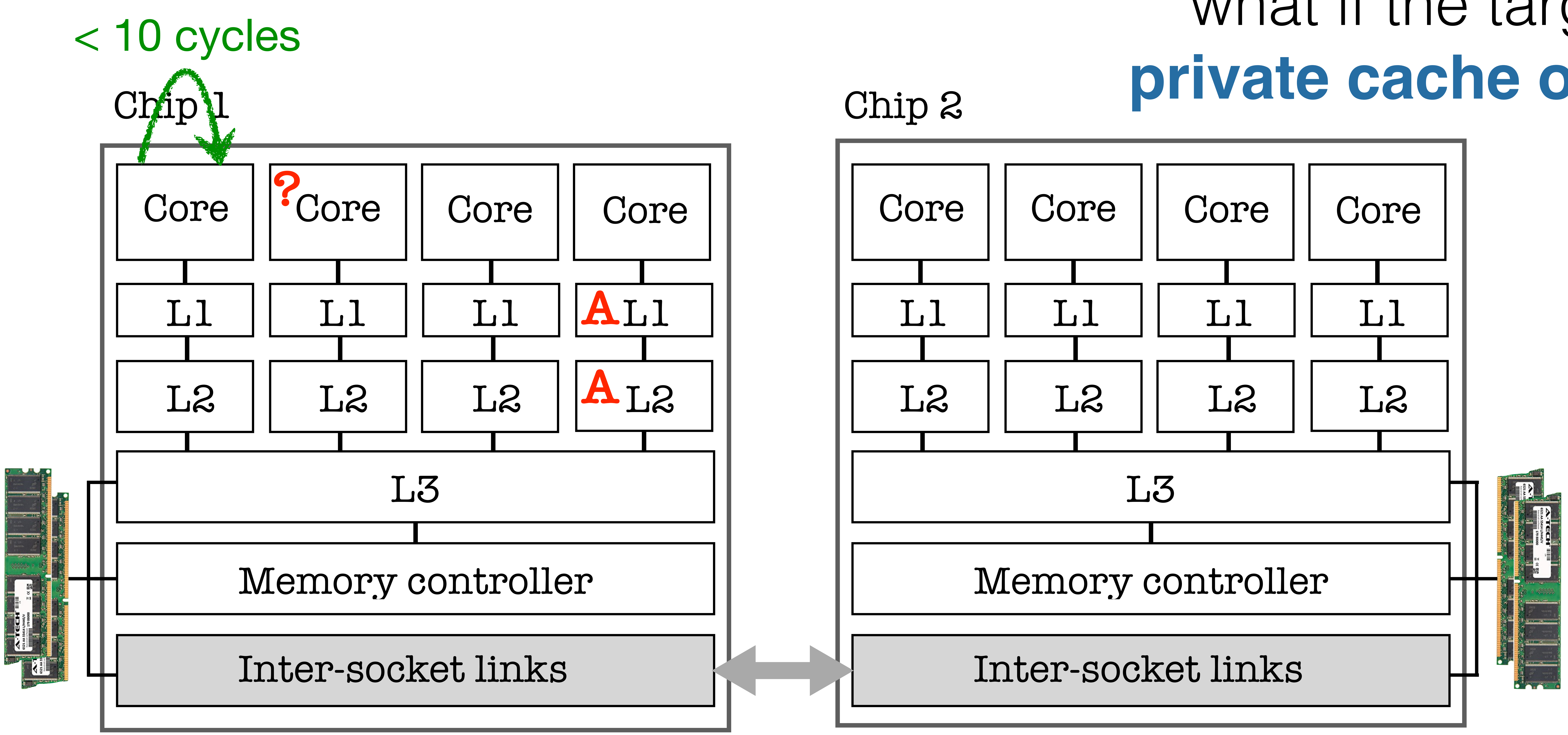


internals of a **multisocket multicore server**

Cache hierarchy

Optimizing data access

what if the target data is in the **private cache of another core**?

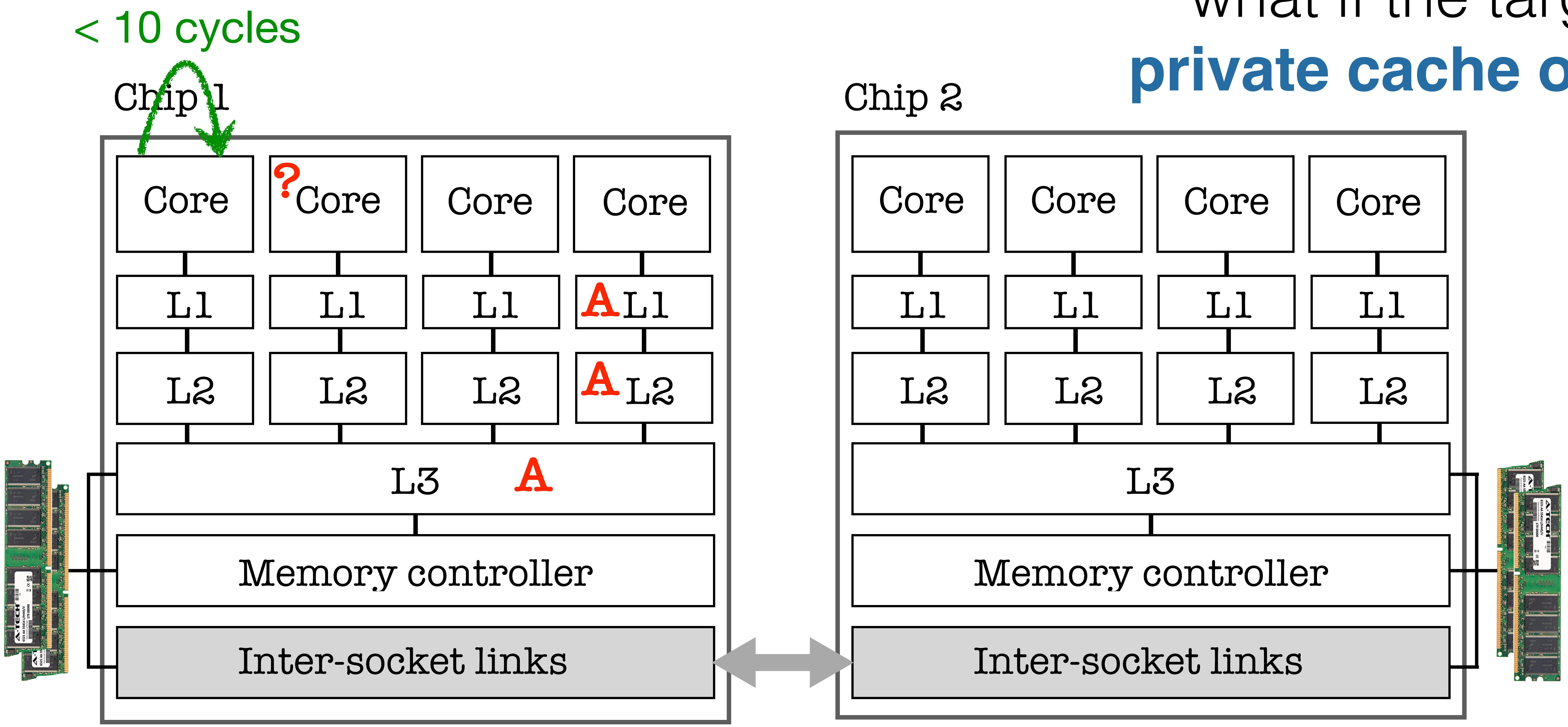


internals of a **multisocket multicore server**

Cache hierarchy

Optimizing data access

what if the target data is in the **private cache of another core**?

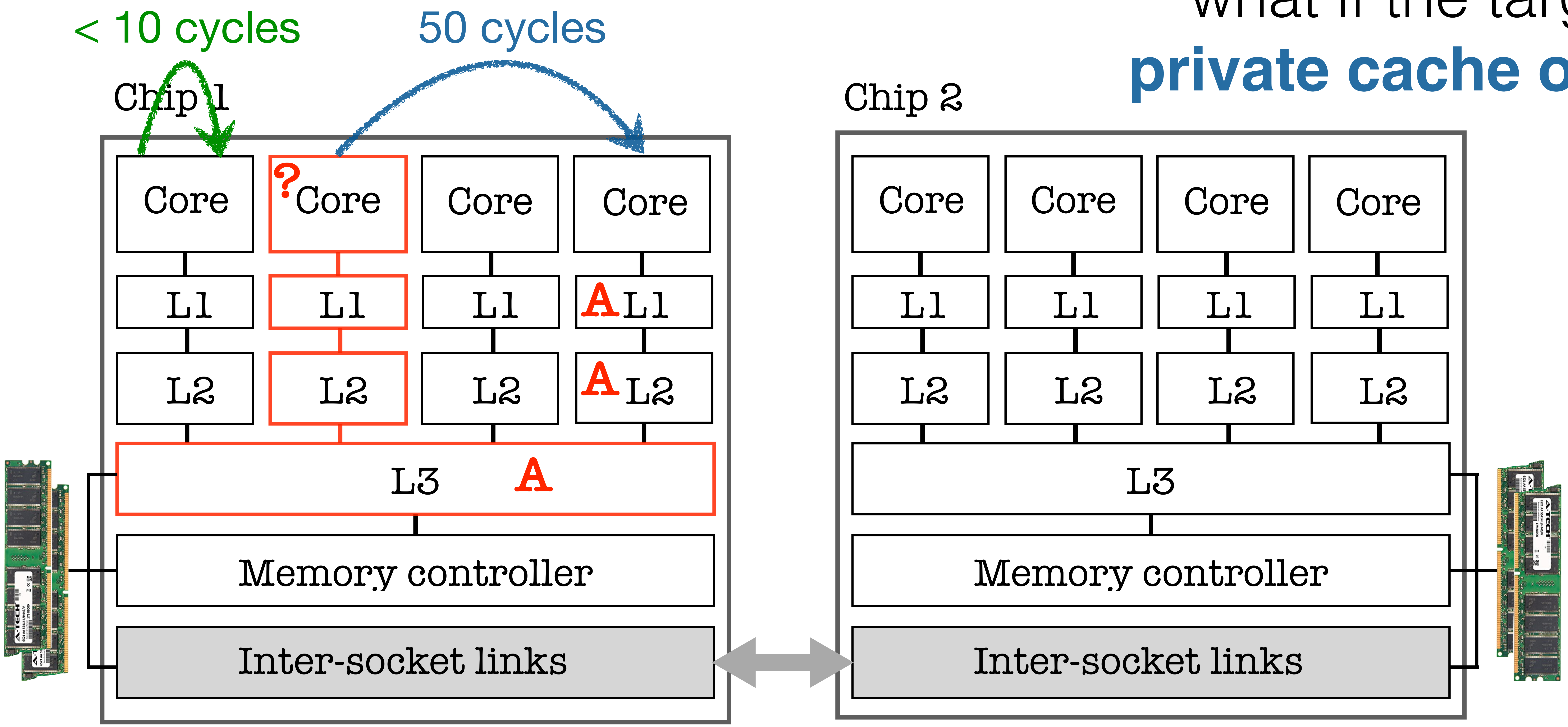


internals of a **multisocket multicore server**

Cache hierarchy

Optimizing data access

what if the target data is in the **private cache of another core**?

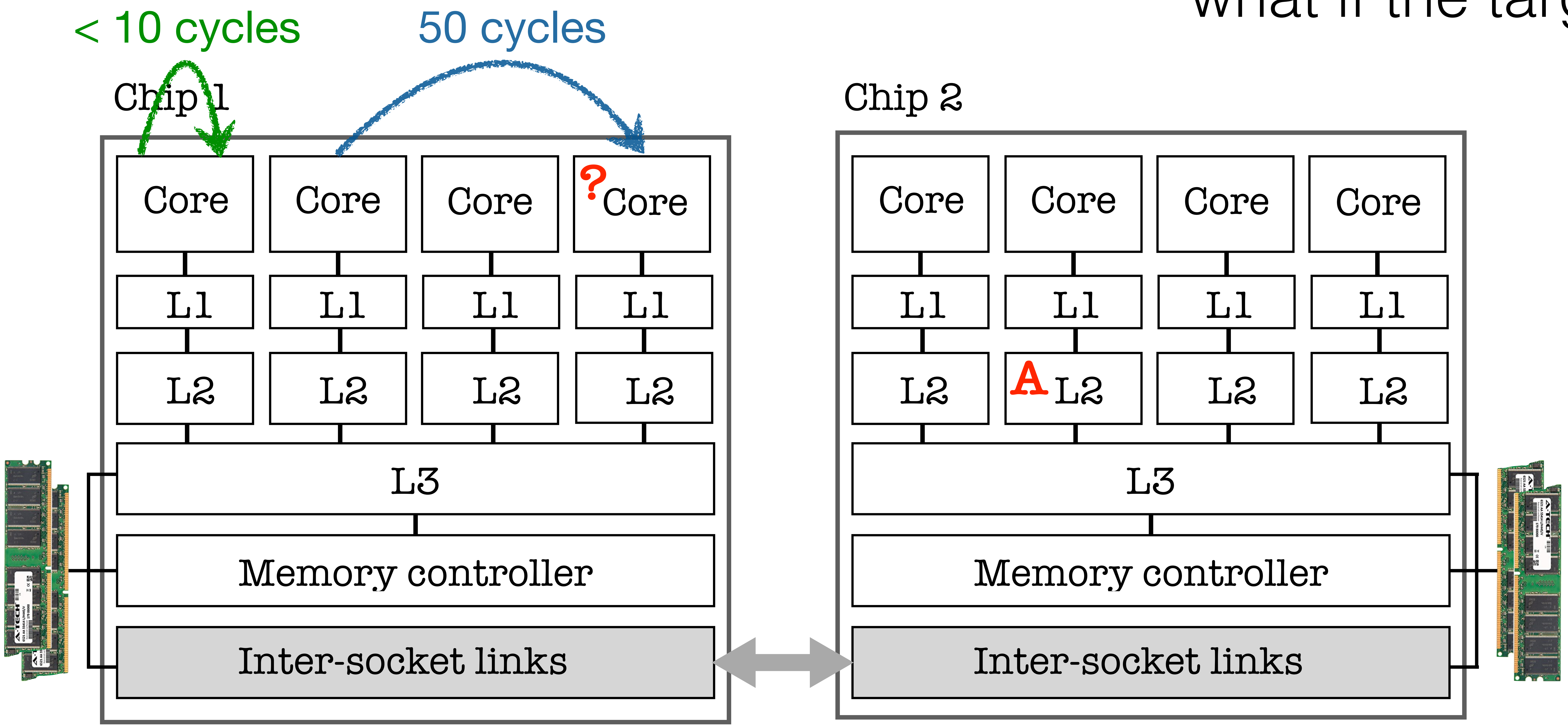


internals of a **multisocket multicore server**

Cache hierarchy

Optimizing data access

what if the target data is in the **another chip**?

