Indexing

CIT5940

Program Memory vs. Disk Space

You’re the CTO of a new tech company *Feebo*. Congrats on your 3M new users and all of the hundreds of photos each of them like to share!

- Software runs on machines with RAM—**program memory**—that is fast but comes in short supply.
- Data is stored in centers with thousands of hard drives—**“disks”**—that are cheap but take a while to read from & write to.

The success of *Feebo* relies on your ability to write software that accesses huge amount of information on disk, but quickly! Every millisecond spent waiting for an app view to load is another few users dropped...

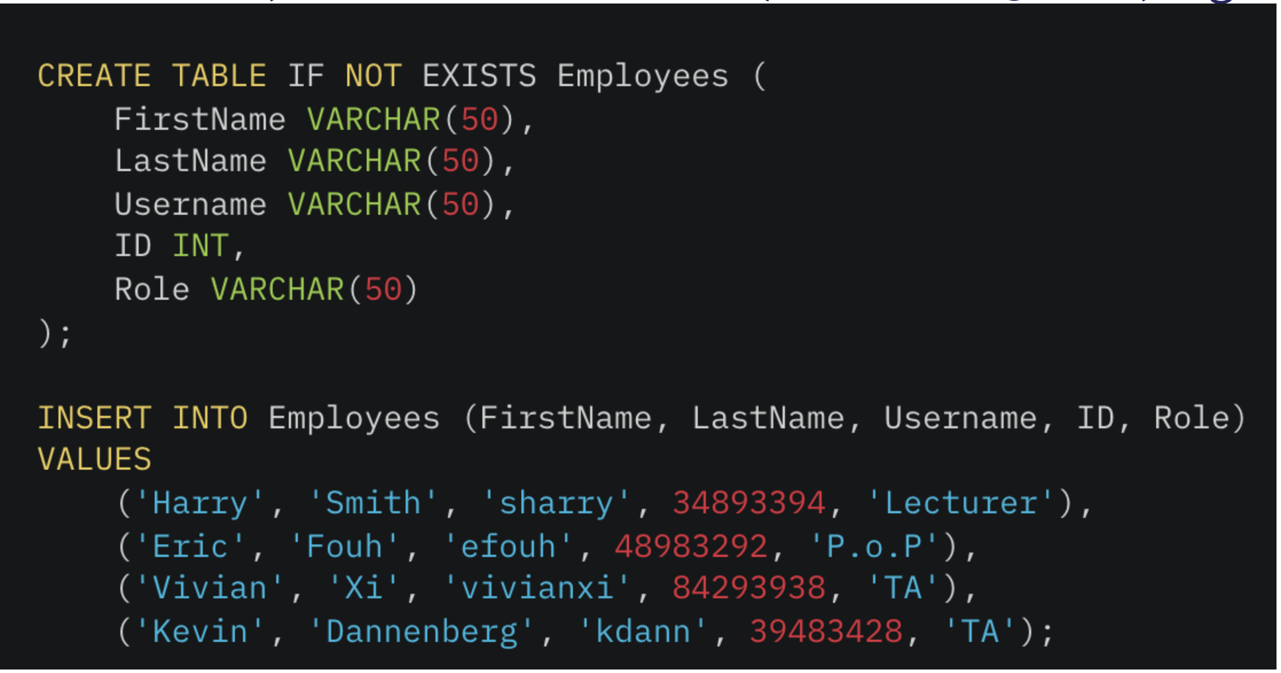
Introduction

- You'll often work with a dataset that doesn't entirely fit into your program's memory
- **Indexing**: the process of associating a *search key* with the location (on disk) of a corresponding data record
    - Think of the index in a textbook: given a topic name, it tells you where to go find more information about that topic.
- Remember: program memory is fast but expensive, whereas disk space is slow and cheap.
    - Do as little seeking on disk as possible with **carefully planned indices.**

Index vs. Database

An **index** is used to enable fast queries over a database. A **database** is just some collection of data records (like a data structure), which could take the form of:

- An array, a tree, or a hash system of records
- A RDB (a “classical” database) or a NOSQL DB (a “graph based” database)



```sql
CREATE TABLE IF NOT EXISTS Employees (
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Username VARCHAR(50),
    ID INT,
    Role VARCHAR(50)
);

INSERT INTO Employees (FirstName, LastName, Username, ID, Role)
VALUES
    ('Harry', 'Smith', 'sharry', 34893394, 'Lecturer'),
    ('Eric', 'Fouh', 'efouh', 48983292, 'P.o.P'),
    ('Vivian', 'Xi', 'vivianxi', 84293938, 'TA'),
    ('Kevin', 'Dannenberg', 'kdann', 39483428, 'TA');
```

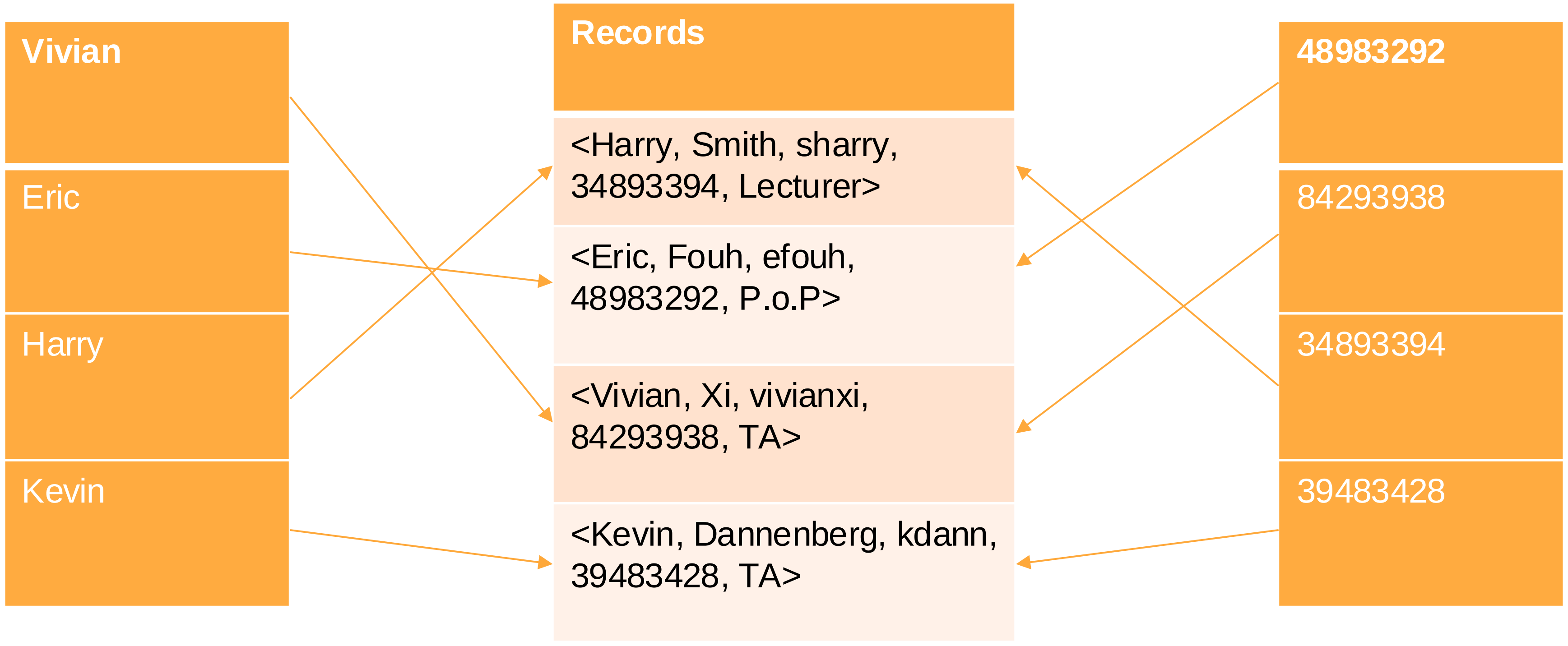
Creating a relational database in SQL

Records
<Harry, Smith, sharry, 34893394, Lecturer>
<Eric, Fouh, efouh, 48983292, P.o.P>
<Vivian, Xi, vivianxi, 84293938, TA>
<Kevin, Dannenberg, kdann, 39483428, TA>

An array-like view of the records stored.

Index

- The index **does not store the record**
- The index **stores a *reference* to the record**
- A collection of records can be supported by **multiple indices**


Vivian
Eric
Harry
Kevin
Records
<Harry, Smith, sharry, 34893394, Lecturer>
<Eric, Fouh, efouh, 48983292, P.o.P>
<Vivian, Xi, vivianxi, 84293938, TA>
<Kevin, Dannenberg, kdann, 39483428, TA>
48983292
84293938
34893394
39483428

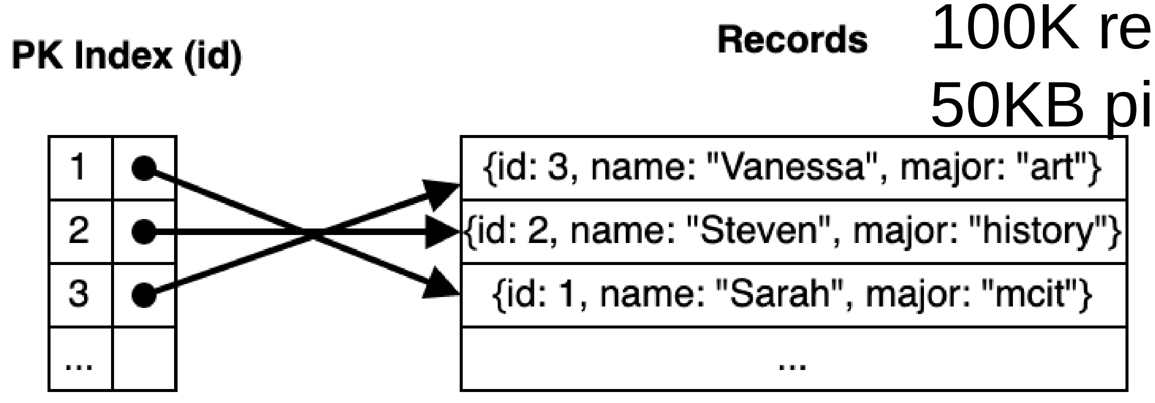
How do the two different indices here let you query the database?

How to Build an Index: Primary Key

- Each record of a database normally has a unique identifier (field/attribute)
- **Primary key:** an attribute that uniquely identifies a record
    - ID number, Penn ID, Social Security Number, etc.
    - Often a number or a short string, which can be expressed using far fewer bits compared to the full record.
- First idea: create a table of primary keys to search through

Primary Key Index

- Associates each *primary key* value with a pointer to the actual record on disk


PK Index (id)
Records
1
2
3
...
{id: 3, name: "Vanessa", major: "art"}
{id: 2, name: "Steven", major: "history"}
{id: 1, name: "Sarah", major: "mcit"}
...

