Introduction to

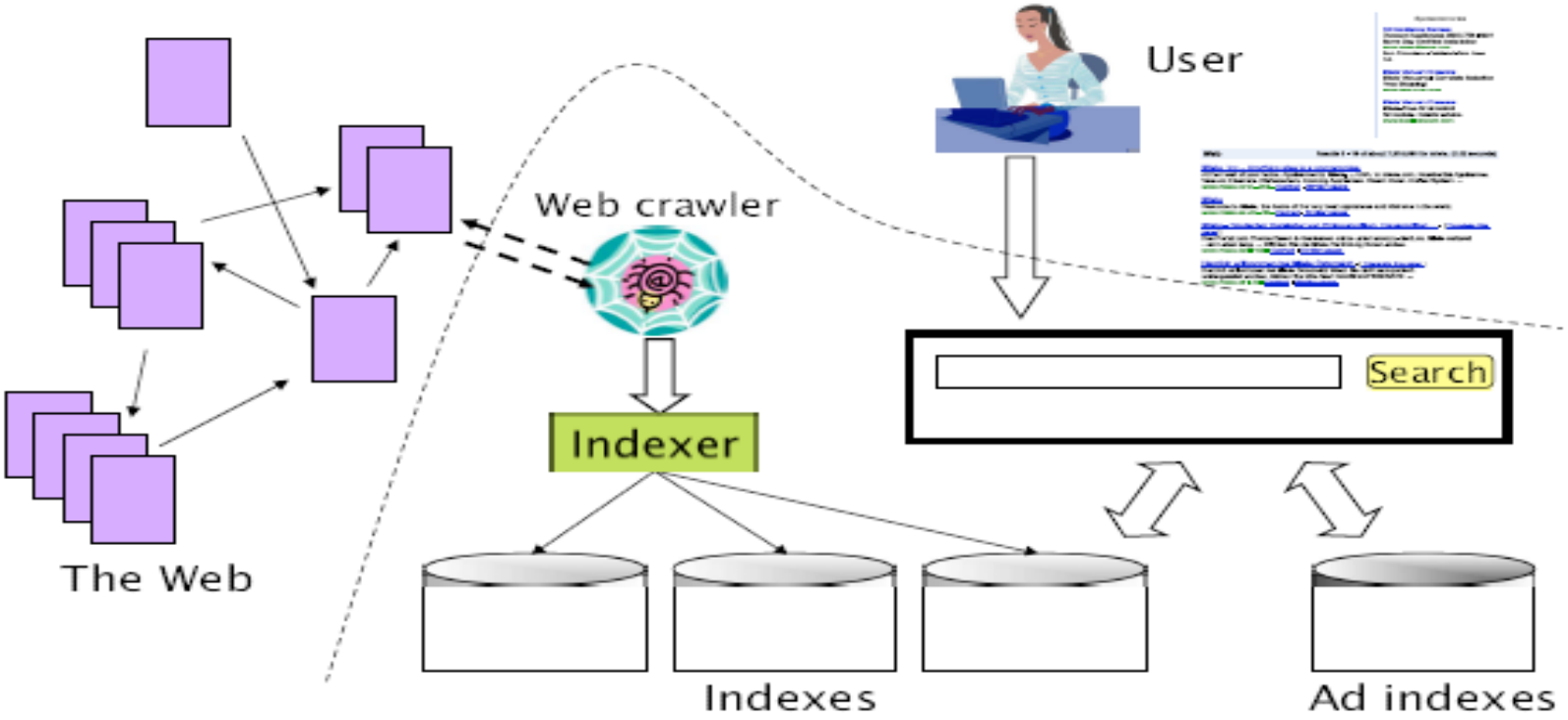
Information Retrieval

Lecture 17: Crawling and web indexes

- Crawling
  - Features
  - Crawling operation
  - Crawling Architecture
  - Distributing the Crawler
  - DNS resolution
  - URL Frontier
- Distributing indexes
- Connectivity servers

Crawling

- Web crawling is the process by which we gather pages from the Web together with the link structure that interconnects them, in order to index them and support a search engine.



Features a crawler *must (compulsory)* provide

- Be Polite: Respect implicit and explicit politeness considerations
  - Only crawl allowed pages
  - 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