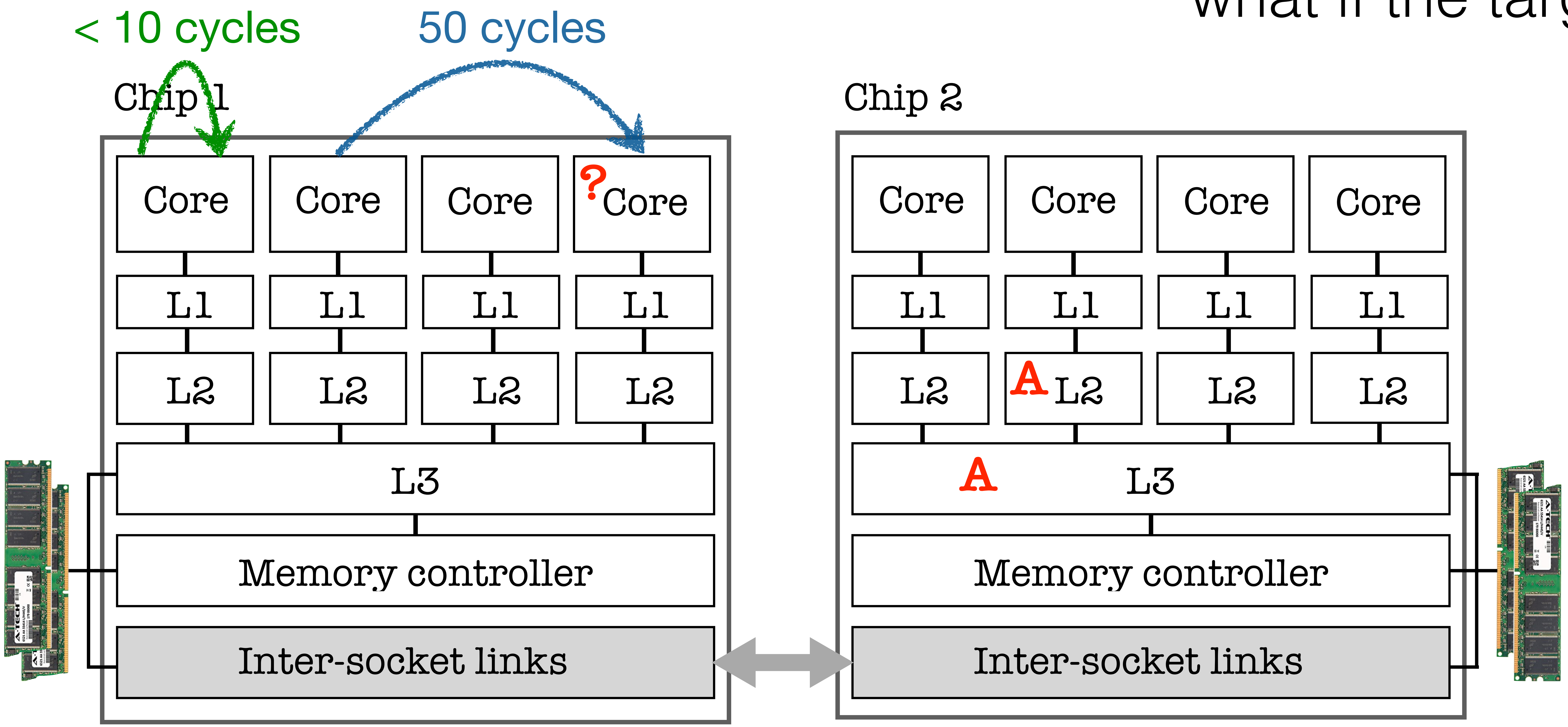


internals of a **multisocket multicore server**

Cache hierarchy

Optimizing data access

what if the target data is in the **another chip**?

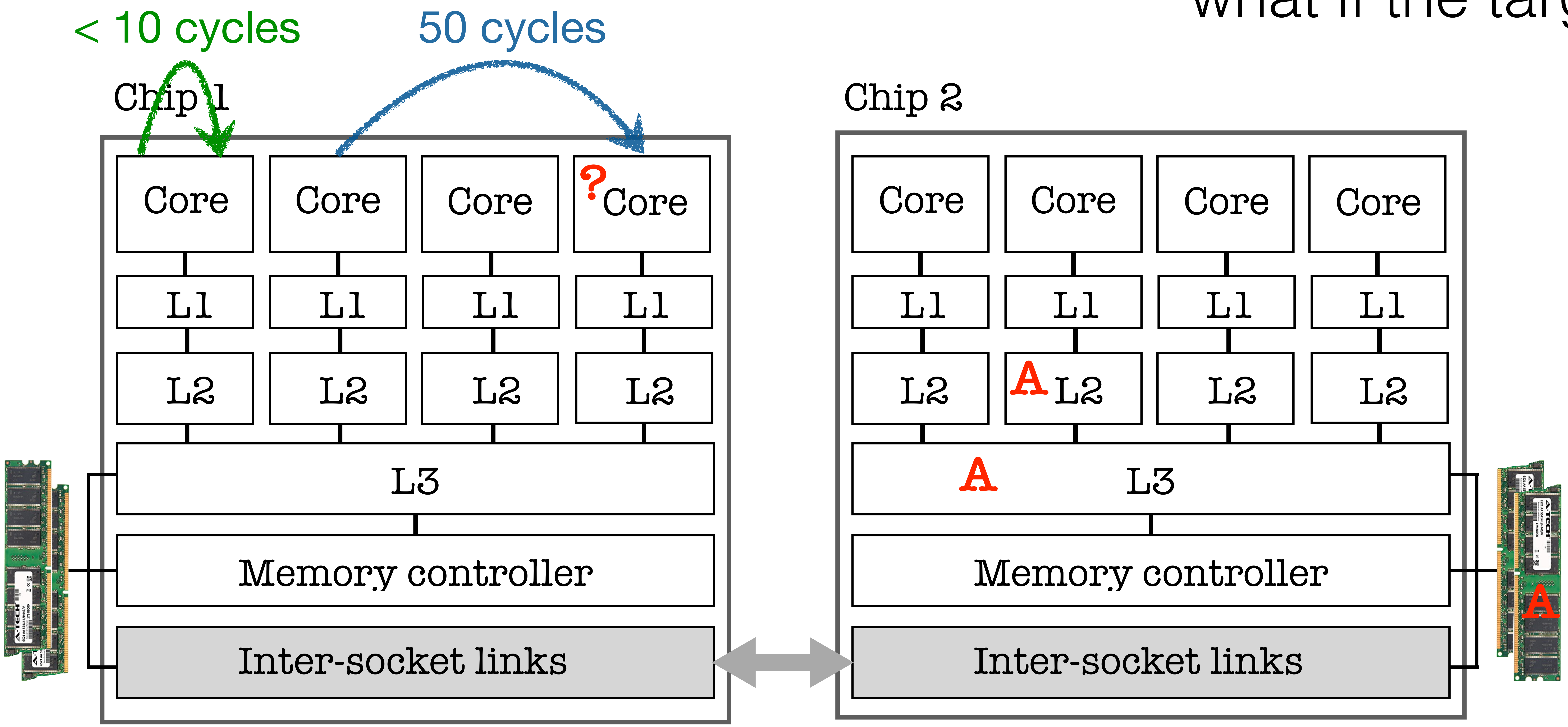


internals of a **multisocket multicore server**

Cache hierarchy

Optimizing data access

what if the target data is in the **another chip**?

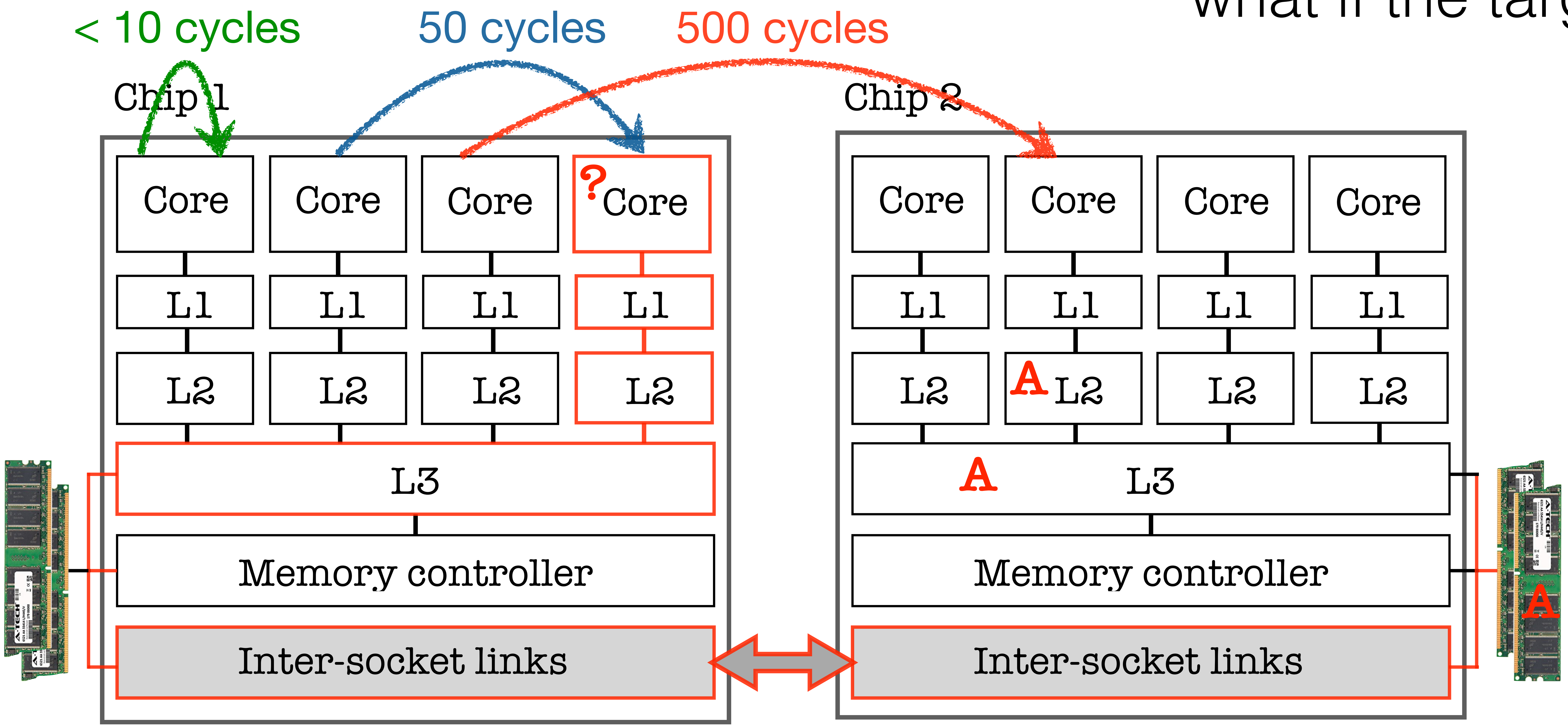


internals of a **multisocket multicore server**

Cache hierarchy

Optimizing data access

what if the target data is in the **another chip**?



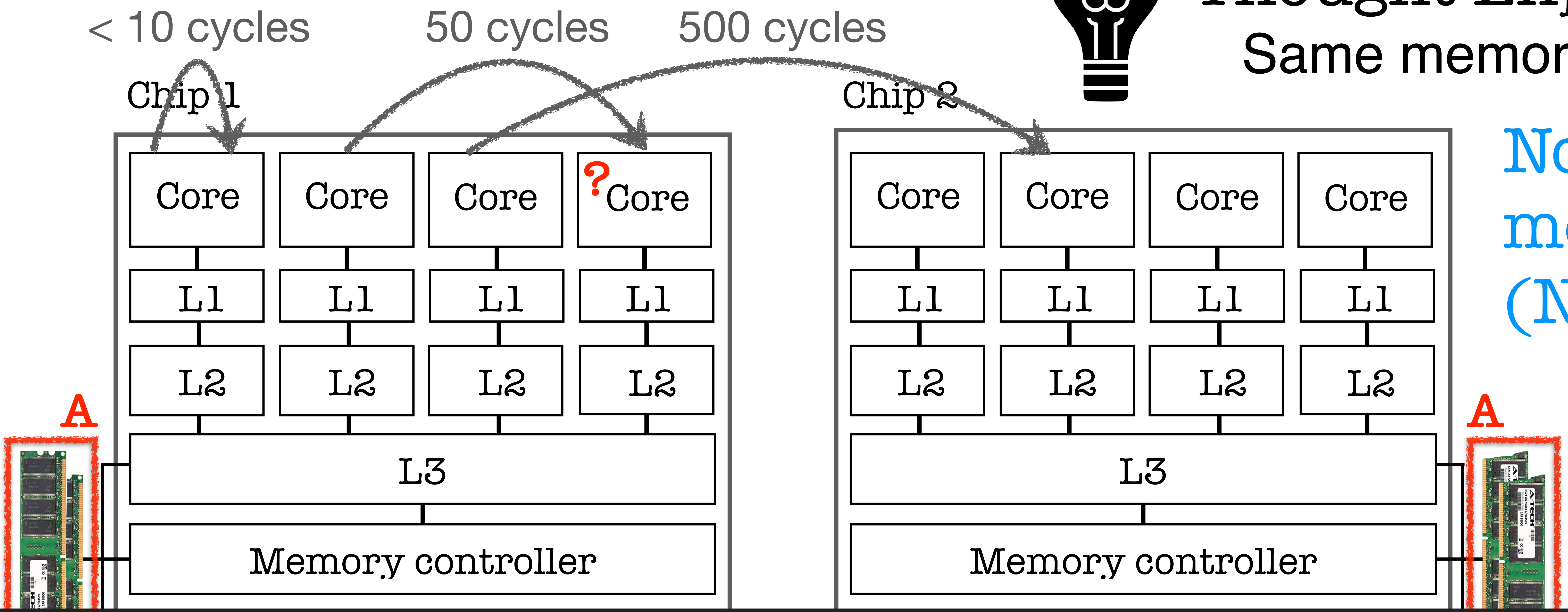
internals of a **multisocket multicore server**

Cache hierarchy

Optimizing data access

Thought Experiment 6

Same memory access time?



Non-uniform memory access (NUMA)

We data is placed in cache matters!

Disks

What are they really?

Brandeis UNIVERSITY

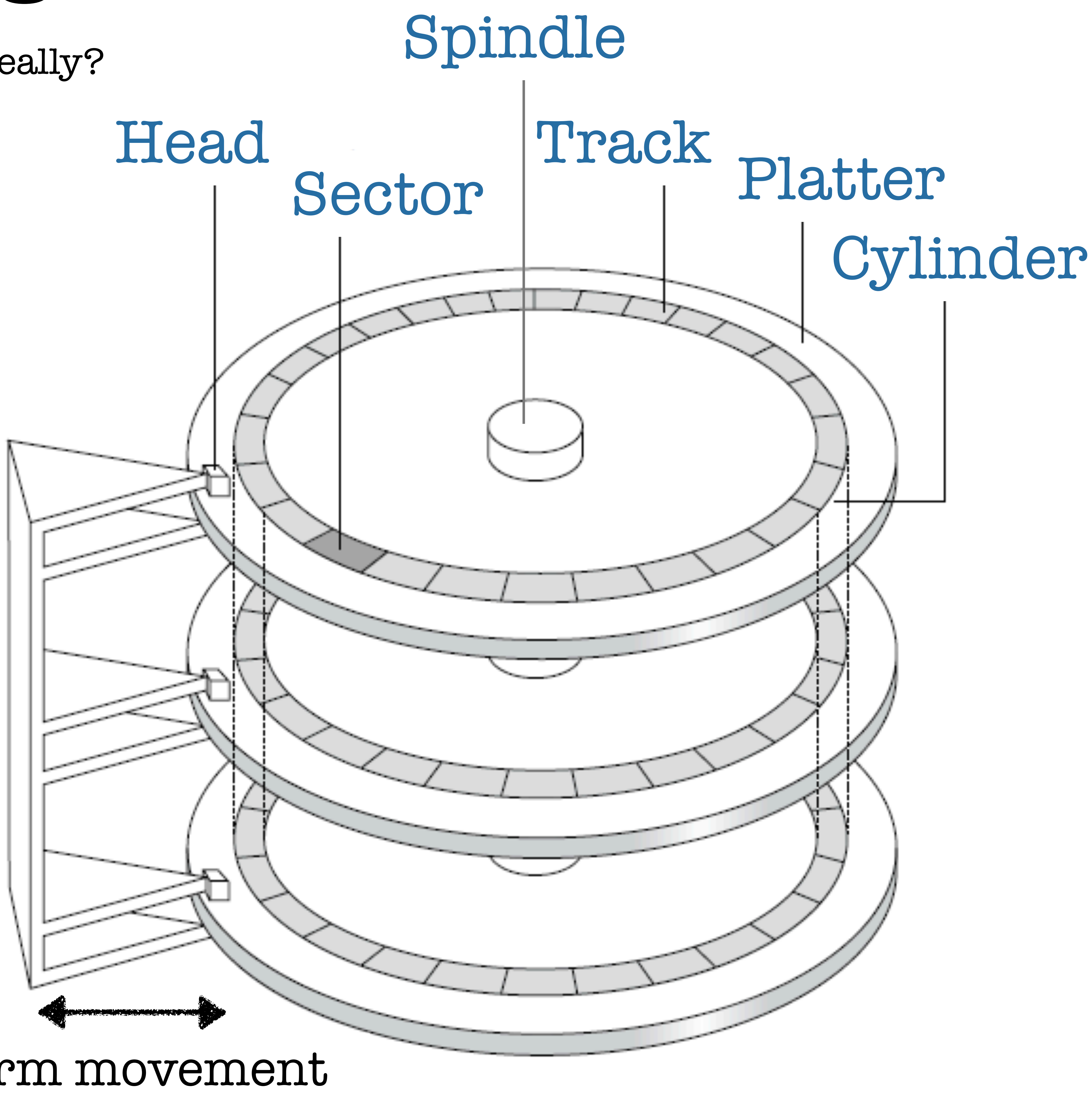
Disks

What are they really?