100K records with a 50KB picture each...

~1MB

~1GB

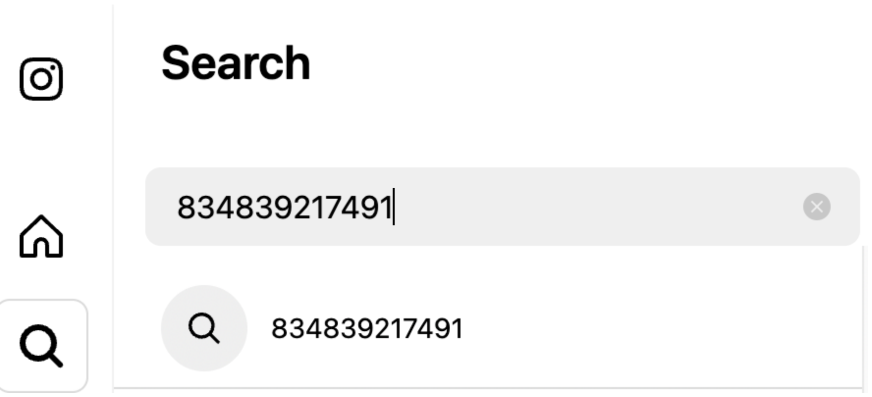
*If I have **n** records and a PK index storing the keys in a sorted array, what runtime can I expect when looking up a record by its primary key?*

Primary Key: Caveats

- Primary key often not known by the user of the database
- Primary key often not useful when searching for a record.
- Database searches often performed using attributes other than the primary key (name, age, major, salary, etc.)

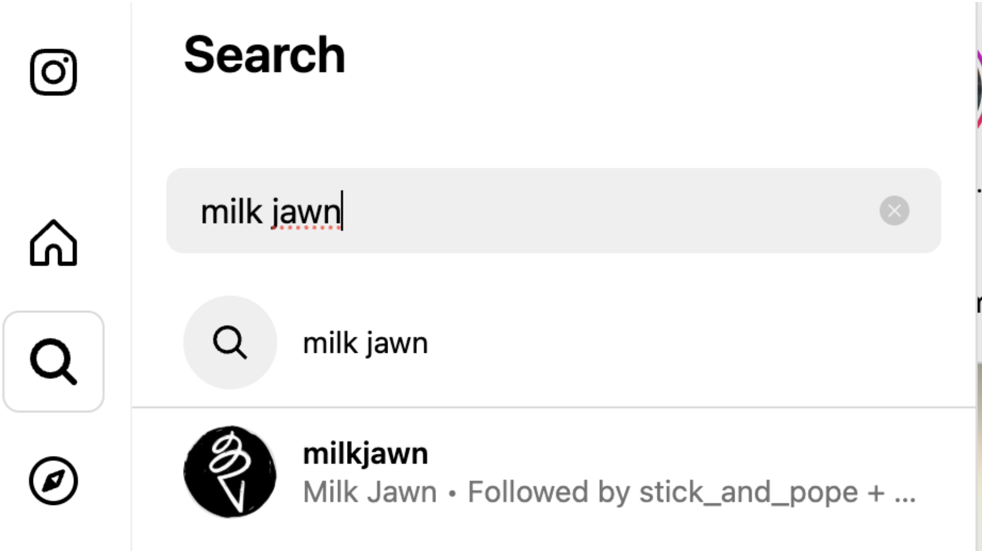
How to Build an Index: Secondary Keys

- **Secondary key:** a key field in a record where a particular key value might be duplicated in multiple records
  - such as salary, name, major, etc.
- Secondary key is more likely to be used by a user as a search key than is the record's primary key
  - Can't be used to uniquely identify a record, though


Search

834839217491

834839217491


Search

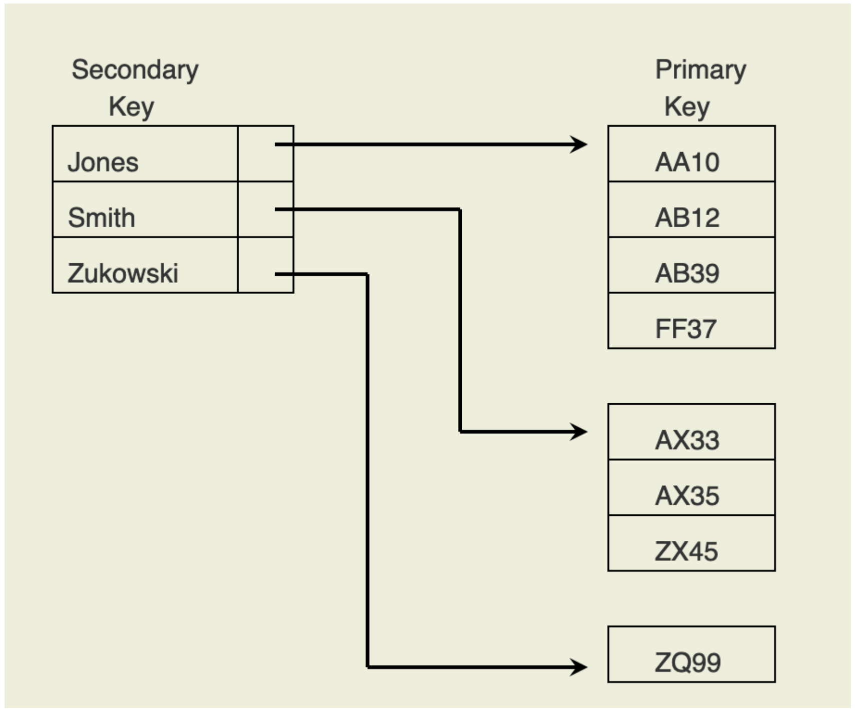
milk jawn

milk jawn

milkjawn
Milk Jawn • Followed by stick_and_pope + ...

Secondary Key

- **Secondary key index**: associates a secondary key value with the primary key of each record having that secondary key value



Search using SK Index:
- Search for your secondary key to obtain a reference to the batch of primary keys whose records have that secondary key
- For each potential primary key match, look up the record associated with that primary keys using a PK index
- If you have a match, you’re done!
- If there’s no match, keep looking at the next possible primary key.

Database indexing

- Linear indexing
- Hash-based indexing
- Tree-based indexing

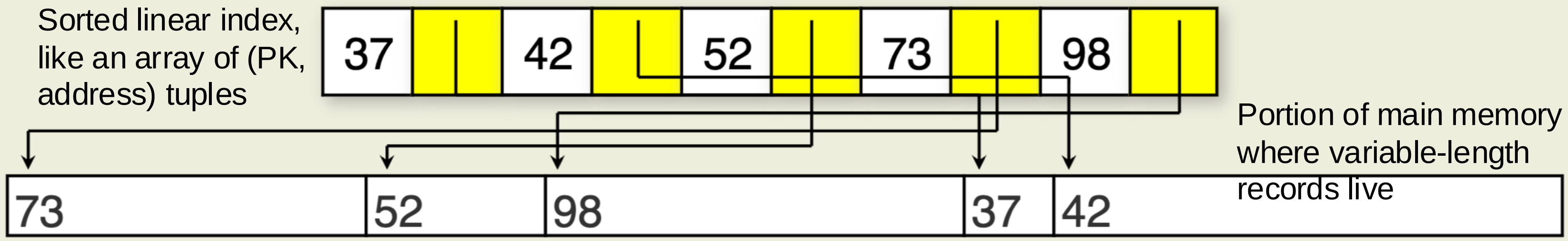
Index File

- **Index file**: a file whose records consist of key-value pairs where the values are referencing the complete records stored in another file

Where have we used an index file before in this course?

Linear indexing

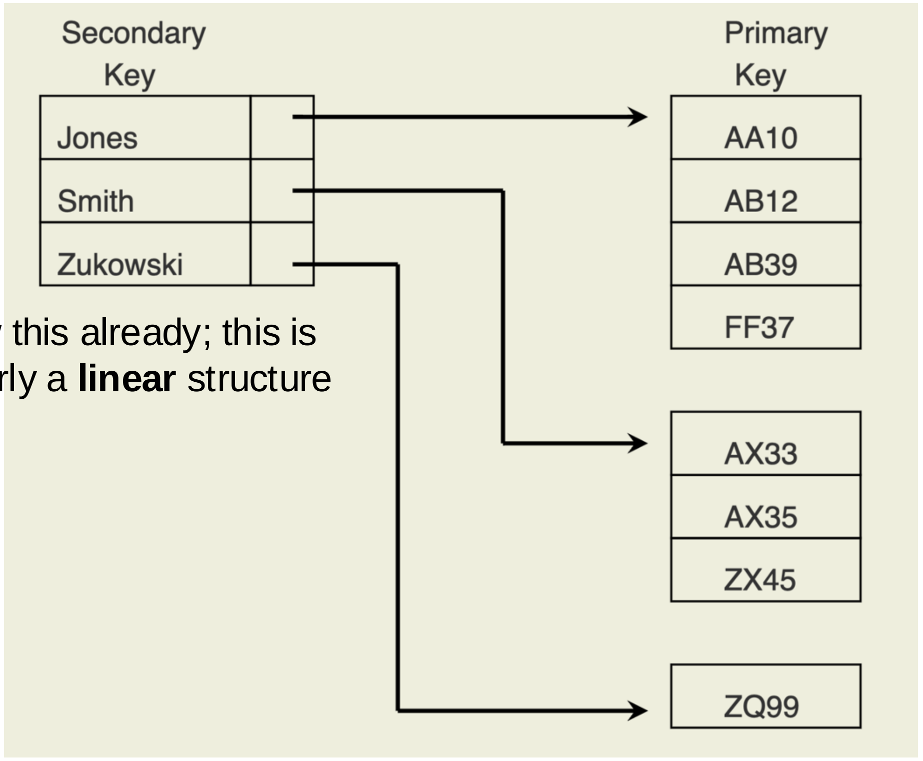
- Linear index: *index file* organized as a sequence of *key-value pairs* where the *keys* are in sorted order and the pointers either
    - **Point to the position of the complete record on disk (pictured)**
    - Point to the position of the *primary key* in the primary key index
- Linear index amenable to binary search (efficient search)


Sorted linear index, like an array of (PK, address) tuples
37
42
52
73
98
Portion of main memory where variable-length records live
73
52
98
37
42

Linear indexing

- Linear index: *index file* organized as a sequence of *key-value pairs* where the *keys* are in sorted order and the pointers either
  - Point to the position of the complete record on disk
  - **Point to the position of the *primary key* in the primary key index (pictured)**
- The secondary key index is called the **inverted list**

