

EECS 395/495 Lecture 5: Web Crawlers

Doug Downey

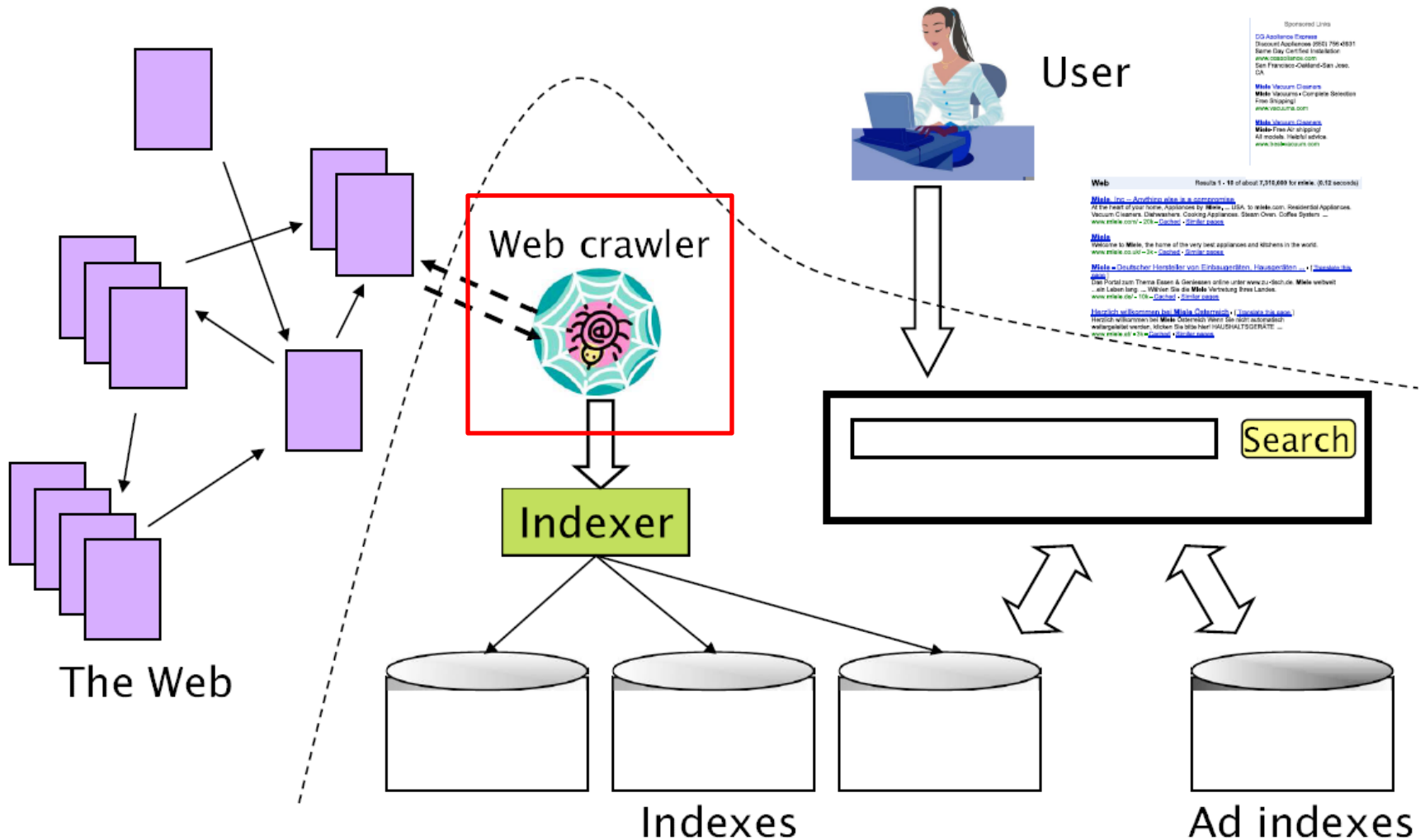
Interlude: US Searches per User

Year	Searches/month (mlns)	Internet Users (mlns)	Searches/user-month
2008	10800	220	49.1
2009	14300	227	63.0
2010	15400	239	64.4
2011	16900	245	69.0
2012	18400	(est) 252	73.0