Arm assembly moves in and out to point to the correct **track**

Platters move around the spindle to get the desired **sector**

One head reads/writes at a time



Brandeis UNIVERSITY

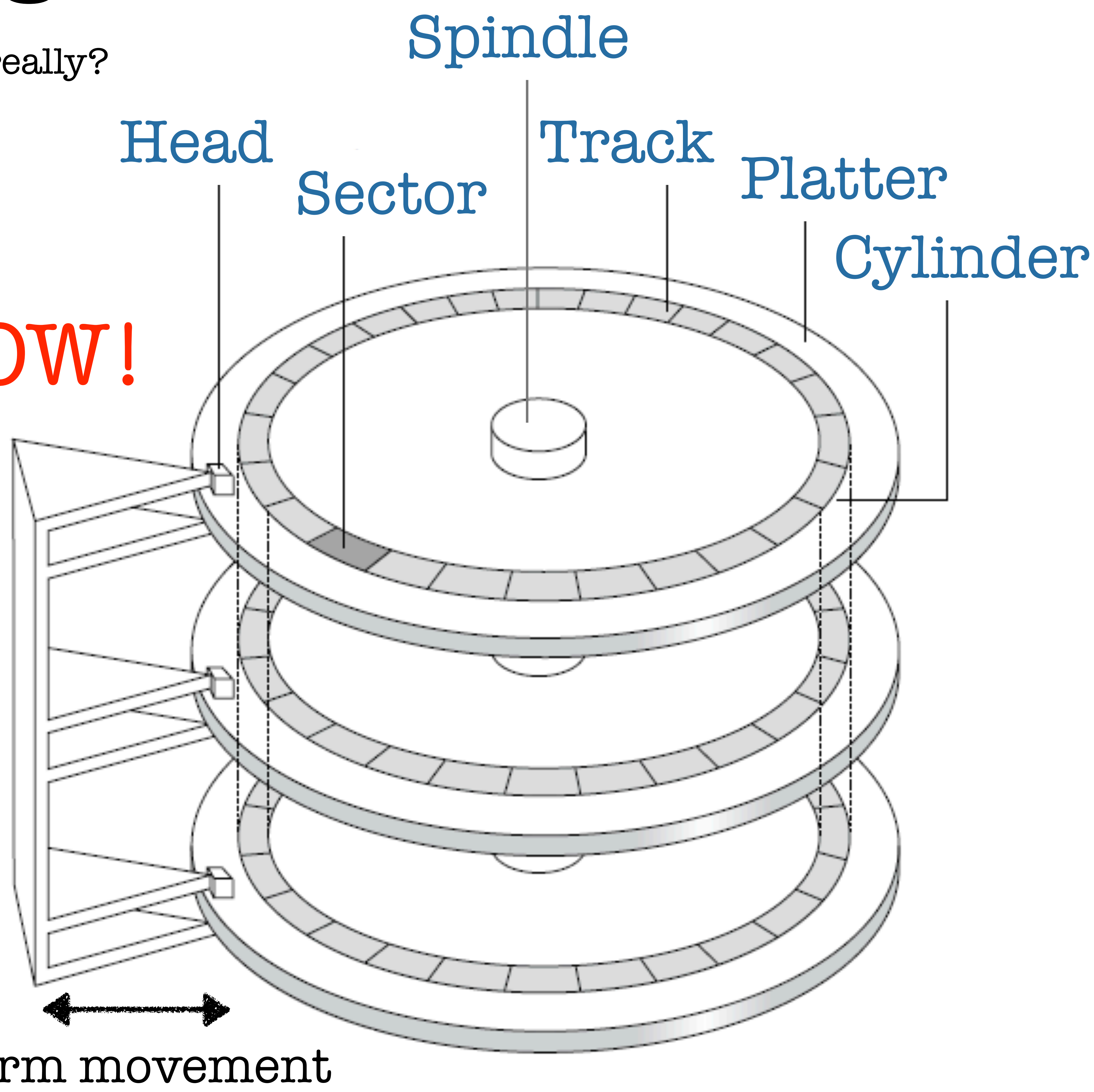
Disks

What are they really?

Time to access a **page of data**:

1. Find the **track** (move arm to track)
   seek latency
2. Find the **sector** (rotate the platters)
   rotational latency
3. Read/Write **page** (head does this)
   transfer latency

SLOW!



Flash disks

Around for >30 years, now!

Writes, **reads**, and **deletes** happen **electronically**!

no mechanical component

Data is still stored in **pages** (typically **4KB**)

Random reads are almost **as fast as sequential reads**

the 10%-20% difference in speed owes to **prefetching**

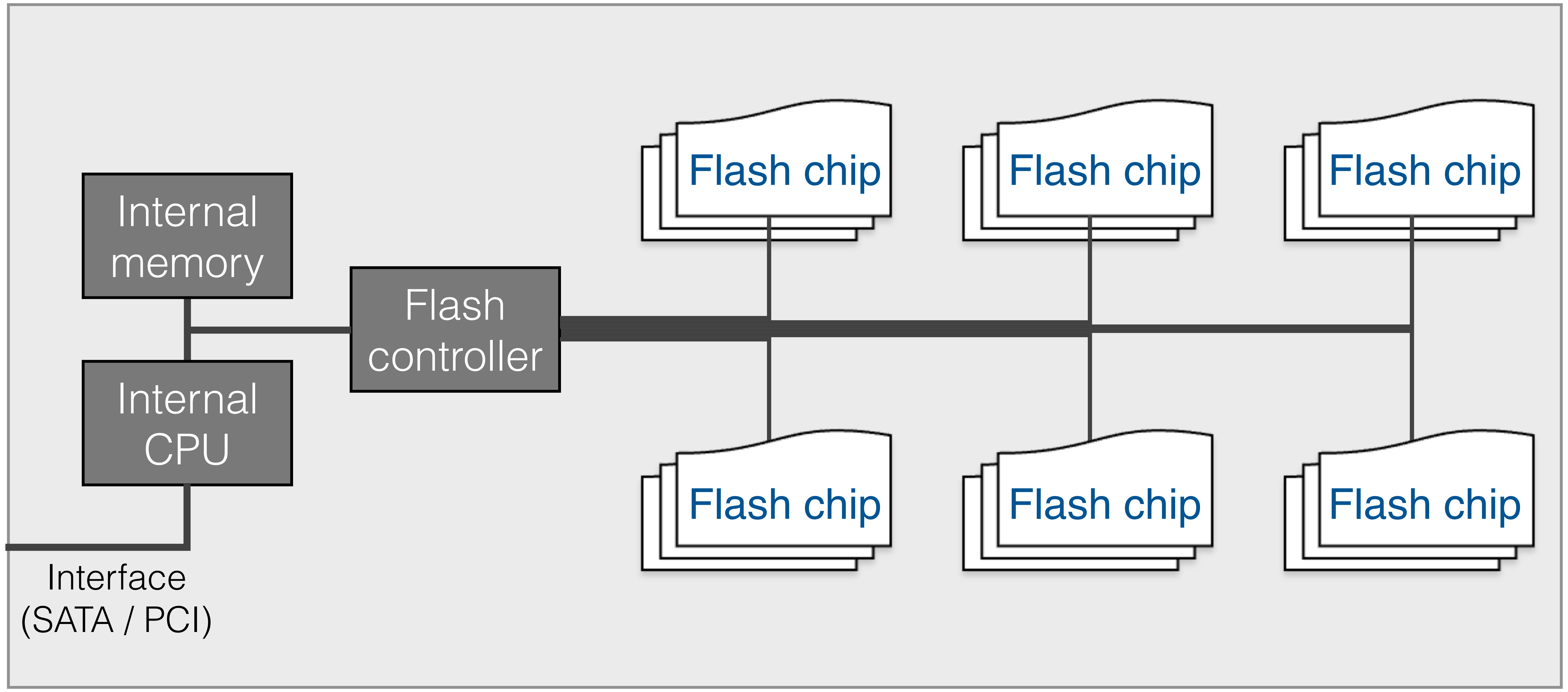
But, **random writes** are **slower than random reads**

