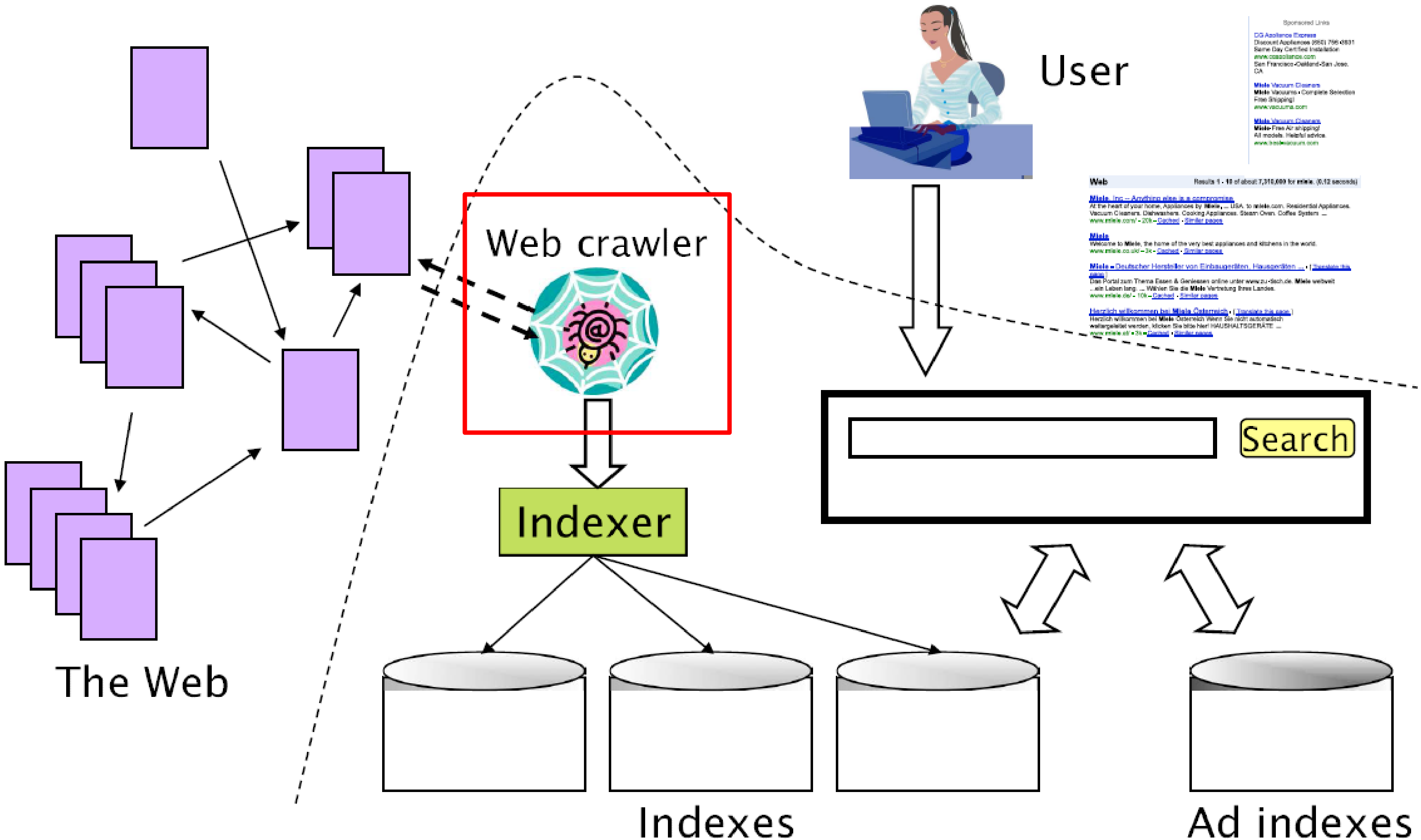
EECS 395/495 Lecture 5: Web Crawlers

Doug Downey

How Does Web Search Work?

