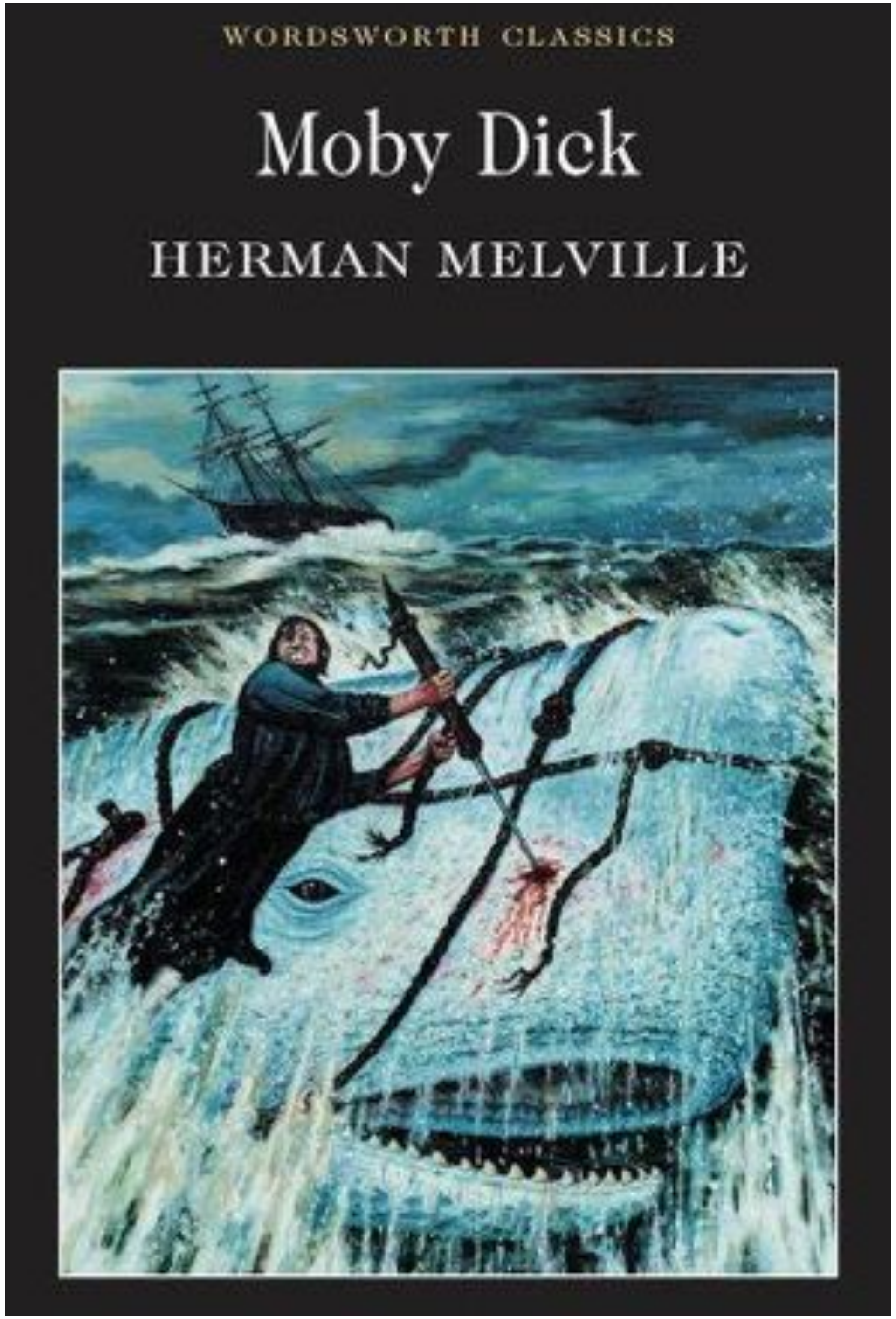
Storage Locality and counting words

Counting the words in a long text

- 218718 words long
- 17150 different words
- How many times does each word occur?



Task: count the number of occurrences of each word in very long text.

- **Input:** Call me Ishmael. Some years ago--never mind how long precisely—having little or no money in my purse, and nothing particular to interest me onshore, I thought I would sail …
- Moby dick: about 1.3MByte
- Desired output:
  - Call: 354
  - Me: 53423
  - Ismael: 1322
  - ….

Simple solution

- Iterate over words. Update counter for current word.

```
In [5]:
%%time
def simple_count(list):
    D={}
    for w in list:
        if w in D:
            D[w]+=1
        else:
            D[w]=1
    return D
D=simple_count(all)
```

```
CPU times: user 49.5 ms, sys: 5.95 ms, total: 55.5 ms
Wall time: 53.1 ms
```

Lets use a sorted list

```
=== unsorted list:
the,vernacular,but,as,for,you,ye,carrion,rogues,turning,to,
the,three,men,in,the,rigging,for,you,i,mean,to,mince,ye,up,
for

=== sorted list:
lines,lingered,lingered,lingered,lingered,lingered,lingerin
g,lingering,lingering,lingering,lingering,lingering,lingeri
ng,lingering,lingers,lingo,lingo,lining,link,link,linked,li
nked,linked,linked,links,links
```

5

8

Sort-based solution

```
def sort_count(list):
    t0=time()
    S=sorted(list)
    t1=time()
    D={}
    current=''
    count=0
    for w in S:
        if current==w:
            count+=1
        else:
            if current!='':
                D[current]=count
            count=1
            current=w
    t2=time()
    return D,t1-t0,t2-t1
D,sort_time,count_time=sort_count(all)
print 'sort time= %5.1f ms, count time=%5.1f ms'%(1000*
```

Sort words

Iterate over sorted list

Count occurrences of same word

Switch on word boundry

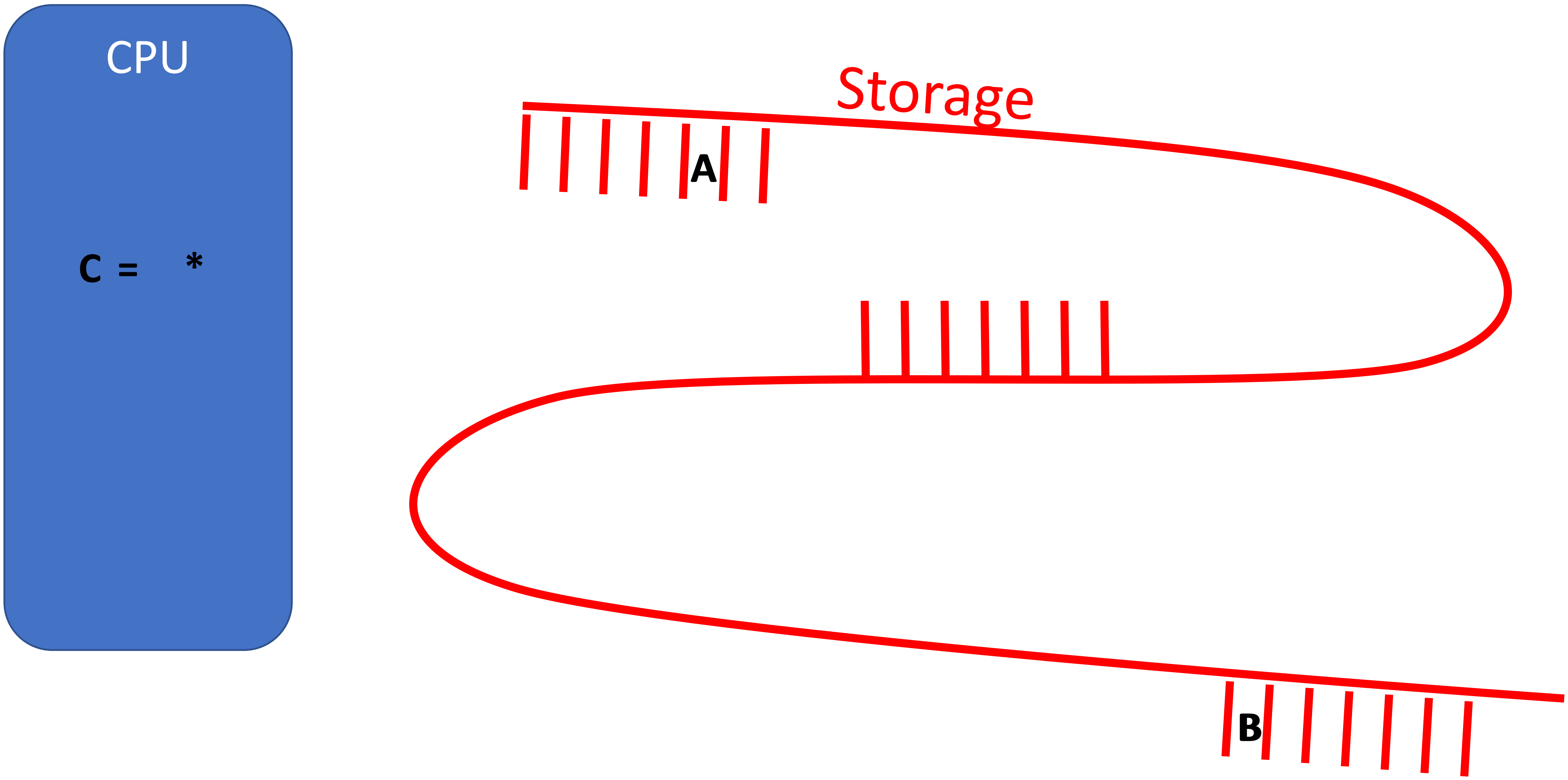
```
sort time= 103.0 ms, count time= 37.6 ms
CPU times: user 138 ms, sys: 5.33 ms, total: 143 ms
Wall time: 143 ms
```