an **asymmetry** exists **between reads and writes** on SSDs

Internals of **flash** disks

Let's have a sneak peak

SSD

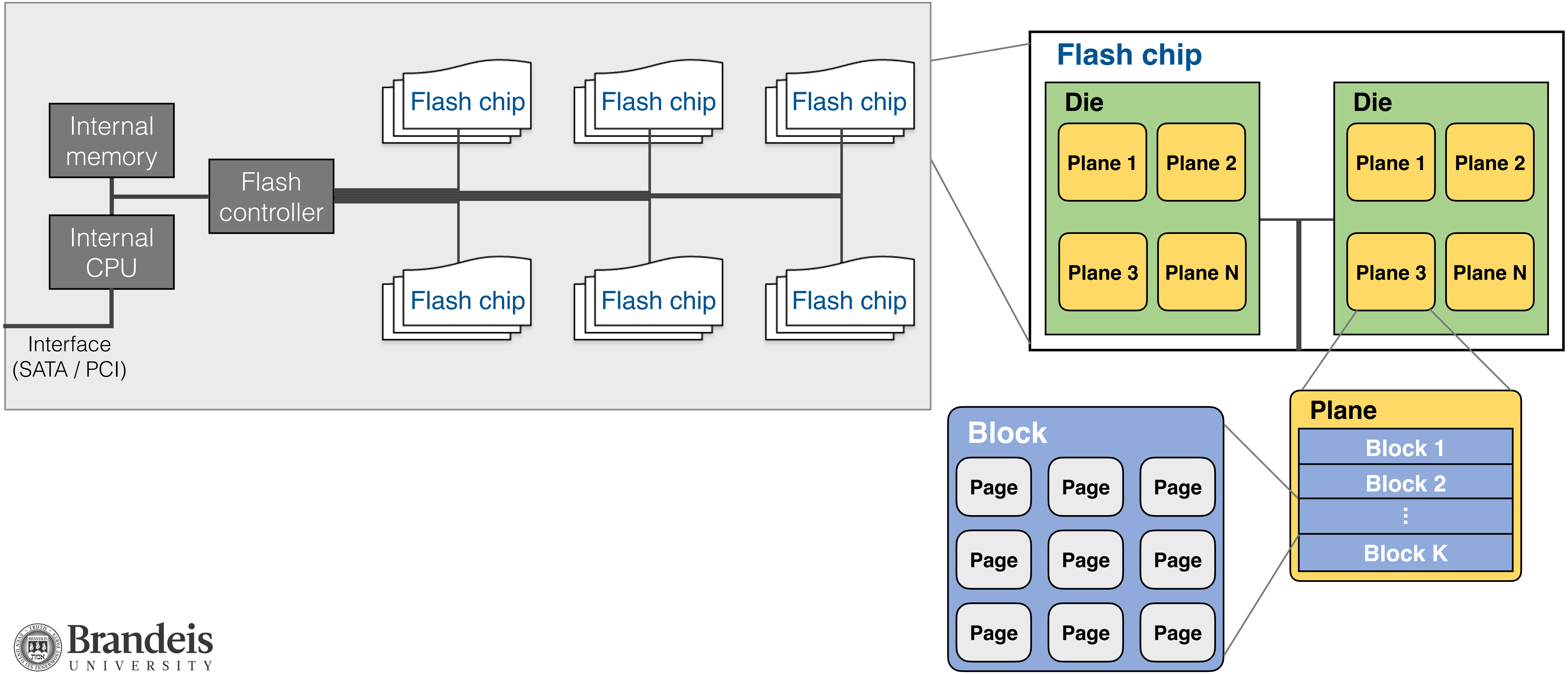


Reads/Writes can happen parallely in multiple flash chips

Internals of **flash** disks

SSD

Let's have a sneak peak

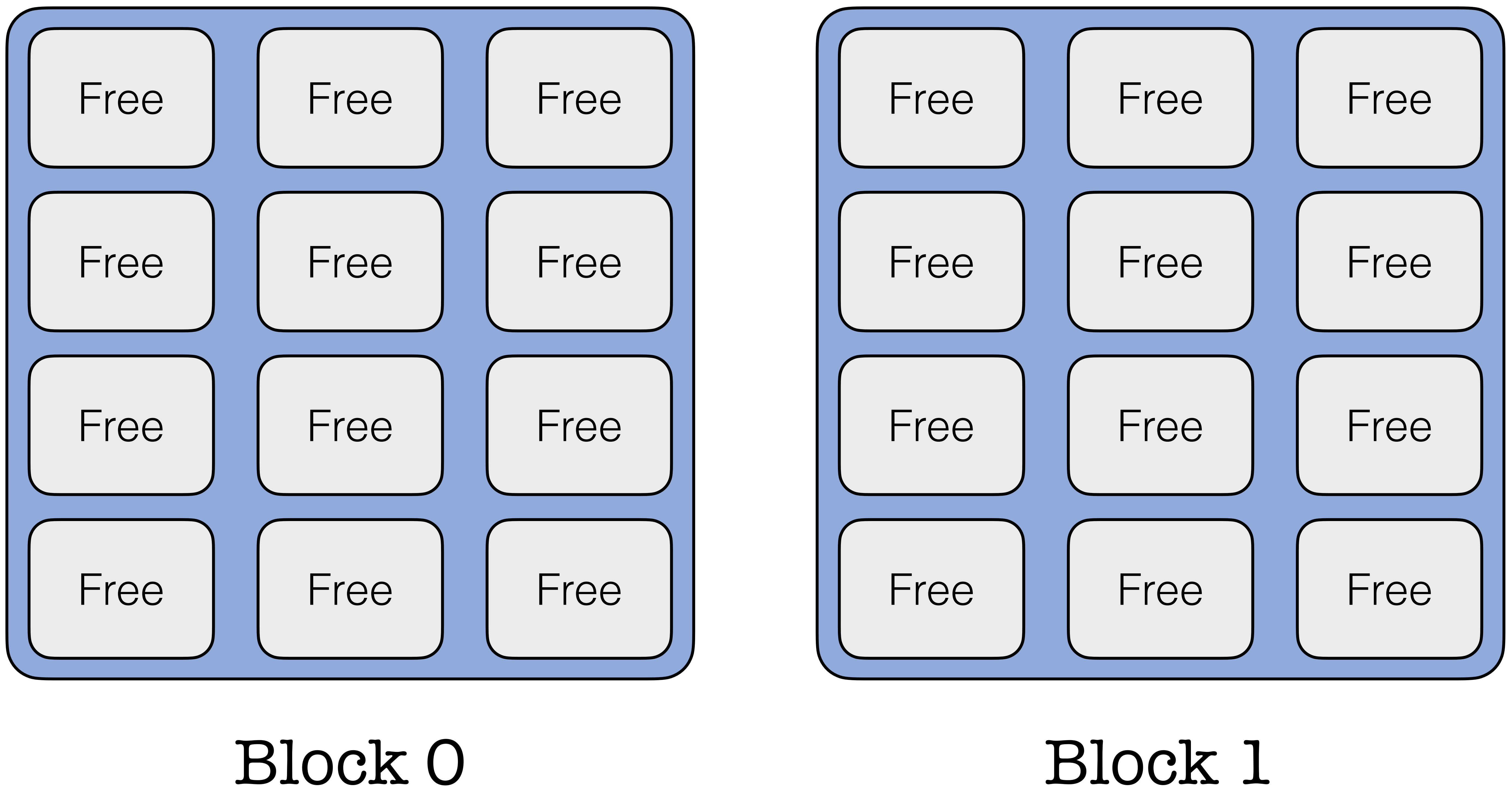


Brandeis UNIVERSITY

Writes in **SSDs**

Writes are out of place

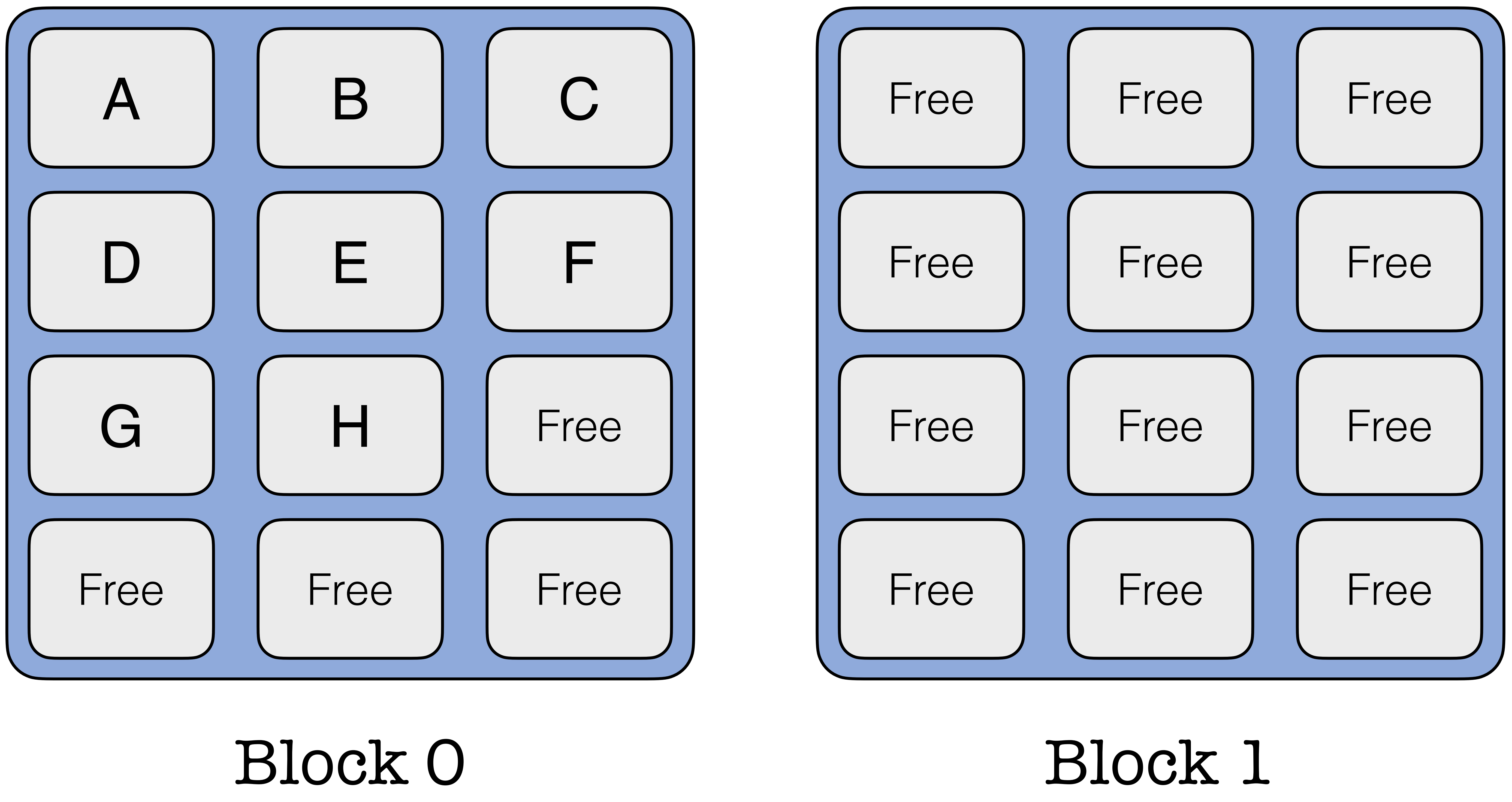
Insert
A, B, C, D, E, F, G, H



Writes in SSDs

Writes are out of place

Insert
A, B, C, D, E, F, G, H

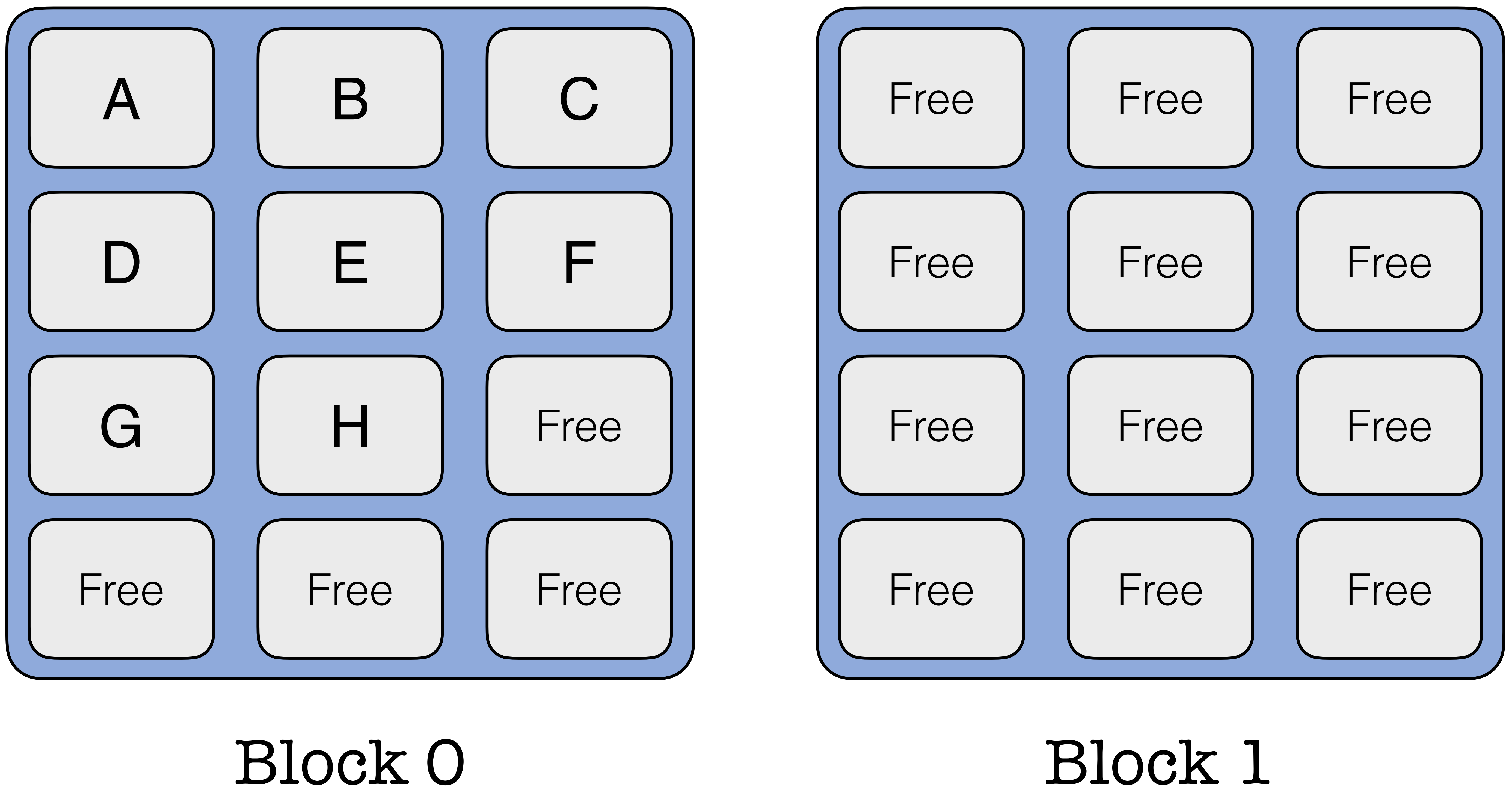


Writes in **SSDs**

Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D

