CSE 250: The Memory Hierarchy

Lecture 36

Dec 01, 2023

Reminders

- PA3 Implementation due Sun, Dec 3
- Course Evals Bonus
  - Get to 90% completion across all 3 sections, we'll release an exam question.
  - More details to be posted on Piazza.

Lies!

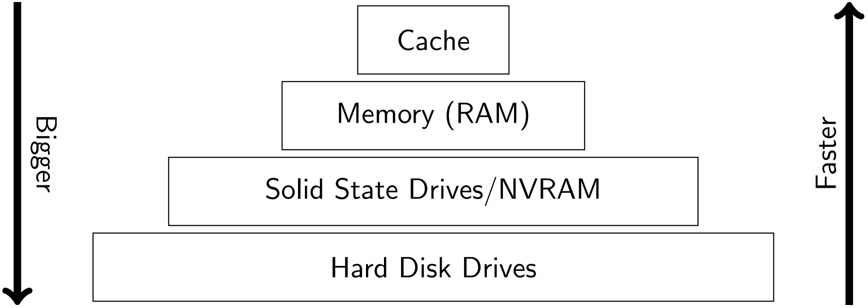
- **Lie 1:** Any array access is $O(1)$
  - This is the RAM model of computation
    - Simple, useful, but not perfect
  - Real-World Hardware isn't this elegant
    - The Memory Hierarchy: L1 Cache, L2 Cache, L3 Cache, RAM, SSD, HDD, Tape...
    - Non-Uniform Memory Access CPUs: AMD Ryzen
- **Lie 2:** The constant factors don't matter

These are useful simplifications at 50k ft, but they don't tell the whole story.

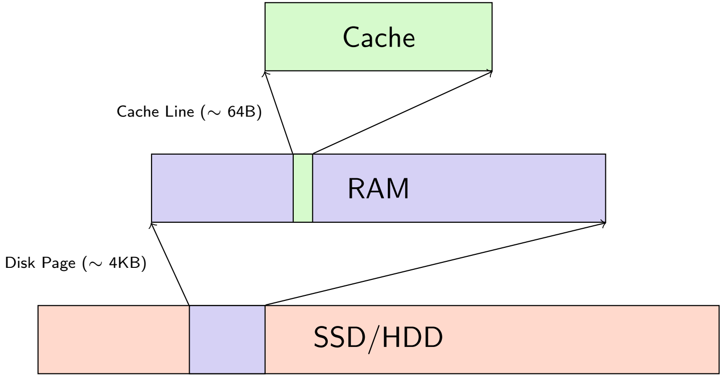
Algorithm Bounds

- Runtime Complexity
  - The algorithm takes $O(\ldots)$ steps/cpu cycles/time
- Memory Complexity
  - The algorithm needs $O(\ldots)$ MB of RAM
- IO Complexity
  - The algorithm performs $O(\ldots)$ accesses to slower memory.
  - Sometimes separately tracks reads and writes.
  - Sometimes considers > 2 memory speeds.

The Memory Hierarchy (simplified)



The Memory Hierarchy (simplified)



Reading an Array Element

Is the array element in cache?

- **Yes**: Return it (1-4 clock cycles)
- **No**: Is the array entry in RAM?
  - **Yes**: Load it from RAM into cache (10s of clock cycle)
  - **No**: Load it from SSD (100s of clock cycles)

- 1s of clock cycles: **Tiny Constant**
- 10s of clock cycles: **So-So Constant**
- 100s of clock cycles: **Huge Constant**

Reading an Array Element

It matters whether we're reading from cache, memory, or disk!

Today: Memory vs Disk

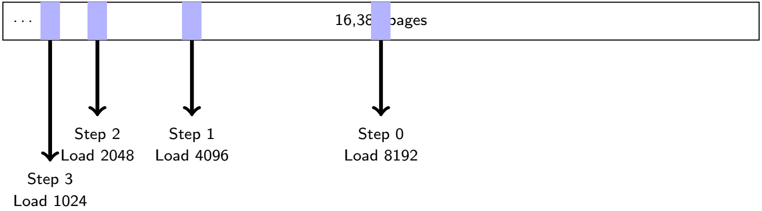
Ground Rules: Disk vs RAM

- All data starts off in a file on disk
  - Data has to be in RAM before we can access it
  - Data is loaded in 4KB chunks ("pages")
  - The amount of available RAM is finite.
  - Deallocating a page is one instruction
    - ... unless it was modified and needs to be written back
- 3 features describe an algorithm
  - The number of instructions (runtime complexity)
  - The number of disk reads/writes (IO complexity)
  - The number of pages of RAM required (memory complexity)

Similar rules apply to any pair of levels of the memory hierarchy.

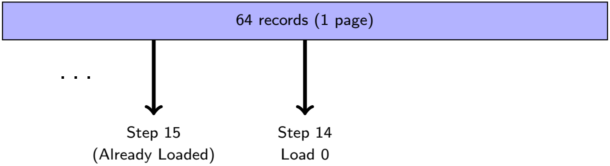
Binary Search

- `Map<K,V>` as an Array: 2^{20} ($\sim$ 1M) Records
  - 64 bytes each (8 byte key, 56 byte value)
  - 64 MB of data, 16,384 4k pages, 64 records/page
- Binary Search: $\sim \log(2^{20}) = 20$ steps.
  - Answer at position 0



Binary Search

- `Map<K,V>` as an Array: 2^{20} ($\sim$ 1M) Records
  - 64 bytes each (8 byte key, 56 byte value)
  - 64 MB of data, 16,384 4k pages, 64 records/page
- Binary Search: $\sim \log(2^{20}) = 20$ steps.
  - Answer at position 0
  - ... 13 steps, 13 loads, then....