Summary

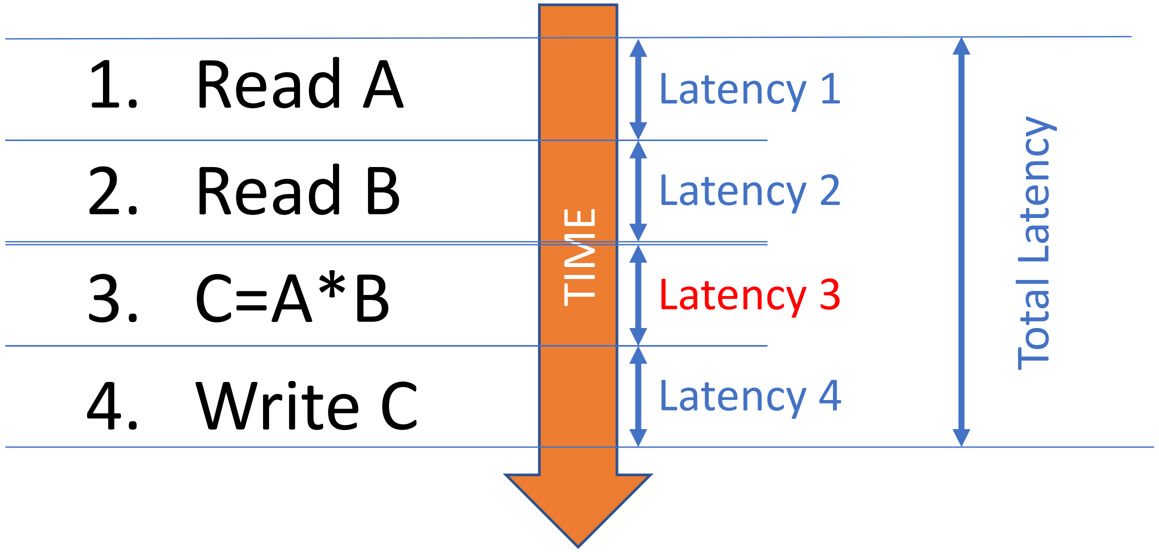
- Sorting improves memory locality for word counting
- Improved memory locality reduces run-time
- Why? Because computer memory is organized in a hierarchy.

Storage Latency

Small and Fast vs. Large and Slow


CPU
C = *
Storage
A
B

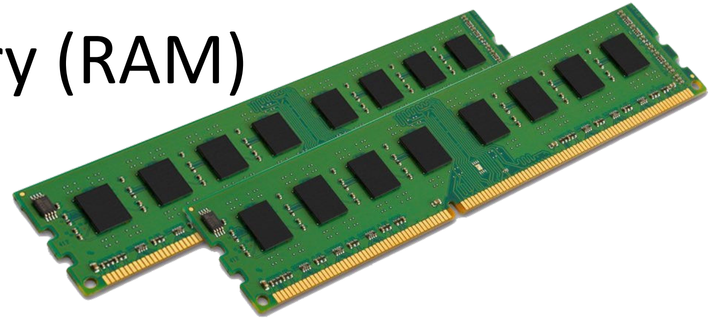
Latencies



With big data, most of the latency is memory latency (1,2,4), not computation (3)

Storage Types

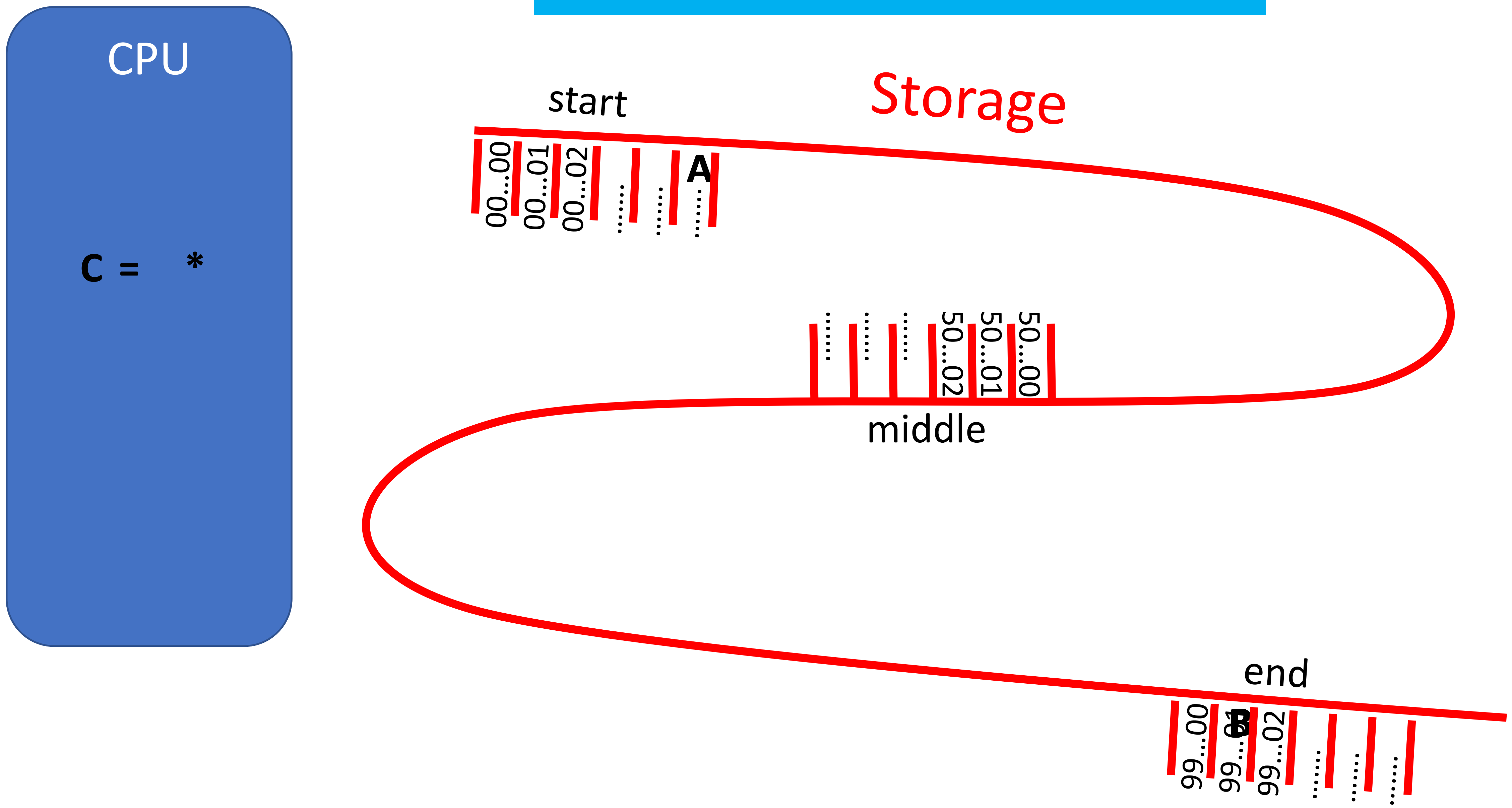
- Main Memory (RAM)



- Spinning disk

- Remote computer

NON-LOCAL STORAGE ACCESS

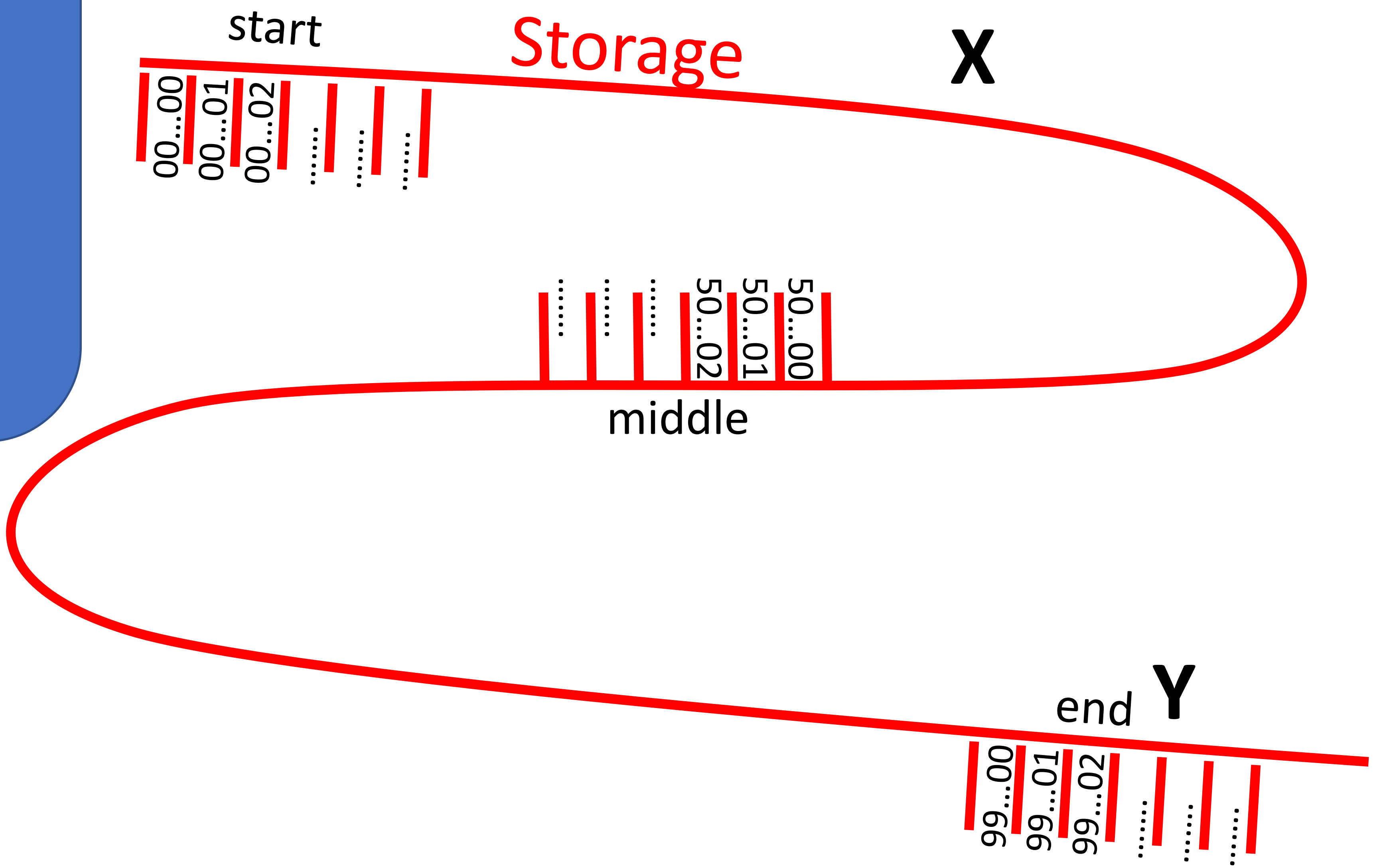


LOCAL STORAGE ACCESS

CPU

```
A=0
For i in range(100000):
    A+= X[i]

For i in range(100000):
    A-= Y[i]
```



