ifis

Institut für Informationssysteme
Technische Universität Braunschweig

Information Retrieval and Web Search Engines

Lecture 11: Web Crawling

Wolf-Tilo Balke

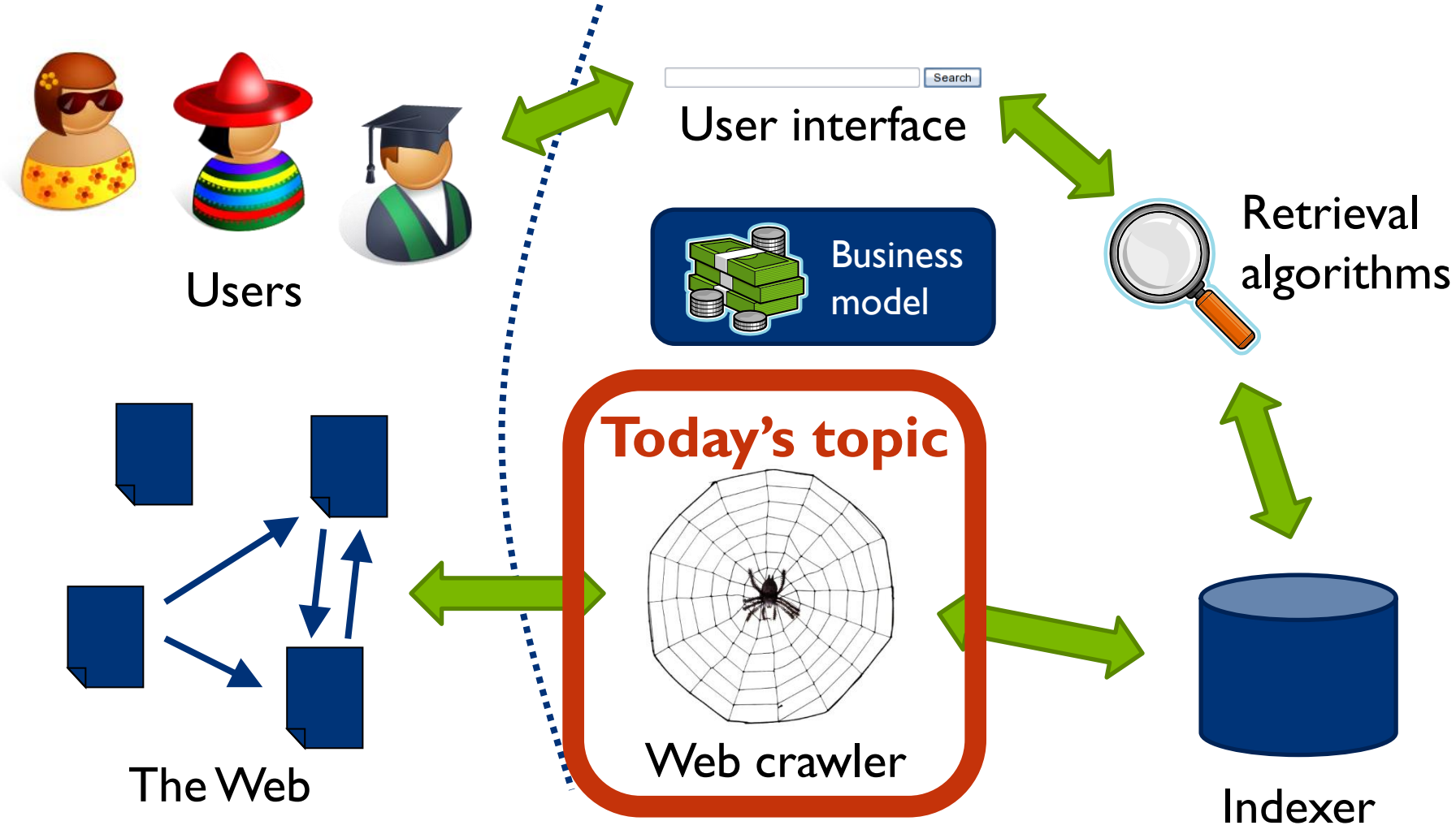
Muhammad Usman

Institut für Informationssysteme
Technische Universität Braunschweig

Overview of Web Retrieval

Summary

A typical Web search engine:

