DATA20021

University of Helsinki, Department of Computer Science

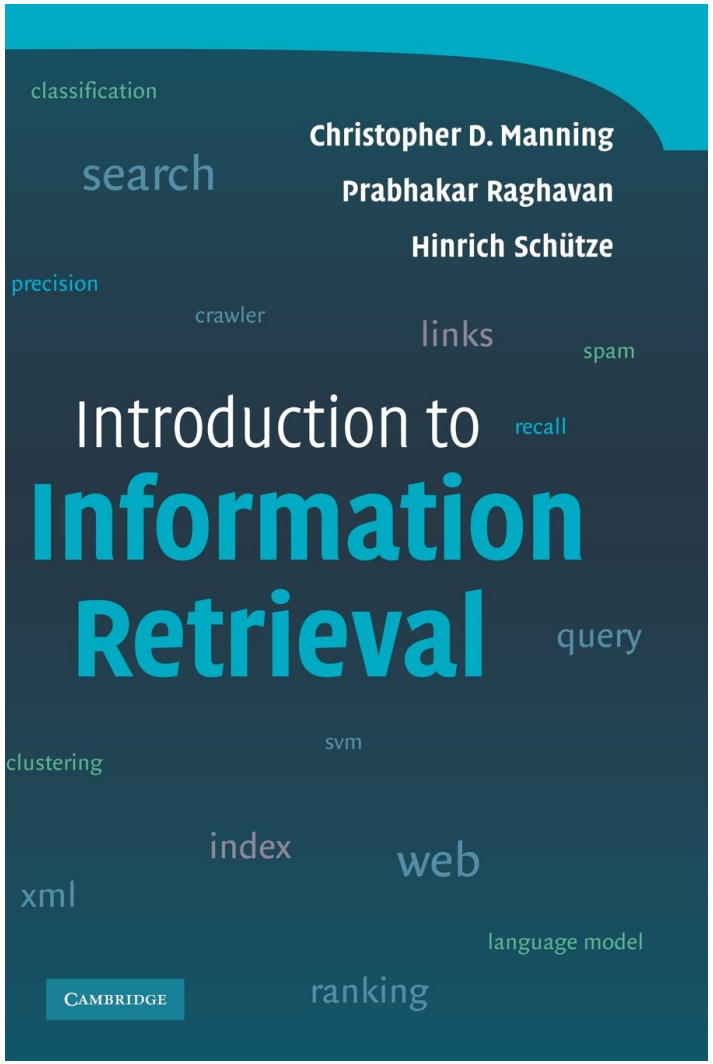
Information Retrieval

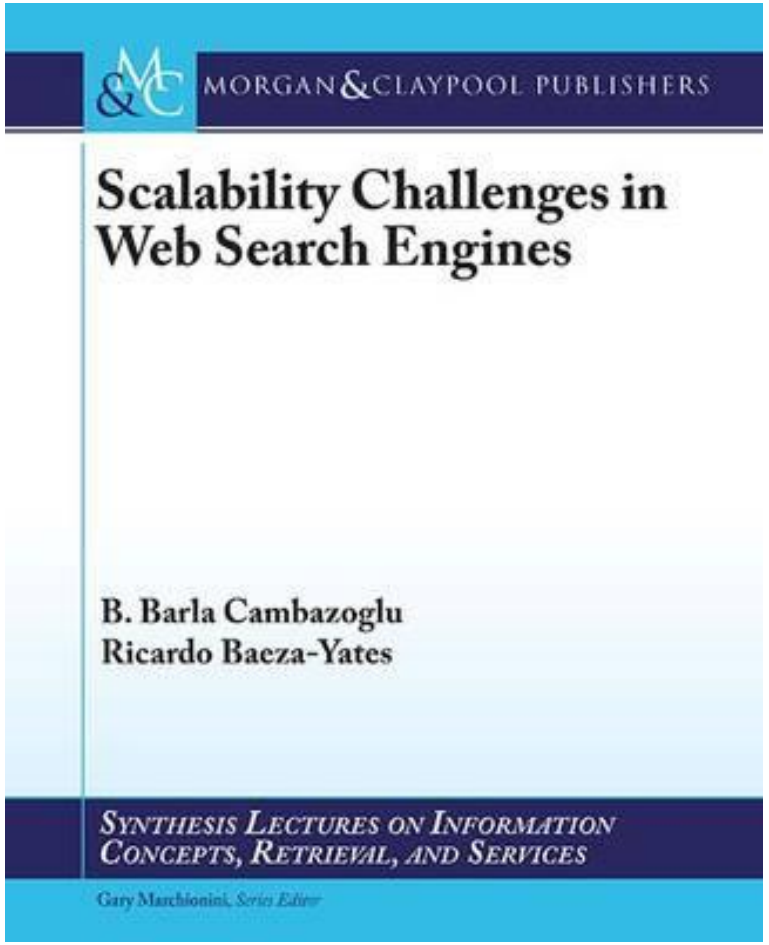
Lecture 5: Web Crawling

Simon J. Puglisi

puglisi@cs.helsinki.fi

Spring 2020


classification
search
Christopher D. Manning
Prabhakar Raghavan
Hinrich Schütze
precision
crawler
links
spam
Introduction to
recall
Information
Retrieval
query
svm
clustering
index
web
xml
language model
ranking
CAMBRIDGE


MORGAN & CLAYPOOL PUBLISHERS
Scalability Challenges in Web Search Engines
B. Barla Cambazoglu
Ricardo Baeza-Yates
SYNTHESIS LECTURES ON INFORMATION CONCEPTS, RETRIEVAL, AND SERVICES
Gary Marchionini, Series Editor

Today's lecture…

16.1: Introduction to Indexing
- Boolean Retrieval model
- Inverted Indexes

21.1: Index Compression
- unary, gamma, variable-byte coding
- (Partitioned) Elias-Fano coding (used by Google, facebook)

23.1: Index Construction
- preprocessing documents prior to search
- building the index efficiently

28.1: Web Crawling
- getting documents off the web at scale
- architecture of a large scale web search engine

30.1: Query Processing
- scoring and ranking search results
- Vector-Space model

- Web search engines create *web repositories*
  - They cache the Web on their local machines

- Web repositories provide fast access to copies of the pages on the Web, allowing faster indexing and better search quality

- A search engine aims to minimize the potential differences between it's local repo and the Web
  - Coverage and freshness allow better quality answers
  - Very challenging due to fast and continuous evolution of the web: huge changes in pages and content every second

- The Web repository maintains only the most recently crawled versions of web pages
  - Raw HTML, but compressed, on a filesystem (not a DBMS)
  - Also a catalog containing location on disk, size, timestamp

- Mechanisms for both bulk and random access to stored pages are provided
  - Bulk access is used, e.g., by the indexing system
  - Random access for, e.g., query-biased snippet generation

Google navy blue mittens

Web Images Videos News More Search tools

About 979,000 results (0.49 seconds)

Images for navy blue mittens

Report images

More images for navy blue mittens

Team USA - Go USA Olympics Knit Mittens - Navy Blue ...

www.amazon.com/Team-USA-Olympics-Knit-**Mittens**/.../B00GLEIY8O

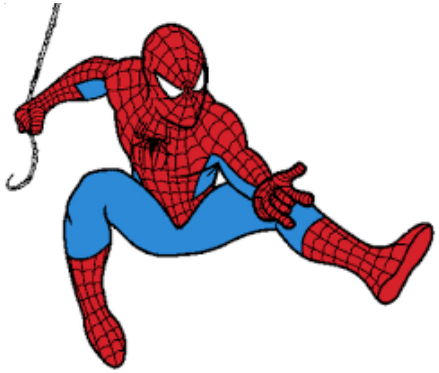
★★★½☆ Rating: 3.6 - 8 reviews

It takes years of training for Team USA athletes to get to the highest level of competition. Support their hard work and dedication today by purchasing your very ...

Popular items for navy blue mittens on Etsy

https://www.etsy.com/market/**navy_blue_mittens**

Shop outside the big box, with unique items for **navy blue mittens** from thousands of independent designers and vintage collectors on Etsy.



- A *web crawler* (or *(ro)bot* or *spider*) is the subsystem of the search engine that downloads pages from the Web for storage in the web repository

- A large-scale web crawler is responsible for two different tasks that are performed in tandem
  - Task1: locate previously undiscovered URLs by iteratively following hyperlinks between web pages
    - Aim: increase coverage
  - Task 2: refetch from the Web contents of pages that are currently stored in the repository
    - Aim: maintain freshness
- Crawlers are faced with a large number of scalability and efficiency challenges, many stemming from factors external to the crawlers themselves

- In search engine companies, multiple crawlers are operational at a given point in time…

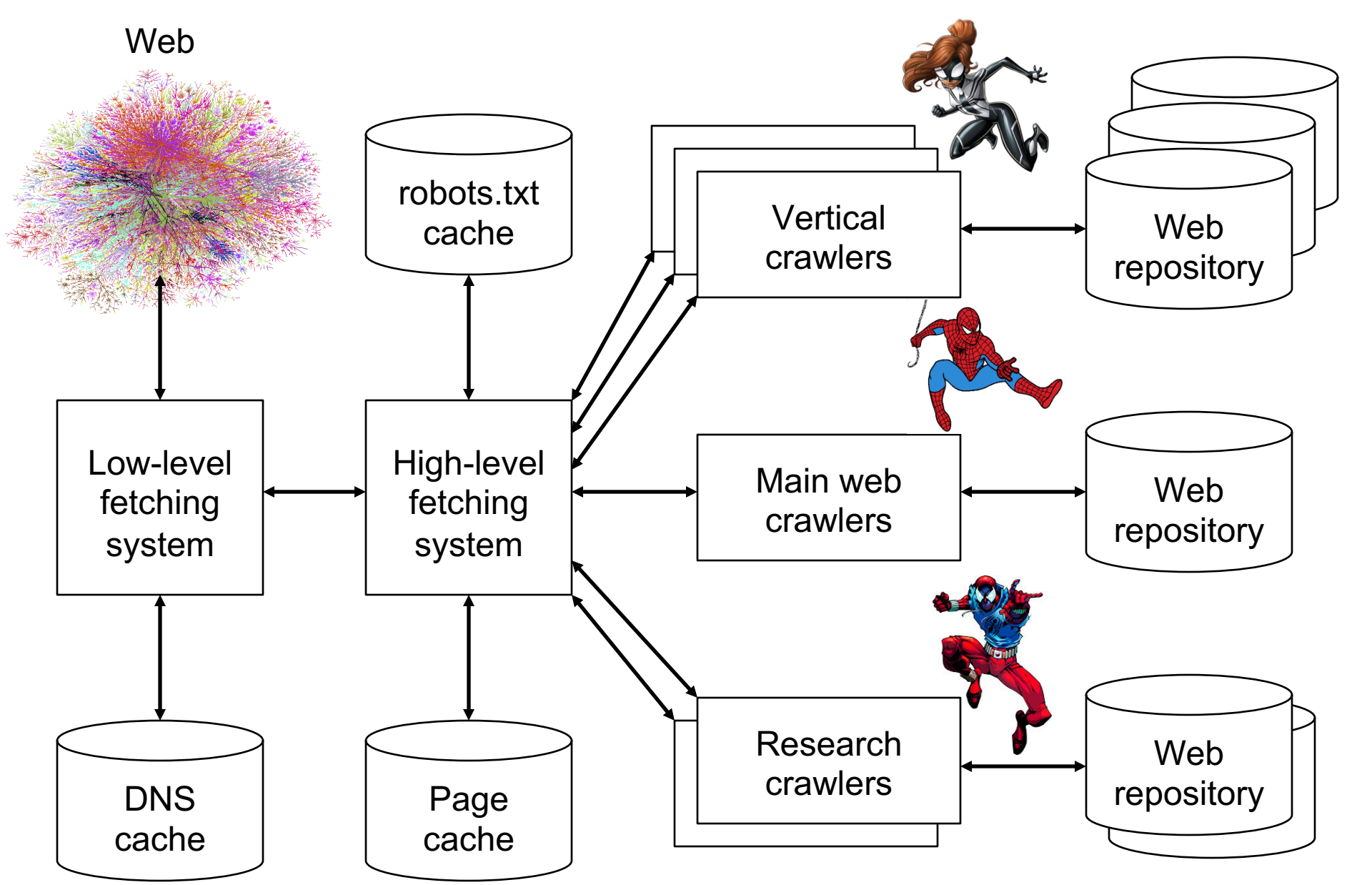
- …by different departments that work independently and may serve different purposes
  - crawler for main search engine
  - some vertical crawlers (for news, say)
  - some research/experimental crawlers

- Multiple, uncoordinated web crawlers pose risks
  - Saturation of the companies network bandwidth
  - Overloading websites with crawl requests (DoS)

- To remedy these coordination problems, search engines usually implement a *web page fetching system*
  - shared by multiple crawlers

- WPFS is a gateway between crawlers and the Web
  - Whenever a crawler needs to fetch a page from the Web, it issues a request to the fetching system with a URL

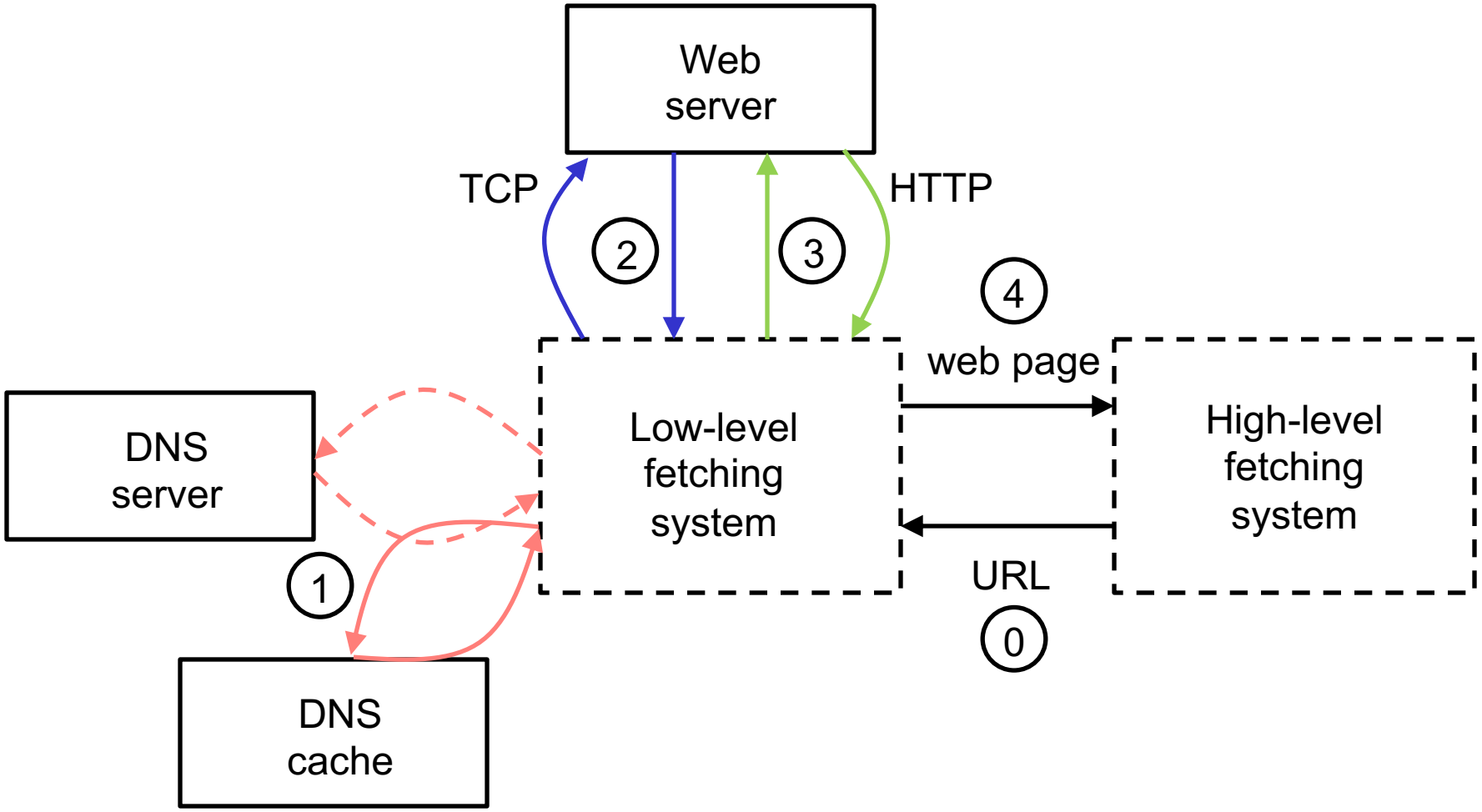
- In practice, usually composed of two subsystems:
  - High-level fetching system (HLFS): between crawlers & the
  - Low-level fetching system (LLFS): between the high-level fetching system & the Web