Summary

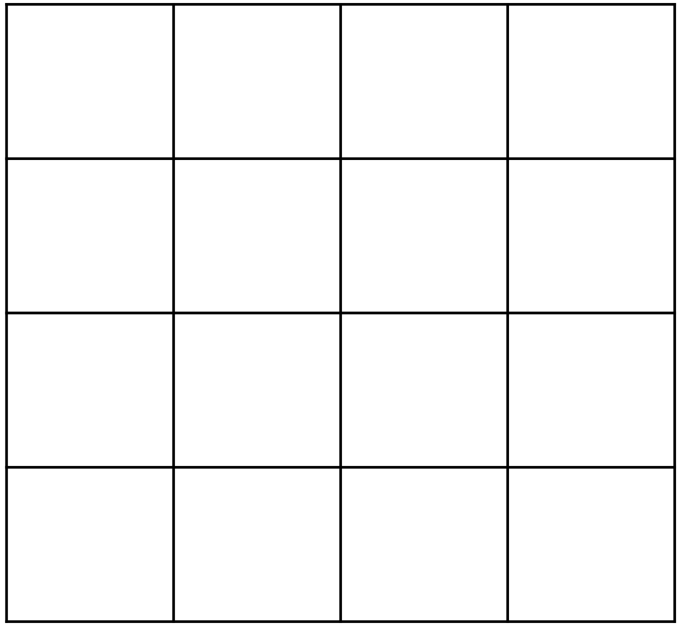
- The major source of latency in data analysis is reading and writing to storage
- Different types of storage offer different latency, capacity and price.
- Big data analytics revolves around methods for organizing storage and computation in ways that maximize speed while minimizing cost.
- Next, Caches and the memory Hierarchy.

Caches and the
Memory Hierarchy

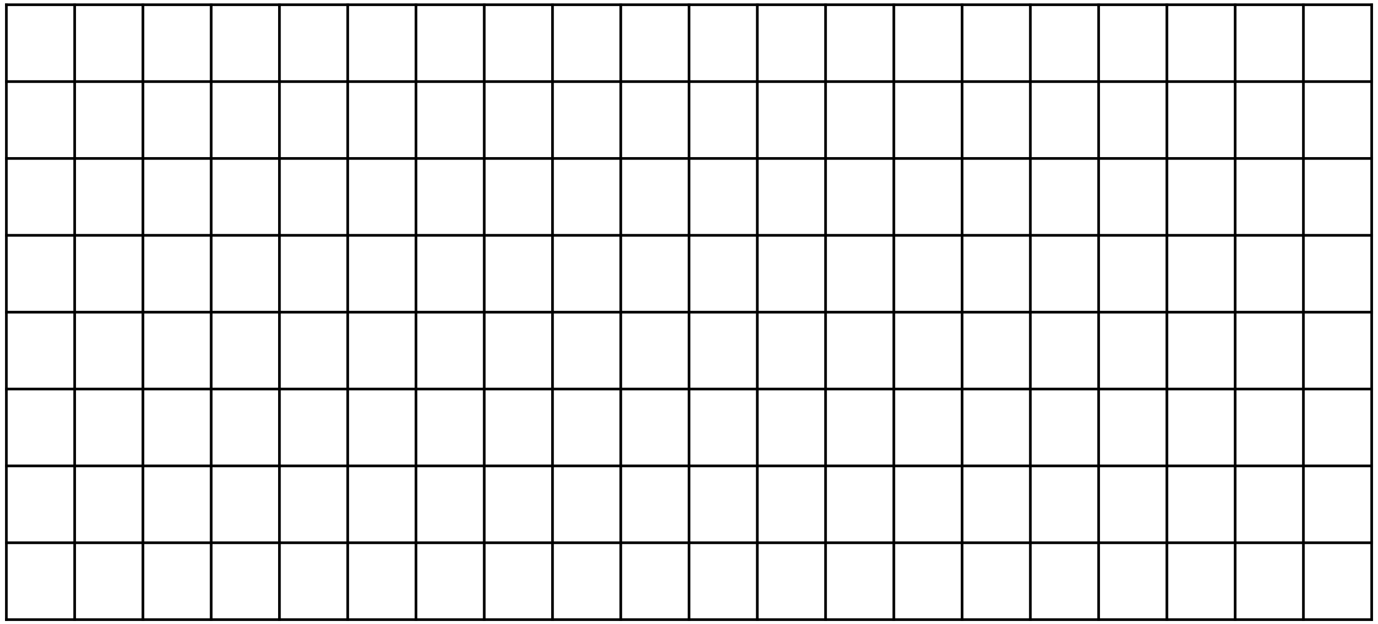
Latency, size and price of computer memory

