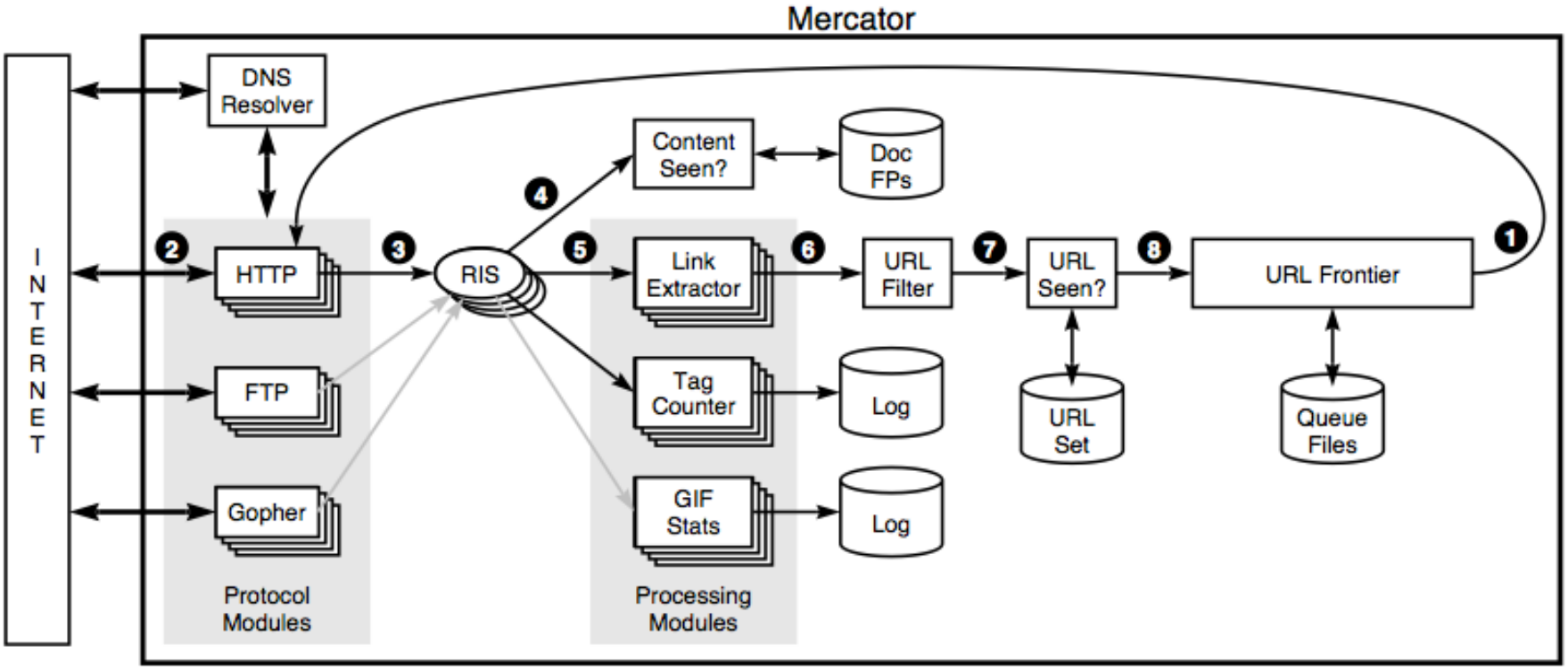
IR3: Scaling Web Search



Slides by Andrew DeOrio

Where we are

- Two lectures ago, we used the words on a page to rank search results
  - Rank on document content
- Last time we used the links between web pages to improve search results
  - Rank on document importance
- Today, we'll cover speed and scaling

A few numbers

- 5 B - 100 B web pages
  - Let's say 10 Billion for concreteness
- Assume 10KB per compressed page
  - 100 TB data to index
- 1 minute - 1 month freshness
- 3-5B queries *per day* on Google alone

Agenda

- **Crawler design**
- Deduplication
- Inverted index construction
- Distributed search architecture

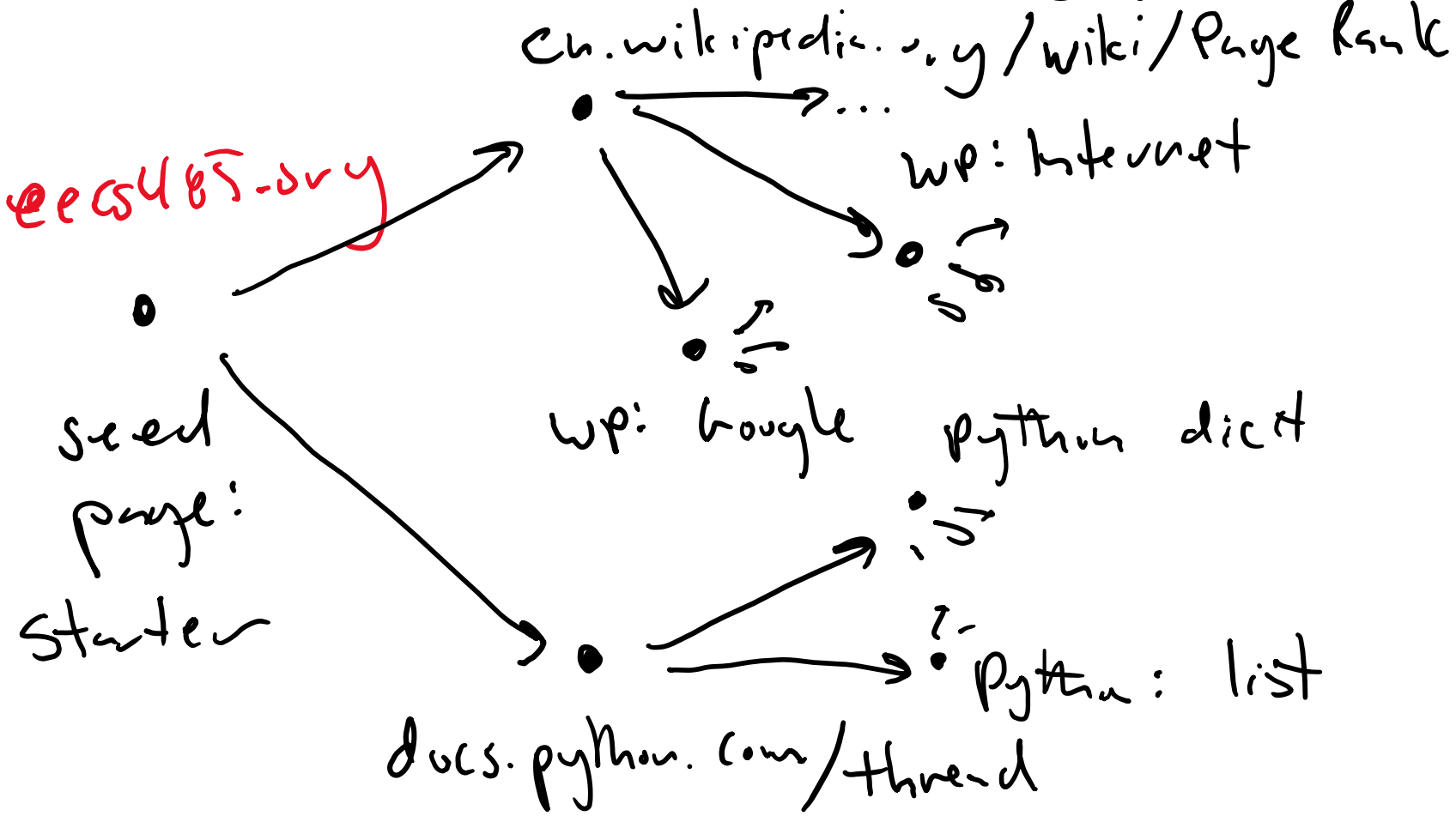
Crawlers

- To search the web, we need to build an index and link graphs
- Step 1: download the entire web
- Web Robots AKA Web Wanderers, Crawlers, or Spiders
- Example: `googlebot`
  - http://www.robotstxt.org/db/googlebot.html

