Introduction to

Information Retrieval

Crawling and Duplicates

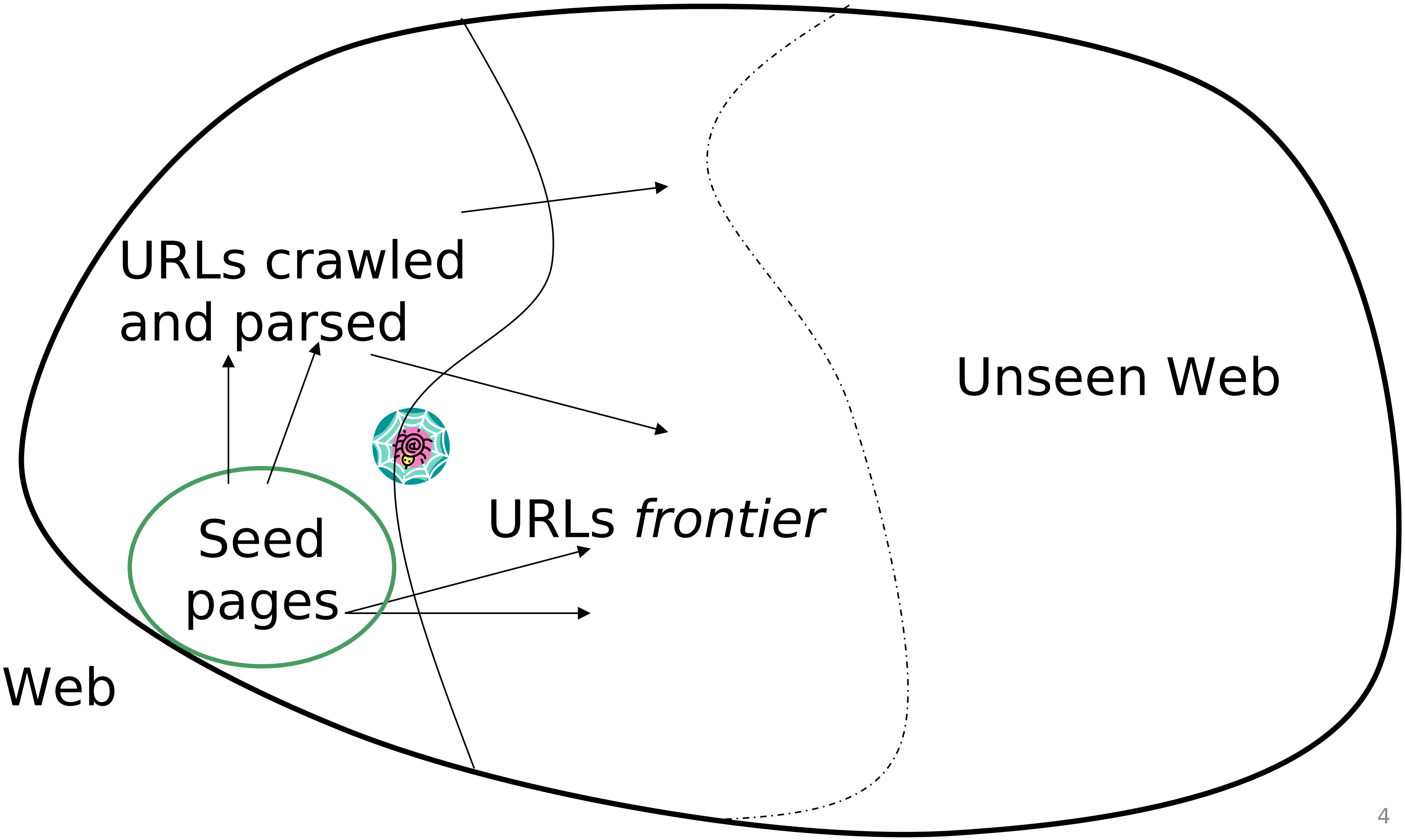
This lecture

- Web Crawling
- (Near) duplicate detection

Basic crawler operation

- Begin with known “seed” URLs
- Fetch and parse them
  - Extract URLs they point to
  - Place the extracted URLs on a queue
- Fetch each URL on the queue and repeat
- Breadth First crawling