Given a budget, we need to trade off

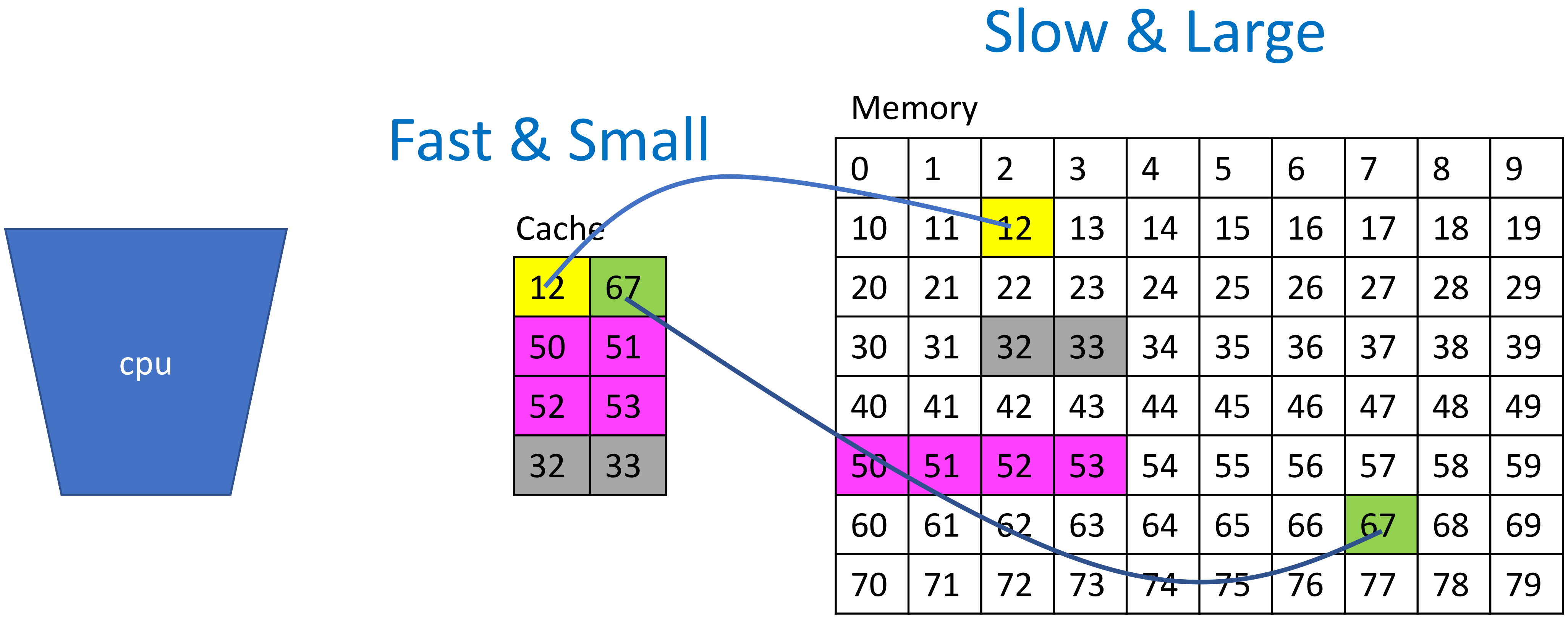
$10: Fast & Small



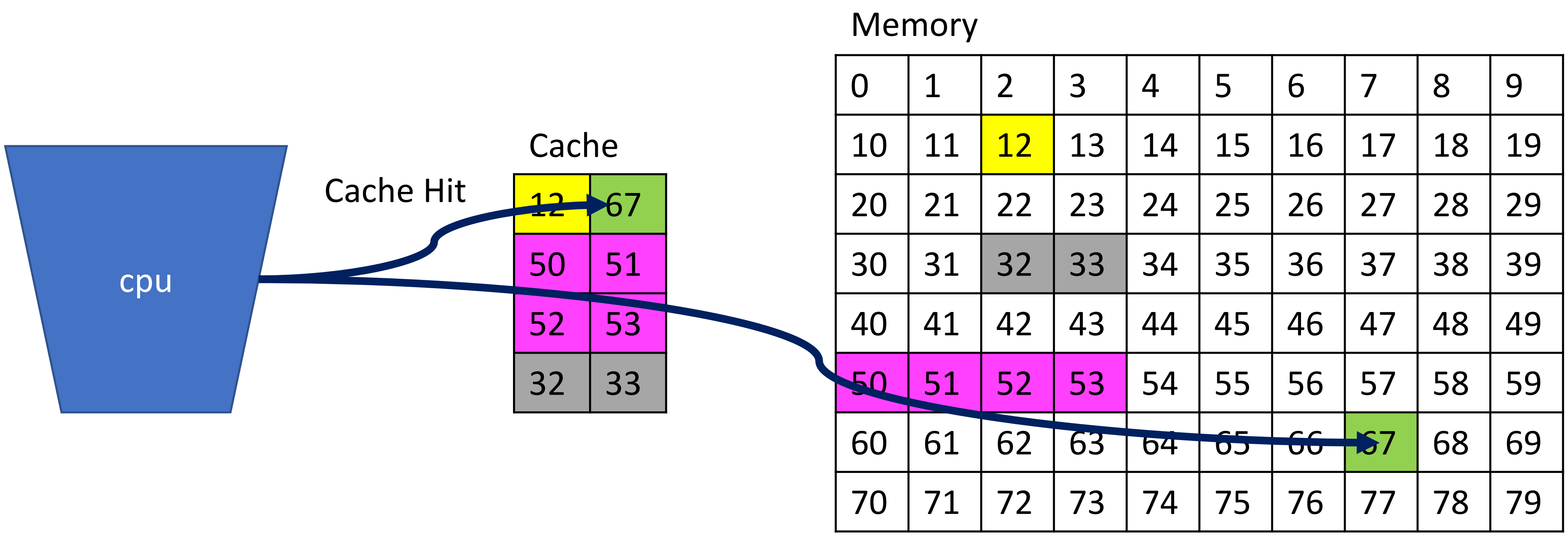
$10: Slow & Large



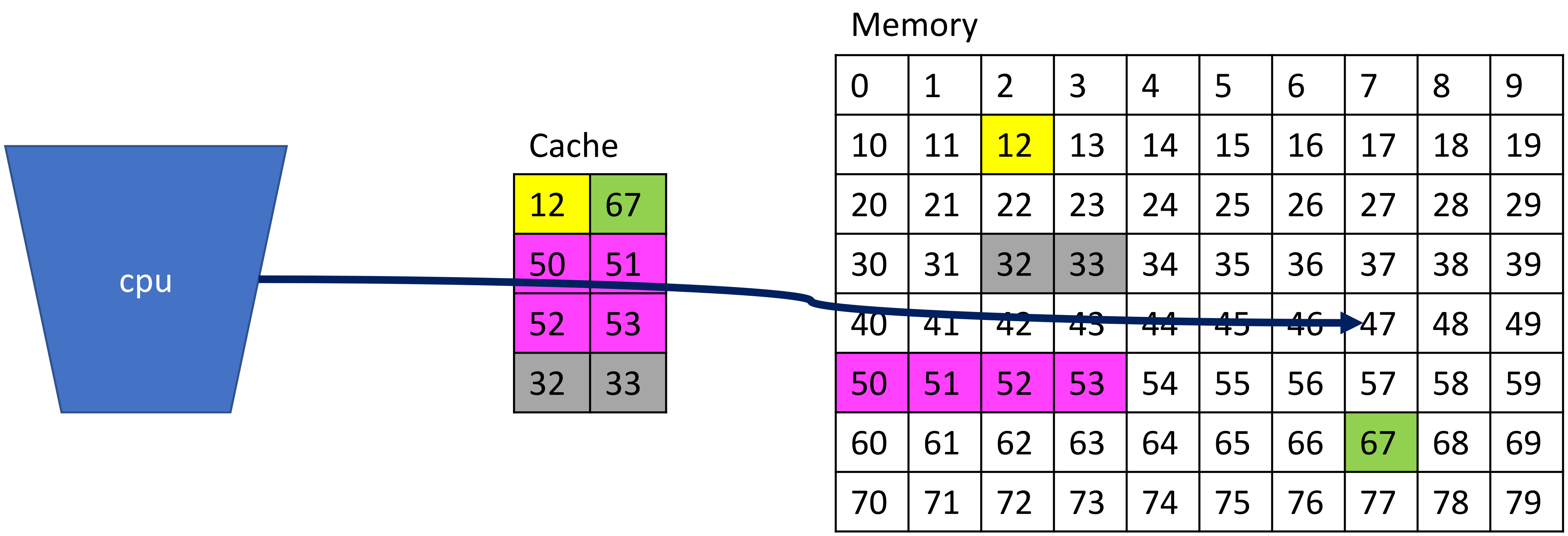
Cache: The basic idea



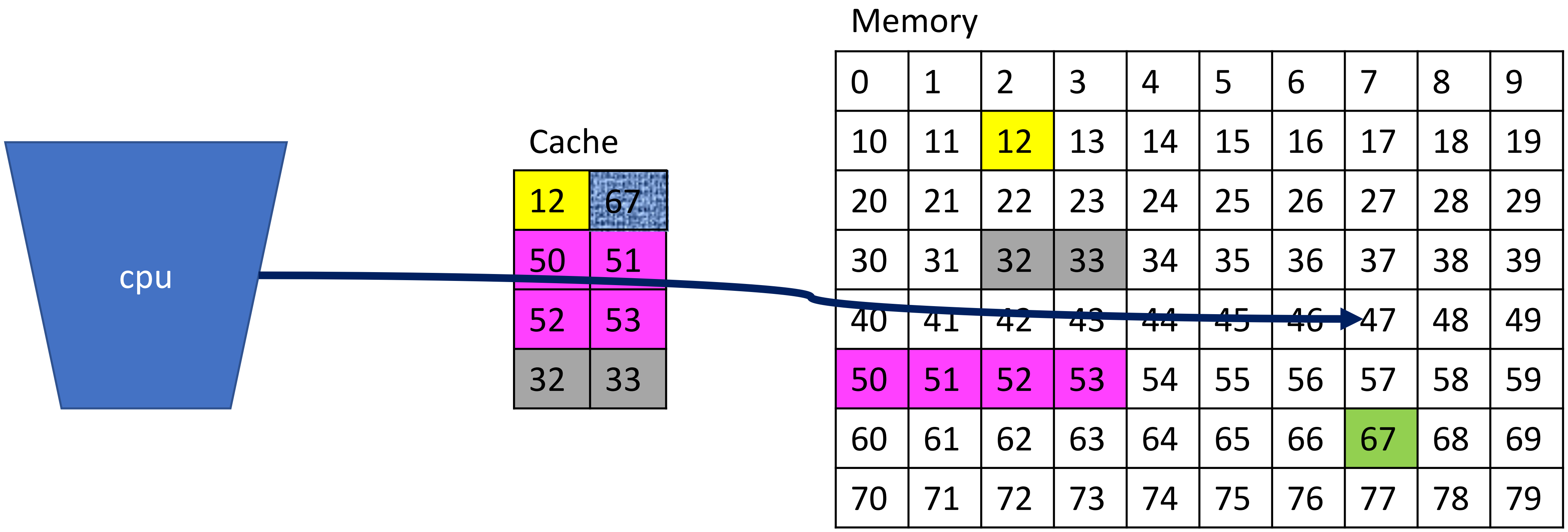
Cache Hit


Memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
Cache
12
67
50
51
52
53
32
33
Cache Hit
cpu

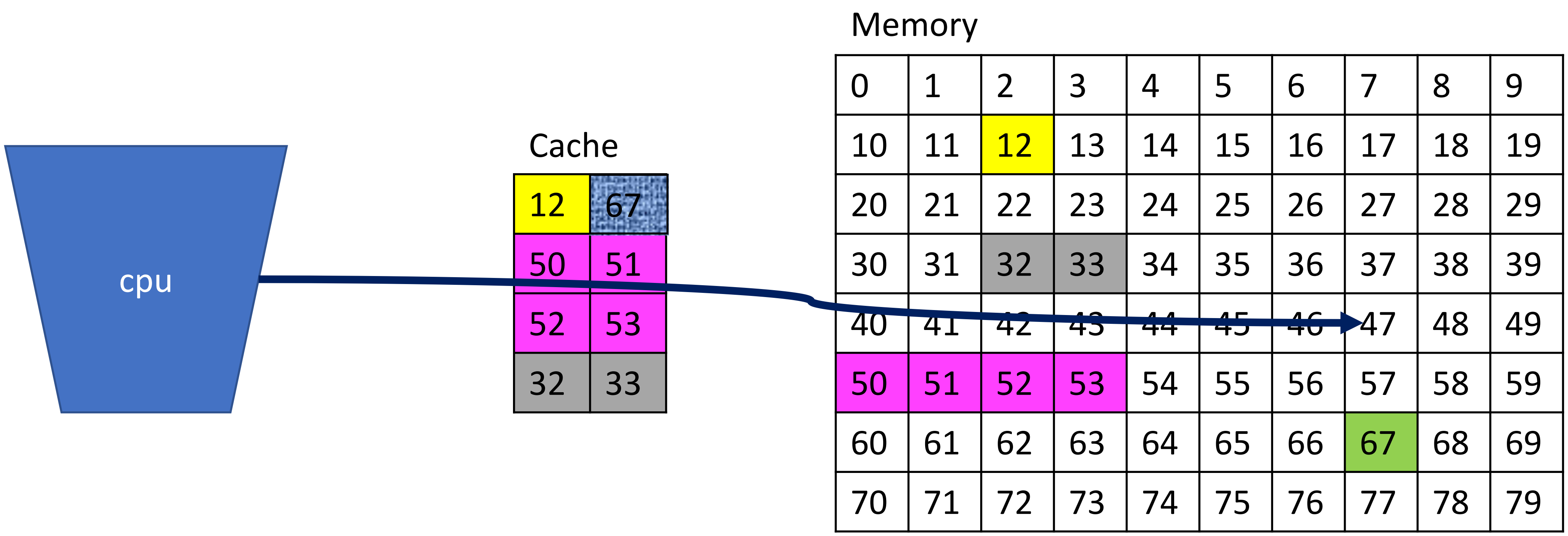
Cache Miss


cpu
Cache
12
67
50
51
52
53
32
33
Memory
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79