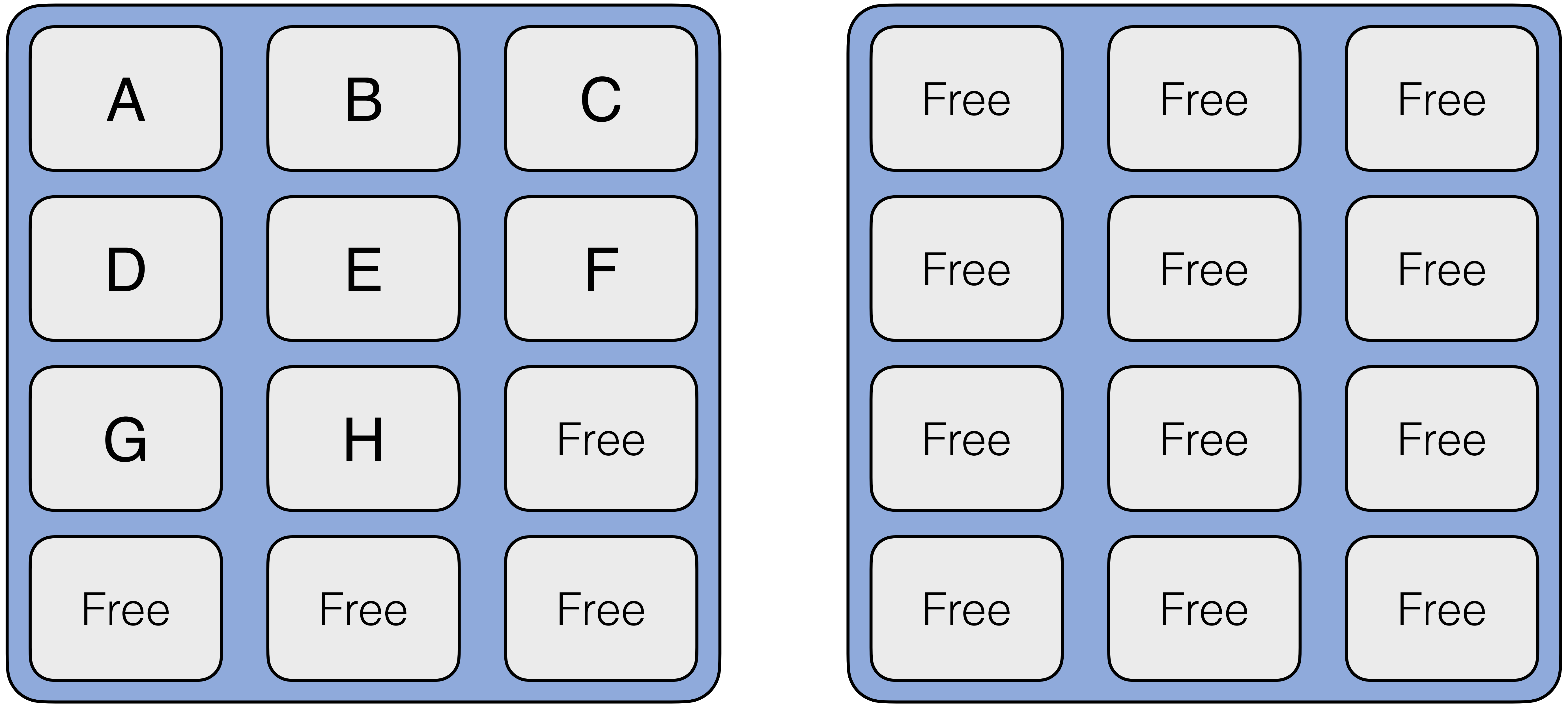


Writes in SSDs

Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D



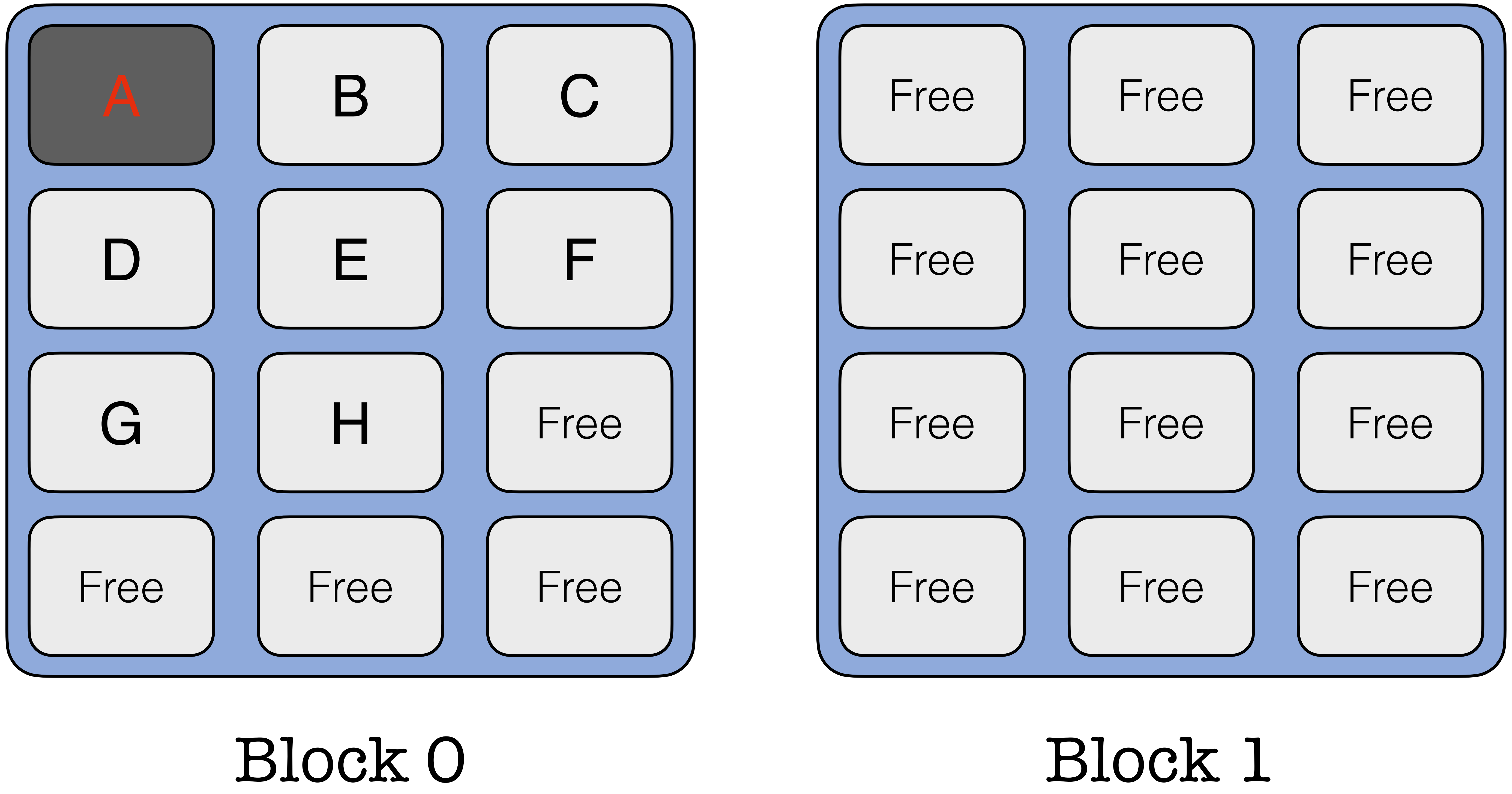
Block 0 Block 1

Writes in SSDs

Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D



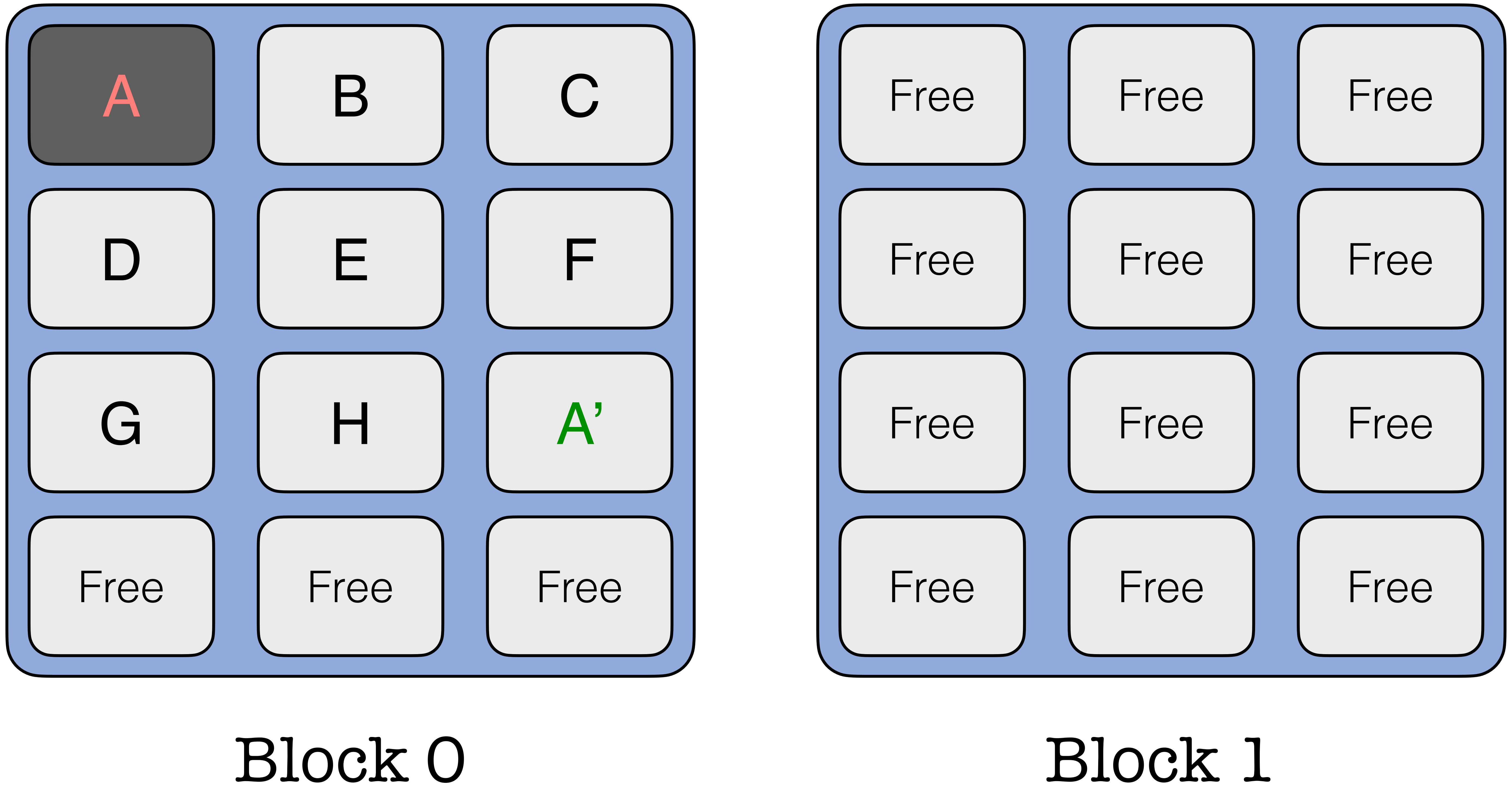
Brandeis University

Writes in **SSDs**

Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D

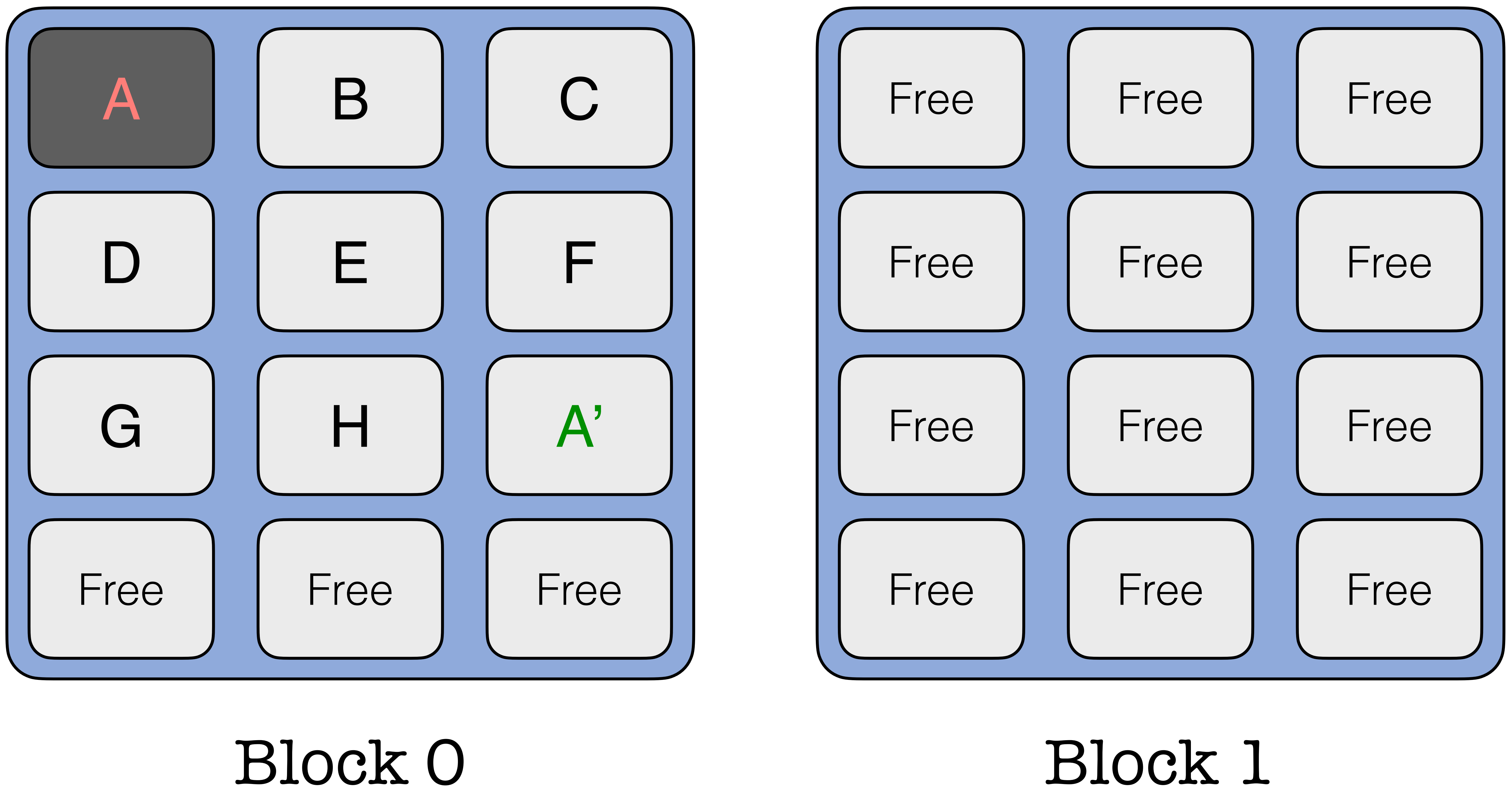


Writes in SSDs

Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D

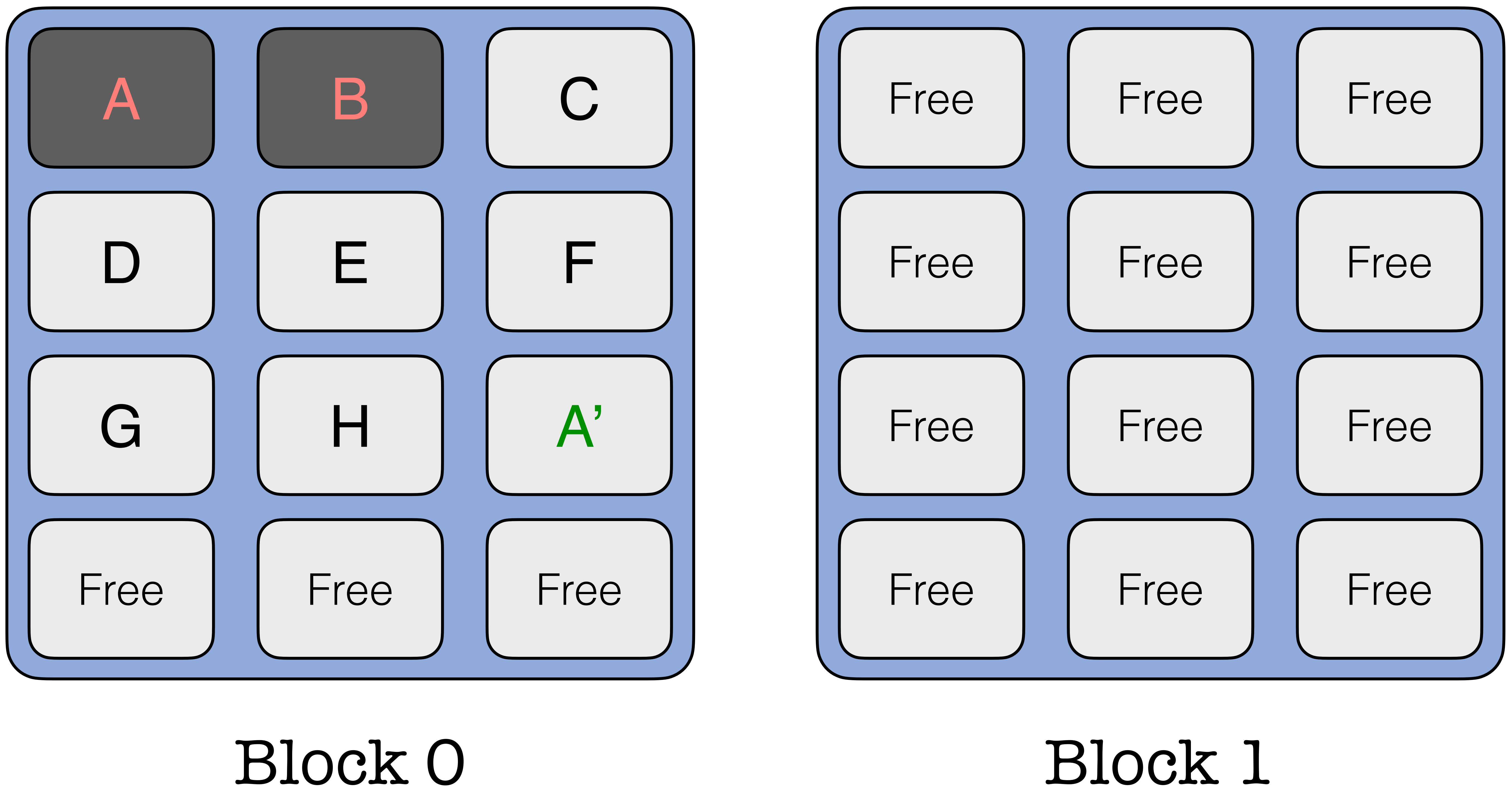


Writes in SSDs

Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D