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

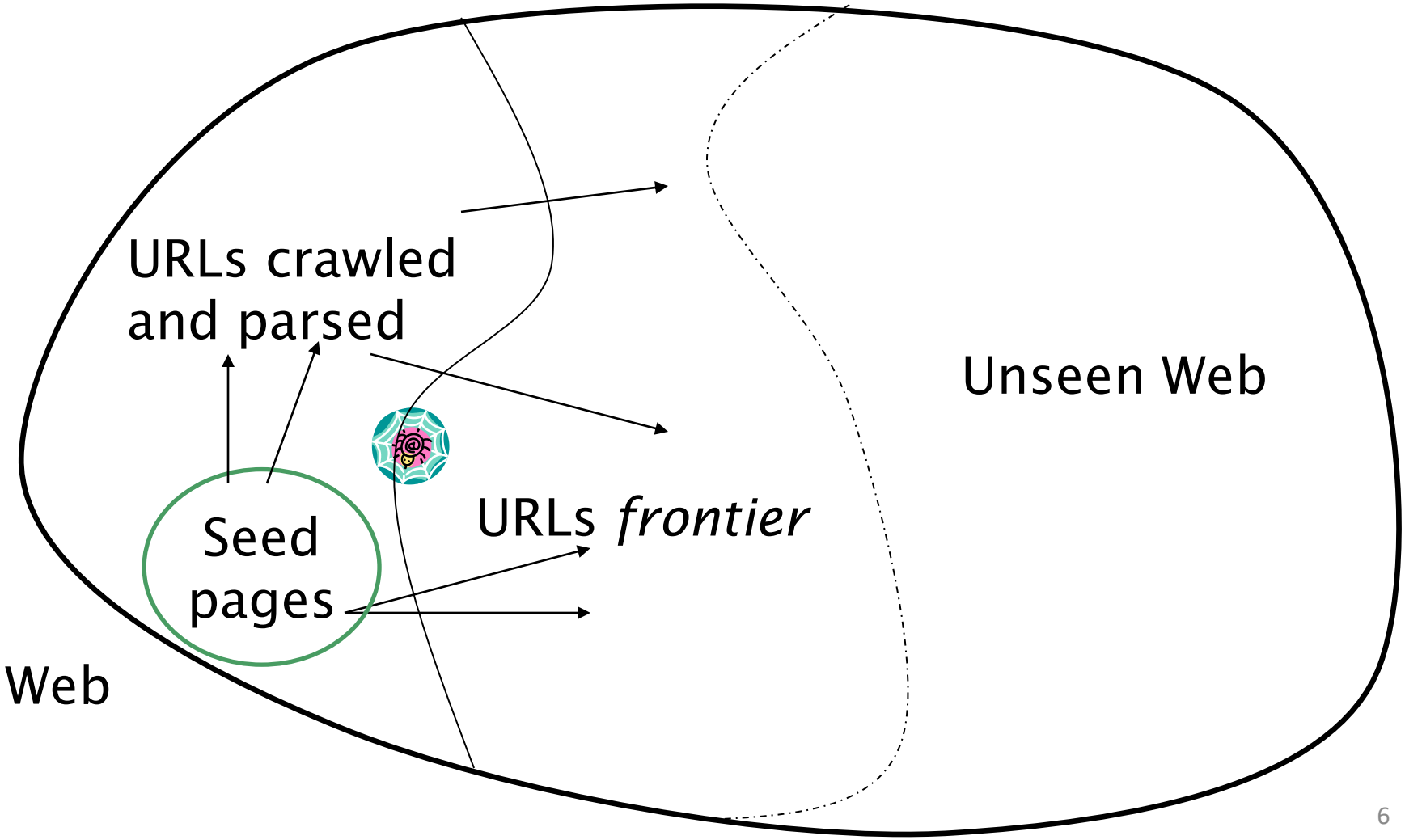
Features a crawler *should (one thinks best)* provide

- **Distributed:** The crawler should have the ability to execute in a distributed fashion across multiple machines.
- **Scalable:** The crawler architecture should permit scaling up the crawl rate by adding extra machines and bandwidth.
- **Performance and efficiency:** The crawl system should make efficient use of various system resources including processor, storage and network bandwidth.
- **Quality:** the crawler should be biased towards fetching "useful" pages first.
- **Freshness:** In many applications, the crawler should operate in continuous mode: it should obtain fresh copies of previously fetched pages.
- **Extensible:** Crawlers should be designed to be extensible in many ways – to cope with new data formats, new fetch protocols, and so on. This demands that the crawler architecture be modular.

Basic crawler operation

- The crawler begins with "seed" URLs
- It picks a URL from this seed set, then fetches the web page at that URL.
- The fetched page is then parsed, to extract both the text and the links from the page
- The extracted text is fed to a text indexer .
- The extracted links (URLs) are then added to a *URL frontier*, which consists of URLs whose corresponding pages have yet to be fetched by the crawler.
- Initially, the URL frontier contains the seed set; as pages are fetched, the corresponding URLs are deleted from the URL frontier.
- In continuous crawling, the URL of a fetched page is added back to the frontier for fetching again in the future.

Crawling picture



- Some basic properties any crawler should satisfy:

1. Only one connection should be open to any given host at a time.
2. A waiting time of a few seconds should occur between successive requests to a host.
3. Politeness restrictions should be obeyed.