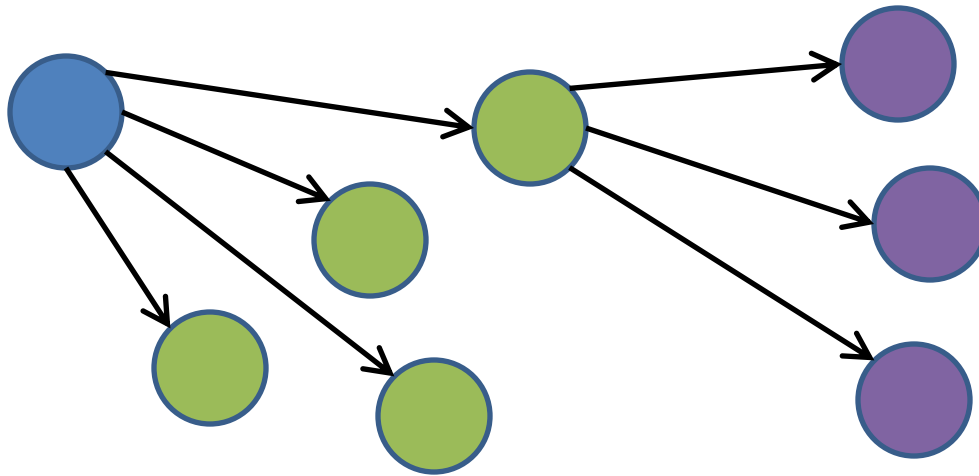
How Does Web Search Work?

- Inverted Indices
- **Web Crawlers**
- Ranking Algorithms

Web Crawlers



Crawling



$\text{Crawl}(urls = \{p_1, \dots, p_n\})$

Retrieve some p_i from $urls$

$urls -= p_i$

$urls += p_i$'s links

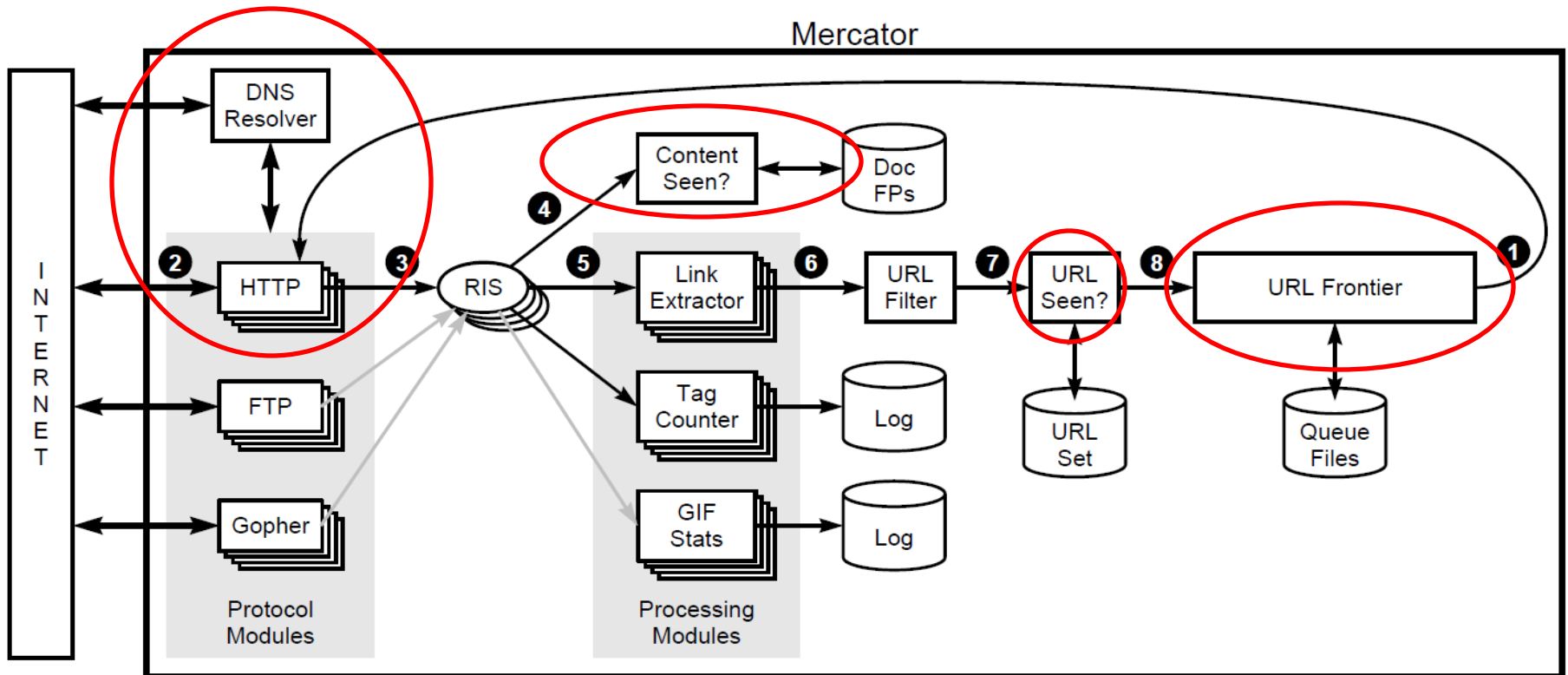
Questions:

Breadth-first or depth-first?

Where to start?

How to scale?

Basic Crawler Design



Our discussion mostly based on:

Mercator: A Scalable, Exensible Web Crawler (1999)

by Allan Heydon, Marc Najork

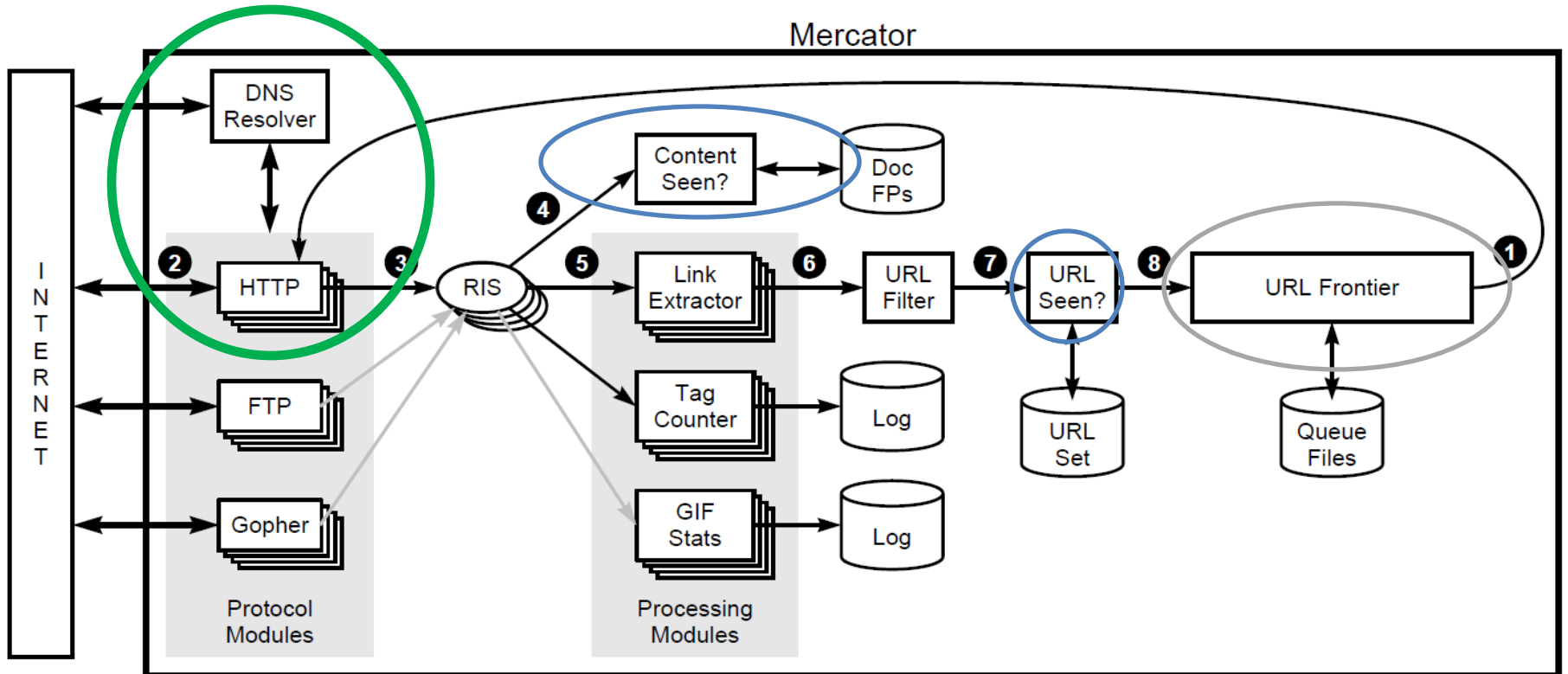
URLs: The Final Frontier

- URL frontier
 - Stores urls to be crawled
 - Typically breadth-first (LIFO queue)
- Mercator uses a *set* of subqueues
 - One subqueue per Web-page-fetching thread
 - When a new url is enqueued, the destination subqueue is based on host name
 - So access to a given host is serial
 - For politeness and bottleneck-avoidance