Crawling picture



Simple picture – complications

- Web crawling isn’t feasible with one machine
  - All of the above steps are usually distributed
- Malicious pages
  - Spam pages
  - Spider traps
- Even non-malicious pages pose challenges
  - Latency/bandwidth to remote servers vary
  - Webmasters’ stipulations
    - How “deep” should you crawl a site’s URL hierarchy?
  - Site mirrors and duplicate pages
- Politeness – don’t hit a server too often

What any crawler *must* do

- Be Polite: Respect implicit and explicit politeness considerations

- Be Robust: Be immune to spider traps and other malicious behavior from web servers

Explicit and implicit politeness

- Explicit politeness: specifications from webmasters on what portions of a site can be crawled
  - robots.txt (see next slide)

- Implicit politeness: even with no specification, avoid hitting any site too often

Robots.txt

- Protocol for giving spiders ("robots") limited access to a website, originally from 1994
  - www.robotstxt.org/robotstxt.html
- Website announces its request on what can(not) be crawled
  - For a server, create a file `/robots.txt`
  - This file specifies access restrictions

What any crawler *should* do

- Be capable of distributed operation: designed to run on multiple distributed machines
- Be scalable: designed to increase the crawl rate by adding more machines
- Performance/efficiency: permit full use of available processing and network resources

What any crawler *should* do

- Fetch pages of “higher quality” first
- Continuous operation: Continue fetching fresh copies of a previously fetched page
- Extensible: Adapt to new data formats, protocols

URL frontier

- URLs that have been discovered, but are yet to be crawled
- Can include multiple pages from the same host
- Must avoid trying to fetch them all at the same time
- Must try to keep all crawling threads busy

