CSE 250

Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

The Memory Hierarchy

Announcements

- Autograder for PA4 Tests is now open

LIES!

Lie #1: Accessing any element of an array of any length is ***O*(1)**

- This assumes the "RAM" model of computation
    - Simple, but not perfect
- Real-world hardware isn't this simple
    - Memory is hierarchical
    - Non-Uniform Memory Access (NUMA)

Lie #2: The constants don't matter…

Algorithmic Complexity

Remember: ***O*****(*f*(*n*))** placed bounds on *growth functions* in general. Not necessarily only for runtime growth functions…

Algorithmic Complexity

Remember: ***O*(*f*(*n*))** placed bounds on *growth functions* in general. Not necessarily only for runtime growth functions…

Runtime Bounds (or Runtime Complexity)

- The algorithm takes ***O*(...)** time

Algorithmic Complexity

Remember: ***O*(*f*(*n*))** placed bounds on *growth functions* in general. Not necessarily only for runtime growth functions…

Runtime Bounds (or Runtime Complexity)

- The algorithm takes ***O***(...) time

Memory Bounds (or Memory Complexity)

- The algorithm needs ***O***(...) storage

Algorithmic Complexity

Remember: ***O*(*f*(*n*))** placed bounds on *growth functions* in general. Not necessarily only for runtime growth functions…

Runtime Bounds (or Runtime Complexity)

- The algorithm takes ***O***(...) time

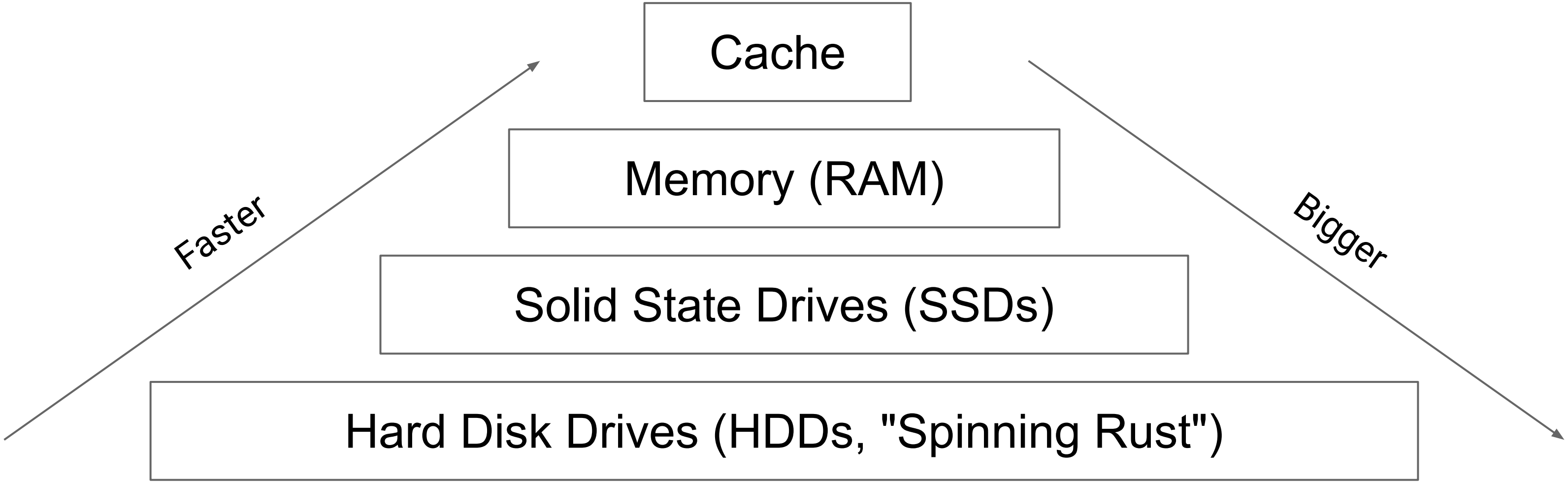
Memory Bounds (or Memory Complexity)

- The algorithm needs ***O***(...) storage

I/O Bounds (or I/O Complexity)

- The algorithm performs ***O***(...) accesses to slower memory

The Memory Hierarchy (simplified)



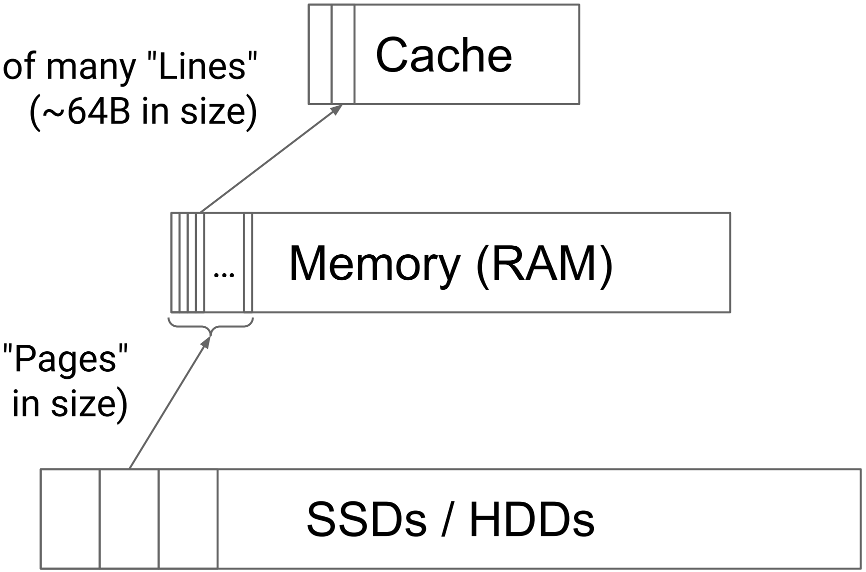
Cache

Memory (RAM)

Solid State Drives (SSDs)

Hard Disk Drives (HDDs, "Spinning Rust")

The Memory Hierarchy (simplified)



Cache is made of of many "Lines"
(~64B in size)

Disk is made up of many "Pages"
(~4KB in size)

Reading an Array Entry

In order to read an Array Entry:

1. Is the array entry in cache?

Reading an Array Entry

In order to read an Array Entry:

1. Is the array entry in cache?
    a. Yes: Return it (1-4 clock cycles)
    b. No: Is it in real memory?

Reading an Array Entry

In order to read an Array Entry:

1. Is the array entry in cache?
    a. Yes: Return it (1-4 clock cycles)
    b. No: Is it in real memory?
        i. Yes: Load it into a cache line (10s of cycles)
        ii. No: Load it from a page of virtual memory (100s of cycles)

Reading an Array Entry

In order to read an Array Entry:

Tiny constant

1. Is the array entry in cache?
    a. Yes: Return it (1-4 clock cycles)

OK constant

    b. No: Is it in real memory?

HUGE constant

        i. Yes: Load it into a cache line (10s of cycles)
        ii. No: Load it from a page of virtual memory (100s of cycles)

In practice, these constants do matter!

Ground Rules: Disk vs RAM

1. All data starts off in a file on disk
    a. Need to load data into RAM before accessing it
    b. Load data in 4KB pages
    c. Amount of RAM is finite
2. Must describe 3 features of an algorithm
    a. Number of instructions (runtime complexity)
    b. Number of data loads (I/O complexity)
    c. Number of pages of RAM required (memory complexity)

Note: Similar rules apply to any pair of levels in the hierarchy

Binary Search

Example:

2^{20} records, 64 bytes each (8 byte key, 56 byte value)

64 MB of data total, 16,384 pages, 64 records per page

How many steps to binary search this data?

Binary Search

Example:

2^{20} records, 64 bytes each (8 byte key, 56 byte value)

64 MB of data total, 16,384 pages, 64 records per page

How many steps to binary search this data? **log(2^{20}) = 20 steps**

Binary Search

Example:

2^{20} records, 64 bytes each (8 byte key, 56 byte value)

64 MB of data total, 16,384 pages, 64 records per page

Let's assume the target is at position 0

16,384 pages

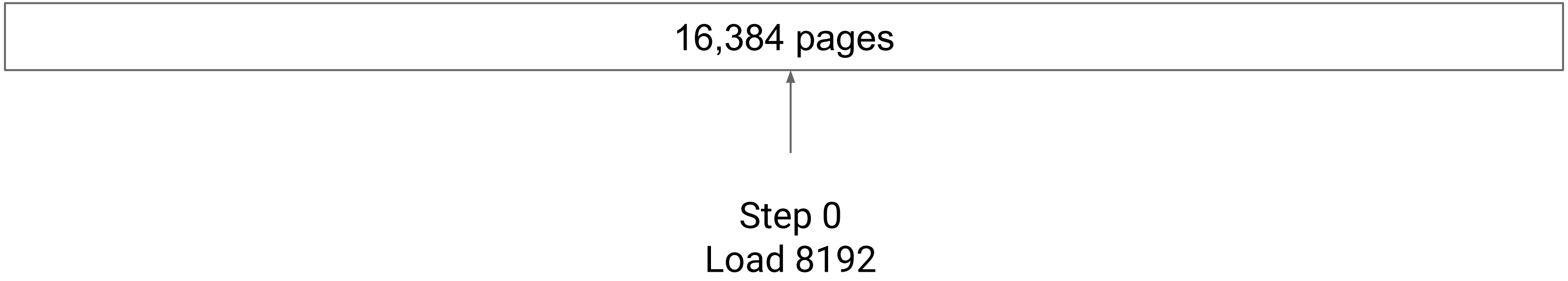
Binary Search

Example:

2^{20} records, 64 bytes each (8 byte key, 56 byte value)

64 MB of data total, 16,384 pages, 64 records per page

Let's assume the target is at position 0



16,384 pages

Step 0
Load 8192

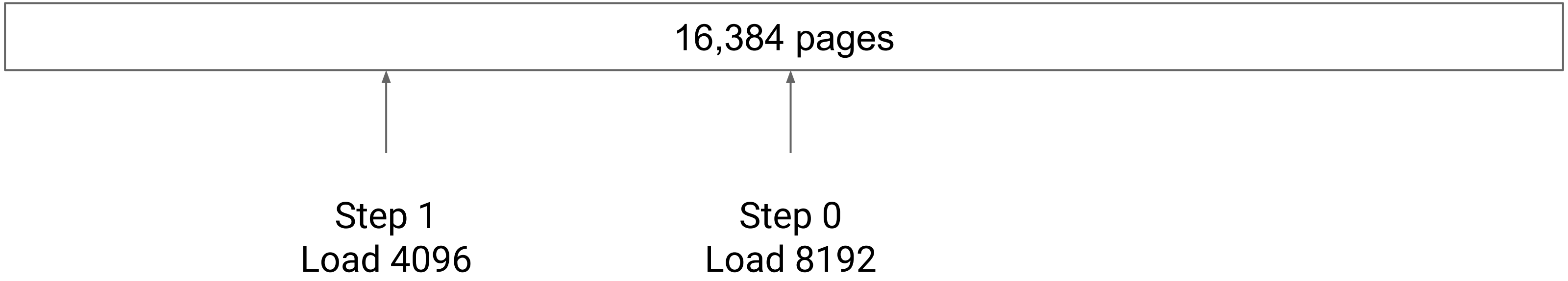
Binary Search

Example:

2^{20} records, 64 bytes each (8 byte key, 56 byte value)

64 MB of data total, 16,384 pages, 64 records per page

Let's assume the target is at position 0



16,384 pages

Step 1
Load 4096

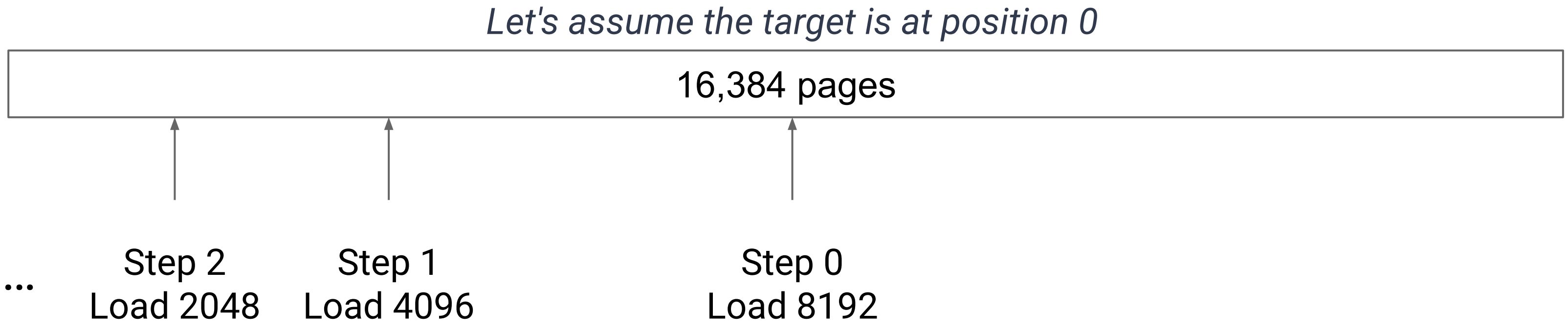
Step 0
Load 8192

Binary Search

Example:

2^{20} records, 64 bytes each (8 byte key, 56 byte value)

64 MB of data total, 16,384 pages, 64 records per page



Binary Search

Example:

2^{20} records, 64 bytes each (8 byte key, 56 byte value)

64 MB of data total, 16,384 pages, 64 records per page

Let's assume the target is at position 0

Page 0 (64 records)

Step 14
(log(16,384) = 14)
Load Page 0

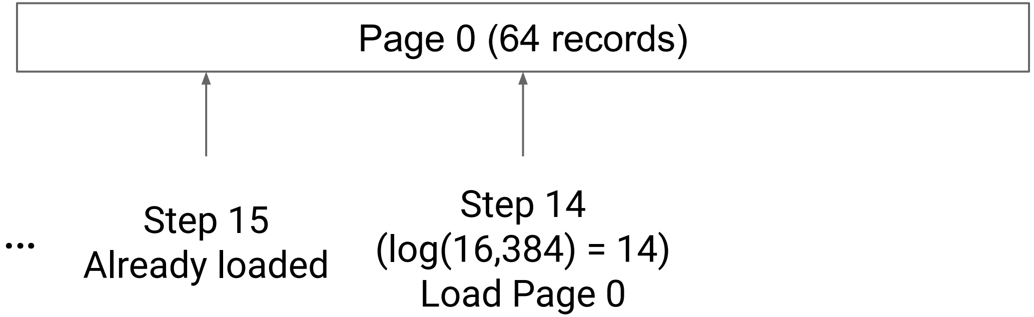
Binary Search

Example:

2^{20} records, 64 bytes each (8 byte key, 56 byte value)

64 MB of data total, 16,384 pages, 64 records per page

Let's assume the target is at position 0



Binary Search: Complexity

Steps 0 - 14: Sloooooow…each one loaded a new page (15 pages loaded)

Steps 15-19: Fast! All access the page loaded on step 14

Runtime complexity = ***O*****(log(*****n*****))**

What's the memory complexity?

Binary Search: Complexity

Steps 0 - 14: Sloooooow…each one loaded a new page (15 pages loaded)

Steps 15-19: Fast! All access the page loaded on step 14

Runtime complexity = $\boldsymbol{O(\log(n))}$

What's the memory complexity?

How many pages do we need loaded at one time?

Binary Search: Complexity

Steps 0 - 14: Slooooow…each one loaded a new page (15 pages loaded)

Steps 15-19: Fast! All access the page loaded on step 14

Runtime complexity = ***O*****(log(*****n*****))**

What's the memory complexity? ***O*****(1)**

How many pages do we need loaded at one time? **1 page…we only care about the maximum memory we will need at any one time**

Binary Search: Complexity

Steps 0 - 14: Sloooooow…each one loaded a new page (15 pages loaded)

Steps 15-19: Fast! All access the page loaded on step 14

Runtime complexity = **$O(\log(n))$**

What's the memory complexity? **$O(1)$**

How many pages do we need loaded at one time? **1 page…we only care about the maximum memory we will need at any one time**

What about I/O complexity?

Binary Search: I/O Complexity

Let's set up some variables:

- ***n*** - total number of records
- ***R*** - record size (in Bytes)
- ***P*** - page size (in Bytes)
- ***C*** - $\lfloor R/P \rfloor$ records per page

Binary Search: I/O Complexity

Binary Search does log(*n*) steps broken into two stages:

Stage 1: Each request has to load a new page into memory

Stage 2: The remaining requests all happen in the same page

Binary Search: I/O Complexity

Binary Search does log(*n*) steps broken into two stages:

Stage 1: Each request has to load a new page into memory

Stage 2: The remaining requests all happen in the same page

Remember: Our page size is fixed…***C*** records per page
Therefore: The last **log(*C*)** binary search steps are all on the same page

Binary Search: I/O Complexity

Binary Search does log(*n*) steps broken into two stages:

Stage 1: Each request has to load a new page into memory

- **log(*n*) - log(*C*)** steps

Stage 2: The remaining requests all happen in the same page

- **log(*C*)** steps

Remember: Our page size is fixed…***C*** records per page
Therefore: The last **log(*C*)** binary search steps are all on the same page

Binary Search: I/O Complexity

Binary Search does **$O(\log(n) - \log(C))$** loads from memory

Therefore: I/O complexity of Binary Search is **$\log(n)$**

Binary Search: Complexity

Binary Search Complexity:

- Runtime Complexity: $\boldsymbol{O(\log(n))}$
- Memory Complexity: $\boldsymbol{O(1)}$
- I/O Complexity: $\boldsymbol{O(\log(n))}$

How can we improve on this?