- Problem: no list of all the pages

Web crawling

- Solution: do a traversal of the link graph



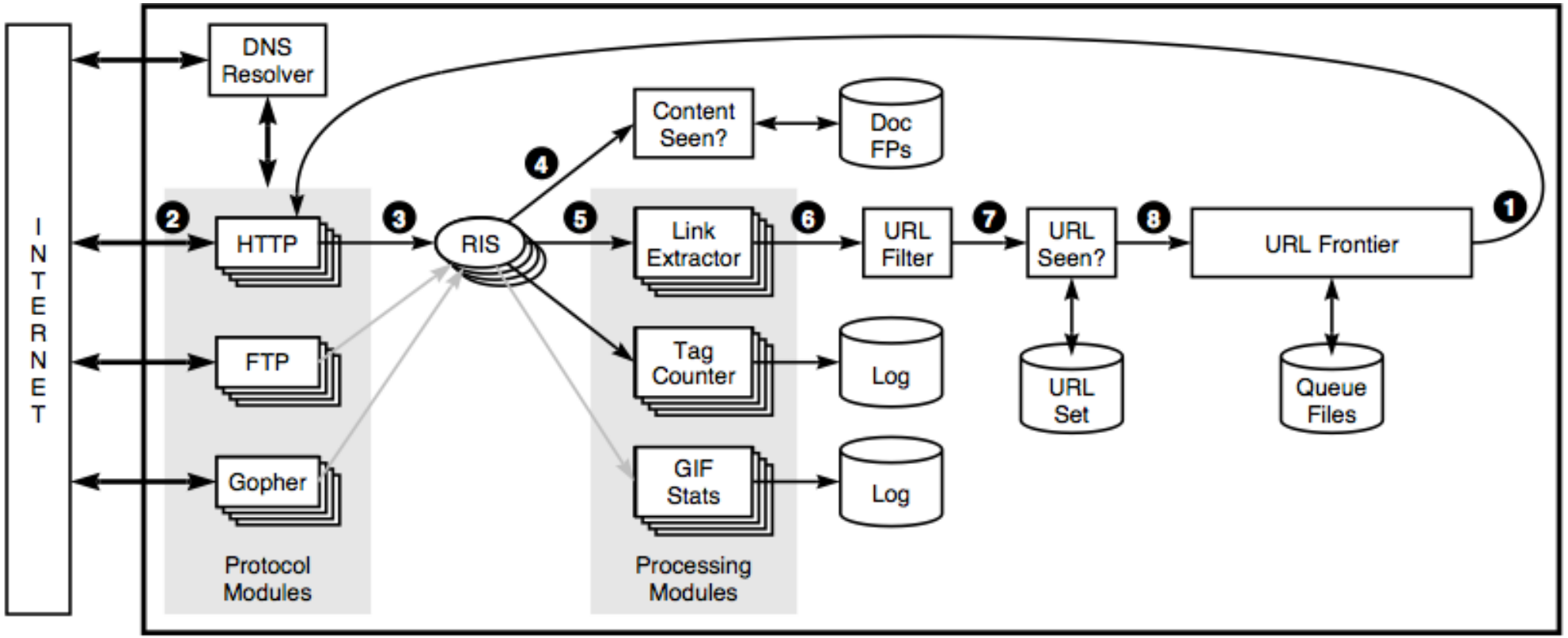
Web crawling: graph traversal

- Have a list of "seed pages", follow all links like a BFS/DFS
  - Most of the Internet is a connected component

Mercator

- Web crawler example: Mercator
- Mercator was the AltaVista crawler (1998)
- Exceptionally well-documented and useful to study, despite the many years
- Starts with seed URLs

Mercator



http://webpages.uncc.edu/sakella/courses/cloud09/papers/Mercator.pdf

Password protected content

- Problem: password-protected content
  - Bank records, Facebook, SnapChat, Dropbox, Google products, there's a lot
  - Crawler can't download it
- Solution: no solution. Can't index these pages.

Deep web

- *Surface web*: content indexed by search engines
- *Deep web*: content not indexed by search engines
- Size of deep web likely several orders of magnitude larger than surface web


SURFACE WEB
Google
Bing
Wikipedia
DEEP WEB
Academic Information
Medical Records
Legal Documents
Scientific Reports
Subscription Information
Contains 90% of the information on the Internet, but is not accessible by Surface Web crawlers.
Social Media
Multilingual Databases
Financial Records
Government Resources
Competitor Websites
Organization-specific Repositories
(DARK WEB)
A part of the Deep Web accessible only through certain browsers such as Tor designed to ensure anonymity. Deep Web Technologies has zero involvement with the Dark Web.
Illegal Information
TOR-Encrypted sites
Political Protests
Drug Trafficking sites
Private Communications

Client-side dynamic pages

- Problem: how to download client-side dynamic pages?
- Solution 1: skip those pages
- Solution 2: JS interpreter in crawler

```
<html>
<body>
    <div id="reactEntry>Loading...</div>
    <script src="..."></script>
</body>
</html>
```

Crawlers "not nice" to servers

- Problem: crawlers generate a **lot** of traffic
  - CPU and bandwidth costs money
- Website owners don't like if you crawl too fast and will block you
- Solution: limit # requests per site per second

Not all sites want to be indexed

- Problem: some websites don't want to be indexed
- Reasons not to index:
  - Personal creations: e.g. artwork on Google Images
  - Parts of dynamic websites: e.g. search engines
- Solution: robots.txt

robots.txt

https://github.com/robots.txt

```
User-agent: Googlebot
Disallow: /*/download
Disallow: /*/revisions
Disallow: /*/*/issues/new
Disallow: /*/*/issues/search
Disallow: /*/*/commits/*/*
Disallow: /*/*/commits/*?author
Disallow: /*/*/commits/*?path
...
```

Agenda

- Crawler design
- **Deduplication**
- Inverted index construction
- Distributed search architecture

Deduplication motivation

- Problem: Many web pages are duplicates
  - Has a page changed meaningfully compared to the last time we downloaded it?
  - Is it just a clone of another site? (weirdly common)

- How to detect duplicate pages in an efficient way?

Straightforward algorithm

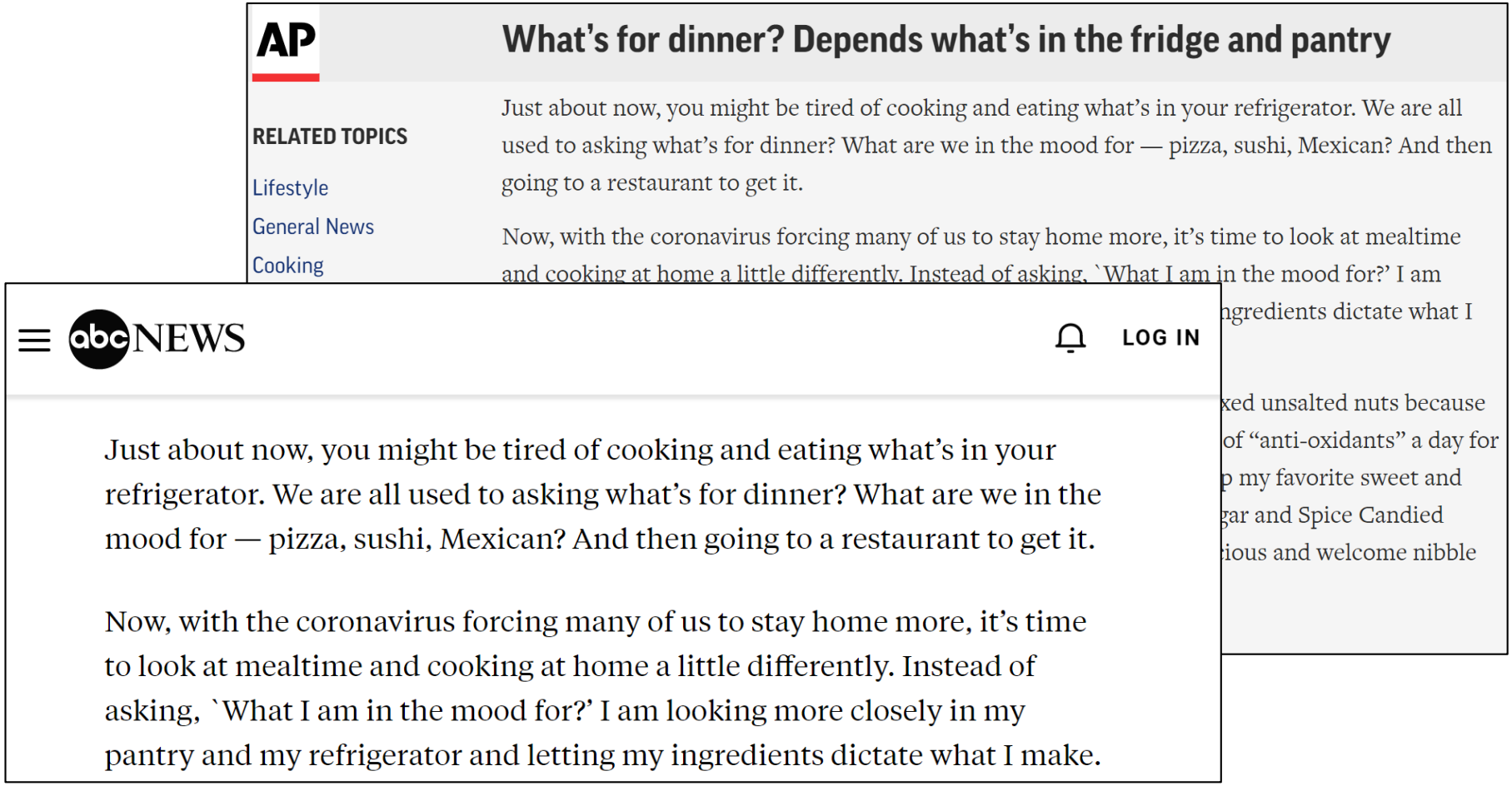
- Hash of a document
  - Hash("It grows in dry places...") = 8d2f908...
- When crawling a new document
  - Compute hash
  - Look up in hash table
    - If present, it's a duplicate
    - If not, put in the hash table
- How many comparisons for N web pages?