Etiquette

- Robots.txt
 - Tells you which parts of site you shouldn't crawl
 - Web-page fetchers cache this for each host

Basic Crawler Design



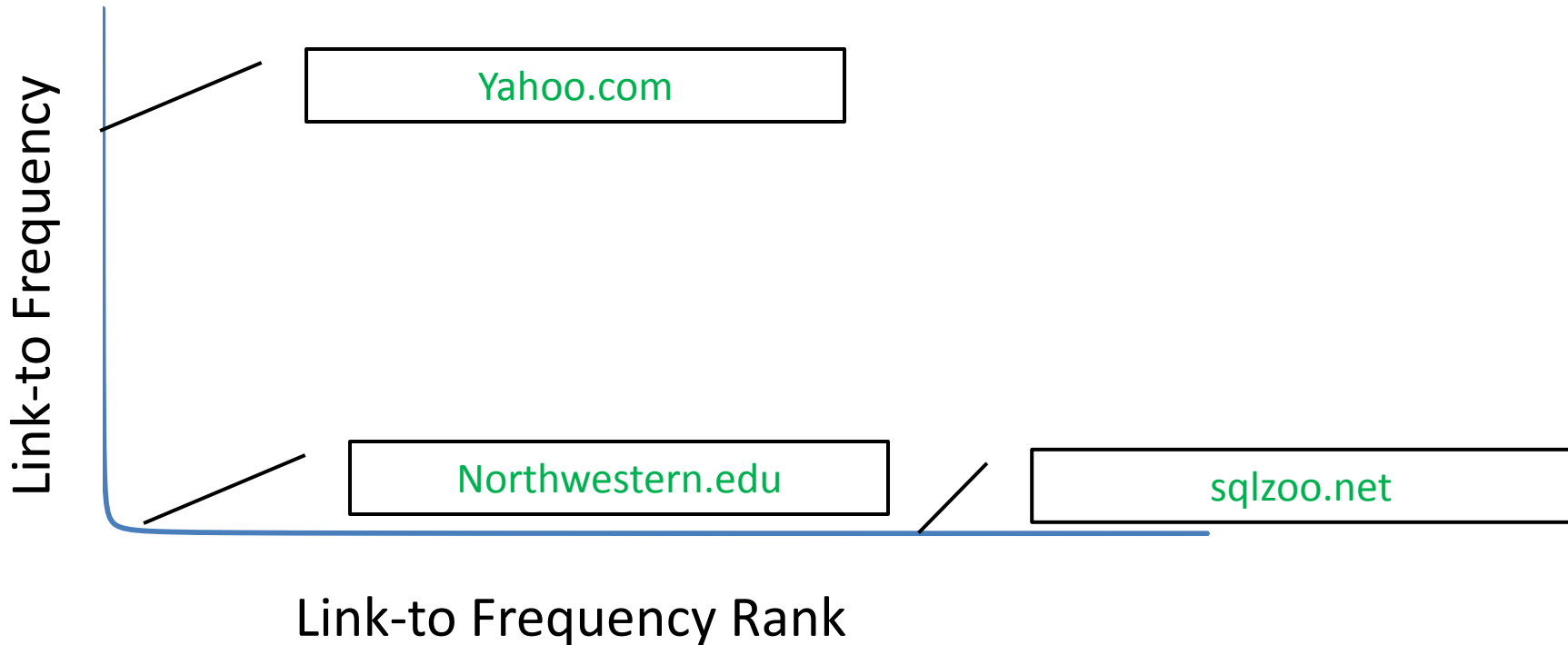
Mercator: A Scalable, Exensible Web Crawler (1999)
by Allan Heydon, Marc Najork

HTTP/DNS

- Multiple Web-page-fetching threads download documents from specified host(s)
- Domain Name Service
 - Map host to IP address
 - www.yahoo.com -> 209.191.93.52
 - Well known bottleneck of crawlers
 - Exacerbated by synchronized interfaces
 - Solution: caching, asynchronous access

