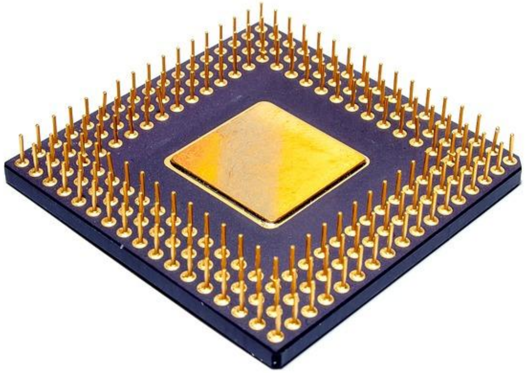
Storage and Memory Hierarchy

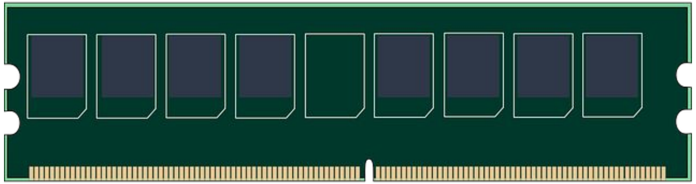
CS165

What is the memory hierarchy ?

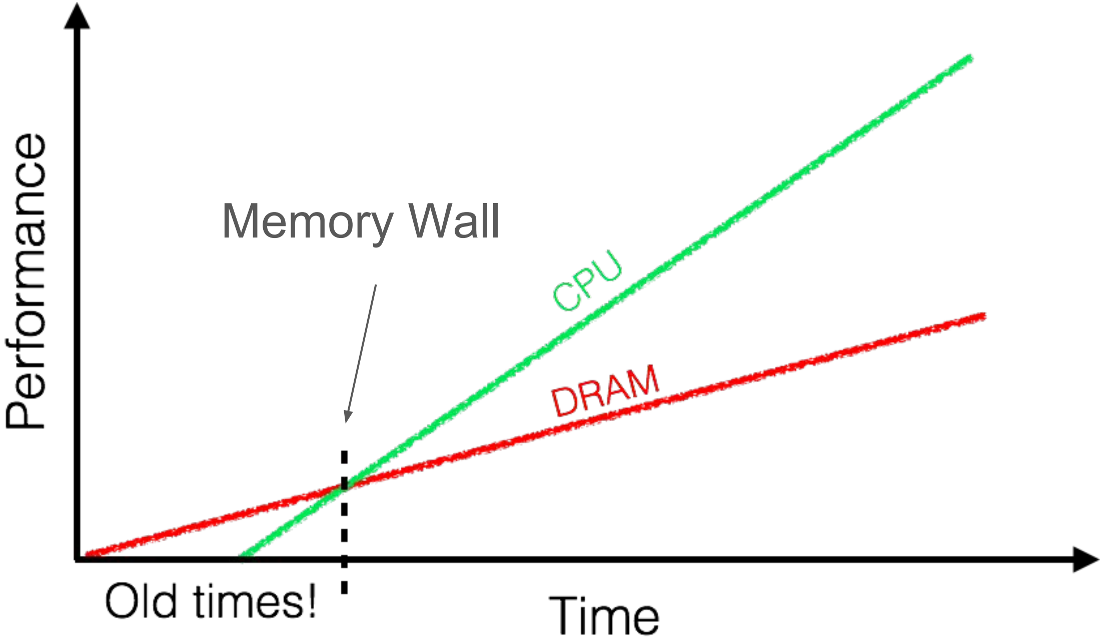

L1
<1ns
L2
~3ns
L3
~10ns
Main Memory
~100ns
Flash
~100µs
HDD / Shingled HDD
~2ms
Faster
Smaller
More expensive
Bigger
Cheaper
Slower

Why have such a hierarchy?



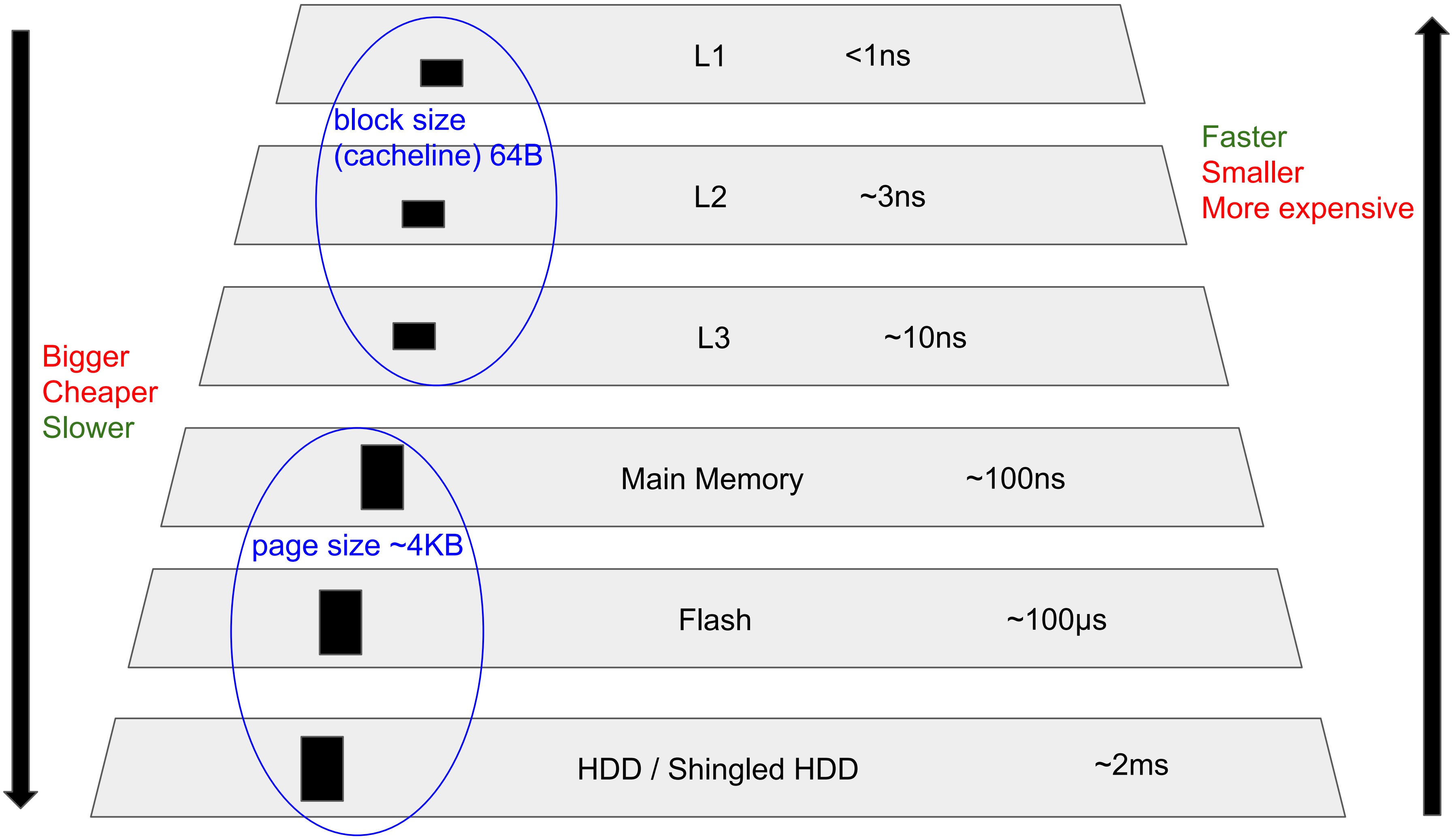


Which one is faster?

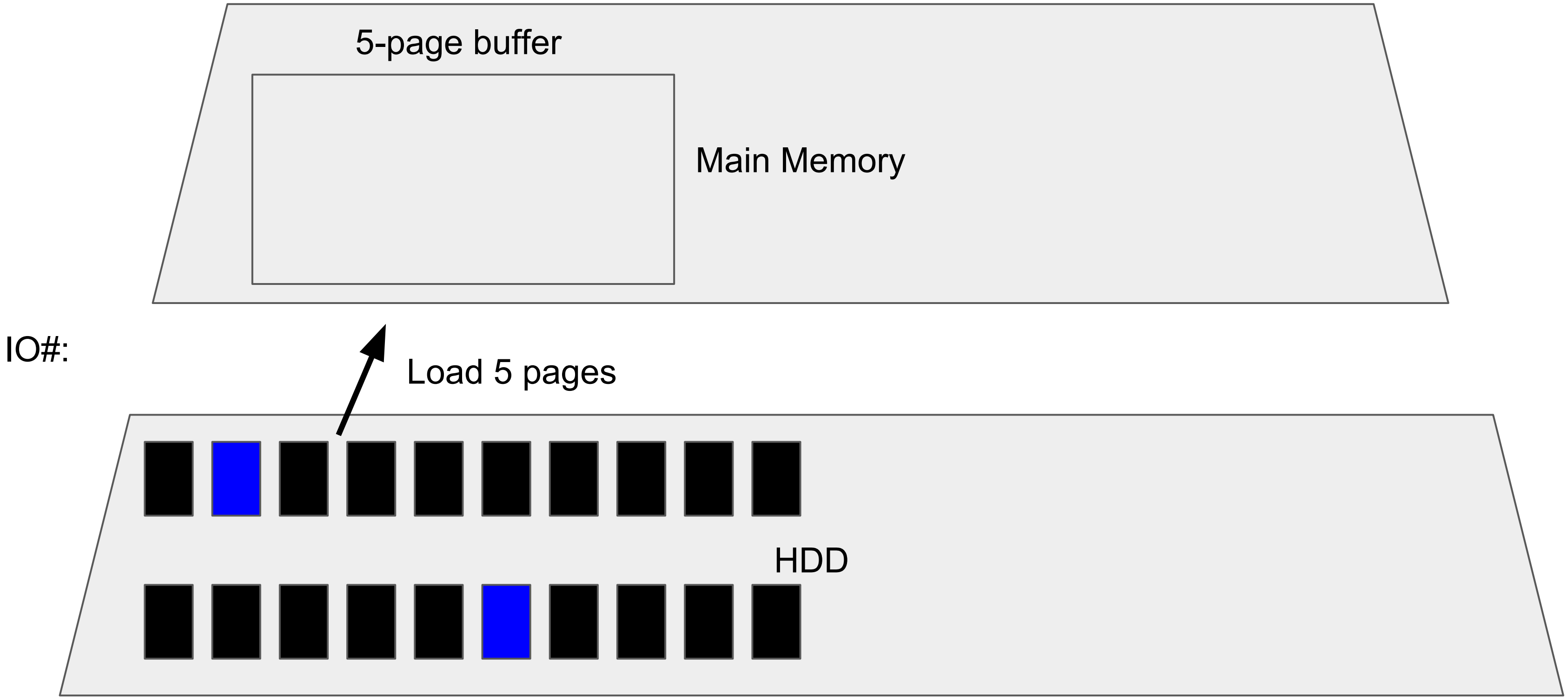


As the gap grows, we need a *deeper* memory hierarchy

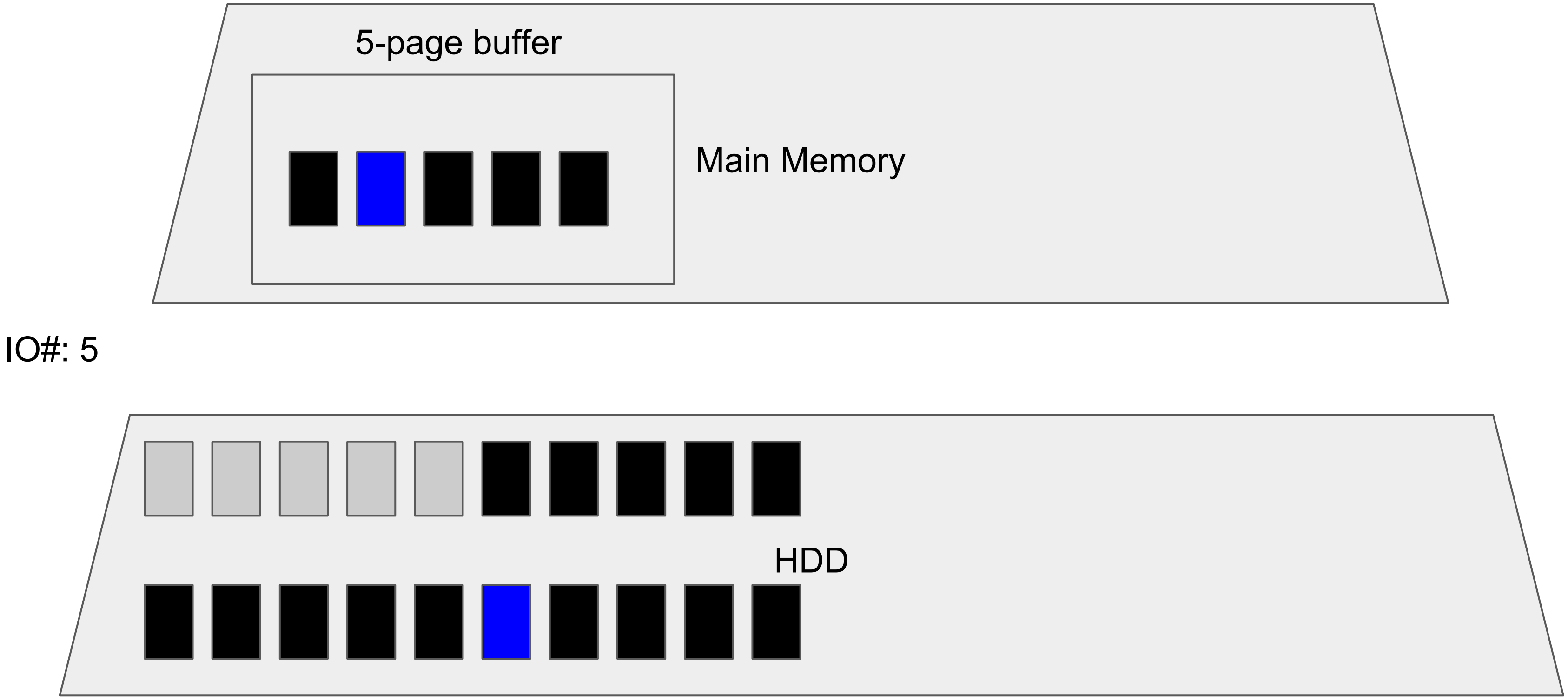
Access Granularity


L1
<1ns
block size
(cacheline) 64B
L2
~3ns
L3
~10ns
Main Memory
~100ns
page size ~4KB
Flash
~100μs
HDD / Shingled HDD
~2ms
Faster
Smaller
More expensive
Bigger
Cheaper
Slower