Mercator crawler

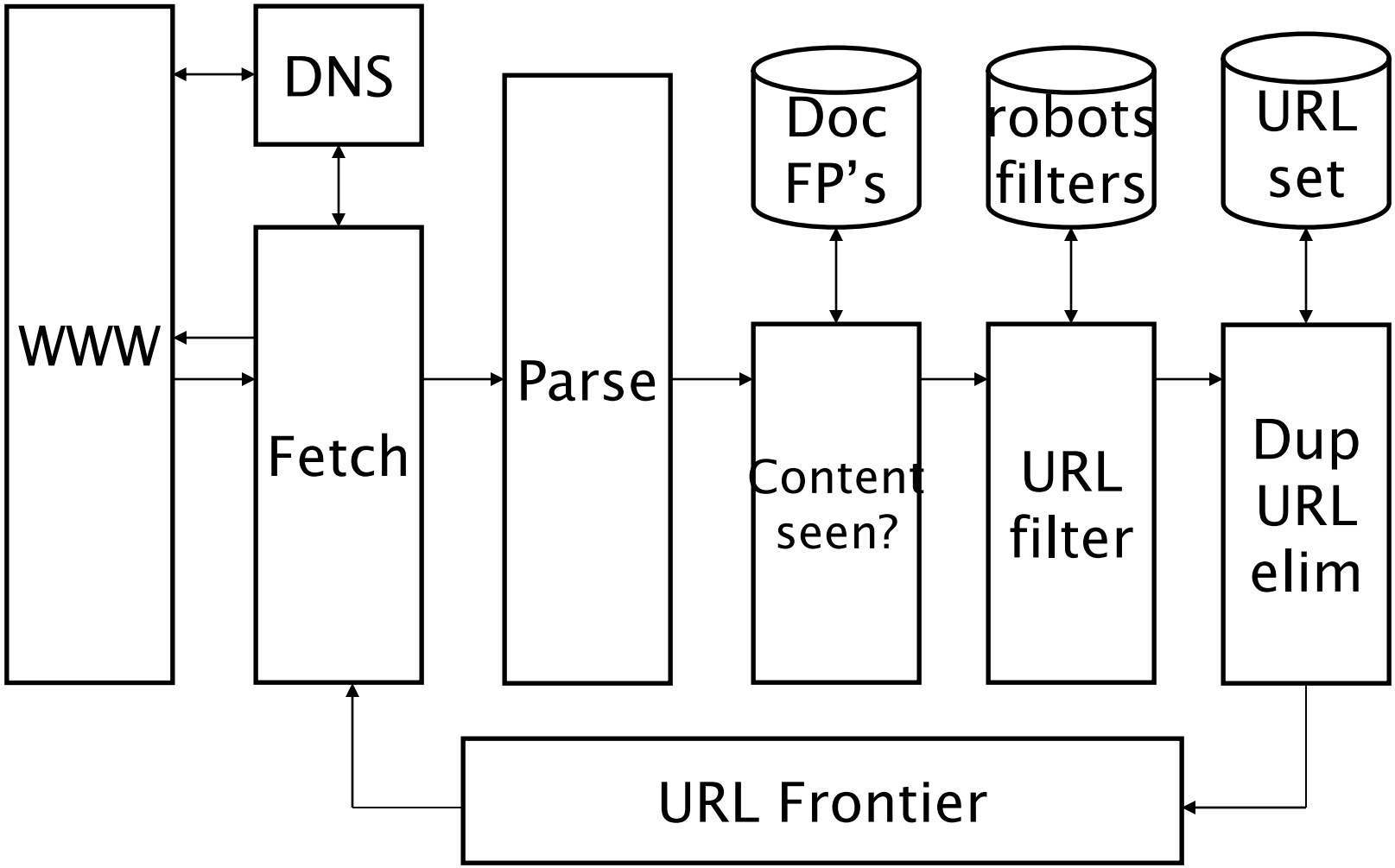
- It is the basis of a number of research and commercial crawlers.
- A multi-threaded design to address several bottlenecks in the overall crawler system in order to attain this fetch rate (100s of pages per second).

Crawler architecture

- The simple scheme for crawling demands several modules

1. The URL frontier, containing URLs yet to be fetched in the current crawl (but is back in the frontier for re-fetching).
2. A *DNS resolution* module that determines the web server from which to fetch the page specified by a URL.
3. A fetch module that uses the http protocol to retrieve the web page at a URL.
4. A parsing module that extracts the text and set of links from a fetched web page.
5. A duplicate elimination module that determines whether an extracted link is already in the URL frontier or has recently been fetched