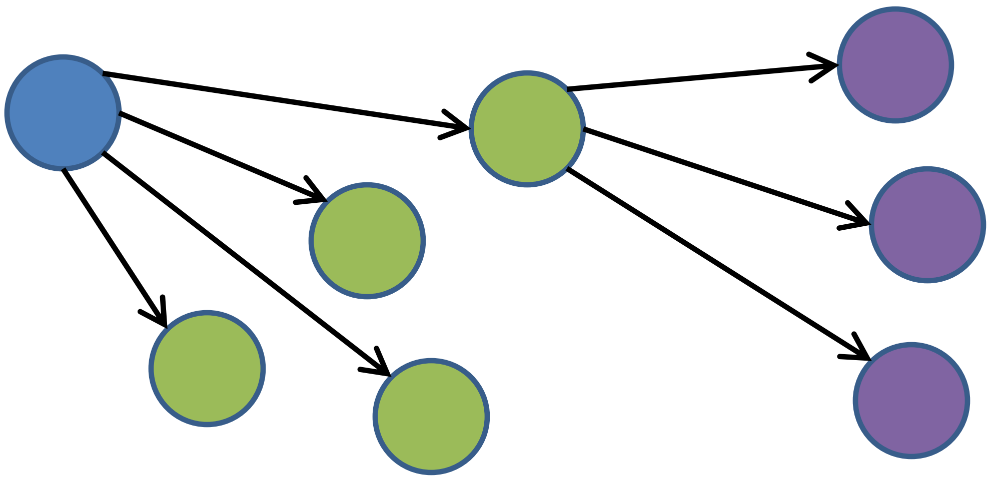
- Inverted Indices
- **Web Crawlers**
- Ranking Algorithms

Web Crawlers



http://www-csli.stanford.edu/~hinrich/information-retrieval-book.html

Crawling



Crawl(*urls* = {p_1,…p_n})

Retrieve some p_i from *urls*

urls -= p_i

urls += *p*’s links

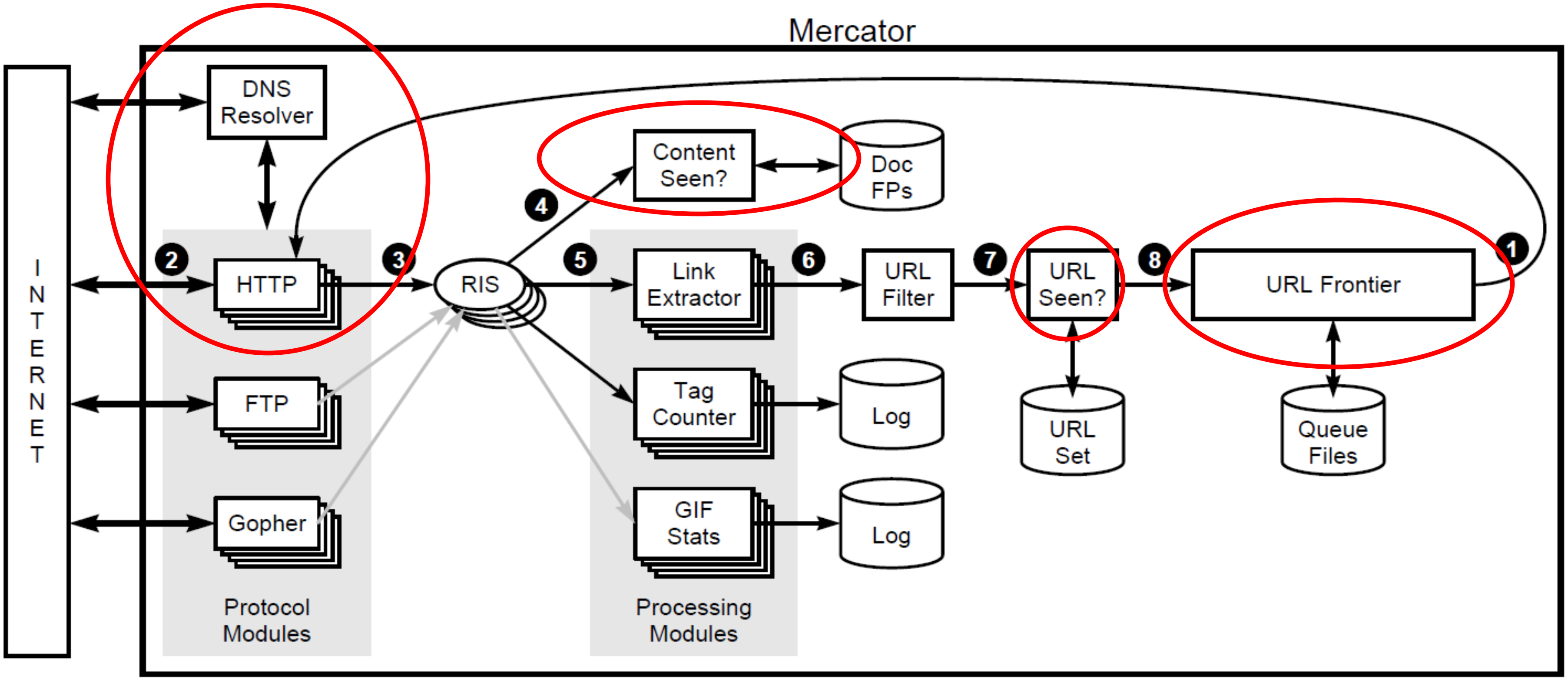
Questions:

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

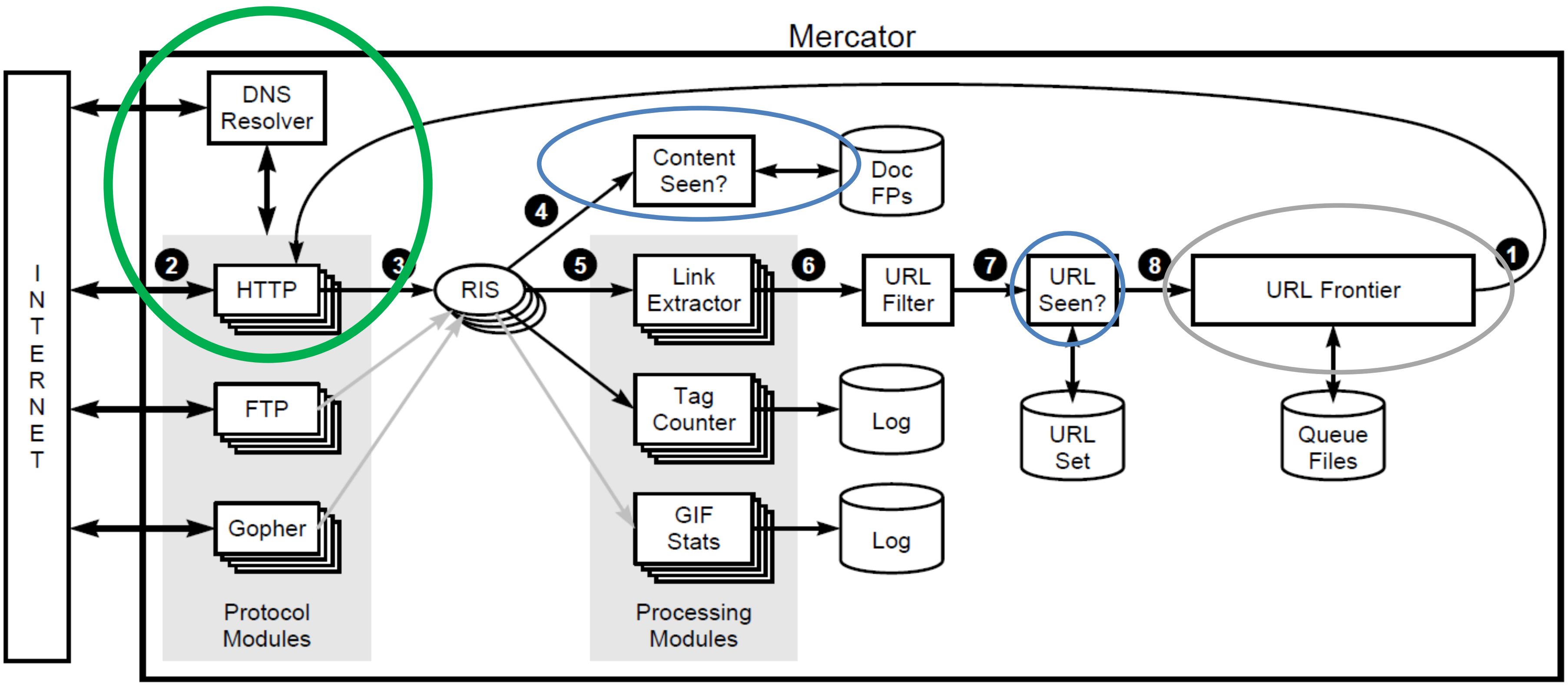
URLs: The Final Frontier

- 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

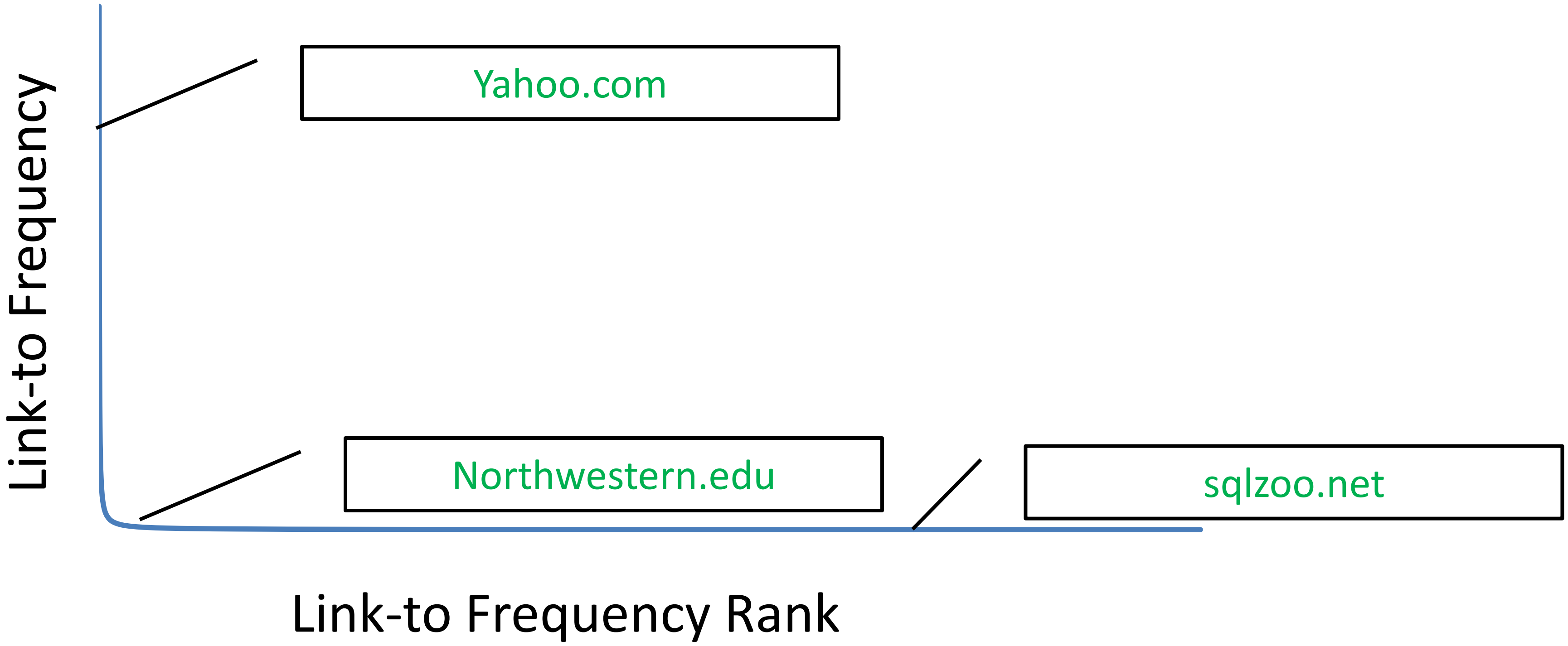
HTTP/DNS

- 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