Observations

Observation 1:

- Total size of records: 64MB = 2^{20} x sizeof(key + data)
- Total size of keys only: 8MB = 2^{20} x sizeof(key)

Observation 2:

- The first stage doesn't care what array index the record is at, just the page it is on
- Each page stores a contiguous range of keys…

Fence Pointers

Idea: Precompute the greatest key stored on each page

- ***n*** total records, ***C*** records per page, ***n/C*** keys required
- For our example, 2^{20} records needs 2^{14} pages, therefore 2^{14} keys
  - 2^{20} 64 byte records need 64MB memory
  - 2^{14} 8 byte keys only needs 512KB memory
- Call this a "Fence Pointer Table" and store it in memory

RAM: 2^{14} = 16,384 keys (Fence Pointer Table)

Disk: 16,384 pages (Actual Data)

Fence Pointer Example

Binary Search for 321

RAM (Fence Pointer Table):

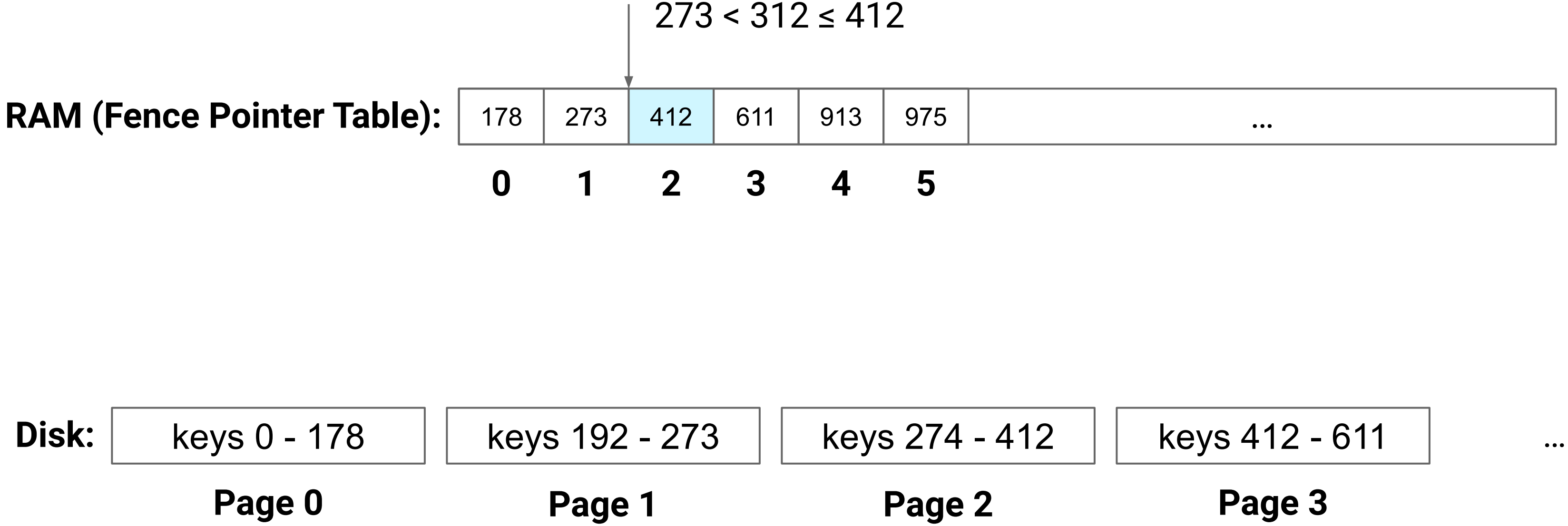
178	273	412	611	913	975	...
0	**1**	**2**	**3**	**4**	**5**	

Disk:

keys 0 - 178	keys 192 - 273	keys 274 - 412	keys 412 - 611	...
Page 0	**Page 1**	**Page 2**	**Page 3**	

Fence Pointer Example

Binary Search for 321



273 < 312 ≤ 412

RAM (Fence Pointer Table):

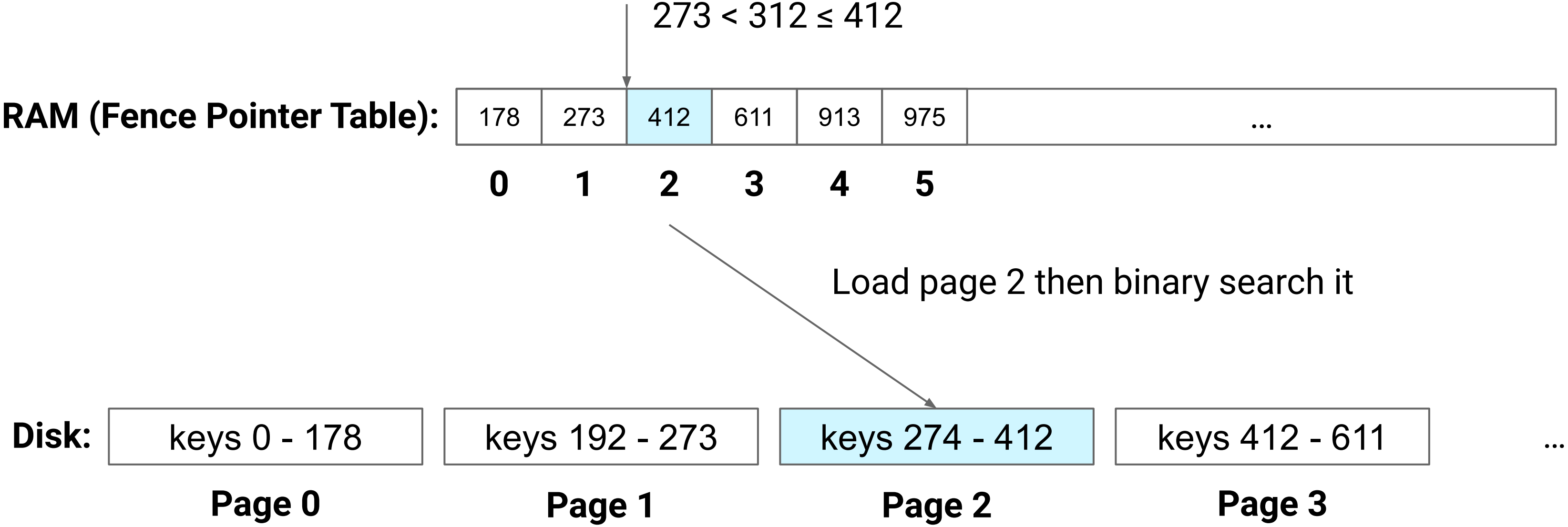
178	273	412	611	913	975	...
0	**1**	**2**	**3**	**4**	**5**	

Disk:

keys 0 - 178	keys 192 - 273	keys 274 - 412	keys 412 - 611	...
Page 0	**Page 1**	**Page 2**	**Page 3**	

Fence Pointer Example

Binary Search for 321



$273 < 312 \leq 412$

RAM (Fence Pointer Table):

178	273	412	611	913	975	...
0	**1**	**2**	**3**	**4**	**5**	

Load page 2 then binary search it

Disk:

keys 0 - 178	keys 192 - 273	keys 274 - 412	keys 412 - 611	...
Page 0	**Page 1**	**Page 2**	**Page 3**	

Binary Search with Fence Pointers

Step 1: Binary search the fence pointer table

- **$O(\log(n) - \log(C))$** steps
- All in memory, 0 disk reads

Step 2: Load page

- 1 step, 1 disk read

Step 3: Binary search within page

- **$O(\log(C))$** steps
- All in memory, 0 disk reads

Binary Search with Fence Pointers

Step 1: Binary search the fence pointer table

- **$O(\log(n) - \log(C))$** steps
- All in memory, 0 disk reads

Step 2: Load page

- 1 step, 1 disk read

Step 3: Binary search within page

- **$O(\log(C))$** steps
- All in memory, 0 disk reads

Runtime: $O(\log(n))$

I/O: $O(1)$

Memory?

Binary Search with Fence Pointers

Step 1: Binary search the fence pointer table

- **$O(\log(n) - \log(C))$** steps
- All in memory, 0 disk reads

Step 2: Load page

- 1 step, 1 disk read

Step 3: Binary search within page

- **$O(\log(C))$** steps
- All in memory, 0 disk reads

Runtime: $O(\log(n))$

I/O: $O(1)$

Memory: $O(n)$

We need the entire fence pointer table in memory at all times :(

What about Runtime/Memory Complexity?

Records per page, *C*, is a constant, size of the fence pointer table is *n* / *C*

What about Runtime/Memory Complexity?

Records per page, C, is a constant, size of the fence pointer table is n / C

Runtime Complexity: $\log(n/C) + \log(C) = O(\log(n))$

- Search the fence pointer table, then search the page

What about Runtime/Memory Complexity?

Records per page, C, is a constant, size of the fence pointer table is n / C

Runtime Complexity: $\log(n/C) + \log(C) = O(\log(n))$

- Search the fence pointer table, then search the page

I/O Complexity: 1 page read = $O(1)$

- Load the single page found by searching the fence pointer table

What about Runtime/Memory Complexity?

Records per page, C, is a constant, size of the fence pointer table is n / C

Runtime Complexity: $\log(n/C) + \log(C) = O(\log(n))$

- Search the fence pointer table, then search the page

I/O Complexity: 1 page read = $O(1)$

- Load the single page found by searching the fence pointer table

Memory Complexity: $O(n/C + C) = O(n)$

- Need to store the fence pointer table (**at all times**), and one additional page that we load after the fence pointer table search

What about Runtime/Memory Complexity?

Records per page, C, is a constant, size of the fence pointer table is n / C

Runtime Complexity: $\log(n/C) + \log(C) = O(\log(n))$

- Search the fence pointer table, then search the page

I/O Com

- Loa … e

$O(n)$ is not ideal… and what if the fence pointer table gets too big for memory?

