Crawling and web indexes

CE-324: Modern Information Retrieval

Sharif University of Technology

M. Soleymani

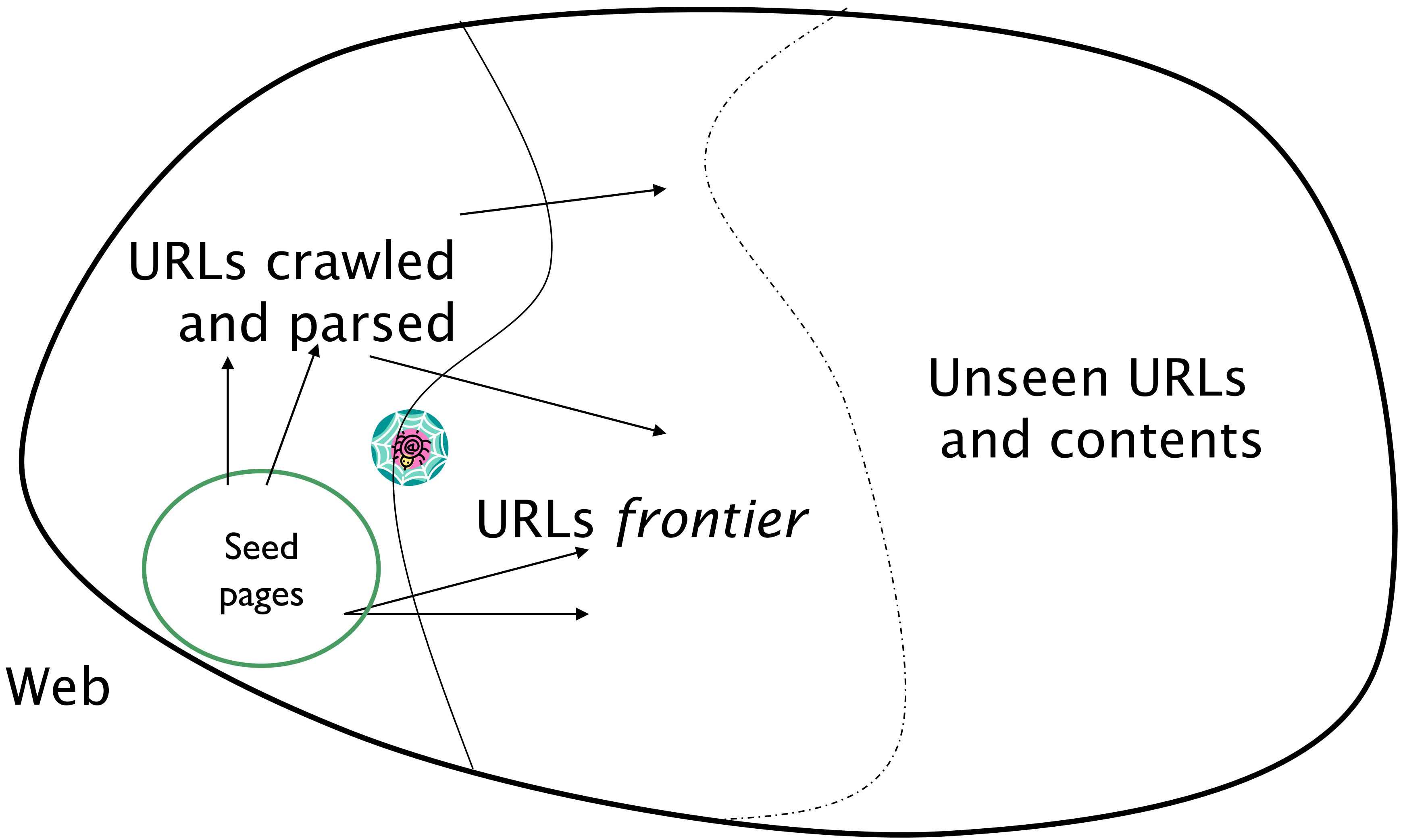
Fall 2016

Most slides have been adapted from: Profs. Manning, Nayak & Raghavan (CS-276, Stanford)

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

Crawling picture



What any crawler *must* do

- Be Polite: Respect implicit and explicit politeness considerations
  - Only crawl allowed pages
  - Respect *robots.txt* (more on this shortly)

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

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 (Cont'd)

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

Explicit and implicit politeness

- Explicit politeness: specifications from webmasters on what portions of site can be crawled
  - robots.txt
- 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/wc/norobots.html
- Website announces its request on what can(not) be crawled
  - For a server, create a file `/robots.txt`
  - This file specifies access restrictions

Robots.txt example

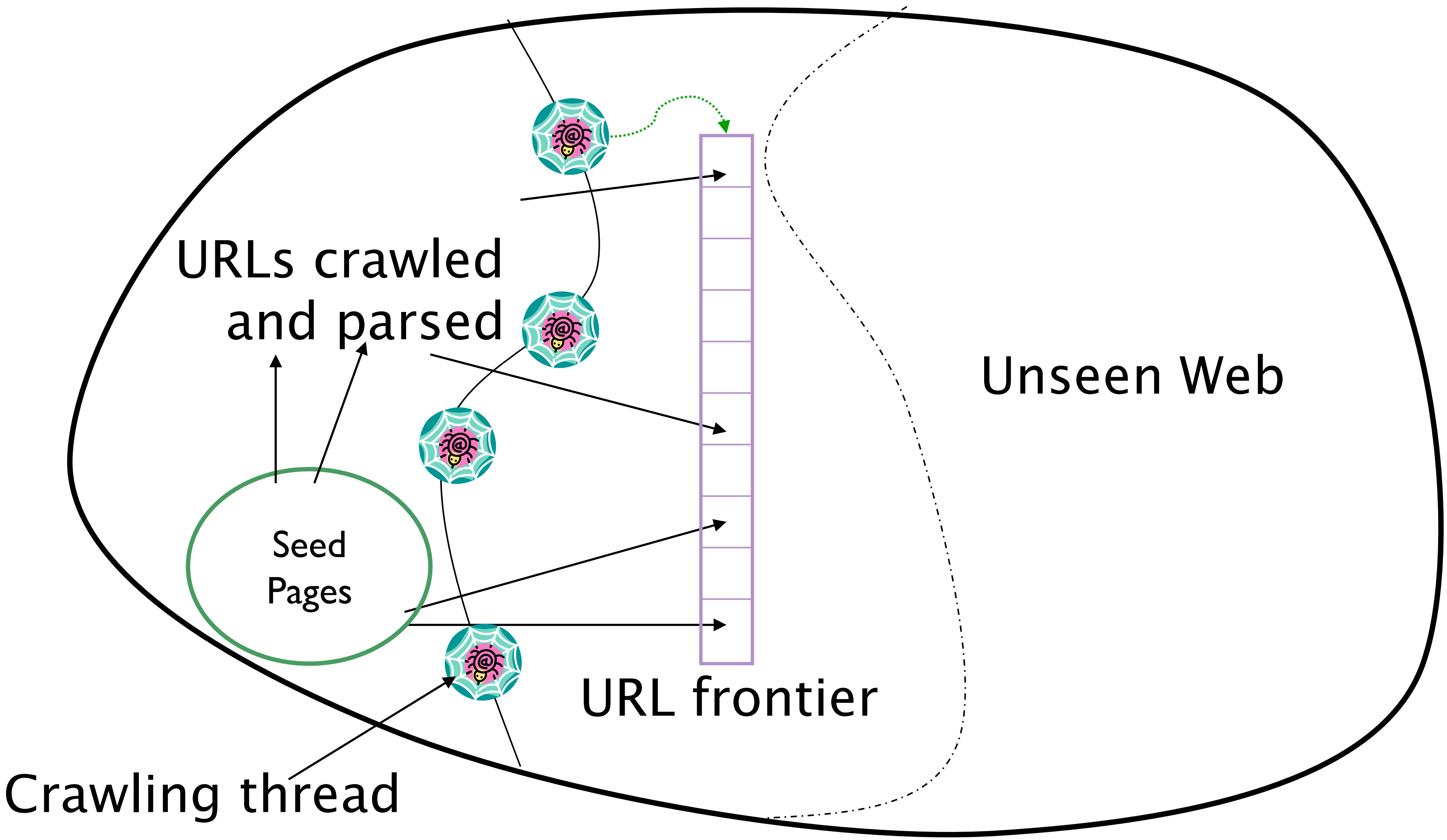
- No robot should visit any URL starting with "/yoursite/temp/", except the robot called “searchengine":

```
User-agent: *
Disallow: /yoursite/temp/

User-agent: searchengine
Disallow:
```