Is this PK part **linear?**

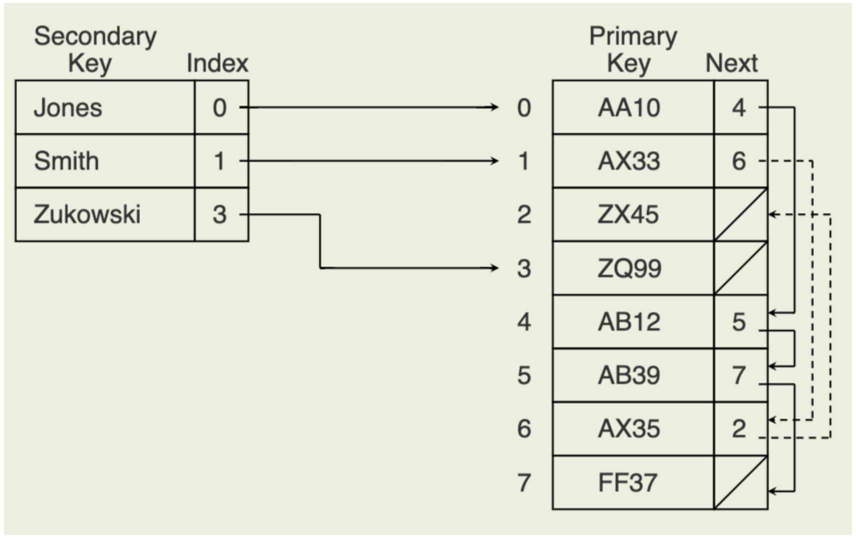


Saw this already; this is clearly a **linear** structure

Linear indexing

- Linear index: *index file* organized as a sequence of *key-value pairs* where the *keys* are in sorted order and the pointers either
  - Point to the position of the complete record on disk
  - **Point to the position of the *primary key* in the primary key index (pictured)**
- The secondary key index is called the **inverted list**

A better implementation: keep primary keys in an array for better space efficiency.



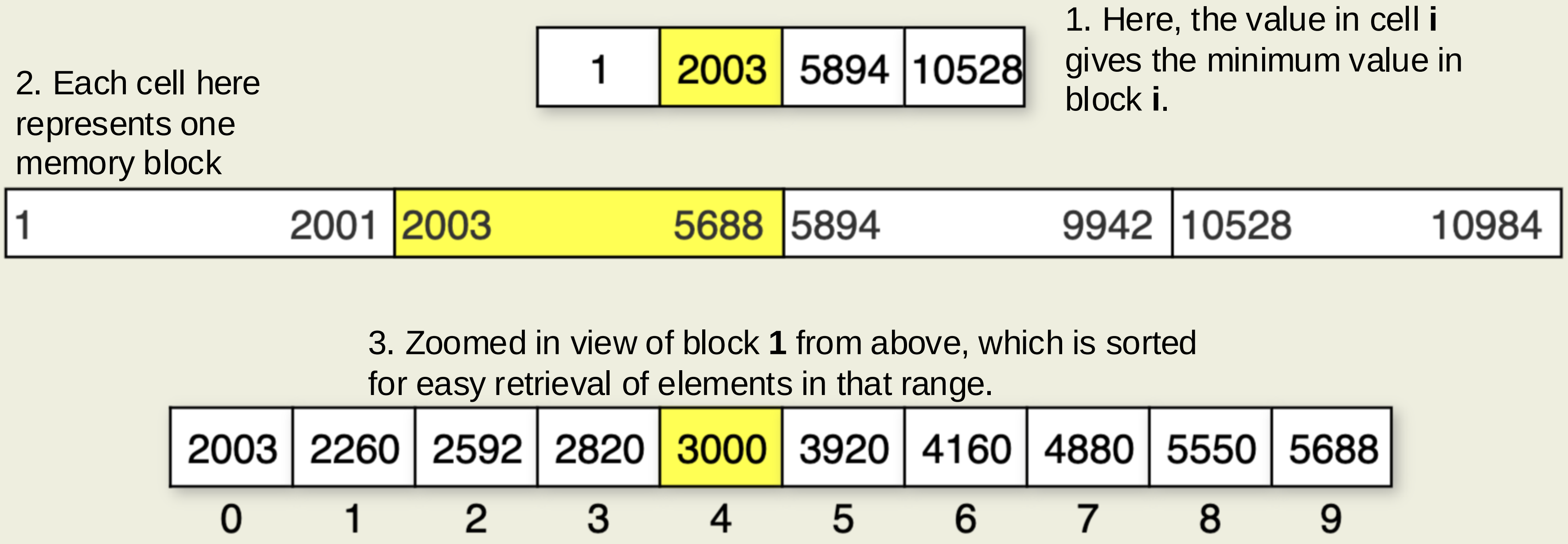
Second-level index

- Linear Index as implemented so far is good when:
  - Keys are much smaller than records
  - The dataset is not too large
  - i.e. when the primary keys can all be kept in memory

- What if all primary keys can't be kept in memory?
  - For large databases, linear index array/LL cannot fit in memory
  - Leads to expensive search because of several disk accesses

Second-level index

- Solution:
  - **Second-level index** stored in main memory (array)
    - NOT NECESSARILY RELATED TO SECONDARY KEYS: confusing name ☹


1. Here, the value in cell **i** gives the minimum value in block **i**.
1
2003
5894
10528
2. Each cell here represents one memory block
1
2001
2003
5688
5894
9942
10528
10984
3. Zoomed in view of block **1** from above, which is sorted for easy retrieval of elements in that range.
2003
2260
2592
2820
3000
3920
4160
4880
5550
5688
0
1
2
3
4
5
6
7
8
9

Second-level index

- Solution:
  - **Second-level index** stored in main memory (array)
    - NOT NECESSARILY RELATED TO SECONDARY KEYS
  - Index file stored across several blocks (on disk)
  - Second-level index stores the first key value in the corresponding disk block of the index file
  - Search requires 2 disk accesses: (1) load the block of the index file containing the key, (2) retrieve the record
    - Better than a linear search over all records, which might have to load many blocks

Linear indexing: Drawback

- Insertion and deletion are expensive
    - All secondary indices must be updated: the entire contents of the array might be shifted
- Secondary key indexes contain duplicates: space expensive

Interlude: HW6

Goal of HW6

- Build a news aggregator that allows a user to view articles based on the terms contained inside of them
- Tasks:
    - Reasoning about ethics and social impact
    - Using an RSS feel to crawl webpages **(!!!)**
    - Calculating TF-IDF for a corpus of documents
    - Creating an inverted index for each term-based search
    - Generating a term list and incorporating autocomplete

JSoup, RSS, & HTML

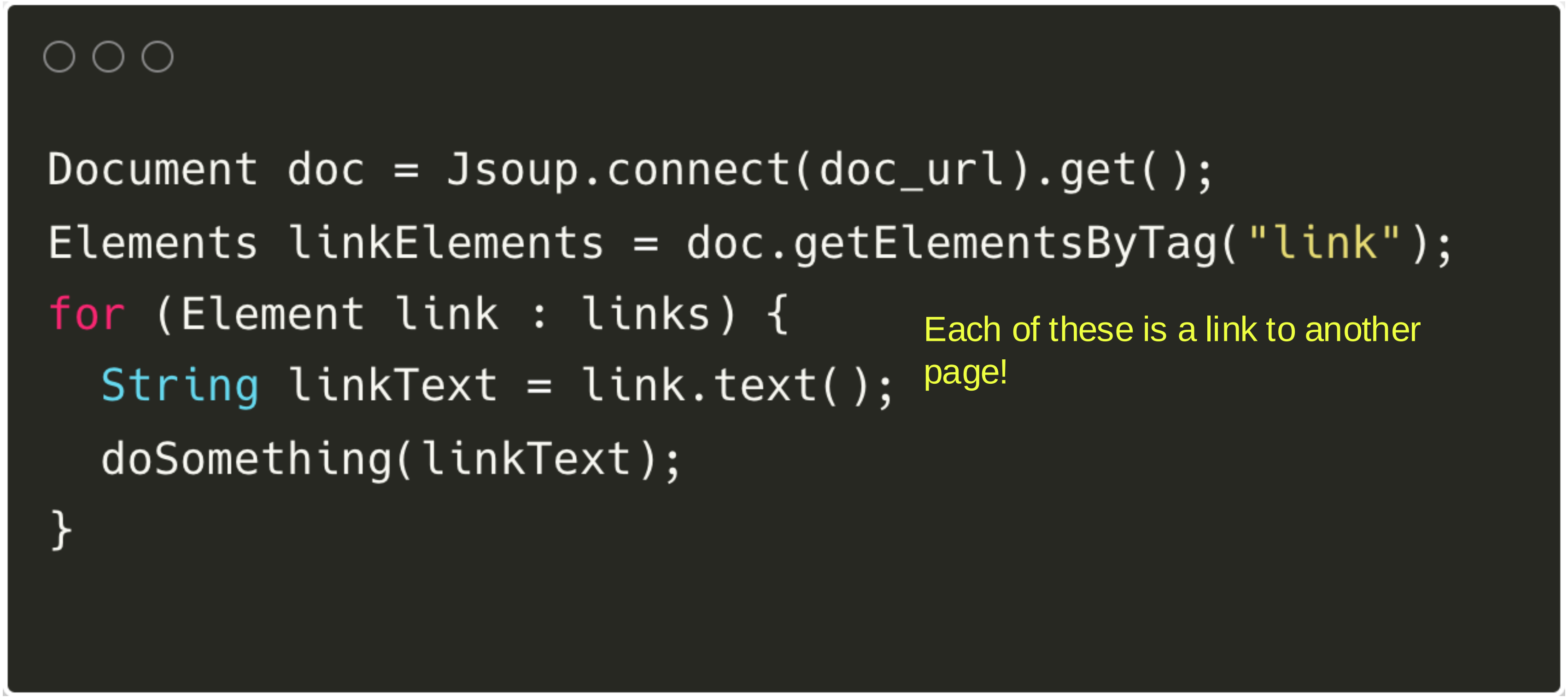
- We’re connecting to the internet and parsing documents hosted remotely.
- https://www.seas.upenn.edu/~cit5940/sample_rss_feed.xml

```
<rss version="2.0">
  <title>Hw6 Sample RSS Feed</title>
  <description>Sample RSS feed for CIT594 news aggregator</description>
  <link>http://localhost:8090/page1.html</link>
  <link>http://localhost:8090/page2.html</link>
  <link>http://localhost:8090/page3.html</link>
  <link>http://localhost:8090/page4.html</link>
  <link>http://localhost:8090/page5.html</link>
</rss>
```

  - RSS, pictured above, is a markup language that allows you to specify a series of data sources (LINK: rss for podcasts.)