Straightforward algorithm

- Hash of a document
  - Hash("It grows in dry places...") = 8d2f908...
- When crawling a new document
  - Compute hash
  - Look up in hash table
    - If present, it's a duplicate
    - If not, put in the hash table
- How many comparisons for N web pages?
  - O(N) = O(N) lookups at O(1) per lookup

Why not compare entire document?

- Documents can be similar but not exactly the same



AP

What's for dinner? Depends what's in the fridge and pantry

RELATED TOPICS

Lifestyle

General News

Cooking

Just about now, you might be tired of cooking and eating what's in your refrigerator. We are all used to asking what's for dinner? What are we in the mood for — pizza, sushi, Mexican? And then going to a restaurant to get it.

Now, with the coronavirus forcing many of us to stay home more, it's time to look at mealtime and cooking at home a little differently. Instead of asking, `What I am in the mood for?' I am

ngredients dictate what I

xed unsalted nuts because
of "anti-oxidants" a day for
p my favorite sweet and
gar and Spice Candied
ious and welcome nibble

abcNEWS

LOG IN

Just about now, you might be tired of cooking and eating what's in your refrigerator. We are all used to asking what's for dinner? What are we in the mood for — pizza, sushi, Mexican? And then going to a restaurant to get it.

Now, with the coronavirus forcing many of us to stay home more, it's time to look at mealtime and cooking at home a little differently. Instead of asking, `What I am in the mood for?' I am looking more closely in my pantry and my refrigerator and letting my ingredients dictate what I make.

Quantifying similarity

- Measure similarity of a pair of docs, A and B
- Treat documents as sets of words
  - A = {"cooking", "AP", ...}
  - B = {"cooking, "ABC", ...}

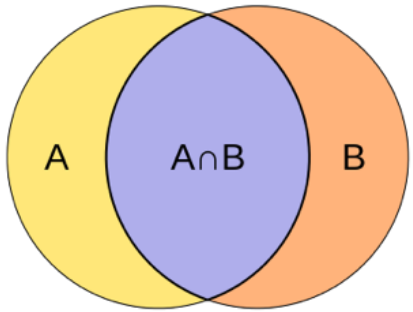
- Jaccard similarity

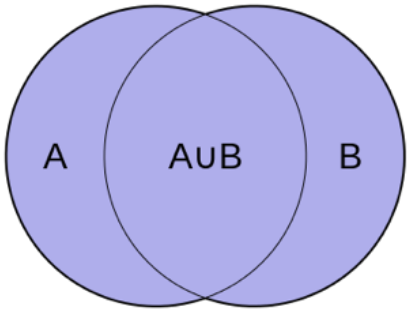
$$sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Jaccard similarity coefficient

- Size of the intersection / size of the union

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$





- 0 for disjoint sets, 1 for equal sets

Order of words

- Problem: If we compare two documents "exactly", then the order of exact words matters too much.
- Problem: If we compare two documents as sets of words using Jaccard similarity, then the order of words doesn't matter at all.

- Solution: shingles

Shingles

- *Shingles*: sequences of k words
  - Sometimes characters instead of words

- Example: "this is some text"
  - 2-shingles: {"this is", "is some", "some text"}
  - 3-shingles: {"this is some", "is some text"}

Shingles + Jaccard similarity

- Compute 3-shingles for doc A and doc B
- Compute Jaccard similarity between A and B
- If Jaccard exceeds threshold, then they're duplicates

Pairs of documents

- Because Jaccard similarity is a property of *pairs*, we compute it across all pairs

- How many pairs of documents are there?

$$\binom{n}{2} = \frac{n(n-1)}{2} \Rightarrow O(n^2)$$

n is # pages on Internet

Improving efficiency

- Goal: Pre-compute some information about every page which makes computing Jaccard similarity faster

- Non-goal: reduce O(n^2) comparisons. Need more algorithms than we'll cover in 485

Trick: hash shingles

- Compute hash of each shingle in docs A and B
  - O(n) string comparison, for string length n
  - O(1) numeric comparison

0x0c54

```
shingles_a = { "oun", "ada", "ntr", ... }
```

```
shingles_b = { "int", "arg", "oun", ...}
```

0x93cc 0x0c54

0f...

Trick: random shingles

- Problem: still must compute set union and intersection for n shingles
  - n is the number of shingles in a doc
- Solution: random shingle subset

Hashing and random values

- Problem: How to select a random hash value?
  - `shingle_hashes_a = {0x0cba, 0xb3f4, 0x499a, ...}`
  - `shingle_hashes_b = {0x39ff, 0x93cc, 0x0cba, ...}`
- Solution 1: select the element at a random index
  - What if container doesn't support random access?
- Solution 2: select the min hashed value
  - The hash function maps inputs uniformly over the output
  - Selecting the *min(h(x))* is the same as selecting a random item *x*!

Hashing and similarity

min

```
shingle_hashes_a = {0x0cba, 0xb3f4, 0x499a, ...}
shingle_hashes_b = {0x39ff, 0x93cc, 0x0cba, ...}
```

min

- Select min hash value from A, $h_{min}(A)$
  - This is a random shingle from A
- Select min hash value from B, $h_{min}(B)$
  - This is a random shingle from B
- Compare
  - If $h_{min}(A) == h_{min}(B)$, indicates similarity

MinHash + Jaccard similarity

- What is the probability that two random hashes match?
- It's the Jaccard similarity score!