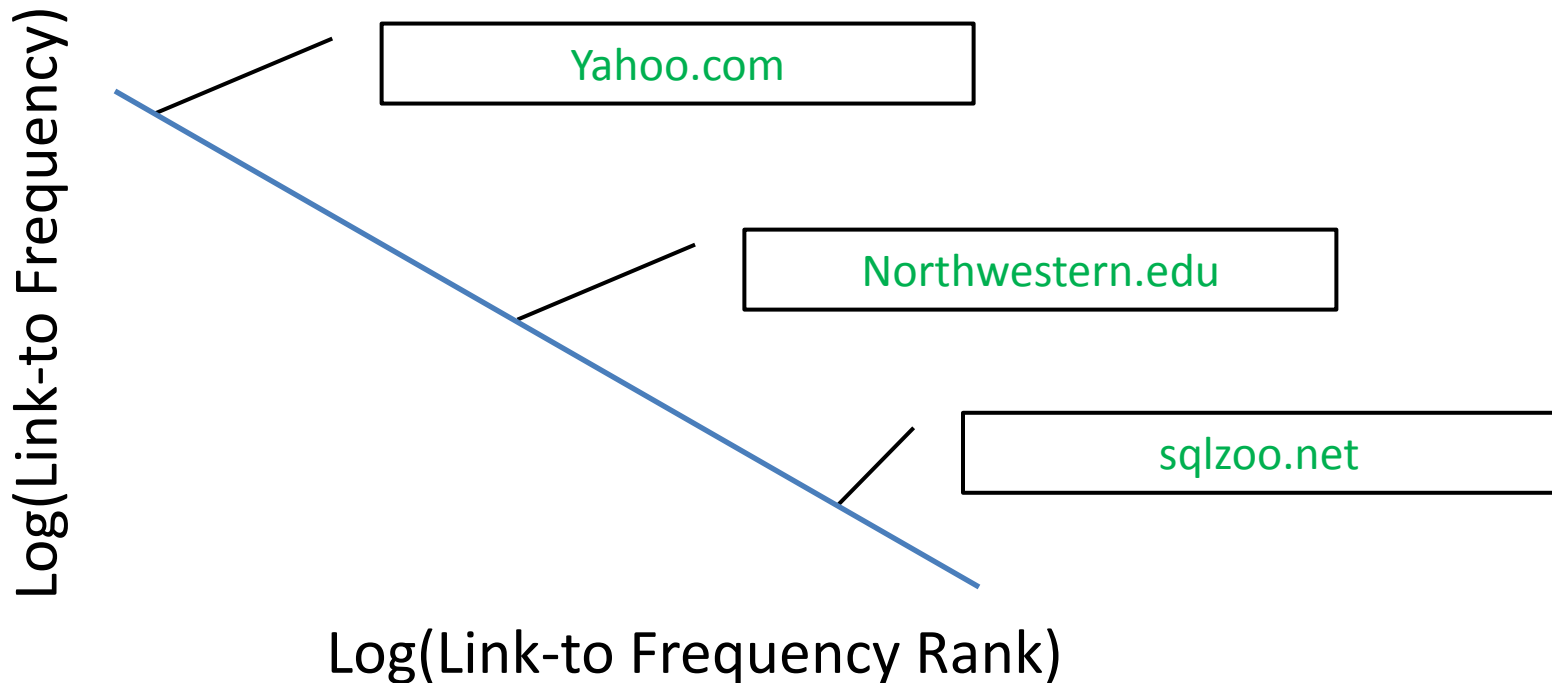
Caching Web Data

- Zipf Distribution
 - Freq(*i*th most frequent element) $\propto i^{-z}$



Caching Web Data

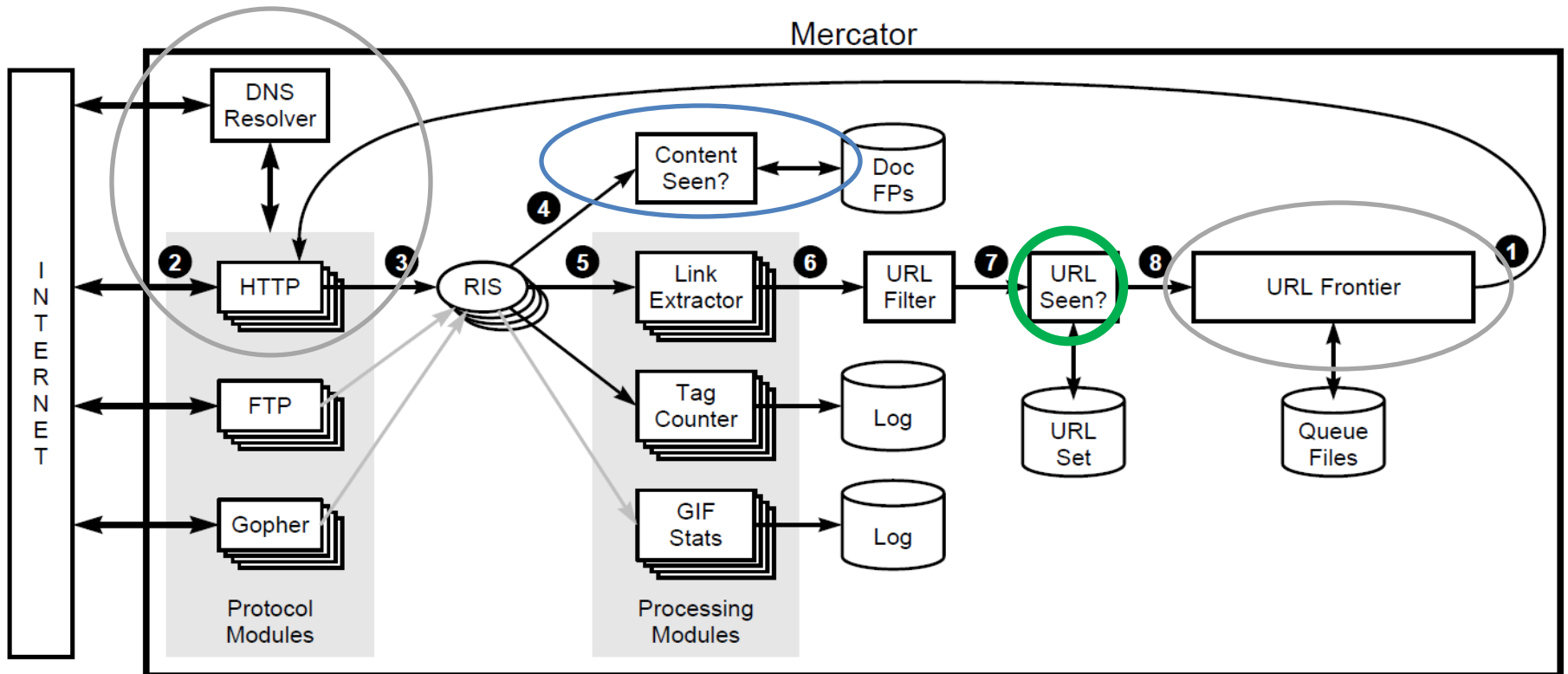
- Zipf Distribution
 - $\log \text{Freq}(i\text{th most frequent element}) \propto -z \log i$



Caching Web Data

- Thus:
 - Caching several hundred thousand hosts in memory saves a large number of disk seeks
 - Caching the rest on disk saves many DNS requests
- “Zipf’s Law” also applies to:
 - Web page accesses, term frequency, link distribution (in-degree and out-degree), search query frequency, etc.
 - City sizes, incomes, earthquake magnitudes...

Basic Crawler Design



Mercator: A Scalable, Exensible Web Crawler (1999)
by Allan Heydon, Marc Najork

URL Seen Test

- *Huge* number of duplicate hyperlinks
 - Est. 20x number of pages
- Goal: keep track of which URLs you've downloaded
- Problem: Lots of data
 - So use hashes

Brief Review of Hashing

- Hash function
 - Maps a data object (e.g., a URL) to a fixed-size binary representation
 - Mercator used Rabin's *fingerprinting* algorithm
 - For n strings of length $< m$, and fingerprint of length k

$$P(\text{two strings have same representation}) \leq nm^2 / 2^k$$

Some applications of Rabin's fingerprinting method
[Broder, 1993]

Hashing for URL Seen Test

- For urls of length < 1000 and 20,000,000,000 Web pages, 128-bit hashes:

$$P(\text{any two strings have same representation}) \leq nm^2 / 2^k \\ \approx 10^{-15}$$

- Tiny chance of *any* collisions
- Thus: just store & check the *hashes*
 - Space savings: **12x** (assuming avg. 100 bytes/url)
 - Also translates to efficiency gains