JSoup, RSS, & HTML

- But wait: how do you connect to the internet?
    - Download JSoup and add it to your Eclipse/IntelliJ project (instructions included in writeup)
    - Then, to manipulate each **link** in an RSS feed:

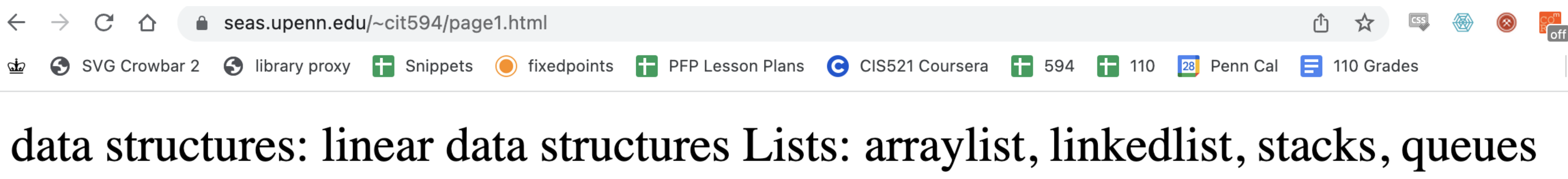


```
Document doc = Jsoup.connect(doc_url).get();
Elements linkElements = doc.getElementsByTag("link");
for (Element link : links) {
  String linkText = link.text();
  doSomething(linkText);
}
```

Each of these is a link to another page!

JSoup, RSS, & HTML

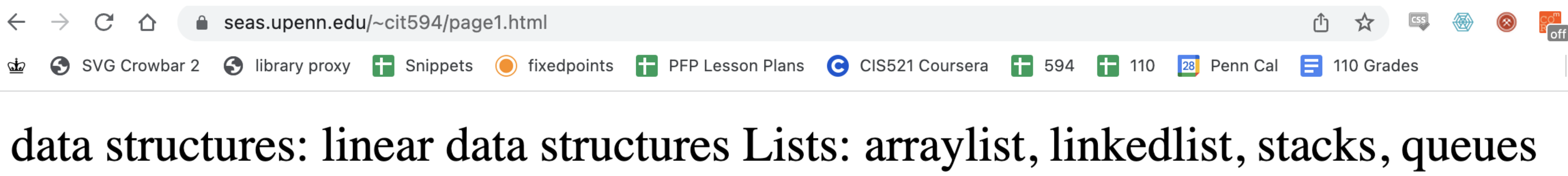
- But wait: how do you connect to the internet?
  - Download JSoup and add it to your project (instructions included in writeup)
  - You can manipulate each **link** in an RSS feed.
  - For each link, navigate to that page to find the list of terms contained in that page
  - Use JSoup the same way to manipulate HTML as the RSS document.



data structures: linear data structures Lists: arraylist, linkedlist, stacks, queues

JSoup, RSS, & HTML

- But wait: how do you connect to the internet?
    - Download JSoup and add it to your project (instructions included in writeup)
    - You can manipulate each **link** in an RSS feed.
    - For each link, navigate to that page to find the list of terms contained in that page
    - Use JSoup the same way to manipulate HTML as the RSS document.



seas.upenn.edu/~cit594/page1.html

data structures: linear data structures Lists: arraylist, linkedlist, stacks, queues

seas.upenn.edu/~cit5940/page1.html
(the document name)

→

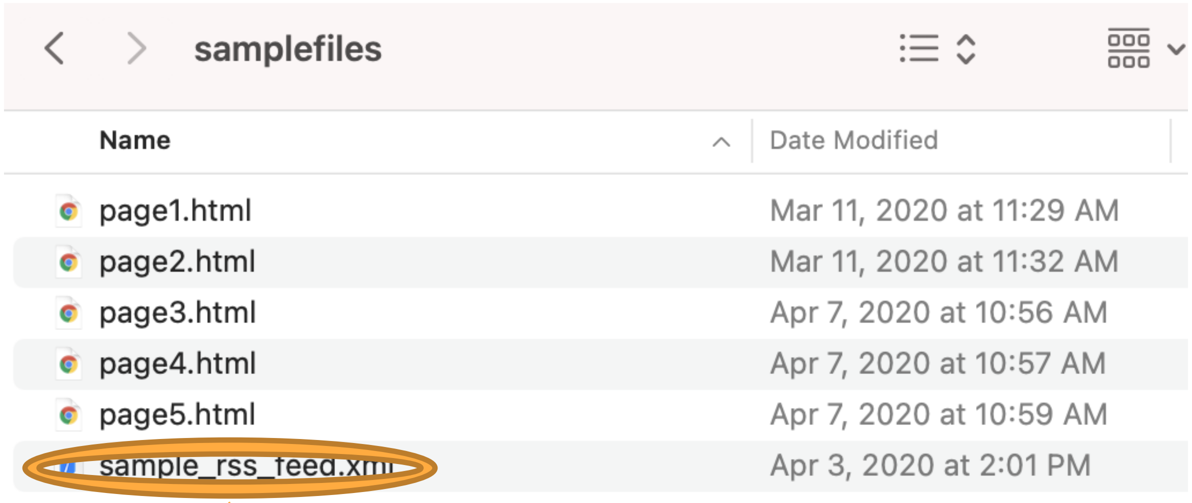
<data, structures, linear, data, structures, lists, arraylist, linkedlist, stacks, queues>
(the list of terms)

Testing and the internet

- You should write your own test cases as always, but how to host your own RSS feed for access?

https://www.python.org/downloads/

Writing your own testing files



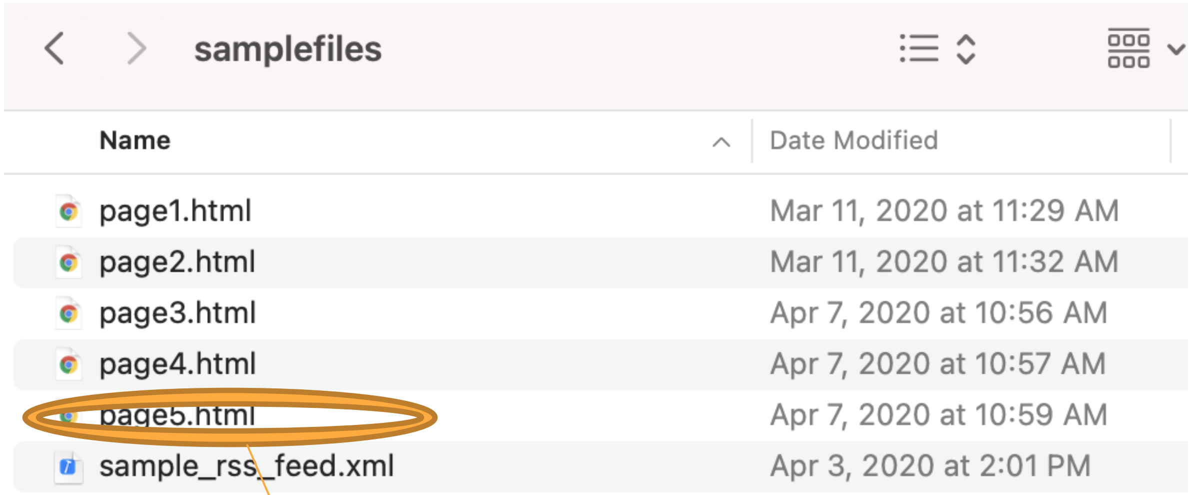
Name	Date Modified
page1.html	Mar 11, 2020 at 11:29 AM
page2.html	Mar 11, 2020 at 11:32 AM
page3.html	Apr 7, 2020 at 10:56 AM
page4.html	Apr 7, 2020 at 10:57 AM
page5.html	Apr 7, 2020 at 10:59 AM
sample_rss_feed.xml	Apr 3, 2020 at 2:01 PM

```
  <rss version="2.0">
      <title>Hw6 Sample RSS Feed</title>
      <description>Sample RSS feed for CIT594 news aggregator</description>
      <link>http://localhost:8090/page1.html</link>
      <link>http://localhost:8090/page2.html</link>
      <link>http://localhost:8090/page3.html</link>
      <link>http://localhost:8090/page4.html</link>
      <link>http://localhost:8090/page5.html</link>

</rss>
```

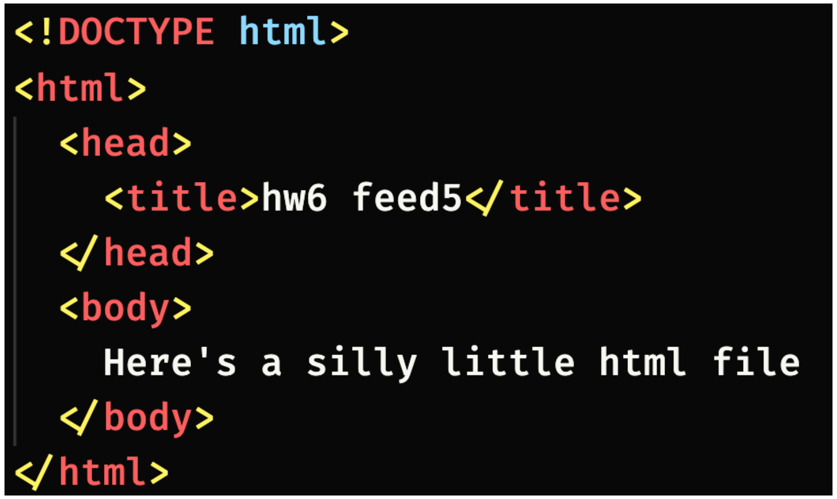
Keep the base URL for your pages http://localhost:8090 and write whatever you want in the other .html files in your directory!