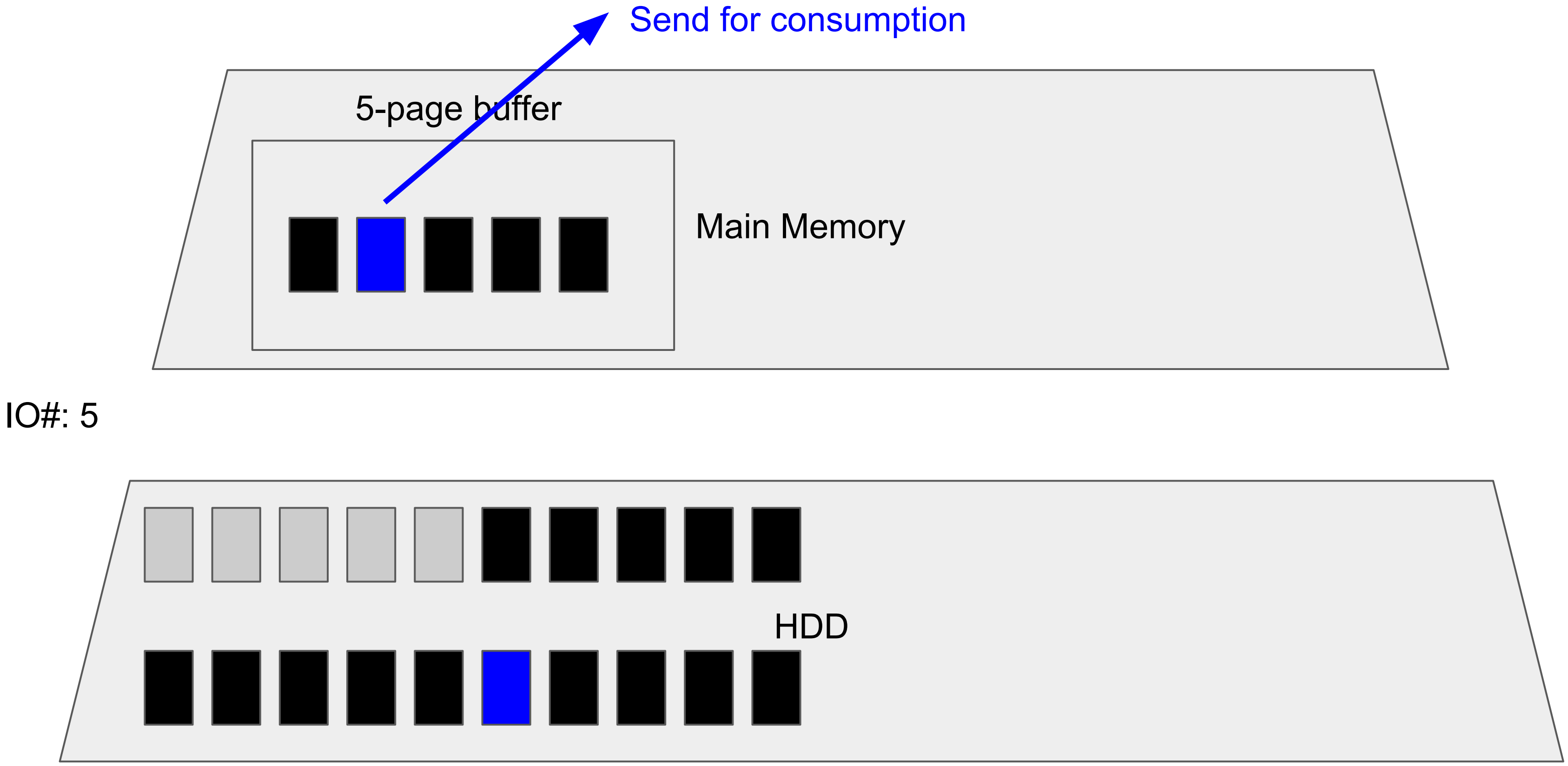
IO cost: Scanning a relation to select 10%



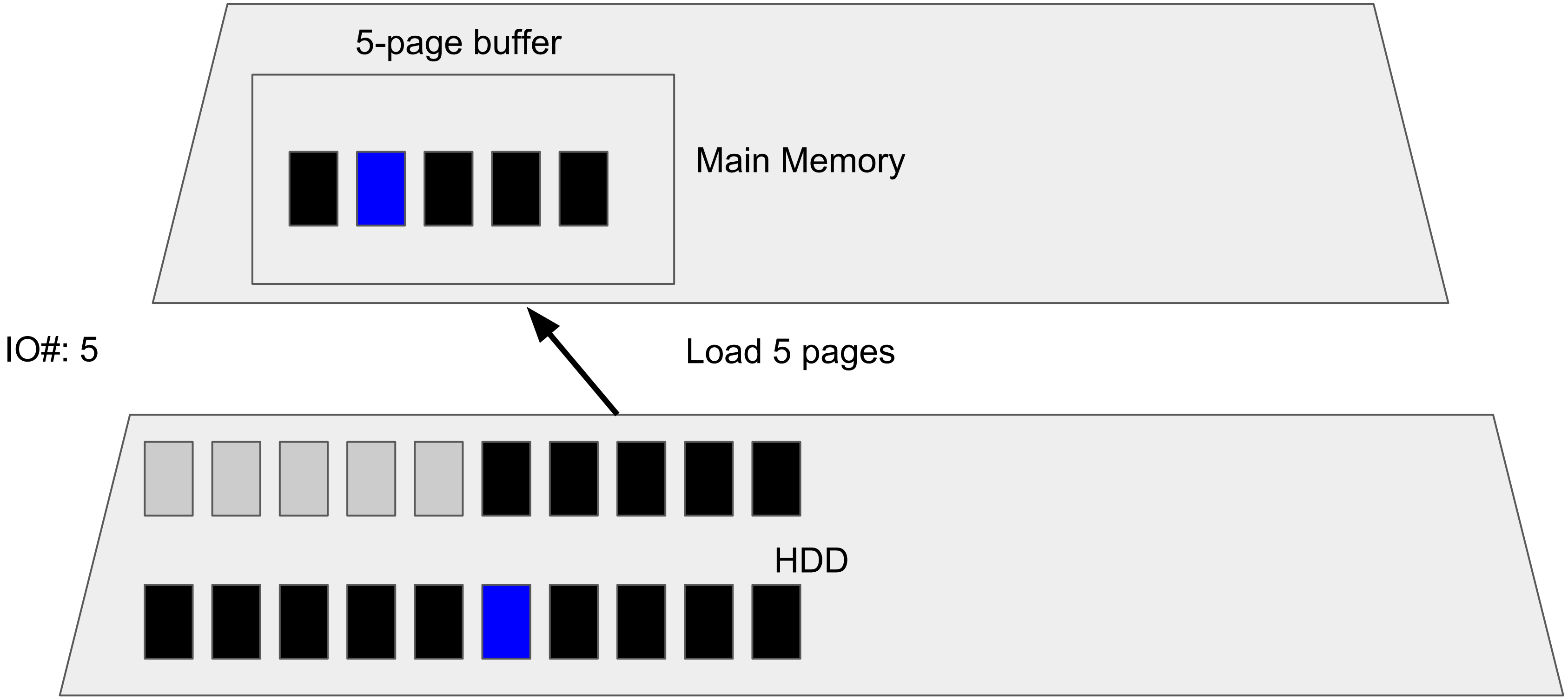
IO cost: Scanning a relation to select 10%



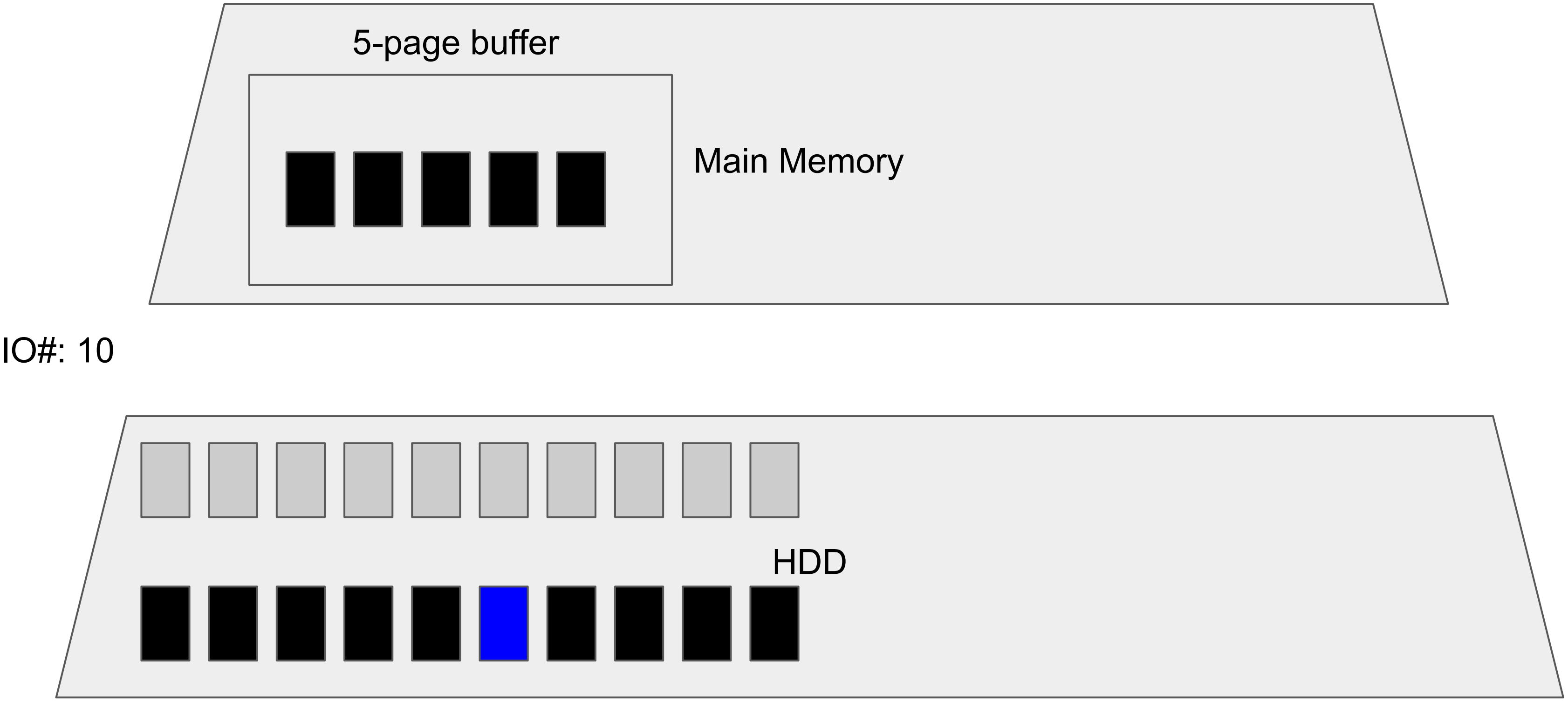
IO cost: Scanning a relation to select 10%



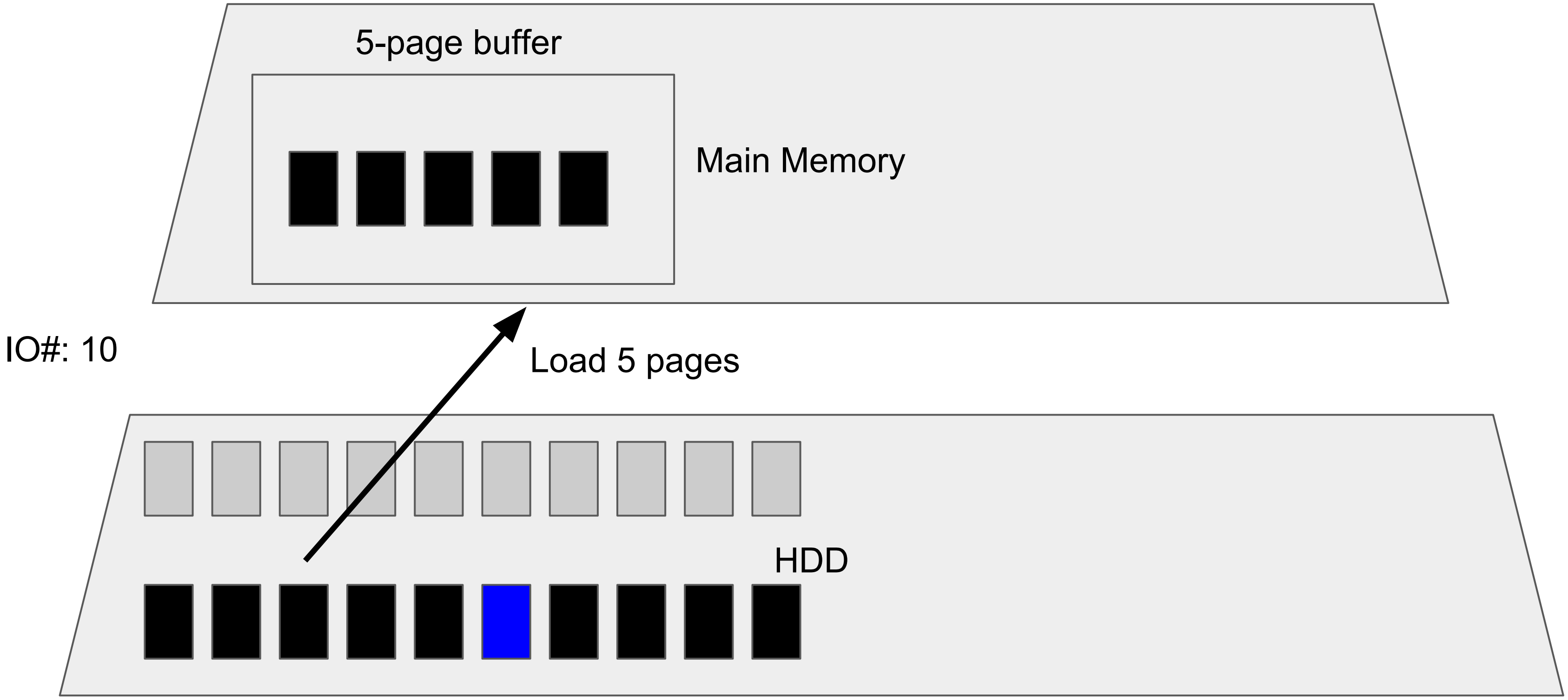
IO cost: Scanning a relation to select 10%