Processing steps in crawling

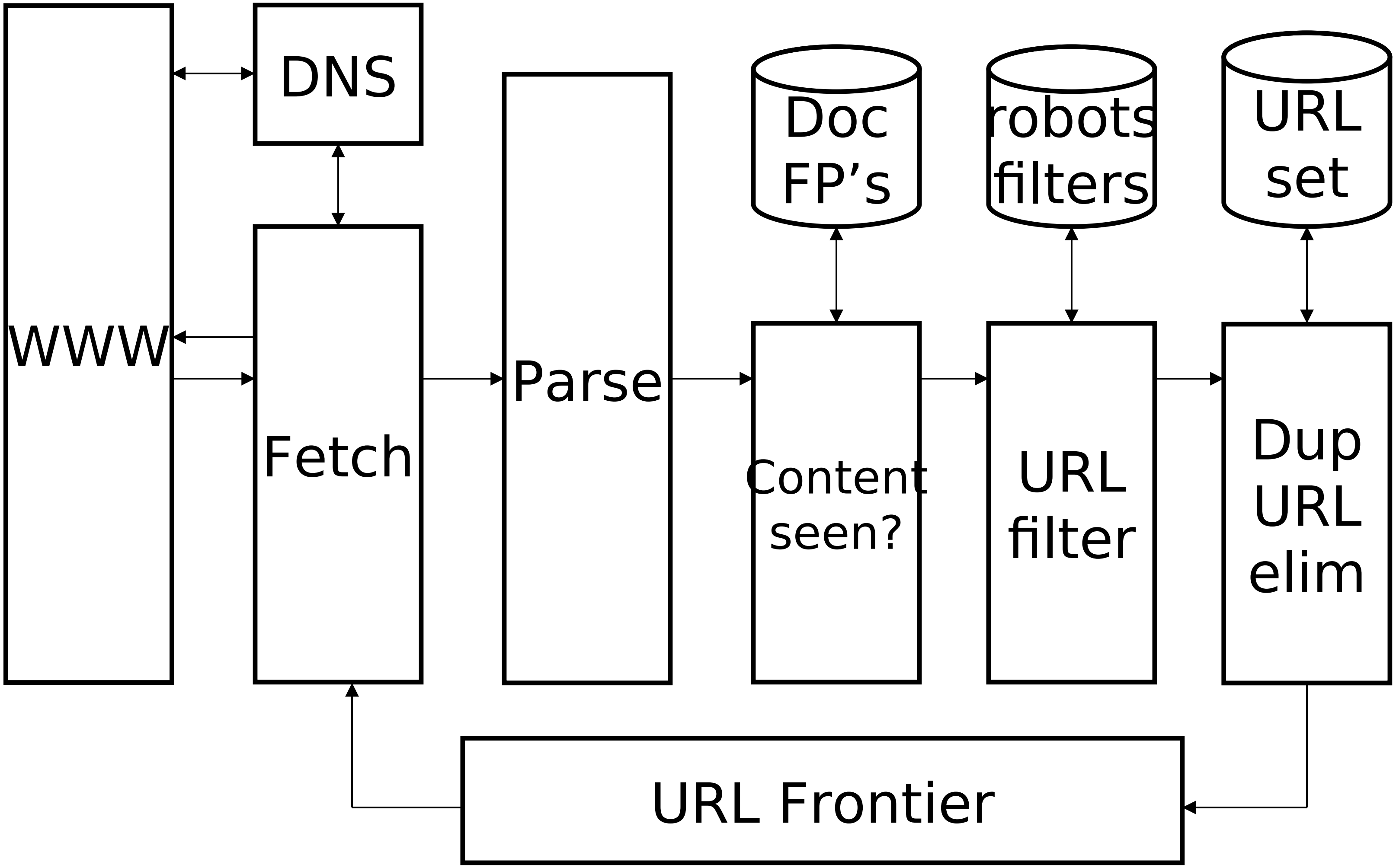
- Pick a URL from the frontier

Which one?

- Fetch the document at the URL
- Parse the URL
  - Extract links from it to other docs (URLs)
- Check if URL has content already seen
  - If not, add to indexes
- For each extracted URL
  - Ensure it passes certain URL filter tests
  - Check if it is already in the frontier (duplicate URL elimination)

E.g., only crawl .edu, obey robots.txt, etc.

Basic crawl architecture



Parsing: URL normalization

- When a fetched document is parsed, some of the extracted links are *relative* URLs
- E.g., http://en.wikipedia.org/wiki/Main_Page has a relative link to /wiki/Wikipedia:General_disclaimer which is the same as the absolute URL http://en.wikipedia.org/wiki/Wikipedia:General_disclaimer
- During parsing, must normalize (expand) such relative URLs

Content seen?

