CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu

Dr. Oliver Kennedy
okennedy@buffalo.edu

212 Capen Hall

Day 33
ISAM Indexes

Recap

BinarySearch requires $O(\log(n))$ steps…but this is not the whole picture!

Recap

BinarySearch requires $\boldsymbol{O}(\mathbf{log}(\boldsymbol{n}))$ steps…but this is not the whole picture!

- **Runtime Complexity:** $\boldsymbol{O}(\mathbf{log}(\boldsymbol{n}))$ steps required

Recap

BinarySearch requires $O(\log(n))$ steps…but this is not the whole picture!

- **Runtime Complexity:** $O(\log(n))$ steps required
- **Memory Complexity:** $O(1)$ memory required
    - We only ever need one page loaded at a time

Recap

BinarySearch requires ***O*(log(*n*))** steps…but this is not the whole picture!

- **Runtime Complexity: *O*(log(*n*))** steps required
- **Memory Complexity: *O*(1)** memory required
  - We only ever need one page loaded at a time
- **IO Complexity: *O*(log(*n*))** pages loaded
  - If a page can hold ***C*** records, the last **log(*C*)** search steps occur within that one page
  - But the first ***O*(log(n)-log(*C*)) = *O*(log(*n*))** steps each load a new page

Recap

BinarySearch requires ***O*(log(*n*))** steps…but this is not the whole picture!

- **Runtime Complexity: *O*(log(*n*))** steps required
- **Memory Complexity: *O*(1)** memory required
  - We only ever need one page loaded at a time
- **IO Complexity: *O*(log(*n*))** pages loaded
  - If a page can hold ***C*** records, the last **log(*C*)** search steps occur within that one page
  - But the first ***O*(log(n)-log(*C*)) = *O*(log(*n*))** steps each load a new page

How can we do better?

Solution

Trivial Solution:

- Load the entire array into memory
    - Load it once, and then reuse that memory for all searches

Solution

Trivial Solution:

- Load the entire array into memory
  - Load it once, and then reuse that memory for all searches

Problem: What if the array is too big to fit in memory?

Solution

Trivial Solution:

- Load the entire array into memory
  - Load it once, and then reuse that memory for all searches

Problem: What if the array is too big to fit in memory?

Question: Do we need to preload the entire array to avoid page loads?

Improving Binary Search

Observation 1: The records are much bigger than the search keys

Improving Binary Search

Observation 1: The records are much bigger than the search keys

- 64MB required to store 2^{20} 64B records
- 8MB required to store 2^{20} 8B keys

Improving Binary Search

Observation 1: The records are much bigger than the search keys

- 64MB required to store 2^{20} 64B records
- 8MB required to store 2^{20} 8B keys

Observation 2: Pages store contiguous ranges of keys

Improving Binary Search

Observation 1: The records are much bigger than the search keys

- 64MB required to store 2^{20} 64B records
- 8MB required to store 2^{20} 8B keys

Observation 2: Pages store contiguous ranges of keys

- If we know what range of keys a page stores, we don't need to load pages that don't contain the key we are looking for

Fence Pointers

Idea: Store the largest key of each page in an in-memory data structure

Fence Pointers

Idea: Store the largest key of each page in an in-memory data structure

- Precompute this (hopefully smaller) data structure
- Re-use this in-memory data structure for all searches to find the page that stores the search key
    - Only load that one page, instead of one page per step of the search

Fence Pointers Example

Let's say our records are 64B, keys are 8B, our pages can hold 64 records, and $n=2^{20}$ records:

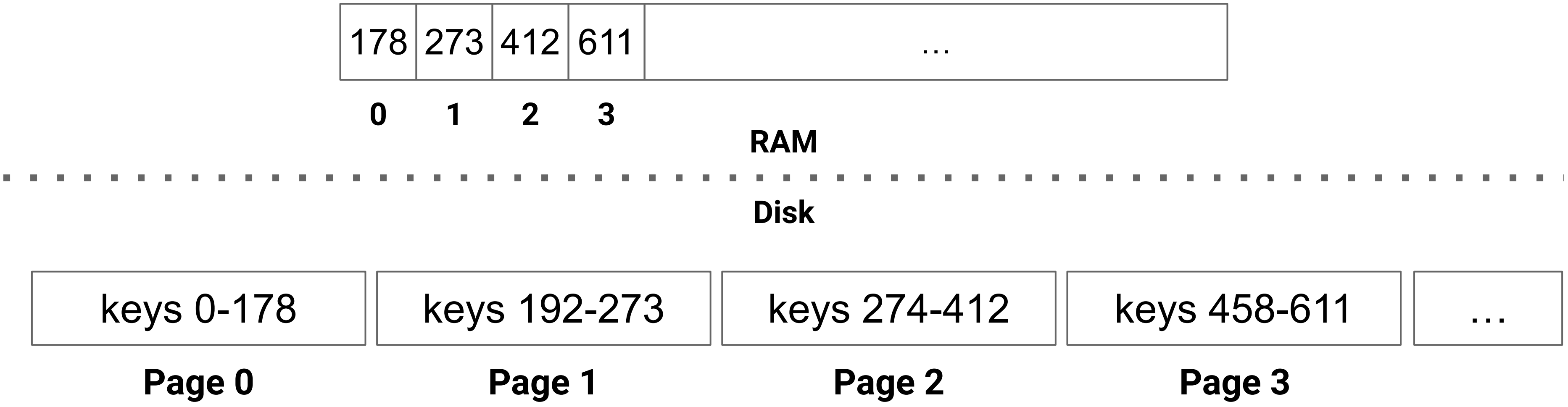
- 2^{20} 64B records = **64MB**
- 2^{20} records / 64 = 2^{14} pages
- 2^{14} 8B keys = **512KB** ← Store these keys in a "Fence Pointer Table"

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

Disk: 16,384 pages, 64MB total (the actual data)