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

- Pr[h_{min}(A) == h_{min}(B)] == Jaccard(A, B)

Calcuating Jaccard with MinHash

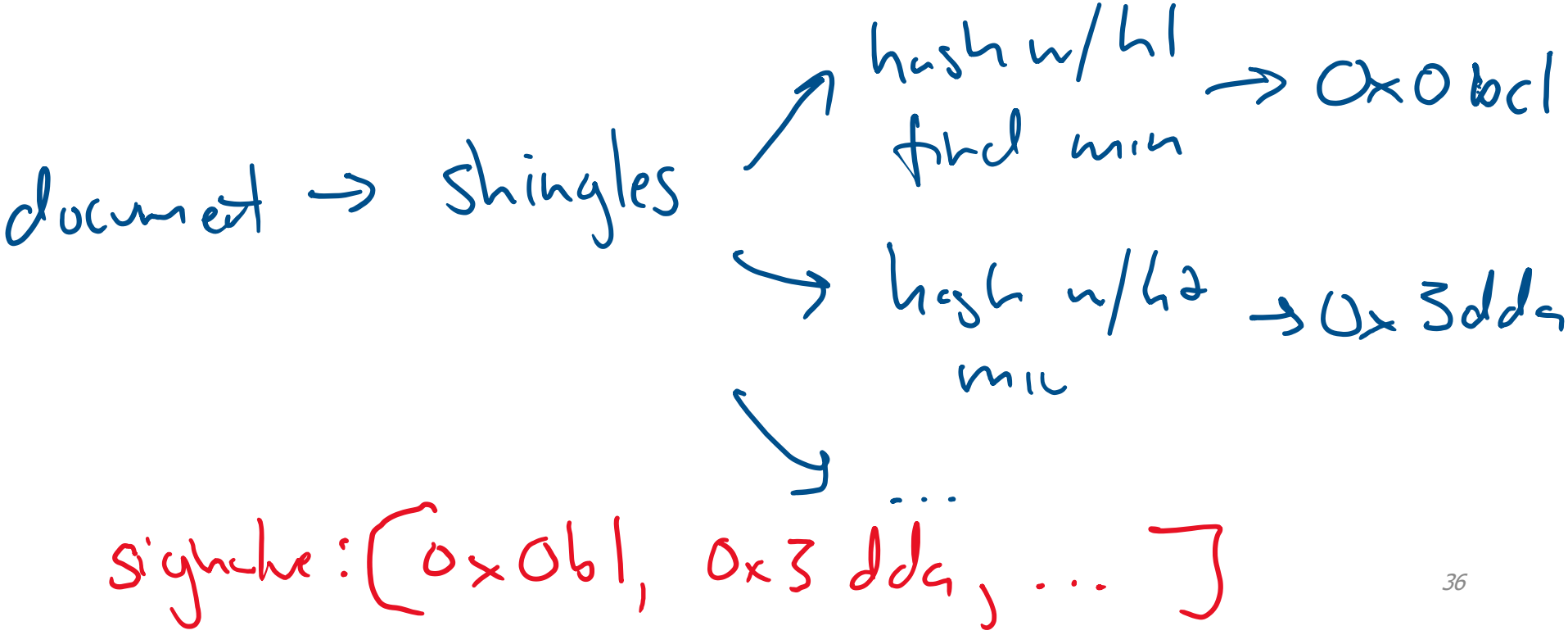
- $Pr[h_{min}(A) == h_{min}(B)] == Jaccard(A, B)$
- Do this with *k* hash functions to draw *k* samples
  - Or simply the k smallest values
- Number that match / total

MinHash idea

- Why MinHash?
- Computing hash values for a set is O(N)
  - For fixed k
  - Sometimes called computing the “signature” of the set
  - Compute this once
- Comparing two sets is now constant time (fixed k)
  - Do this many times when crawling or indexing

MinHash signatures

- Ahead of time, we can compute *signatures* for documents
- Given hash functions h1, h2, h3, ...



Comparing using signatures

```
# document signatures
[0x0b1a, 0x3dda, 0x43cf]
[0x6c4f, 0x3222, 0x232a]

# comparison document signature
[0x0b1a, 0x3222, 0x232a]
```

Score 1/3 = 0.3̄

Score 2/3 = 0.6̄

↓

MinHash approx. of Jaccard Similarity

Agenda

- Crawler design
- Deduplication
- **Inverted index construction**
- Distributed search architecture

Serving results - speed

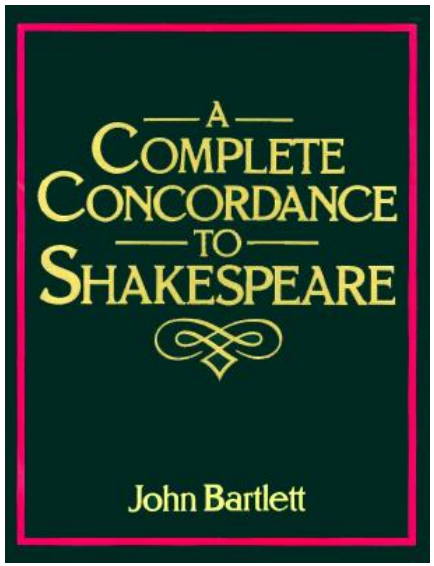
- After crawling is finished, we have a (big) database of documents
- Now we need to build an index

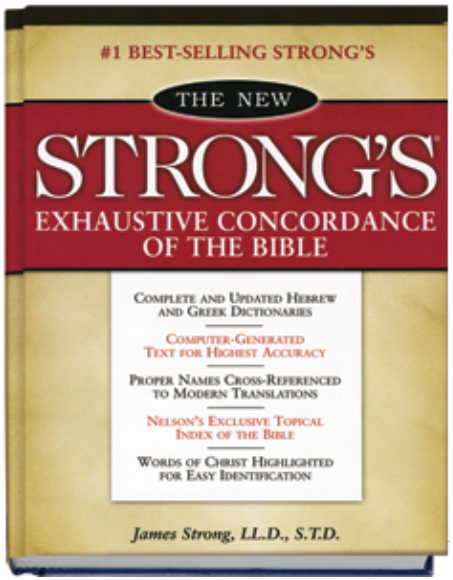
Inverted index

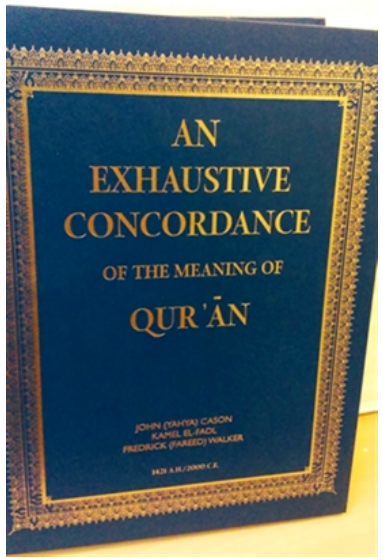
- A *forward index* is a list of words in each document
  - Doc -> words
- An *inverted index* maps words to docs that contain those words
  - Word -> docs
- For each word, list all the documents where that word can be found
- Key to fast query processing

Inverted index

- Inverted indexes were around before computers
- Example: concordance
- List of every word, in alphabetical order
- Constructed manually before computers!!!