Robots.txt example: nih.gov

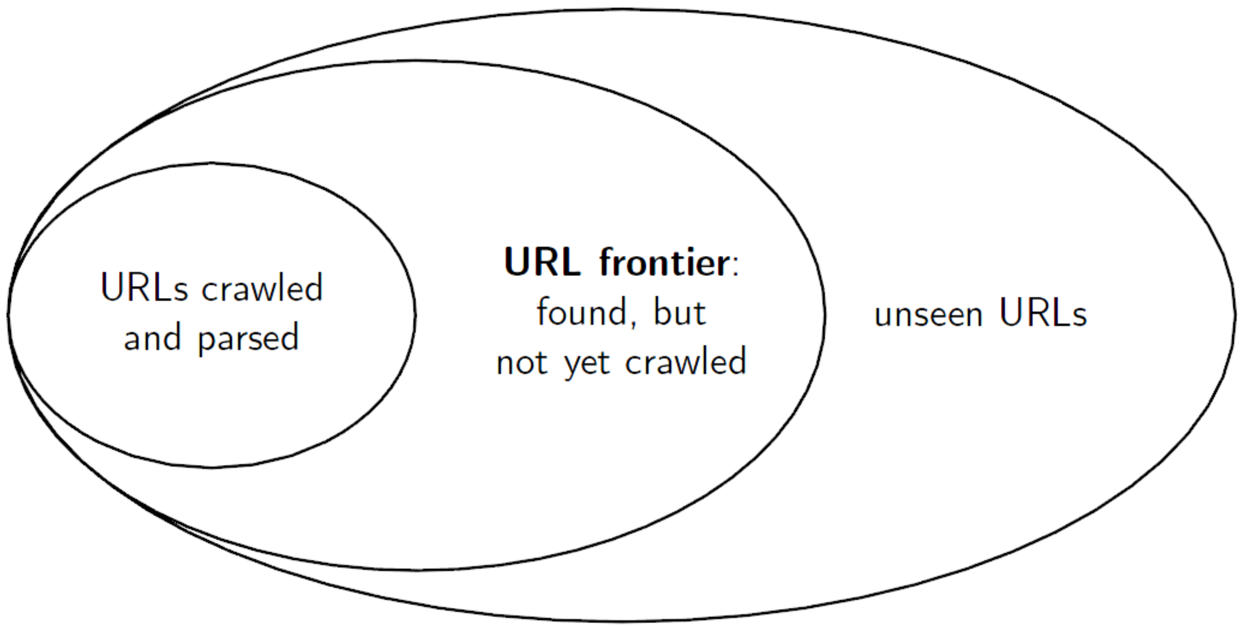
```
User-agent: PicoSearch/1.0
Disallow: /news/information/knight/
Disallow: /nidcd/
...
Disallow: /news/research_matters/secure/
Disallow: /od/ocpl/wag/

User-agent: *
Disallow: /news/information/knight/
Disallow: /nidcd/
...
Disallow: /news/research_matters/secure/
Disallow: /od/ocpl/wag/
Disallow: /ddir/
Disallow: /sdminutes/
```

Updated crawling picture



URL frontier

- The URL frontier is the data structure that holds and manages URLs we've seen, but that have not been crawled yet.
- Can include multiple pages from the same host
  - Must avoid trying to fetch them all at the same time
- Must keep all crawling threads busy