Two-level fetching system shared by different crawlers

- The low-level fetching system implements basic network operations, such as resolving DNS of URLs to IP addresses and fetching page content from Web

www.newyorker.com/ ⇨ 151.101.0.239



Low level process involved in fetching a page from the Web

- Main input to LLFS is a URL, provided by the HLFS
  - URL is parsed to extract web server's domain name
- DNS server contacted to map domain name to IP address
  - DNS mapping can be a bottleneck for web crawlers
  - (domain name, IP) pairs are cached locally with an expiry timestamp
  - prevents repetitive access to the DNS server to resolve same domain
  - caching significantly reduces the overhead of DNS resolution.
- With IP address, LLFS opens a TCP connection to web server, then an HTTP connection.
  - Page content downloaded over HTTP connection is passed to HLFS
  - Multiple HTTP requests may be issued to web server over single TCP connection

- HLFS implements various crawling mechanisms and policies applicable to all operational crawlers in the search engine
  - Implements *throttling mechanism* to prevent LLFS from being flooded with too many download requests
  - Implements server-, host-, *politeness constraints*, to prevent websites being overloaded by page requests

- HLFS also ensures LLFS establishes only a single connection to a particular server at any given time
  - Assign requested webpages to a download queue
  - All requests bound for same website added to same queue
  - Each queue to a unique crawling thread

- HLFS usually maintains a cache of recently downloaded pages
  - Different crawlers take pages from cache if possible

- High-level fetching system also ensures that the *robots exclusion protocol* is obeyed by every crawler
- Many websites publish a `robots.txt` file
- `robots.txt` includes instructions guiding crawlers as to how they should crawl pages on the website
- `robots.txt` files are maintained in a cache by the HLFS (with a timestamp) to avoid retrieving it for every crawling request

```
#robots.txt file

User-agent: googlebot          #all services
Disallow: /private/            #disallow this directory

User-agent: googlebot-news     #only the news service
Disallow: /                    #on everything

User-agent: *                  #all robots
Disallow: /something/          #on this directory

User-agent: *                  #all robots
Crawl-delay: 10                #wait at least 10 secs

Disallow: /dir1/               #disallow this directory
Disallow: /dir1/myfile.html    #allow a subdirectory

Host: www.example.com          #use this mirror
```

An example robots.txt file.

- Sometimes `robots.txt` contains a link to a *sitemap file*

- Sitemaps are XML files created by webmasters to inform web crawlers about URLs served at the site
  - can include metadata about the served web pages
  - e.g., last update time, change frequency, relative importance

- Crawlers can use this information to prioritize downloads, increase coverage, and improve freshness

```
<!- sitemap.xml file -->

<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/
sitemap/0.9">

<url>
<loc>http://www.test.com/</loc>
<lastmod>2020-01-25</lastmod>
<changefreq>weekly</changefreq>
<priority>0.65</priority>
</url>

<url>
...
</url>
...

</urlset>
```

An example sitemap.xml file.

- Having a shared web page fetching system:
  - Relieves burden of implementing same functionality in different crawlers
  - Provides robust/highly-tuned platform for efficient downloads (and for avoiding unnecessary downloads)

- Given availability of the low- and high-level fetching systems a crawler's responsibility is reduced to:
  - Increasing coverage (via downloading new pages), and
  - Refreshing the repository by fetching already downloaded pages

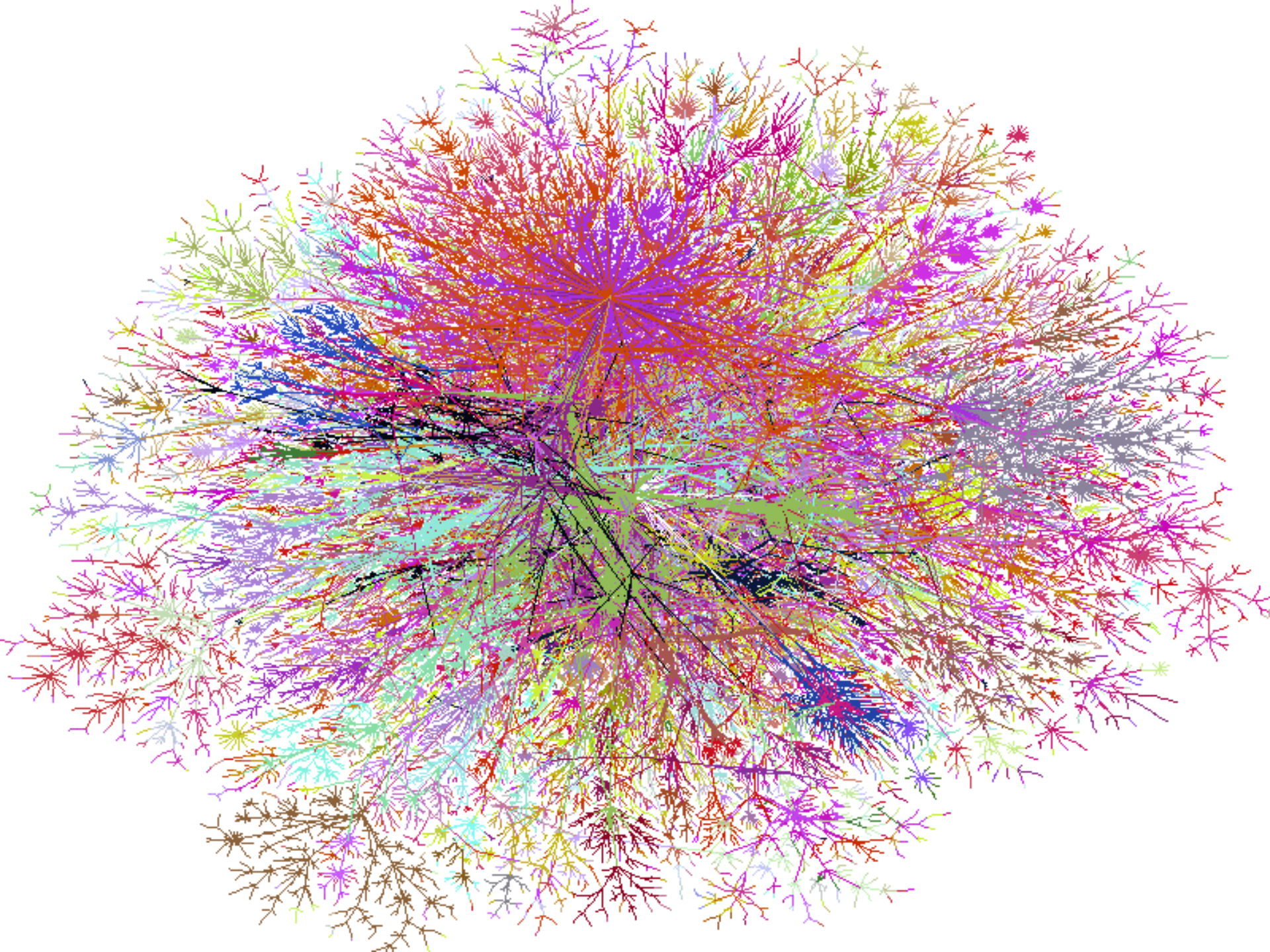
- To this end, the crawler maintains two separate (logical) download queues for discovery and refresh
  - The main algorithmic challenge for designers is then to devise ways to effectively prioritise the URLs in these queues

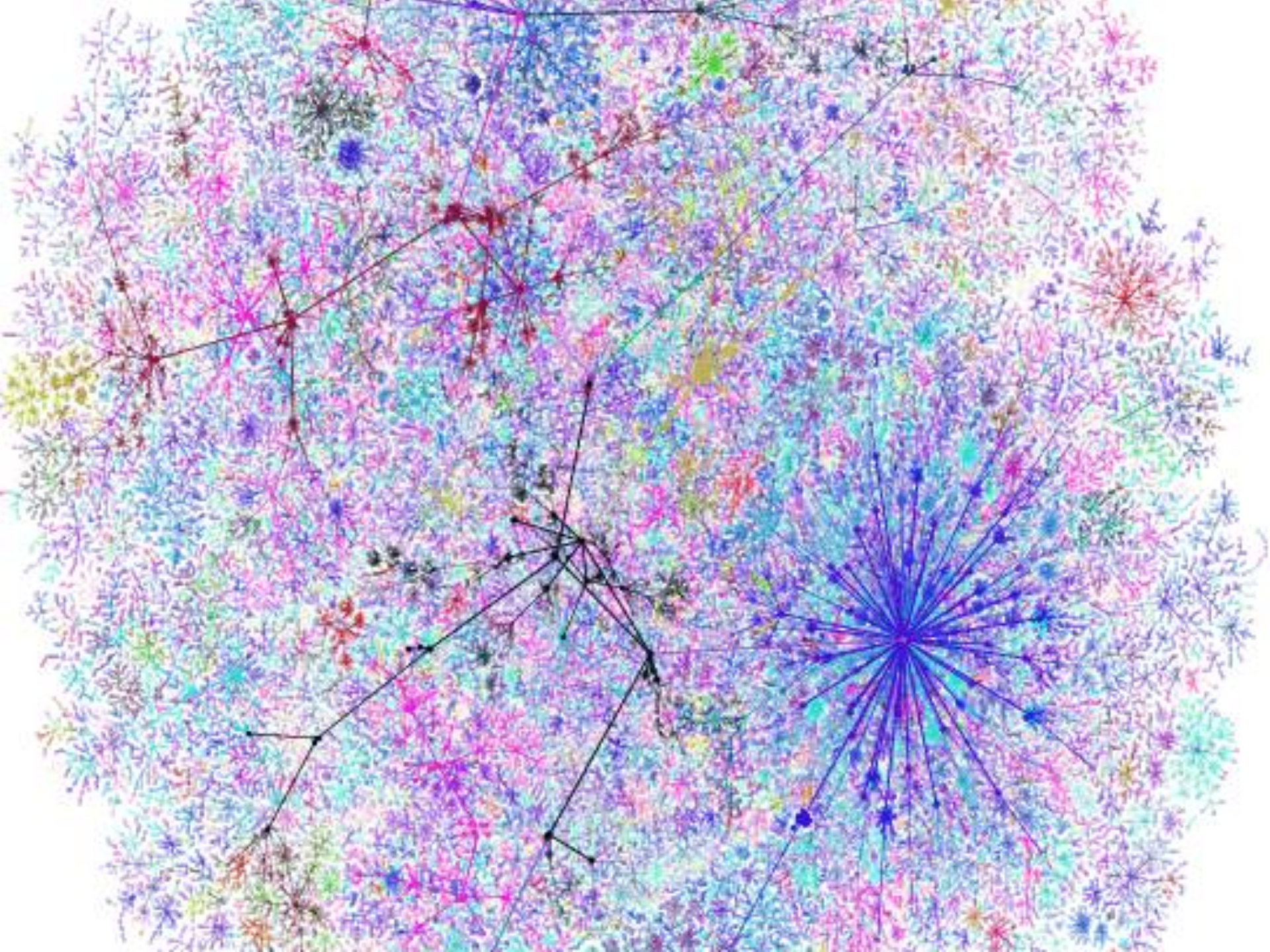
Extending the Web Repository…

- Crawlers rely on hyperlinks to discover new URLs

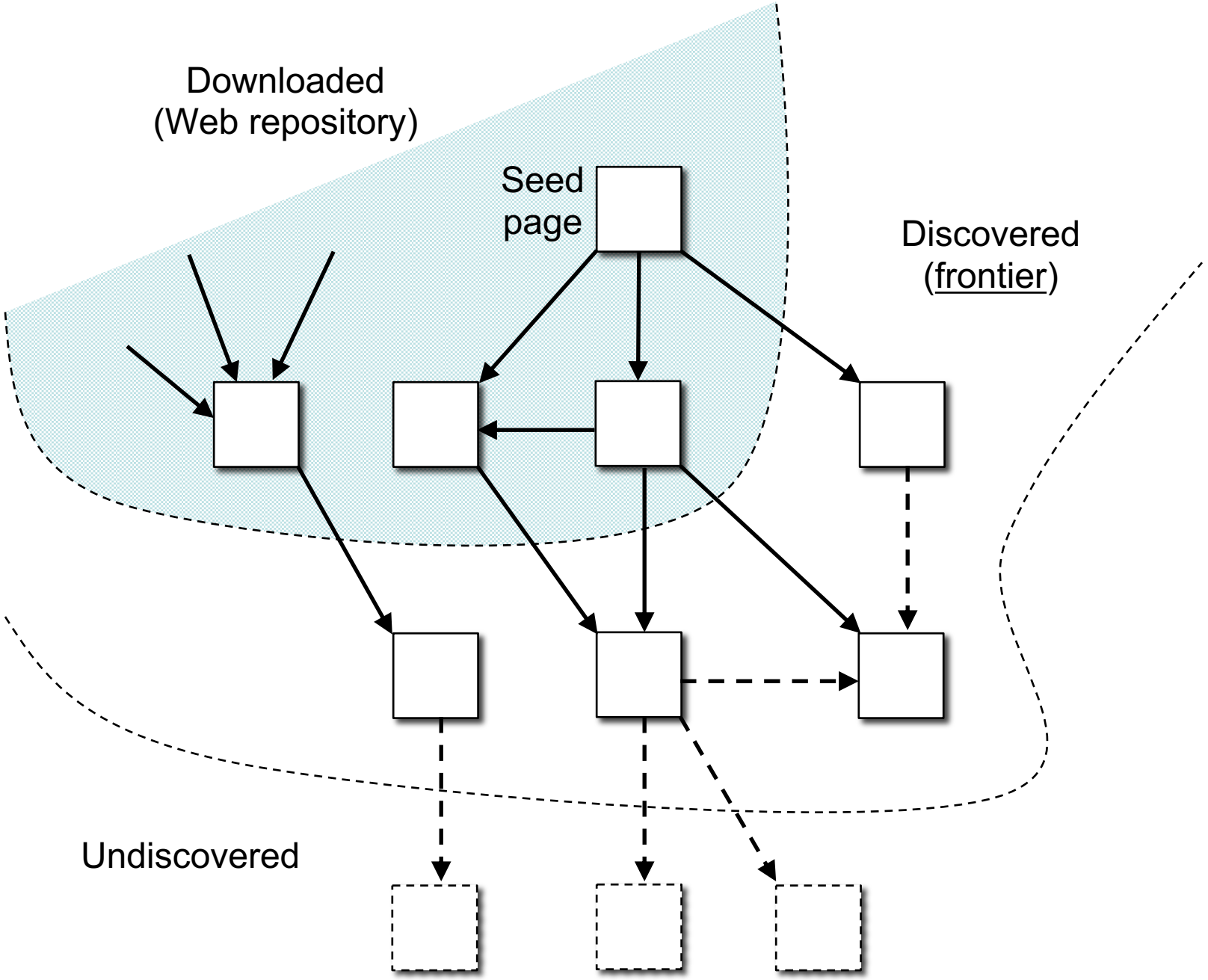
- Web pages and underlying hyperlink structure can be viewed as a directed graph (the *Web graph*)
  - Each vertex corresponds to a web page
  - Hyperlink to another page → directed edge between the corresponding vertices of the graph
  - A crawler discovers new pages by traversing the vertices connected by edges of the graph