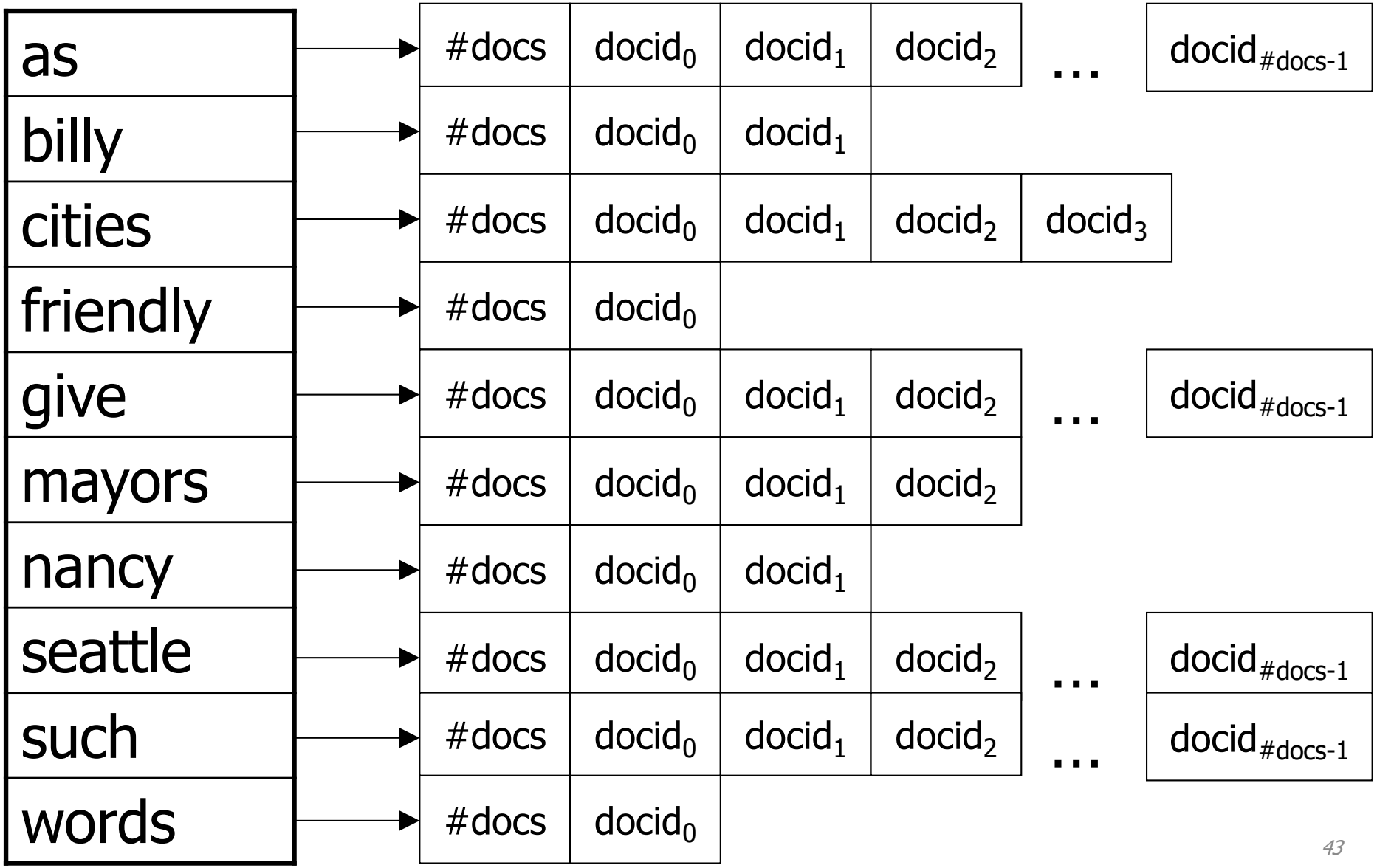
Inverted index

- openshakespeare.org concordance: `horatio`

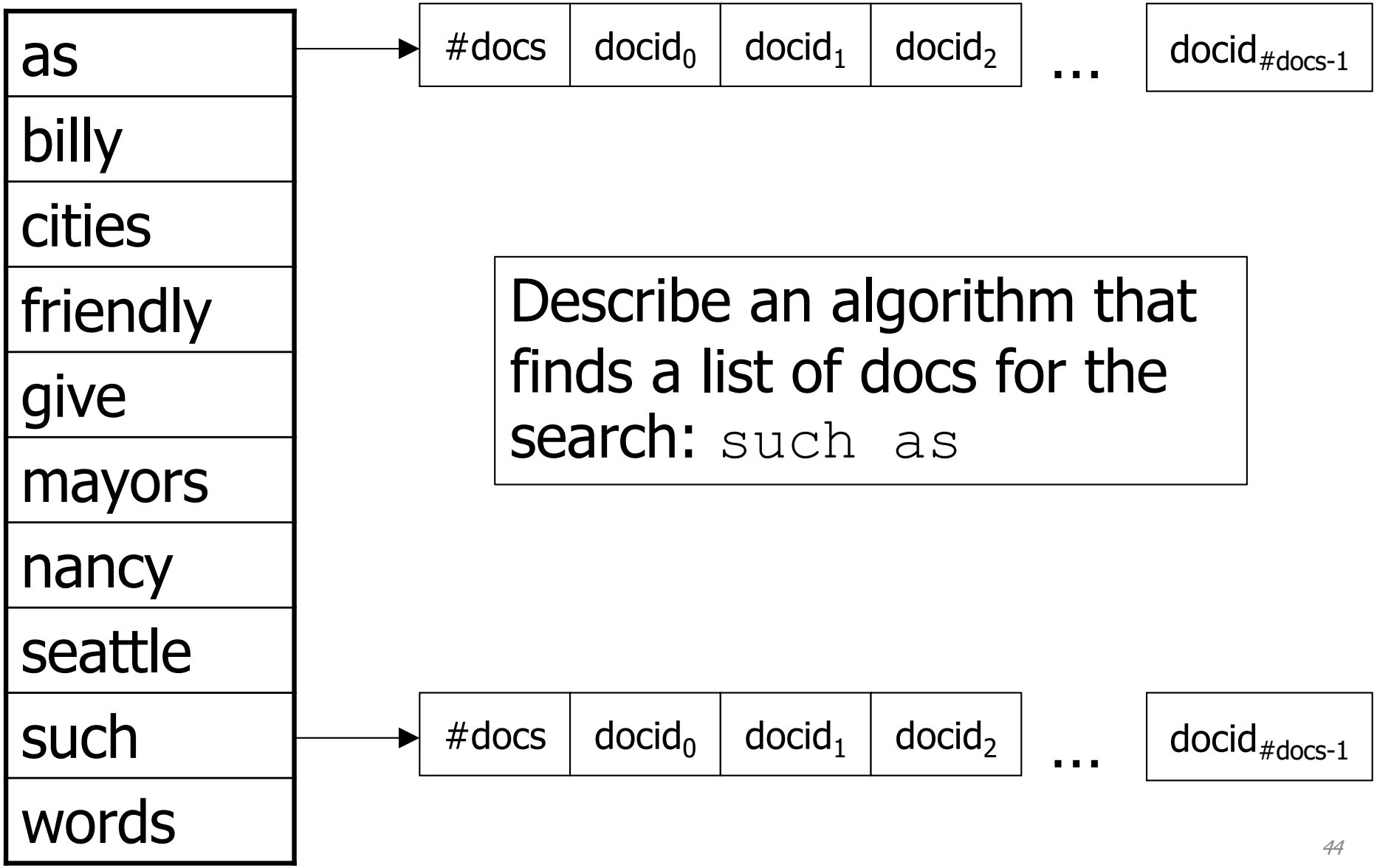
#	Work	Character	Line	Text
1	**Hamlet** [I, 1]	**Bernardo**	13	Well, good night. If you do meet **Horatio** and Marcellus, The rivals of my watch, bid them make haste.
2	**Hamlet** [I, 1]	**(stage directions)**	16	Enter **Horatio** and Marcellus.
3	**Hamlet** [I, 1]	**Bernardo**	26	Say- What, is **Horatio** there ?
4	**Hamlet** [I, 1]	**Bernardo**	29	Welcome, **Horatio**. Welcome, good Marcellus.
5	**Hamlet** [I, 1]	**Marcellus**	32	**Horatio** says 'tis but our fantasy, And will not let belief take hold of him Touching this dreaded sight, twice seen of us. Therefore I have entreated him along, With us to watch the minutes of this night, That, if again this apparition come, He may approve our eyes and speak to it.
6	**Hamlet** [I, 1]	**Marcellus**	54	Thou art a scholar; speak to it, **Horatio**.

http://www.opensourceshakespeare.org/search/search-results.php?link=con&searchtype=exact&hamlet&keyword1=horatio