Writing your own testing files



samplefiles

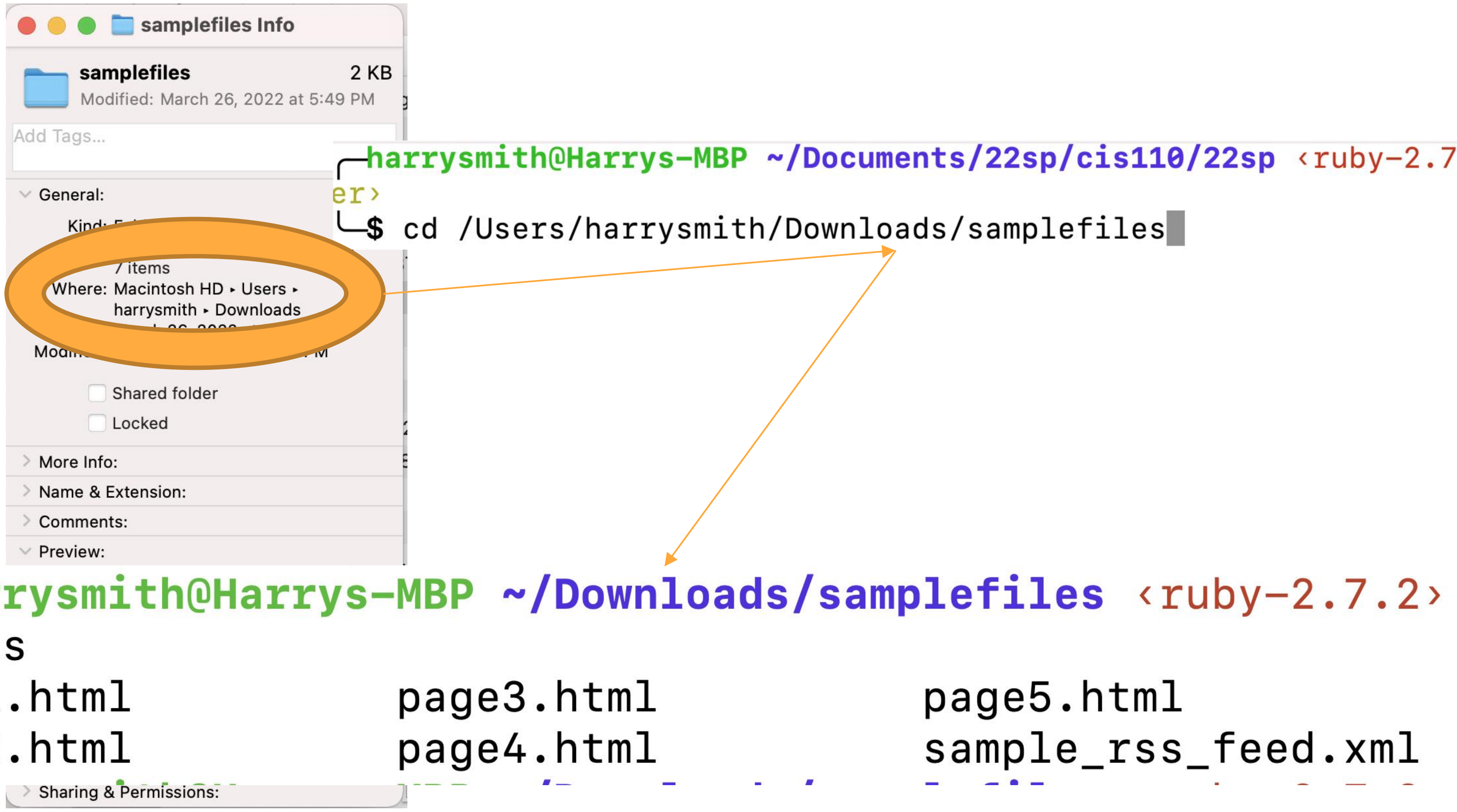
Name	Date Modified
page1.html	Mar 11, 2020 at 11:29 AM
page2.html	Mar 11, 2020 at 11:32 AM
page3.html	Apr 7, 2020 at 10:56 AM
page4.html	Apr 7, 2020 at 10:57 AM
page5.html	Apr 7, 2020 at 10:59 AM
sample_rss_feed.xml	Apr 3, 2020 at 2:01 PM



```
<!DOCTYPE html>
<html>
  <head>
    <title>hw6 feed5</title>
  </head>
  <body>
    Here's a silly little html file
  </body>
</html>
```

Put whatever text you want in the bodies of your custom pages for testing. This is page5.html

Using a terminal, navigate to the directory containing your sample files.



samplefiles Info

samplefiles 2 KB
Modified: March 26, 2022 at 5:49 PM

Add Tags...

General:

7 items
Where: Macintosh HD ▸ Users ▸ harrysmith ▸ Downloads

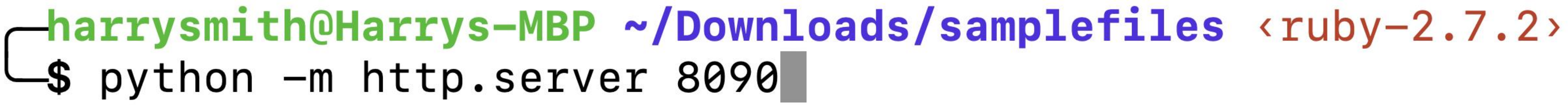
Shared folder
Locked

More Info:
Name & Extension:
Comments:
Preview:
Sharing & Permissions:

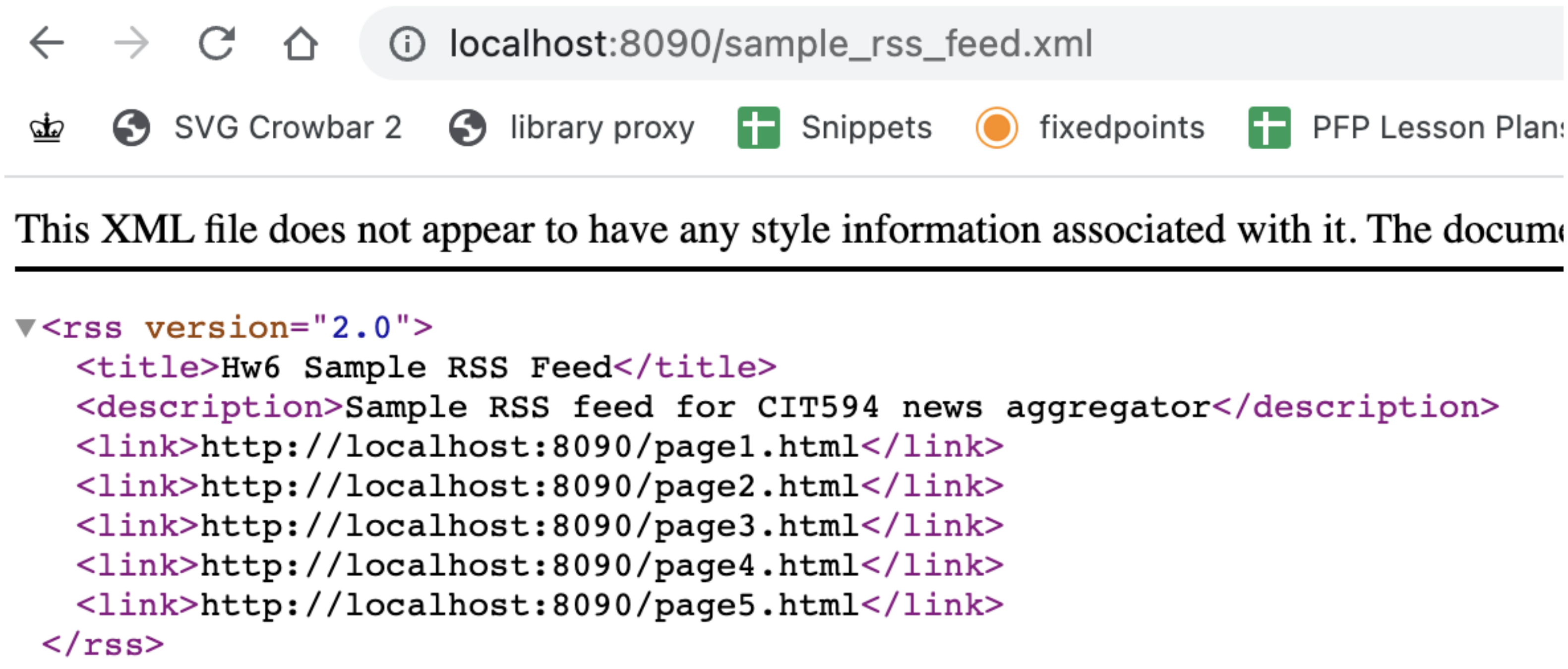
```
┌─harrysmith@Harrys-MBP ~/Documents/22sp/cis110/22sp ‹ruby-2.7
er›
└─$ cd /Users/harrysmith/Downloads/samplefiles
```

```
┌─harrysmith@Harrys-MBP ~/Downloads/samplefiles ‹ruby-2.7.2›
└─$ ls
page1.html          page3.html          page5.html
page2.html          page4.html          sample_rss_feed.xml
```

Start a web server on port 8090



```
harrysmith@Harrys-MBP ~/Downloads/samplefiles ‹ruby-2.7.2›
$ python -m http.server 8090
```



This XML file does not appear to have any style information associated with it. The docum

```
▼<rss version="2.0">
   <title>Hw6 Sample RSS Feed</title>
   <description>Sample RSS feed for CIT594 news aggregator</description>
   <link>http://localhost:8090/page1.html</link>
   <link>http://localhost:8090/page2.html</link>
   <link>http://localhost:8090/page3.html</link>
   <link>http://localhost:8090/page4.html</link>
   <link>http://localhost:8090/page5.html</link>
 </rss>
```

Friday’s Recitation Activity

- Hash a sequence of numbers using a few different hash systems (e.g. linear probing, linear probing with steps, quadratic probing, pseudo-random probing, double hashing)
- Identifying what a hash system is based on the probe functions used
- Describing an algorithm for using a linear search to find records in a range (not code)

TF-IDF: term frequency-inverse document frequency

- Term frequency: how often does a term appear in a particular document?
- Document frequency: how many documents does a particular term appear in?

- - TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document)
  - IDF(t) = log_e(Total number of documents / Number of documents with term t in it)
  - TF-IDF(t) = TF * IDF

Motto: A term has a high TF-IDF score in a given document if it appears a lot in that document but not very often in every document.

TF-IDF: term frequency-inverse document frequency

- Document 1: "The solar system consists of the Sun and eight planets. Mercury is the closest planet to the Sun, while Neptune is the furthest."
- Document 2: "Mercury is a chemical element with the symbol Hg. Mercury is commonly known as quicksilver and was formerly named hydrargyrum."
- Document 3: "The planet Venus has a thick atmosphere that traps heat, making it the hottest planet in our solar system despite not being the closest to the Sun."
- Document 4: "The Sun provides solar energy that can be captured using photovoltaic panels and converted into electricity."

TF-IDF: term frequency-inverse document frequency

- Document 1: "The solar system consists of the Sun and eight planets. Mercury is the closest planet to the Sun, while Neptune is the furthest."
- Document 2: "Mercury is a chemical element with the symbol Hg. Mercury is commonly known as quicksilver and was formerly named hydrargyrum."
- Document 3: "The planet Venus has a thick atmosphere that traps heat, making it the hottest planet in our solar system despite not being the closest to the Sun."
- Document 4: "The Sun provides solar energy that can be captured using photovoltaic panels and converted into electricity."

TF of “the” in document 1: high or low?
IDF of “the: high or low?