Memory Complexity: $O(n/C + C) = O(n)$

- Need to store the fence pointer table (**at all times**), and one additional page that we load after the fence pointer table search

Improving on Fence Pointers

At some point, we will have to store the fence pointers on Disk…

In our current example with **4KB pages**, and **8B keys**,
we can fit **512 keys per page**

Improving on Fence Pointers

At some point, we will have to store the fence pointers on Disk…

In our current example with **4KB pages**, and **8B keys**,
we can fit **512 keys per page**

Idea: What if we binary search the fence pointers on disk?

Improving on Fence Pointers

With our current example:

- We can store 512 8B keys per 4KB page (2^9 keys per page)
- 2^{20} records / 64 records per page = 2^{14} pages of records
- 2^{14} fence pointer keys = 2^5 pages
- Binary search of the pointer key pages will require **log(2^5) = 5 loads**

In general: log(*n*) - log(*C*) - log(keys/page)

Improving on Fence Pointers

With our current example:

- We can store 512 8B keys per 4KB page (2^9 keys per page)
- 2^{20} records / 64 records per page = 2^{14} pages of records
- 2^{14} fence pointer keys = 2^5 pages
- Binary search of the pointer key pages will require **log(2^5) = 5 loads**

In general: log(*n*) - log(*C*) - log(keys/page) ← Still ***O*****(log(*n*))**

Improving on Fence Pointers

IO Complexity: $\log(n) - \log(C_{\text{data}}) - \log(C_{key}) = O(\log(n))$

- **C_{data} = records per page** *(ie: 64)*
- **C_{key} = keys per page** *(ie: 512)*

Can we improve our search of the on-disk Fence Pointer Table…?

Improving on Fence Pointers

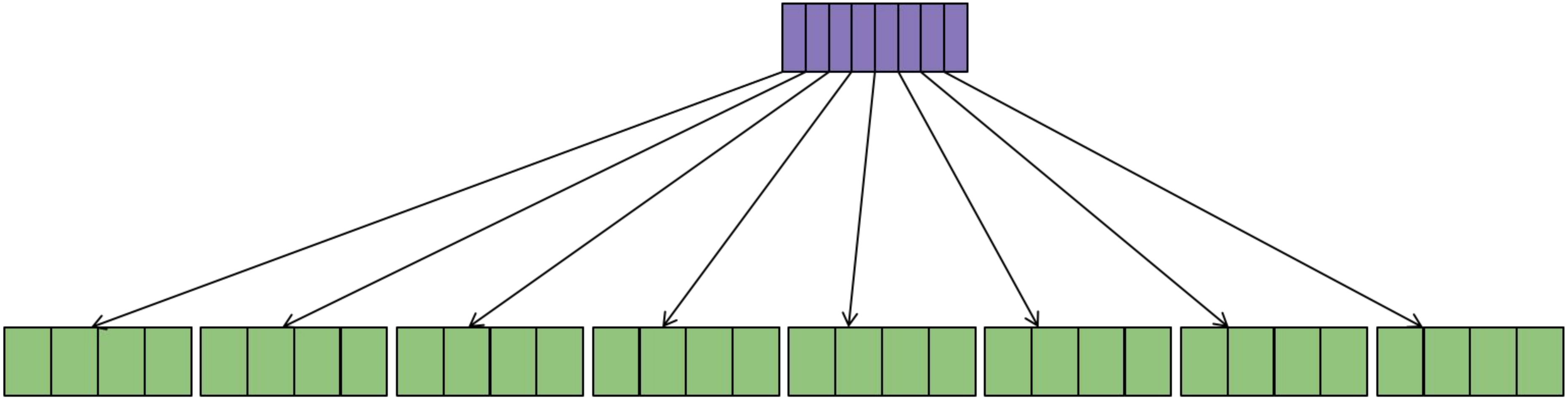
Idea: A fence pointer table for our fence pointer table!

(and if that fence pointer table is too big…a fence pointer table for that table…and so on and so on and so on…until we have one that fits in memory)

Improving on Fence Pointers

🟪 Fence pointer array (in memory)

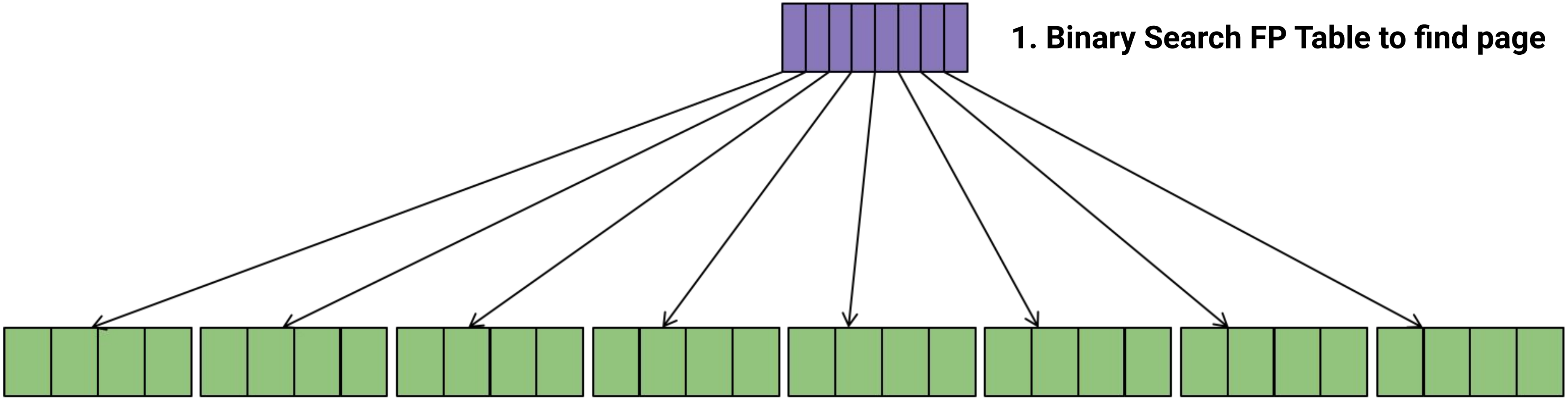
🟩 Page of actual data



Improving on Fence Pointers

🟪 Fence pointer array (in memory)

🟩 Page of actual data

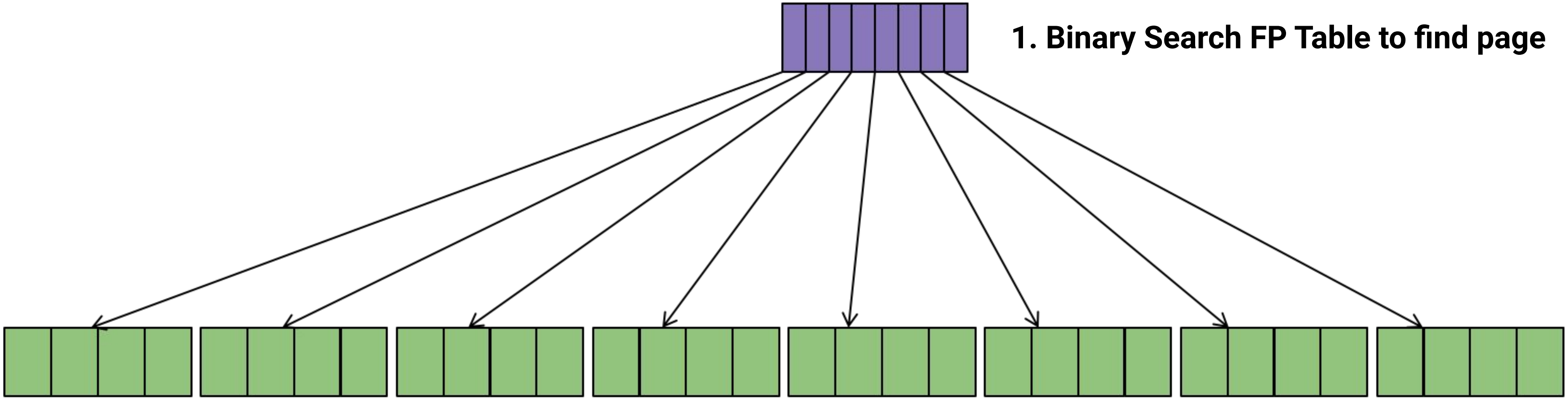


1. Binary Search FP Table to find page

Improving on Fence Pointers

🟪 Fence pointer array (in memory)