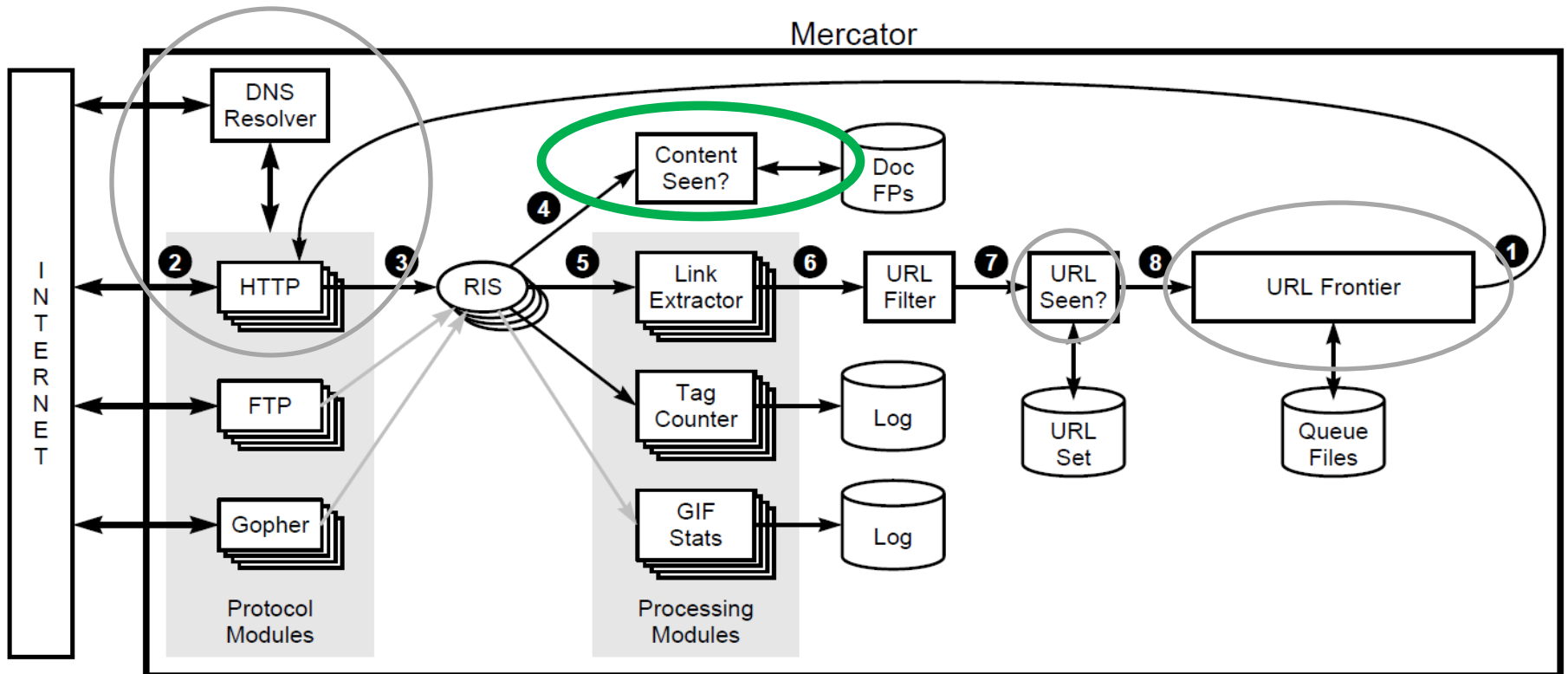
Storing/Querying URLs Seen

- We have a list of URL hashes we've seen
 - Smaller than text urls, but still large
 - 20B urls * 16 bytes/url = **320GB**
- How do we store/query it?
 - Store it in sorted form (with in-memory index)
 - Query with binary search

Wrinkles

- Store *two* hashes instead of one
 - One for host name, one for rest of url
 - Store as <host name hash><rest of url hash>
 - Why?
 - Exploit disk buffering
- Also use an in-memory cache of popular URLs
- In Mercator, all this together results in about **1/6** seek and read ops per URL Seen test

Basic Crawler Design



Mercator: A Scalable, Exensible Web Crawler (1999)

by Allan Heydon, Marc Najork

Content Seen?