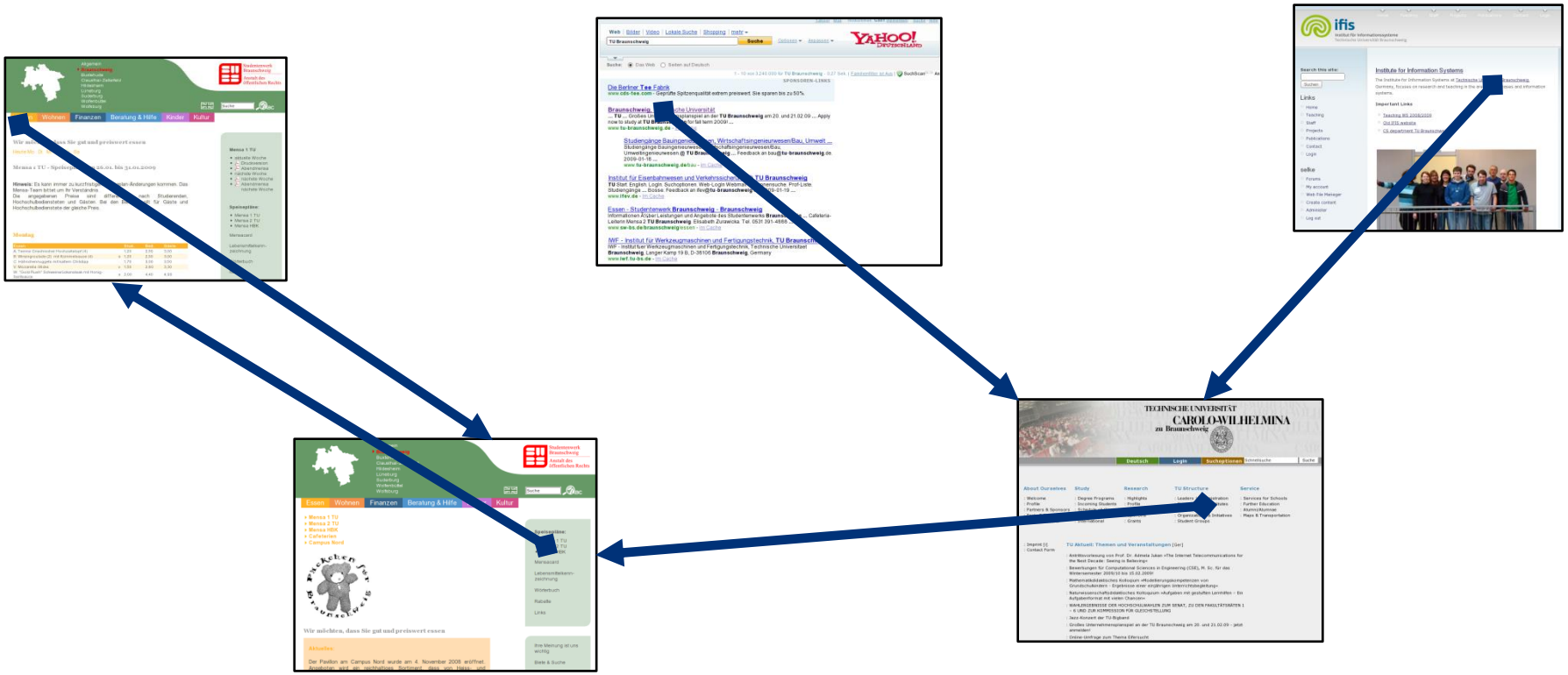


Web Crawling

1. How the Web Works
2. Web Crawling

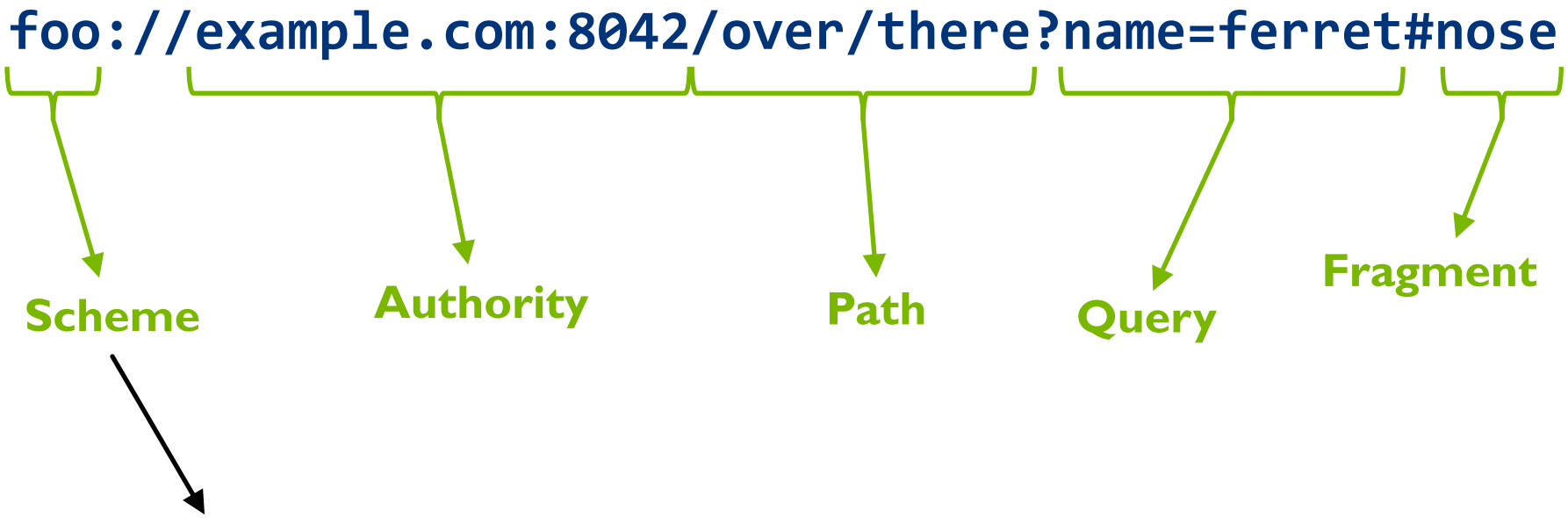
The Web

The World Wide Web
=
Resources + hyperlinks



Web Resources

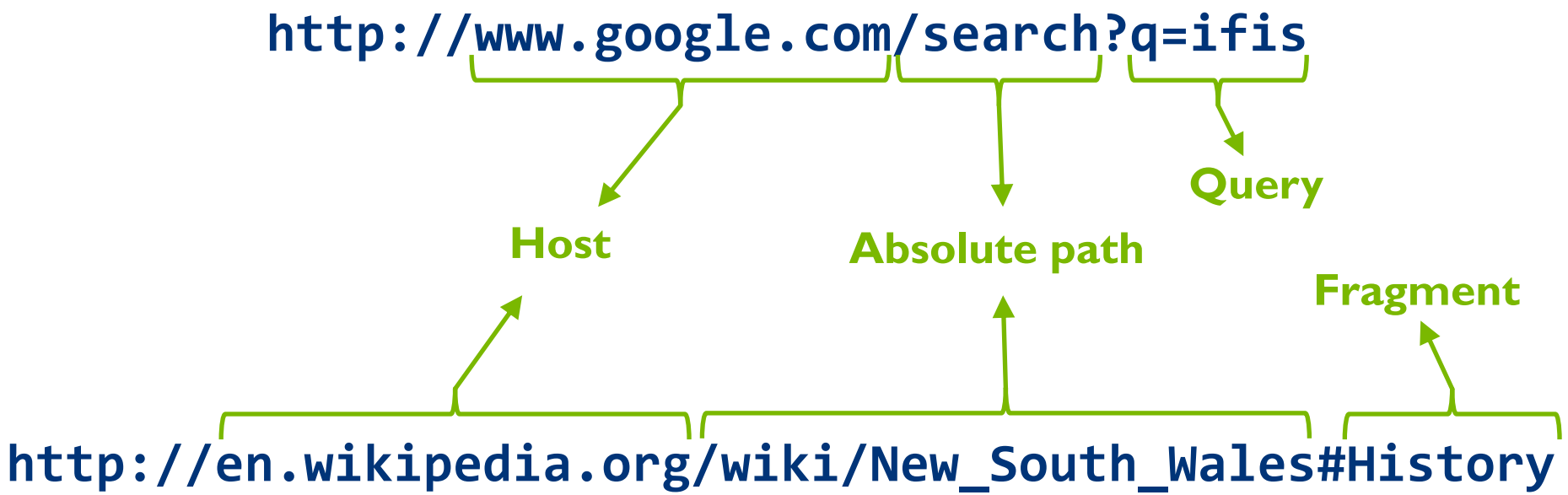
Web Resources are uniquely identified by Uniform Resource Identifiers (URIs):



Most common:
HTTP, the Hypertext Transfer Protocol

HTTP

Typical HTTP URIs look like this:



Normalized URIs

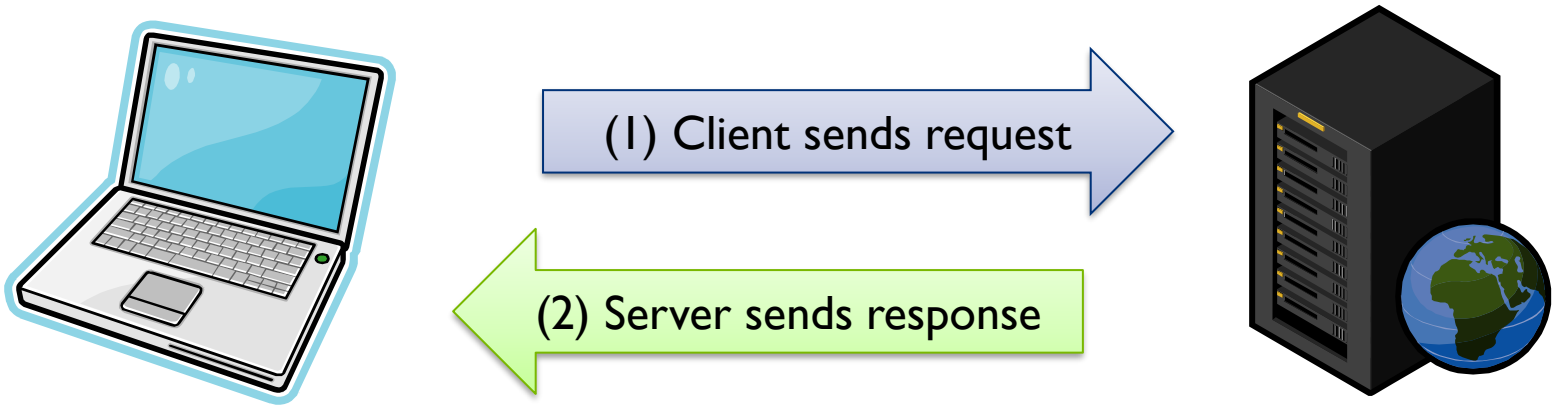
- In HTTP, every **URI** has a **normalized form**
- Normalization affects:
  - (Un)quoting of special characters (e.g. %7E represents ~)
  - Case normalization (i.e. transform the hostname to lowercase)
  - Remove the default port (HTTP's default port is 80)
  - Remove path segments "." and ".."
  - …

```
http://abc.COM:80/~smith/home.html
http://ABC.com/%7Esmith/home.html
http://ABC.com:/%7Esmith/home.html?
http://abc.com:/~smith/../~smith/home.html?
http://ABC.com/./../~smith/home.html
```

```
http://abc.com/~smith/home.html
```