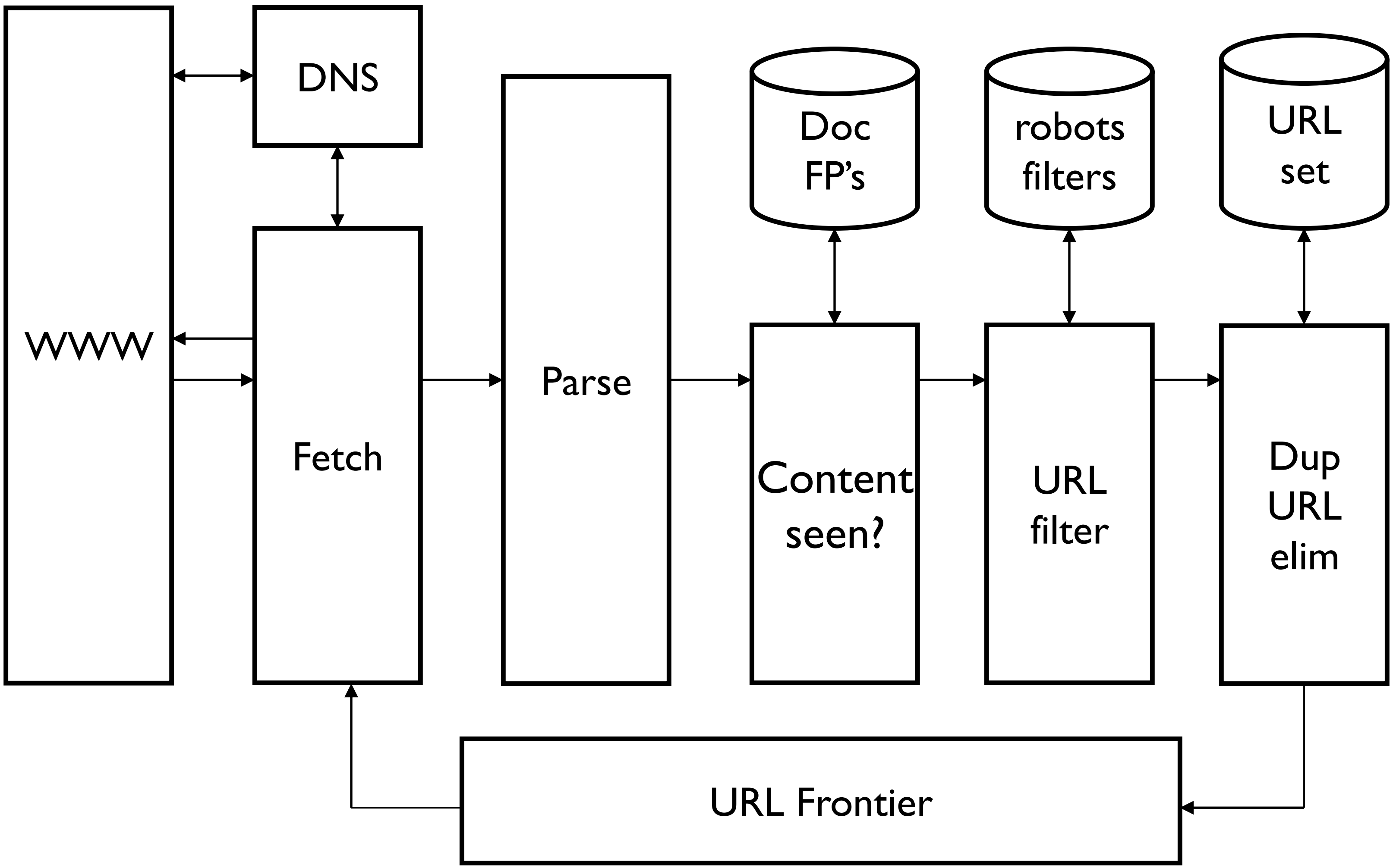


Processing steps in crawling

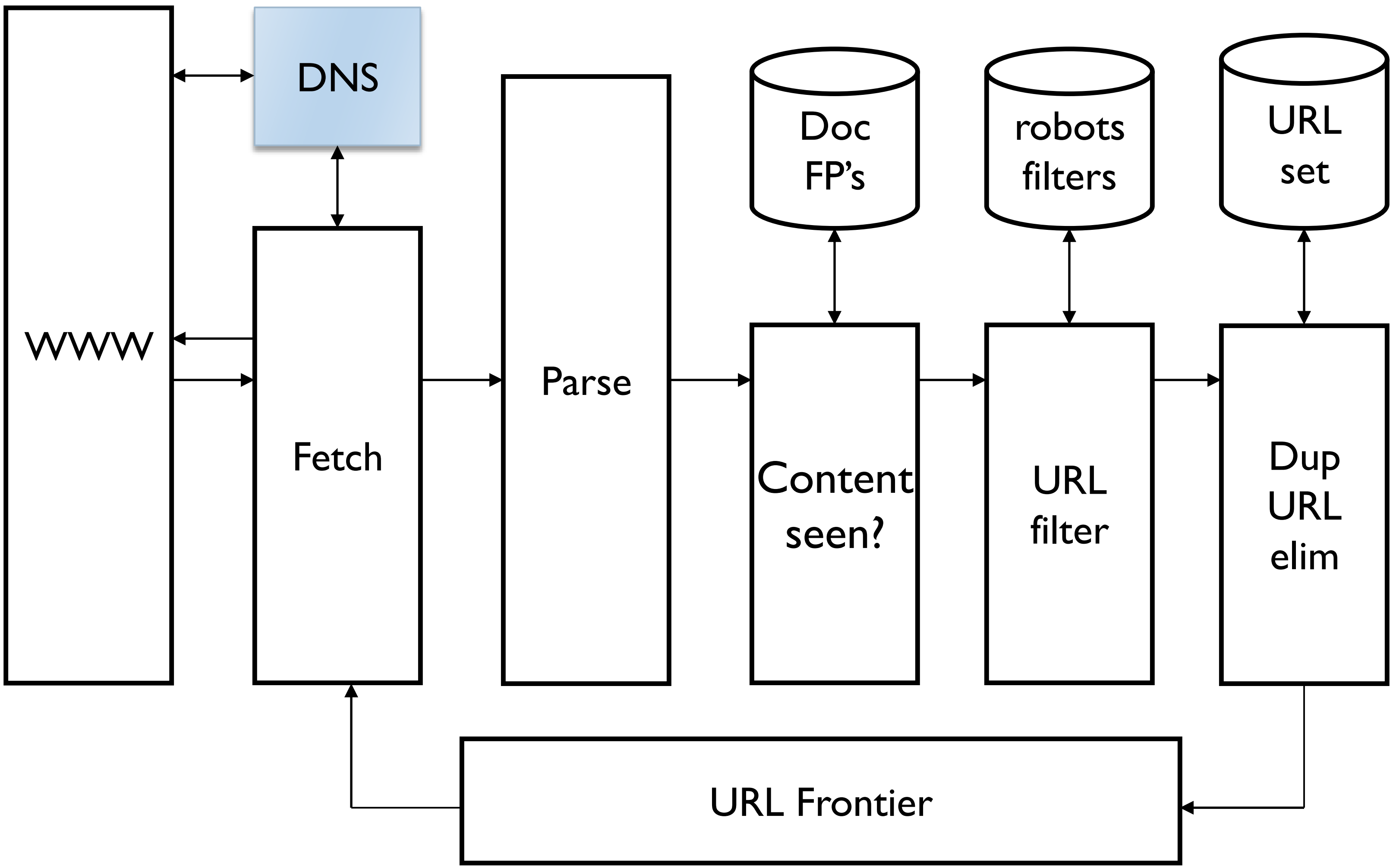
- Pick a URL from the frontier
- Fetch the doc at the URL
- Parse the URL
  - Extract links from it to other docs (URLs)
- Check if URL has content already seen
  - If not, add to indexes
- For each extracted URL
  - Ensure it passes certain URL filter tests
  - Check if it is already in the frontier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

Which one?