- Initially crawler needs some *seed* web pages that act as entry points to the Web graph
  - Usually so-called *hub* pages, which point to a large number of potentially important web pages





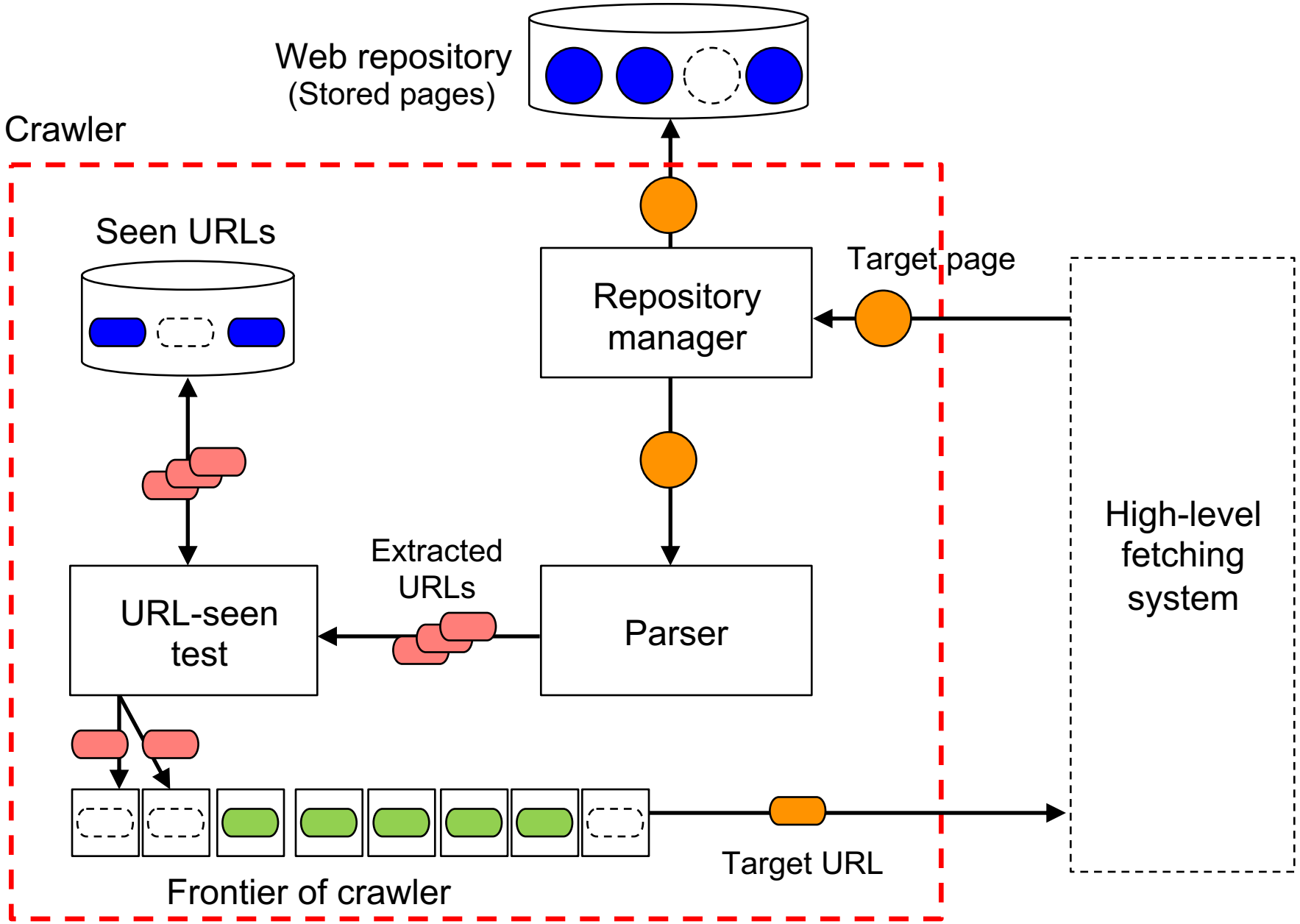
- The *discovery process* splits pages on the Web into three disjoint sets:
  1. Pages whose content is already downloaded by the crawler
  2. Pages whose URLs are discovered (by following hyperlinks pointing to those pages), but whose content is not yet downloaded (the *frontier* of the crawler)
  3. Pages whose URLs are not yet discovered


Downloaded
(Web repository)
Seed
page
Discovered
(frontier)
Undiscovered

- Early web crawlers performed discovery in batches:
  - In each session/batch, web pages were downloaded until a time limit of page limit was reached
  - Motivated by scalability: throughout web crawling, certain data structures grow and eventually "choke" the crawler

- Modern large-scale web crawlers perform discovery incrementally and continuously…

- In each iteration:
  1. Crawler selects a target URL from the URLs currently available on its Frontier
  2. URL goes to high-level FS for download
  3. Returned HTML content goes to Repository Manager, which decides if the page enters the repo
  4. Content then goes to the Parser
     - URLs contained in page are extracted and normalized
     - Normaliztion: lowercase, relative to absolute paths, "www" prefix added or removed depending on redirections, et c.
  5. Normalized URLs are looked up in a data structure that maintains all URLs seen so far: *URL seen test*
  6. Previously unseen URLs are added to the data structure, expanding the crawler's Frontier


Web repository
(Stored pages)
Crawler
Seen URLs
Repository
manager
Target page
High-level
fetching
system
Extracted
URLs
URL-seen
test
Parser
Target URL
Frontier of crawler

- Data structures used in the discovery process are updated in an incremental fashion and keep growing as new pages are downloaded and links discovered
  - Efficiency is important

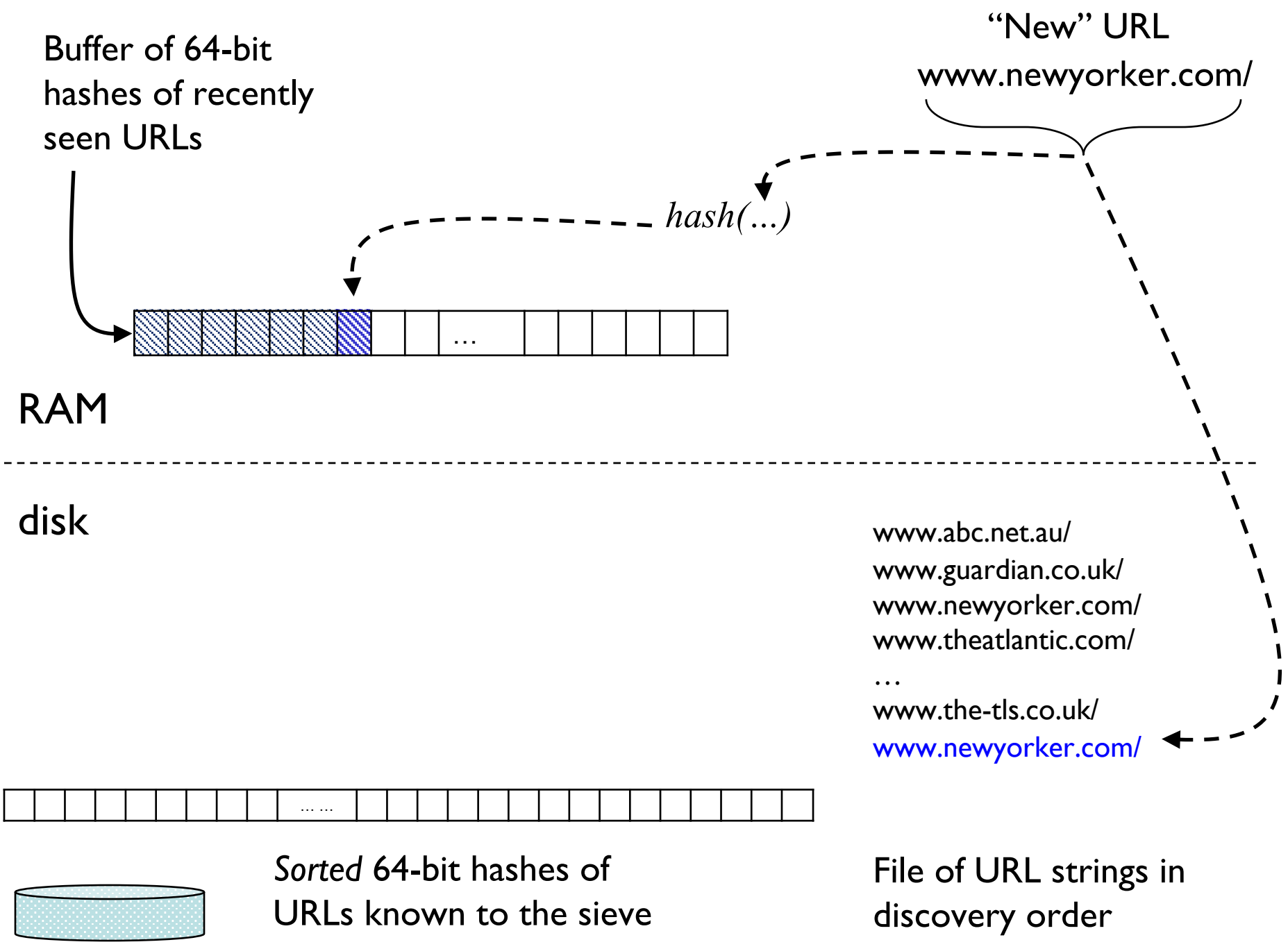
- Given the vast number of URLs that need to be maintained, these data structures will not fit entirely in memory

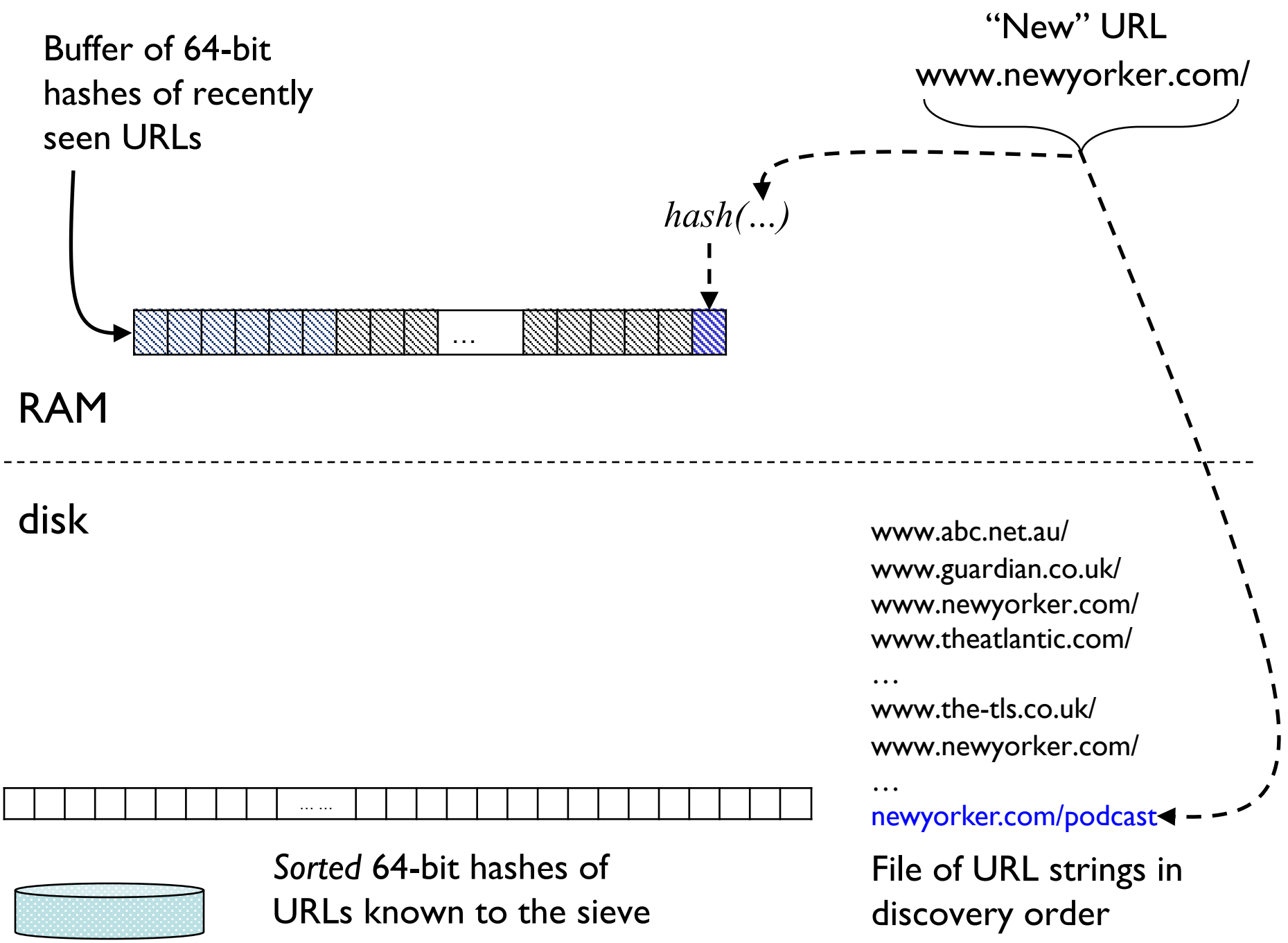
- The URL-seen-test data structure needs to be implemented with care…

- A *sieve* is a "queue with memory"
  - Provides *enqueue* and *dequeue* primitives (like a queue)
  - Each element enqueued will eventually be dequeued
  - *Sieve guarantees that if an element is enqueued multiple times it will only be dequeued once*