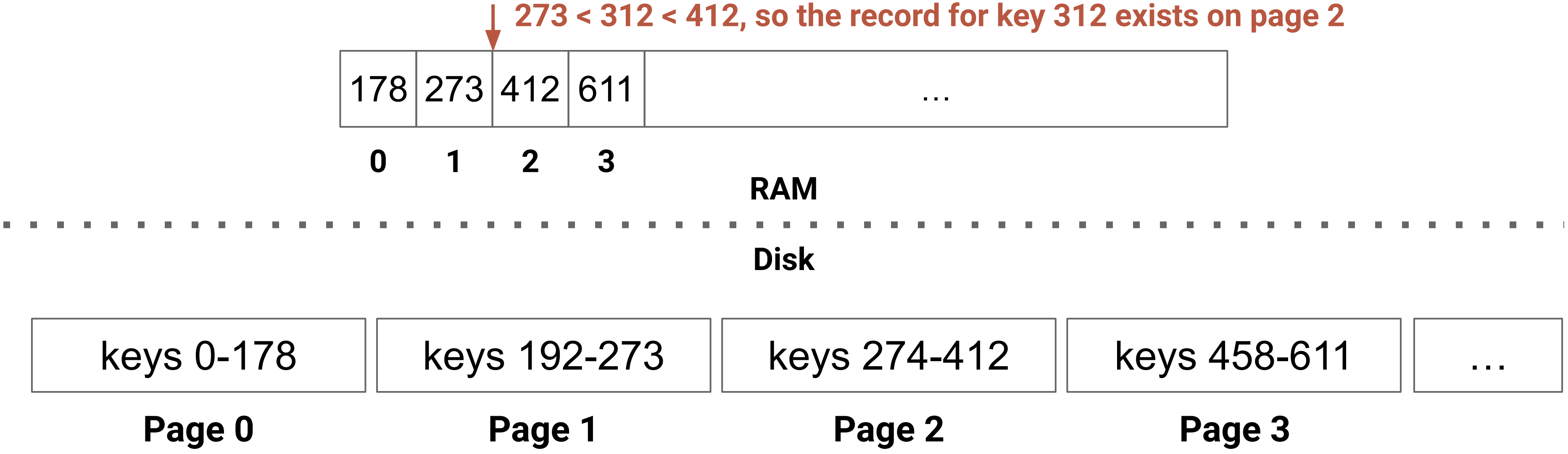
Fence Pointers Example

To find a record with key 312, for example, we binary search the fence pointer table first to find the page. Then search that page for the record.



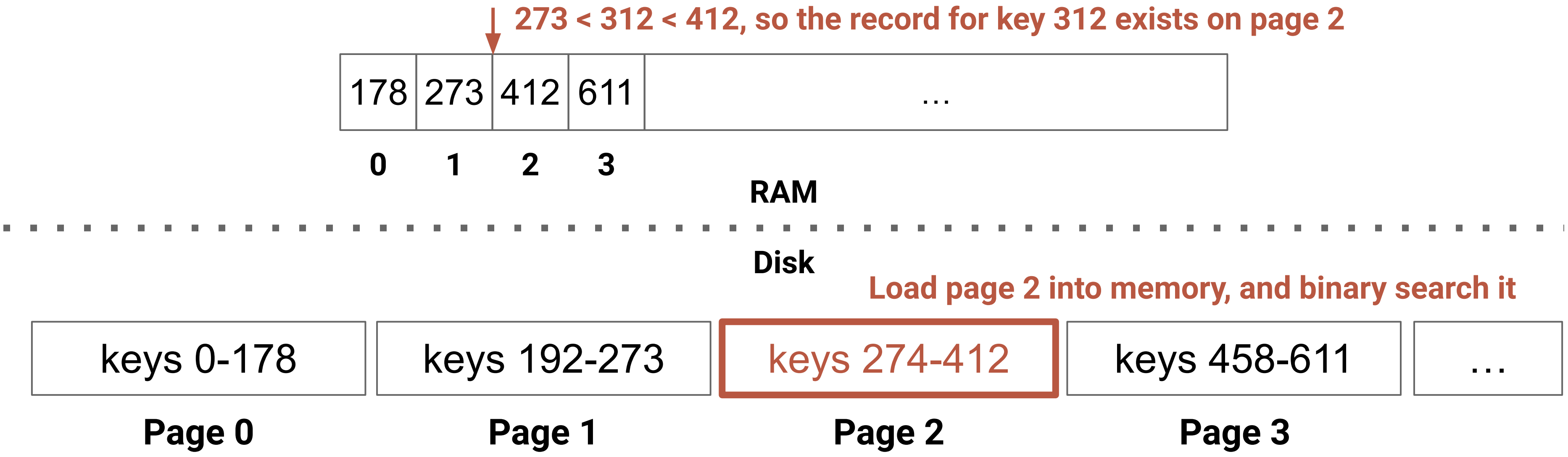
Fence Pointers Example

To find a record with key 312, for example, we binary search the fence pointer table first to find the page. Then search that page for the record.



Fence Pointers Example

To find a record with key 312, for example, we binary search the fence pointer table first to find the page. Then search that page for the record.



Binary Search with Fence Pointers

Step 1: Binary search the fence pointer table

- All in memory, so IO complexity is 0

Binary Search with Fence Pointers

Step 1: Binary search the fence pointer table

- All in memory, so IO complexity is 0

Step 2: Load page

- One load, so IO complexity is $\boldsymbol{O}(1)$

Binary Search with Fence Pointers

Step 1: Binary search the fence pointer table

- All in memory, so IO complexity is 0

Step 2: Load page

- One load, so IO complexity is $\boldsymbol{O(1)}$

Step 3: Binary search within page

- All in memory, so IO complexity is 0

Binary Search with Fence Pointers

Step 1: Binary search the fence pointer table

- All in memory, so IO complexity is 0

Step 2: Load page

- One load, so IO complexity is ***O*(1)**

Step 3: Binary search within page

- All in memory, so IO complexity is 0

Totaly IO Complexity: *O*(1)

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

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

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

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

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 (2^2 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(records/page**) - log(**keys/page**)**

Improving on Fence Pointers

With our current example:

- We can store 512 8B keys per 4KB page (2^2 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(records/page) **- log**(keys/page) **= *O*(log(*n*))...**