Basic crawl architecture



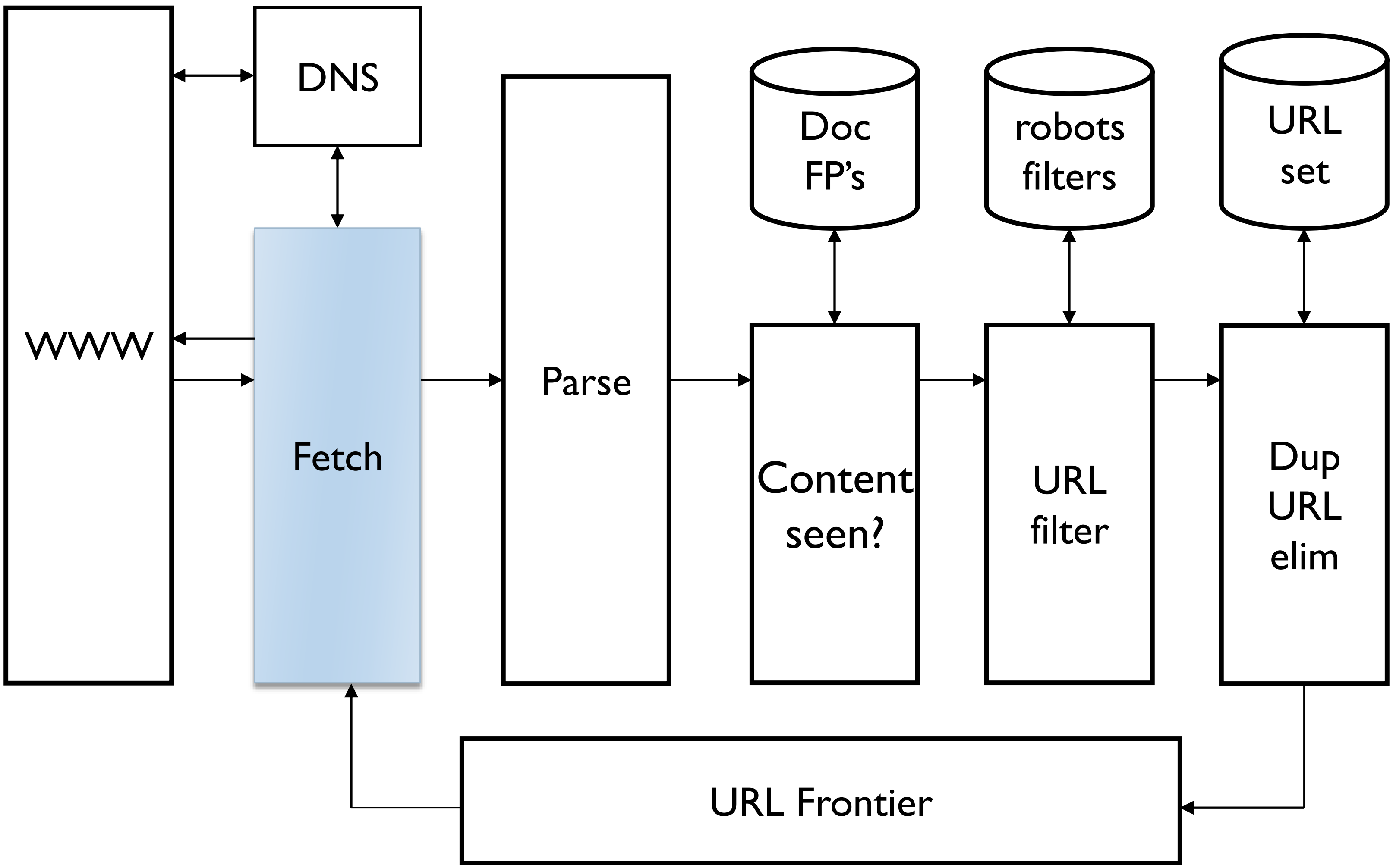
Basic crawl architecture



DNS (Domain Name Server)

- A lookup service on the internet
  - Given a URL, retrieve IP address of its host
  - Service provided by a distributed set of servers – thus, lookup latencies can be high (even seconds)
- Common OS implementations of DNS lookup are *blocking*: only one outstanding request at a time
- Solutions
  - DNS caching
  - Batch DNS resolver – collects requests and sends them out together