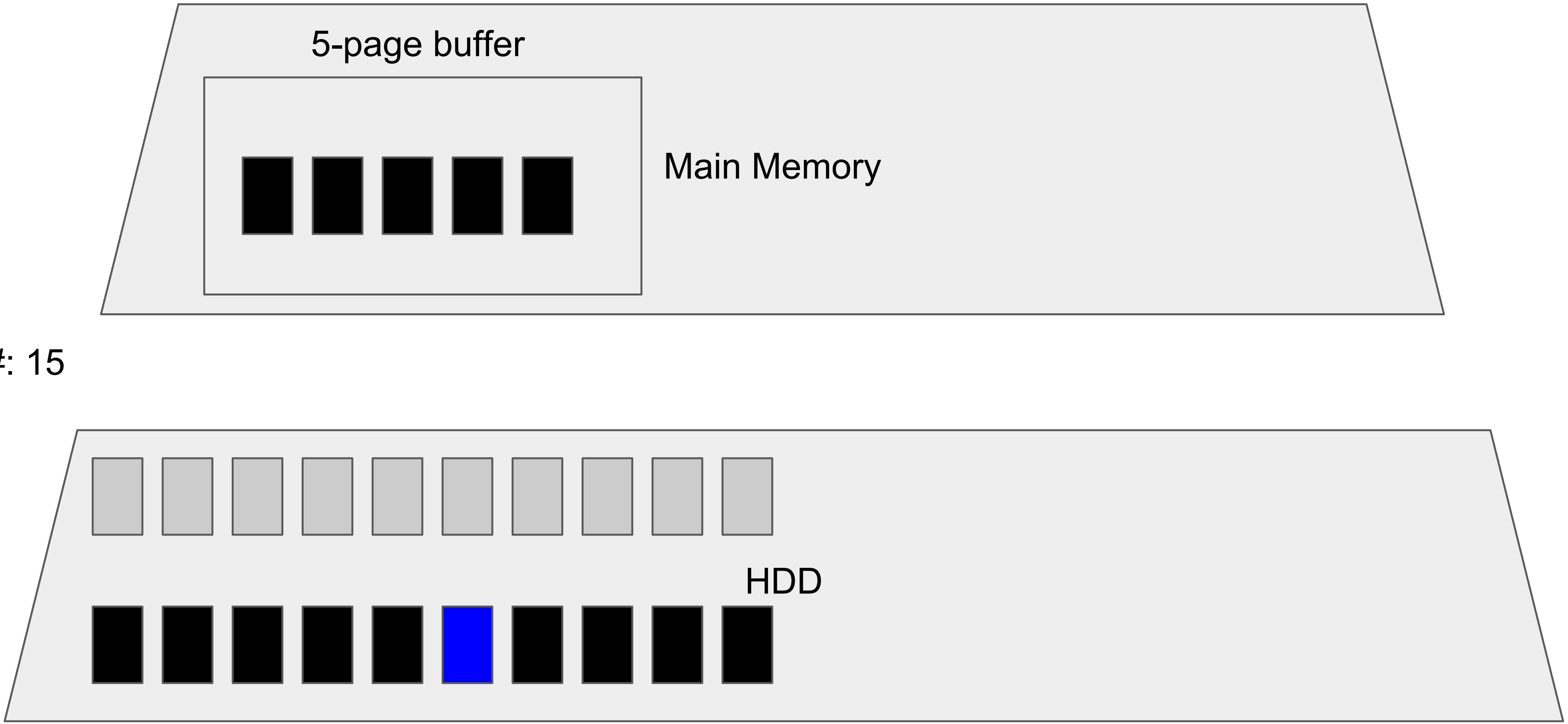
IO cost: Scanning a relation to select 10%



IO cost: Scanning a relation to select 10%

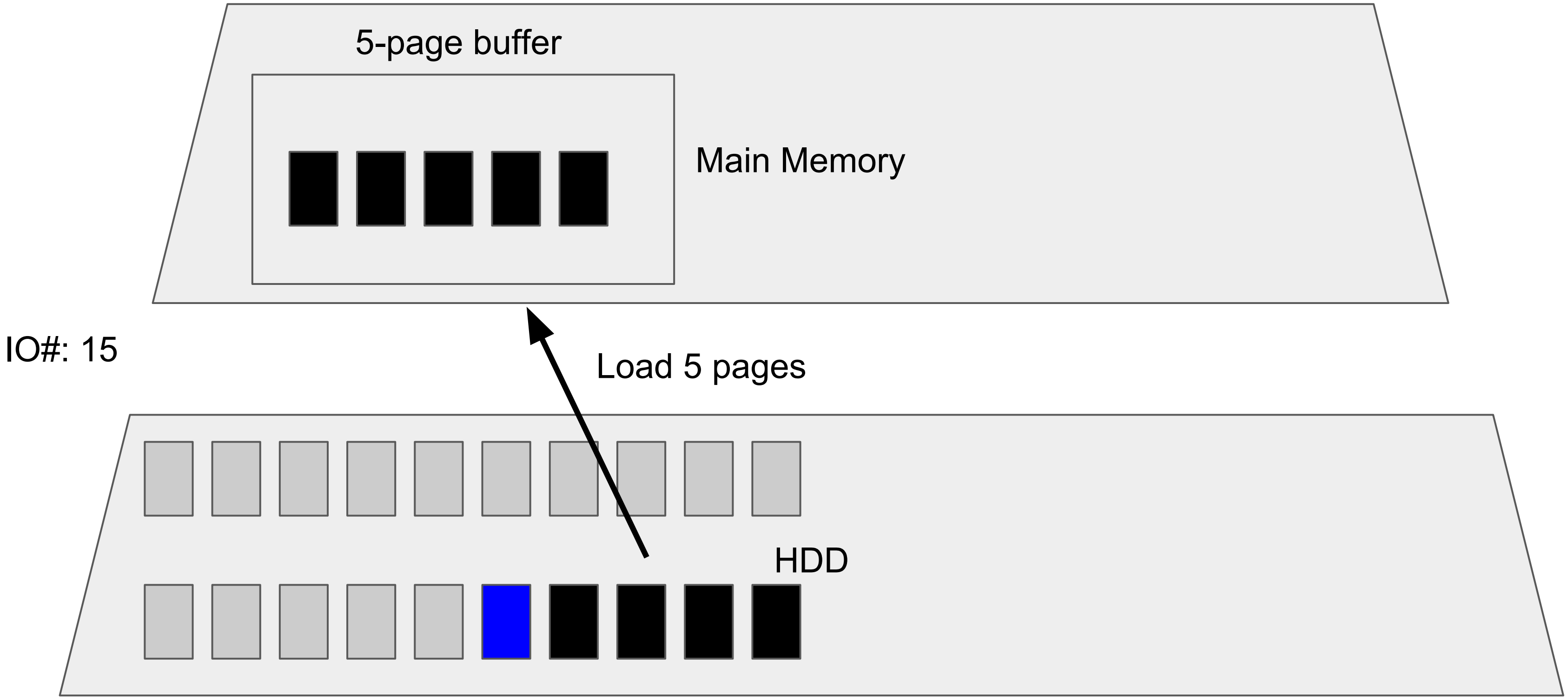


IO cost: Scanning a relation to select 10%

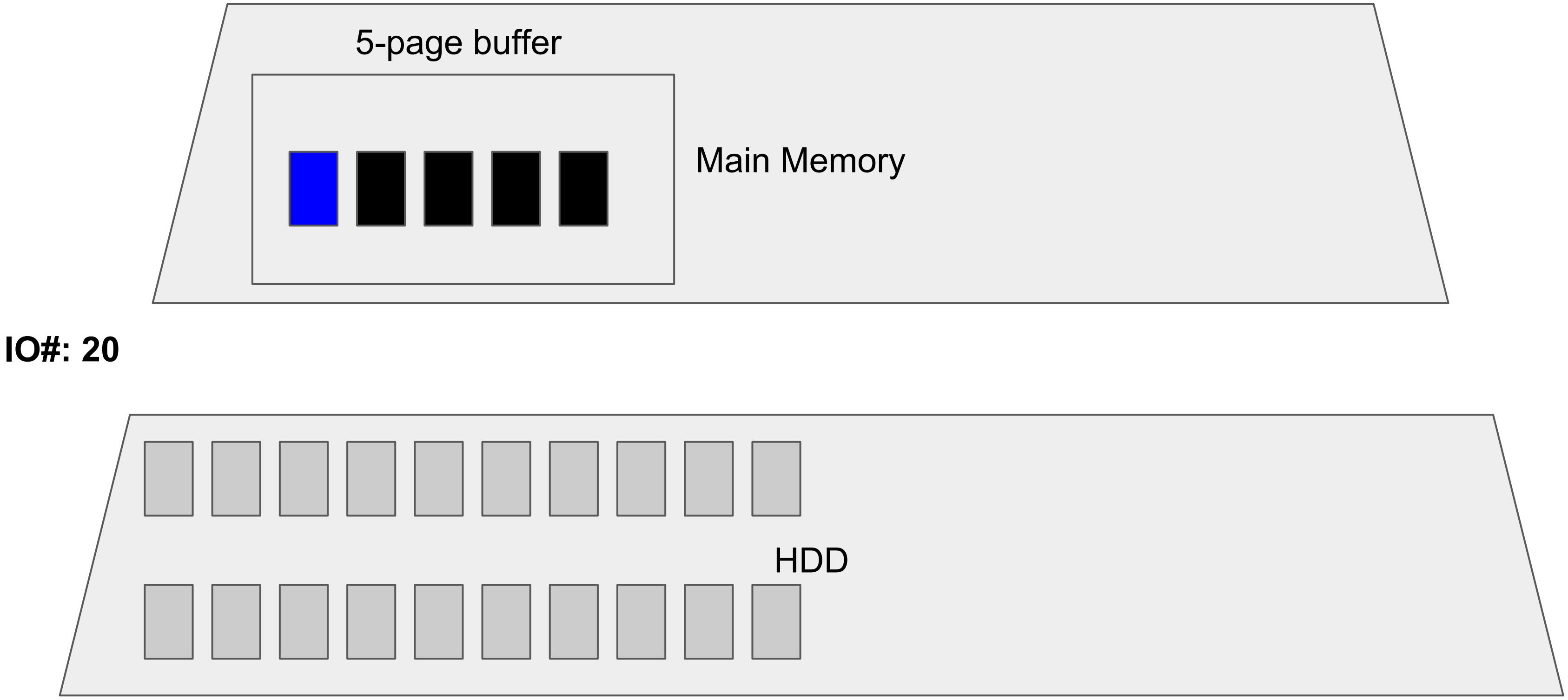


IO#: 15

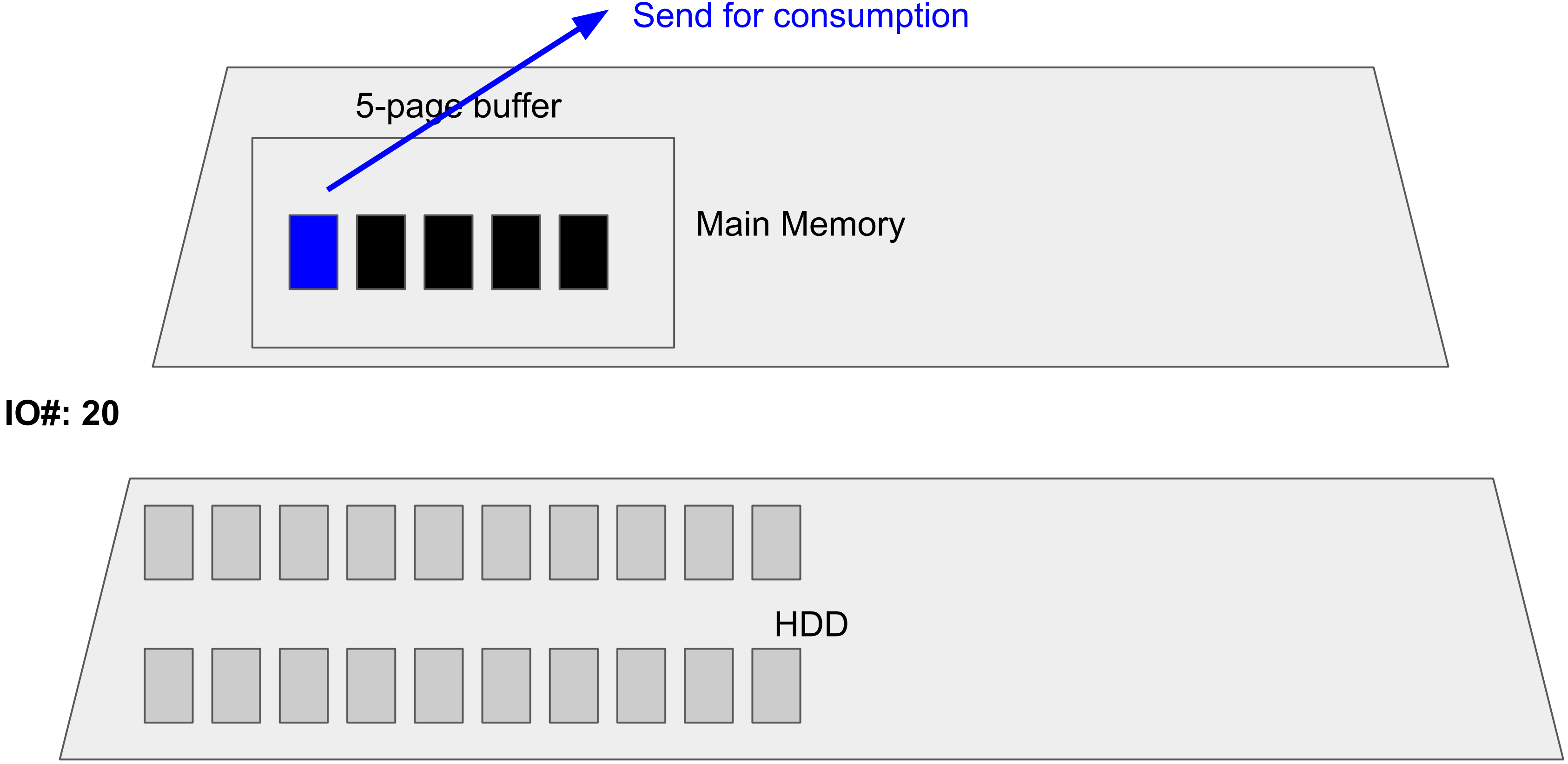
IO cost: Scanning a relation to select 10%