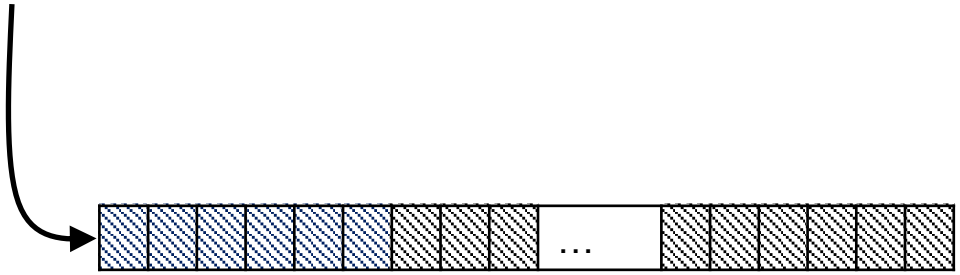
- Sieves of URLs are fundamental d.s. for crawlers
  - Main implementation issue: unbounded exponential growth in the number of discovered URLs

- Easy to write enqueued URLs to disk, guaranteeing a URL is only returned once requires ad hoc d.s.
  - Standard dictionary implementations (hash tables, tries, B-trees) require too much RAM


Buffer of 64-bit hashes of recently seen URLs
"New" URL
www.newyorker.com/
hash(...)
RAM
disk
www.abc.net.au/
www.guardian.co.uk/
www.newyorker.com/
www.theatlantic.com/
…
www.the-tls.co.uk/
www.newyorker.com/
Sorted 64-bit hashes of URLs known to the sieve
File of URL strings in discovery order


Buffer of 64-bit hashes of recently seen URLs
"New" URL
www.newyorker.com/
hash(...)
RAM
disk
www.abc.net.au/
www.guardian.co.uk/
www.newyorker.com/
www.theatlantic.com/
…
www.the-tls.co.uk/
www.newyorker.com/
…
newyorker.com/podcast
Sorted 64-bit hashes of URLs known to the sieve
File of URL strings in discovery order

Buffer of 64-bit hashes of recently seen URLs

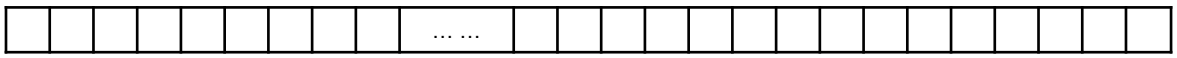


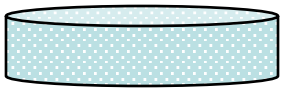
RAM

---

disk

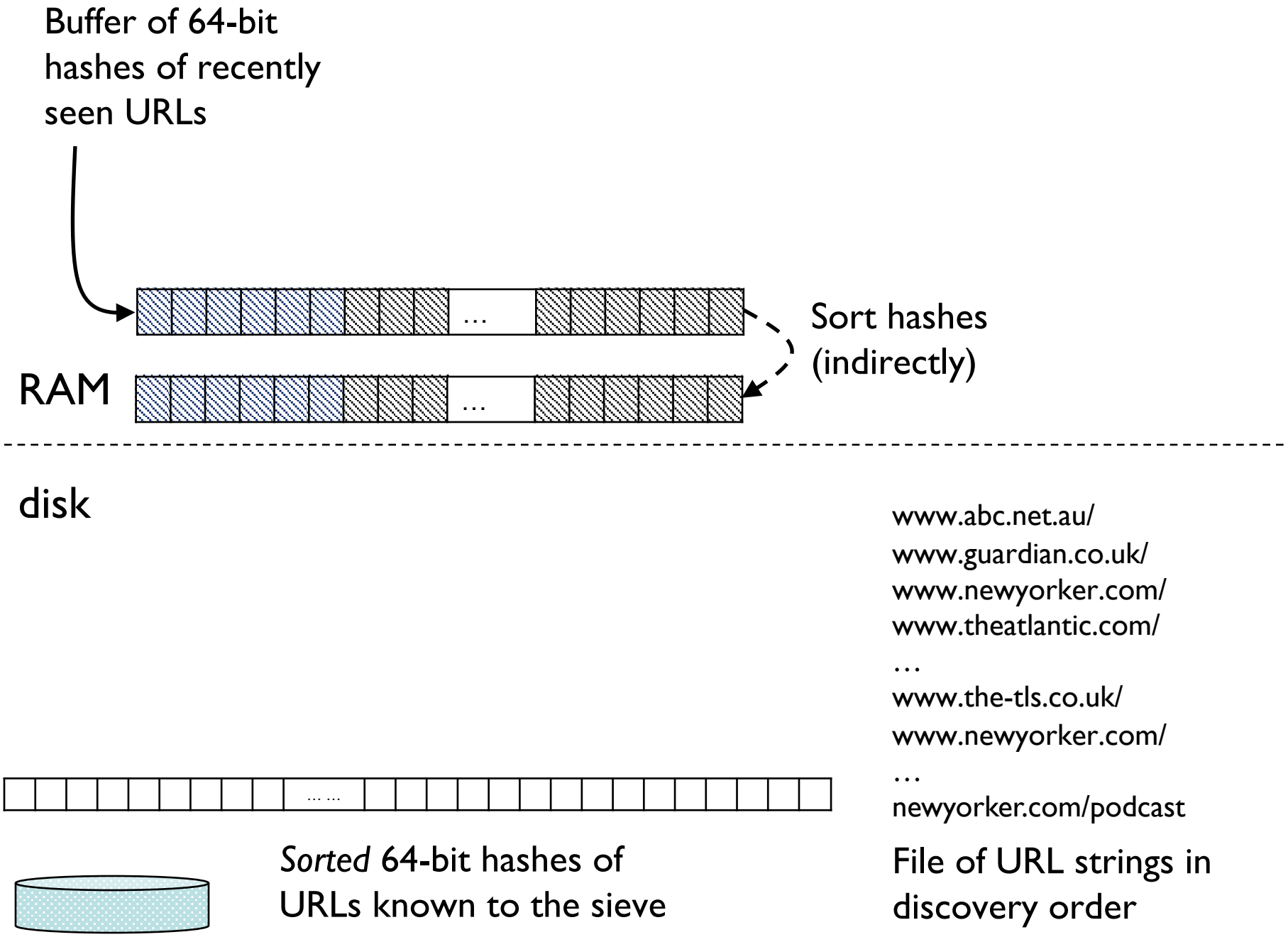
www.abc.net.au/
www.guardian.co.uk/
www.newyorker.com/
www.theatlantic.com/
…
www.the-tls.co.uk/
www.newyorker.com/
…
newyorker.com/podcast



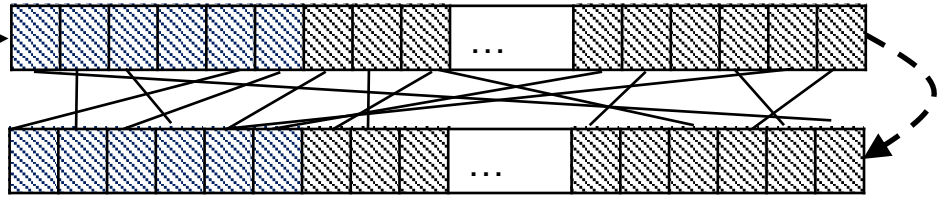


Sorted 64-bit hashes of URLs known to the sieve

File of URL strings in discovery order


Buffer of 64-bit hashes of recently seen URLs
Sort hashes (indirectly)
RAM
disk
www.abc.net.au/
www.guardian.co.uk/
www.newyorker.com/
www.theatlantic.com/
…
www.the-tls.co.uk/
www.newyorker.com/
…
newyorker.com/podcast
Sorted 64-bit hashes of URLs known to the sieve
File of URL strings in discovery order

Buffer of 64-bit hashes of recently seen URLs



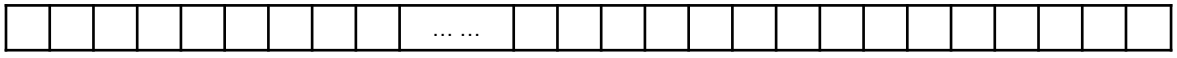
Sort hashes (indirectly)

RAM

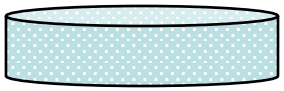
---

disk

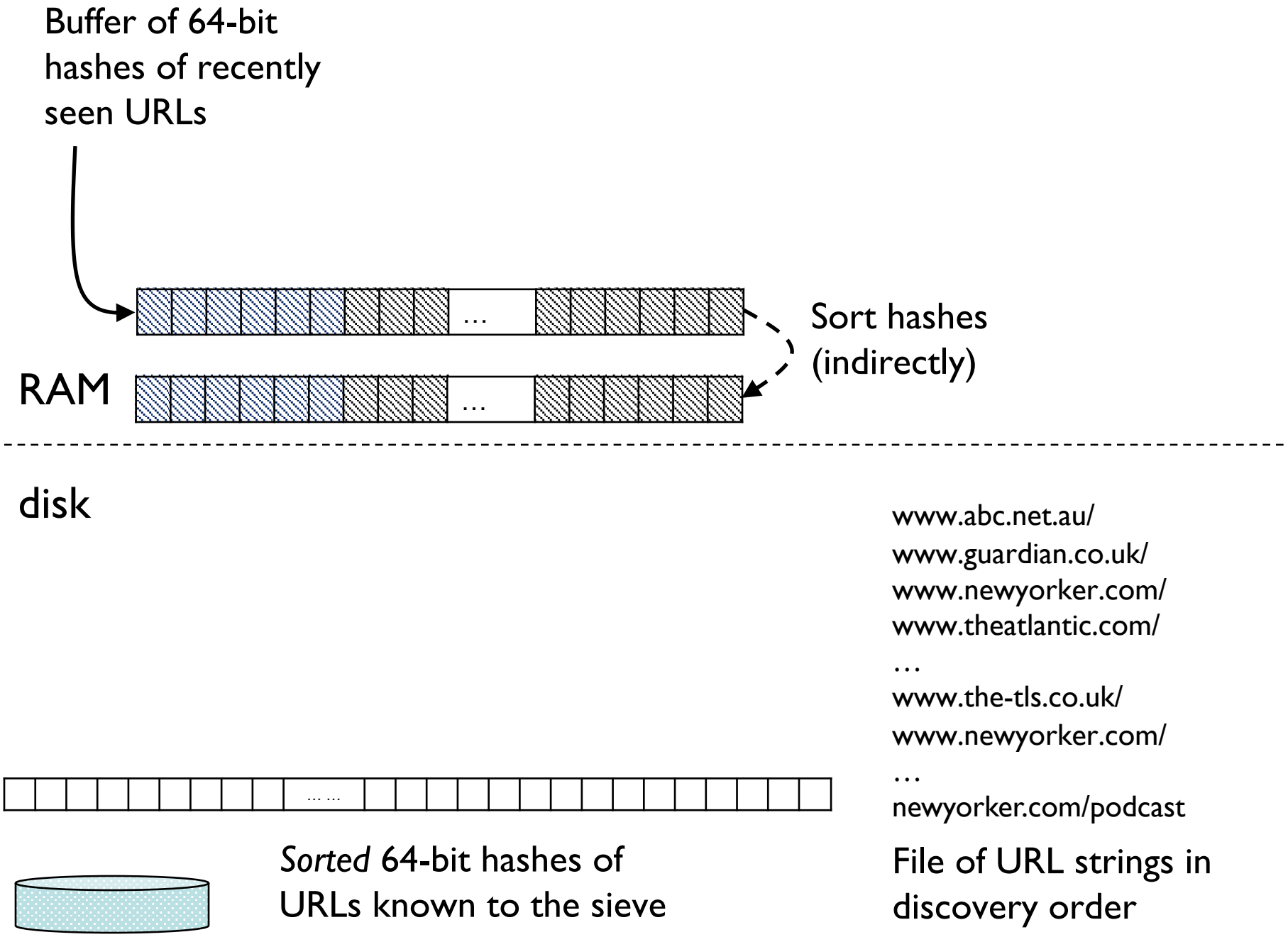
www.abc.net.au/
www.guardian.co.uk/
www.newyorker.com/
www.theatlantic.com/
…
www.the-tls.co.uk/
www.newyorker.com/
…
newyorker.com/podcast