How Does HTTP Work?

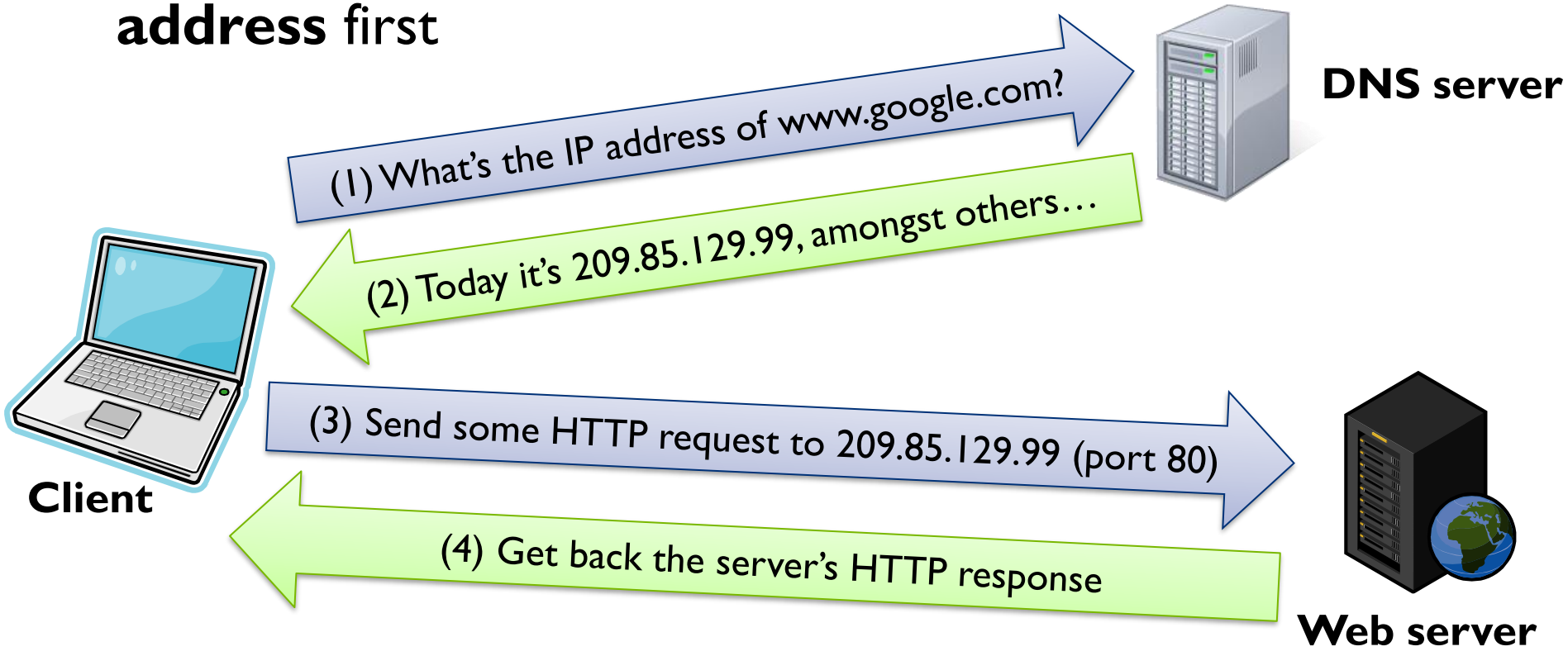
- HTTP is a **request/response** standard between a **client** and a **server**



- HTTP works on top of **TCP/IP**
  - Servers are identified by **IP addresses** (e.g. 134.169.32.171)
  - Hostnames are mapped to IP adresses using the **Domain Name System (DNS)**
  - There is a **many-to-many relationship** between IP addresses and hostnames

How Does HTTP Work?