IO cost: Scanning a relation to select 10%

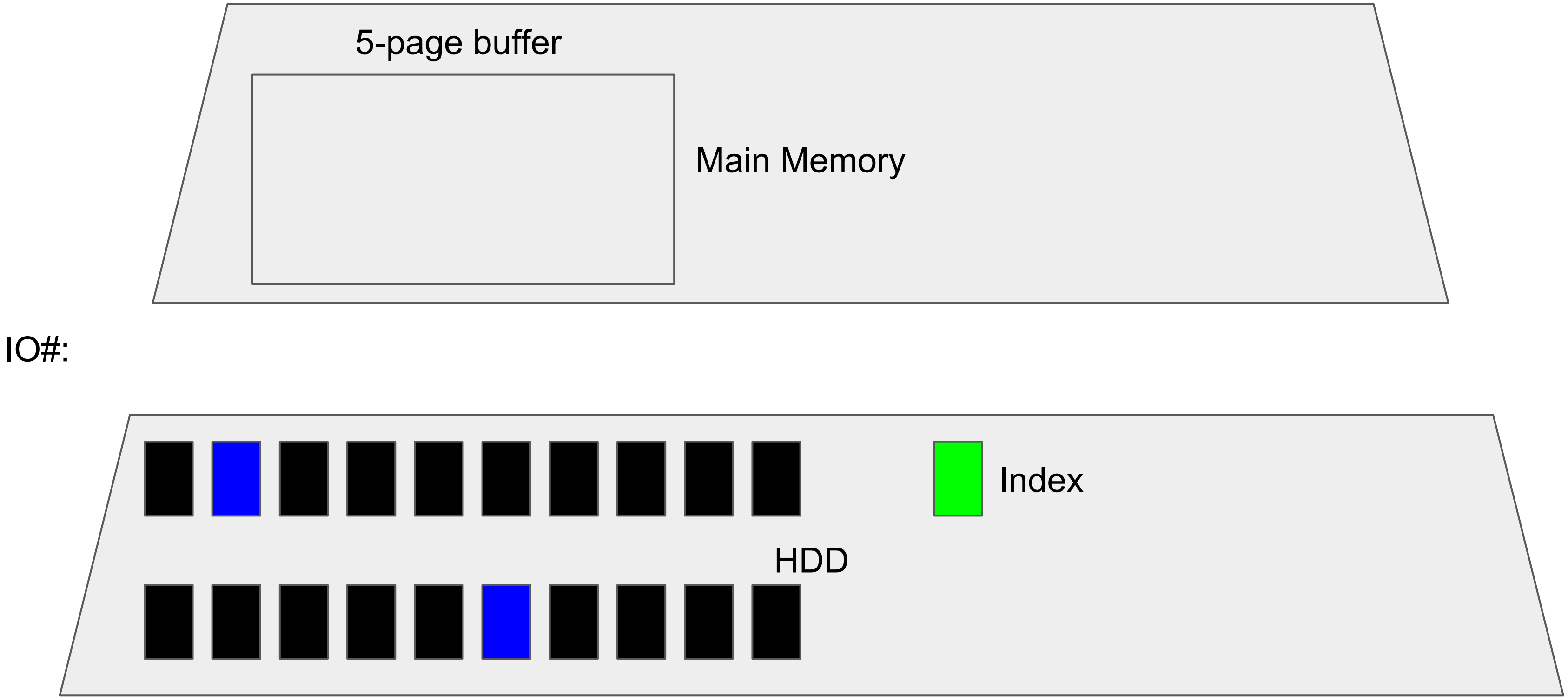


IO cost: Scanning a relation to select 10%

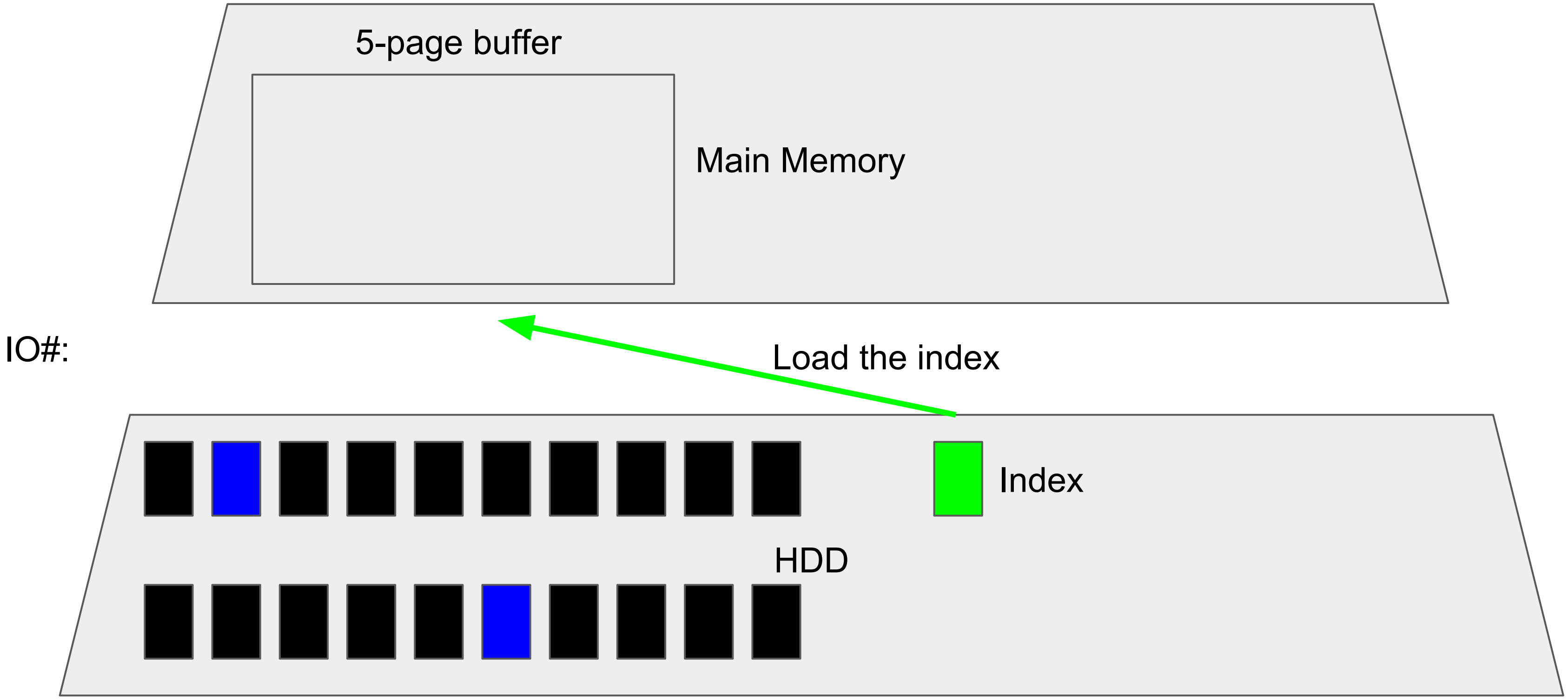


What if we had an oracle (index)?

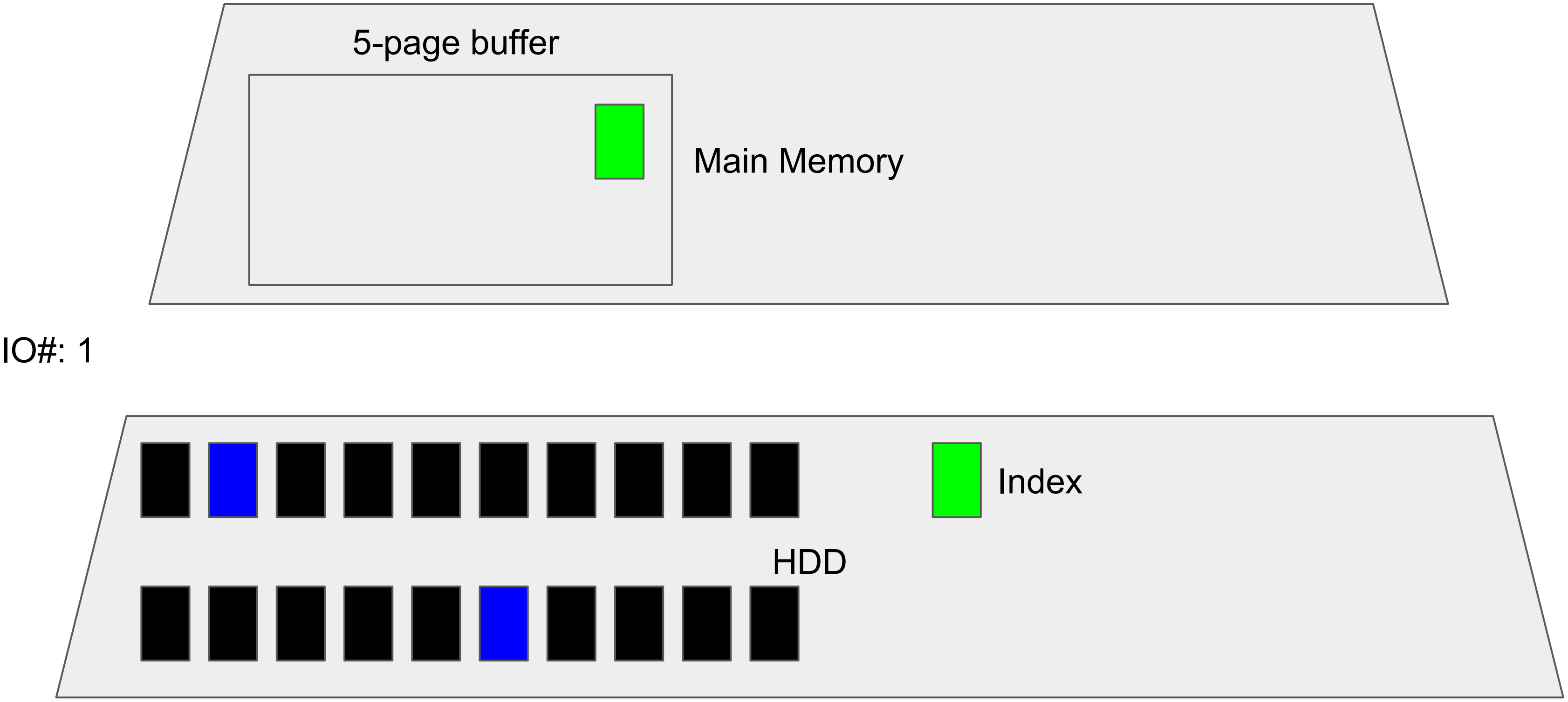
IO cost: Scanning a relation to select 10%