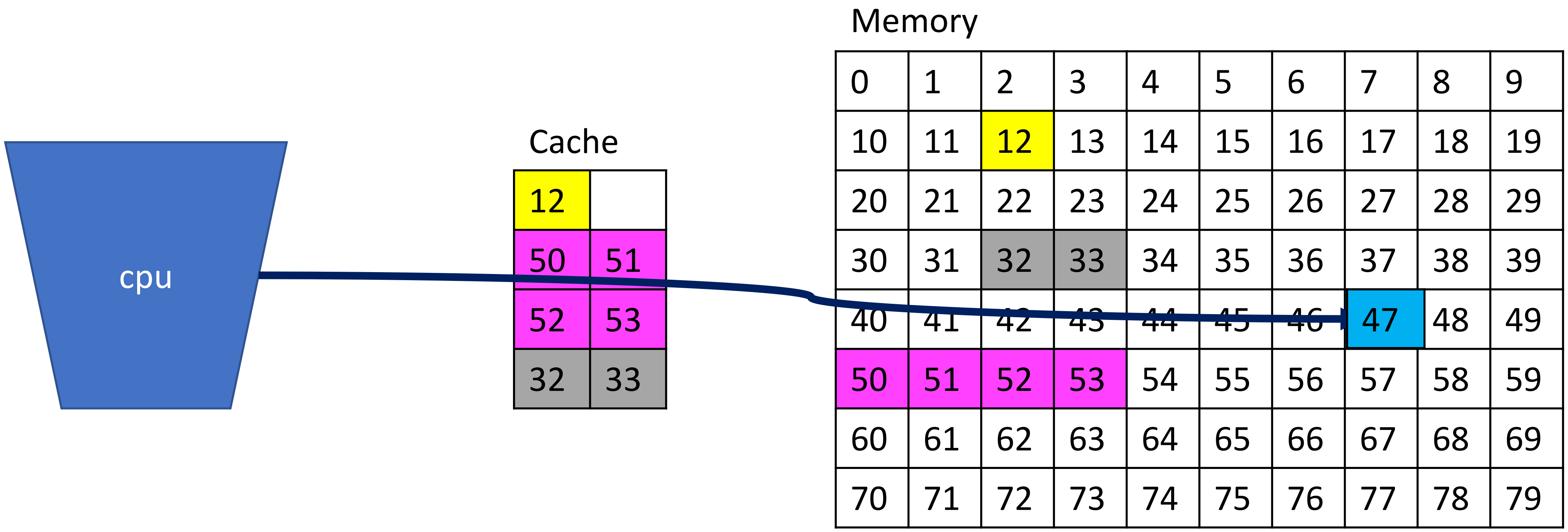
Cache Miss Service: 1) Choose byte to drop



Cache Miss Service: 2) write back



Cache Miss Service: 3) Read In



Access Locality

- The cache is effective If most accesses are hits.
  - Cache Hit Rate is high.
- **Temporal Locality**: Multiple accesses to **same** address within a short time period

Spatial locality

- **Spatial Locality**: Multiple accesses to close-together addresses in short time period.
  - The difference between two sums.
  - Counting words by sorting
- Benefiting from spatial locality
  - Memory is partitioned into **Blocks/Lines** rather than single bytes.
  - Moving a block of memory takes much less time than moving each byte individually.
  - Memory locations that are close to each other are likely to fall in the same block.
  - Resulting in more cache hits.

Cache: Lines / Blocks

Memory

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

Supports Spatial locality

Unsorted word count / poor locality

```
=== unsorted list:
the,vernacular,but,as,for,you,ye,carrion,rogues,turning,to,
```

- Consider the memory access to the dictionary D:
- Count without sort:
  D[the]=12332,…,D[but]=943,………,D[vernacular]=10,………….,D[for]=..
- Temporal locality for very common words like “the”
- No spatial locality

sorted word count / good locality

```
=== sorted list:
lines,lingered,lingered,lingered,lingered,lingered,lingerin
g,lingering,lingering,lingering,lingering,lingering,lingeri
ng,lingering,lingers,lingo,lingo,lining,link,link,linked,li
nked,linked,linked,links,links
```

Entries to D are added one at a time.

1. D[lines]=33
2. D[lines]=33, D[lingered]=5
3. D[lines]=33, D[lingered]=5, D[lingering]=8

Assuming new entries are added at the end, this gives spatial locality.

Spatial locality makes code run faster

Summary

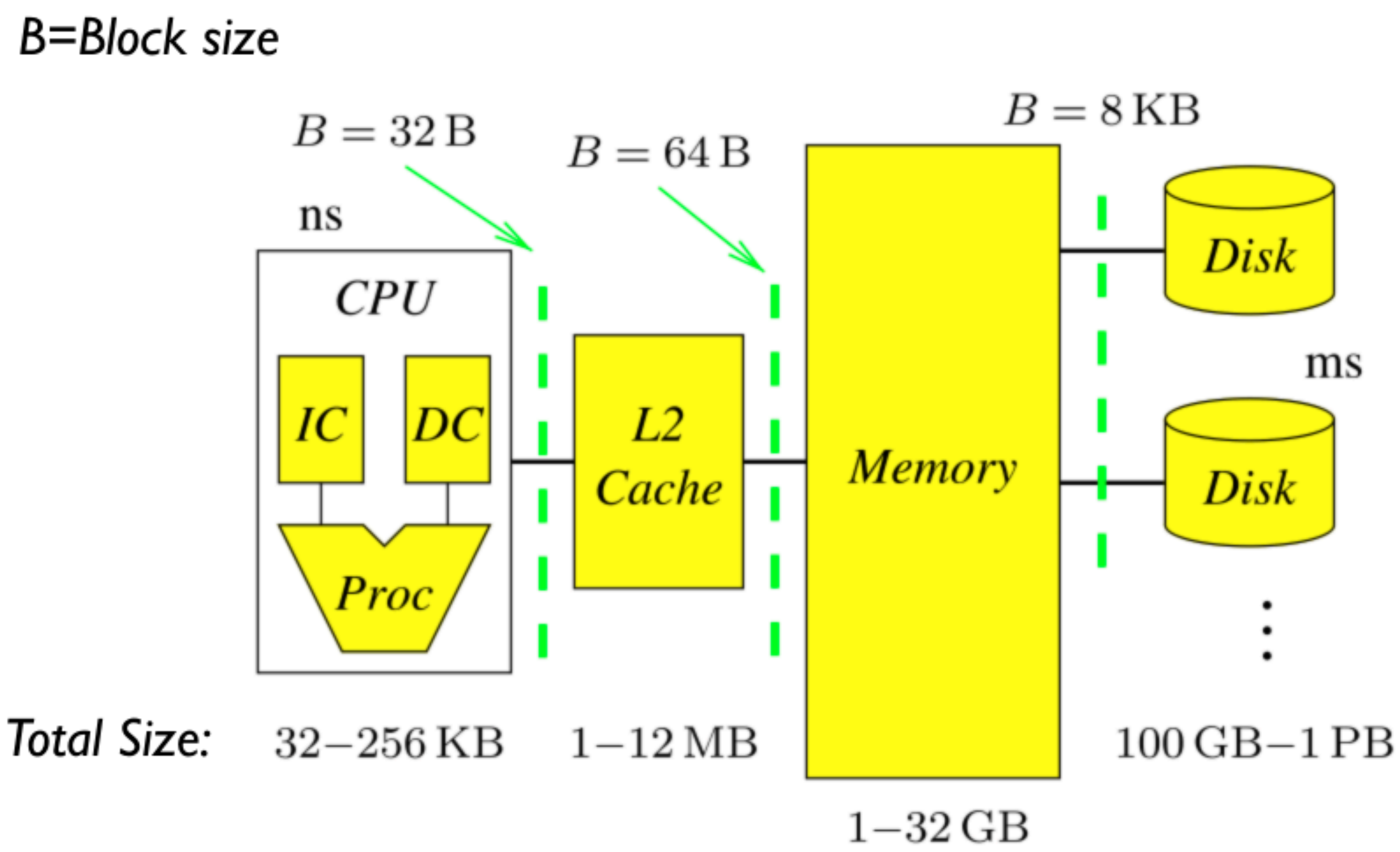
- Caching reduces storage latency by bringing relevant data close to the CPU.
- This requires that code exhibits access locality:
  - Temporal locality: Accessing the same location multiple times
  - Spatial locality: Accessing neighboring locations.