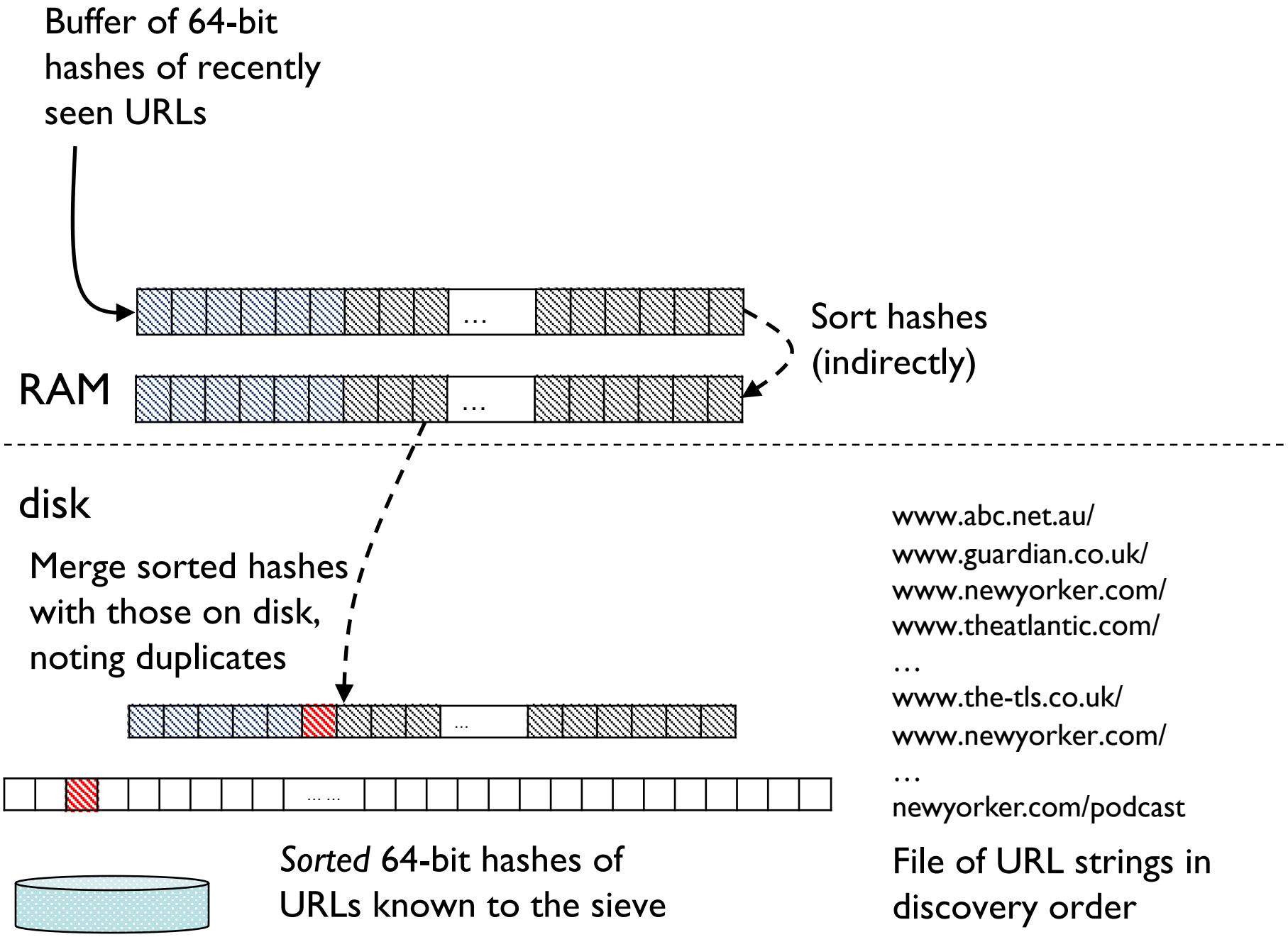


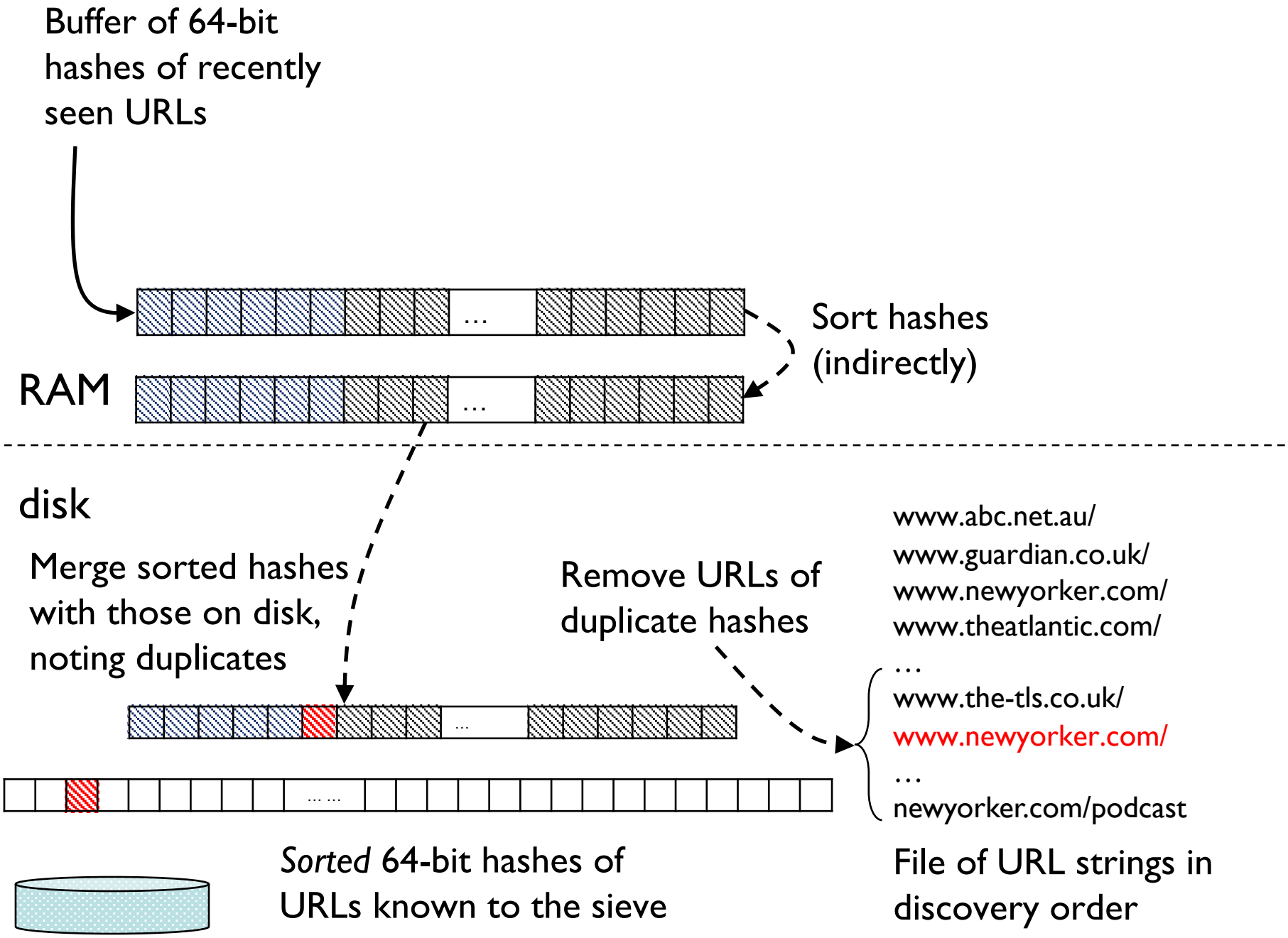
Sorted 64-bit hashes of URLs known to the sieve

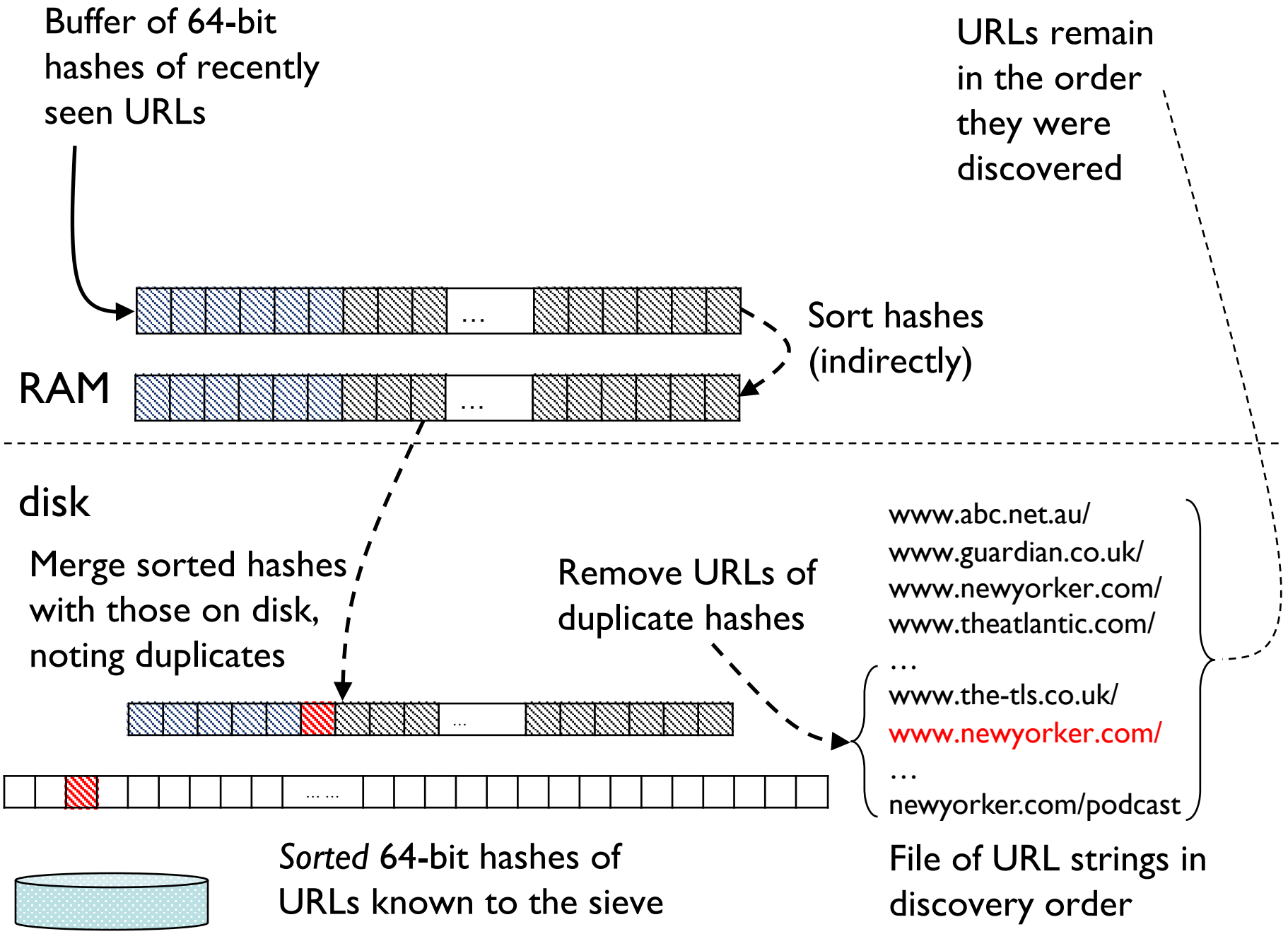


File of URL strings in discovery order


Buffer of 64-bit hashes of recently seen URLs
Sort hashes (indirectly)
RAM
disk
www.abc.net.au/
www.guardian.co.uk/
www.newyorker.com/
www.theatlantic.com/
…
www.the-tls.co.uk/
www.newyorker.com/
…
newyorker.com/podcast
Sorted 64-bit hashes of URLs known to the sieve
File of URL strings in discovery order


Buffer of 64-bit hashes of recently seen URLs
Sort hashes (indirectly)
RAM
disk
Merge sorted hashes with those on disk, noting duplicates
www.abc.net.au/
www.guardian.co.uk/
www.newyorker.com/
www.theatlantic.com/
…
www.the-tls.co.uk/
www.newyorker.com/
…
newyorker.com/podcast
Sorted 64-bit hashes of URLs known to the sieve
File of URL strings in discovery order


Buffer of 64-bit hashes of recently seen URLs
Sort hashes (indirectly)
RAM
disk
Merge sorted hashes with those on disk, noting duplicates
Remove URLs of duplicate hashes
www.abc.net.au/
www.guardian.co.uk/
www.newyorker.com/
www.theatlantic.com/
…
www.the-tls.co.uk/
www.newyorker.com/
…
newyorker.com/podcast
Sorted 64-bit hashes of URLs known to the sieve
File of URL strings in discovery order


Buffer of 64-bit hashes of recently seen URLs
URLs remain in the order they were discovered
Sort hashes (indirectly)
RAM
disk
Merge sorted hashes with those on disk, noting duplicates
Remove URLs of duplicate hashes
www.abc.net.au/
www.guardian.co.uk/
www.newyorker.com/
www.theatlantic.com/
…
www.the-tls.co.uk/
www.newyorker.com/
…
newyorker.com/podcast
Sorted 64-bit hashes of URLs known to the sieve
File of URL strings in discovery order

- All operations in the sieve require only sequential (i.e. scanning) access to all files involved
  - Scanning disks is efficient due to continuous disk head movement and hardware prefetching

- Points of note…

- 64-bit fingerprints can give rise to collisions with significant probability when crawling more than a few hundred billion URLs per crawler
  - Bigger fingerprint sizes OK, but complicate implementation

- Hashes stored on disk are just sorted (64-bit) integers and so can be compressed with integer codes, such as those we used for inverted index compression (Elias-Fano, etc.) to speed up scanning

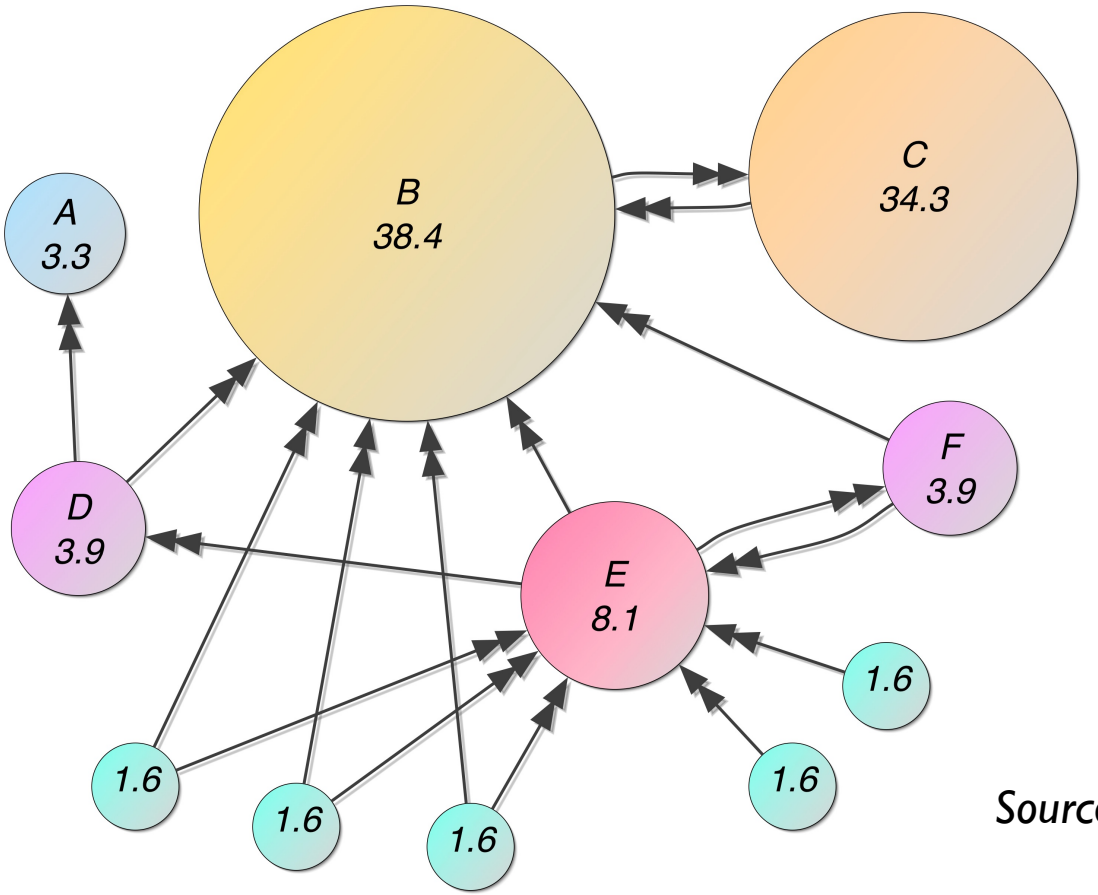
- Another implementation issue is about the order in which pages are downloaded from the Web
  - Random order
  - Breadth-first order is common (the order in which the URLs are discovered by the crawler)

- From the perspective of the search engine, some web pages are more valuable than others
  - Commercial web crawlers employ URL prioritization strategies to download important pages earlier, resulting in a higher-quality web repository and search results

- Two complementary approaches to measure page quality…

- 1st Exploit the connectivity of pages in the web graph.
  - In degree (# of pages that link to page) is simple measure

- 1st Exploit the connectivity of pages in the web graph.
  - In degree (# of pages that link to page) is simple measure
  - PageRank is a well-known spin on this idea