Basic crawl architecture

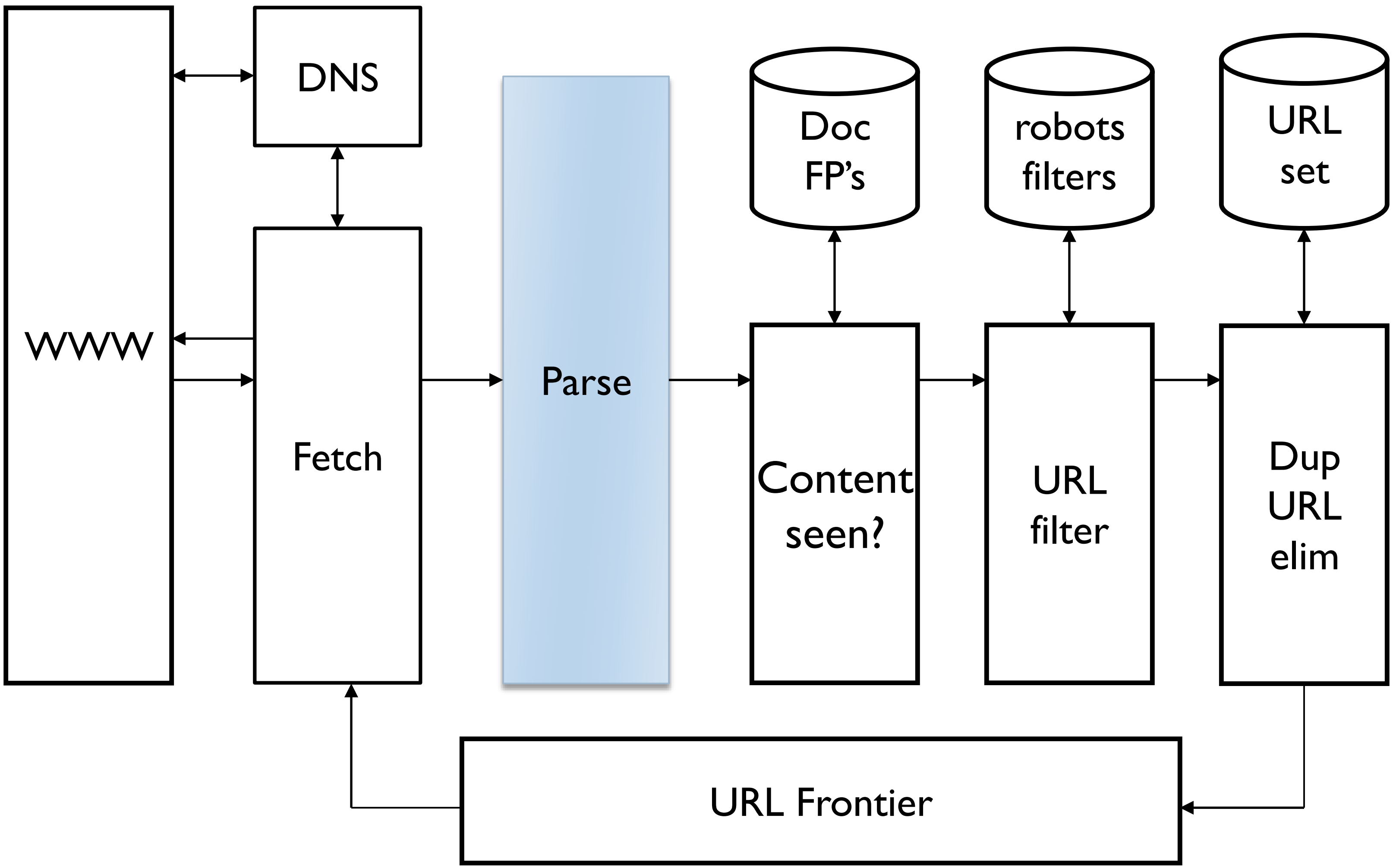


Parsing: URL normalization

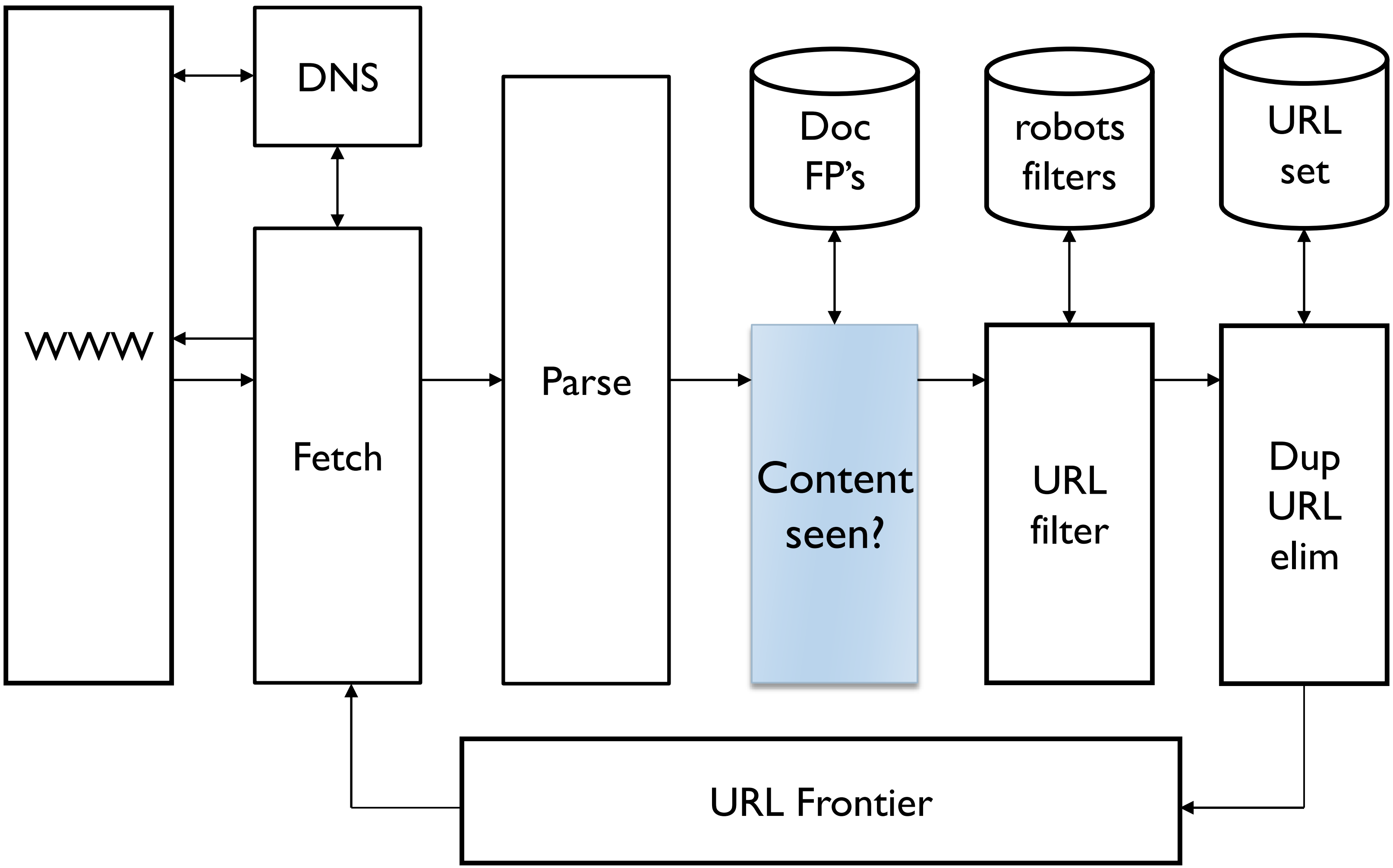
- When a fetched document is parsed, some of the extracted links are *relative* URLs
  - E.g., http://en.wikipedia.org/wiki/Main_Page has a relative link to **/wiki/Wikipedia:General_disclaimer** which is the same as the absolute URL http://en.wikipedia.org/wiki/Wikipedia:General_disclaimer