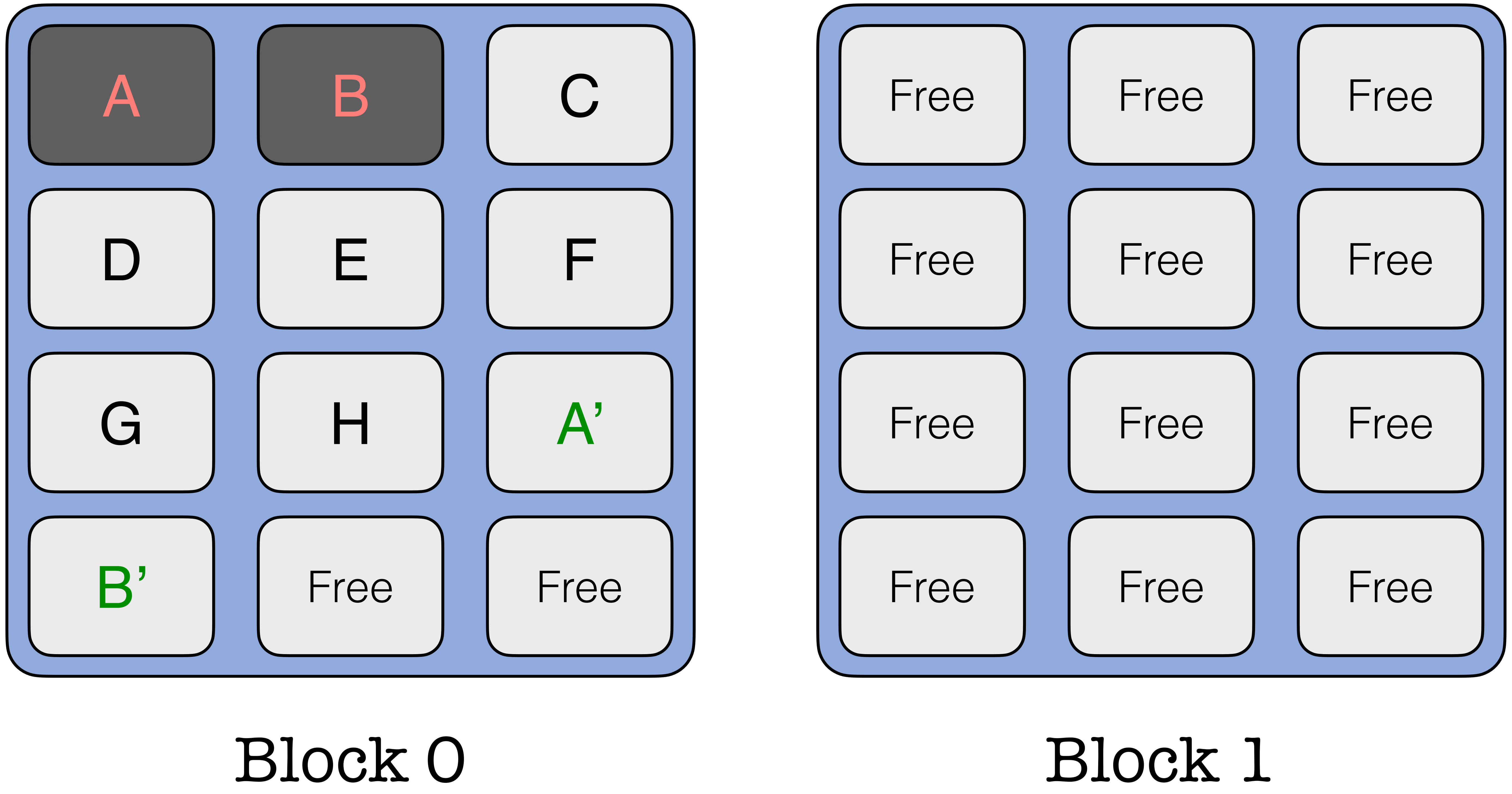
Brandeis UNIVERSITY

Writes in SSDs

Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D

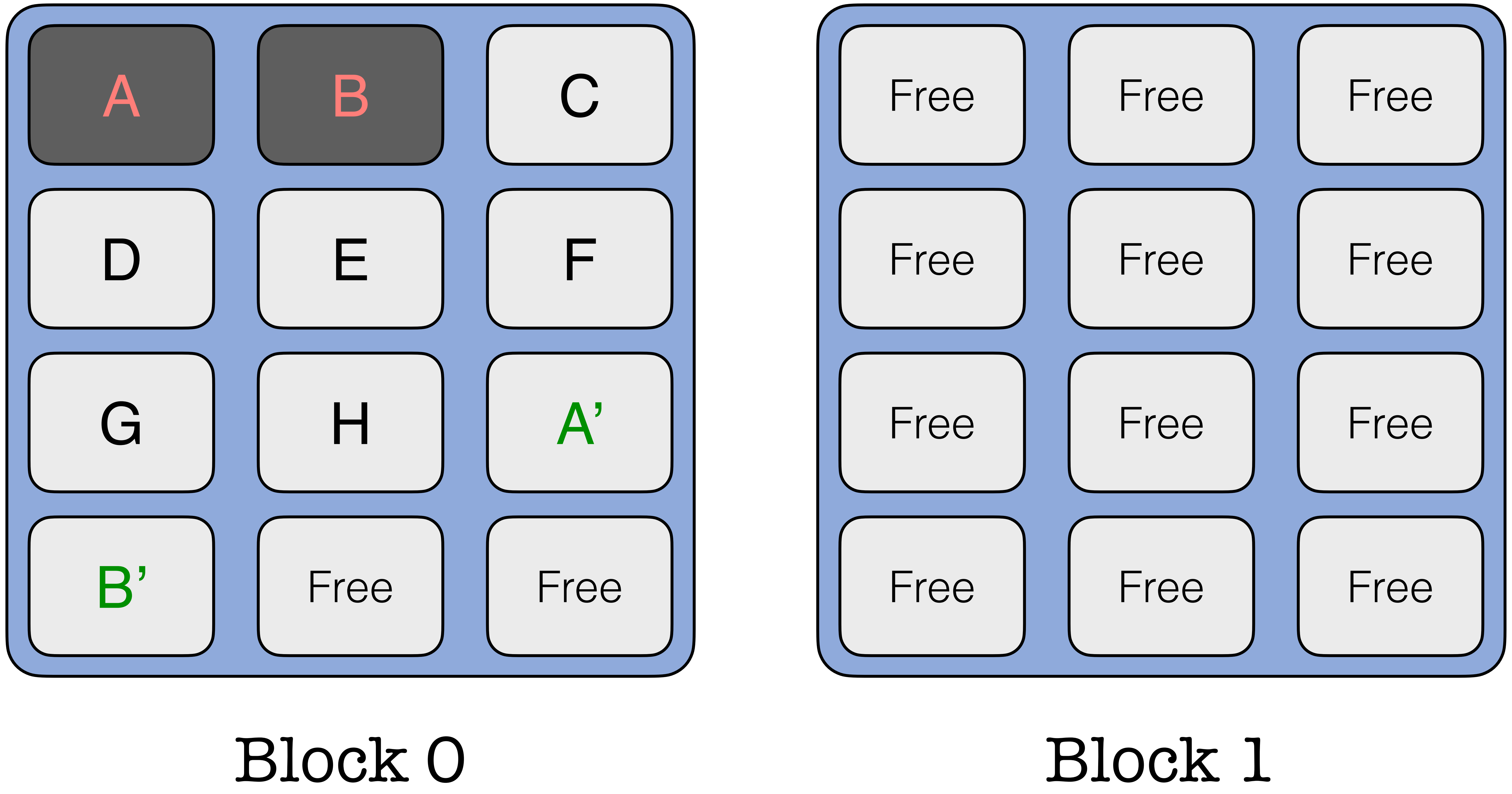


Writes in SSDs

Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D



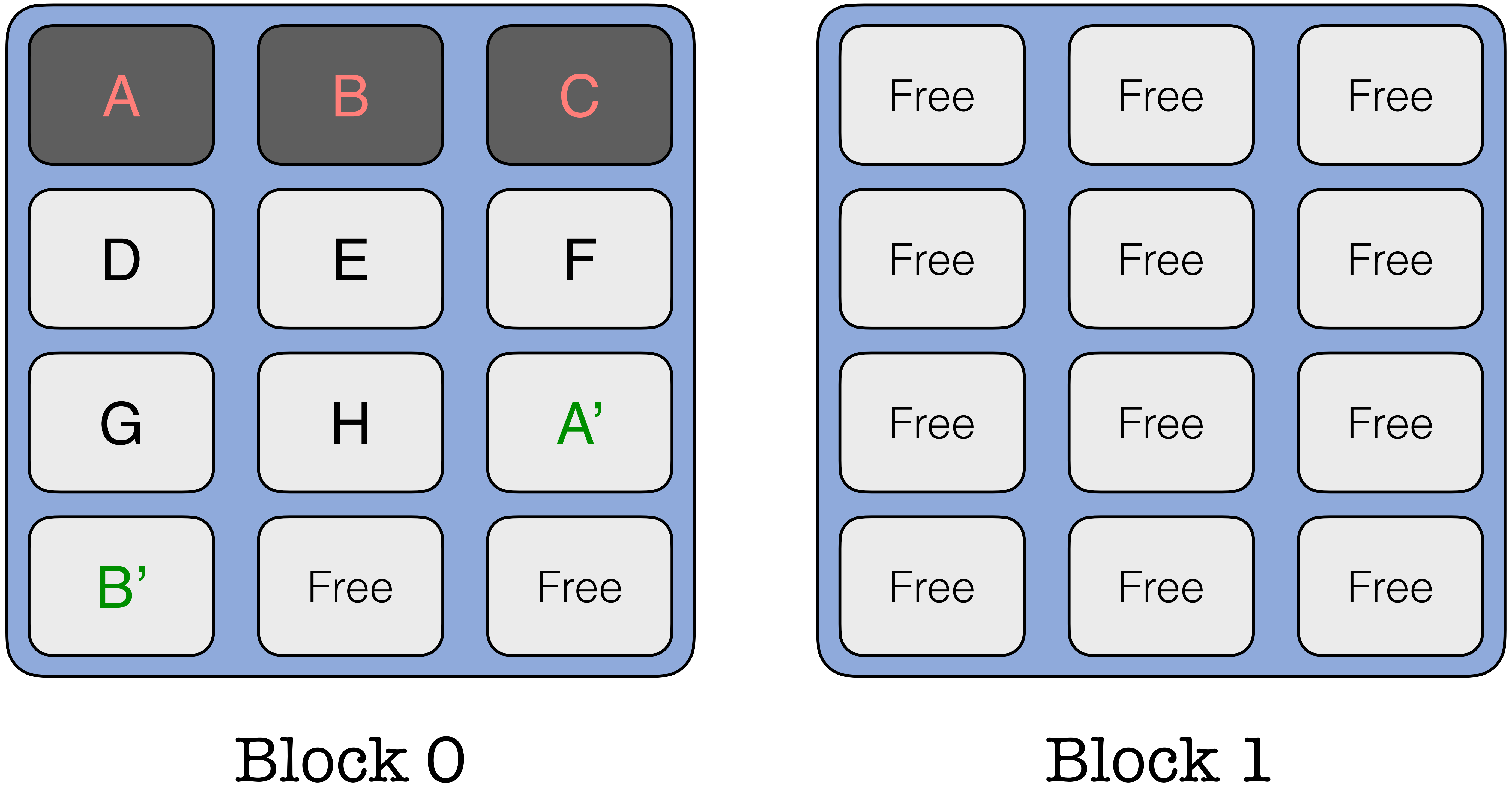
Brandeis UNIVERSITY

Writes in **SSDs**

Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D

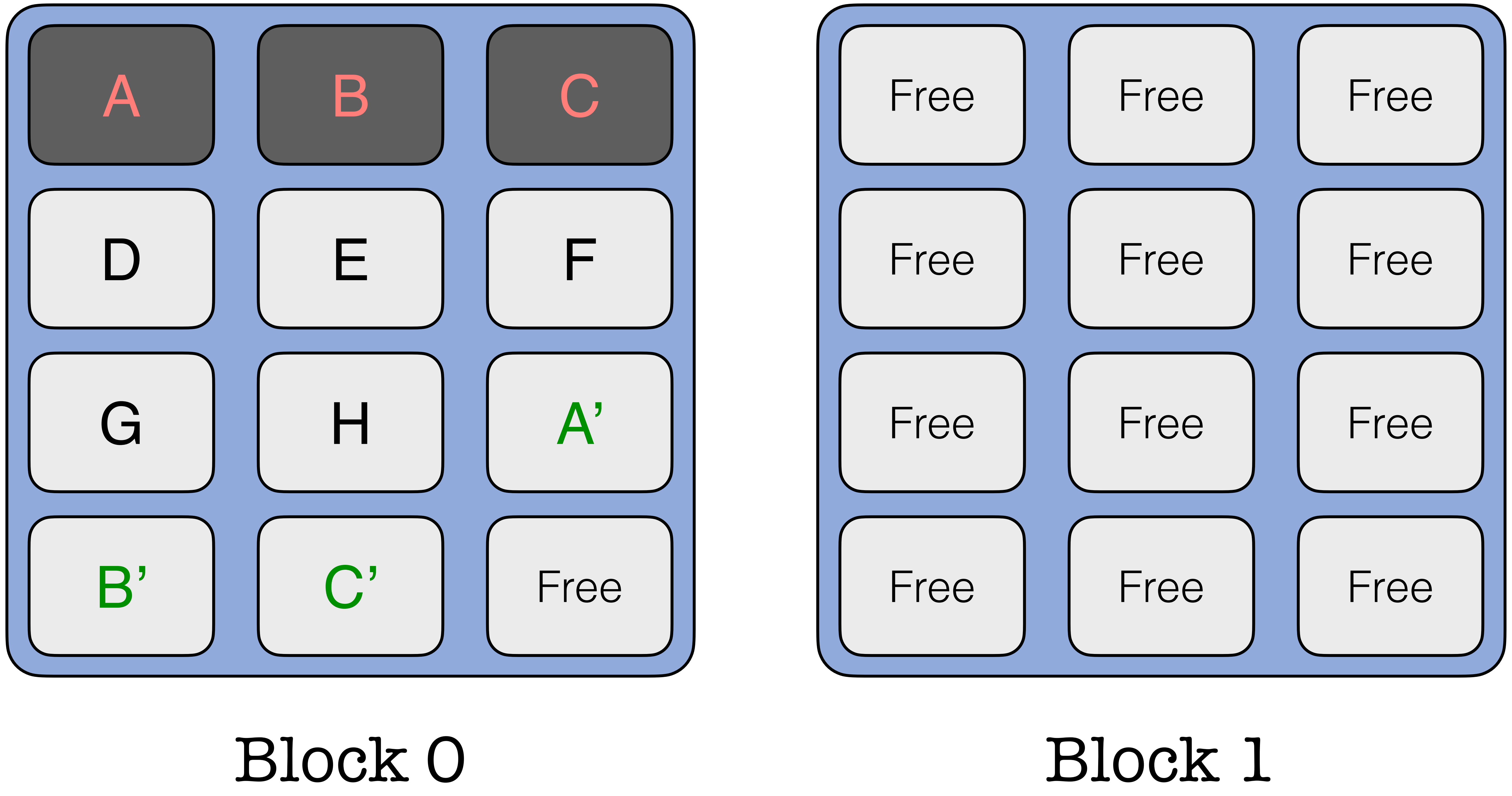


Writes in SSDs

Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D

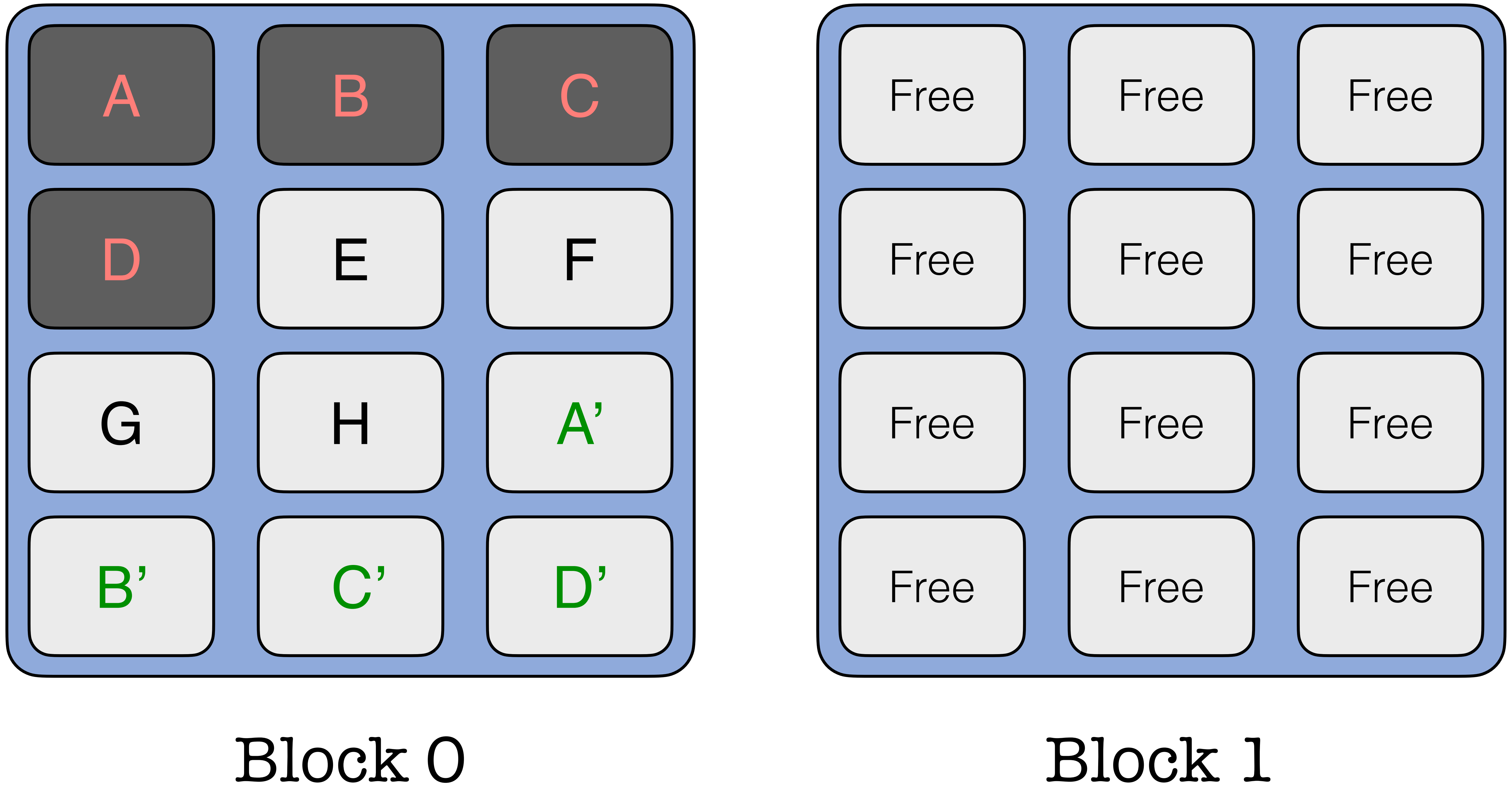


Writes in SSDs

Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D