Inverted index

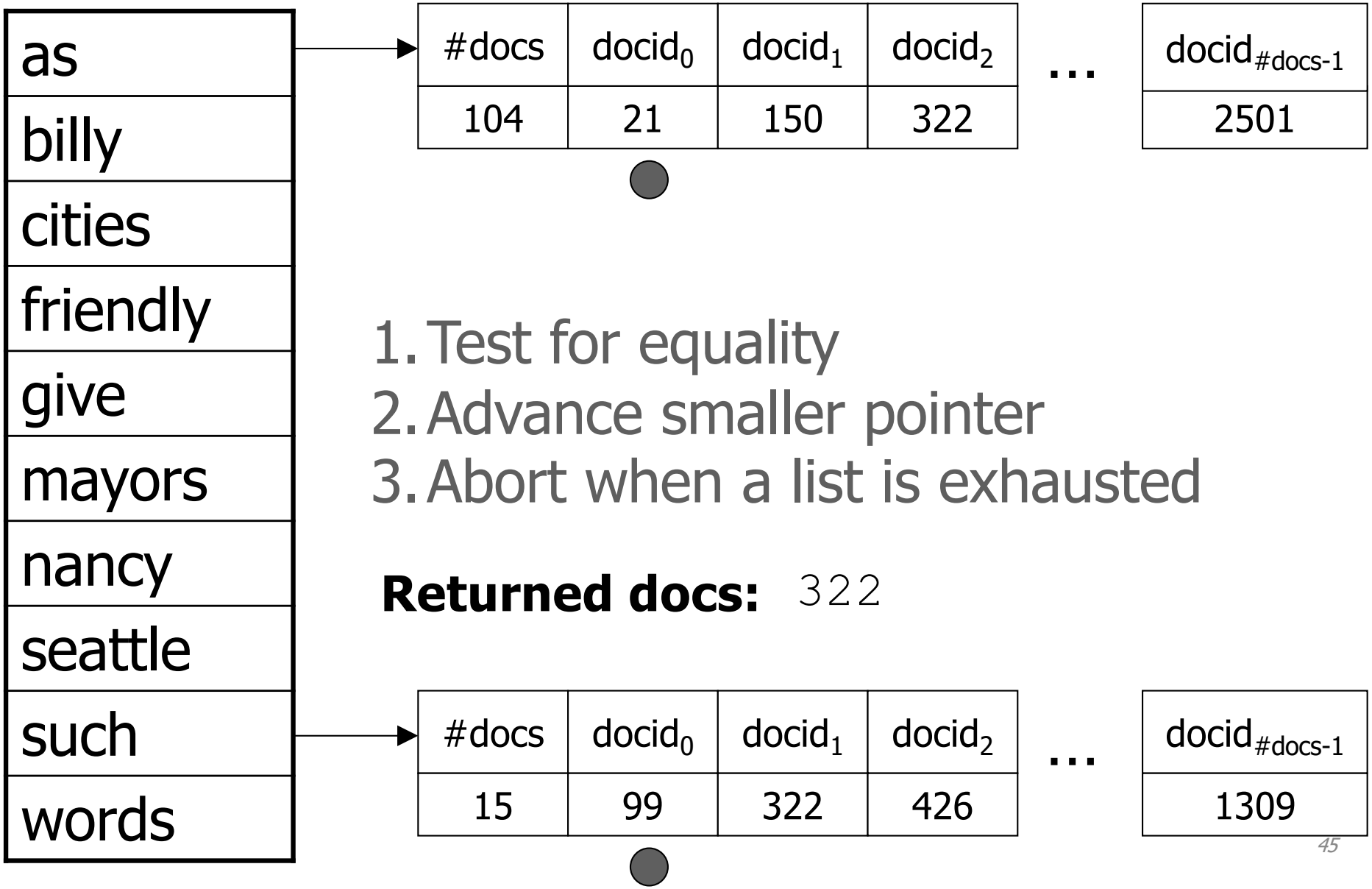


Inverted index exercise



Describe an algorithm that finds a list of docs for the search: `such as`

Solution: merge intersection



1. Test for equality
2. Advance smaller pointer
3. Abort when a list is exhausted

Returned docs: 322

Inverted index construction

- Inverted index is very large
  - Full text index usually larger than the text
  - How can we build it efficiently?
- Remember:
  - Magnetic disk seeks are very expensive (5ms)
  - Continuous disk reads or writes are OK (50-120MB/sec)
  - Machines can have a lot of memory or flash (can be TB), but magnetic disk is much cheaper per byte
  - Input is the tokenized document set

Basic tasks

1. Compile term-termid, doc-docid maps
2. Assemble all termid-docid pairs
3. Sort pairs first by termid, then docid
4. Write out in inverted-index form

- EASY!
- Well, not if docs won't fit into memory

Block sort-based indexing

- *External sort algorithms* work on sets larger than memory
- Block-Sort-Based Index Algorithm:

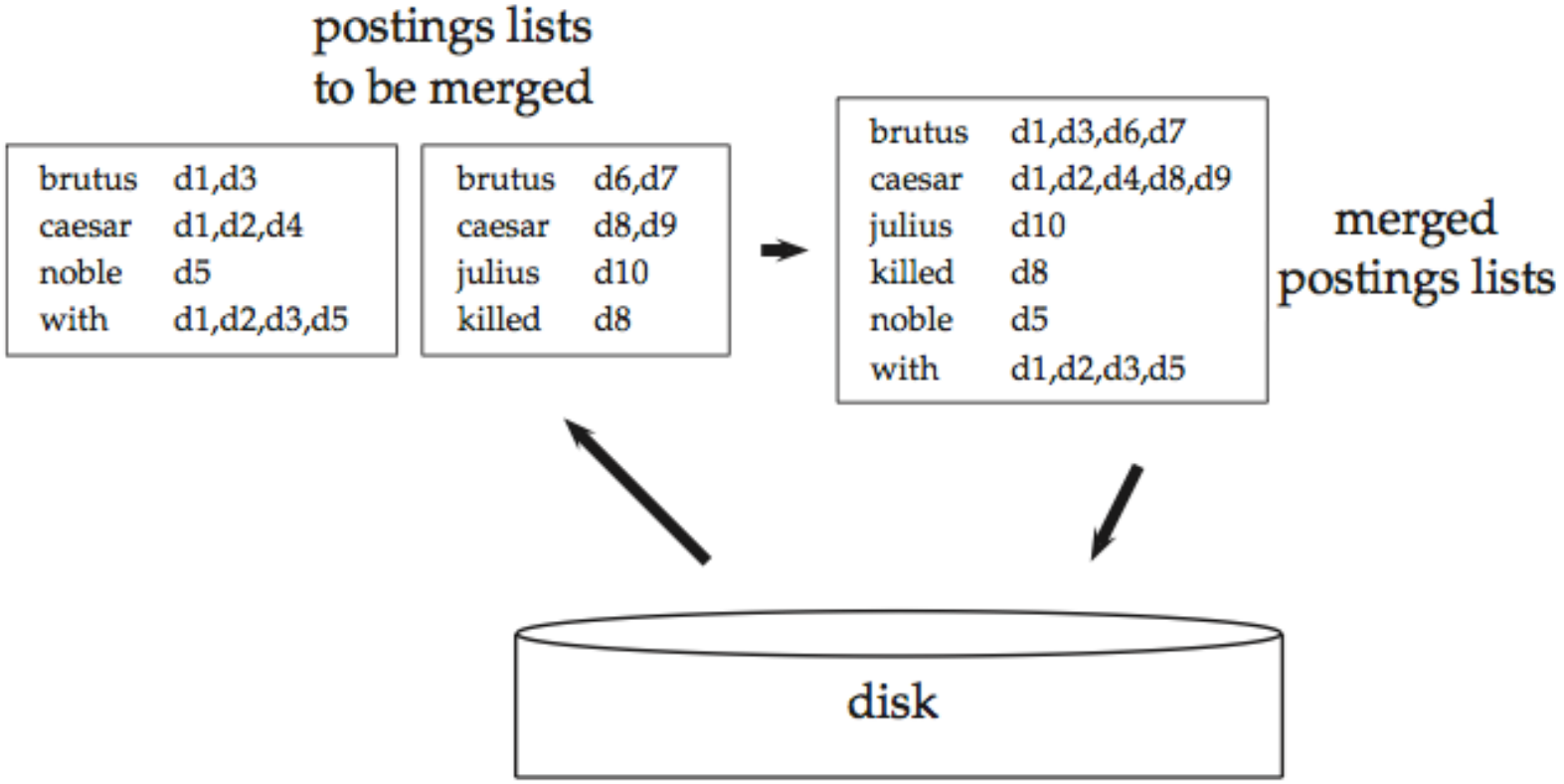
```
n = 0
while docsRemain
   n++
   block = ParseNextBlock()
   BSBI-Invert(block)
   WriteToDisk(block, f_n)
MergeBlocks(f_1, …, f_n) => f_merged
```

Block sort-based indexing

- `ParseNextBlock` accumulates termid-docid pairs in memory until block is full
- `BSBI-Invert` generates small in-memory inverted index

- So: we build a series of small in-memory inverted indexes, writing each one to disk
- Finally: we merge them