🟩 Page of actual data



1. Binary Search FP Table to find page

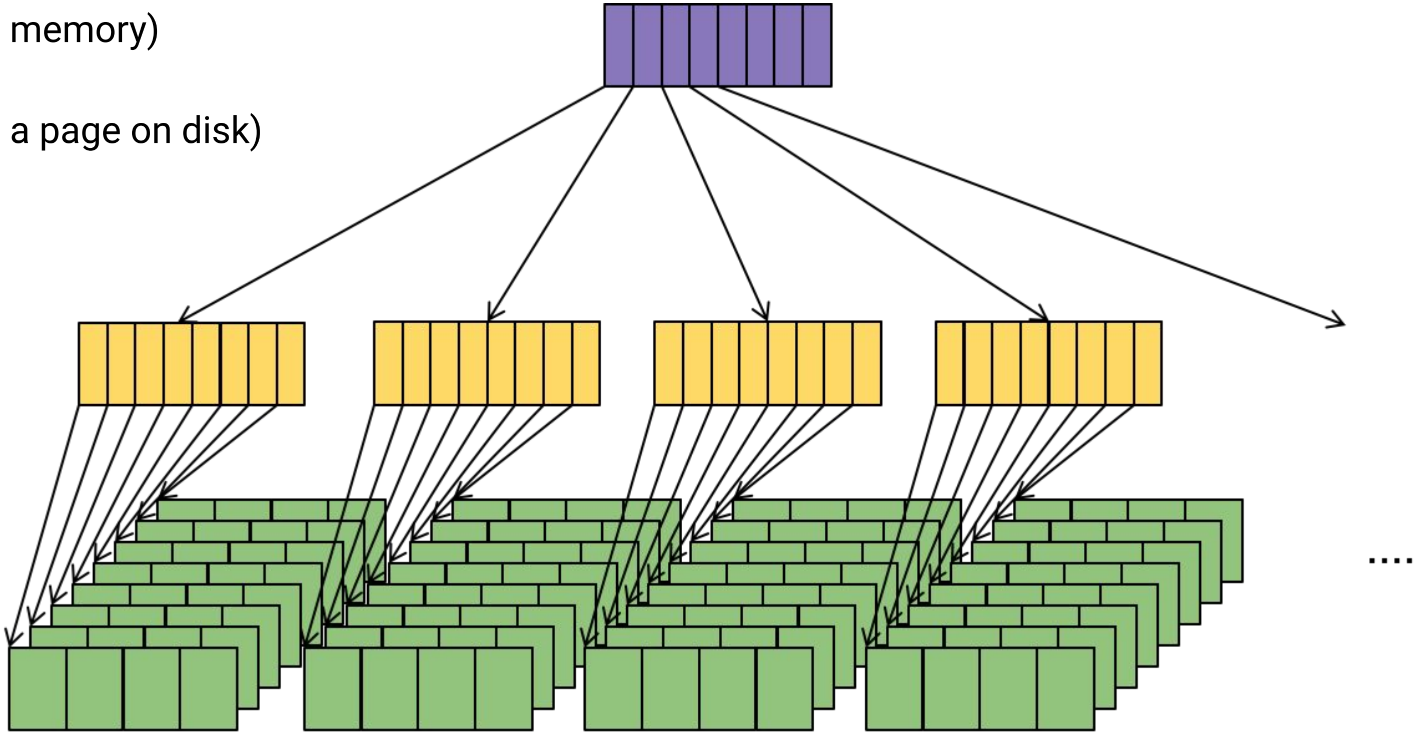
2. Load page and binary search for record

Improving on Fence Pointers

🟪 Fence pointer array (in memory)

🟨 Fence pointer array (in a page on disk)

🟩 Page of actual data

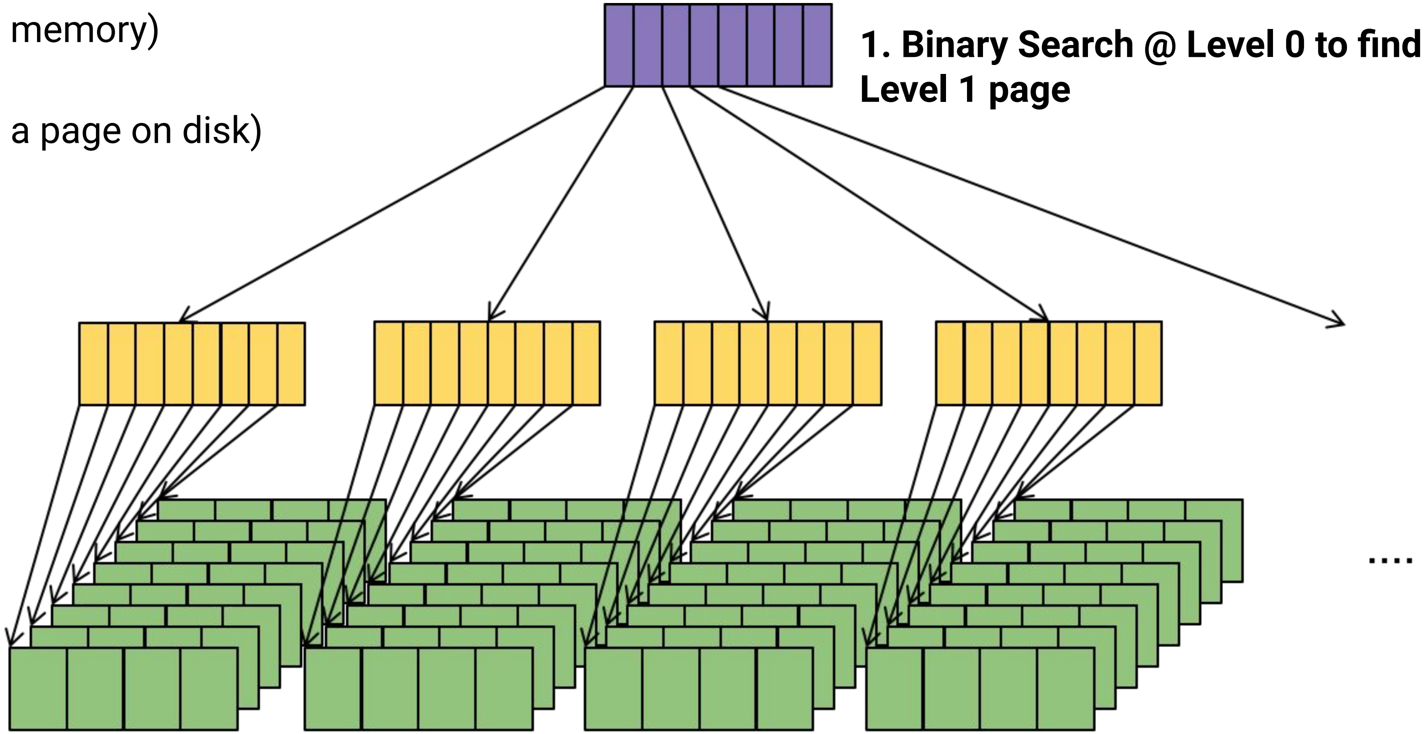


Improving on Fence Pointers

🟪 Fence pointer array (in memory)

🟨 Fence pointer array (in a page on disk)

🟩 Page of actual data



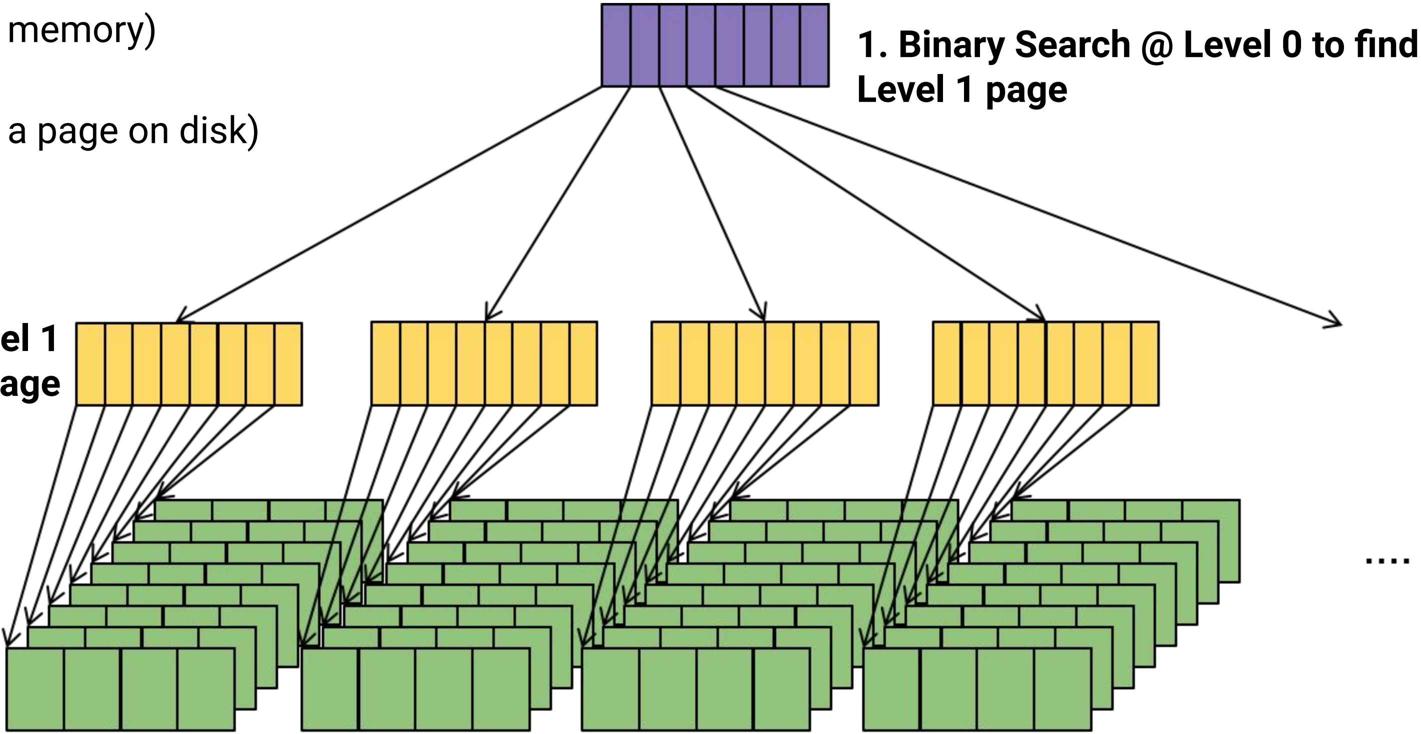
1. Binary Search @ Level 0 to find Level 1 page