Basic crawl architecture



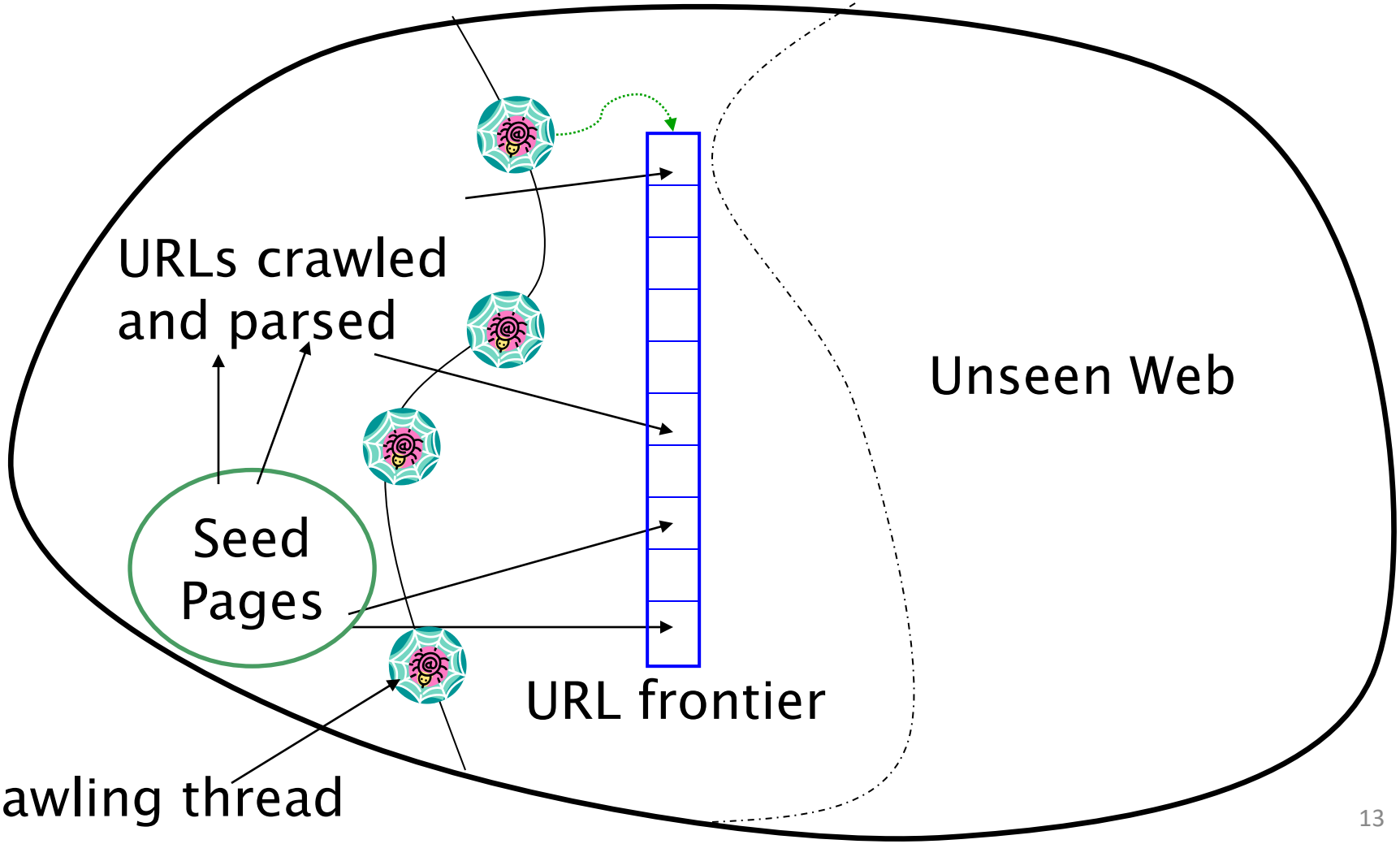
- A crawler thread begins by taking a URL from the frontier and fetching the web page at that URL, generally using the http protocol.
- The fetched page is then written into a temporary store
- Next, the page is parsed and the text as well as the links in it are extracted.
- The text (with any tag information – e.g., terms in boldface) is passed on to the indexer.
- Link information including anchor text is also passed on to the indexer for use in ranking
- In addition, each extracted link goes through a series of tests to determine whether the link should be added to the URL frontier.
  - the thread tests whether a web page with the same content has already been seen at another URL (Finger print checksum / Shingles Doc FPs)
- Next, a *URL filter* is used to determine whether the extracted URL should be excluded from the frontier based on tests. For instance, the crawl may seek to exclude certain domains (say, all .com URLs) – in this case the test would simply filter out the URL if it were from the .com domain.

- A similar test could be inclusive rather than exclusive.
- Many hosts use the *Robots Exclusion Protocol*. This is done by placing a file with the name robots.txt at the root of the URL hierarchy at the site.
- Example robots.txt file - that specifies that no robot should visit any URL whose position in the file hierarchy starts with /yoursite/temp/, except for the robot called "searchengine".

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

- Next, a URL should be *normalized* as the HTML encoding of a link from a web page *p* indicates the target of that link relative to the page *p*.
- Finally, the URL is checked for duplicate elimination: if the URL is already in the frontier or (in the case of a non-continuous crawl) already crawled, we do not add it to the frontier.