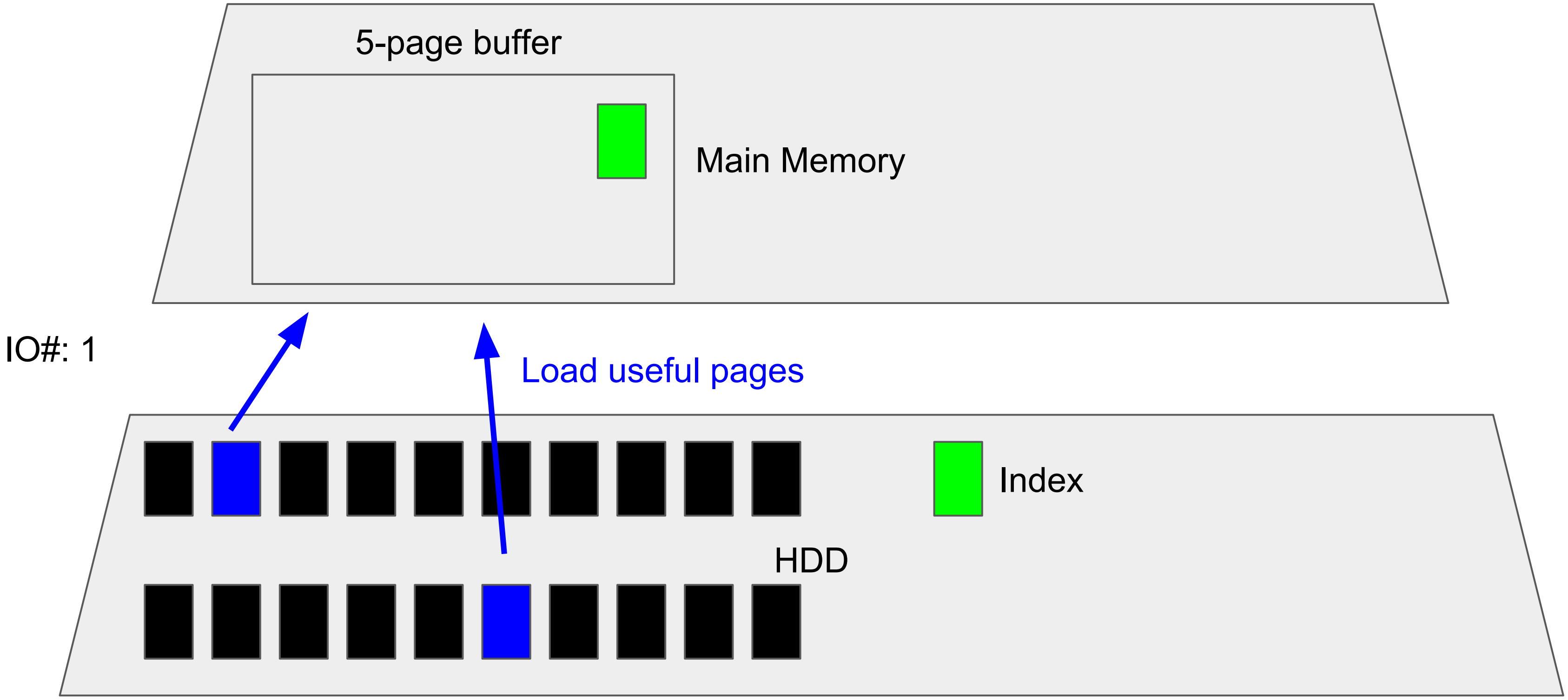
IO cost: Use an index to select 10%



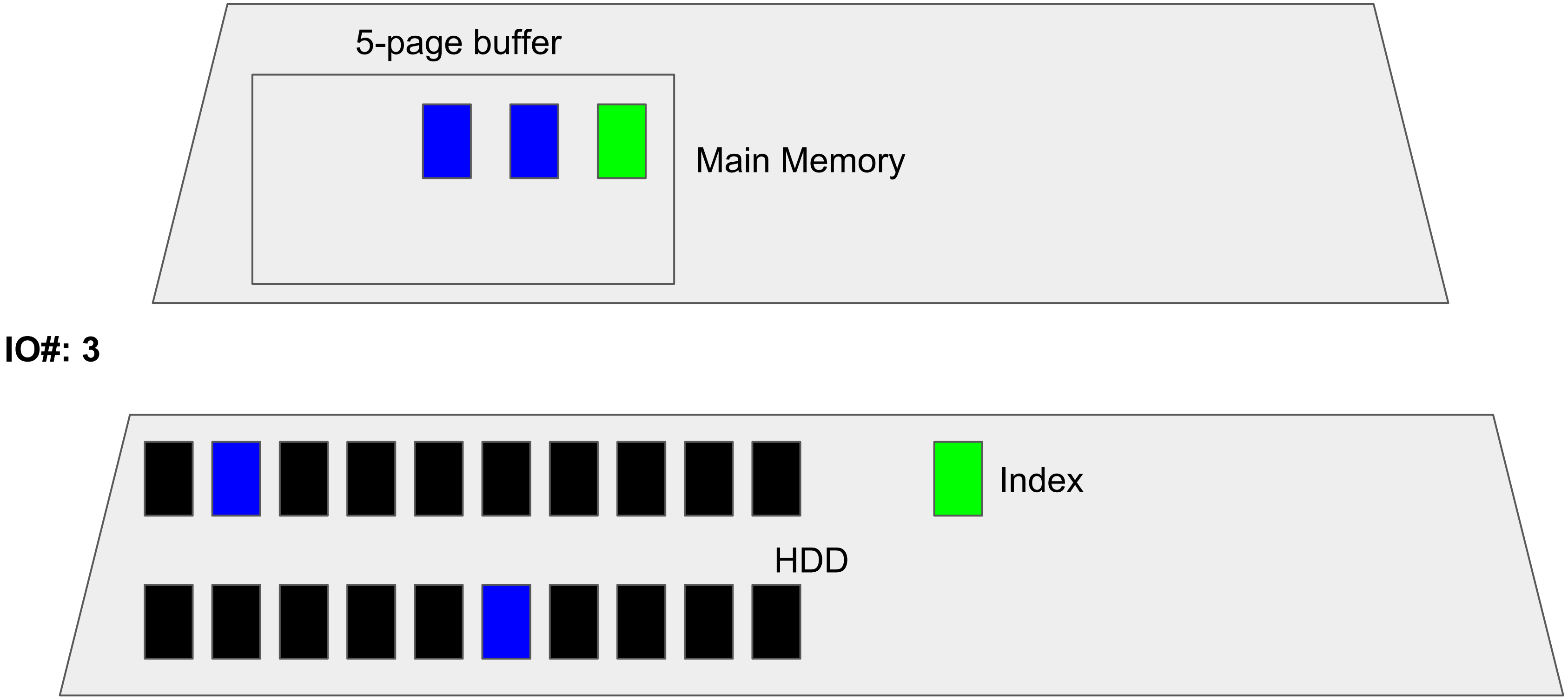
IO cost: Use an index to select 10%



IO cost: Use an index to select 10%

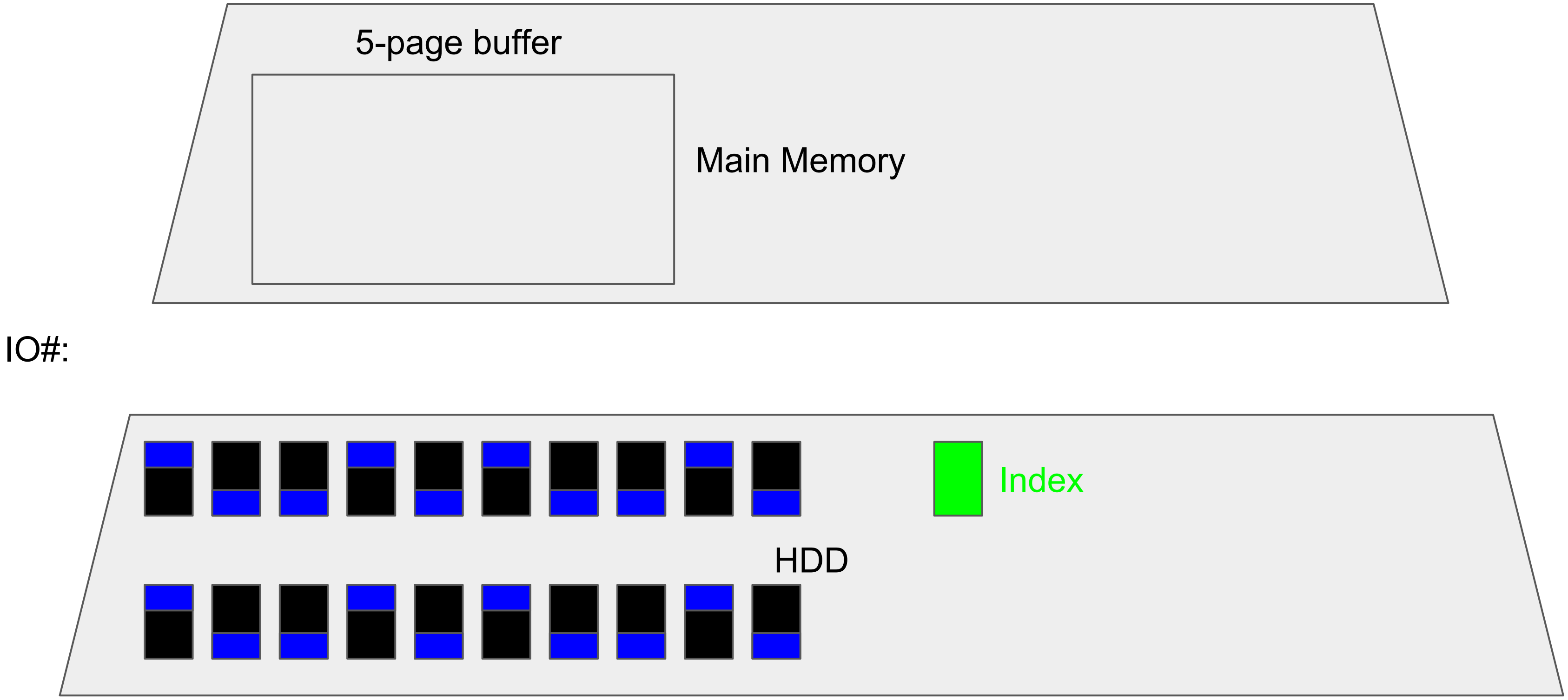


IO cost: Use an index to select 10%

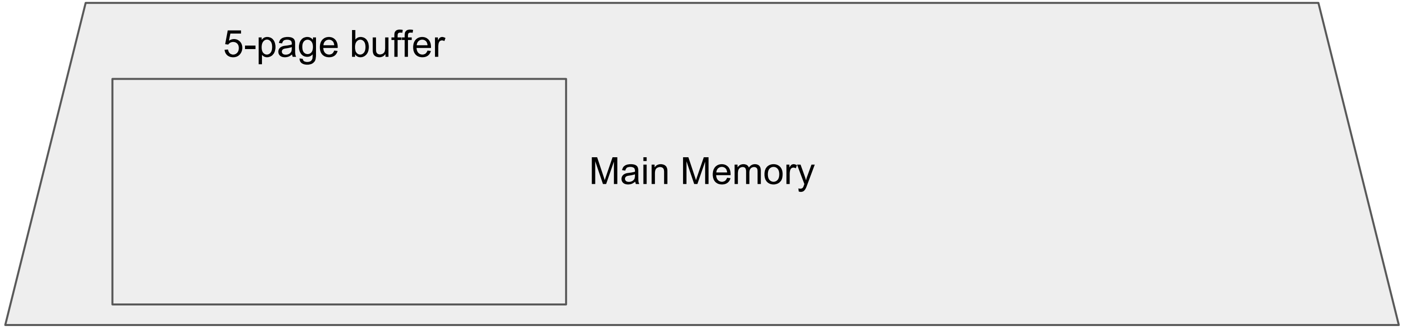


What if useful data is in all pages?

Scan or Index ?

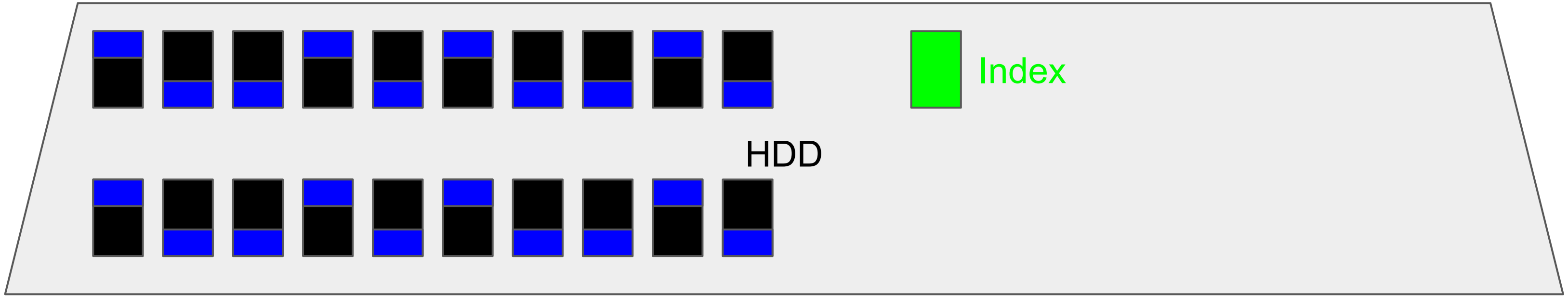


Scan or Index ?

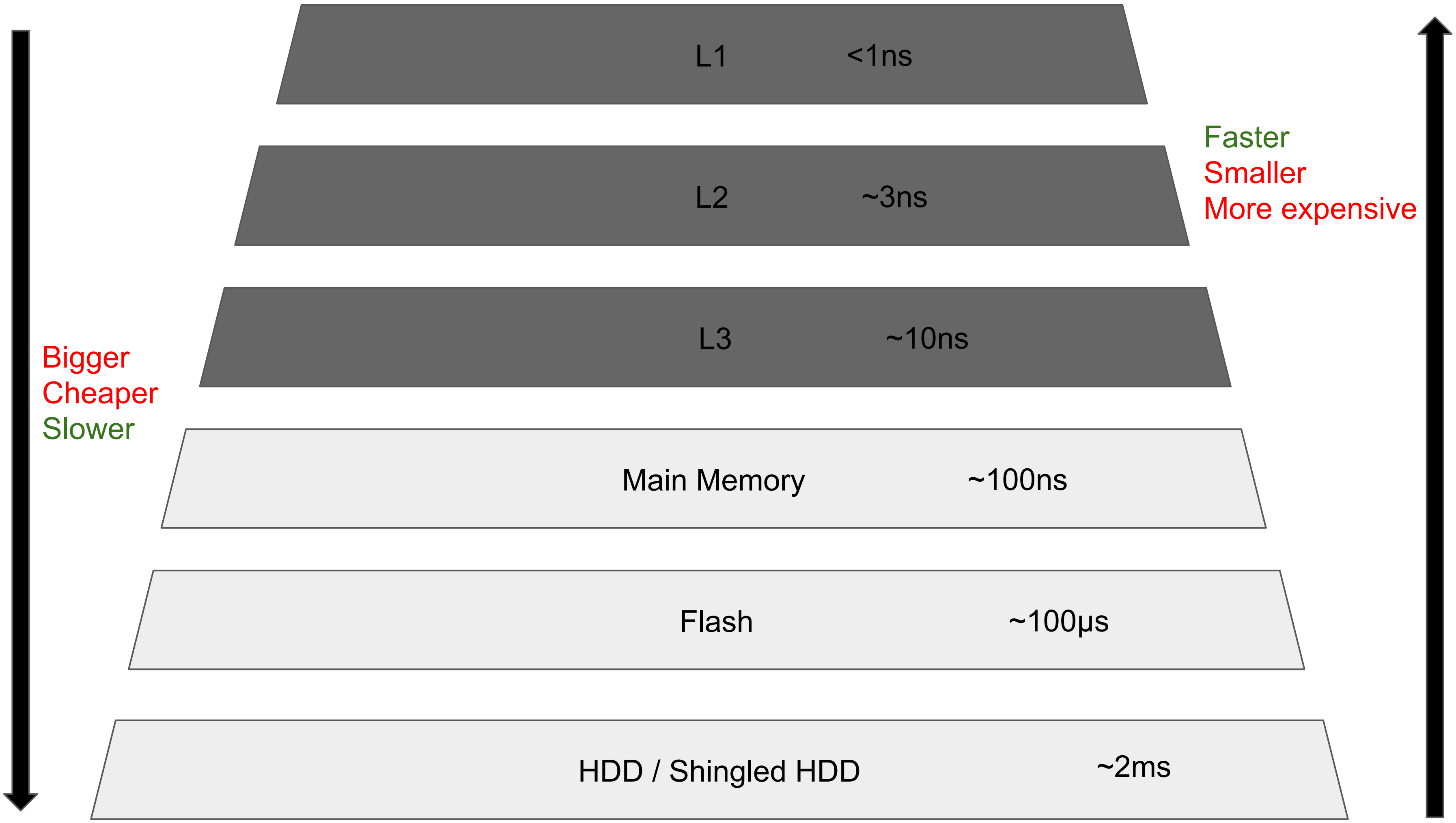


IO#: 20 with scan

IO#: 21 with index



Cache Hierarchy

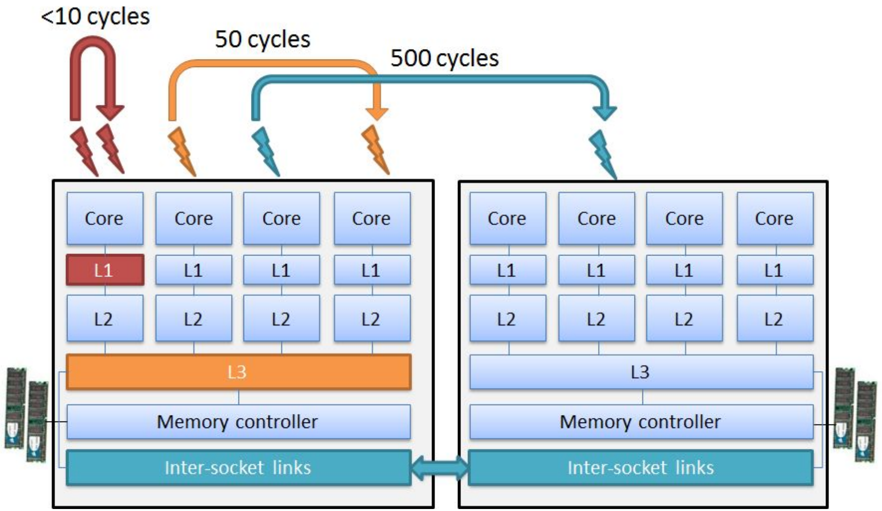


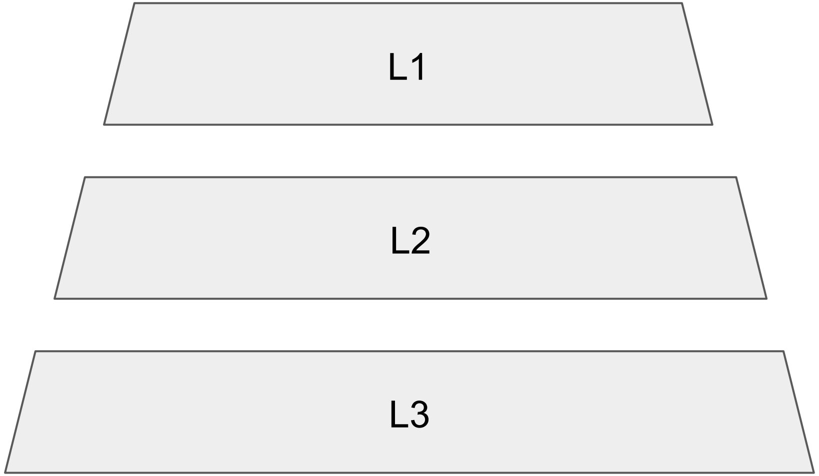
L1
<1ns
L2
~3ns
L3
~10ns
Main Memory
~100ns
Flash
~100µs
HDD / Shingled HDD
~2ms
Faster
Smaller
More expensive
Bigger
Cheaper
Slower

Cache Hierarchy

What is a core?

What is a socket?