Improving on Fence Pointers

🟪 Fence pointer array (in memory)

🟨 Fence pointer array (in a page on disk)

🟩 Page of actual data



1. Binary Search @ Level 0 to find Level 1 page

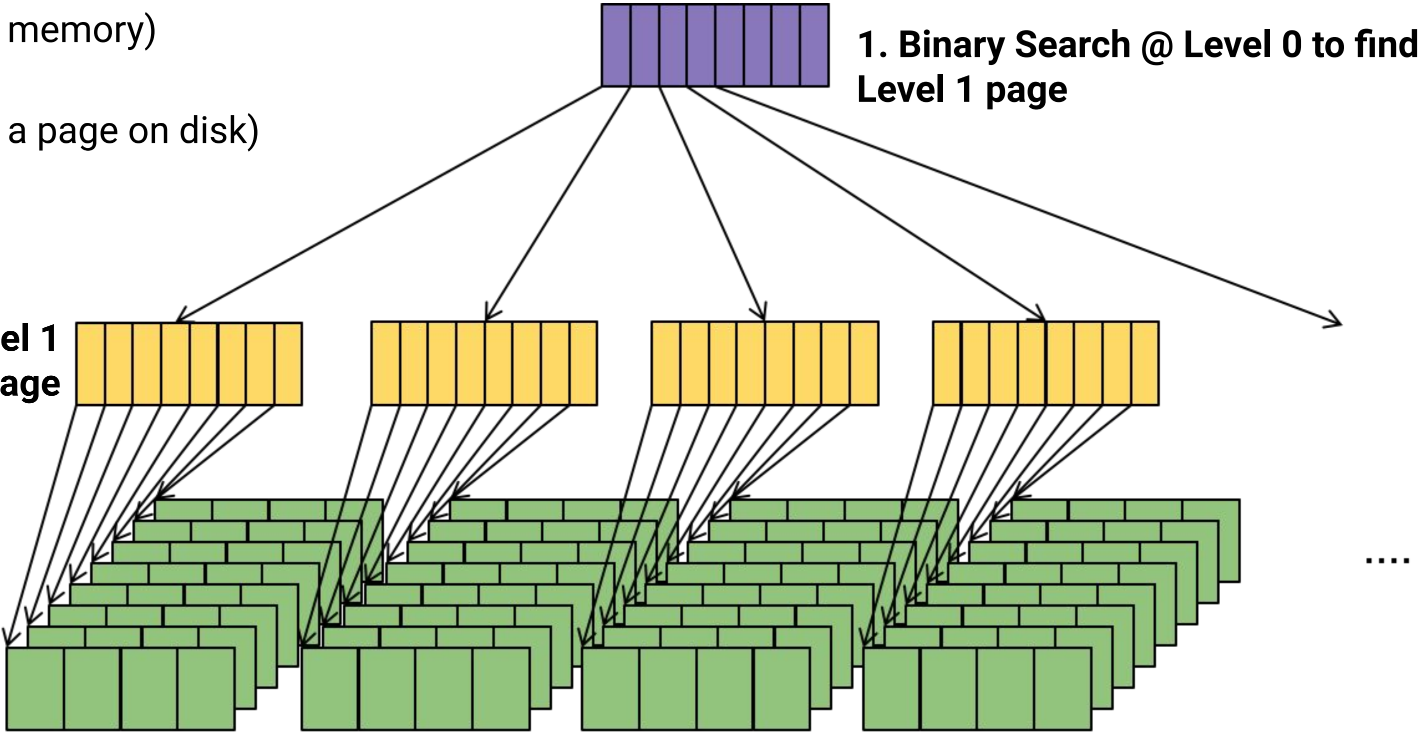
2. Load and search Level 1 page to find data page

Improving on Fence Pointers

🟪 Fence pointer array (in memory)

🟨 Fence pointer array (in a page on disk)

🟩 Page of actual data



1. **Binary Search @ Level 0 to find Level 1 page**

2. **Load and search Level 1 page to find data page**

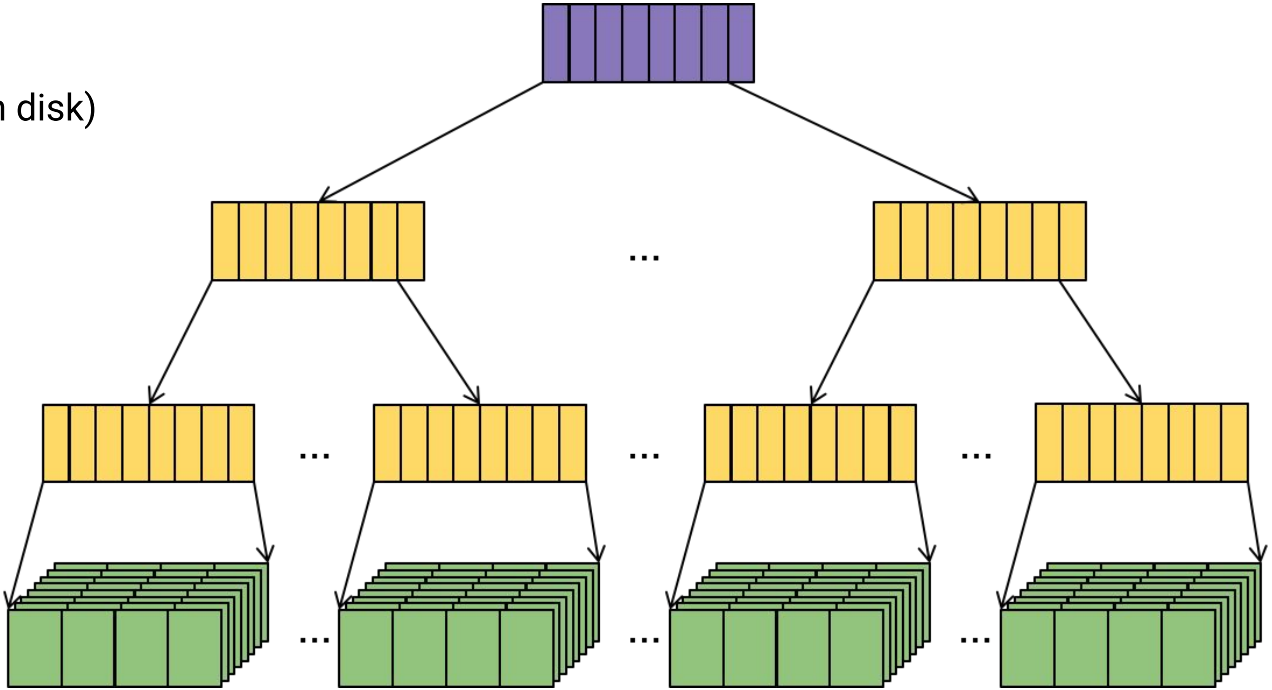
3. **Load and search data page for the record**

Improving on Fence Pointers

🟪 Fence pointer array (in memory)

🟨 Fence pointer array (in a page on disk)

🟩 Page of actual data

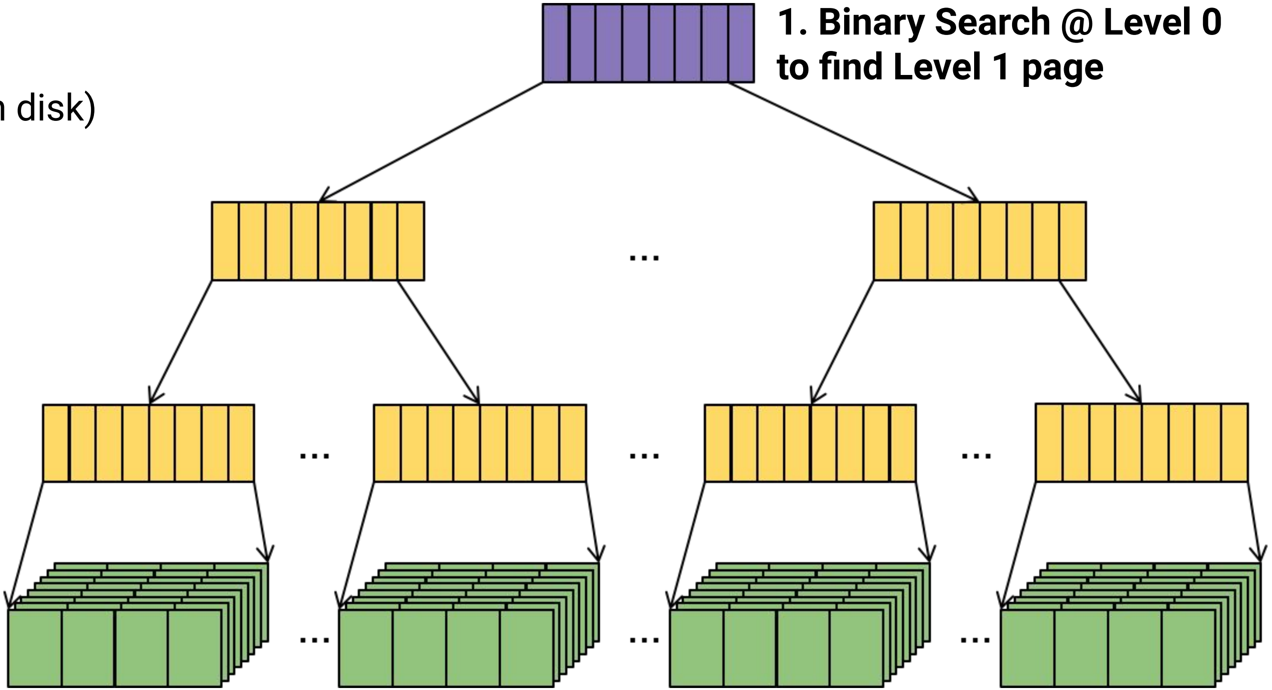


Improving on Fence Pointers

🟪 Fence pointer array (in memory)