- During parsing, must normalize (expand) such relative URLs

Basic crawl architecture



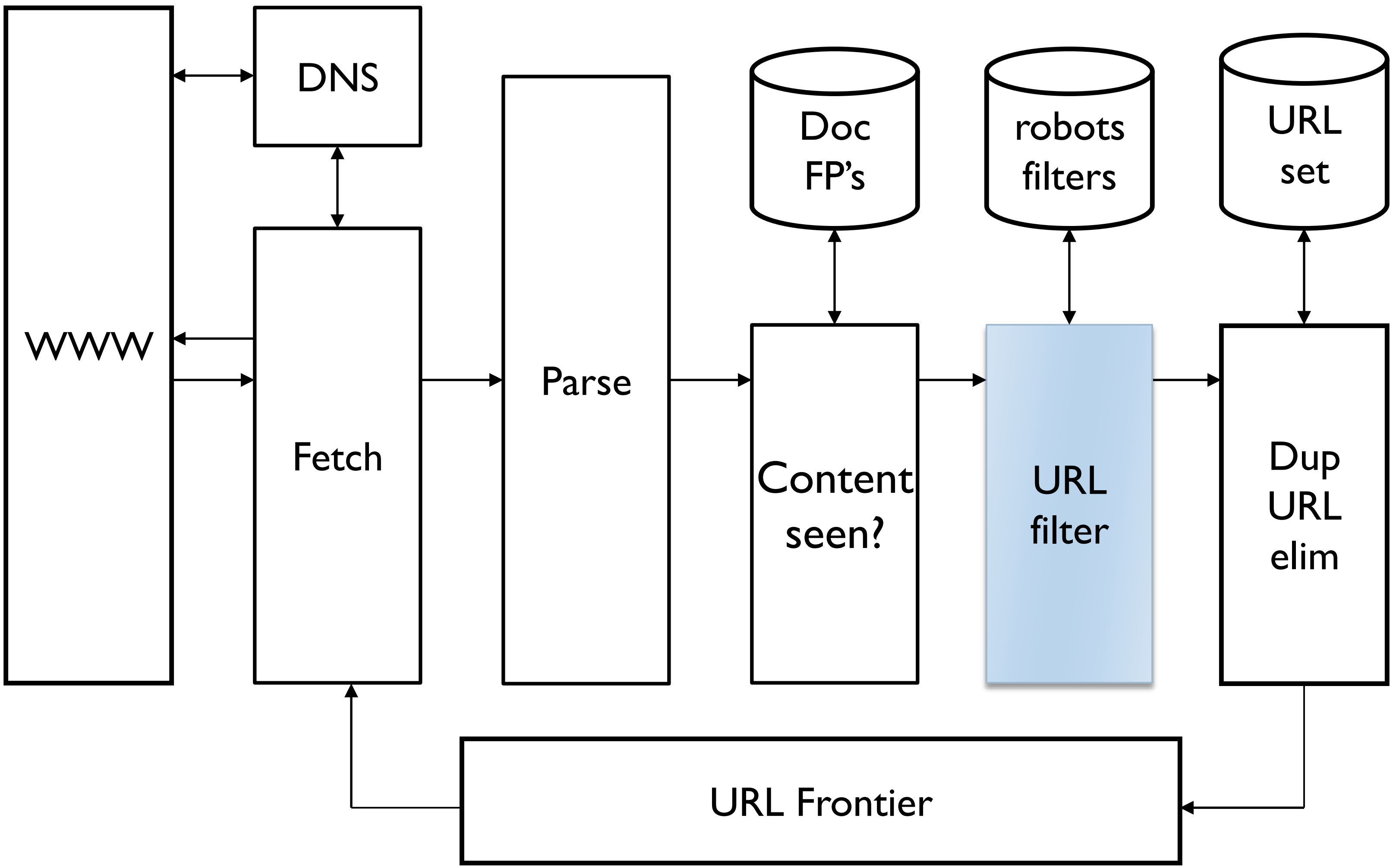
Basic crawl architecture



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

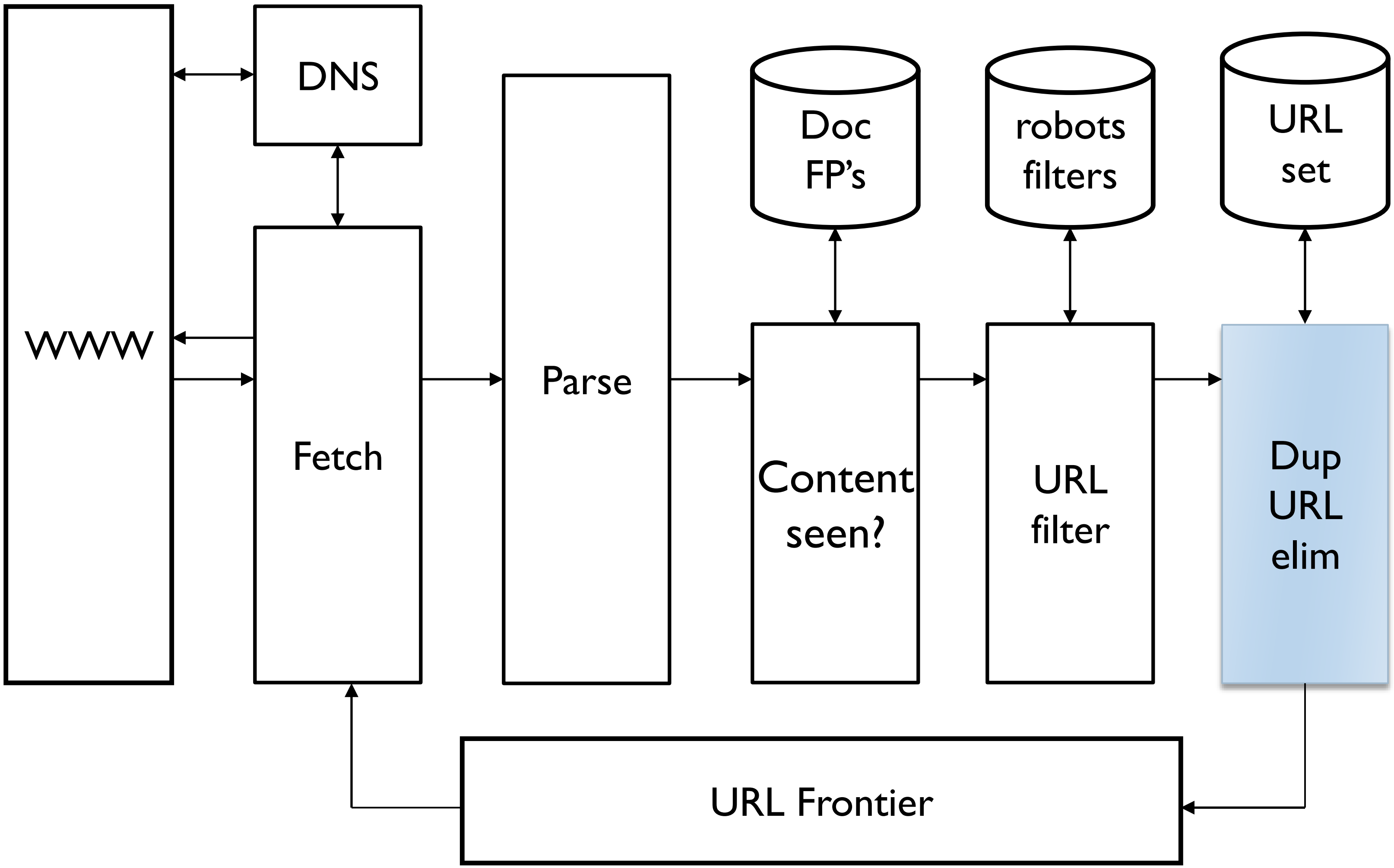
Basic crawl architecture



Filters and robots.txt

- Filters – regular expressions for URL's to be crawled or not
  - E.g., only crawl .edu
  - Filter URLs that we can not access according to robots.txt
- Once a robots.txt file is fetched from a site, need not fetch it repeatedly
  - Doing so burns bandwidth, hits web server
  - Cache robots.txt files

Basic crawl architecture



Duplicate URL elimination

- For a non-continuous (one-shot) crawl, test to see if the filtered URL has already been passed to the frontier

- For a continuous crawl – see details of frontier implementation

Simple crawler: complications

- Web crawling isn't feasible with one machine
  - All steps are distributed
- Malicious pages
  - Spam pages
  - Spider traps
    - Malicious server that generates an infinite sequence of linked pages
    - Sophisticated traps generate pages that are not easily identified as dynamic.
- 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

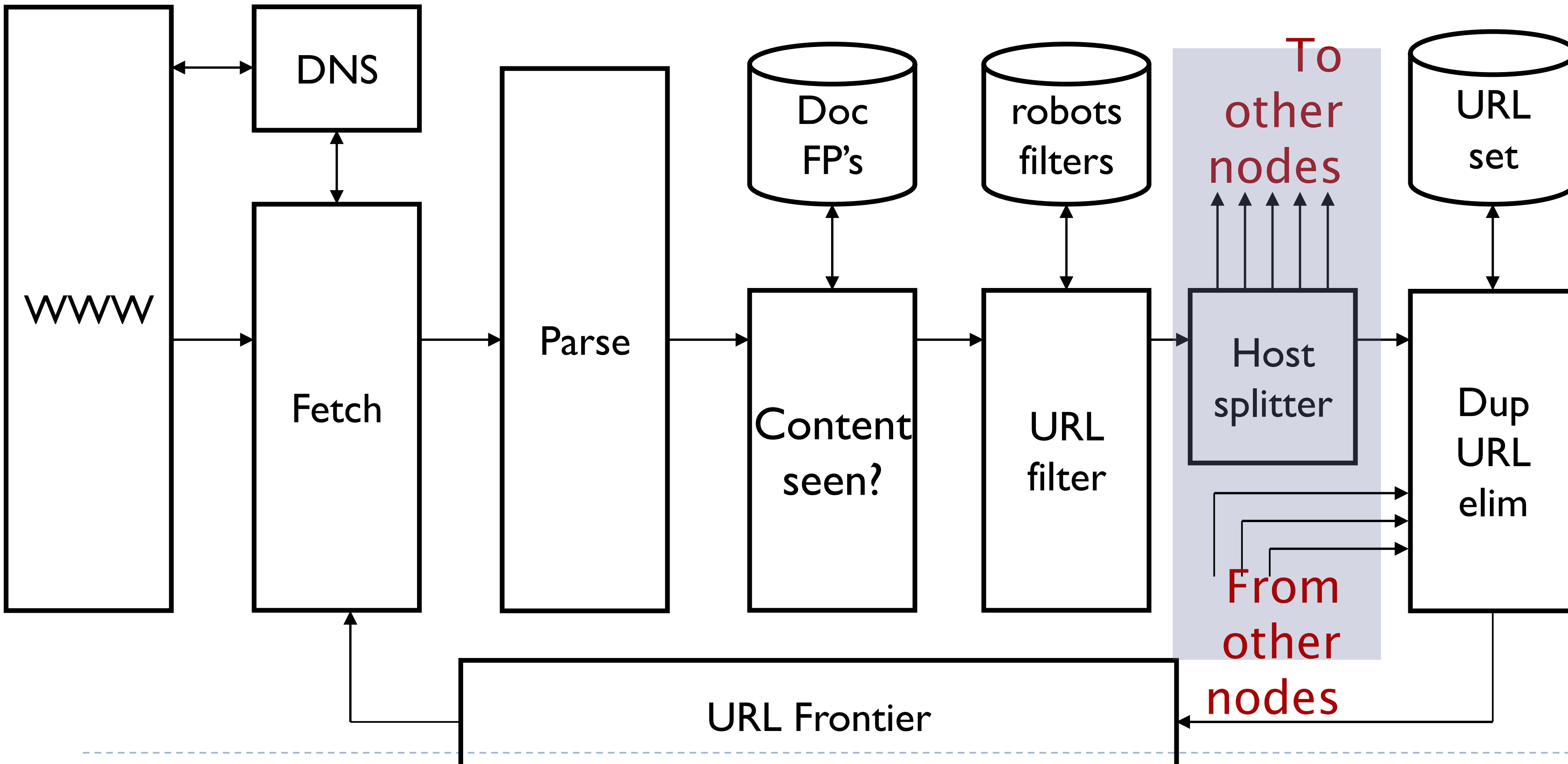
Distributing the crawler

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

Google data centers (wayfaring.com)

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

- <u>Politeness</u>: do not hit a web server too frequently

- <u>Freshness</u>: crawl some pages more often than others
  - E.g., pages (such as News sites) whose content changes often