Binary Search

- Steps 0-14 each load 1 page (15 pages loaded)
  - Sloooooooooow...
- Steps 15-19 access the same page as step 14
  - Fast!

What's the memory complexity?

How does it scale with the # of records?

Complexity

- N: records total
- R: records size (in Bytes)
- P: page size (in Bytes)
- $C = \left\lfloor \frac{R}{P} \right\rfloor$ records per page

Binary Search Complexity (Memory)

- **Stage 1**: Each page is never used again, can discard immediately.
- **Stage 2**: All use the same page

The maximum amount of memory in use <u>at one time</u> is 1 page.
The <u>Working Set</u> size is 1 page

Binary Search Complexity (IO)

- 1 page always has 64 records
  - The last 6 binary search steps are all on the same page.
  - With scaling N
    - 2^{21} records (32GB): 21 binary search steps, 16 loads
    - 2^{22} records (64GB): 22 binary search steps, 17 loads
    - 2^{23} records (128GB): 23 binary search steps, 18 loads

Binary Search Complexity (IO)

- Overall Binary Search Runtime: $O(\log N)$ steps
- Behavior goes through two stages
  - **Stage 1**: Each request goes to a new page (e.g., 0-13)
    - $\log(N) - \log(C)$ $(= \log(N) - \log\left(\frac{R}{P}\right))$
  - **Stage 2**: One load for all requests (e.g., 14-20)
    - $\log(C)$ steps

C is a constant; Total IO complexity is $O(\log N)$

How do we improve Binary Search?

- **Observation 1**
  - $2^{20} \times \texttt{sizeof}(\texttt{key} + \texttt{data}) = 2^{20} \times 64\ B = 64\ MB$ of records

    vs
  - $2^{20} \times \texttt{sizeof}(\texttt{key}) = 2^{20} \times 8\ B = 8\ MB$ of keys
- **Observation 2**
  - We don't care about which array index the record is at...
    - ... only the page it's on
    - ... and each page stores a contiguous range of keys

Fence Pointers

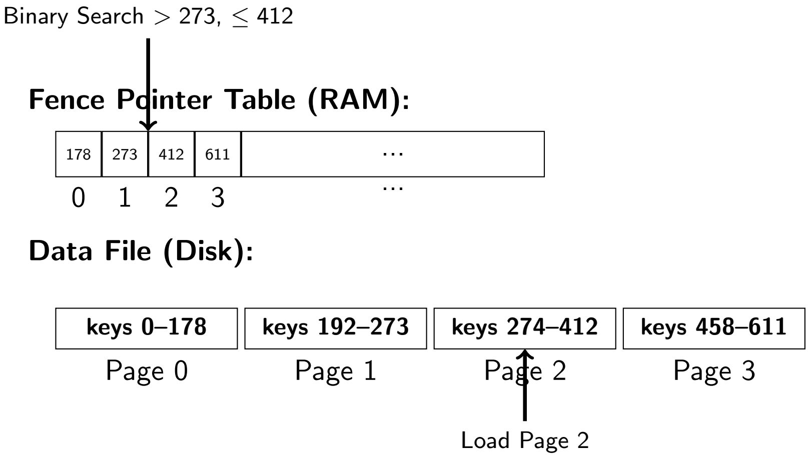
Idea: Store a list of the greatest keys on each page in memory.

- N records; 64 records/page; $\frac{N}{64}$ keys
  e.g. $N = 2^{20}$ records; needs 2^{14} keys
  - 2^{20} 64 byte records $=$ 64 MB
  - 2^{14} 8 byte keys $= 2^{19}$ bytes $=$ 512 **K**B

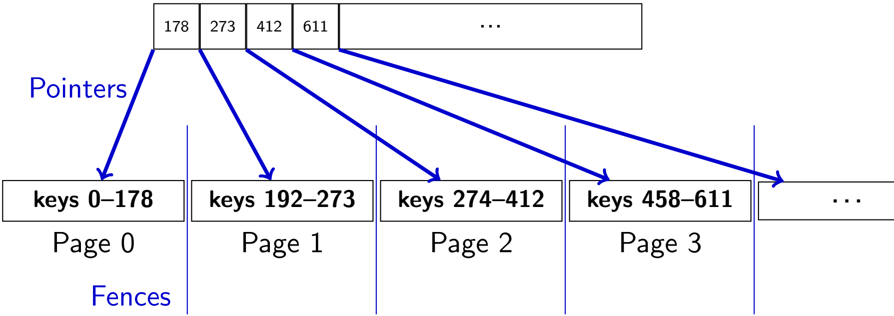
RAM: | $2^{14} = 16{,}384$ keys; 128 pages (Fence Pointer Table)

Disk: | 16,384 pages (Actual Data)

Example


Binary Search $> 273, \leq 412$
Fence Pointer Table (RAM):
178
273
412
611
...
0
1
2
3
...
Data File (Disk):
keys 0–178
keys 192–273
keys 274–412
keys 458–611
Page 0
Page 1
Page 2
Page 3
Load Page 2

Example (Why "fence pointer"?)



Fence Pointers

- **Step 1**: Binary search on the Fence Pointer Table
  - All in-memory (assuming In Memory Fence Pointer Table)
  - IO complexity $= 0$
- **Step 2**: Load page
  - One load
  - IO Complexity $= 1$
- **Step 3**: Binary search within page
  - All in-memory
  - IO Complexity $= 0$

Total IO: One page loaded ($O(1)$)

Fence Pointers

- **Step 1**: The entire fence pointer table is in-memory.
  - The fence pointer table needs to be in-memory always.
- **Steps 2,3**: One extra page loaded

Total Memory: Fence Pointer Table + 1 ($O(N+1) = O(N)$)

Fence Pointers

We can do better...