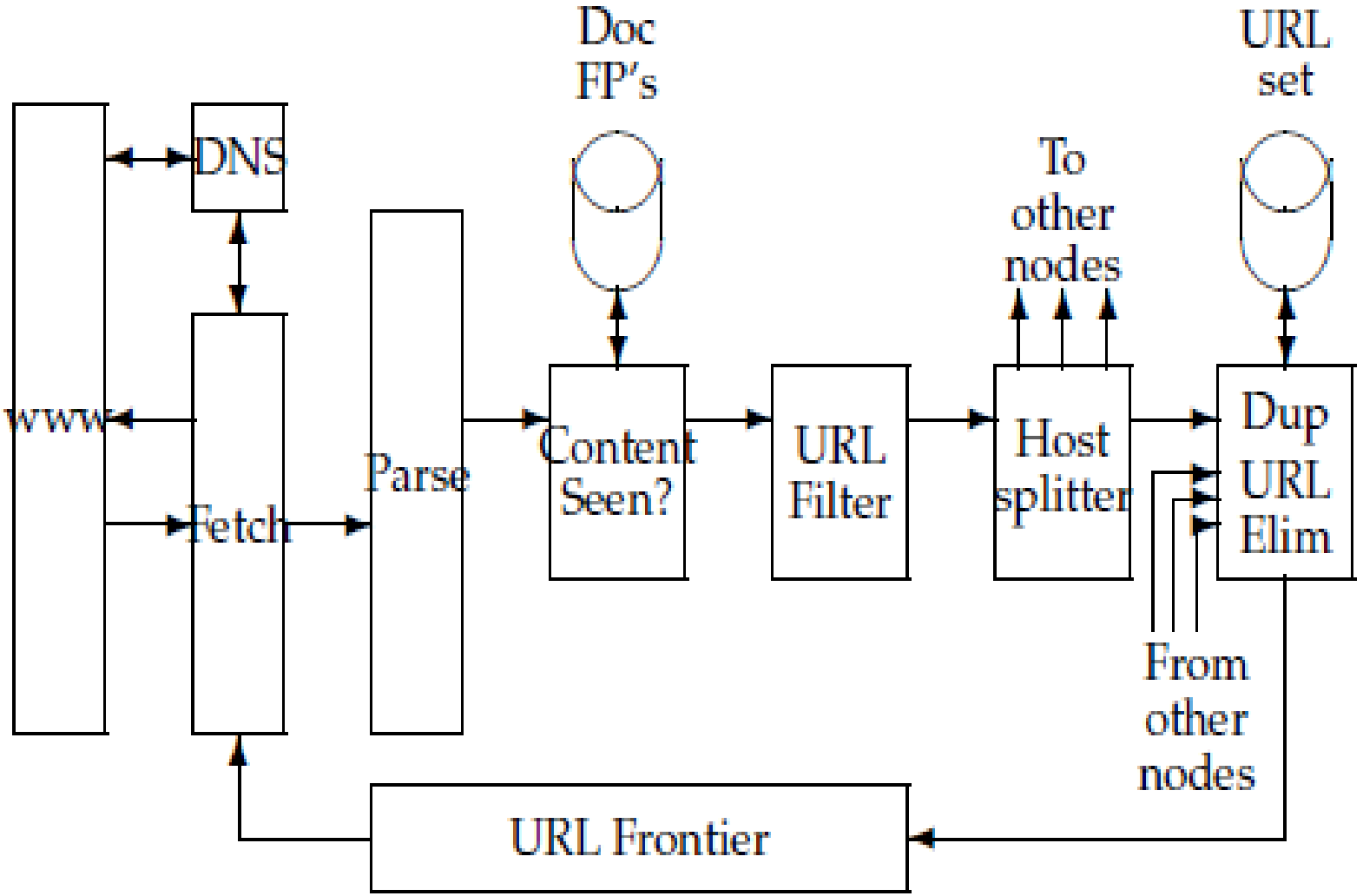
Updated crawling picture



- Certain housekeeping tasks are typically performed by a dedicated thread.
  - This thread is generally idle except that it wakes up once every few seconds to log crawl progress statistics (URLs crawled, frontier size, etc.),
  - Decide whether to terminate the crawl, or
  - checkpoint the crawl (once every few hours of crawling).
  - In checkpointing, a snapshot of the crawler's state (say, the URL frontier) is committed to disk.
  - In the event of a catastrophic crawler failure, the crawl is restarted from the most recent checkpoint.

Distributing the crawler

- Distribution is essential for scaling; it can also be of use in a geographically distributed crawler system where each node crawls hosts "near" it.
- Partitioning the hosts being crawled amongst the crawler nodes can be done by a hash function, or by some more specifically tailored policy (Eg. Geographical division.
- How do the various nodes of a distributed crawler communicate and share URLs?
- Solution is to replicate the basic crawler architecture at each node, with one essential difference: following the URL filter, we use a *host splitter* to dispatch each surviving URL to the crawler node responsible for the URL; thus the set of hosts being crawled is partitioned among the nodes.
- The "Content Seen?" module in the distributed architecture is complicated
  - Document fingerprints/shingles cannot be partitioned based on host name.
  - caching popular fingerprints does not help (since there are no popular fingerprints).
  - Documents change over time and so, in the context of continuous crawling, we must be able to delete their outdated fingerprints/shingles from the content-seen set(s)