Improving on Fence Pointers

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

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

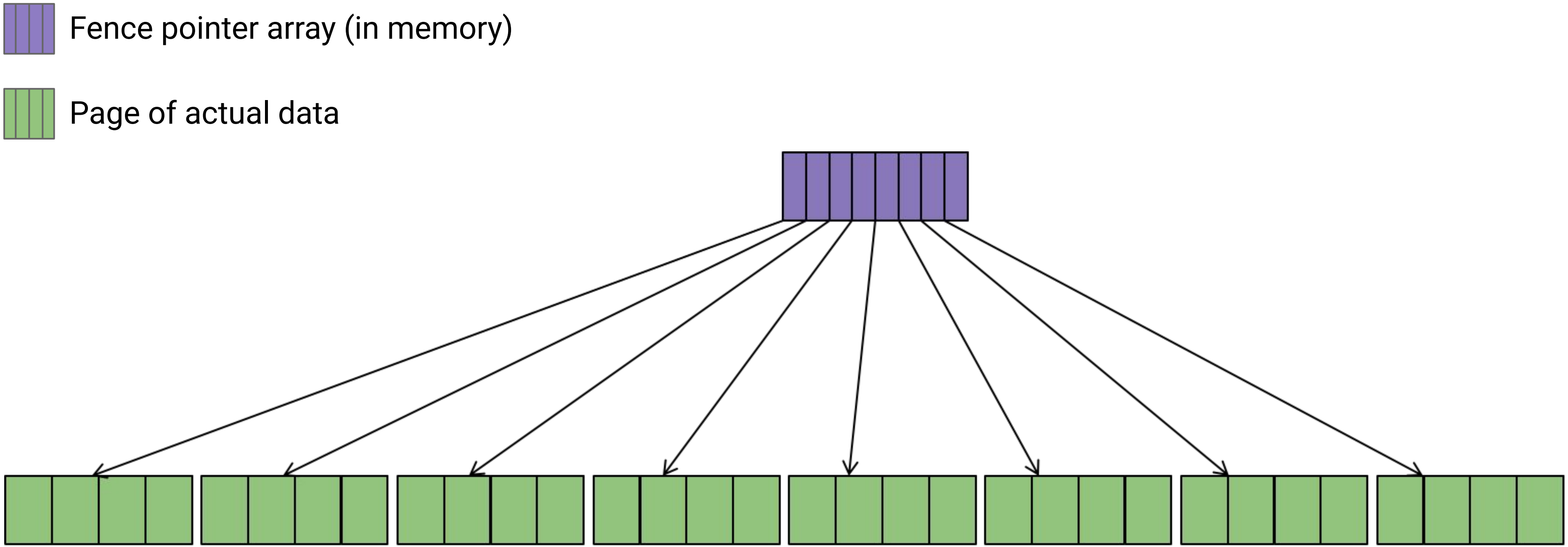
Can we improve our search of the on-disk Fence Pointer Table…?

Improving on Fence Pointers

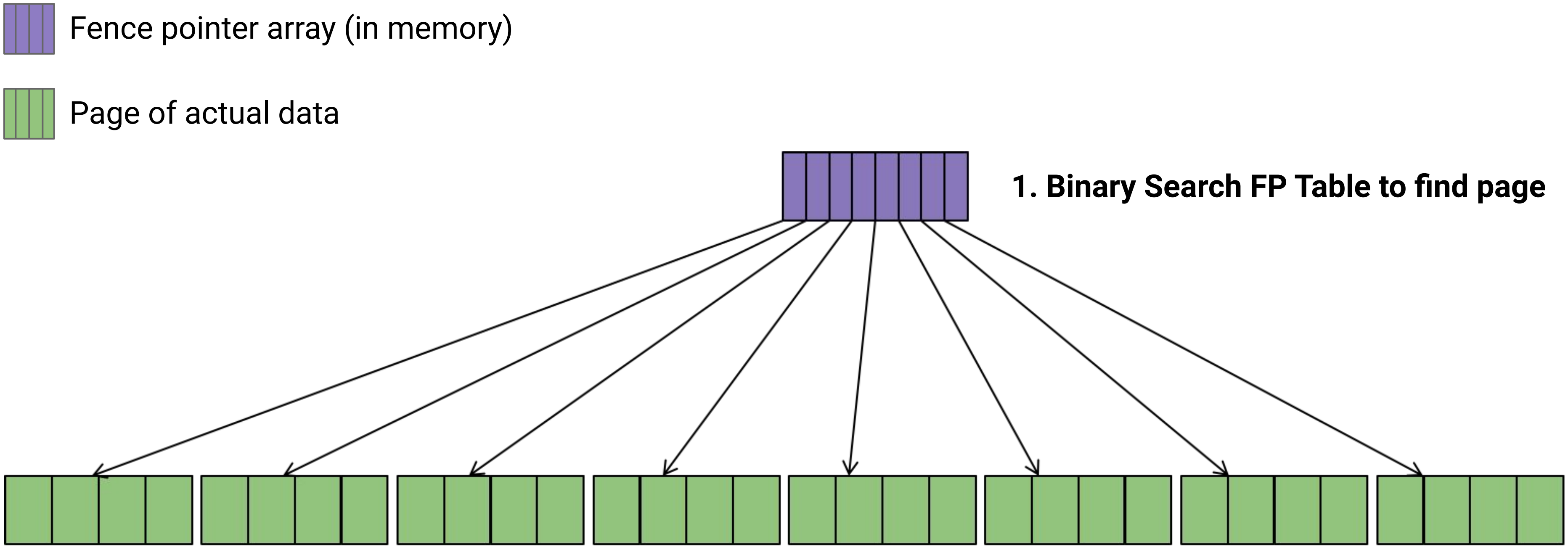
Idea: A fence pointer table for our fence pointer table!

(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)