Cache Hierarchy

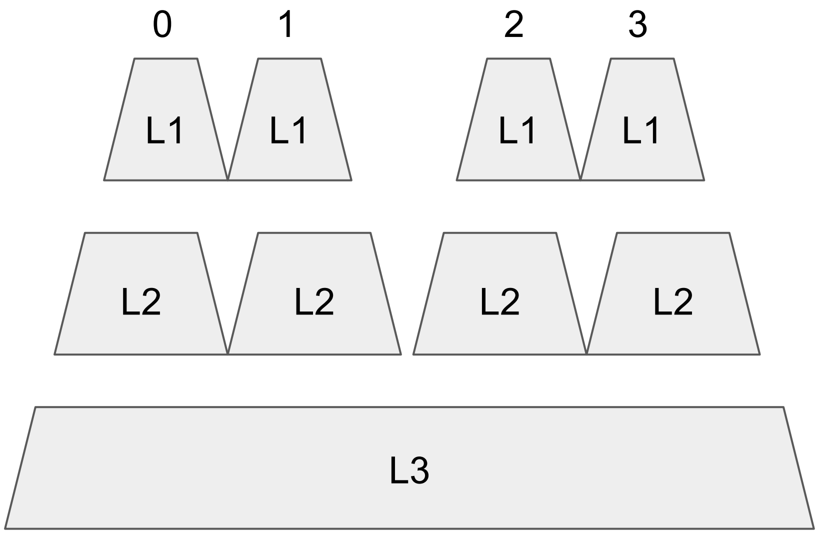
Shared Cache: L3 (or LLC: Last Level Cache)

L3 is physically distributed in multiple sockets

L2 is physically distributed in every core of every socket

Each *core* has its own **private** L1 & L2 cache

All levels need to be *coherent**



Non Uniform Memory Access (NUMA)

Core 0 reads faster when data are in its L1

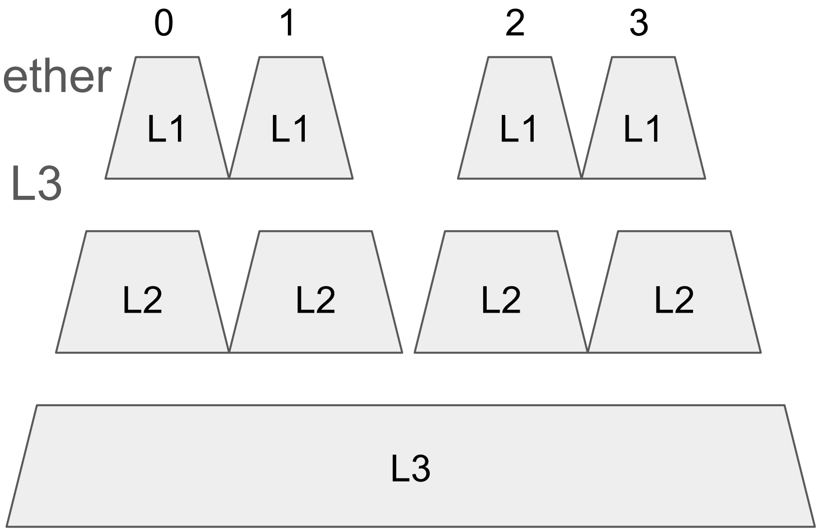
If it does not fit, it will go to L2, and then in L3

Can we control where data is placed?

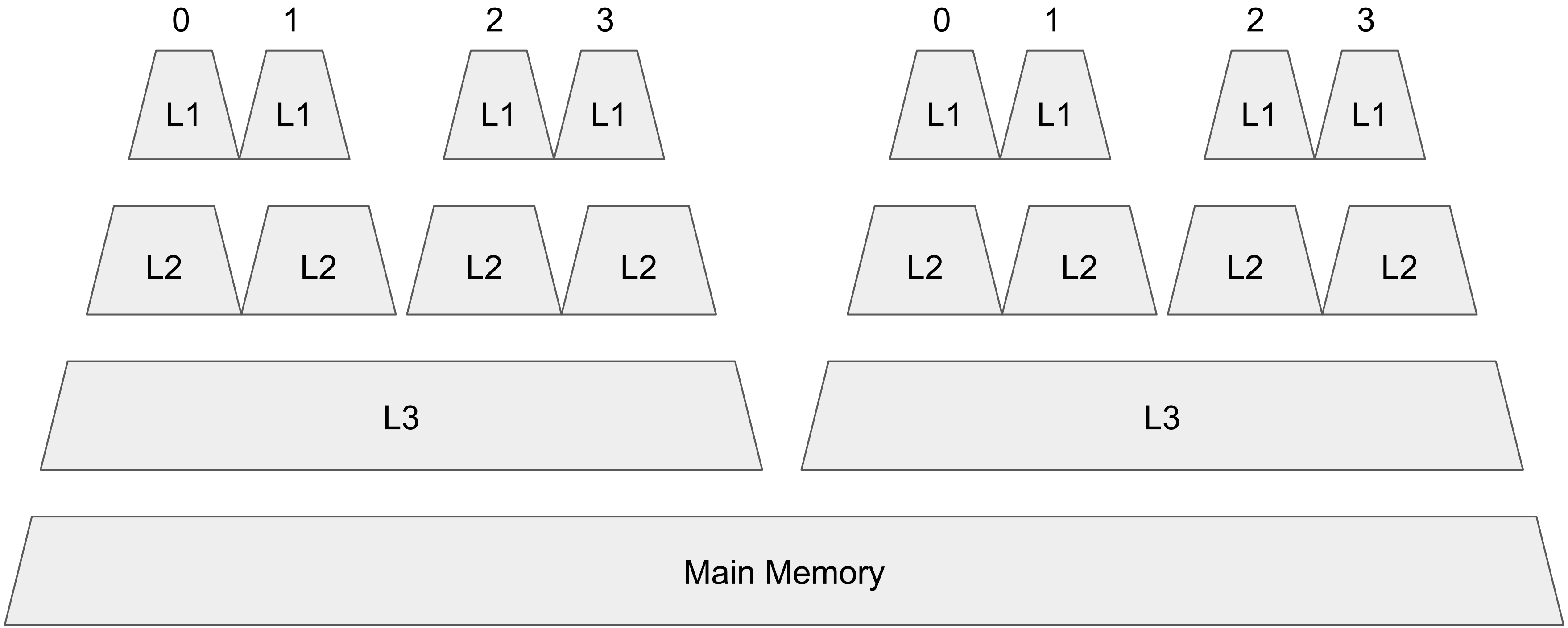
We would like to avoid going to L2 and L3 altogether

But, at least we want to avoid to remote L2 and L3

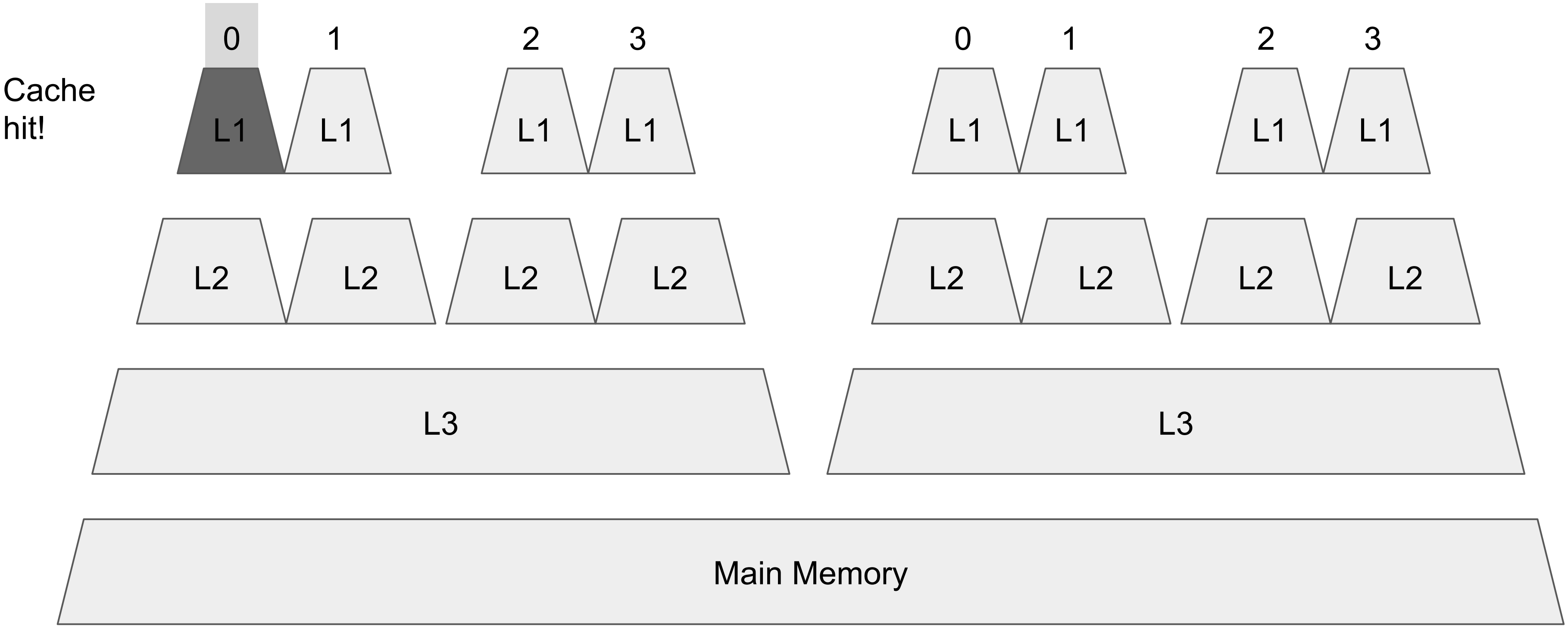
And remember: this is only one socket!
We have multiple of those!



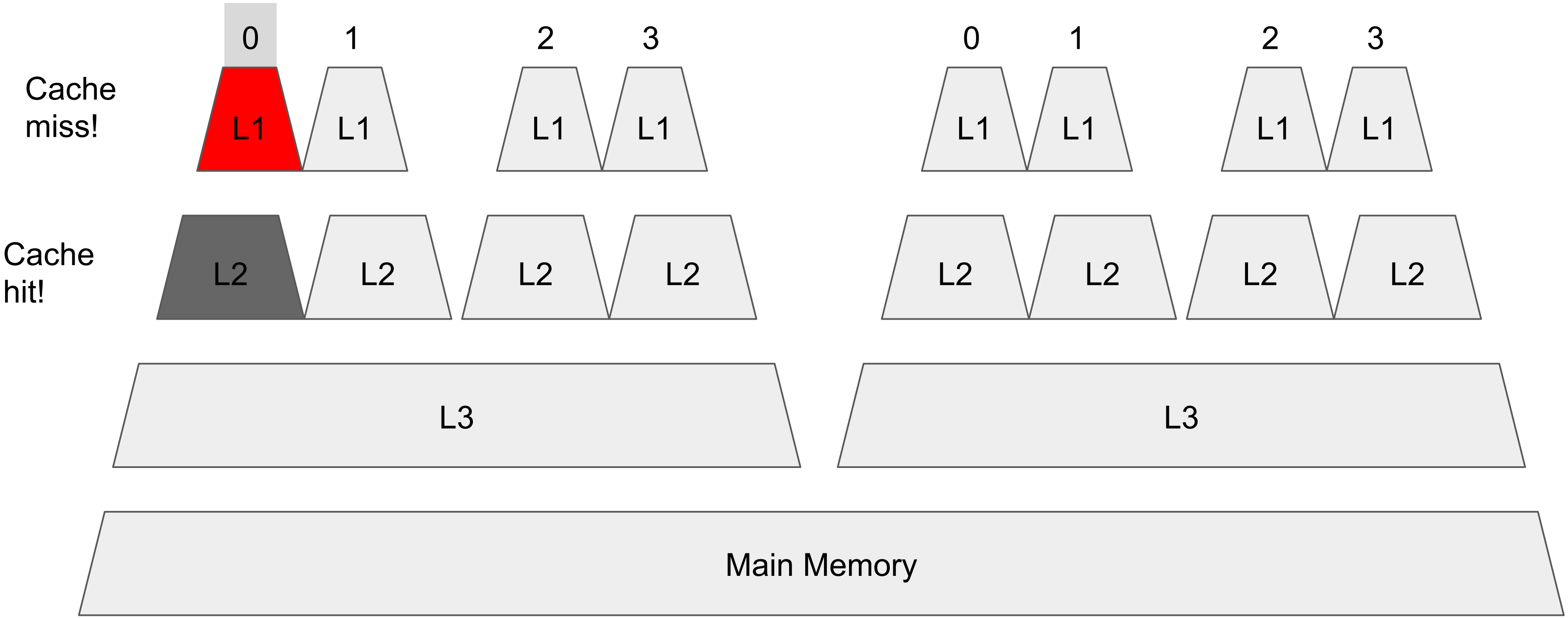
Non Uniform Memory Access (NUMA)



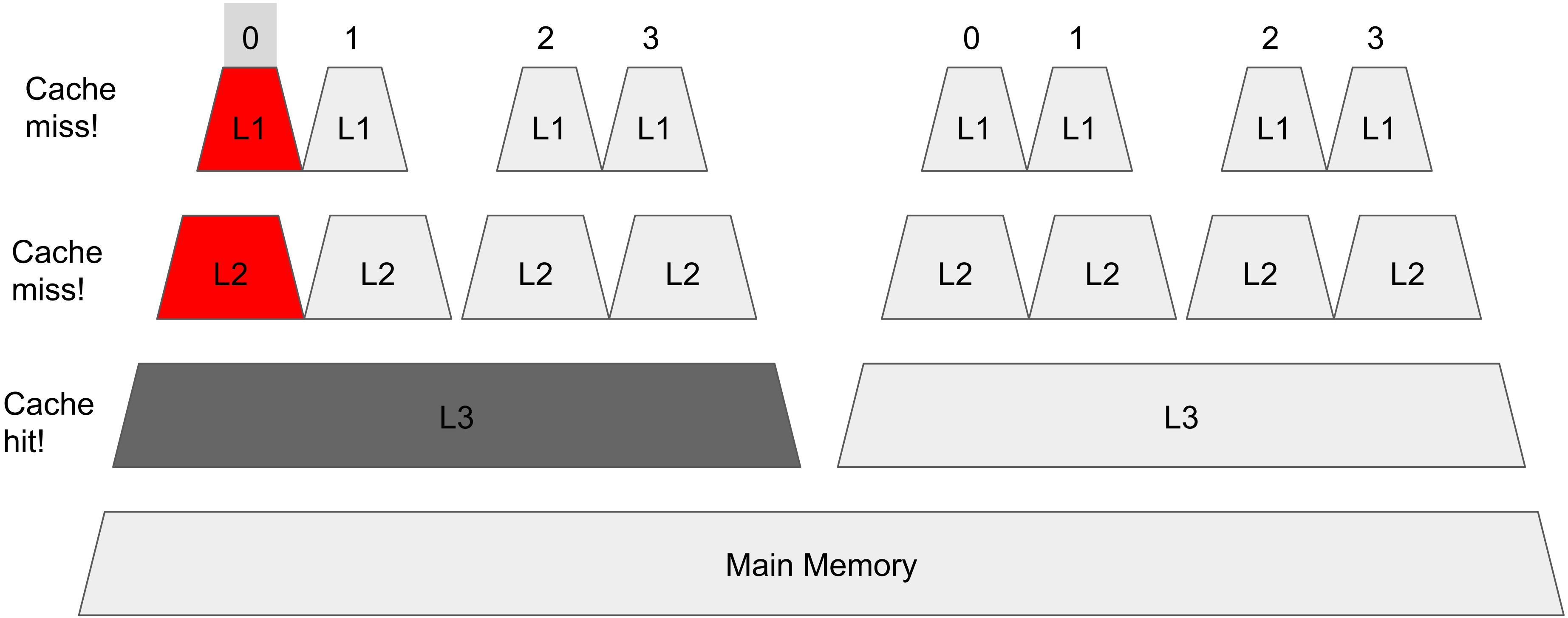
Non Uniform Memory Access (NUMA)