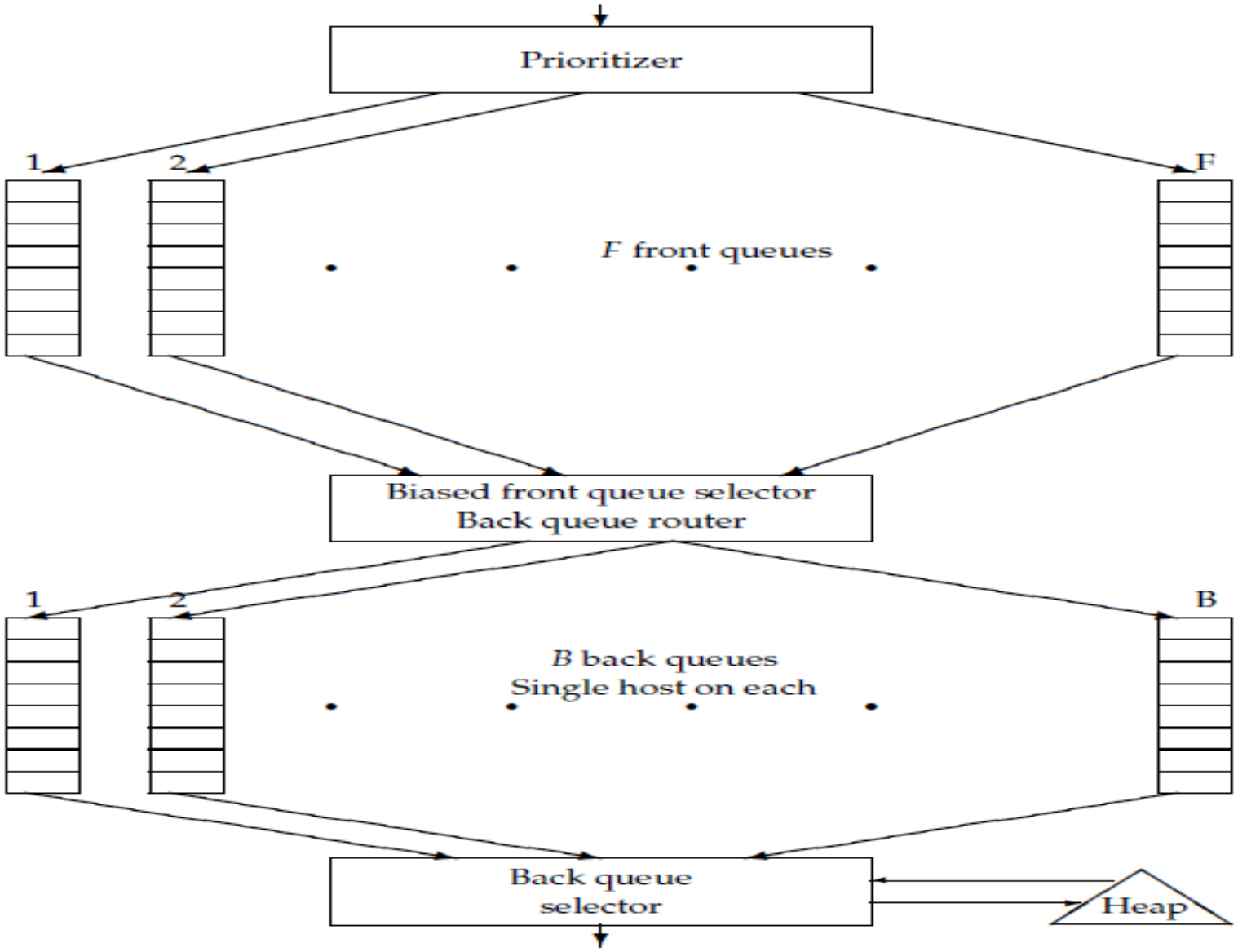
▶ **Figure 20.2** Distributing the basic crawl architecture.

DNS (Domain Name Service) resolution

- Given a URL such as www.wikipedia.org in textual form, translating it to an IP address (in this case, 207.142.131.248) is a process known as *DNS resolution* or DNS lookup.
- DNS resolution is a well-known bottleneck in web crawling.
  - Due to the distributed nature of the Domain Name Service, DNS resolution may entail multiple requests and round-trips across the internet, requiring seconds and sometimes even longer.
  - A standard remedy is to introduce caching: URLs for which we have recently performed DNS lookups are likely to be found in the DNS cache, avoiding the need to go to the DNS servers again.
- Once a request is made by a crawler to the Domain Name Service, other crawler threads at that node are blocked until the first request is completed.
- To circumvent this, most web crawlers implement their own DNS resolver as a component of the crawler.