- TCP/IP is based on IP addresses
- Therefore: When some client want to contact the host `www.google.com`, it has to **look up the host's IP address** first



How Does HTTP Work?

- **How do HTTP requests look like?**
- **Example:** `http://www.google.com/search?q=ifis`

HTTP request:

```
GET /search?q=ifis HTTP/1.1[CRLF]
Host: www.google.com[CRLF]
Connection: close[CRLF]
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)[CRLF]
Accept-Encoding: gzip[CRLF]
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7[CRLF]
Cache-Control: no[CRLF]
Accept-Language: de,en;q=0.7,en-us;q=0.3[CRLF]
[CRLF]
```

Name of resource

Carriage return followed by line feed

"GET" request method

Hostname (since there could be different hosts having the same IP address)

How Does HTTP Work?

- Matching **HTTP response** of www.google.com:

Header

```
HTTP/1.1 200 OK[CRLF]
Cache-Control: private, max-age=0[CRLF]
Date: Tue, 27 Jan 2009 10:03:57 GMT[CRLF]
Expires: -1[CRLF]
Content-Type: text/html; charset=UTF-8[CRLF]
Server: gws[CRLF]
Transfer-Encoding: chunked[CRLF]
[CRLF]
```

Body

```
<!doctype html><head><meta http-equiv=content-type content="text/html;
charset=UTF-8"><title>ifis - Google
Search</title><script></script><style>
[…]
```

Status code (200 means "resource found")

Some information related to caching

MIME type of this resource

The resource itself

How Does HTTP Work?

- Important types of **HTTP requests** are:
  - **GET:**
    Requests a representation of the specified resource
  - **HEAD:**
    Asks for the response identical to the one that would correspond to a GET request, but without the response body (useful to determine whether the resource has changed)
  - **POST:**
    Submits data to be processed (e.g., from an HTML form) to the identified resource, which may result in the creation of a new resource or the updates of existing resources or both

How Does HTTP Work?

- Important types of **HTTP status codes** are:
  - **200 (OK):** Standard response for successful HTTP requests
  - **301 (Moved Permanently):** This and all future requests should be directed to a given URI
  - **302 (Found / Moved Temporarily):** Only this request should be directed to a given URI
  - **304 (Not Modified):** The resource has not been modified since last requested
  - **404 (Not Found):** The requested resource could not be found (but may be available again in the future)
  - **410 (Gone):** The resource requested is no longer available (will not be available again)