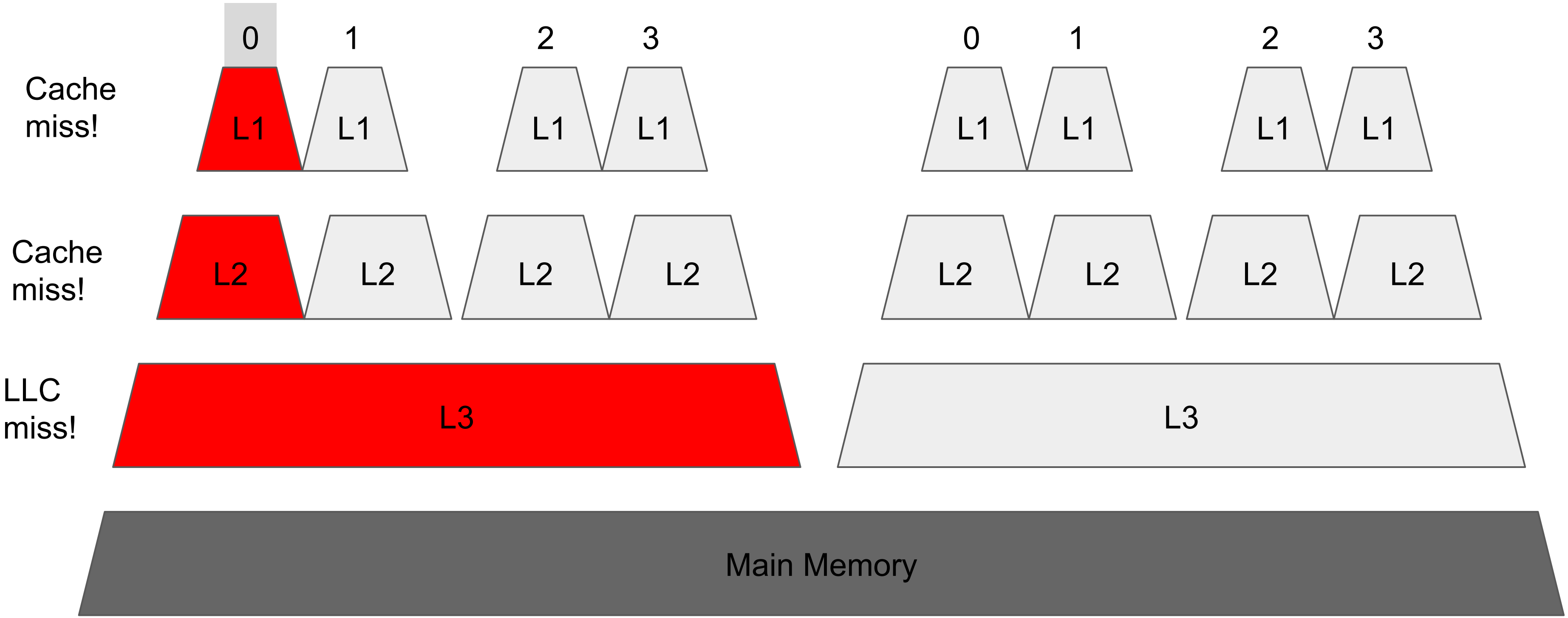
Non Uniform Memory Access (NUMA)



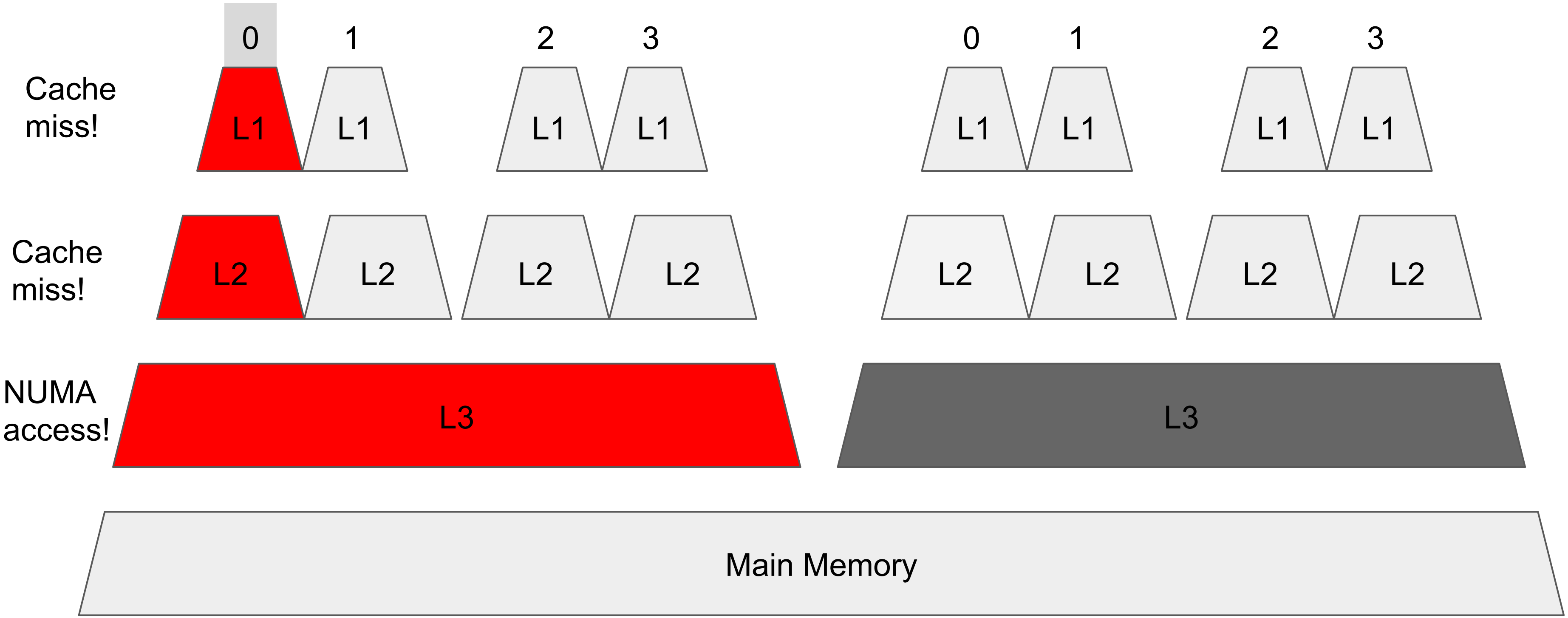
Non Uniform Memory Access (NUMA)



Non Uniform Memory Access (NUMA)



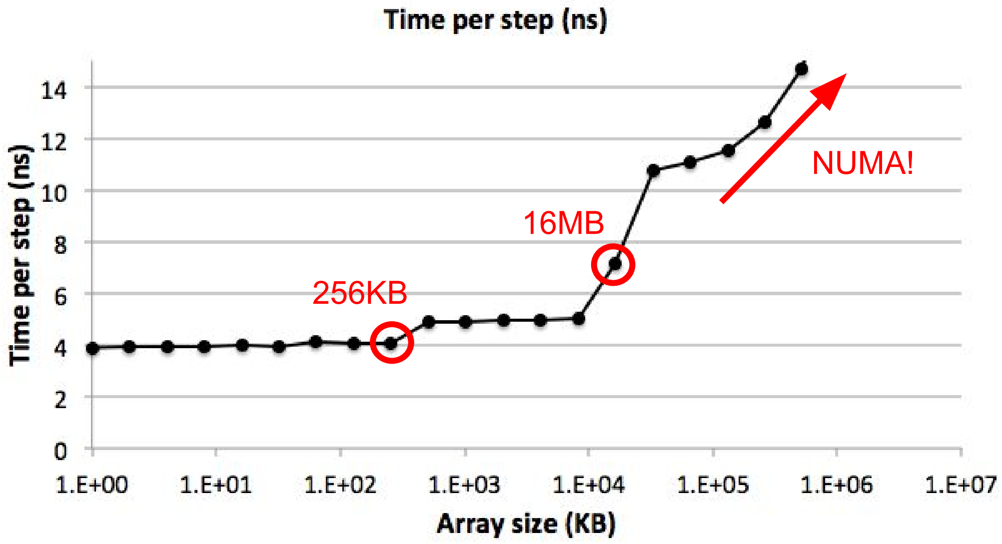
Non Uniform Memory Access (NUMA)



Why knowing the cache hierarchy matters

```
int arraySize;
for (arraySize = 1024/sizeof(int) ; arraySize <= 2*1024*1024*1024/sizeof(int) ; arraySize*=2)
// Create an array of size 1KB to 4GB and run a large arbitrary number of operations
{
   int steps = 64 * 1024 * 1024; // Arbitrary number of steps
   int* array = (int*) malloc(sizeof(int)*arraySize); // Allocate the array
   int lengthMod = arraySize - 1;

   // Time this loop for every arraySize
   int i;
   for (i = 0; i < steps; i++)
   {
      array[(i * 16) & lengthMod]++;
      // (x & lengthMod) is equal to (x % arraySize)
   }
}
```



This machine has:
256KB L2 per core
16MB L3 per socket

Storage Hierarchy

Why not just stay in memory?

Storage Hierarchy

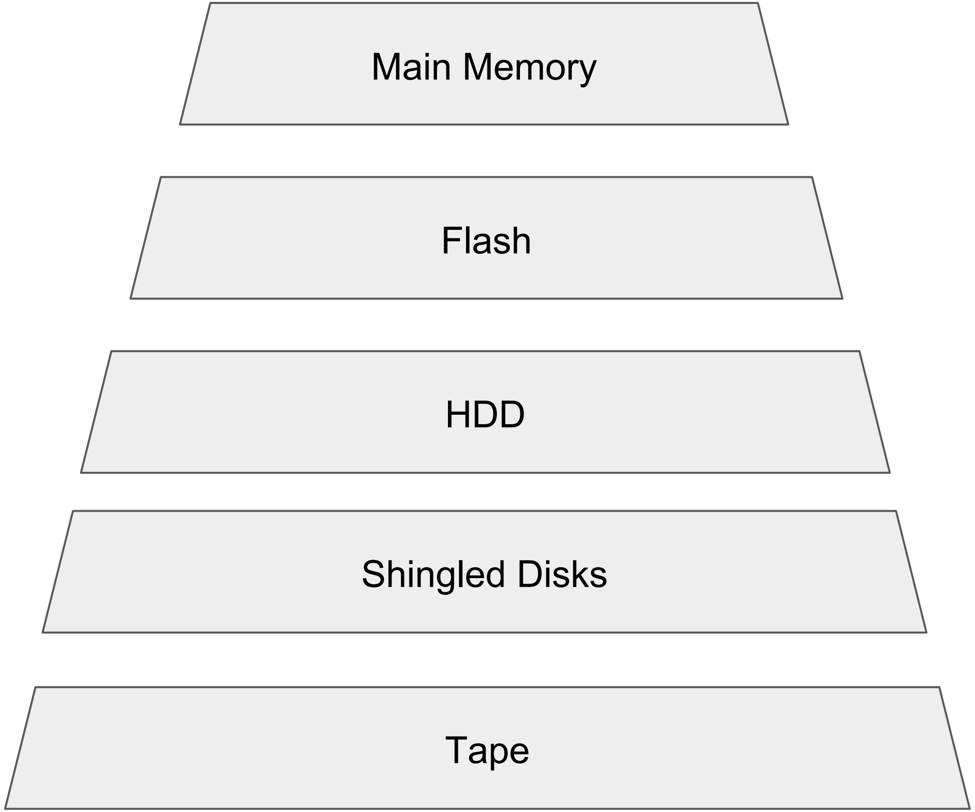
Why not stay in memory?

- Cost
- Volatility

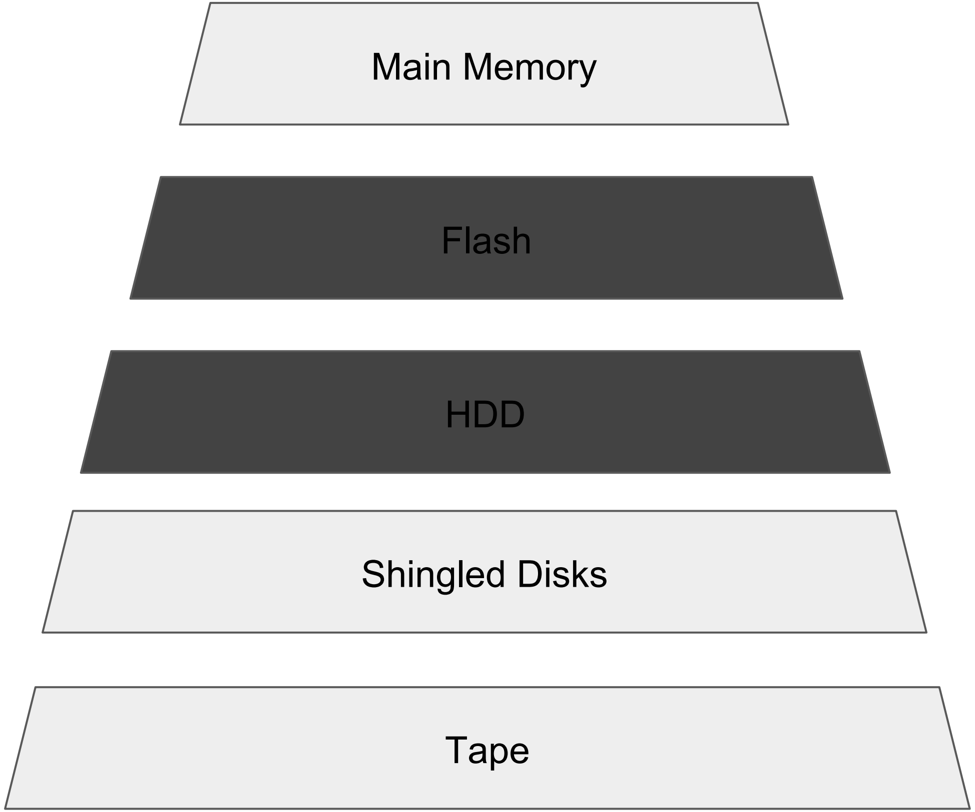
What was missing from memory hierarchy?