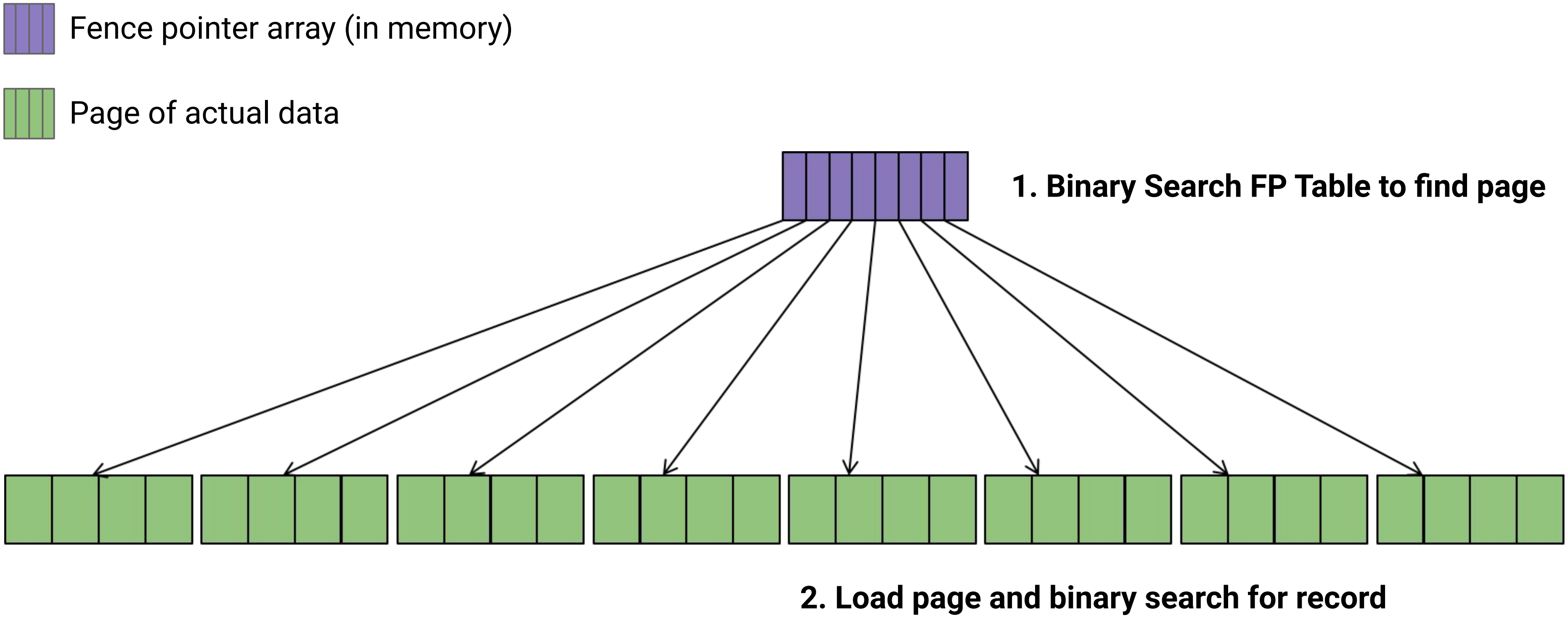
Improving on Fence Pointers



Improving on Fence Pointers

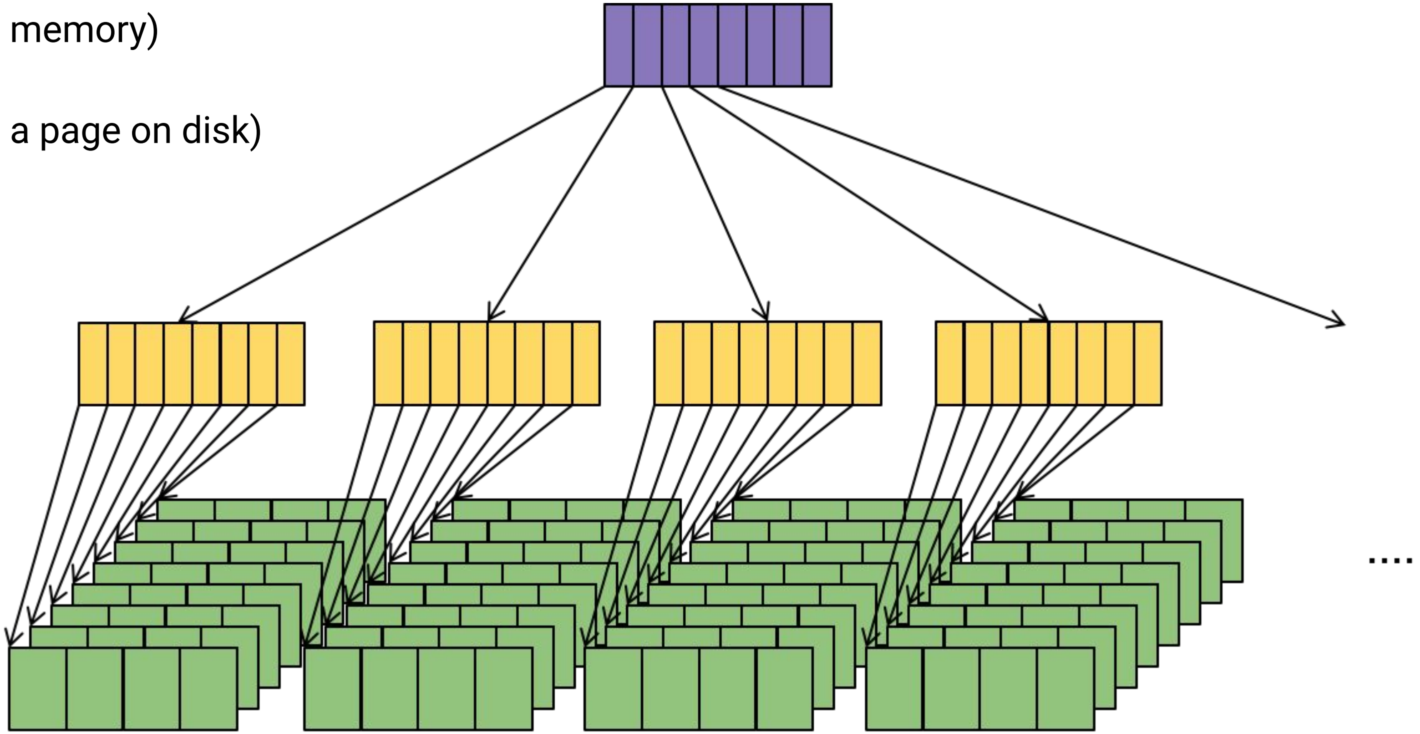


Improving on Fence Pointers