Block merging



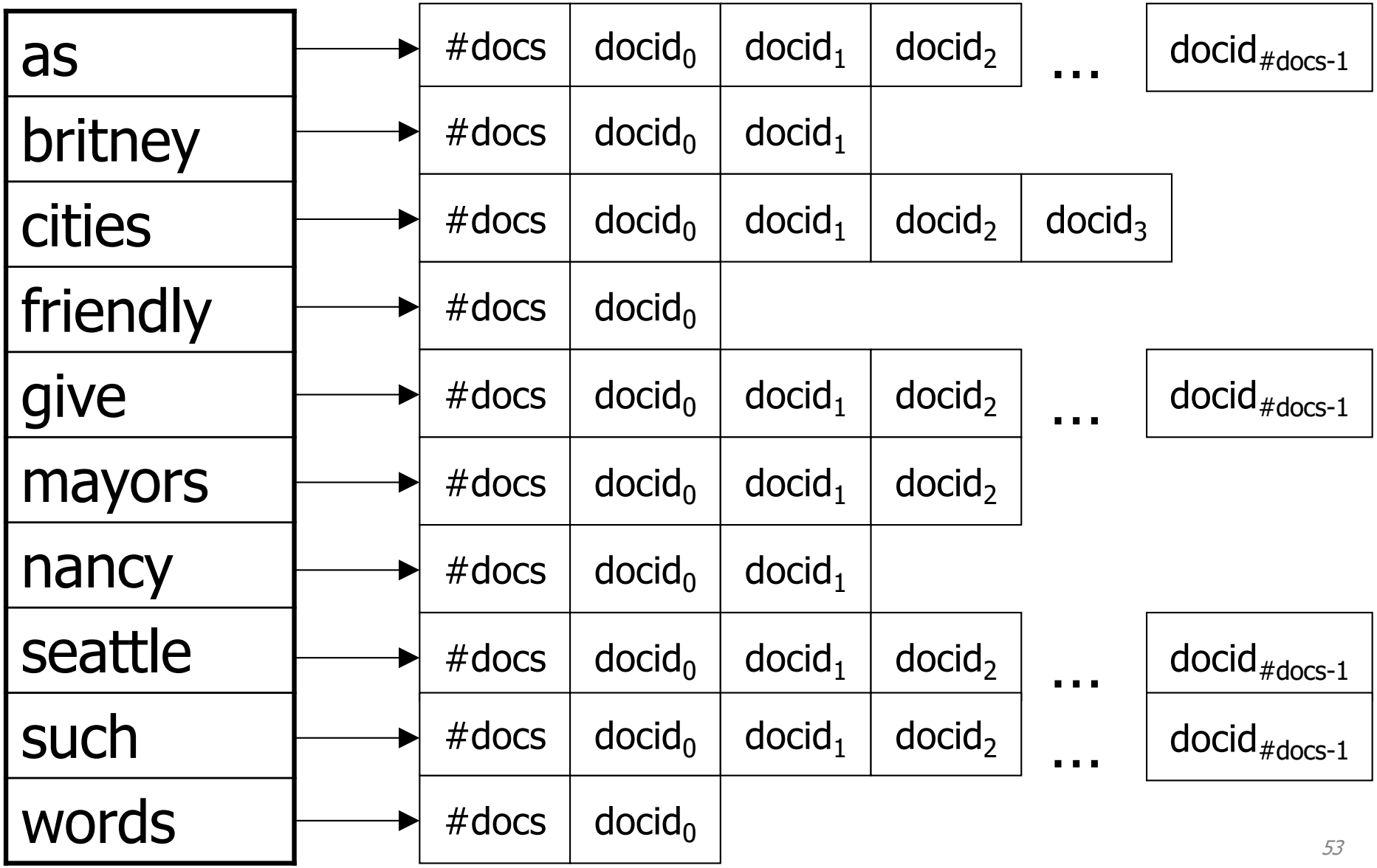
Agenda

- Crawler design
- Deduplication
- Inverted index construction
- **Distributed search architecture**

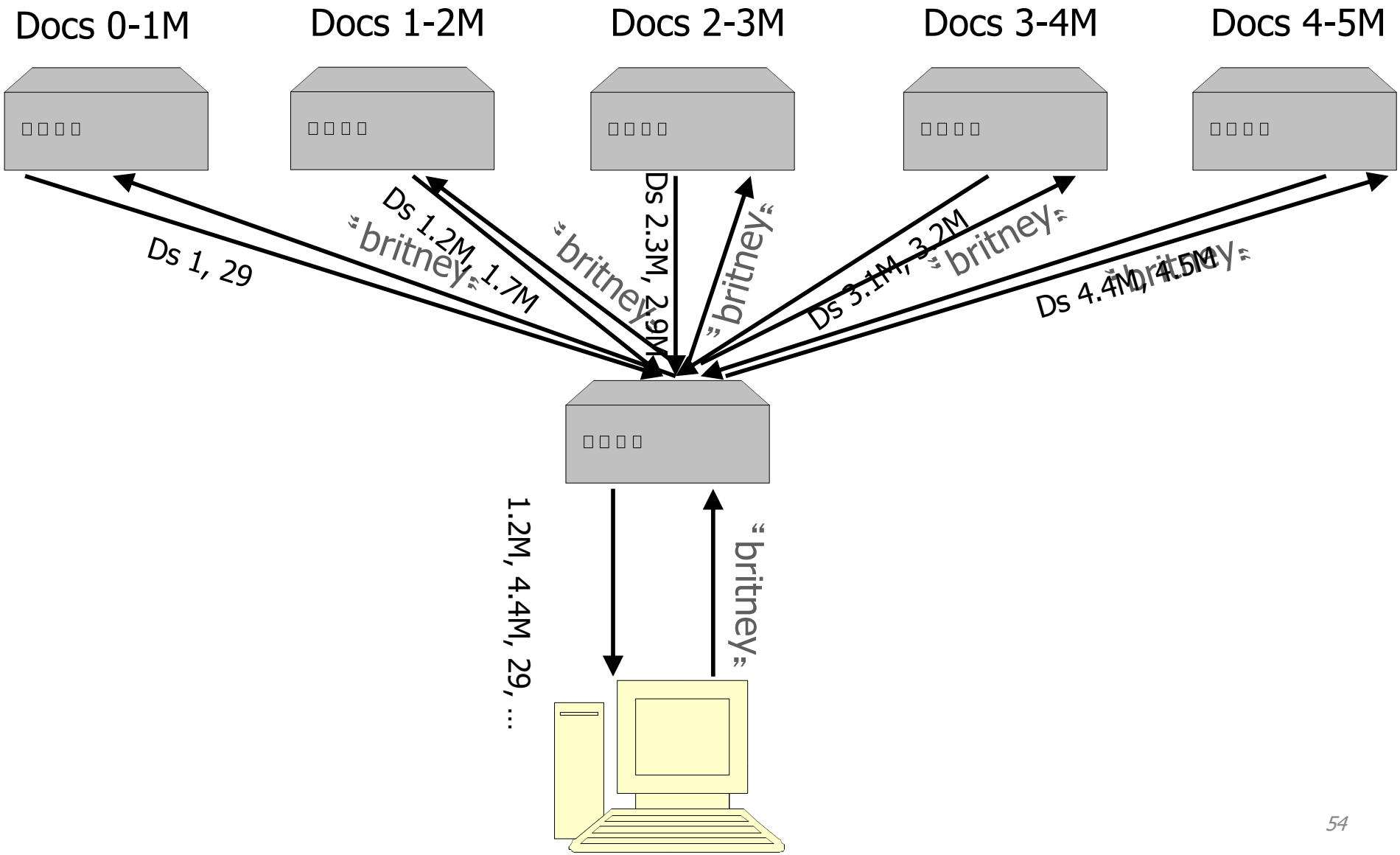
Distributed searching

- Not even the inverted index is small enough for one machine to handle it
  - Billions of docs
  - Hundreds of millions of queries
- Also, what if the machine fails?
- Need to parallelize query processing
  - Segment by document
  - Segment by search term