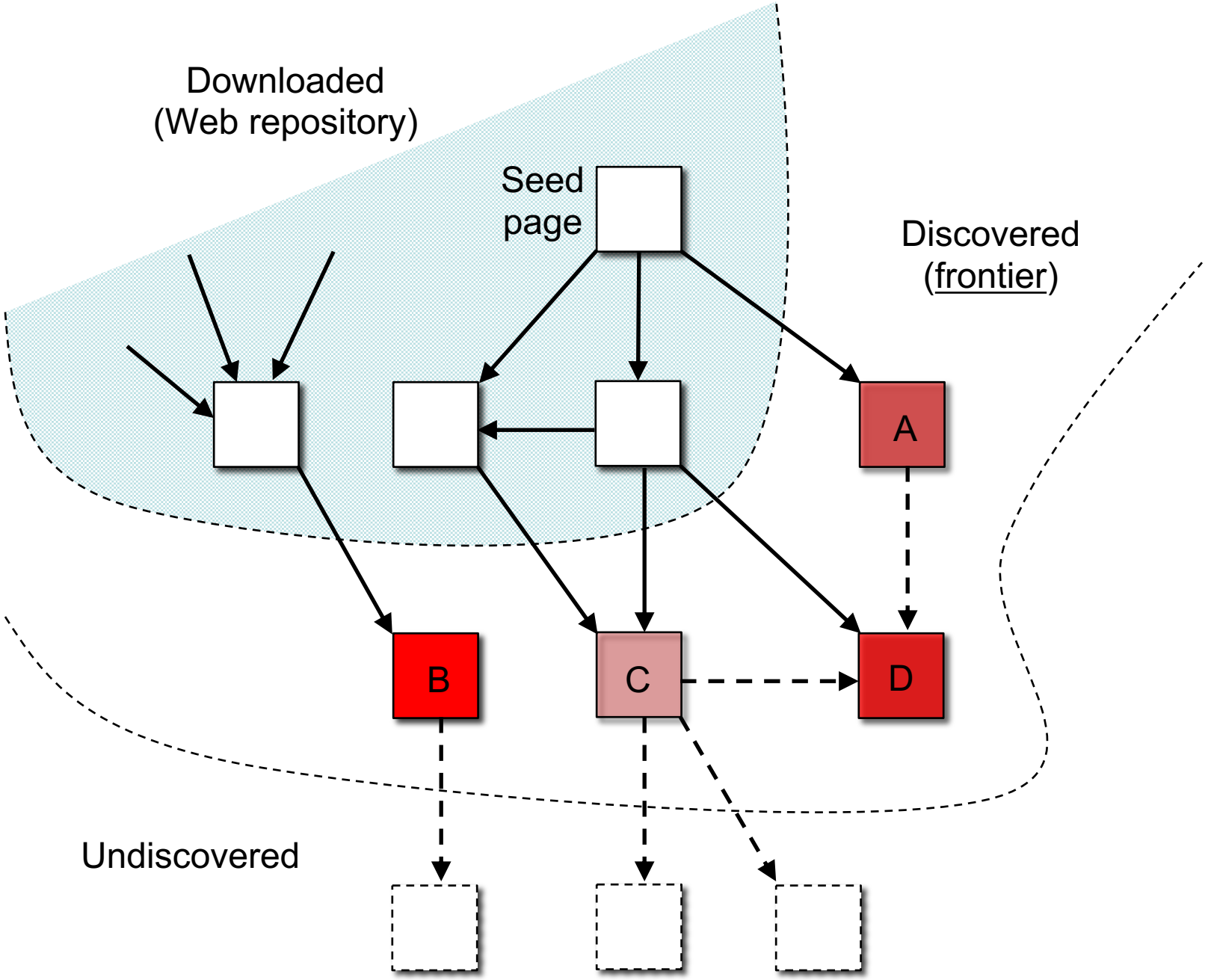


Source: Wikipedia

- 2nd URL prioritization approach is more direct…

- Measure the potential impact the page would make on the search result quality or the users' engagement with the search engine

- Web-centric (e.g. link-based) and user-centric importance measures can be combined into a single importance measure using proper weights


Downloaded
(Web repository)
Seed
page
Discovered
(frontier)
A
B
C
D
Undiscovered

Refreshing the Web Repository…

- URL-seen test constrains every web page to be downloaded at most once, implying that as the content of the pages on the Web change, some repository pages become stale

- Having out of date pages may lead to degradation of the quality of search engine results

- Crawlers cope with this problem by selectively refetching pages over time – a process known as *refreshing*
  - Refreshing decisions are non-trivial (because…)
  - Only way to determine if page's content has changed is to download it
  - Not refreshing a page leads to staleness, refreshing an unmodified page wastes resources

- In what order should we refresh pages?
  - Random? Naive.
  - Refresh important pages first?
    - Using link quality or high search impact
    - Need to be careful not to completely ignore some pages
  - Refresh based on age of page in repo?
    - Maintains average freshness
  - According to page longevity?
    - i.e. How often the page is updated
    - Maybe avoid some redundant downloads this way
    - Prioritize medium longevity: intuition is high longevity pages don't require being refreshed so often, low ones turn stale too quickly

	A	B	C	D
PageRank	0.0003	0.0007	0.0002	0.0001
Average daily click count	47	332	2	1974
Last download time	2 hours ago	1 day ago	8 days ago	6 hours ago
Estimate update frequency	daily	never	per minute	yearly

- Four pages in the repo: A, B, C, and D

- Refreshing based on:
  - Link quality would refresh B first (highest PageRank)
  - Search impact would refresh D first (most clicks)
  - Age-based would refresh C first (it has the oldest copy in the repo)
  - Longevity: A, because B and D have too high longevity, C too low

- Refreshing may be implemented in a similar way to importance-based URL prioritization

- A number of disk-based queues support scheduling of pages for refreshing
  - Each page is assigned to queue according to the estimated utility of refreshing the page
  - Once a page is removed from a queue and refreshed, it is reinserted back to the tail of the same queue
  - Queues are periodically recreated to reflect changes in estimated page utility

Distributed Web Crawling…

Faster Crawlers are Better Crawlers

- Obtaining high coverage of the Web and maintaining freshness of the web repository requires sustaining high page download speeds – the most important efficiency objective for a crawler

- If a crawler can download pages faster it can cope better with the growth and evolution of the Web…

- …and so can achieve a higher web coverage and page freshness, perhaps with some positive impact on search quality as well

Multi-threaded Crawling

- In its simplest form, the low-level fetching system can be implemented as a single thread that fetches one page at a time
- The speed at which pages are downloaded can be increased by simply running more fetcher threads and downloading pages concurrently (multi-threaded crawling)
- Typically, a single crawling node can accommodate ~100 fetcher threads, each handling an HTTP connection to a different web server
  - Using more threads results in too much context switching in the CPU and does not speed things up further

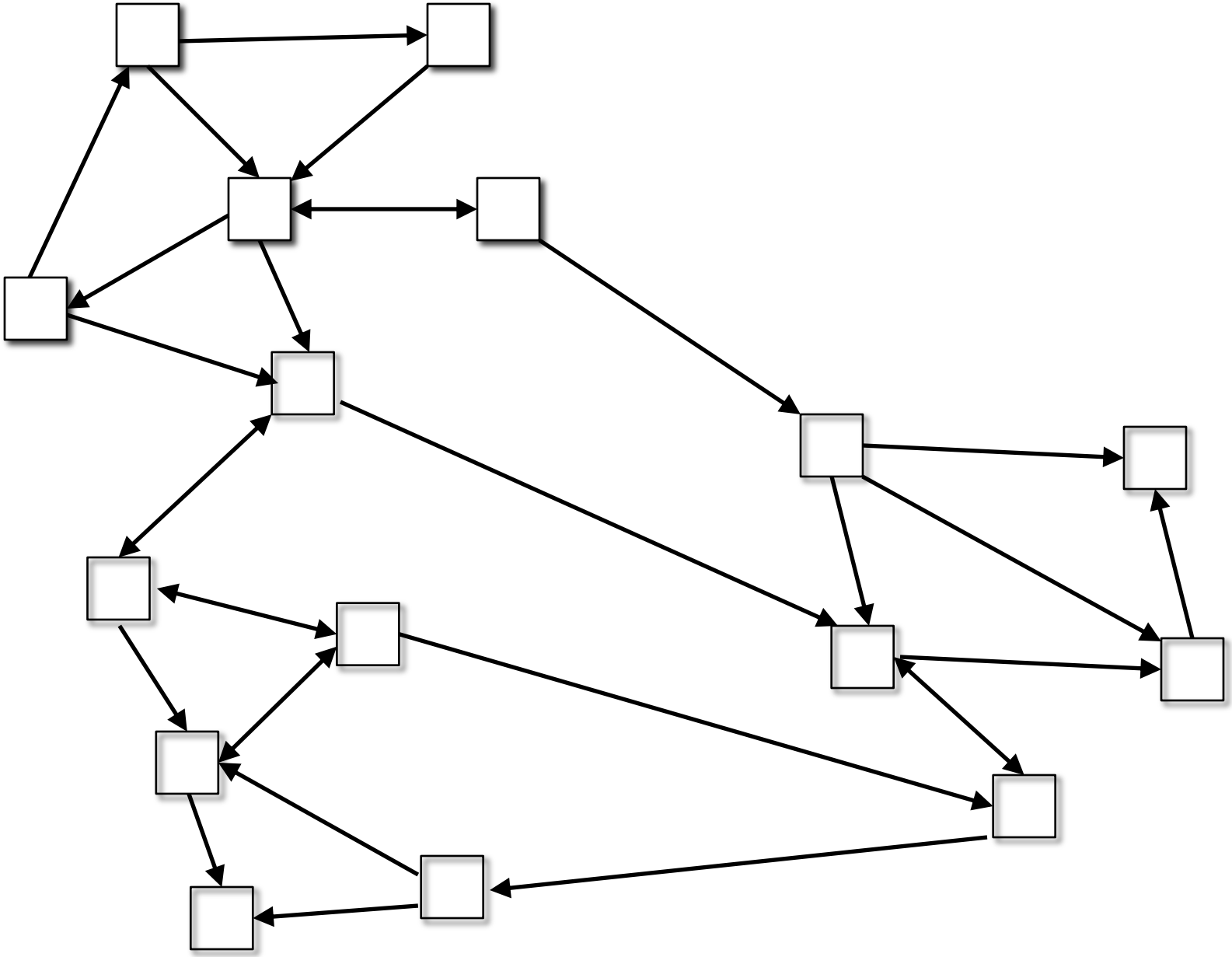
Distributed Crawling

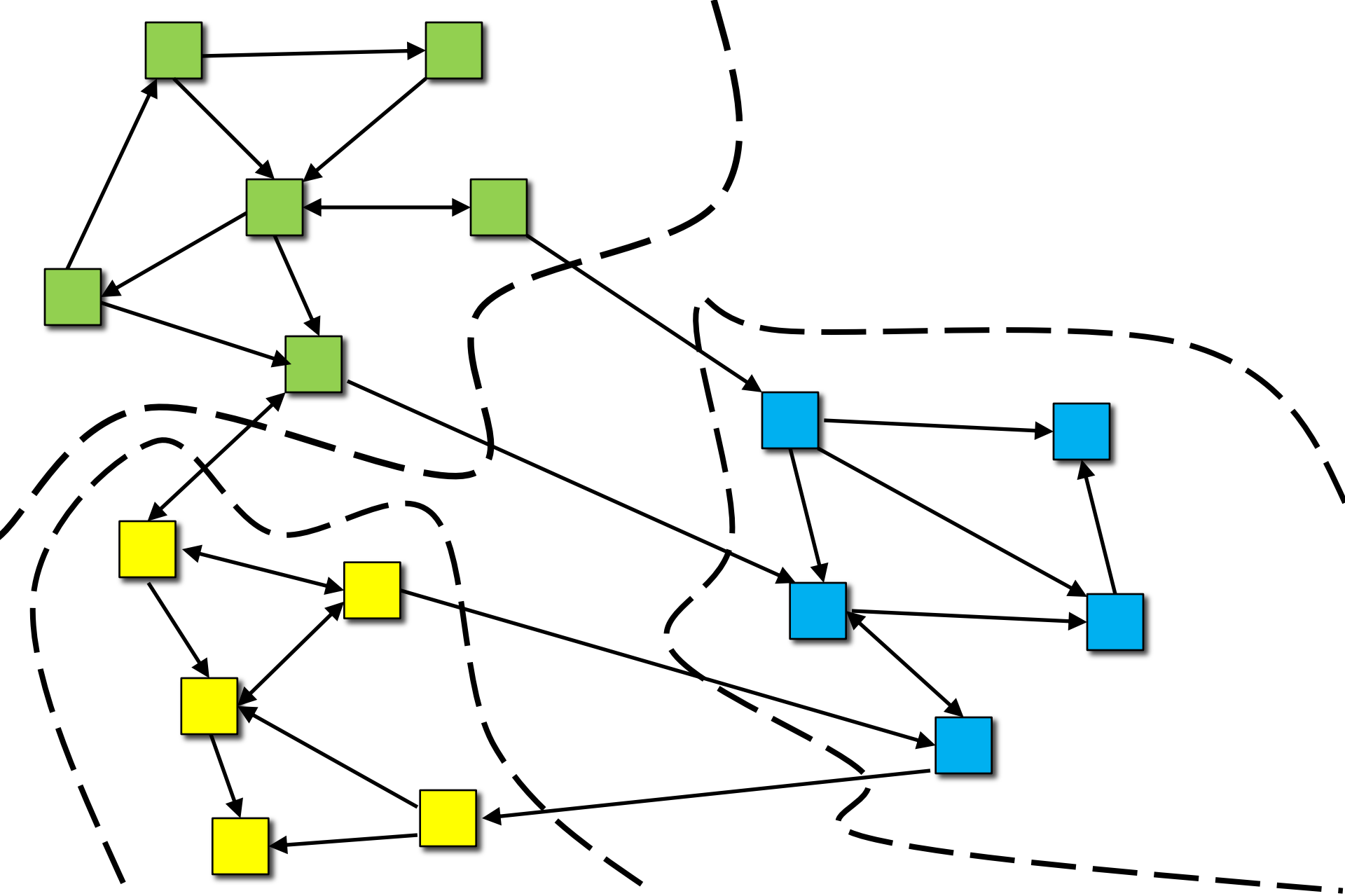
- Network bandwidth is the main bottleneck for a well-implemented crawler
- Usually a single machine and multi threading is not sufficient to saturate the network bandwidth
- Therefore, large-scale web crawlers employ clusters (a few hundred nodes) in order to utilize the network bandwidth as much as possible

Distributed Crawling: Redundancy

- Given a robust crawler that runs on a single node, it is relatively easy to build a parallel one over many nodes
  - At an extreme, crawling nodes can work completely independently
  - Easy to implement, but results in lots of redundant downloads

- One way to eliminate redundancy is to partition the space of URLs among crawling nodes (*firewall mode*)
  - Hash URLs - each node handles a range of hash values
  - Given a uniform hash function, expect balanced workload
  - Guarantees each page will be downloaded at most once