Brandeis University

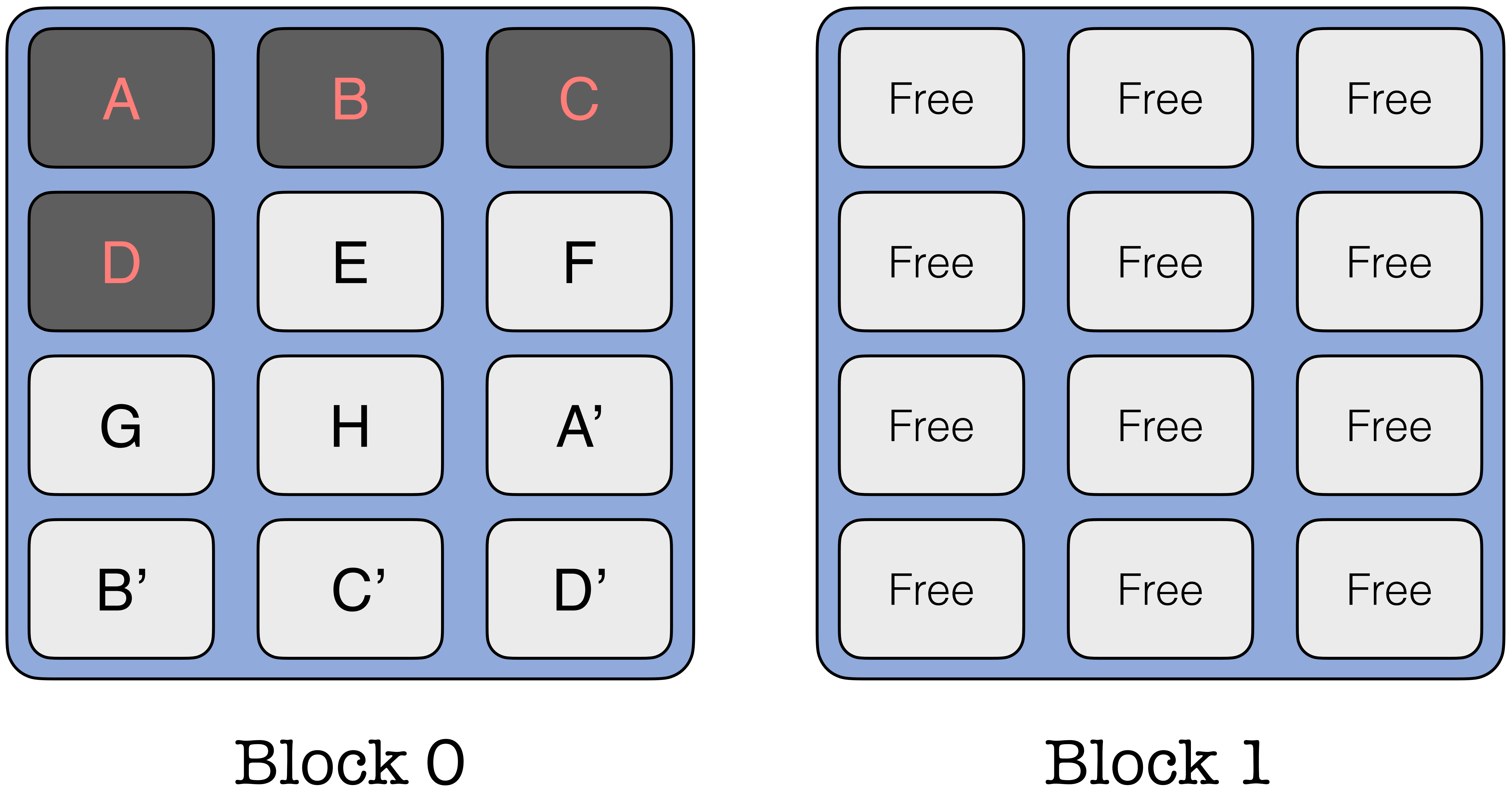
Writes in SSDs

Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D

Insert
M, N, O, P, Q, R



Writes in **SSDs**

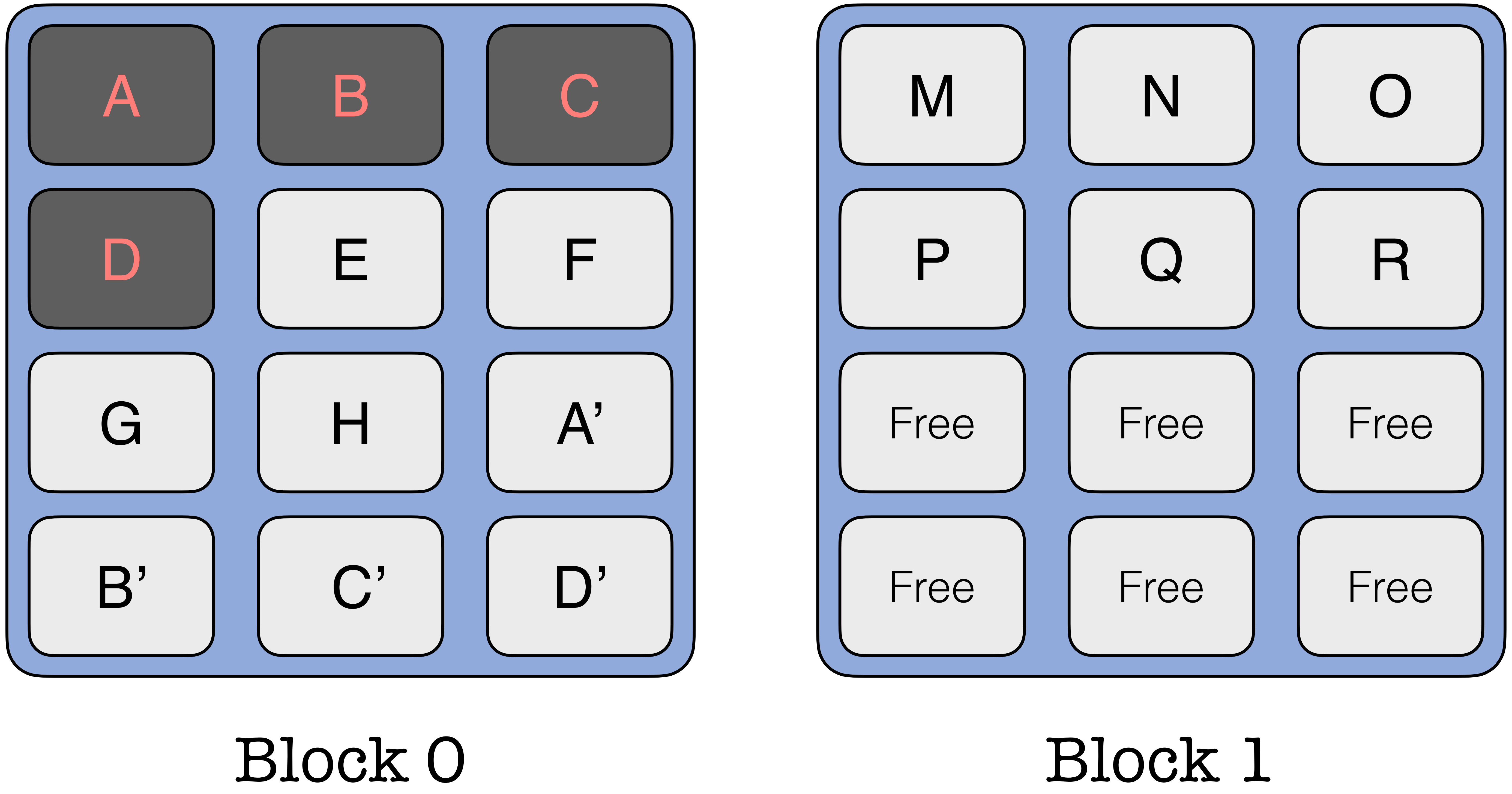
Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D

Insert
M, N, O, P, Q, R

Update
M, N, O, P, Q, R



Writes in **SSDs**

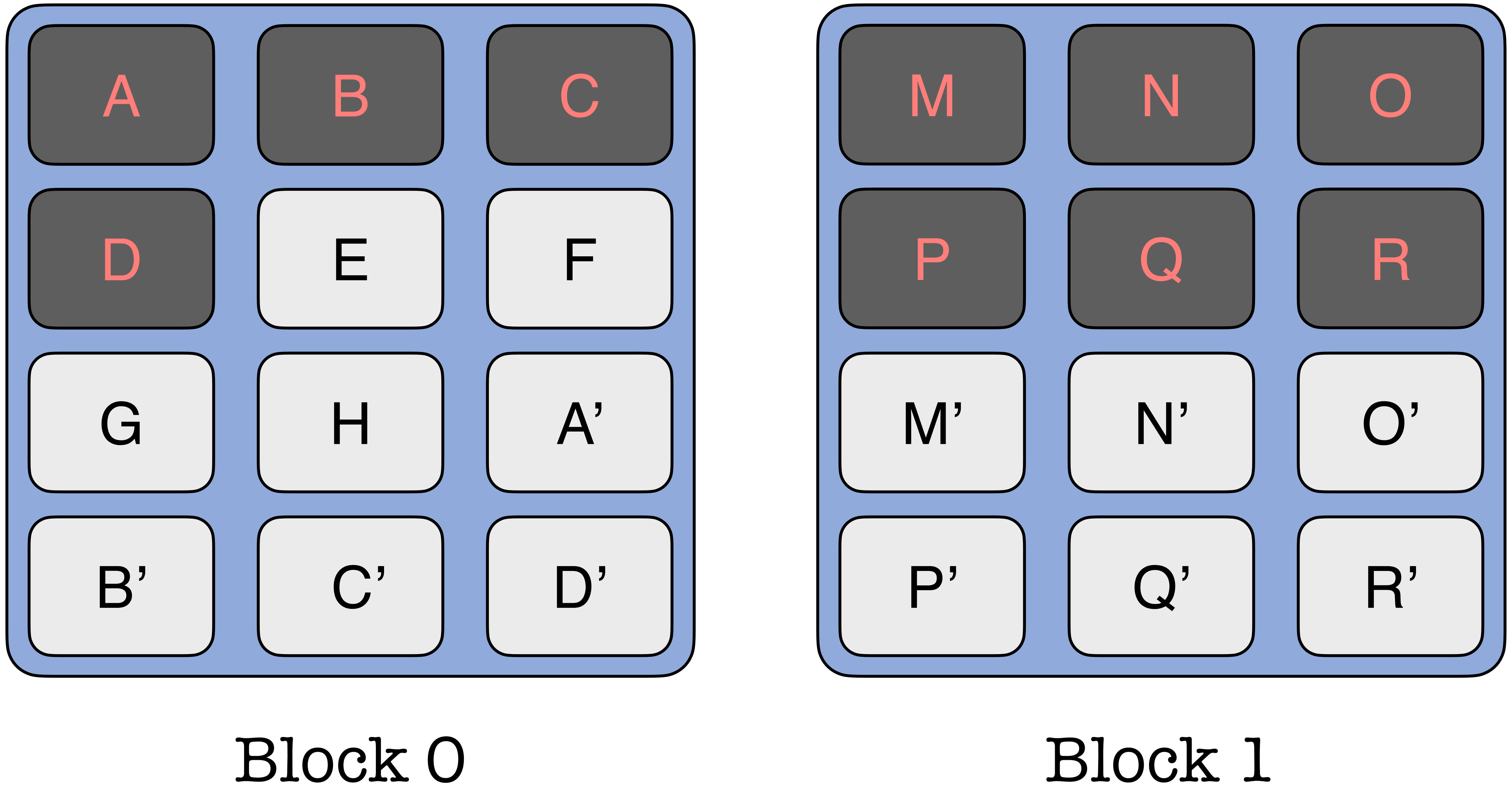
Writes are out of place

Insert
A, B, C, D, E, F, G, H

Update
A, B, C, D

Insert
M, N, O, P, Q, R

Update
M, N, O, P, Q, R



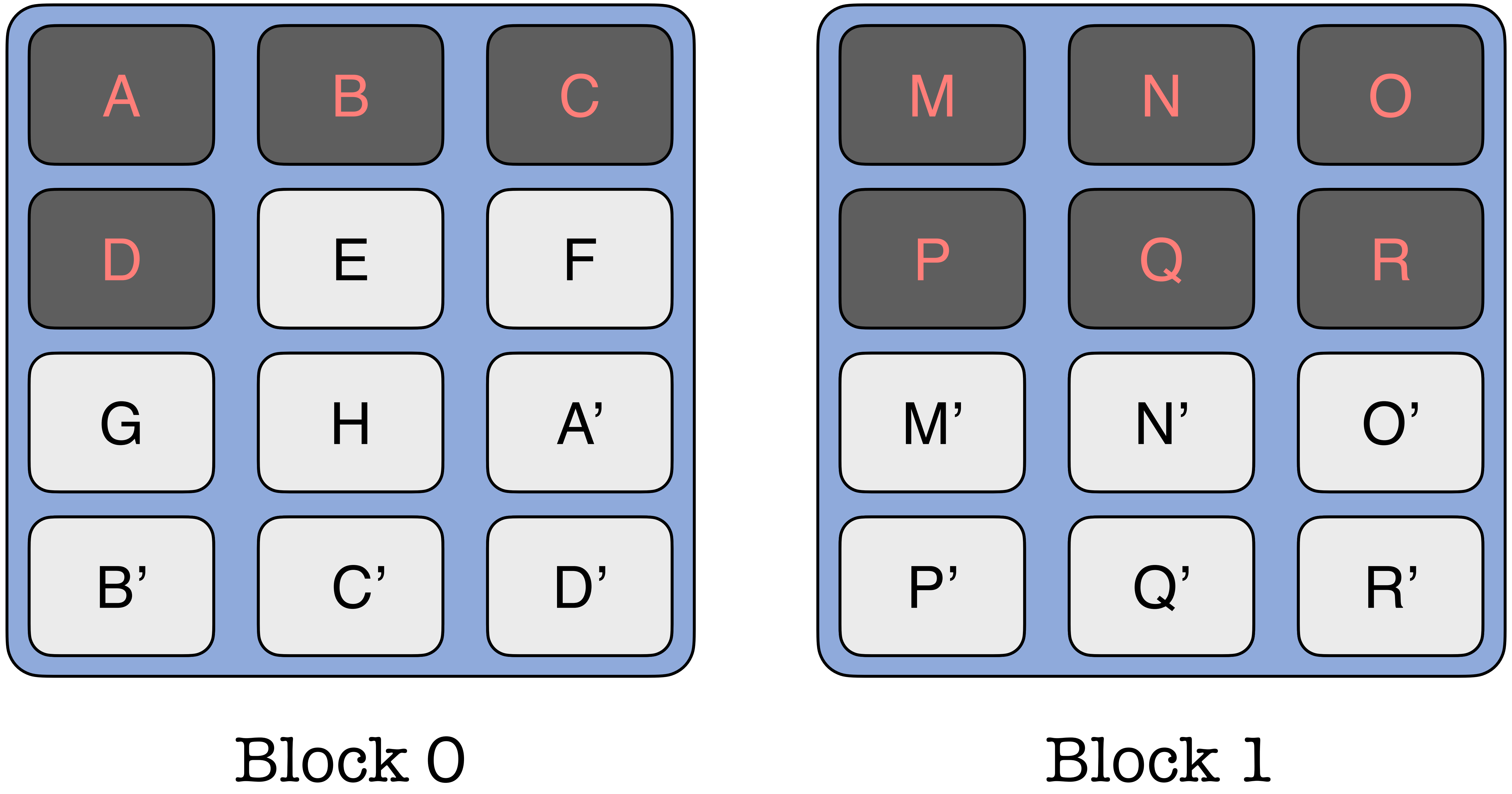
Writes in SSDs

Writes are out of place

What if all blocks are full?