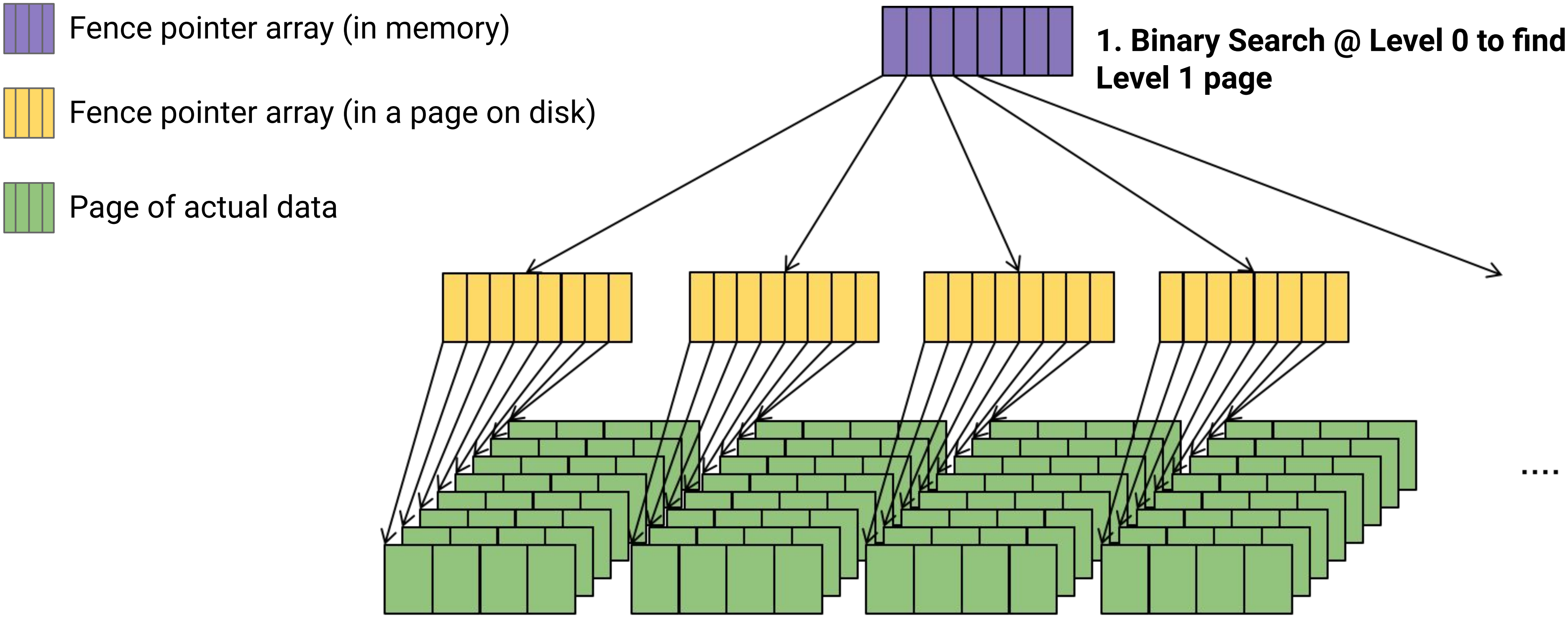


Improving on Fence Pointers

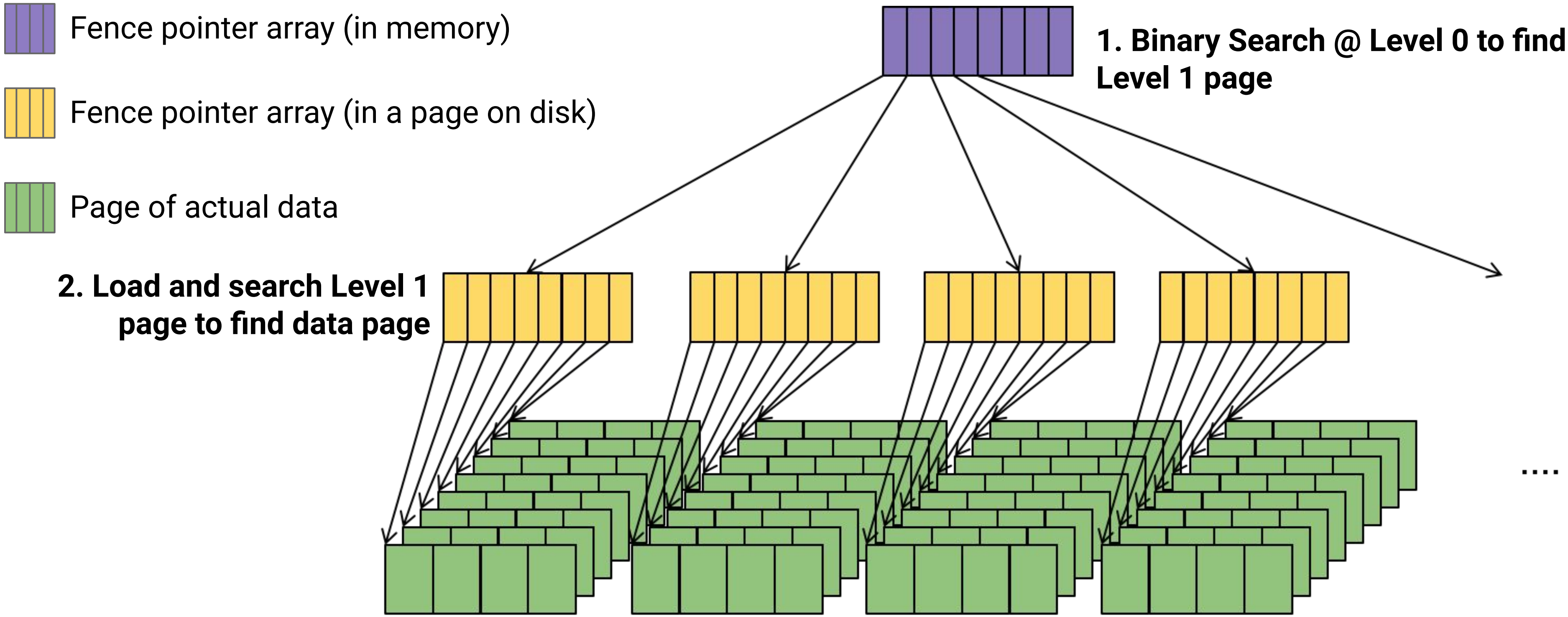
- Fence pointer array (in memory)
- Fence pointer array (in a page on disk)
- Page of actual data