The memory Hierarchy

The Memory Hierarchy

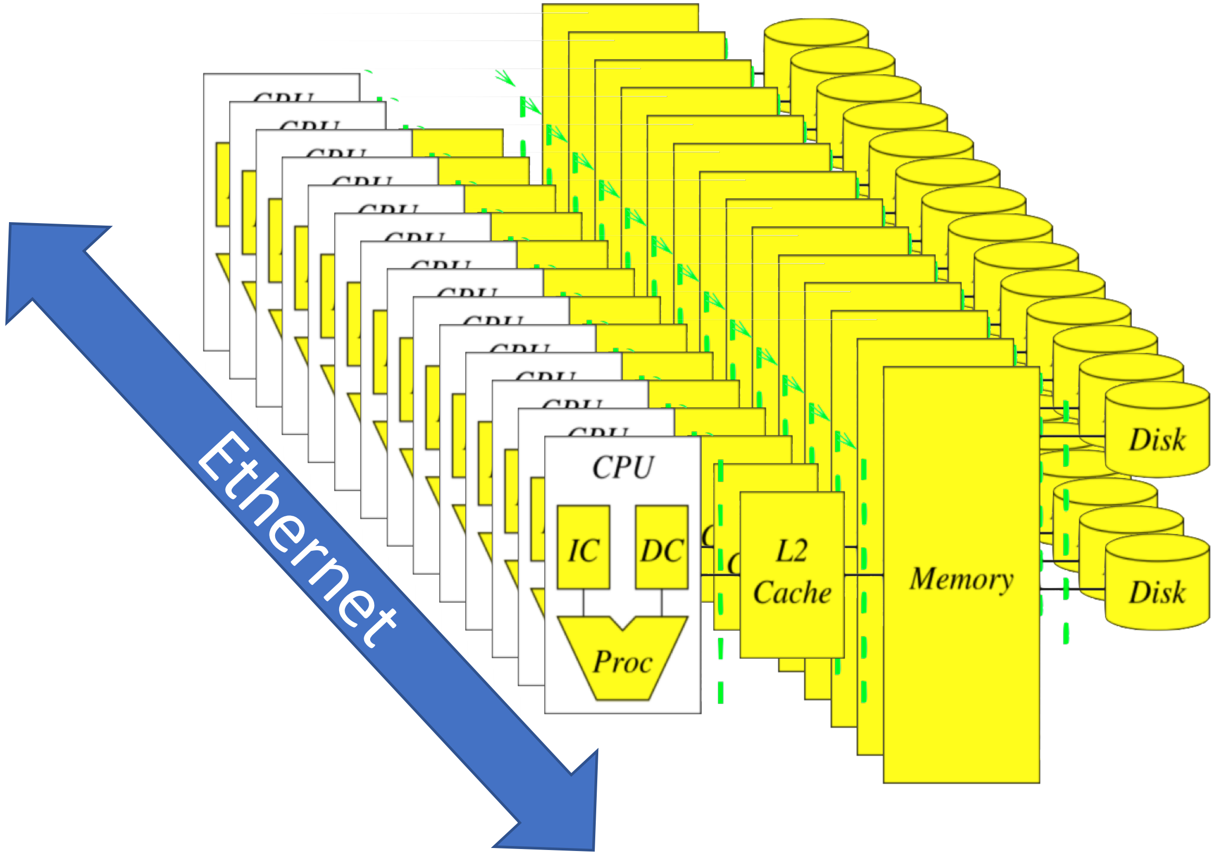
- Real systems have a several levels storage types:
  - Top of hierarchy: Small and fast storage close to CPU
  - Bottom of Hierarchy: Large and slow storage further from CPU
- Caching is used to transfer data between different levels of the hierarchy.
- Programmer / compiler is oblivious:
  - The hardware provides an **abstraction** : memory looks like like a single large array.
- But performance depends on program's access pattern.

The Memory Hierarchy



Computer clusters extend the memory hierarchy

- A data processing cluster is simply many computers linked through an ethernet connection.
- Storage is shared
- Locality: Data to reside on the computer that will use it.
- “Caching” is replaced by “Shuffling”
- Abstraction is spark RDD.



Sizes and latencies in a typical memory hierarchy.

	CPU (Registers)	L1 Cache	L2 Cache	L3 Cache	Main Memory	Disk Storage	Local Area Network
Size (bytes)	1KB	64KB	256KB	4MB	4-16GB	4-16TB	16TB - 10PB
Latency	300ps	1ns	5ns	20ns	100ns	2-10ms	2-10m
Block size	64B	64B	64B	64B	32KB	64KB	1.5-64KB

12 orders of magnitude

6 orders of magnitude

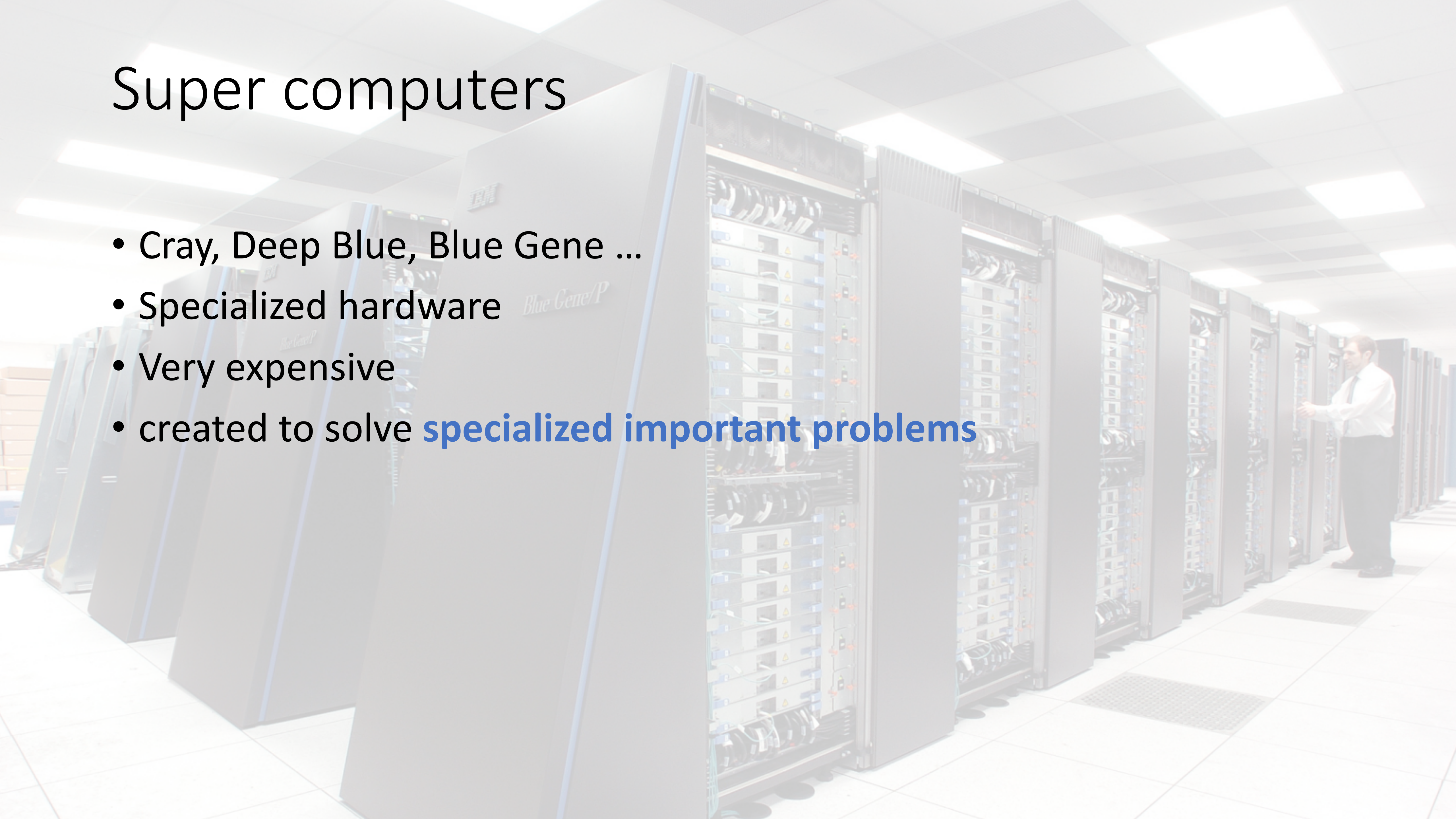
Summary

- Memory Hierarchy: combining storage banks with different latencies.
- Clusters: multiple computers, connected by ethernet, that share their storage.

A short history of affordable massive computing.

Super computers

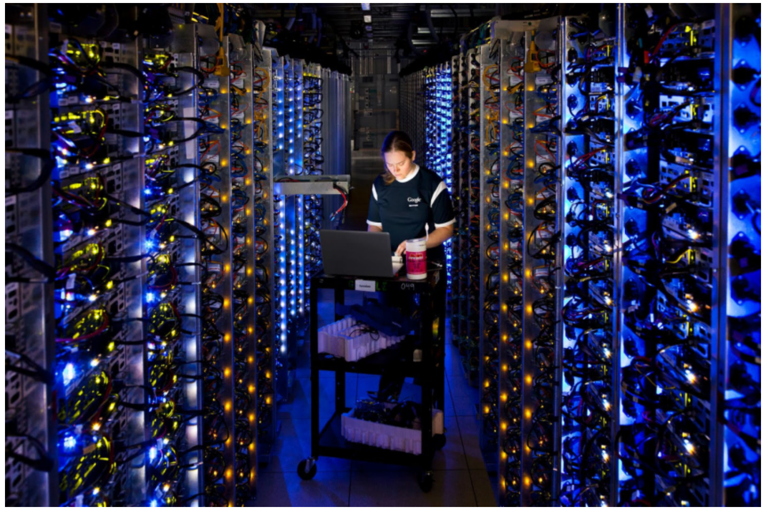
- Cray, Deep Blue, Blue Gene …
- Specialized hardware
- Very expensive
- created to solve **specialized important problems**