- Durability
- Capacity

Storage Hierarchy



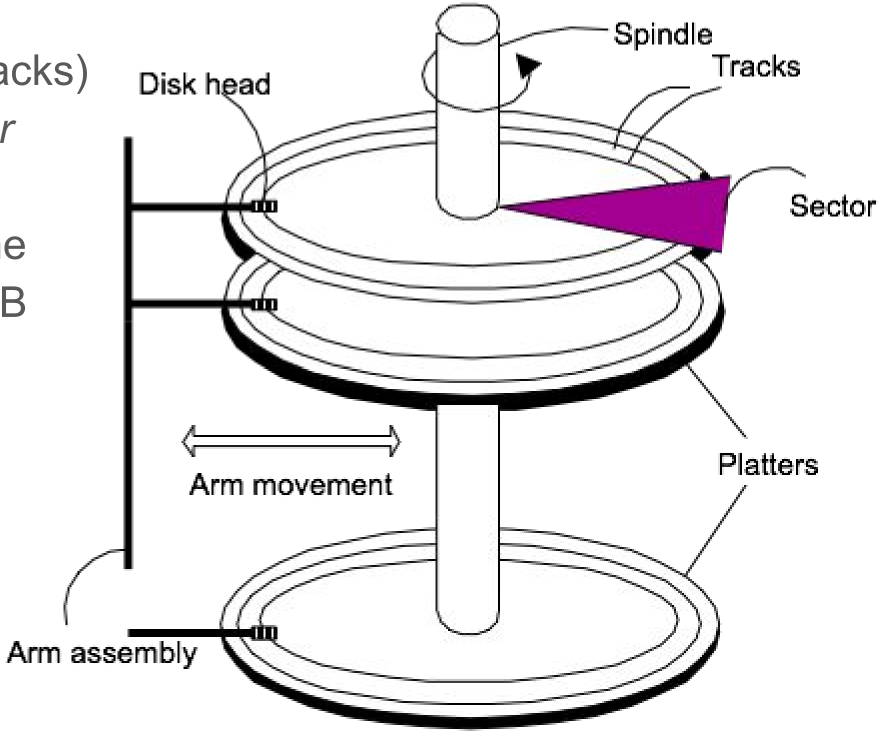
Storage Hierarchy



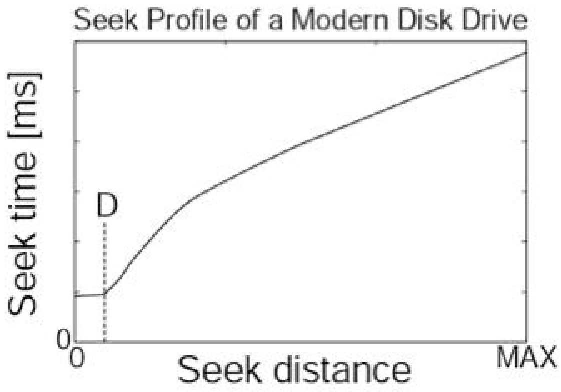
Disks

Secondary durable storage that support both *random* and *sequential* access

- Data organized on pages/blocks (accross tracks)
- Multiple *tracks* create an (imaginary) *cylinder*
- Disk access time:
  seek latency + rotational delay + transfer time
  (0.5-2ms) + (0.5-3ms) + <0.1ms/4KB
- Sequential >> random access (~10x)
- **Goal:** *avoid random access*



Seek time + Rotational delay + Transfer time

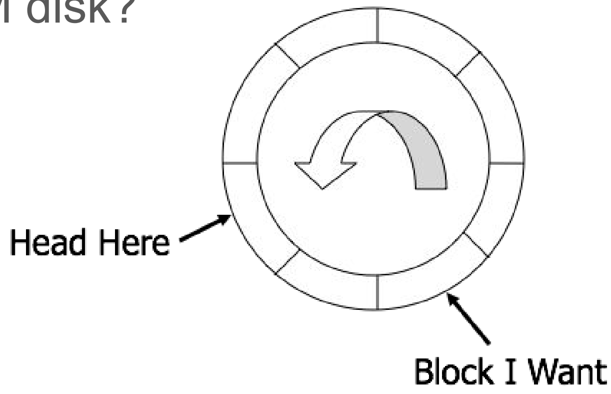


Seek time: the **head** goes to the right **track**

Short seeks are dominated by “settle” time
(D is on the order of hundreds or more)

Rotational delay: The **platter** rotates to the right **sector.**
What is the min/max/avg rotational delay for 10000RPM disk?

Transfer time: <0.1ms / page → more than 100MB/s



Sequential vs. Random Access

Bandwidth for Sequential Access (assuming 0.1ms/4KB):

0.1ms for 4KB → **40MB/s**

Bandwidth for Random Access (4KB):

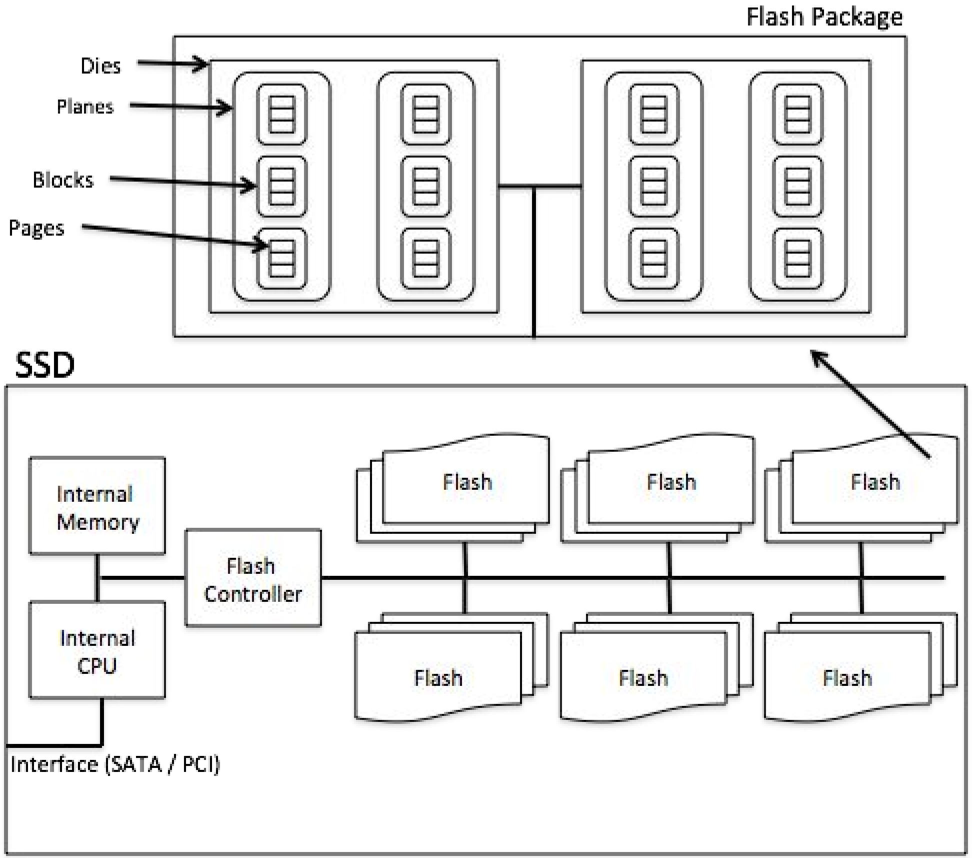
1ms (seek time) + 3ms (rotational delay) + 0.1ms = 4.1ms / 4KB → **1MB/s**

Flash

Secondary durable storage that support both *random* and *sequential* access

- Data organized on pages (similar to disks) which are further grouped to erase blocks
- Main advantage over disks: random read is now much more efficient
- BUT: Slow random writes!
- **Goal:** *avoid random writes*

The internals of flash



Interconnected flash chips

No mechanical limitations

Maintain the block API – compatible with disks layout

Internal parallelism in read/write

Complex software driver

Flash access time

… depends on:

- device organization (internal parallelism)
- software efficiency (driver)
- bandwidth of flash packages
- Flash Translation Layer (FTL), a complex device driver (firmware) which
  - tunes performance and device lifetime

Flash vs HDD

HDD

✓ Large - cheap capacity

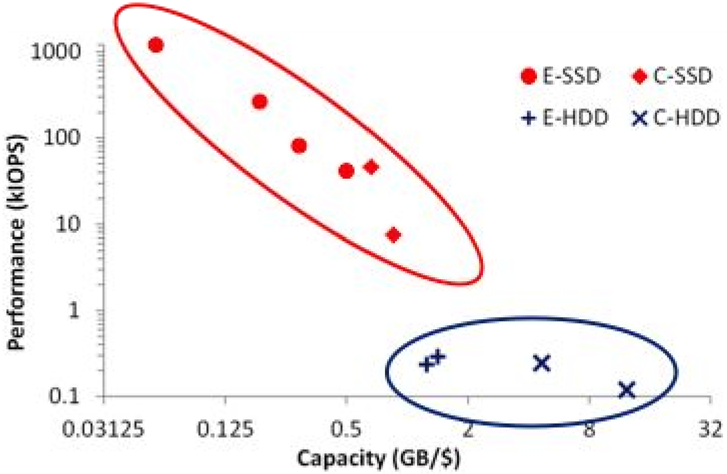
✗ Inefficient random reads

Flash

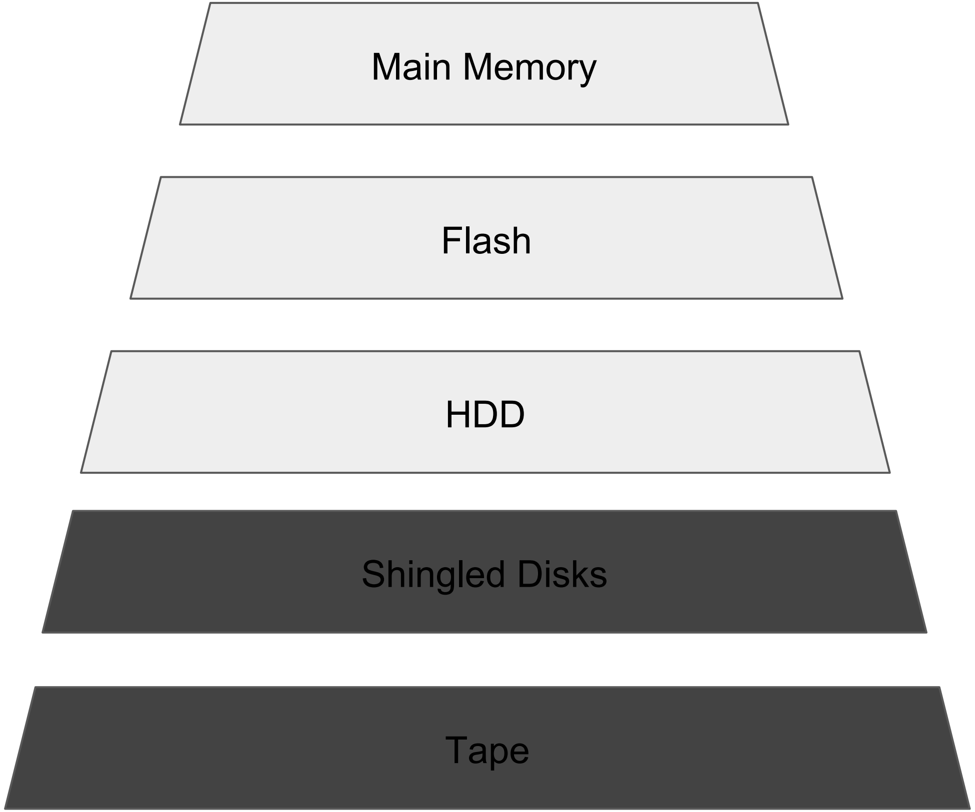
✗ Small - expensive capacity

✓ Very efficient random reads

✗ Read/Write Asymmetry



Storage Hierarchy



Tapes

Data size grows exponentially!

Cheaper capacity:

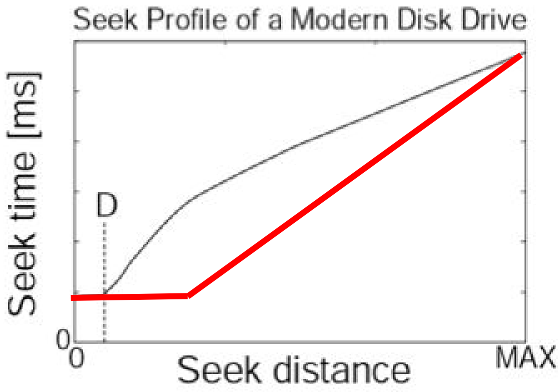
- Increase density (bits/in^2)
- Simpler devices

Tapes:

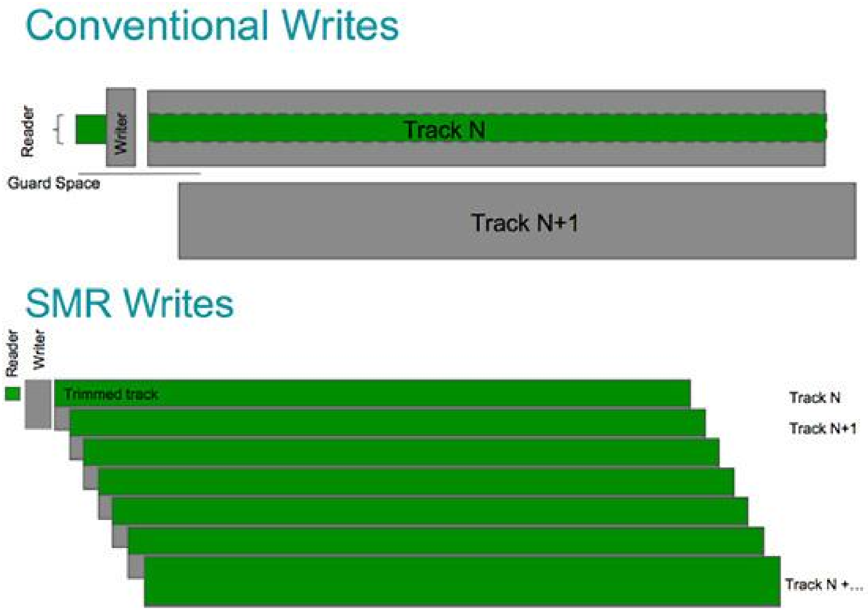
- Magnetic medium that allows only **sequential access** (yes like an old school tape)

Increasing disk density

Very difficult to differentiate between tracks
“settle” time becomes



Writing a track affects neighboring tracks
Create different readers/writers
Interleave writes tracks



Summary

Memory/Storage Hierarchy

Access granularity (pages, blocks, cache-lines)

Memory Wall → deeper and deeper hierarchy