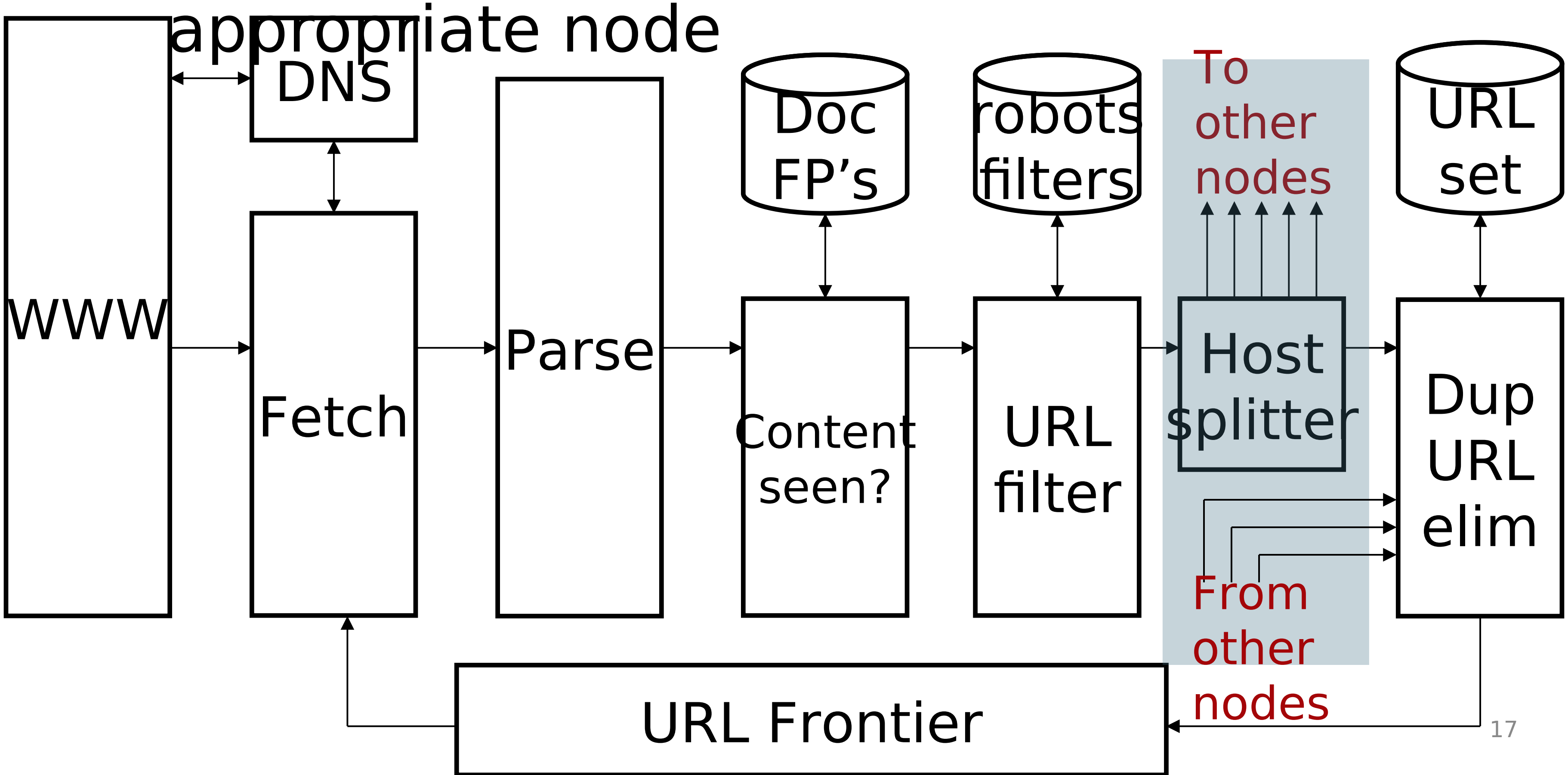
- Duplication is widespread on the web
- If the page just fetched is already in the index, do not further process it
- This is verified using document fingerprints or shingles
    - Second part of this lecture

Distributing the crawler

- Run multiple crawl threads, under different processes – potentially at different nodes
  - May be geographically distributed nodes
- Partition hosts being crawled into nodes
  - Hash used for partition
- How do these nodes communicate and share URLs?

Communication between nodes

- Output of the URL filter at each node is sent to the Dup URL Eliminator of the appropriate node



URL frontier: two main considerations

- Politeness: do not hit a web server too frequently
- Freshness: crawl some pages more often than others
  - E.g., pages (such as News sites) whose content changes often

These goals may conflict with each other.

(E.g., simple priority queue fails – many links out of a page go to its own site, creating a burst of accesses to that site.)

Politeness – challenges

- Even if we restrict only one thread to fetch from a host, can hit it repeatedly
- Common heuristic: insert time gap between successive requests to a host that is >> time for most recent fetch from that host

Introduction to

Information Retrieval

Near duplicate document detection

Duplicate documents

- The web is full of duplicated content

- Strict duplicate detection = exact match
  - Not as common

- But many, many cases of near duplicates
  - E.g., Last modified date the only difference between two copies of a page

Duplicate/Near-Duplicate Detection

- *Duplication*: Exact match can be detected with fingerprints

- *Near-Duplication*: Approximate match
  - Overview
    - Compute syntactic similarity with an edit-distance measure
    - Use similarity threshold to detect near-duplicates, e.g., Similarity > 80% => Documents are “near duplicates”