URL frontier

- The URL frontier at a node is given a URL by its crawl process (or by the host splitter of another crawl process).
- It maintains the URLs and provides in some order them whenever a crawler thread seeks a URL.
- Two important considerations govern the order in which URLs are returned by the frontier.
  - First, high-quality pages that change frequently should be prioritized for frequent crawling.
  - The second consideration is politeness: we must avoid repeated fetch requests o a host within a short time span.
- Goals of URL frontier are to ensure that
  i. only one connection is open at a time to any host;
  ii. a waiting time of a few seconds occurs between successive requests to a host and
  iii. high-priority pages are crawled preferentially.



► **Figure 20.3** The URL frontier. URLs extracted from already crawled pages flow in at the top of the figure. A crawl thread requesting a URL extracts it from the bottom of the figure. En route, a URL flows through one of several *front queues* that manage its priority for crawling, followed by one of several *back queues* that manage the crawler's politeness.

Front Queues

- The two major sub-modules are a
  - set of *F front queues* in the upper portion of the figure, and a
  - set of *B back queues* in the lower part;
  - all of these are FIFO queues.
- The front queues implement the prioritization, while the back queues implement politeness.
- When a URL is added to the frontier, a *prioritizer* first assigns to the URL an integer priority *i* between 1 and *F* based on its fetch history.
  - a document that has exhibited frequent change would be assigned a higher priority.
  - URLs from news services may always be assigned the highest priority.
- Now that it has been assigned priority *i*, the URL is now appended to the *i*th of the front queues.

Back queues

- Each of the *B* back queues maintains the following :

(i) it is nonempty while the crawl is in progress and

(ii) it only contains URLs from a single host.

- An auxiliary table *T* is used to maintain the mapping from hosts to back queues.
- Whenever a back-queue is empty and is being re-filled from a front-queue, table *T* must be updated accordingly.

Host	Back queue
stanford.edu	23
microsoft.com	47
acm.org	12

▶ **Figure 20.4** Example of an auxiliary hosts-to-back queues table.

- A heap with one entry is maintained for each back queue, the entry being the earliest time *te* at which the host corresponding to that queue can be contacted again.

Distributing indexes

- Two obvious alternative index implementations :
  - *partitioning by terms*, also known as global index organization, and
  - *partitioning by documents*, also know as local index organization.
- *Partitioning by terms*
- The dictionary of index terms is partitioned into subsets, each subset residing at a node.
- Along with the terms at a node, we keep the postings for those terms.
- A query is routed to the nodes corresponding to its query terms.
- In principle, this allows greater concurrency.
- Multi-word queries require the sending of long postings lists between sets of nodes for merging, and the cost of this can outweigh the greater concurrency.
- Load balancing the partition can drift with time or exhibit sudden bursts.
- Makes implementation of dynamic indexing more difficult.

- *Partitioning by documents*
- Each node contains the index for a subset of all documents.
- Each query is distributed to all nodes, with the results from various nodes being merged before presentation to the user.
- This strategy trades more local disk seeks for less inter-node communication.
- One difficulty in this approach is that global statistics used in scoring – such as idf – must be computed across the entire document collection even though the index at any single node only contains a subset of the documents.

How do we decide the partition of documents to nodes?

- One simple approach would be to assign all pages from a host to a single node.
- A hash of each URL into the space of index nodes results in a more uniform distribution

- *At query time,*
- the query is broadcast to each of the nodes, each node sends each top k results which are merged to find the top k documents for the query