- Many different URLs contain the same content
 - One website under multiple host names
 - Mirrored documents
 - >20% of Web docs are duplicates/near-duplicates



"affordable discount orlando hotels are just a

Search

[Advanced Search](#)
[Preferences](#)

Web

Results 11 - 11 of 11 for "[affordable discount orlando](#)

[Orlando Discount Travel](#)

Affordable discount Orlando hotels are just a click away! ... BookIt.com® Travel Specials:
Discounted Hotels, Resorts & Vacation Deals ...

www.free-vacations-travel.com/deals/orlando-discount-travel.php - [Similar pages](#)

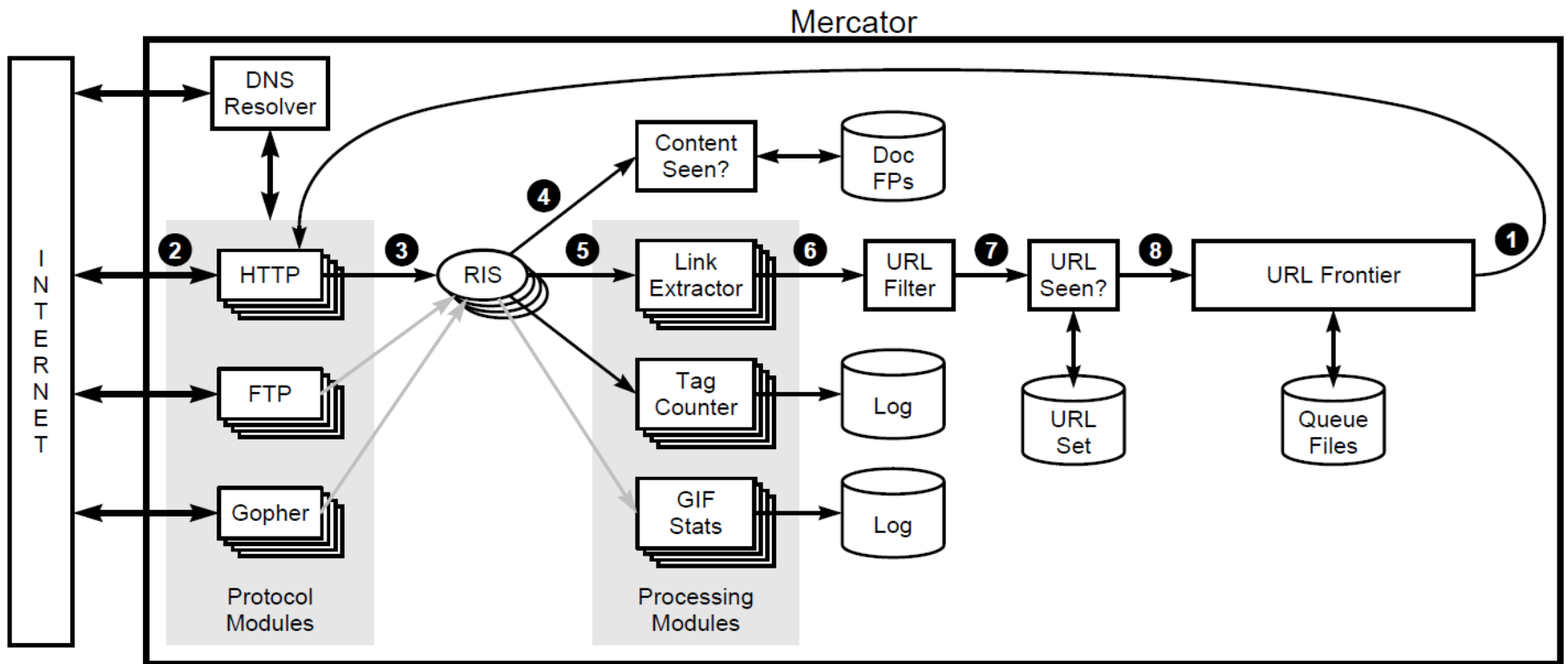
*In order to show you the most relevant results, we have omitted some entries very similar to the 11 already displayed.
If you like, you can [repeat the search with the omitted results included](#).*

How to detect?

- Naïve: store each page's content and check
- Better: use hashes
 - Mercator does this
- Some issues
 - Poor cache locality
 - What about *similar* pages?

Alternative Technique

- Compute binary feature vectors for each doc
 - E.g. term incidence vectors
<1:1, 192:1, 4002:1, 4036:1, ...>
 - Generate 100 random *permutations* of the vectors
e.g. **1->40002, 2->5, 3->1, 4->2031, ...**
 - For each document **D**, store a vector **v_D** containing the *minimum* feature for each permutation.
 - Compare these representations
 - Much smaller than original feature vector
 - ...but comparison is still expensive (see later techniques)



- Next time: Document ranking & Link Analysis