TF-IDF: term frequency-inverse document frequency

- Document 1: "The solar system consists of the Sun and eight planets. Mercury is the closest planet to the Sun, while Neptune is the furthest."
- Document 2: "Mercury is a chemical element with the symbol Hg. Mercury is commonly known as quicksilver and was formerly named hydrargyrum."
- Document 3: "The planet Venus has a thick atmosphere that traps heat, making it the hottest planet in our solar system despite not being the closest to the Sun."
- Document 4: "The Sun provides solar energy that can be captured using photovoltaic panels and converted into electricity."

TF of “the” in document 1: **high**
IDF of “the”: **low**

TF-IDF: term frequency-inverse document frequency

- Document 1: "The solar system consists of the Sun and eight planets. Mercury is the closest planet to the Sun, while Neptune is the furthest."
- Document 2: "Mercury is a chemical element with the symbol Hg. Mercury is commonly known as quicksilver and was formerly named hydrargyrum."
- Document 3: "The planet Venus has a thick atmosphere that traps heat, making it the hottest planet in our solar system despite not being the closest to the Sun."
- Document 4: "The Sun provides solar energy that can be captured using photovoltaic panels and converted into electricity."

TF of “Mercury” in document 2: high or low?
IDF of “Mercury”: high or low?

TF-IDF: term frequency-inverse document frequency

- Document 1: "The solar system consists of the Sun and eight planets. Mercury is the closest planet to the Sun, while Neptune is the furthest."
- Document 2: "**Mercury** is a chemical element with the symbol Hg. **Mercury** is commonly known as quicksilver and was formerly named hydrargyrum."
- Document 3: "The planet Venus has a thick atmosphere that traps heat, making it the hottest planet in our solar system despite not being the closest to the Sun."
- Document 4: "The Sun provides solar energy that can be captured using photovoltaic panels and converted into electricity."

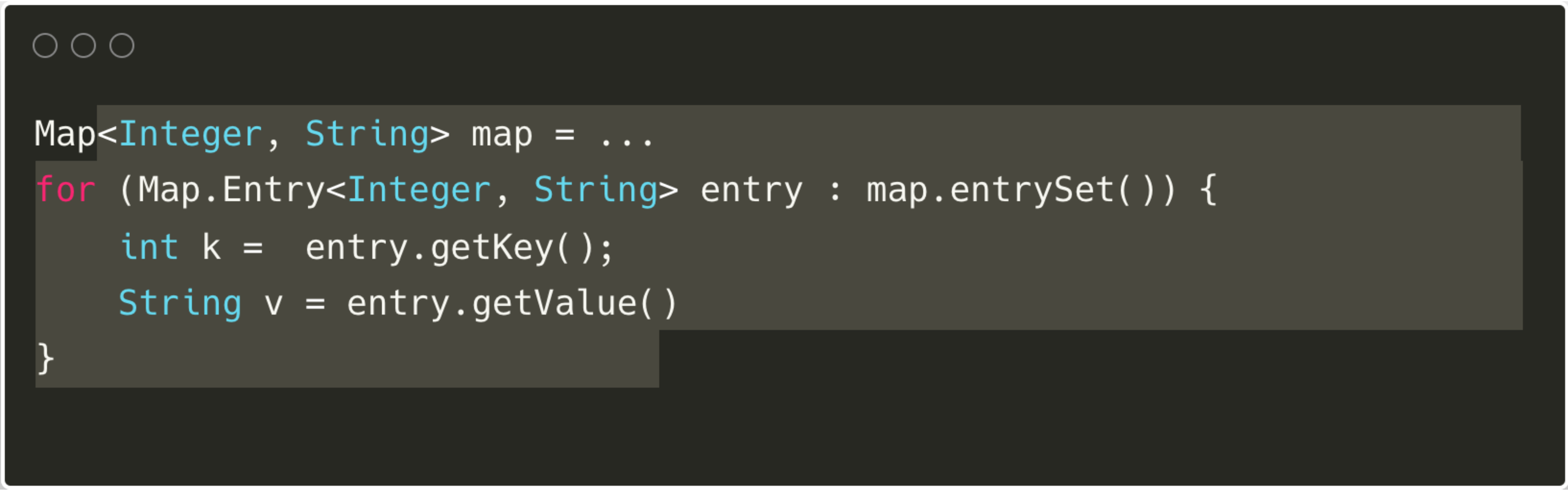
TF of “Mercury” in document 2: **high**
IDF of “Mercury”: **high**

This Assignment's Indices:

- The full records are effectively <Term, Document, TF-IDF>
  - Some term
  - The document in which it appeared
  - The TF-IDF score for that term in that document (relative to this corpus)
- The first index you build looks up full records with the Document as the key
  - Useful for looking up the Terms that appear in a particular Document.
- The second index is **inverted**, mapping a Term to the Document it appears in
  - Useful for looking up in which Document a term had the highest (or lowest) TF-IDF

This Assignment’s Indices:

- Are all maps!
- Quick trick for iteration over maps:
    - **Entry** objects are defined as Key, Value pairs for a particular map



```
Map<Integer, String> map = ...
for (Map.Entry<Integer, String> entry : map.entrySet()) {
    int k =  entry.getKey();
    String v = entry.getValue()
}
```

Linear Index

- An ordered (possibly sorted) collection of entries.
  - Usually traversed using linear search or binary search
  - Linear search:
    - Useful when the index is not sorted
    - Also useful for **range queries**
  - Binary search:
    - Only applicable when the index is sorted
    - Also useful for **range queries**

Linear Index

- Inserting a record into the database:
  - **Unsorted index: O(1)** update (just append the <key, pointer> entry to the end of the index)
  - **Sorted index: O(n)** update (have to insert the entry somewhere in the middle of the index, moving ~50% of the entries on average)
- Deleting a record from the database:
  - **Unsorted/Sorted indices: O(n)** to find the entry and then delete it from the array/list (shift the other stuff)

Linear Index

- Marking a record from the database as inaccessible:
  - **Unsorted index: O(n)** to find the entry and then tag it as “inaccessible”
  - **Sorted index: O(lg n)** to find the entry and then tag it as “inaccessible”

A common pattern: takes less time to “flag” an entry for deletion than it does to actually remove it from your system...

Indexing Example: Facebook’s Haystack

- Photo Storage infrastructure
- Disks are organized in volumes of fixed size
- Haystack index store consists of 2 files:
    - Haystack store (database)
    - Index file (used to rebuild the in-memory index)
- In-memory index used
- Each Haystack store manages multiple volumes
- Each haystack store has one in-memory index file
- Append-Only database

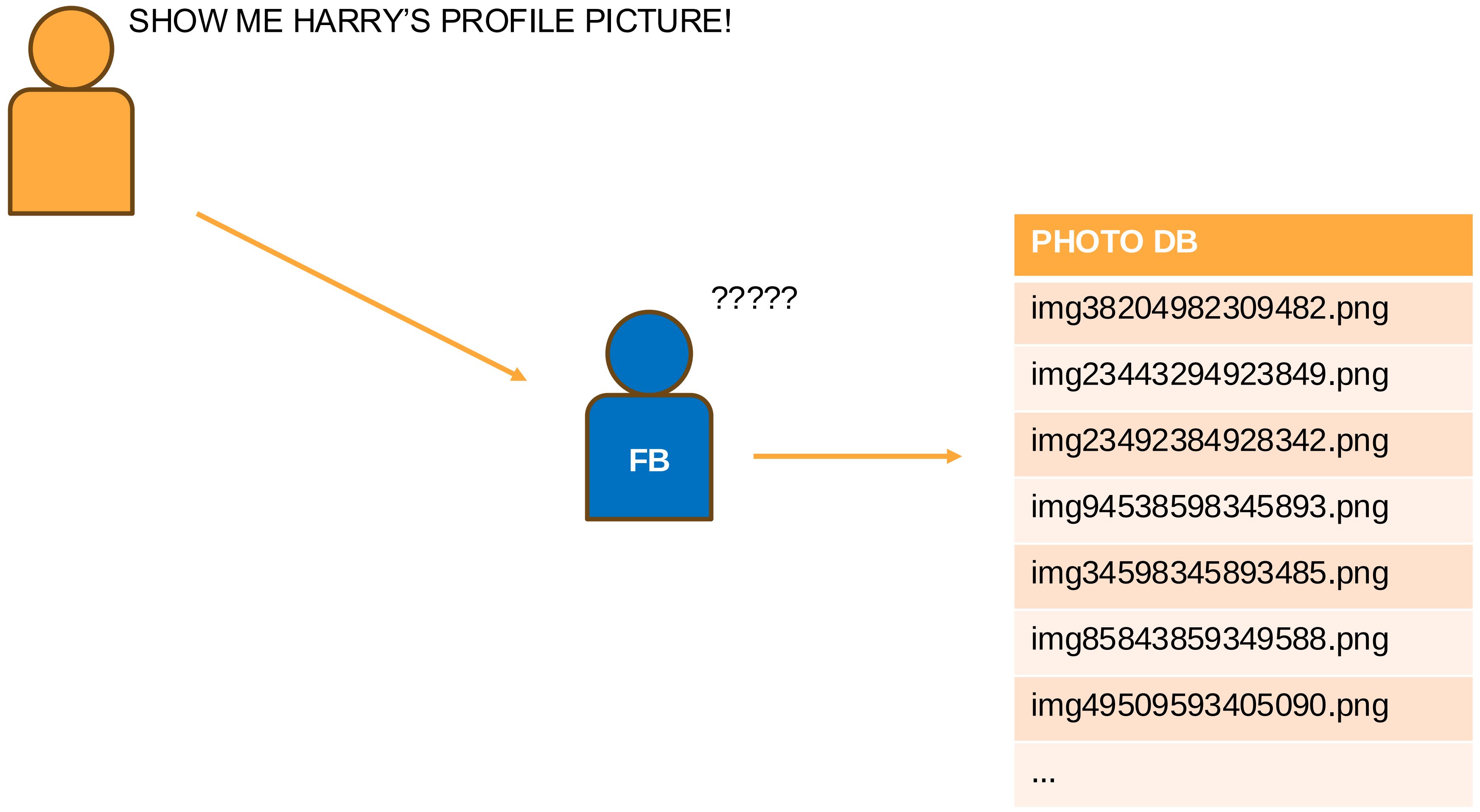
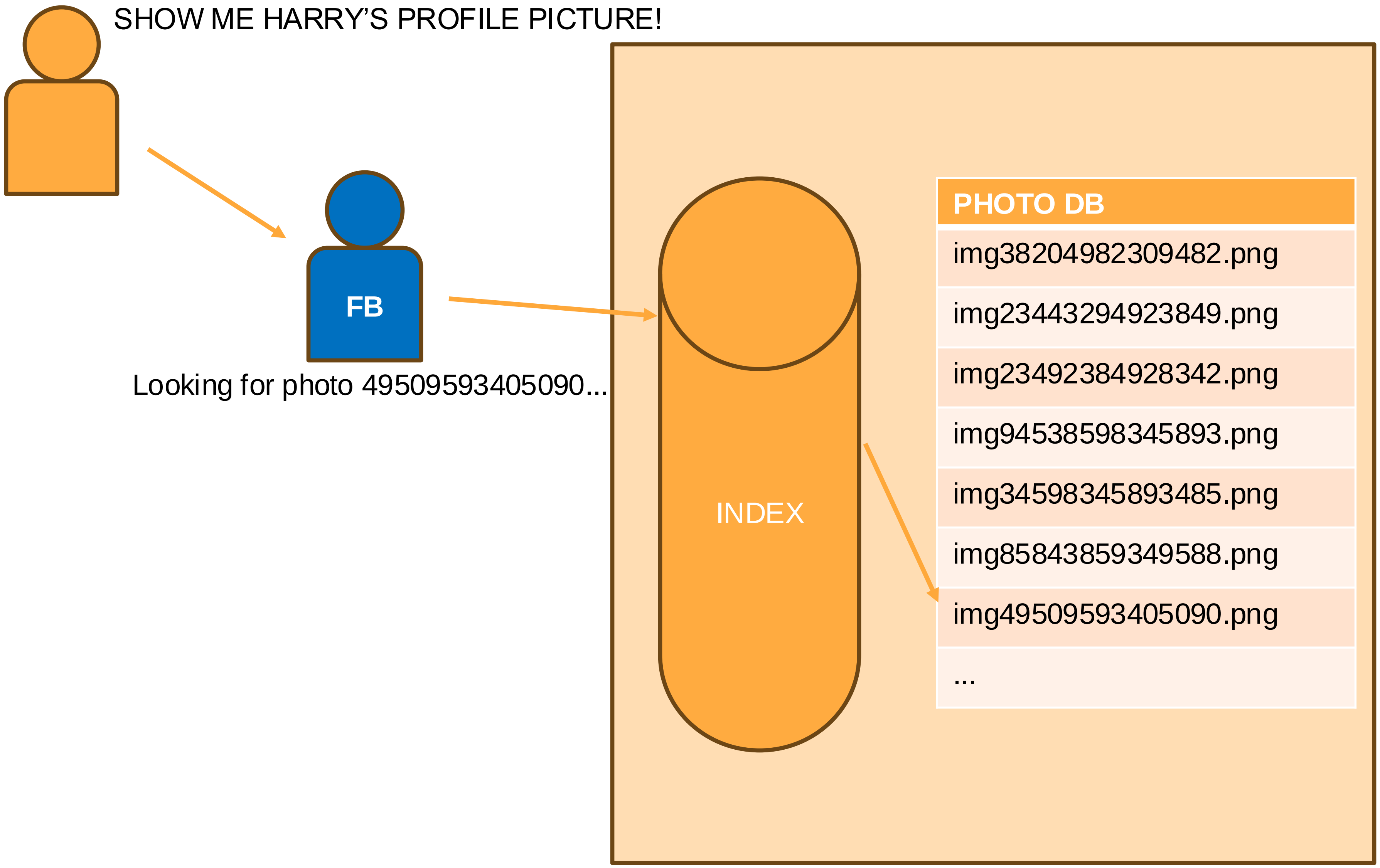