🟨 Fence pointer array (in a page on disk)

🟩 Page of actual data



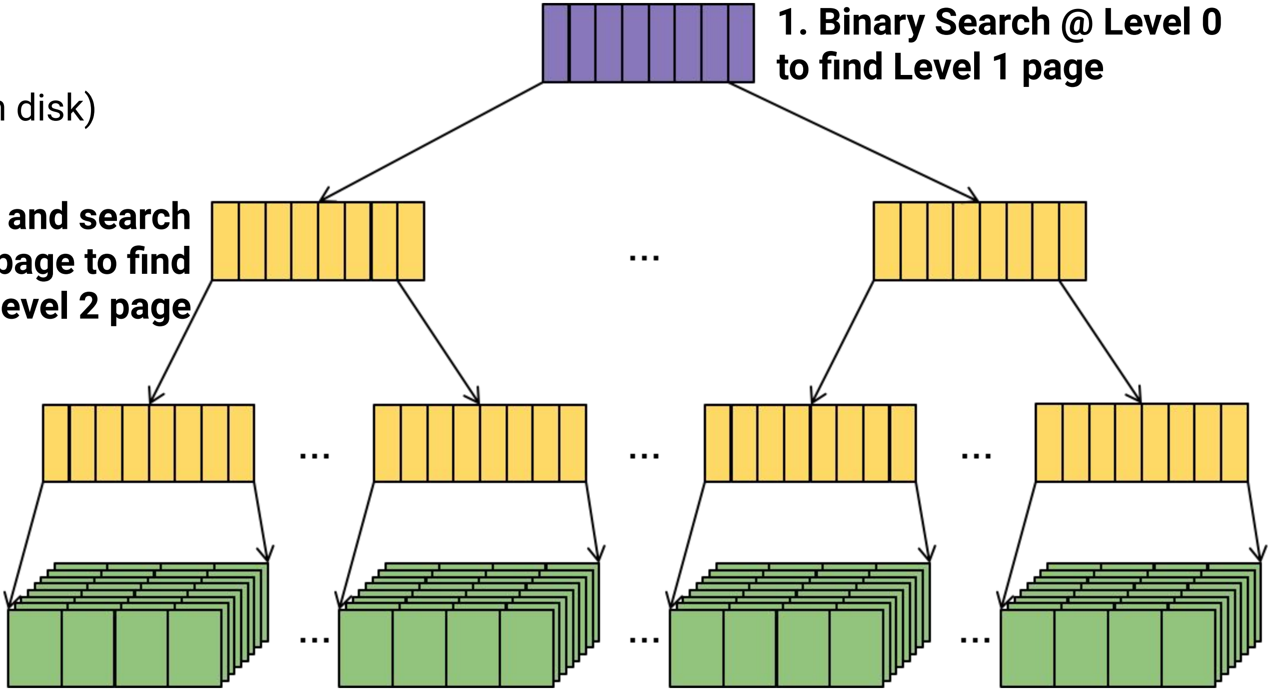
1. Binary Search @ Level 0 to find Level 1 page

Improving on Fence Pointers

🟪 Fence pointer array (in memory)

🟨 Fence pointer array (in a page on disk)

🟩 Page of actual data



1. Binary Search @ Level 0 to find Level 1 page

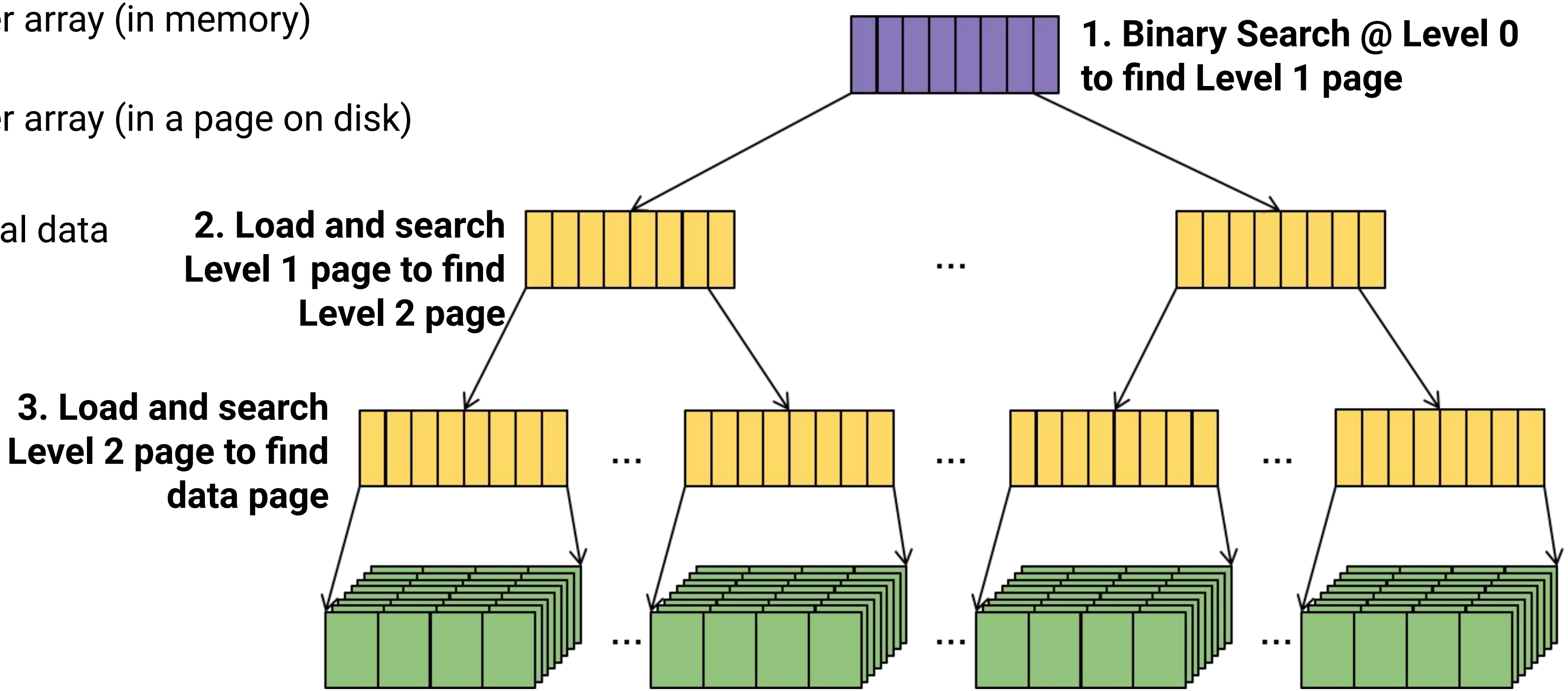
2. Load and search Level 1 page to find Level 2 page

Improving on Fence Pointers

🟪 Fence pointer array (in memory)

🟨 Fence pointer array (in a page on disk)