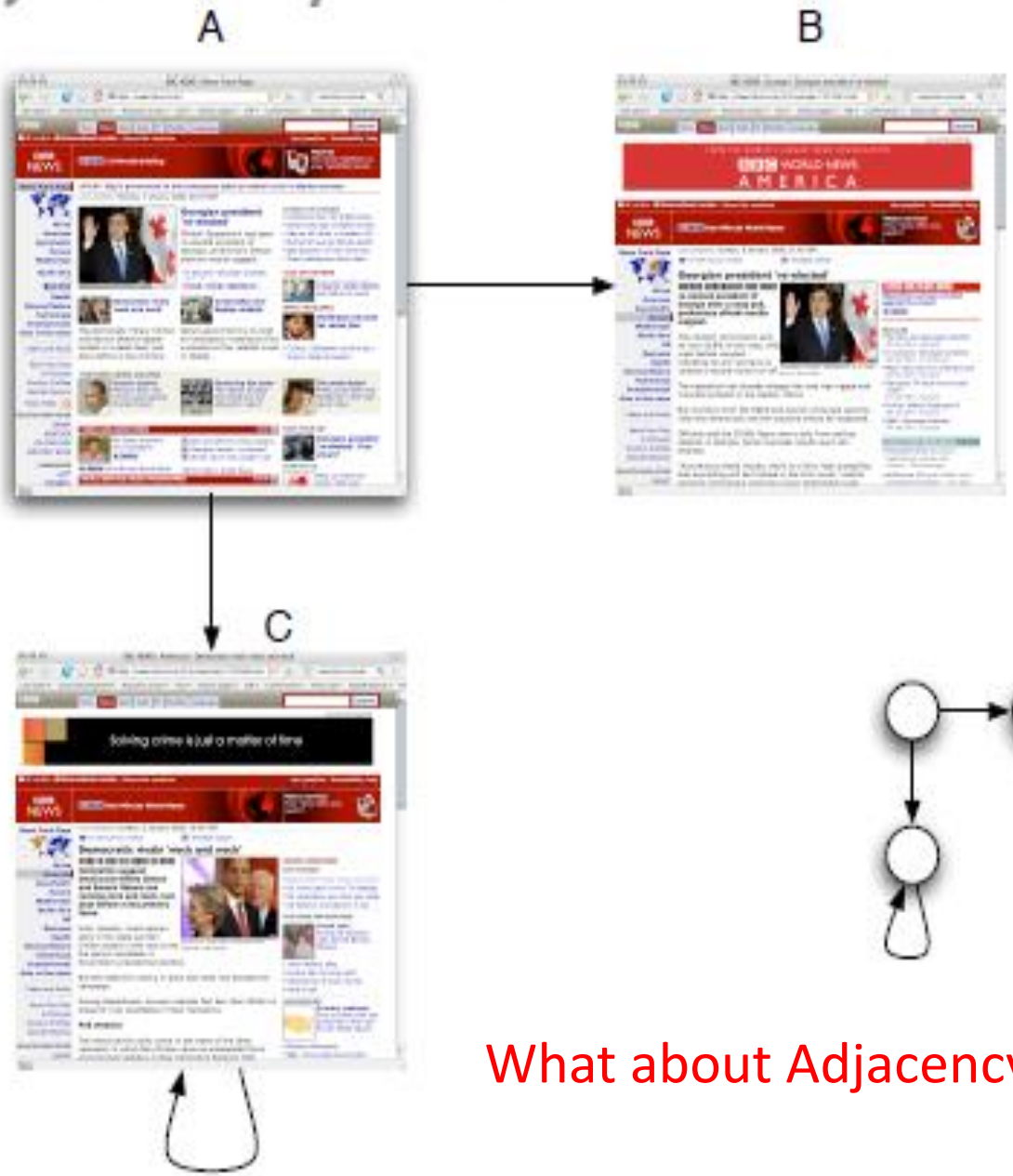
A common implementation heuristic:

- Partition the document collection into indexes of documents that are more likely to score highly on most queries and
- low-scoring indexes with the remaining documents
- Only search the low-scoring indexes when there are too few matches in the high-scoring indexes

Connectivity servers

- Web search engines require a *connectivity server* that supports fast *connectivity queries* on the web graph.
- Typical connectivity queries are
  - *Which URLs link to a given URL?* and
  - *Which URLs does a given URL link to?*
- Assume each URL represented by an integer
- i.e. 4 billion web pages each with ten links to other pages need 32 bits per URL.
- We build an *adjacency table* : it has a row for each web page, with the rows ordered by the corresponding integers.
- The row for any page *p* contains a sorted list of integers, each corresponding to a web page that links to *p*.
- This table permits us to respond to queries of the form *which pages link to p?*
- In similar fashion we build a table whose entries are the pages linked to by *p*.

Adjacency Matrix - Conceptual Idea



	A	B	C
A	0	1	1
B	0	0	0
C	0	0	1

What about Adjacency Lists instead?

```
1: www.stanford.edu/alchemy
2: www.stanford.edu/biology
3: www.stanford.edu/biology/plant
4: www.stanford.edu/biology/plant/copyrigl
5: www.stanford.edu/biology/plant/people
6: www.stanford.edu/chemistry
```

Figure 20.5 A lexicographically ordered set of URLs.

```
1: 1, 2, 4, 8, 16, 32, 64
2: 1, 4, 9, 16, 25, 36, 49, 64
3: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
4: 1, 4, 8, 16, 25, 36, 49, 64
```

Figure 20.6 A four-row segment of the table of links.

- In a *lexicographic* ordering of all URLs, we treat each URL as an alphanumeric string and sort these strings
- To each URL, we assign its position in this ordering as the unique identifying integer.
- Websites have a set of links from each page in the site to a fixed set of pages on the site
- We adopt the following strategy: we walk down the table, encoding each table row in terms of the seven preceding rows.

- 1, 2, 4, 8, 16, 32, 64
- 1, 4, 9, 16, 25, 36, 49, 64
- 1, 2, 3, 5, 6, 13, 21, 34, 55, 89, 144
- 1, 4, 8, 16, 25, 36, 49, 64
  - Encode this as row(-2), -URL(9), +URL(8)

- The use of only the seven preceding rows has two advantages:

(i) the offset can be expressed with only 3 bits; and

(ii) fixing the maximum offset to a small value like seven avoids having to perform an expensive search among many candidate prototypes in terms of which to express the current row.

What if none of the preceding seven rows is a good prototype for expressing the current row?

Space reduction can be obtained using gap encodings