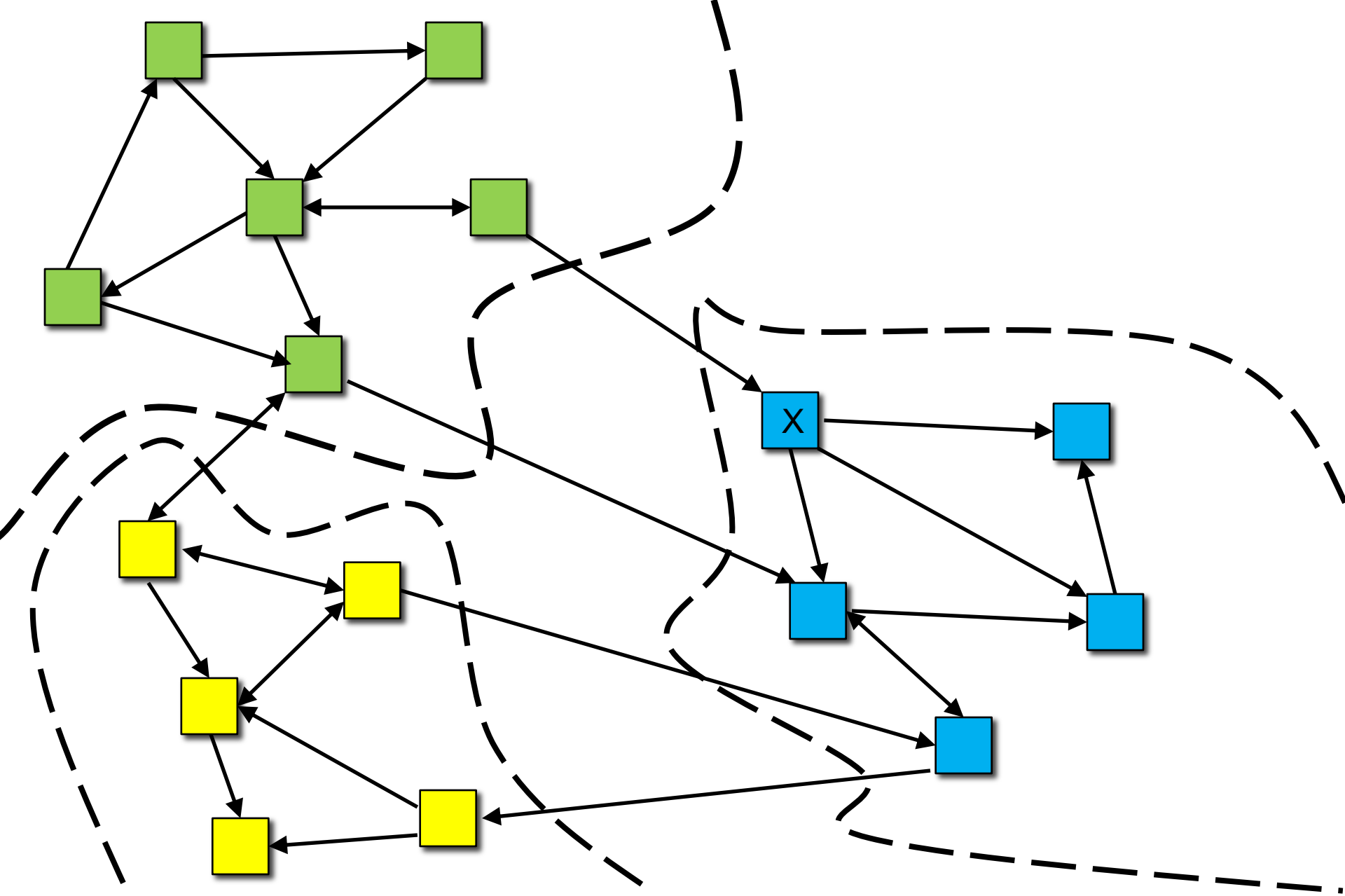




URLs partitioned over nodes in firewall mode

Distributed Crawling: Coordination

- Lack of coordination in firewall mode means that links whose source and destination pages are assigned to different nodes cannot be followed by any node
  - Some pages will never be discovered: loss of coverage

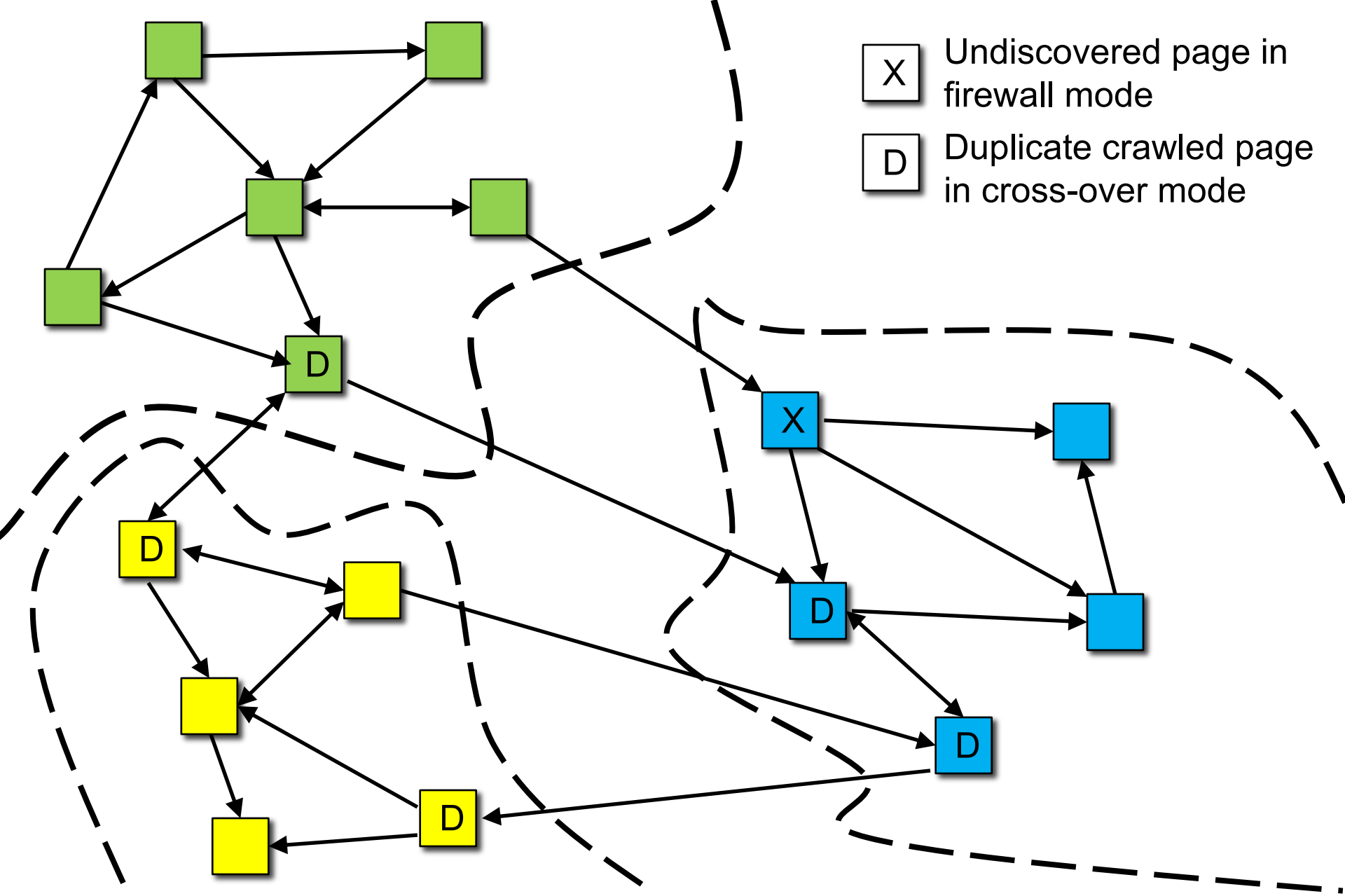


URLs partitioned over nodes in firewall mode

Distributed Crawling: Coordination

- Lack of coordination in firewall mode means that links whose source and destination pages are assigned to different nodes cannot be followed by any node
  - Some pages will never be discovered: loss of coverage

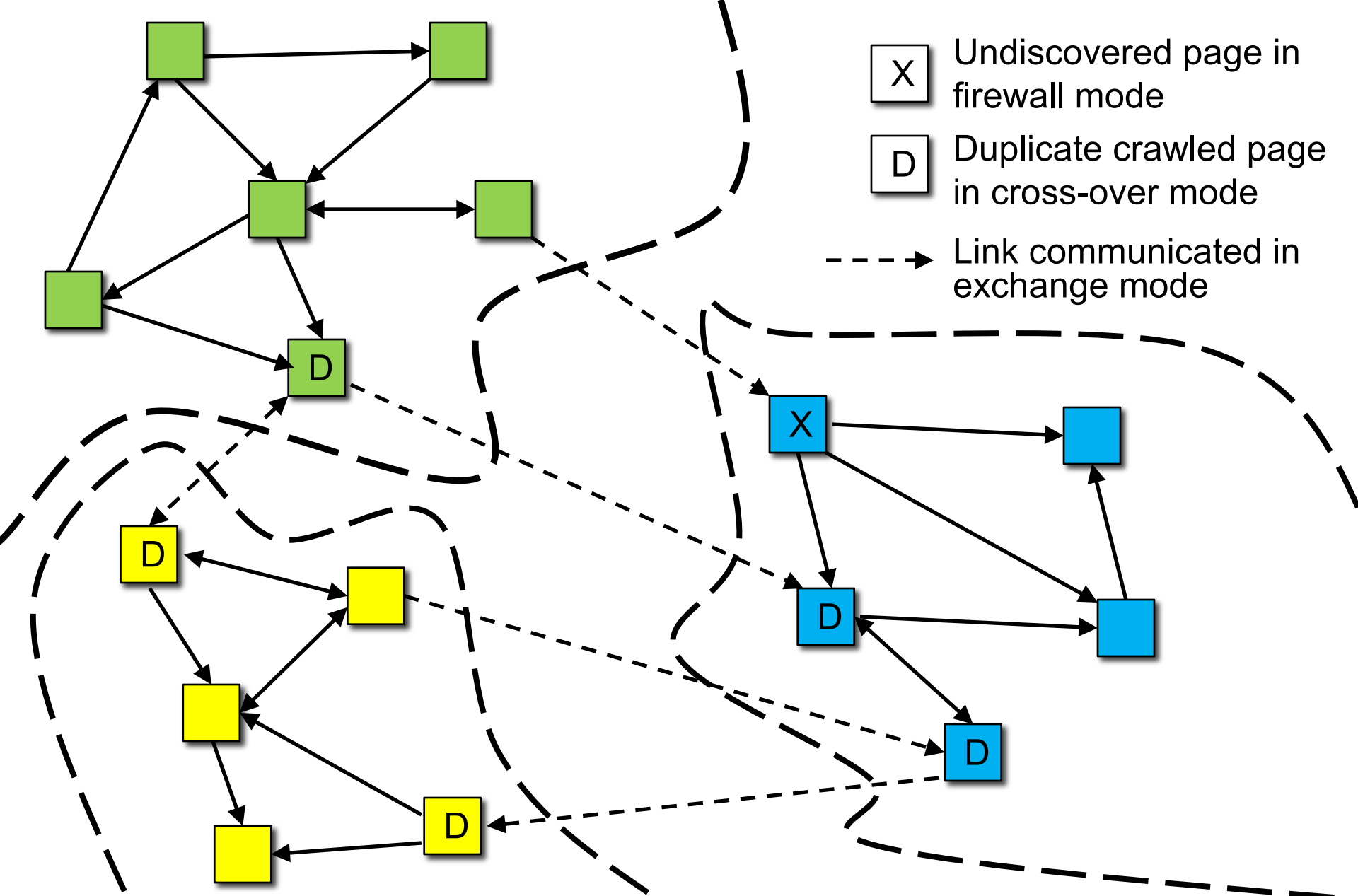
- To solve the discoverability problem: follow a link (only one hop) even when its destination URL is assigned to another node (*cross-over mode*)
  - Solve coverage issue of firewall mode
  - Reintroduces redundancy: some pages may be downloaded multiple times by different nodes



URLs partitioned over nodes in firewall mode

Distributed Crawling: Exchange Mode

- The solution, known as *exchange mode*, eliminates coverage and redundancy issues
- Discovered non-local links are communicated to the crawling nodes responsible for fetching them
- Volume of links needing to be communicated over the network can be significantly reduced by partitioning based on domain names instead of URLs
  - Links found in a website are highly likely to link to pages also inside that website
  - Another optimization: communicate links in batches



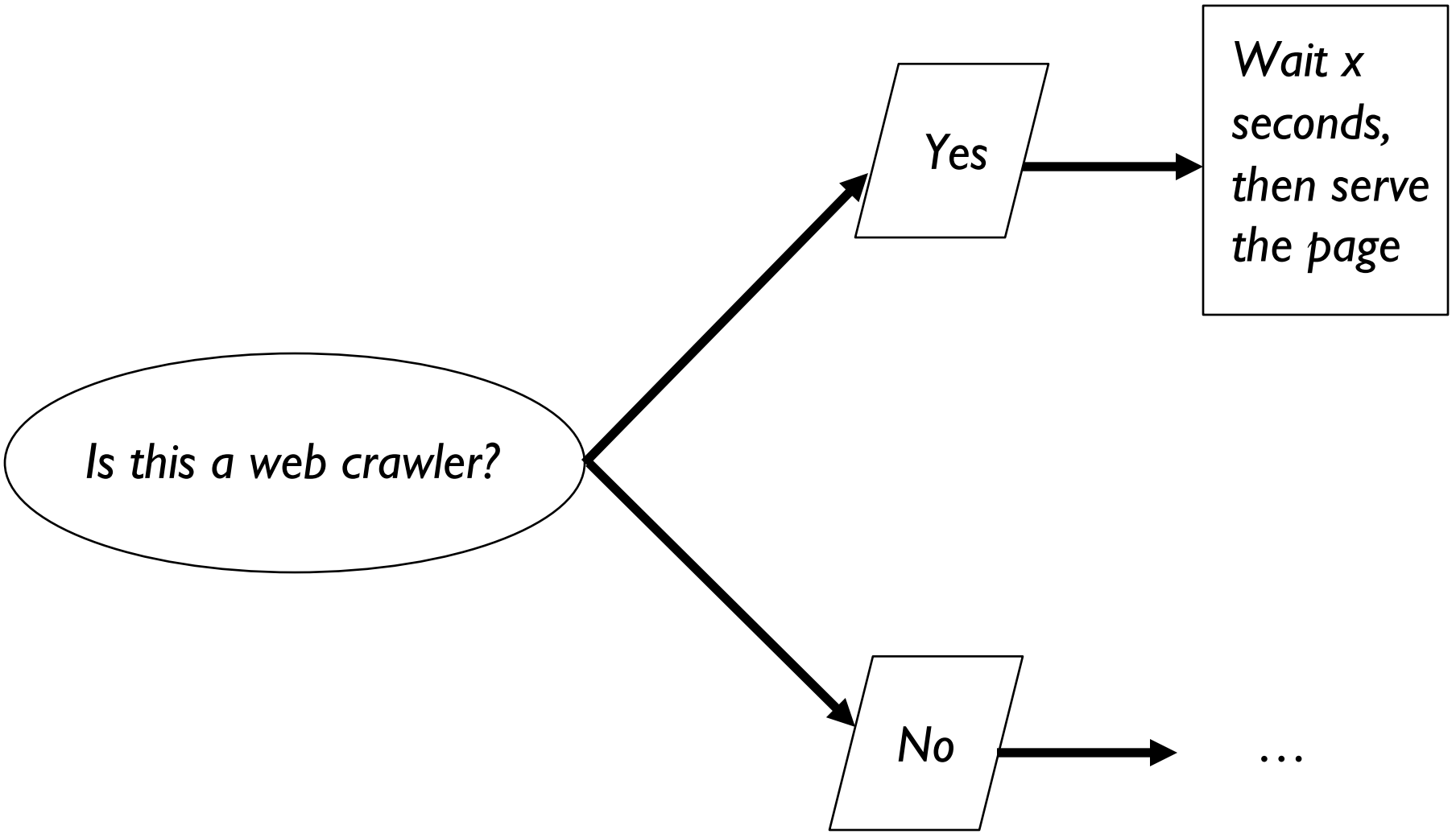
Coordination issues in distributed web crawling illustrated by a web graph partitioned over three crawling nodes.