Data Centers

Data Centers

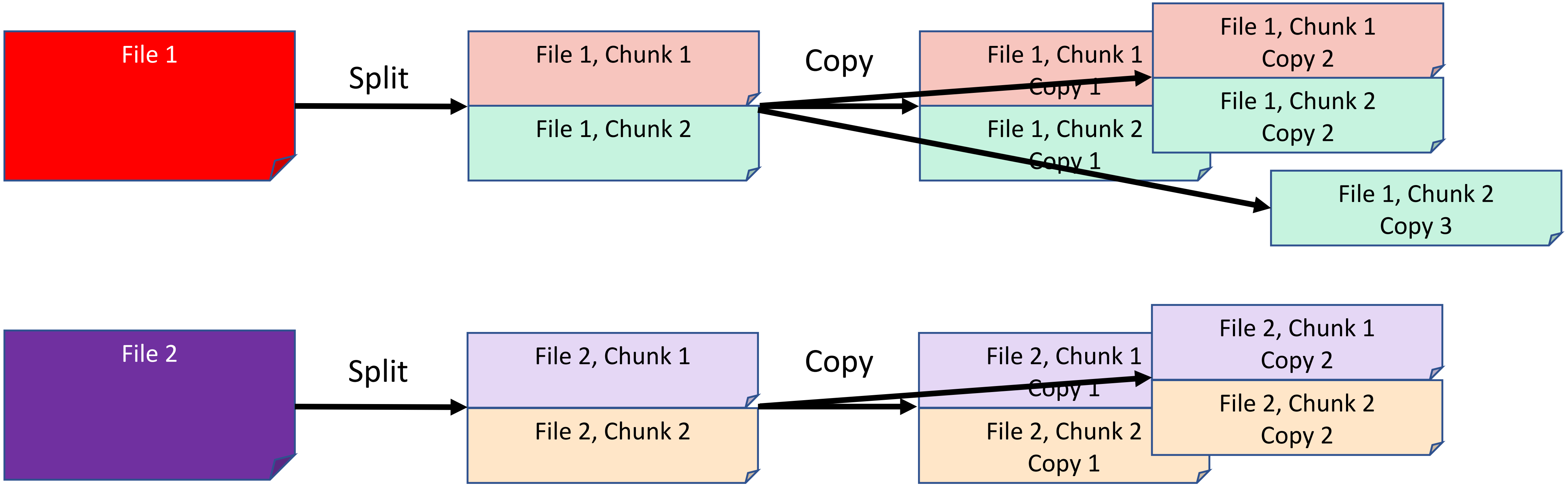
- The physical aspect of ”the cloud”
- Collection of commodity computers
- VAST number of computers (100,000’s)
- Created to provide computation for large and small organizations.
- Computation as a commodity.



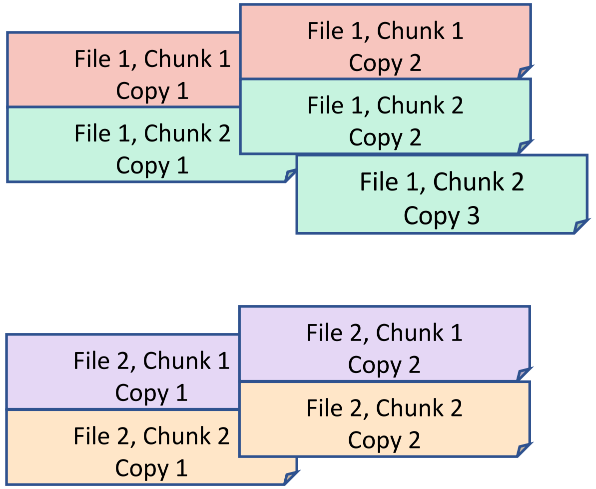
Making History: Google 2003

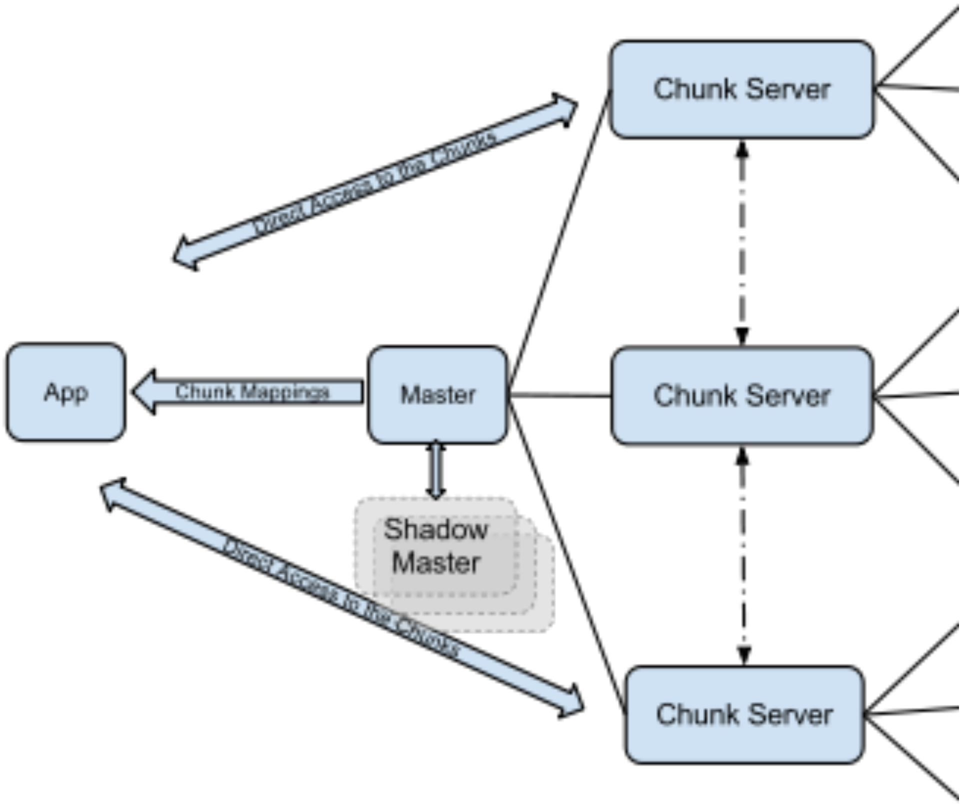
- Larry Page and Sergey Brin develop a method for storing very large files on multiple commodity computers.
- Each file is broken into fixed-size **chunks.**
- Each chunk is stored on multiple **chunk servers**.
- The locations of the chunks is managed by the **master**

HDFS: Chunking files



HDFS: Distributing Chunks

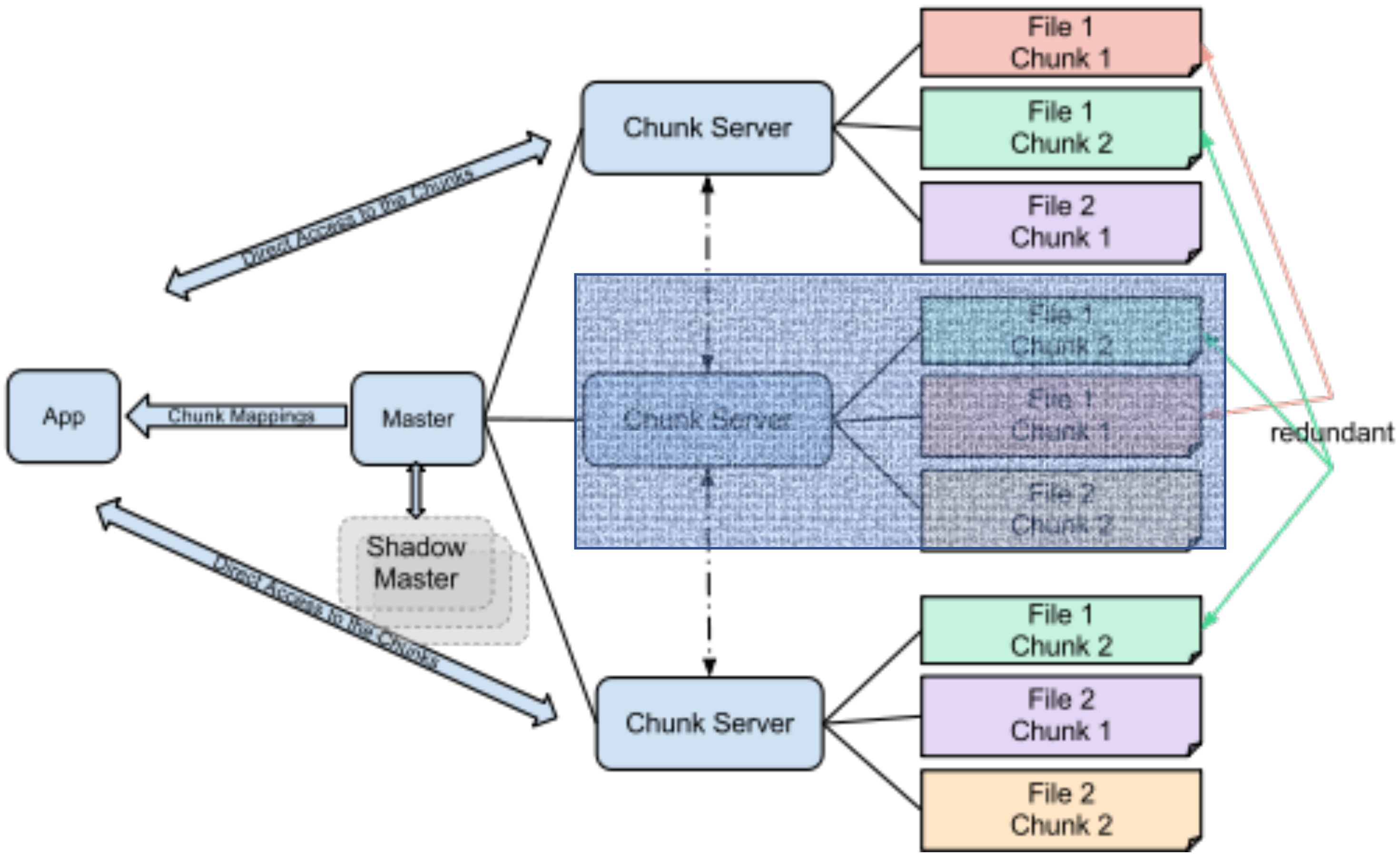




Properties of GFS/HDFS

- **Commodity Hardware:** Low cost per byte of storage.
- **Locality:** data stored close to CPU.
- **Redundancy:** can recover from server failures.
- **Simple abstraction:** looks to user like standard file system (files, directories, etc.) Chunk mechanism is hidden.