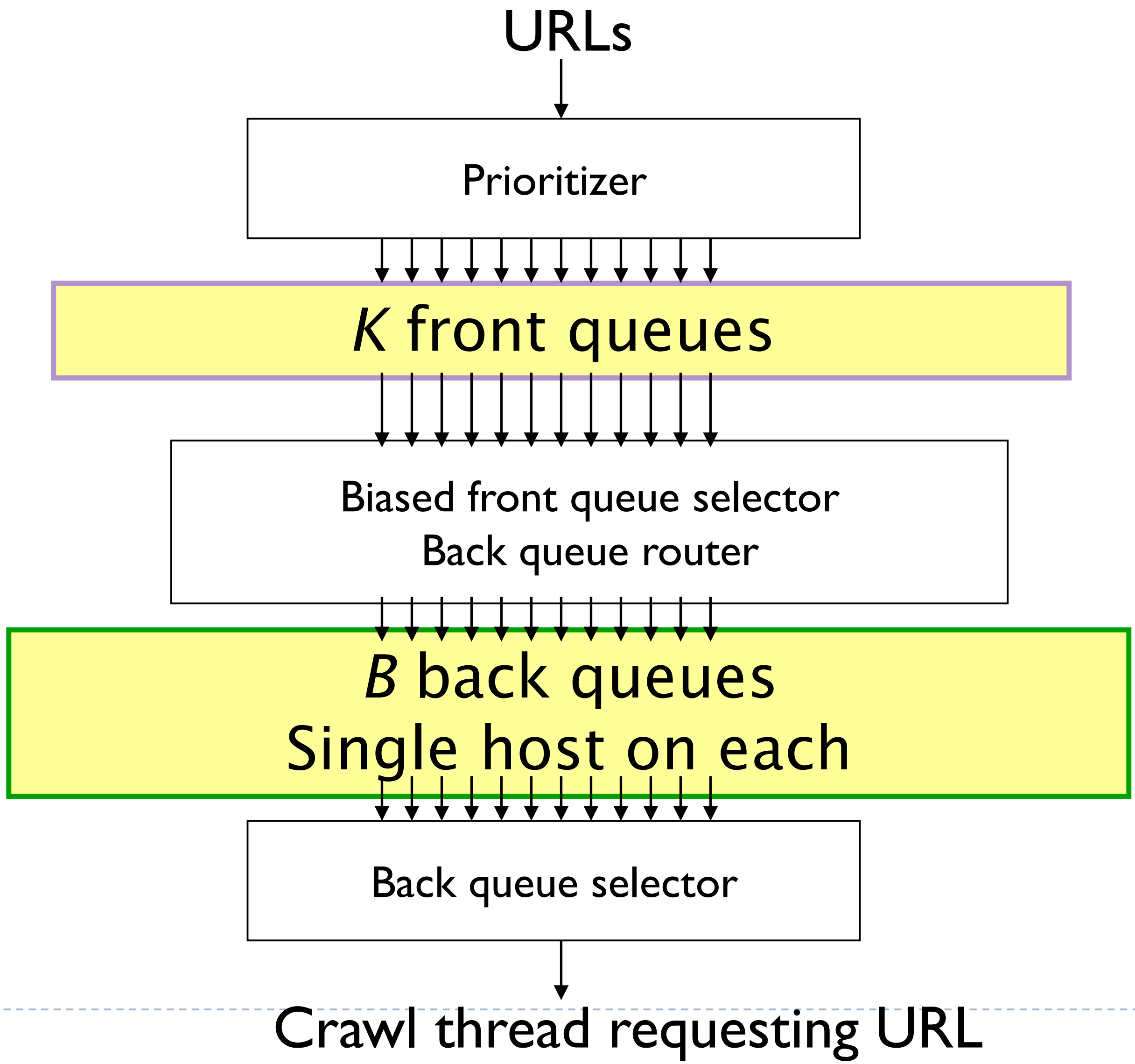
These goals may conflict each other.

(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

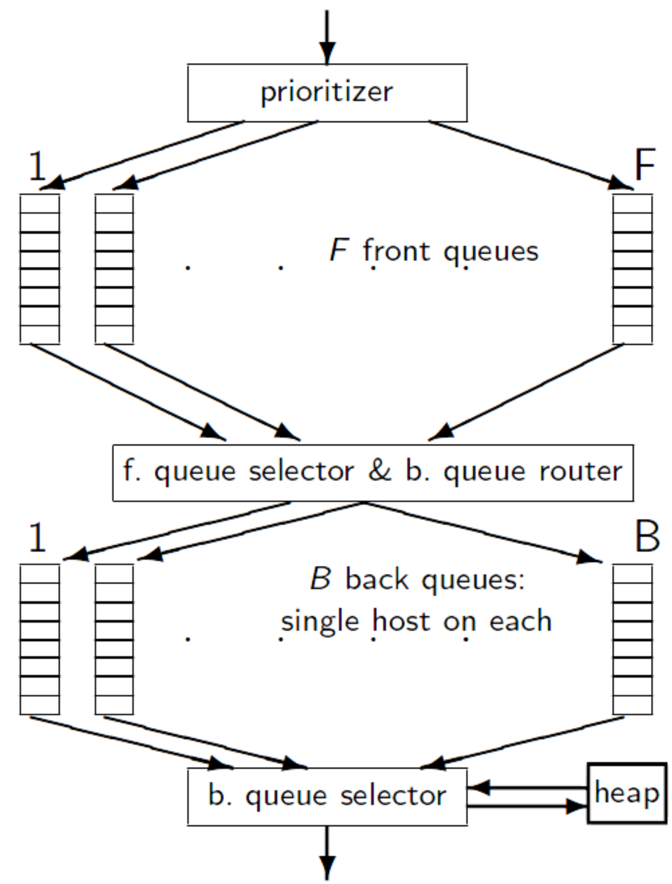
- Even if we restrict only one thread to fetch from a host, can hit it repeatedly
- Common heuristic:
  - Insert time gap between successive requests to a host that is >> time for most recent fetch from that host