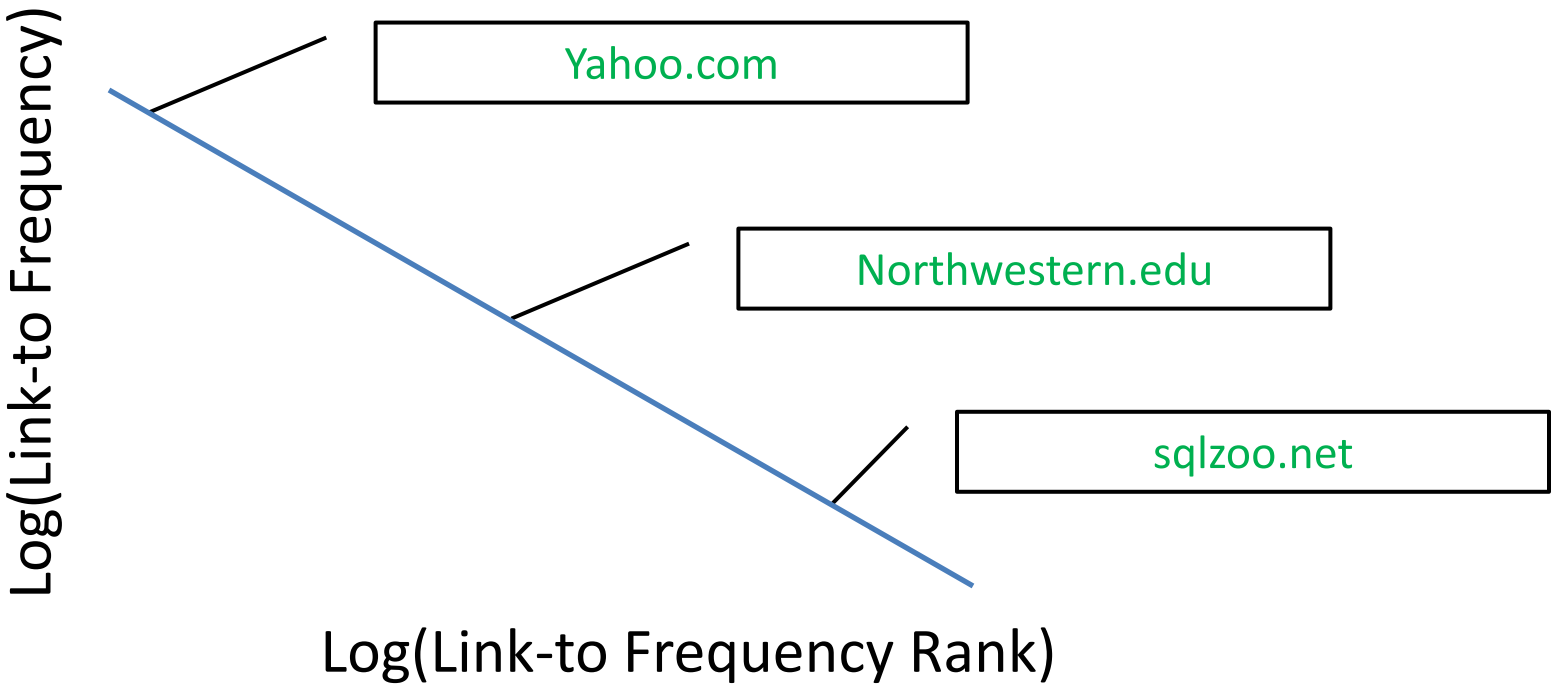
Caching Web Data

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



Caching Web Data

- Zipf Distribution
  - log Freq(ith most frequent element) $\propto$ $-z$ log i

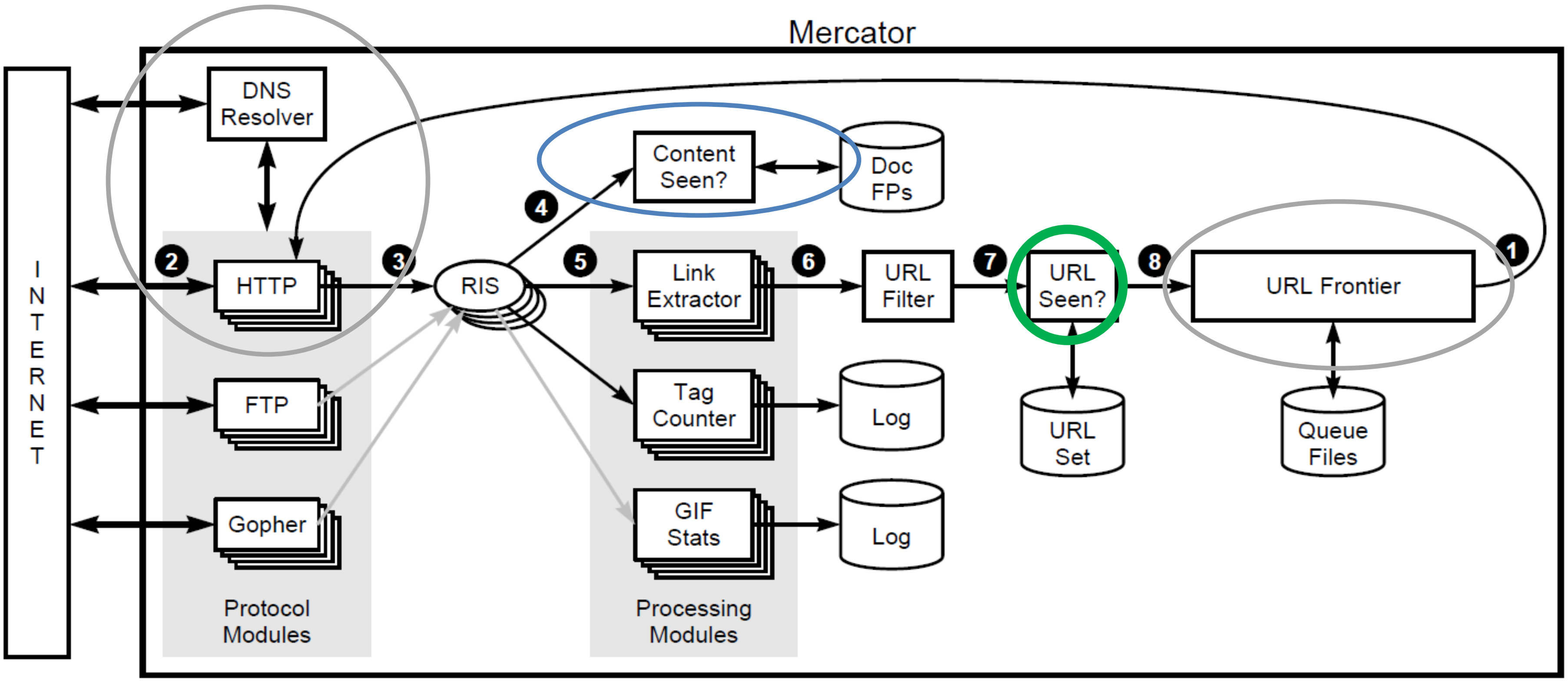


Caching Web Data

- 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 a Rabin's *fingerprinting* algorithm
    - For n strings of length < m, and fingerprint of length k

      P(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, 64-bit hashes:

  P(*any* two strings have same representation) $\leq nm^2 / 2^k$
  ≈ 0.001

- About 1/1000 chance of *any* collisions even using today's numbers
- Thus: just store & check the *hashes*
  - Space savings: **12x** (assuming avg. 100 bytes/url)
  - Also translates to efficiency gains