Factors Affecting Crawling Performance…

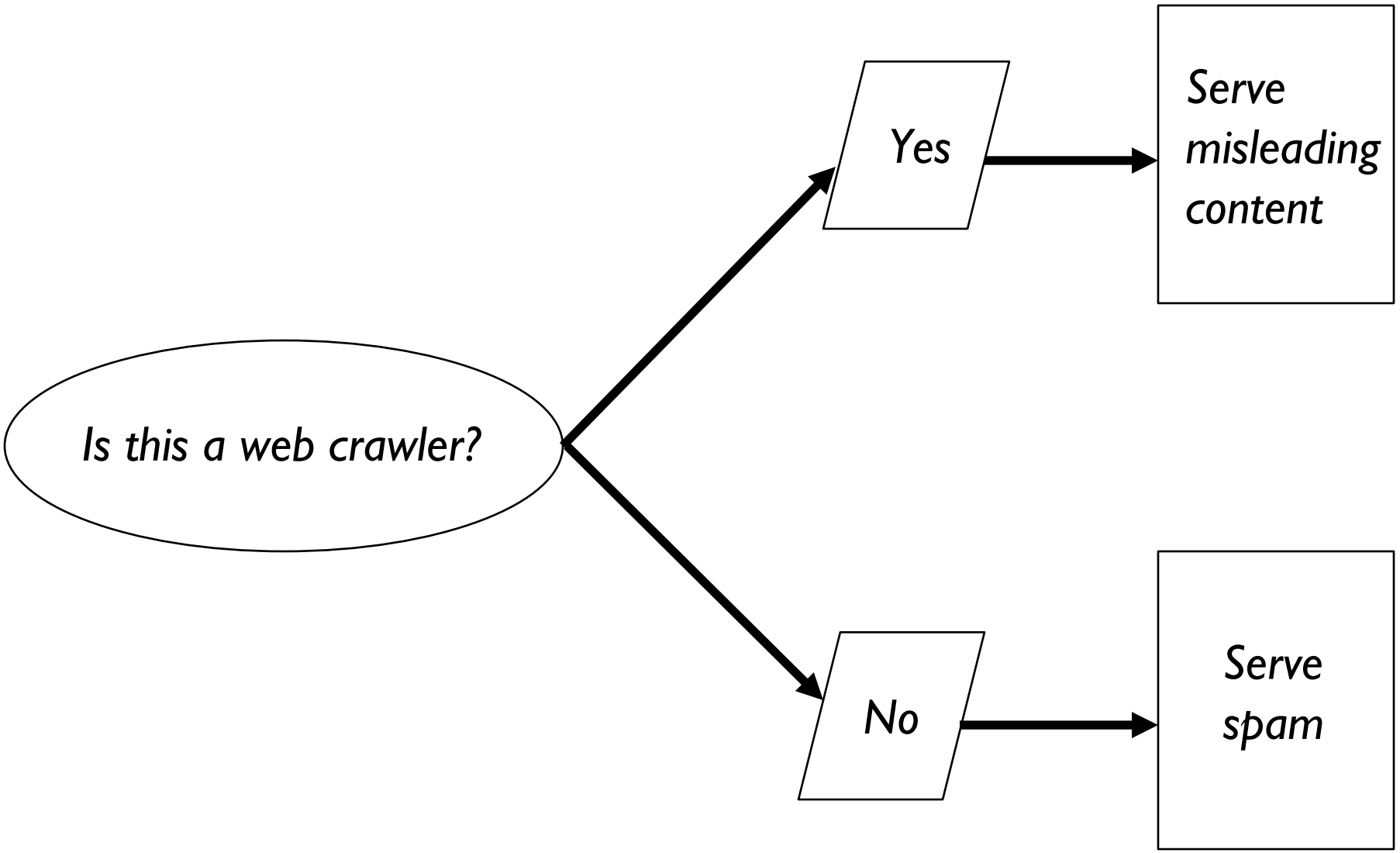
Obstacles for Web Crawlers

- The Web poses various obstacles that can negatively affect the performance of web crawlers
- Some of these obstacles stem from the malicious intent of website owners
- Well-known examples include: delay attacks, spider traps, and link farms
- Although there is no malicious intent involved, certain other situations may also affect crawling performance (such as website mirroring)

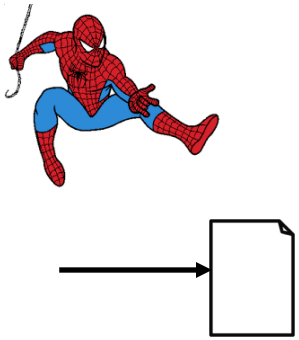
Delay Attack



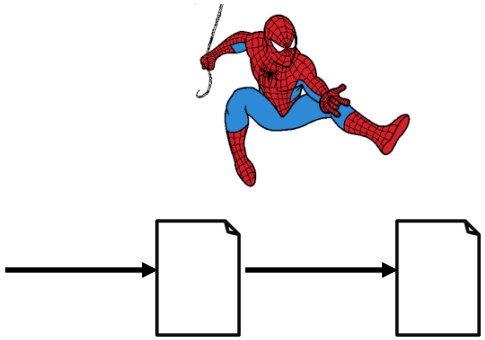
Cloaking (used by spammers)



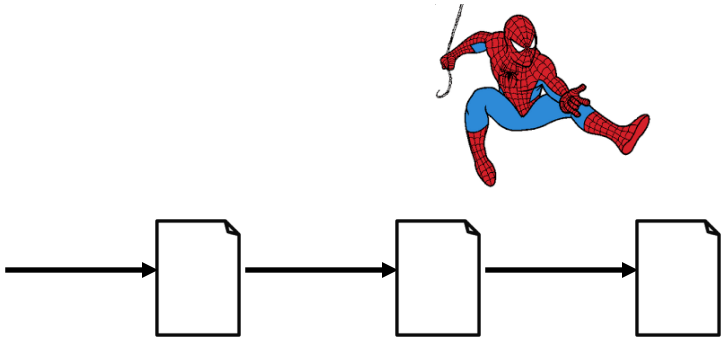
Spider traps



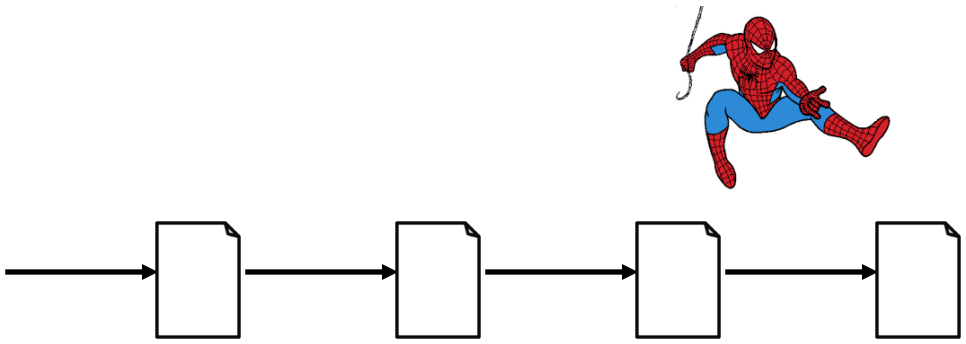
Spider traps



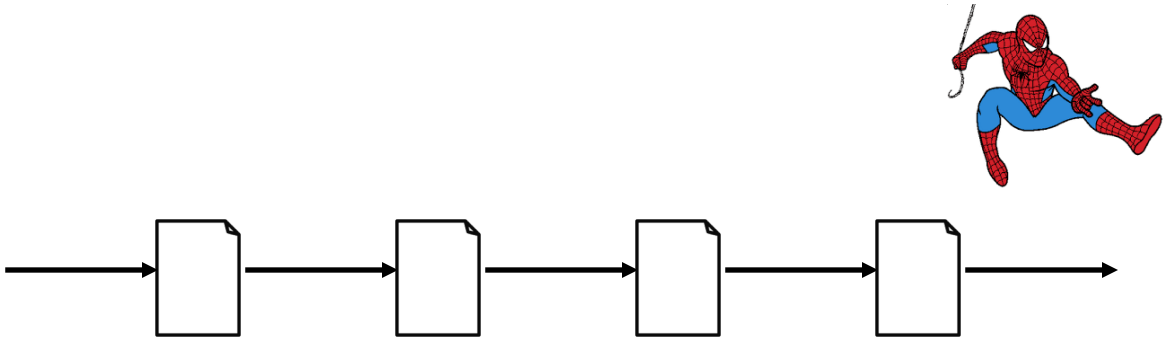
Spider traps



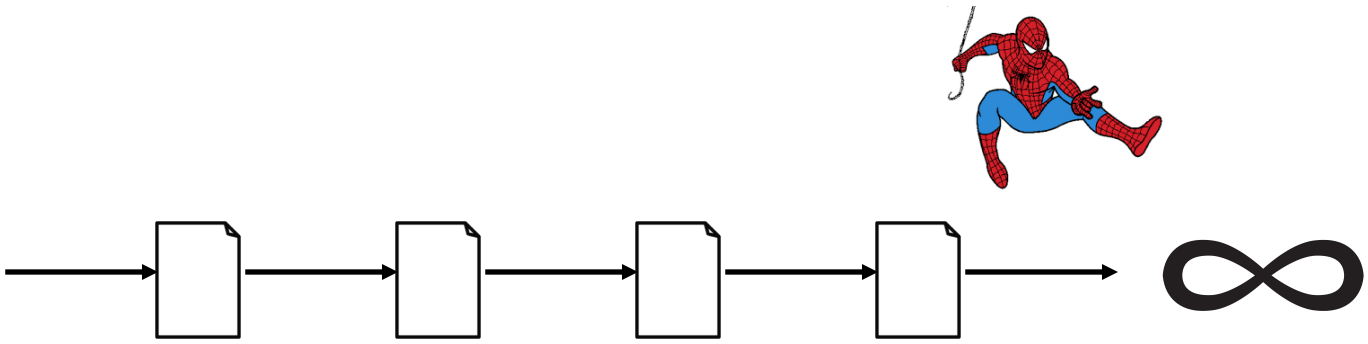
Spider traps



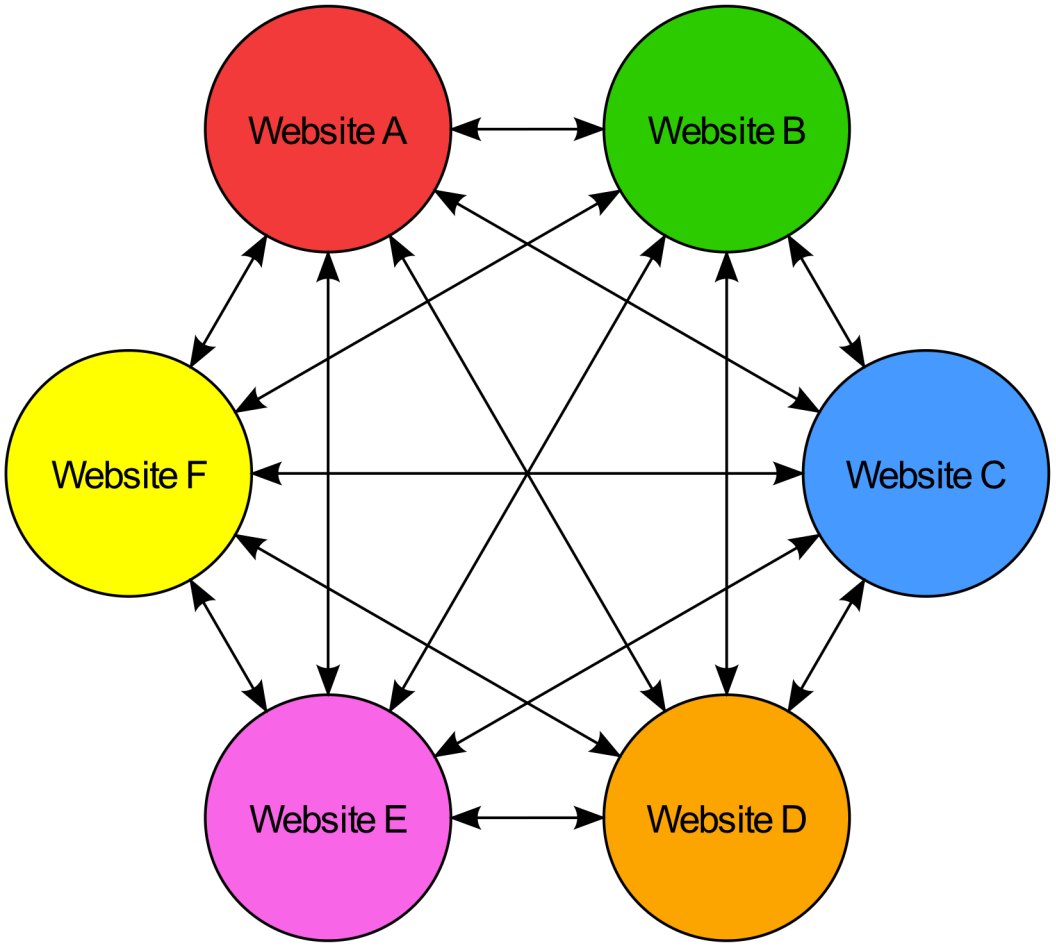
Spider traps



Spider traps



Link farms



- A large group of websites that provide artificially created links among the websites in the group
- Aims to boost certain link-based importance scores computed by search engines in order to move pages in the link farm higher in the results list
- A crawler may allocate a significant portion of its resources to download the pages in link farms

Image source: Wikipedia

Summary

- Scalable web crawlers are vital to search engines
- Building a scalable crawler is a non-trivial endeavor because the data manipulated by the crawler is too big to fit entirely in memory, so there are performance issues relating to how to balance the use of disk, memory, and network bandwidth
- There are many high-quality open-source web crawlers available these days…

A list of open source web crawlers

Crawler	Description
BUbiNG	Distributed Crawler (GNU GPLv3+)
GRUB	Distributed Crawler (GNU GPLv2)
Heritrix	Internet Archive's crawler (Apache license)
Norconex HTTP Collector	Multi-threaded crawler (Apache license)
Nutch	Distributed crawler w Hadoop support (Apache license)
PHP-Crawler	Script-based crawler (BSD license)
Srapy	Crawling framework (BSD license)
Wget	Computer program to retrieve pages (GNU GPLv3+)

Specialized Crawlers

- Besides general purpose web-crawlers used in search engines there are web crawlers designed for specific tasks…
- Hidden web crawlers: discover content that is not accessible via explicitly available link structure (dynamically generated pages, private sites, unlinked pages, scripted content)
- Focused crawlers: discover content relevant to specific topics of interest or aspects such as genre, opinion, geolocation, etc.
- There are also crawlers specialized toward content structure, such as forums.

Next lecture…

16.1: Introduction to Indexing
- Boolean Retrieval model
- Inverted Indexes

21.1: Index Compression
- unary, gamma, variable-byte coding
- (Partitioned) Elias-Fano coding (used by Google, facebook)

23.1: Index Construction
- preprocessing documents prior to search
- building the index efficiently

28.1: Web Crawling
- getting documents off the web at scale
- architecture of a large scale web search engine

30.1: Query Processing
- scoring and ranking search results
- Vector-Space model