URL frontier: Mercator scheme

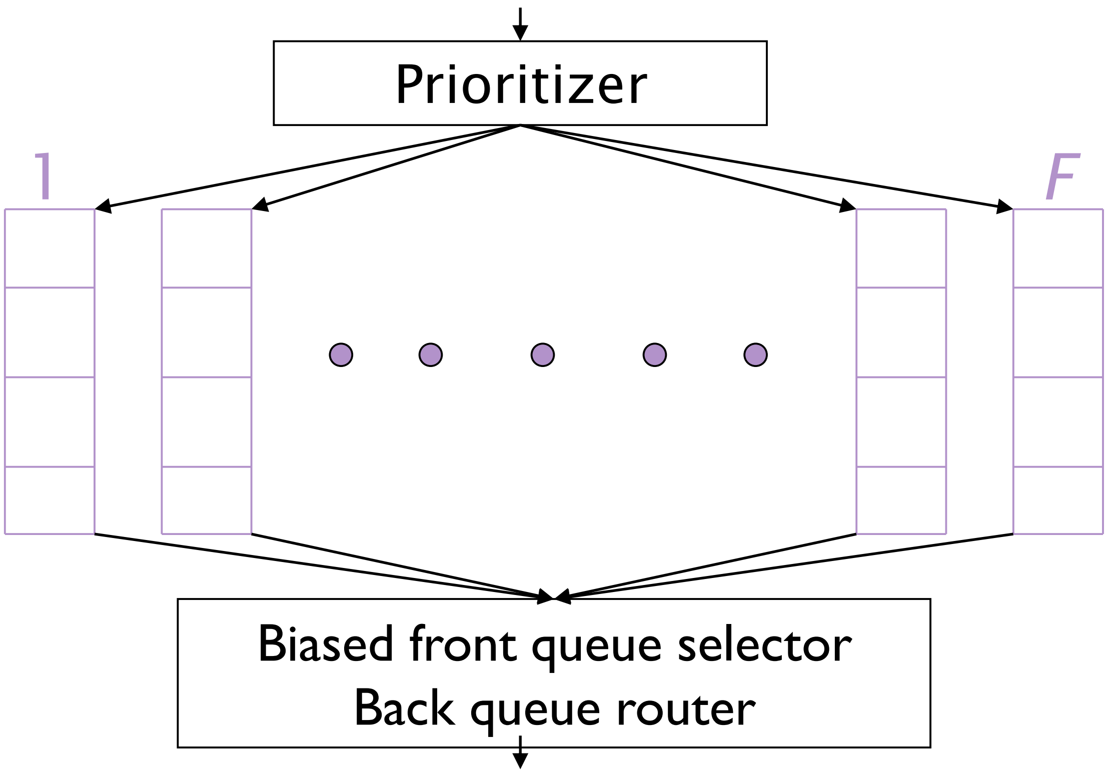


Mercator URL frontier

- URLs flow in from the top into the frontier
- **Front queues** manage prioritization
- **Back queues** enforce politeness
- Each queue is FIFO



Mercator URL frontier: Front queues



Selection from front queues is initiated by back queues

Pick a front queue from which to select next URL

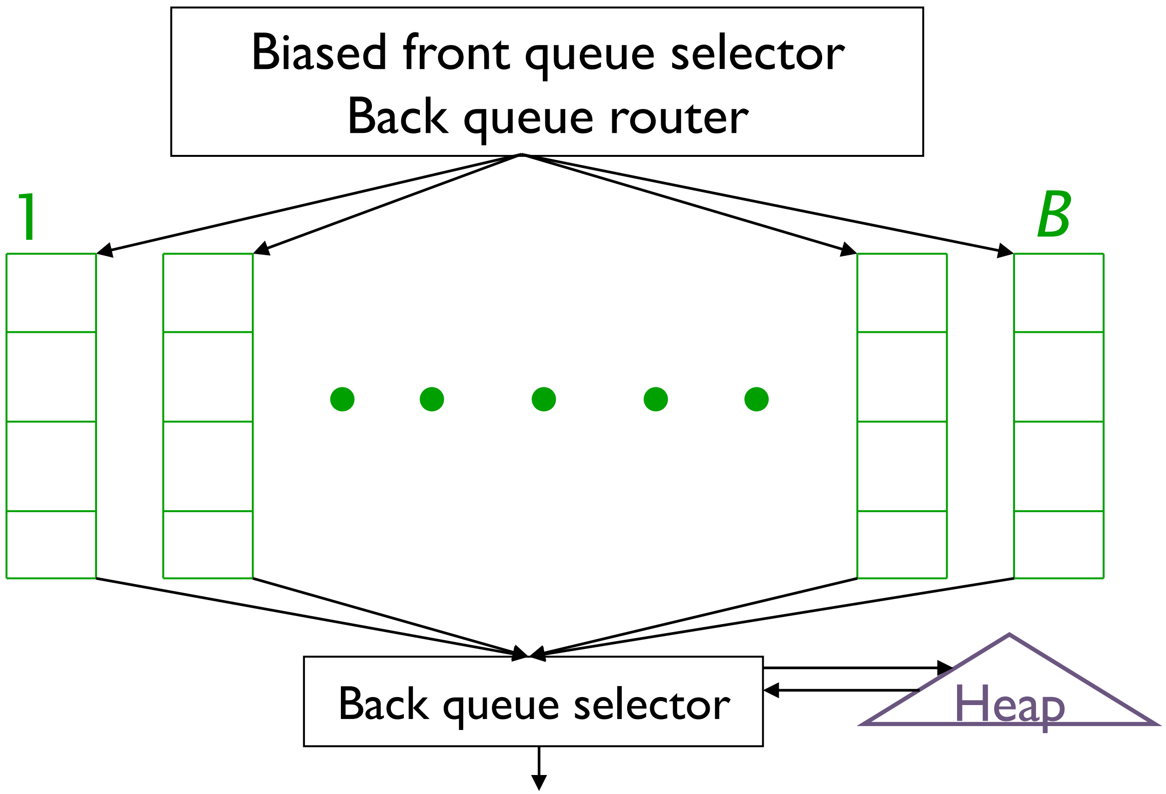
Mercator URL frontier: Front queues

- Prioritizer assigns to URL an integer priority between 1 and *F*
  - Appends URL to corresponding queue
- Heuristics for assigning priority
  - Refresh rate sampled from previous crawls
  - Application-specific (e.g., "crawl news sites more often")

Mercator URL frontier: Biased front queue selector

- When a back queue requests a URL (in a sequence to be described): picks a front queue from which to pull a URL
- This choice can be round robin biased to queues of higher priority, or some more sophisticated variant
  - Can be randomized

Mercator URL frontier: Back queues



Invariant 1. Each back queue is kept non-empty while the crawl is in progress.

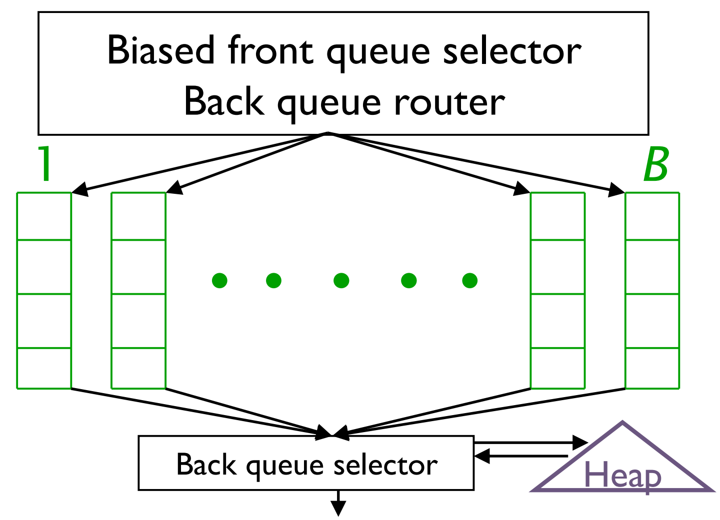
Invariant 2. Each back queue only contains URLs from a single host.

Maintain a table from hosts to back queues.

Host name	Back queue
…	3
	1
	20

Mercator URL frontier: Back queue heap

- One entry for each back queue
- The entry is the earliest time t_e at which the host corresponding to the back queue can be hit again
- This earliest time is determined from
  - Last access to that host
  - Any time buffer heuristic we choose



Mercator URL frontier: Back queue

- A crawler thread seeking a URL to crawl:
  - Extracts the root of the heap
  - Fetches URL at the head of corresponding back queue q
  - if queue $q = \emptyset$ then
    - Repeat

      (i) pull URLs v from front queues

      (ii) add v to its corresponding back queue . . .
    - . . . until we get a v whose host does not have a back queue.
  - Add v to q and create heap entry for q (and also update the table)

Number of back queues B

- Keep all threads busy while respecting politeness
- Mercator recommendation: three times as many back queues as crawler threads

Resources

- IIR Chapter 20
- Mercator: A scalable, extensible web crawler (Heydon et al. 1999)