Redundancy



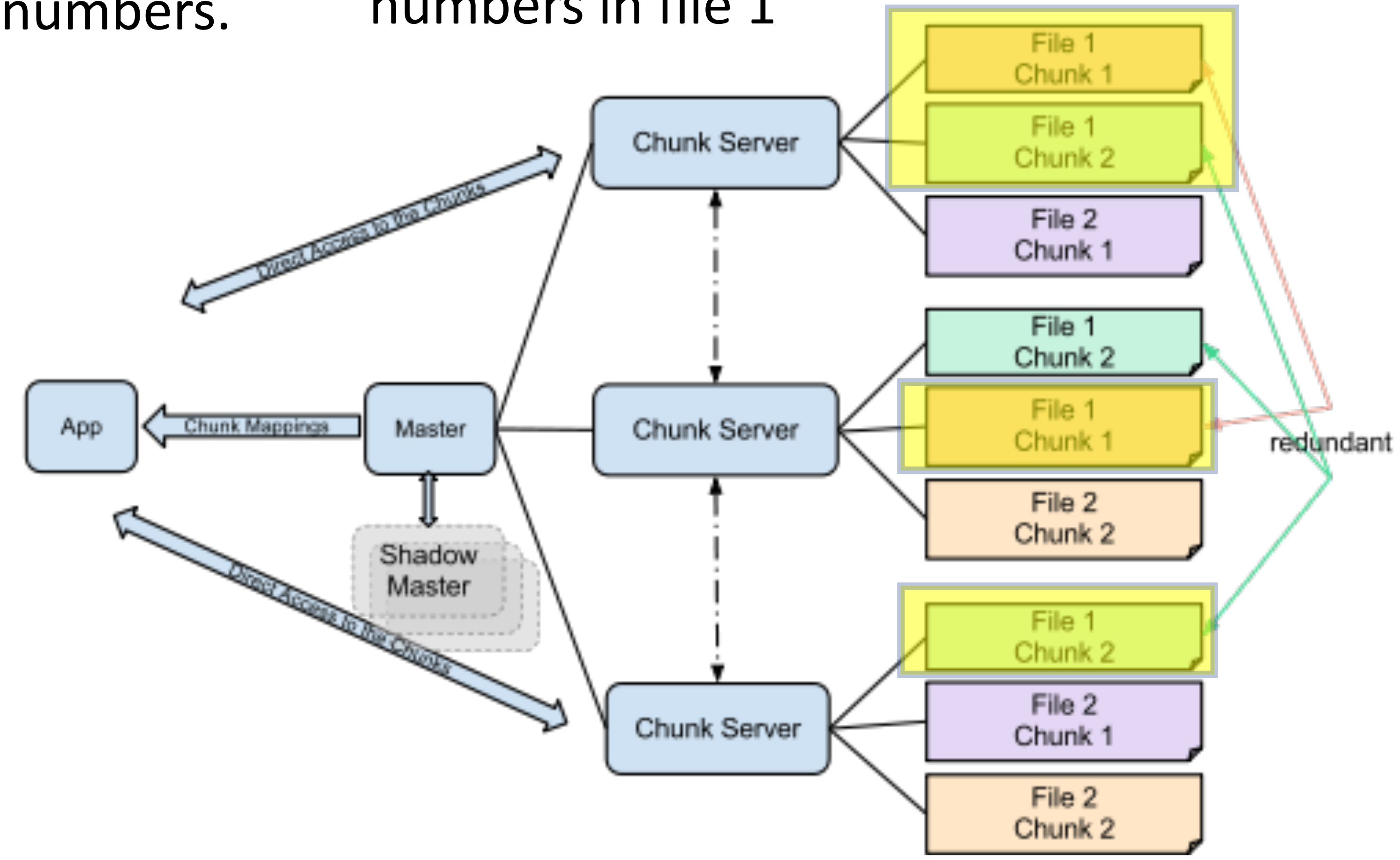
Chunk Server
File 1
Chunk 1
File 1
Chunk 2
File 2
Chunk 1
Direct Access to the Chunks
App
Chunk Mappings
Master
Chunk Server
File 1
Chunk 2
File 1
Chunk 1
File 2
Chunk 2
redundant
Shadow
Master
Direct Access to the Chunks
Chunk Server
File 1
Chunk 2
File 2
Chunk 1
File 2
Chunk 2

Parallelism

Assume File 1 contains a list of numbers.

Task:
Sum all of the numbers in file 1

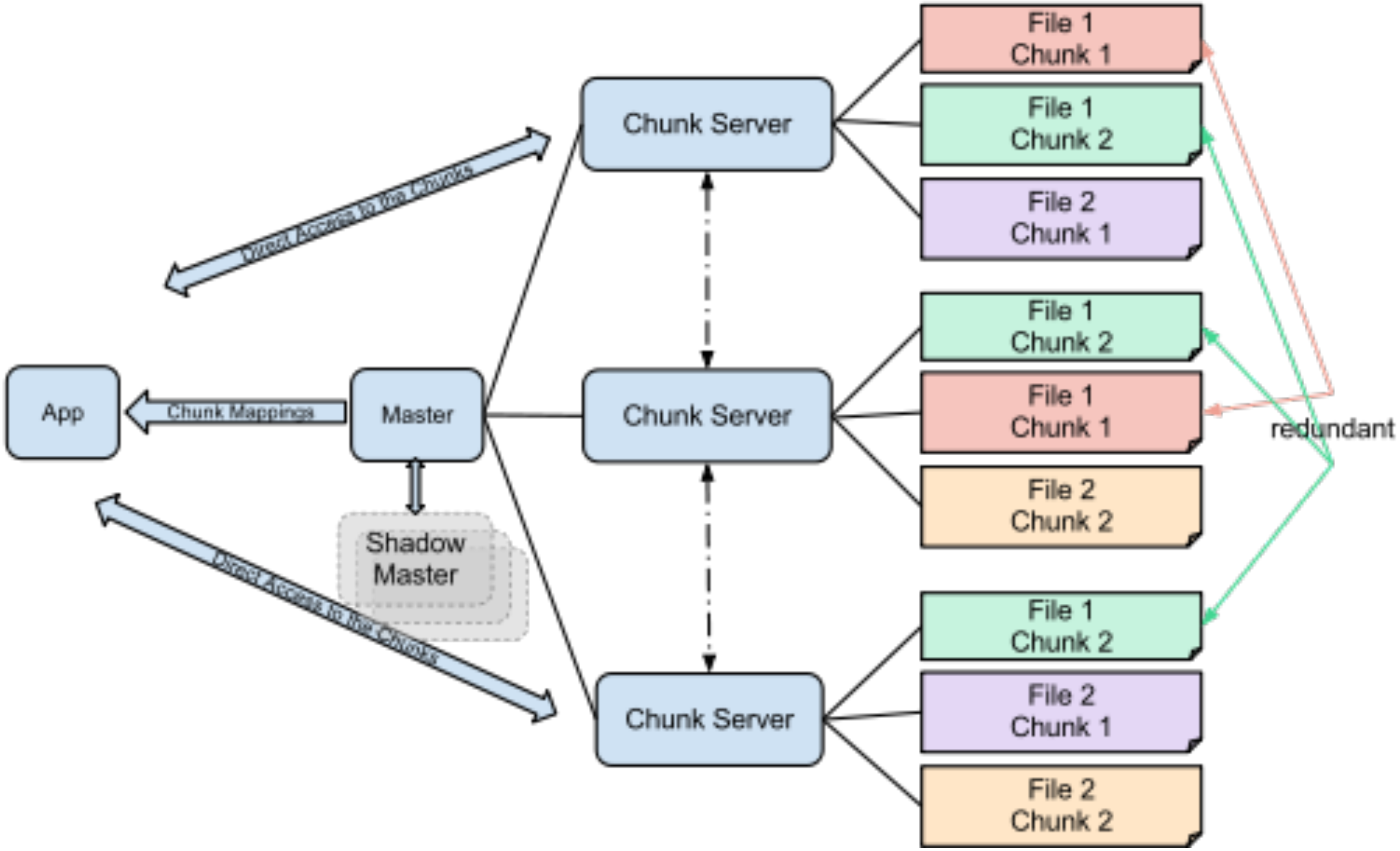
Serial computation: do everything on one computer



Parallel method: process each chunk on a separate computer, then combine.

Locality

Because of redundancy it is likely that at any moment there exists an available worker that contains the chunk the master wishes to process.



Map-Reduce

- HDFS is a **storage** **abstraction**
- **Map-Reduce** is a **computation** **abstraction** that works well with HDFS
- Allows programmer to specify parallel computation without knowing how the hardware is organized.
- We will describe Map-Reduce, using Spark, in a later section.

Spark

- Developed by Matei Zaharia , amplab, 2014
- Hadoop uses shared **file system** (disk)
- Spark uses shared **memory** – faster, lower latency.
- Will be used in this course

- Recall word count by sorting,
  we will redo it using map-reduce!

Summary

- Big data analysis is performed on large clusters of commodity computers.
- HDFS (Hadoop file system): break down files to chunks, make copies, distribute randomly.
- Hadoop Map-Reduce: a computation abstraction that works well with HDFS
- Spark: Sharing **memory** instead of sharing **disk**.