Storing/Querying URLs Seen

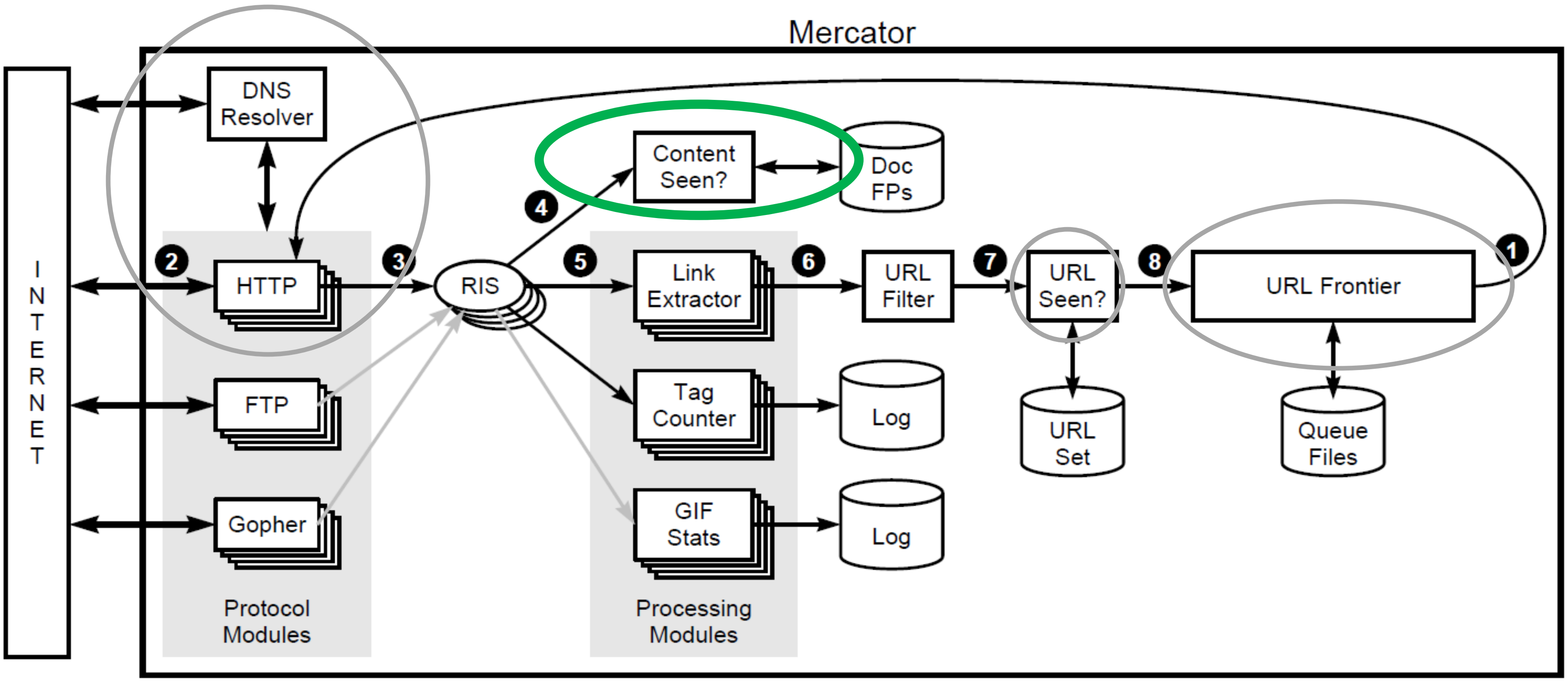
- We have a list of URL hashes we've seen
  - Smaller than text urls, but still large
  - 20B urls * 8 bytes/url = **160GB**