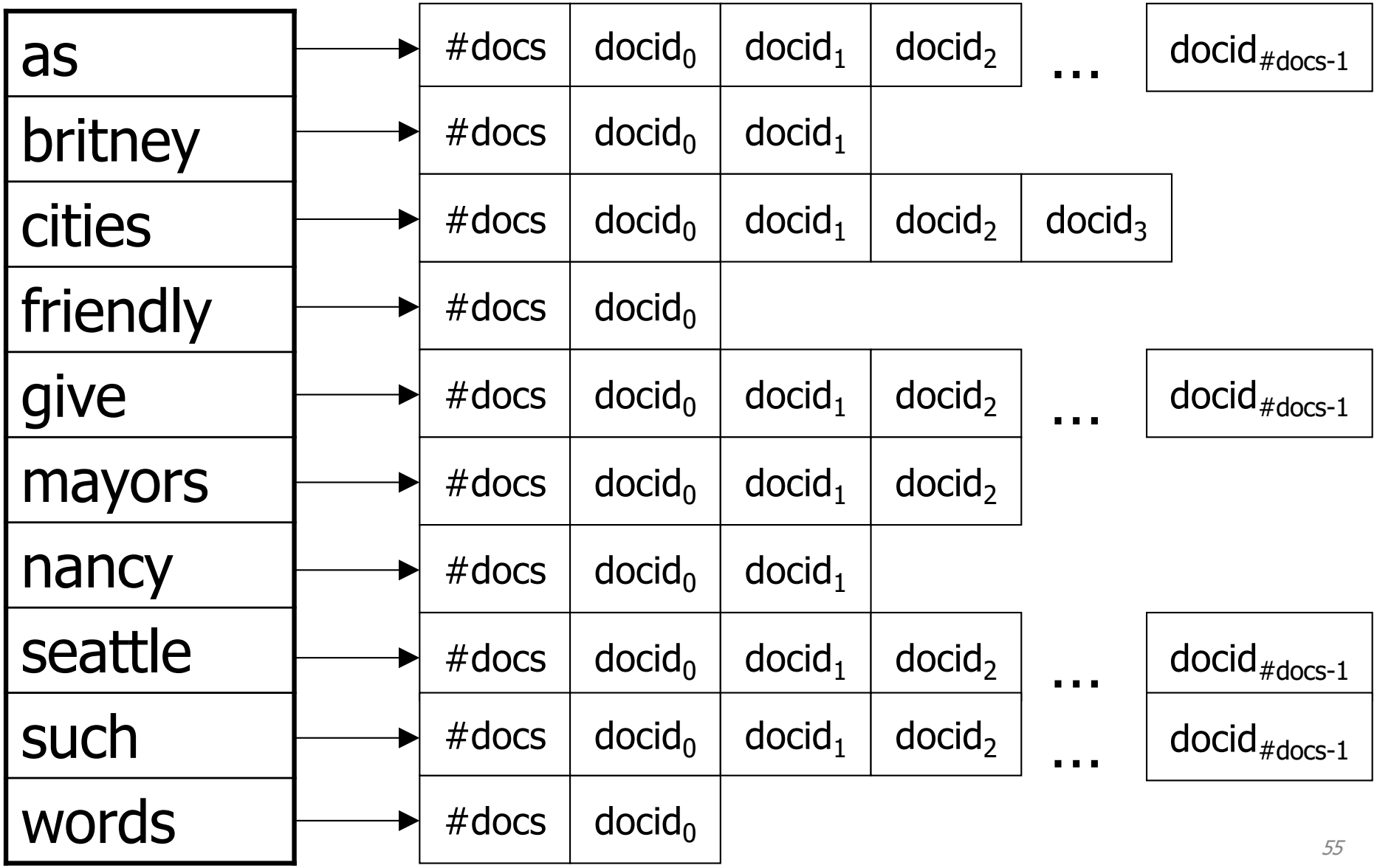
Segment by document (divide cols)



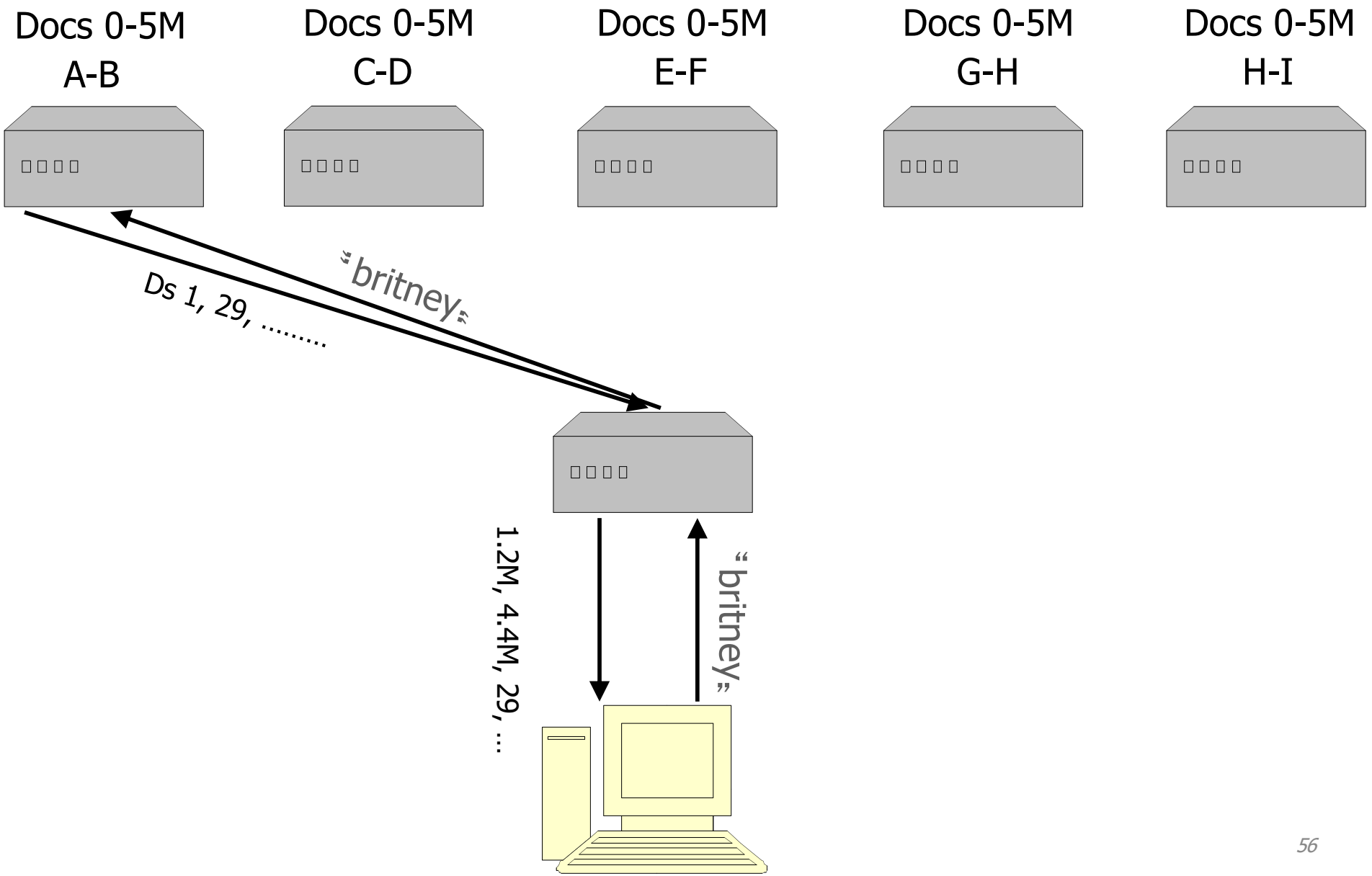
Segment by document



Segment by term (divide rows)



Segment by term



Segmentation

- What are the tradeoffs of segment-by-document vs. segment-by-term?
- What happens if a machine dies?

Segmentation

- Segment by document
  - Easy to partition (just MOD the docid)
  - Easy to add new documents
  - If machine fails, quality goes down but queries don't die
- Segment by term
  - Harder to partition (terms uneven)
  - Trickier to add a new document (need to touch many machines)
  - If machine fails, search term might disappear, but not critical pages (e.g., cnn.com/index.html)

Conclusions

- Information retrieval is the heart of web search
- Document vector model for page content
- Network model for link analysis
- Metrics: precision, recall, Kendall's Tau, Mean Reciprocal Rank
- Implementation: crawler, deduplication, inverted index

- Basis for project 5
- Semester-long course in EECS 486