Computing Similarity

- Features:
  - Segments of a document (natural or artificial breakpoints)
  - Shingles (Word N-Grams)
  - ***a rose is a rose is a rose*** → 4-grams are

    a_rose_is_a

    rose_is_a_rose

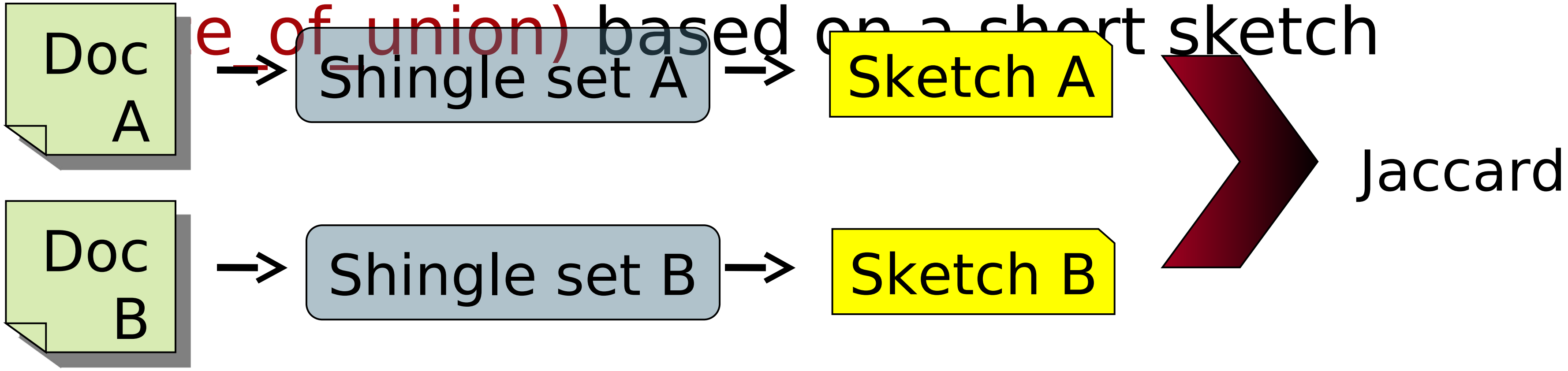
    is_a_rose_is

- Similarity Measure between two docs (= sets of shingles)
  - Jaccard cooefficient: (Size_of_Intersection / Size_of_Union)

Shingles + Set Intersection

- Computing exact set intersection of shingles between all pairs of documents is expensive

- Approximate using a cleverly chosen subset of shingles from each (a *sketch*)
- Estimate (size_of_intersection / e_of_union) based on a short sketch



Doc A

Shingle set A

Sketch A

Doc B

Shingle set B

Sketch B

Jaccard

Sketch of a document

- Create a “sketch vector” (of size ~200) for each document
  - Documents that share ≥ *t* (say 80%) corresponding vector elements are deemed near duplicates
  - For doc *D*, sketch$_D$[*i*] is as follows:
    - Let f map all shingles in the universe to 1..2^m (e.g., f = fingerprinting)
    - Let π_i be a *random permutation* on 1..2^m
    - Pick MIN {π_i(f(s))} over all shingles *s* in *D*

See details in book

Random permutations

- Random permutations are expensive to compute

- Linear permutations work well in practice
  - For a large prime p, consider permutations over $\{0, \ldots, p-1\}$ drawn from the set:

    $F_p = \{\pi_{a,b} : 1 \leq a \leq p-1, 0 \leq b \leq p-1\}$ where

$$\pi_{a,b}(x) = ax + b \bmod p$$

See details in book

Final notes

- Shingling is a *randomized algorithm*
    - It will give us the right (wrong) answer with some probability on *any input*

- We’ve described how to detect near duplication in a pair of documents
- In “real life” we’ll have to concurrently look at many pairs
    - See text book for details