PHOTO DB
img38204982309482.png
img23443294923849.png
img23492384928342.png
img94538598345893.png
img34598345893485.png
img85843859349588.png
img49509593405090.png
...

With millions (billions?) of photos, how does Facebook retrieve photos quickly?


SHOW ME HARRY'S PROFILE PICTURE!
FB
Looking for photo 49509593405090...
INDEX
PHOTO DB
img38204982309482.png
img23443294923849.png
img23492384928342.png
img94538598345893.png
img34598345893485.png
img85843859349588.png
img49509593405090.png
...

Facebook’s Haystack

- Haystack Store: contains “needles,” where each is a **photo** and its requisite searching data.
- Each photo has a key and an alternate key.
- Photos are stored in **no particular order**
    - Roughly, in order of insertion

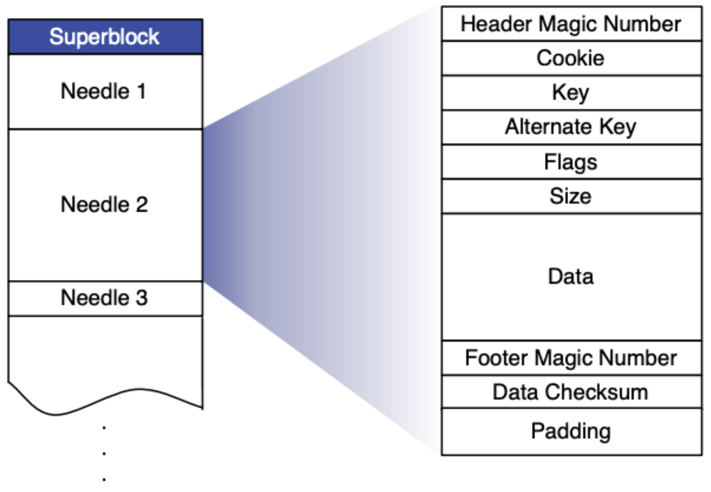


Figure 5: Layout of Haystack Store file

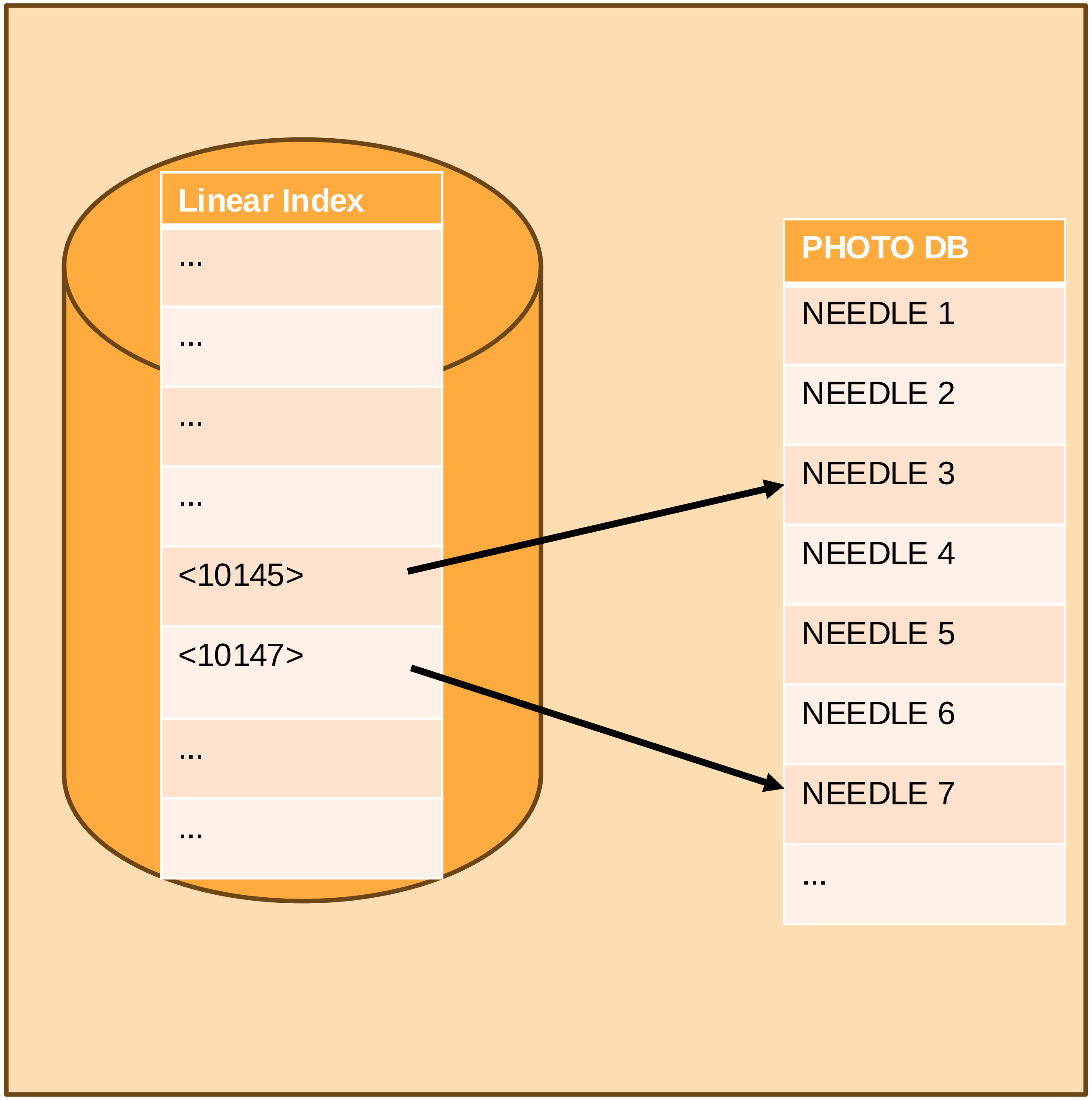
Field	Explanation
Header	Magic number used for recovery
Cookie	Random number to mitigate brute force lookups
Key	64-bit photo id
Alternate key	32-bit supplemental id
Flags	Signifies deleted status
Size	Data size
Data	The actual photo data
Footer	Magic number for recovery
Data Checksum	Used to check integrity
Padding	Total needle size is aligned to 8 bytes