Garbage Collection

1 **keep track** of valid pages



Brandeis University

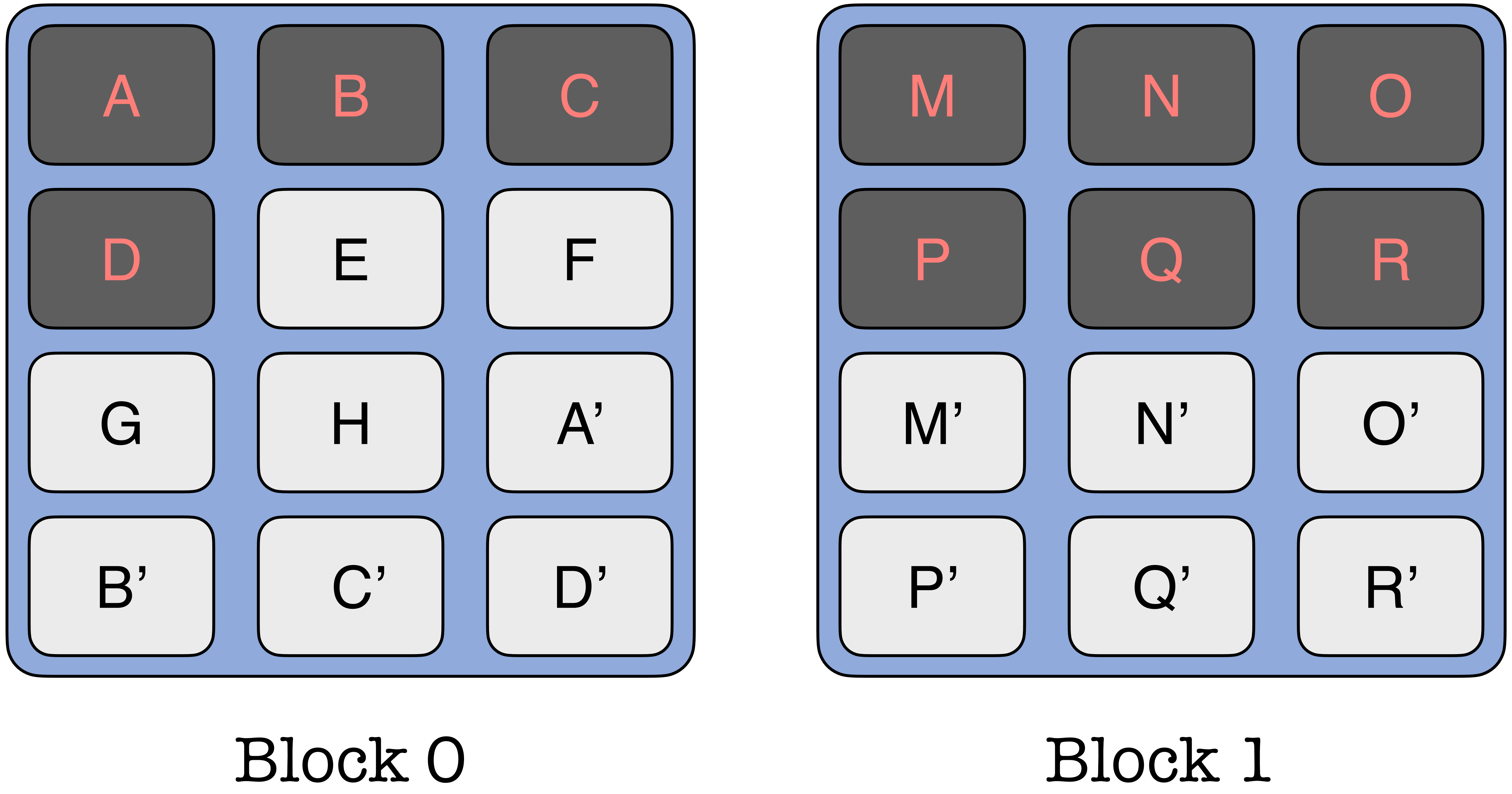
Writes in **SSDs**

Writes are out of place

What if all blocks are full

Garbage Collection

1. **keep track** of valid pages
2. **erase** all pages



valid pages | E | F | G | H | A’ | B’ | C’ | D’ | M’ | N’ | O’ | P’ | Q’ | R’ |

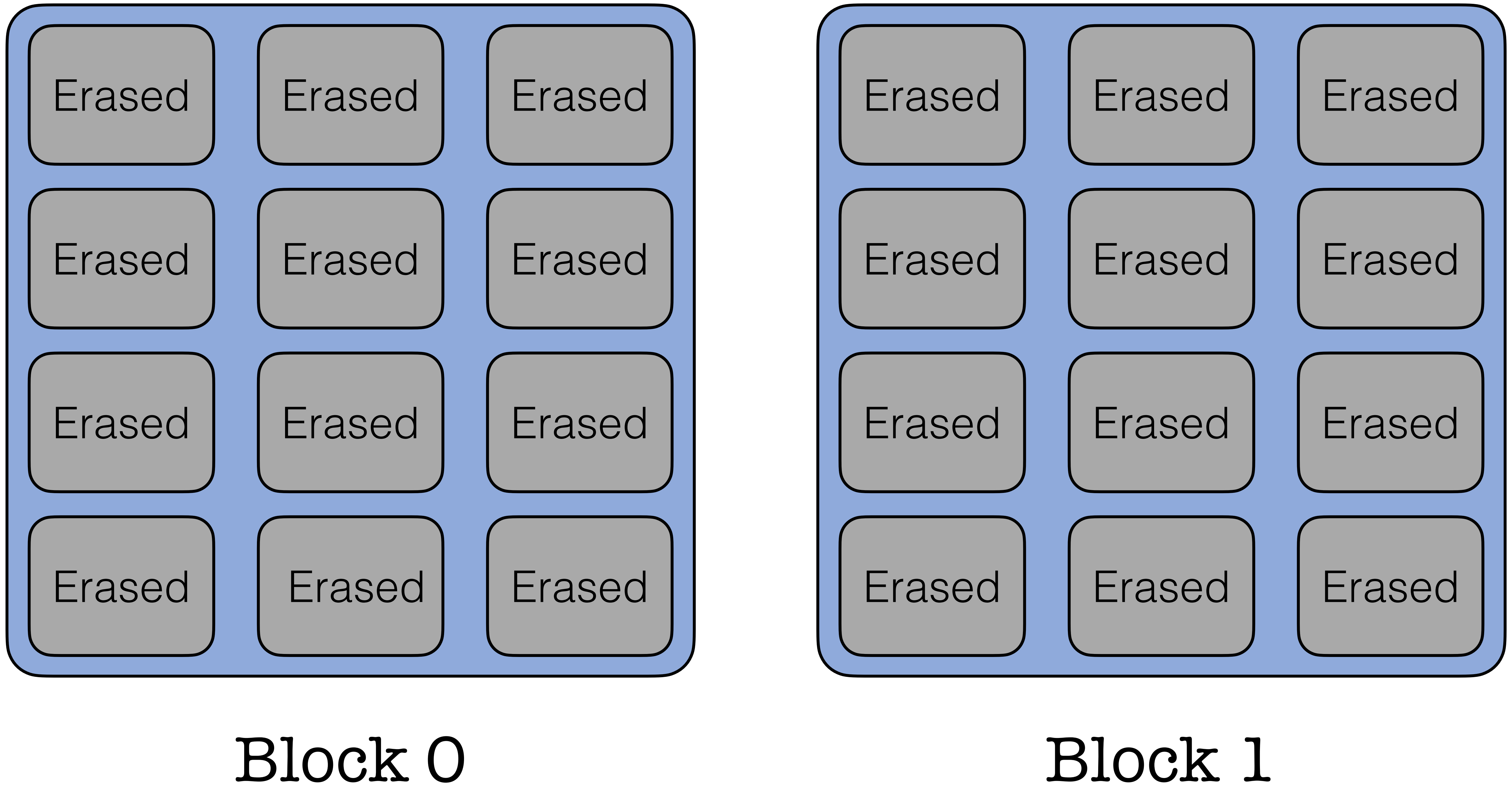
Writes in **SSDs**

Writes are out of place

What if all blocks are full?

Garbage Collection

1. **keep track** of valid pages
2. **erase** all pages



valid pages | E | F | G | H | A' | B' | C' | D' | M' | N' | O' | P' | Q' | R'

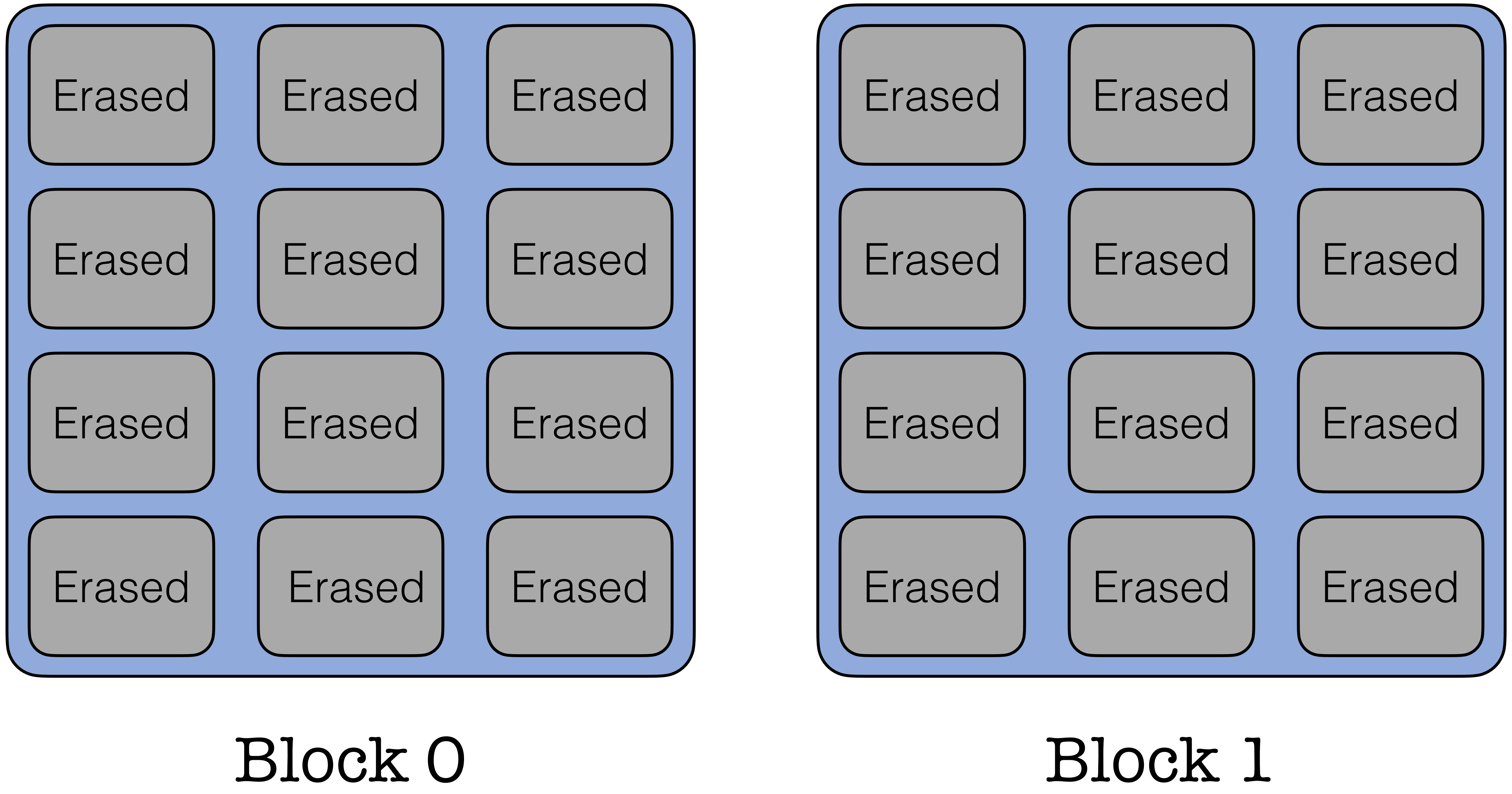
Writes in **SSDs**

Writes are out of place

What if all blocks are full?

Garbage Collection

1. **keep track** of valid pages
2. **erase** all pages
3. **write back** valid pages



valid pages | E | F | G | H | A' | B' | C' | D' | M' | N' | O' | P' | Q' | R' |

Brandeis UNIVERSITY

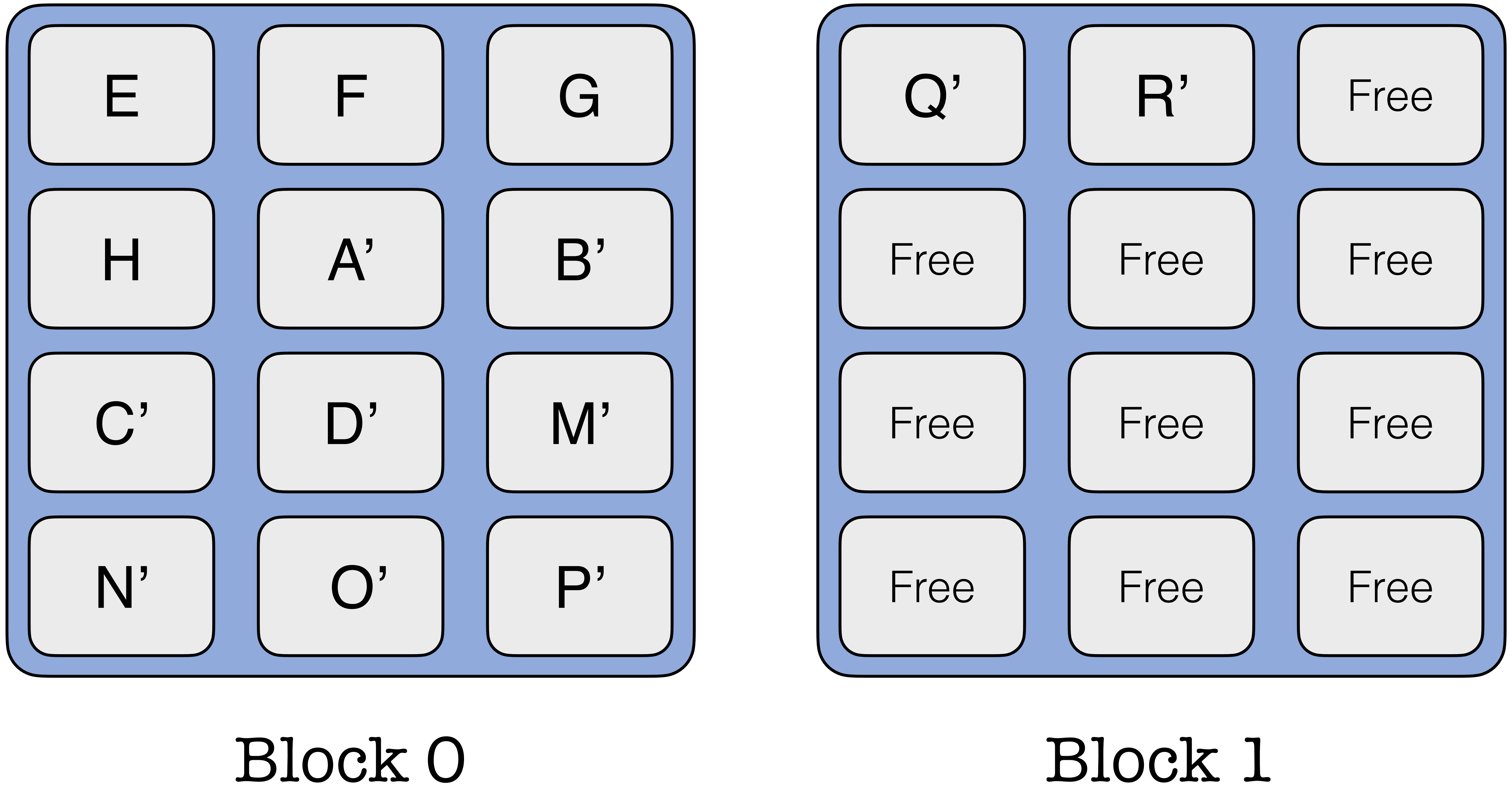
Writes in **SSDs**

Writes are out of place

What if all blocks are full

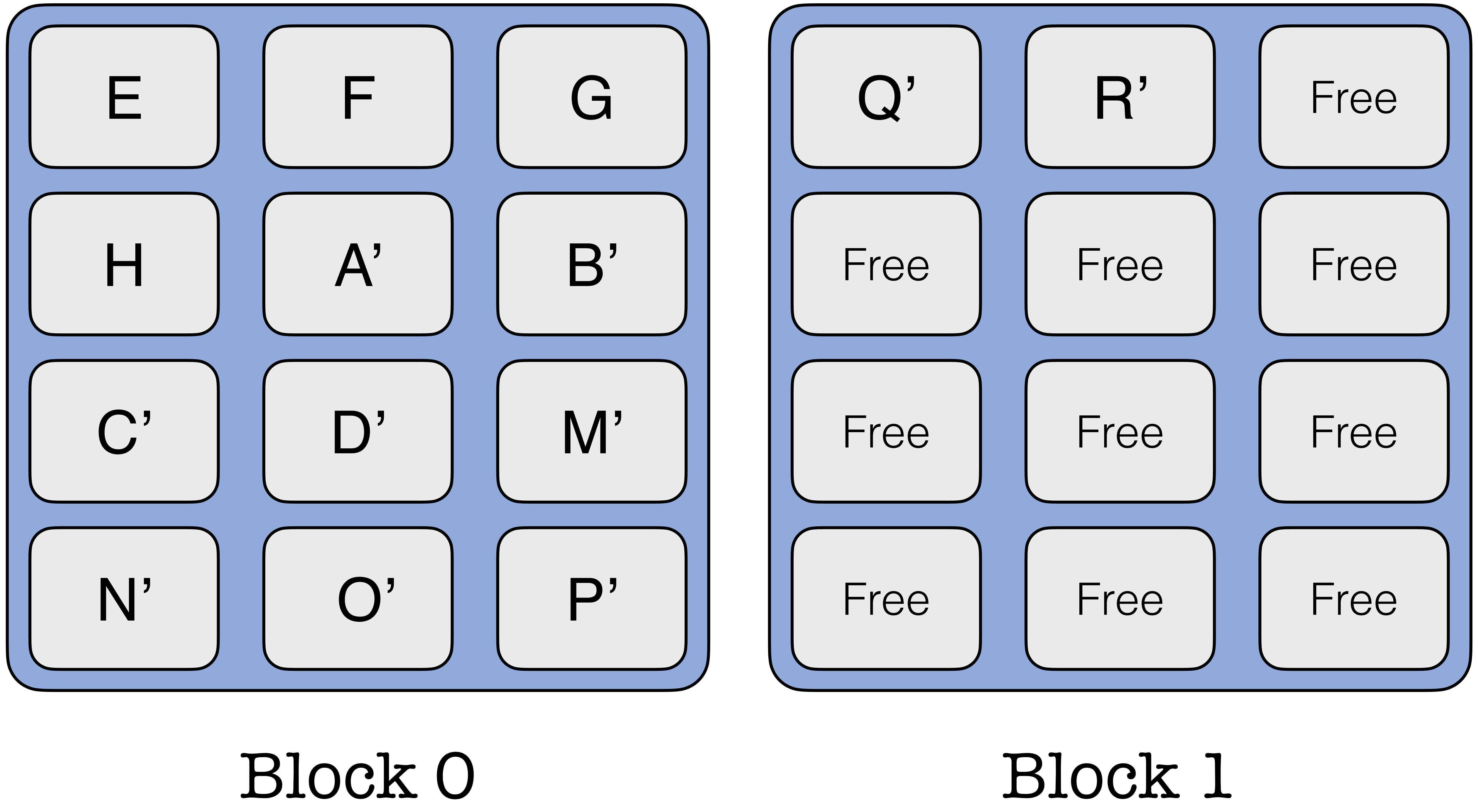
Garbage Collection

1. **keep track** of valid pages
2. **erase** all pages
3. **write back** valid pages



Writes in SSDs

Writes are out of place



Thought Experiment 7

Any **limitations**?

1. high **write amplification**
2. limited **device lifetime**

Now, think of

LSM on SSDs!

Summary

The key takeaways

Data placement is critical!

be it on **storage**, **memory** or **cache**!

Flash operates **electronically** and is **fast**!

updates are **out of place**; suffers from **high write amplification**

Understanding the **underlying hardware** is critical for performance

hardware-aware indexes, **caching**, and **access methods**

Next time in COSI 167A

Row stores vs. Column stores

ACEing the Bufferpool Management Paradigm for Modern Storage Devices

Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine

COSI 167A
Advanced Data Systems

Class 18

Indexing + Modern Hardware Trends

Prof. Subhadeep Sarkar

Brandeis UNIVERSITY

https://ssd-brandeis.github.io/COSI-167A/