🟩 Page of actual data



1. Binary Search @ Level 0 to find Level 1 page

2. Load and search Level 1 page to find Level 2 page

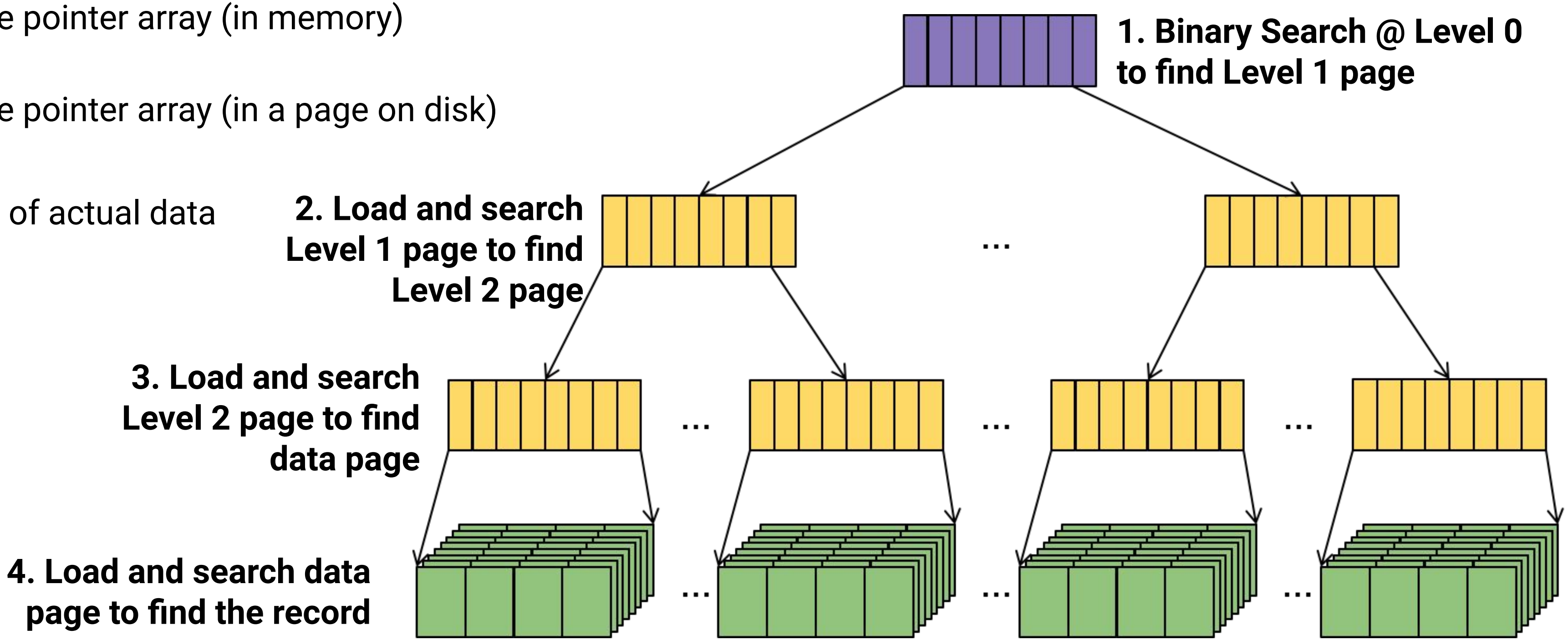
3. Load and search Level 2 page to find data page

Improving on Fence Pointers

🟪 Fence pointer array (in memory)

🟨 Fence pointer array (in a page on disk)

🟩 Page of actual data



1. Binary Search @ Level 0 to find Level 1 page

2. Load and search Level 1 page to find Level 2 page

3. Load and search Level 2 page to find data page

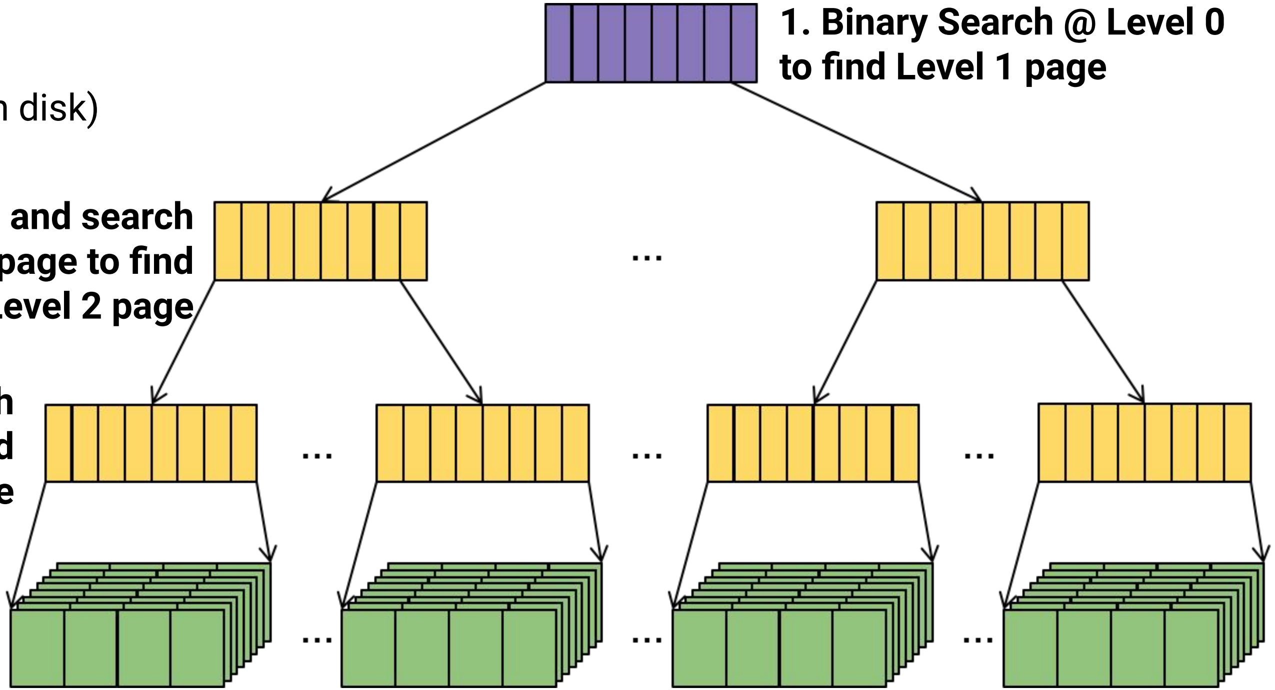
4. Load and search data page to find the record

~~Improving on Fence Pointers~~ ISAM Index

🟪 Fence pointer array (in memory)

🟨 Fence pointer array (in a page on disk)

🟩 Page of actual data



1. Binary Search @ Level 0 to find Level 1 page

2. Load and search Level 1 page to find Level 2 page

3. Load and search Level 2 page to find data page

4. Load and search data page to find the record

ISAM Index

IO Complexity:

- 1 read at L0 (or assume already in memory)
- 1 read at L1
- 1 read at L2
- …
- 1 read at L_{max}
- 1 read at data level

ISAM Index

How many levels will there be (this isn't a binary tree…)

ISAM Index

How many levels will there be (this isn't a binary tree…)

- Level 0: 1 page w/$\boldsymbol{C_{key}}$ keys

ISAM Index

How many levels will there be (this isn't a binary tree…)

- Level 0: 1 page w/$\boldsymbol{C_{key}}$ keys
- Level 1: Up to $\boldsymbol{C_{key}}$ pages w/$\boldsymbol{C_{key}^{2}}$ keys

ISAM Index

How many levels will there be (this isn't a binary tree…)

- Level 0: 1 page w/C_{key} keys
- Level 1: Up to C_{key} pages w/C_{key}^2 keys
- Level 2: Up to C_{key}^2 pages w/C_{key}^3 keys
- …

ISAM Index

How many levels will there be (this isn't a binary tree…)

- Level 0: 1 page w/C_{key} keys
- Level 1: Up to C_{key} pages w/C_{key}^{2} keys
- Level 2: Up to C_{key}^{2} pages w/C_{key}^{3} keys
- …
- Level max: Up to C_{key}^{max} pages w/C_{key}^{max+1} keys

ISAM Index

How many levels will there be (this isn't a binary tree…)

- Level 0: 1 page w/C_{key} keys
- Level 1: Up to C_{key} pages w/C_{key}^{2} keys
- Level 2: Up to C_{key}^{2} pages w/C_{key}^{3} keys
- …
- Level max: Up to C_{key}^{max} pages w/C_{key}^{max+1} keys
- Data Level: Up to C_{key}^{max+1} pages w/$C_{data}C_{key}^{max+1}$ records

ISAM Index

$$n = C_{data} C_{key}^{max+1}$$

ISAM Index

$$n = C_{data} C_{key}^{max+1}$$

$$\frac{n}{C_{data}} = C_{key}^{max+1}$$

ISAM Index

$$n = C_{data} C_{key}^{max+1}$$

$$\frac{n}{C_{data}} = C_{key}^{max+1}$$

$$\log_{C_{key}} \left(\frac{n}{C_{data}} \right) = max + 1$$

ISAM Index

$$n = C_{data} C_{key}^{max+1}$$

$$\frac{n}{C_{data}} = C_{key}^{max+1}$$

$$\log_{C_{key}} \left(\frac{n}{C_{data}} \right) = max + 1$$

$$\log_{C_{key}}(n) - \log_{C_{key}}(C_{data}) = max + 1$$

ISAM Index

$$n = C_{data} C_{key}^{max+1}$$

$$\frac{n}{C_{data}} = C_{key}^{max+1}$$

$$\log_{C_{key}}\left(\frac{n}{C_{data}}\right) = max + 1$$

$$\log_{C_{key}}(n) - \log_{C_{key}}(C_{data}) = max + 1$$

Number of Levels: $O\left(\log_{C_{key}}(n)\right)$

ISAM Index

$$n = C_{data} C_{key}^{max+1}$$

$$\frac{n}{C_{data}} = C_{key}^{max+1}$$

$$\log_{C_{key}}\left(\frac{n}{C_{data}}\right) = max + 1$$

$$\log_{C_{key}}(n) - \log_{C_{key}}(C_{data}) = max + 1$$

Note this isn't base 2!

Number of Levels: $O\left(\boxed{\log_{C_{key}}}(n)\right)$

ISAM Index

Like Binary Search, but "Cache-Friendly"

- Still takes $O(\log(n))$ steps
- Still requires $O(1)$ memory (1 page at a time)
- Now requires $\log_{Ckey}(n)$ loads from disk ($\log_{Ckey}(n) \ll \log_2(n)$)

ISAM Index

What if the data changes?