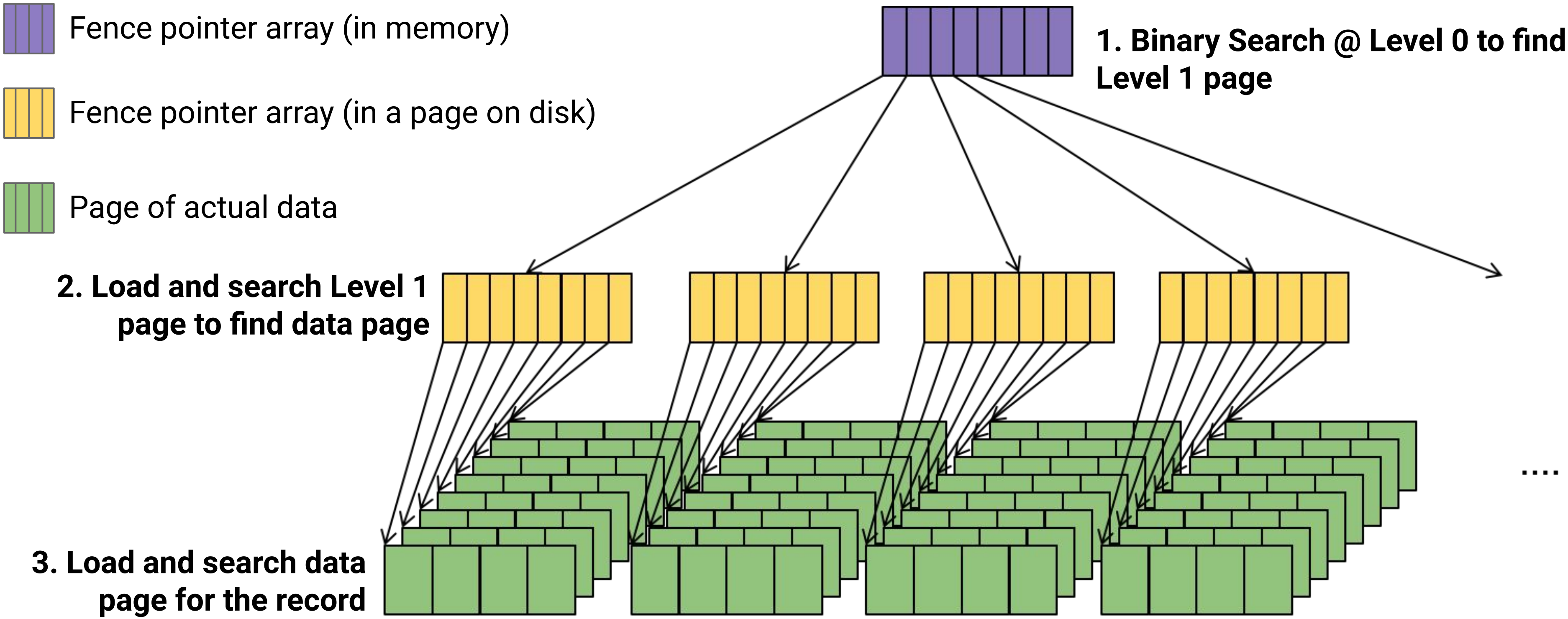
Improving on Fence Pointers



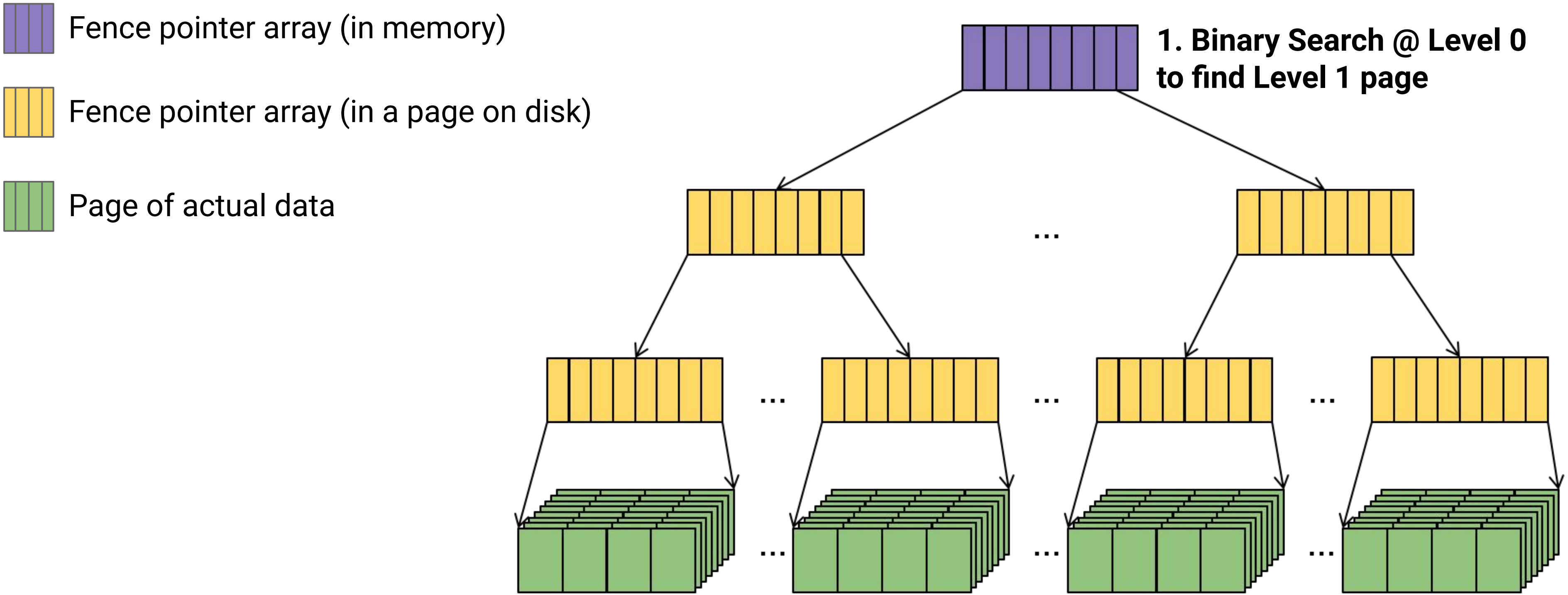
Improving on Fence Pointers



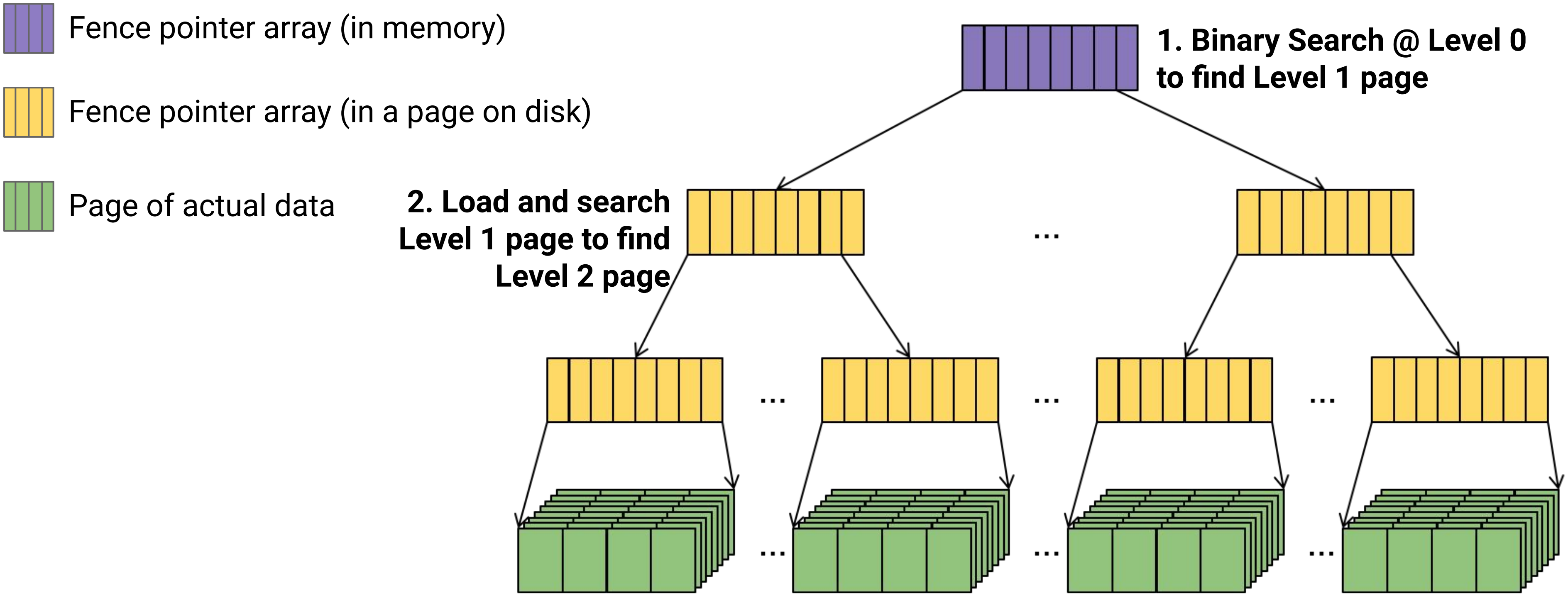
Improving on Fence Pointers



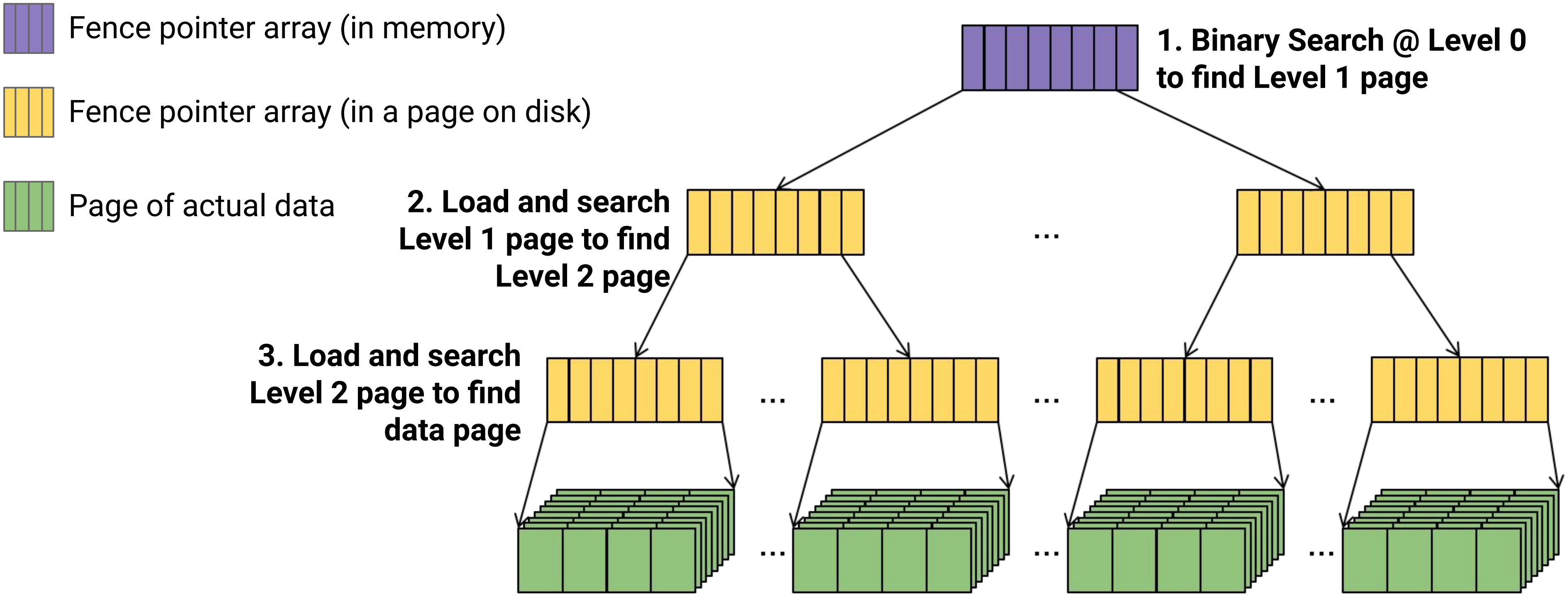
Improving on Fence Pointers



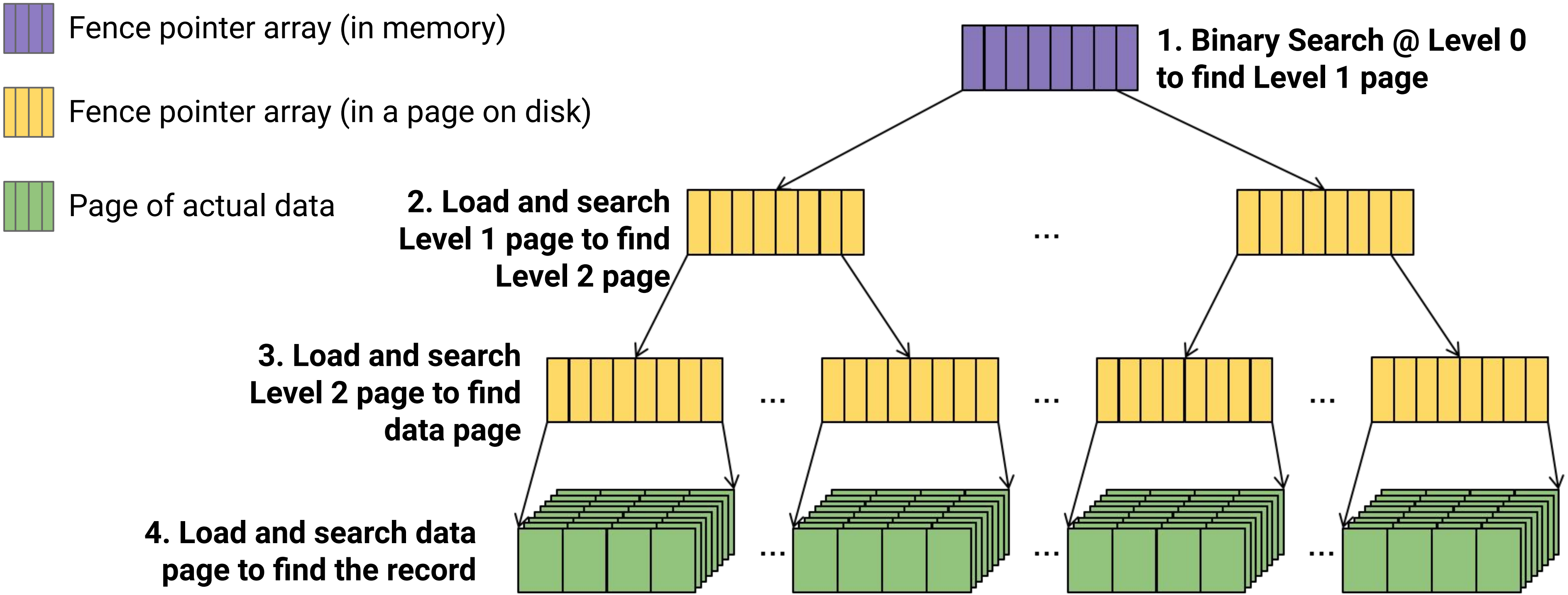
Improving on Fence Pointers



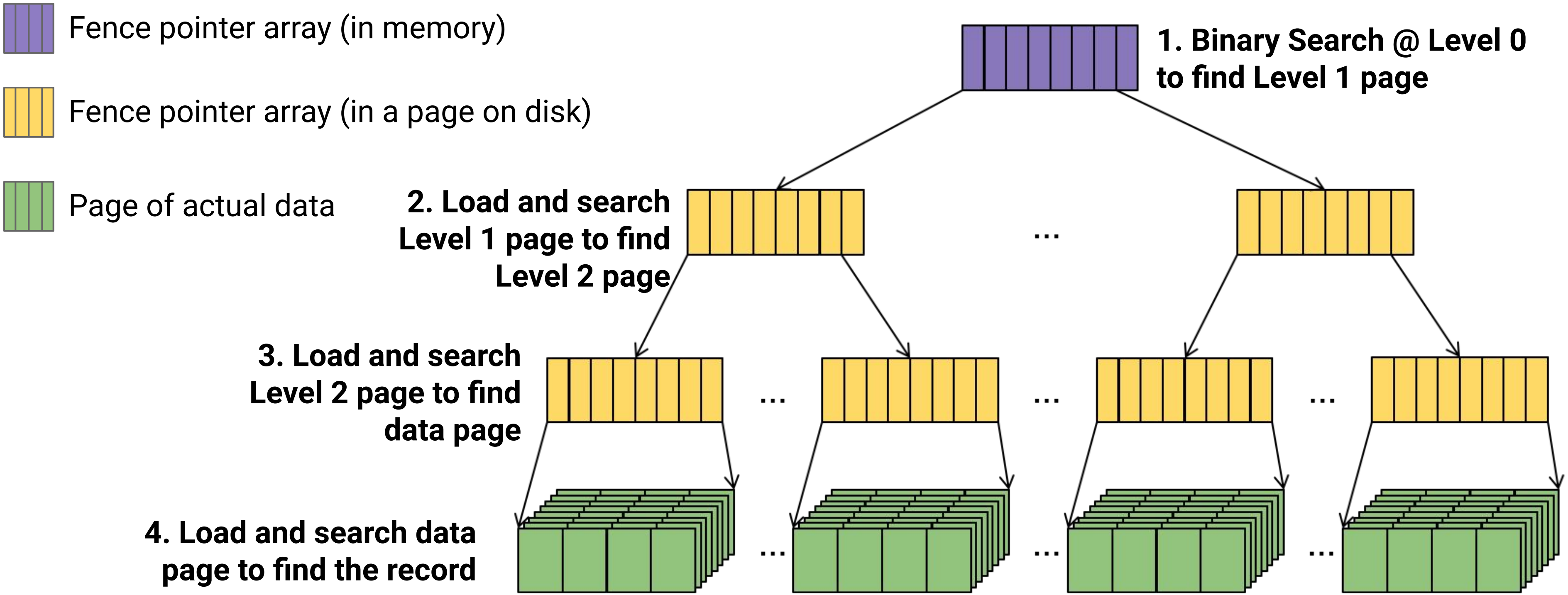
Improving on Fence Pointers



Improving on Fence Pointers



~~Improving on Fence Pointers~~ ISAM Index



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/$\boldsymbol{C_{key}}$ keys
- Level 1: Up to $\boldsymbol{C_{key}}$ pages w/$\boldsymbol{C_{key}^{2}}$ keys
- Level 2: Up to $\boldsymbol{C_{key}^{2}}$ pages w/$\boldsymbol{C_{key}^{3}}$ 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
- Level 2: Up to $\boldsymbol{C_{key}^{2}}$ pages w/ $\boldsymbol{C_{key}^{3}}$ keys
- …
- Level max: Up to $\boldsymbol{C_{key}^{max}}$ pages w/ $\boldsymbol{C_{key}^{max+1}}$ 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: $O\left(\log_{C_{key}}(n)\right)$

ISAM Index

Like BinarySearch, but "Cache-Friendly"

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

ISAM Index

What if the data changes?