Facebook’s Haystack

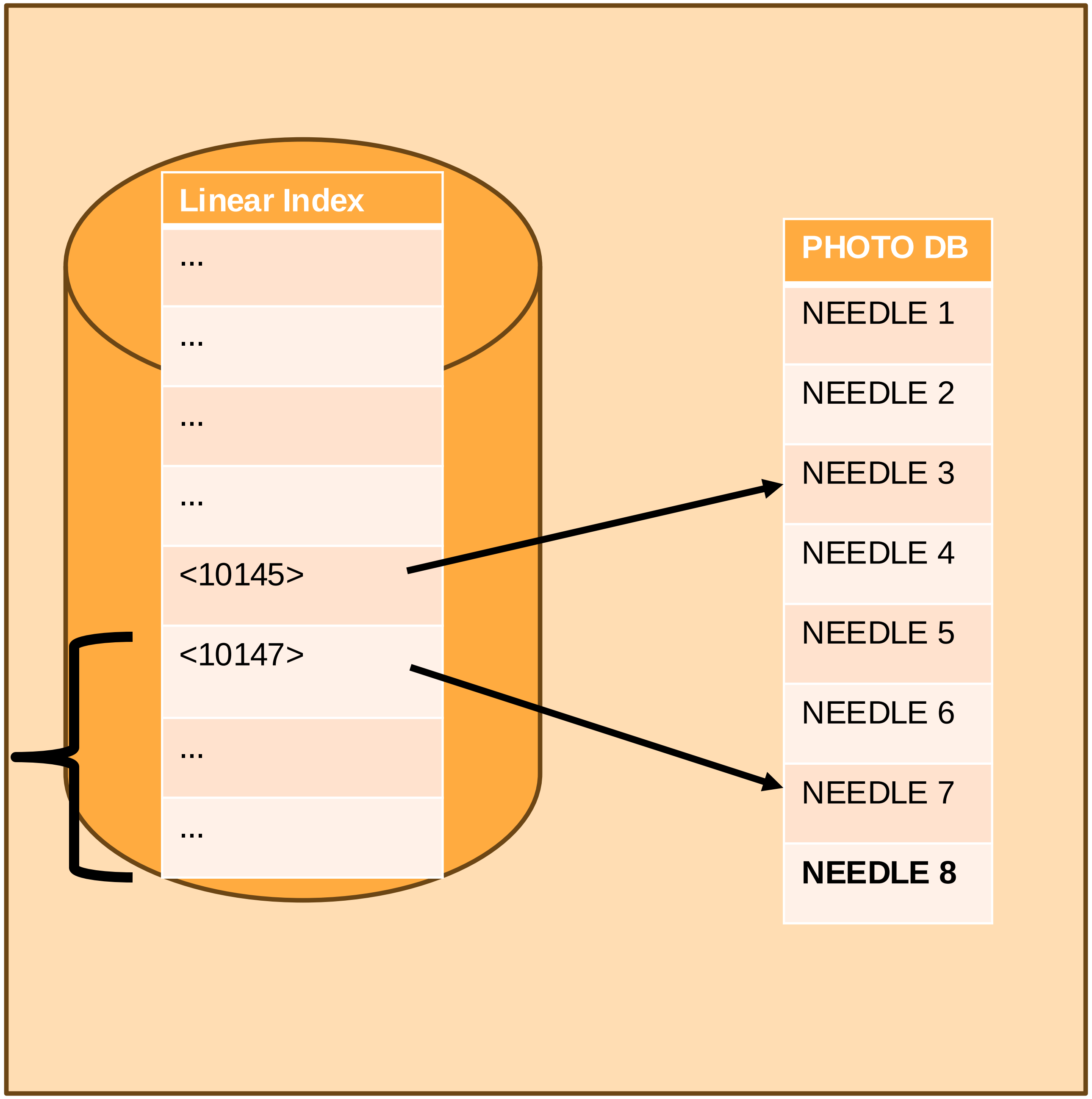
- Needle:
  - Represents a photo stored in the Haystack
  - Uniquely identified by its <Offset, Key, Alternate Key, Cookie> tuple
  - Multiple needles can have the same key
  - Offset: the needle offset in the haystack store
  - Offset is stored in index (file and in-memory)

Facebook’s Haystack

- In-memory index
    - Data structure that maps pairs of (key, alternate key) to the corresponding needle’s flags, size, and offset
    - Key is the photo id
    - Alternate key is the photo’s type. Each photo is scaled to four types/sizes
    - Linear index is inappropriate here...


Linear Index
...
...
...
...
<10145>
<10147>
...
...
PHOTO DB
NEEDLE 1
NEEDLE 2
NEEDLE 3
NEEDLE 4
NEEDLE 5
NEEDLE 6
NEEDLE 7
...

To add a new needle with ID 10146, we have to move everything in the index from here on down by one...

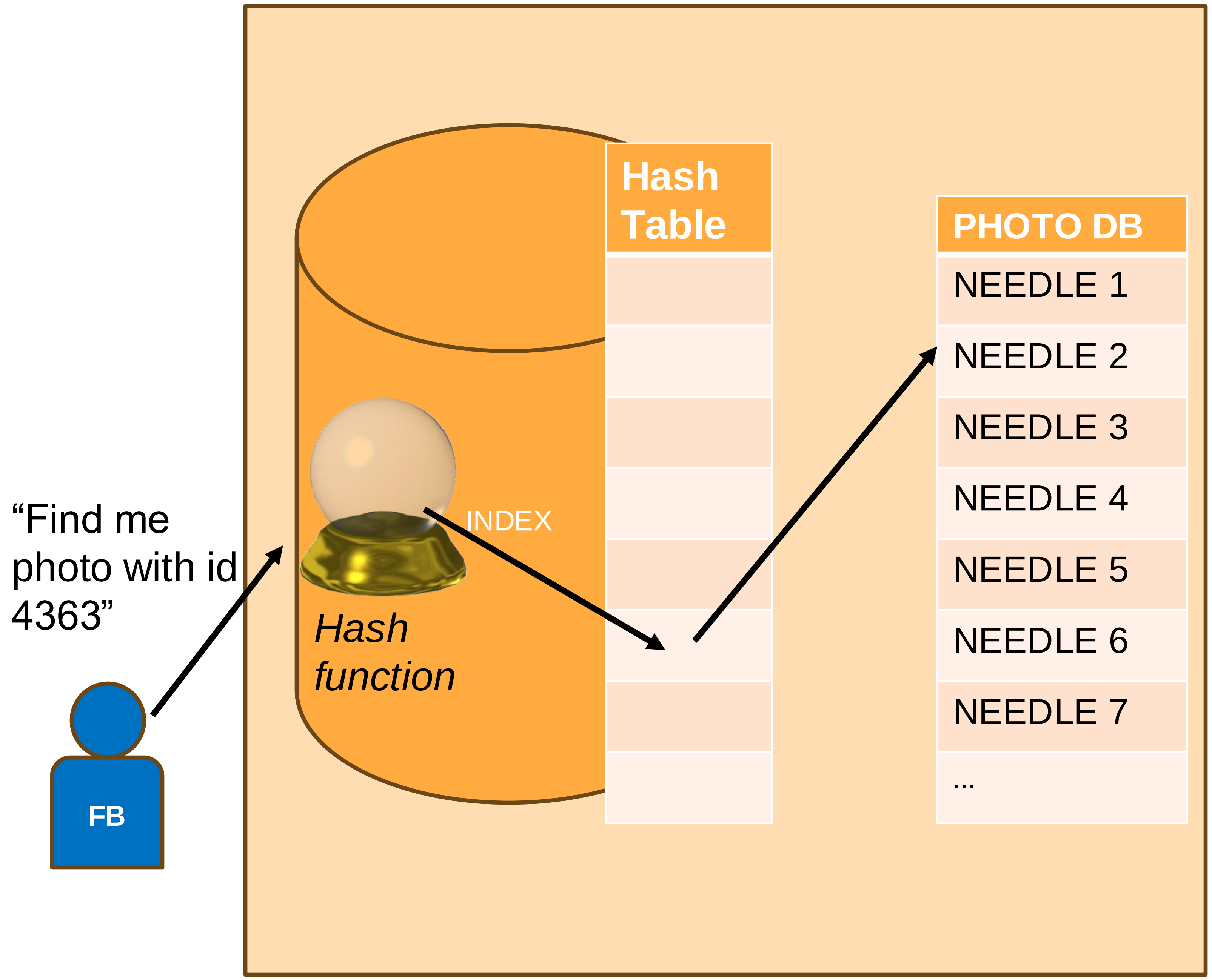


Facebook's Haystack

- In-memory index
  - Data structure that maps pairs of (key, alternate key) to the corresponding needle's flags, size, and offset
  - Key is the photo id
  - Alternate key is the photo's type. Each photo is scaled to four types/sizes
  - Google Sparsehash used (closed hashing + quadratic probing)
    - i.e. a hash map using <key, alternate key> as the hashing key

Hash Table Index

- A table that stores record locations—pointers (e.g. needle offsets)—organized by hash key
    - Searching for a record with a given key:
        - Hash the key and probe until found in the table
        - Follow the pointer located in the table
        - Return the record
    - **O(N/M)** lookup time


"Find me photo with id 4363"
FB
Hash function
INDEX
Hash Table
PHOTO DB
NEEDLE 1
NEEDLE 2
NEEDLE 3
NEEDLE 4
NEEDLE 5
NEEDLE 6
NEEDLE 7
...

Hash Table Index

- Inserting a record into the database:
    - **O(N/M)** index update (inserting the pointer into the table)
- Deleting a record from the database:
    - **O(N/M)** to find the pointer in the table and then "remove" it
    - Removing an entry from a hash table is a fraught process—need to make sure that anything that was inserted after in a probe sequence is still accessible!

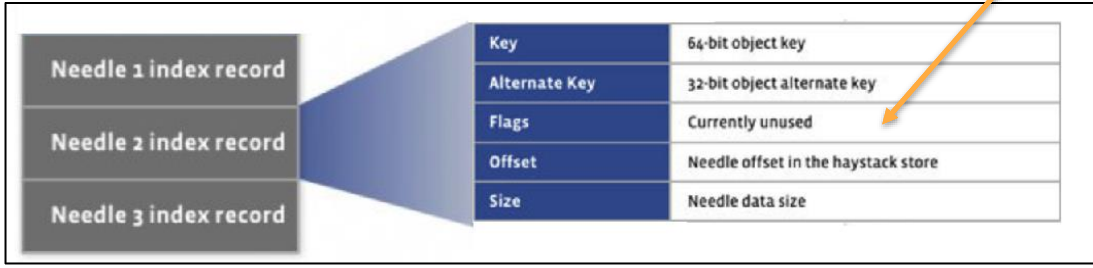
Facebook’s Haystack

- In-memory index

$$hash\ (key, alternate\ key) \longrightarrow location\ of\ needle\ in\ \underbrace{index}$$

Alternate key: large, medium, small, thumbnail

Delete status



Key	64-bit object key
Alternate Key	32-bit object alternate key
Flags	Currently unused
Offset	Needle offset in the haystack store
Size	Needle data size

Facebook’s Haystack: Photo Read

- Exact match query
- Each request contains the photo’s: logical volume id, key, alternate key, and cookie
- Hash function is used to find the photo in the in-memory index

Facebook’s Haystack: Photo Read

- If photo is deleted (flag sets to delete) stop
- Else find the needle in the volume based on the offset, reads the entire needle, performs integrity checks and returns the image
- **One disk access for each request**

Facebook’s Haystack: Photo Write

- Each request contains the logical volume id, key, alternate key, cookie, and data (photo)
- A new needle is created and appended (added at the end) to the Haystack
- A mapping for the new needle is added to the in-memory index

Facebook’s Haystack: Photo Write

- Special case: Photo modification (e.g. after rotation)
- A Needle cannot be overwritten (append-only)
- A new needle is created with the same key and alternate key as the original needle
- The in-memory index is updated: offset is updated to match the new needle

Facebook’s Haystack: Photo Delete

- Flag is set to “delete” in both in-memory index and Haystack store
- Requests to get deleted photos first check the in-memory flag and return errors if that flag is enabled