- 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

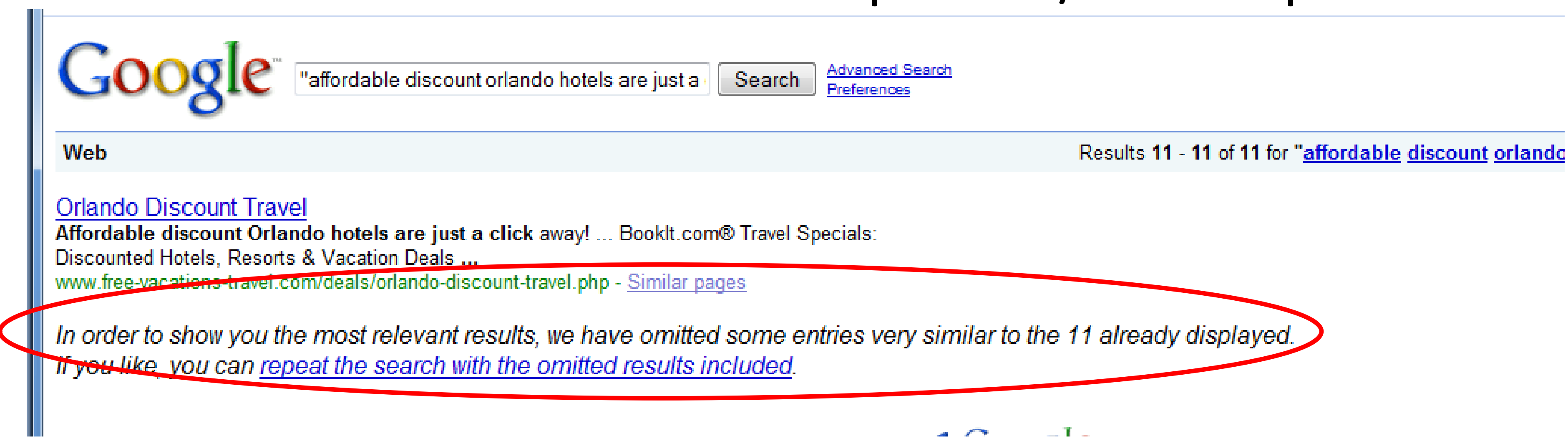
Basic Crawler Design



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

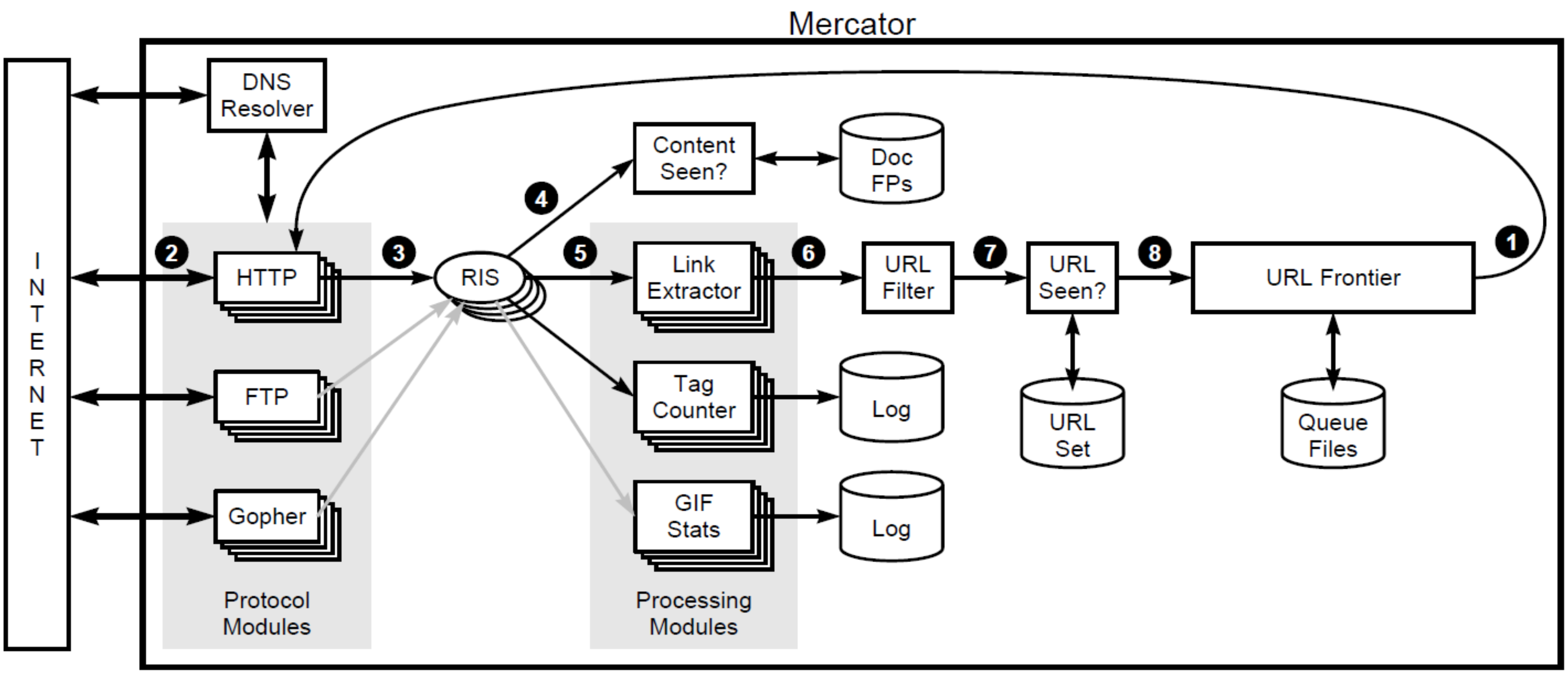


How to detect?

- 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 $\mathbf{v_D}$ 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