HTML

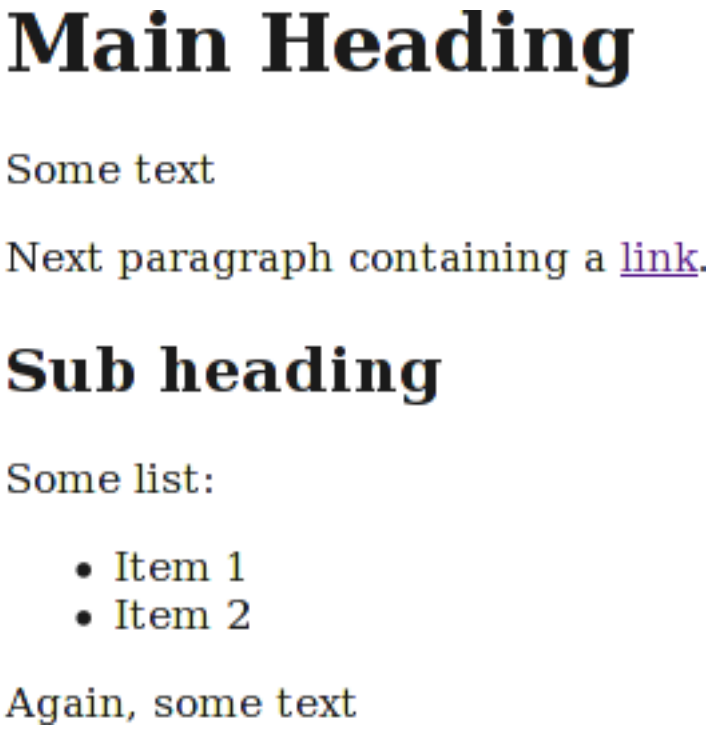
- What we have learned:
  - How Web resources are identified (URIs)
  - How Web resources can be retrieved (HTTP)
- What's still missing: **How do resources look like?**
- Most web resources are of **MIME type text/html,** i.e. they are text documents written using HTML
- HTML stands for **Hypertext Markup Language**
- HTML was invented by Tim Berners-Lee in 1991

HTML

- HTML is a **markup language,** i.e., it provides means to **describe the structure** of text-based information in a document
- In HTML you can denote certain text as…
  - **Headings:**
    ```
    <h1>Main heading</h1>  <h2>Sub Heading</h2>
    ```
  - **Paragraphs:**
    ```
    <p>Some text...</p>
    ```
  - **Lists:**
    ```
    <ul><li>First item</li><li>Second item</li></ul>
    ```
  - **Links:**
    ```
    <a href="http://www.google.com">Link to Google</a>
    ```
  - …

HTML

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<body>
<h1>Main Heading</h1>
<p>Some text</p>
<p>
Next paragraph containing a
<a href="http://www.yahoo.com">link</a>.
</p>
<h2>Sub heading</h2>
<p>Some list:</p>
<ul><li>Item 1</li><li>Item 2</li></ul>
<p>Again, some text</p>
</body>
</html>
```



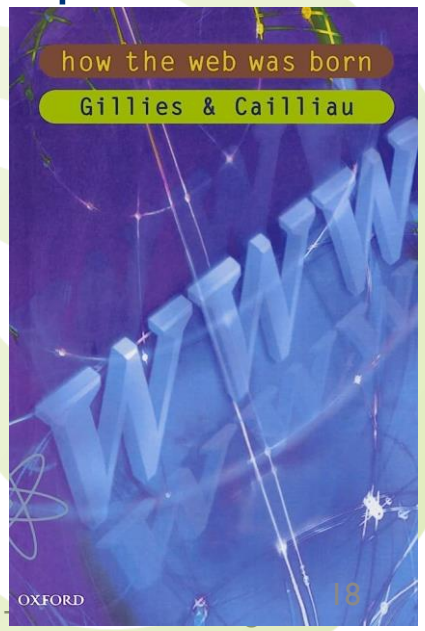
HTML

- Currently, HTML is available in many different versions:
  - 1995: HTML 2.0 (based on SGML)
  - 1997: HTML 3.2
  - 1997: HTML 4.0
  - 1999: HTML 4.01
  - 2000: "ISO HTML"
  - 2000: XHTML 1.0 (based on XML)
  - 2001: XHTML 1.1
  - HTML 5

The Beginnings of the Web

Detour

- Before 1989
  - Hypertext and the Internet are separate, unconnected ideas
- 1989
  - **Tim Berners-Lee** is working at **CERN,** Geneva
  - Researchers from around the world needed to **share data,**
  - "a large **hypertext database** with typed links" proposal
  - Implementing on a **NeXT workstation**



The Beginnings of the Web

Detour

Berners-Lee's NeXTcube:

- 25 MHz CPU, 8 MB–64 MB RAM

The Beginnings of the Web

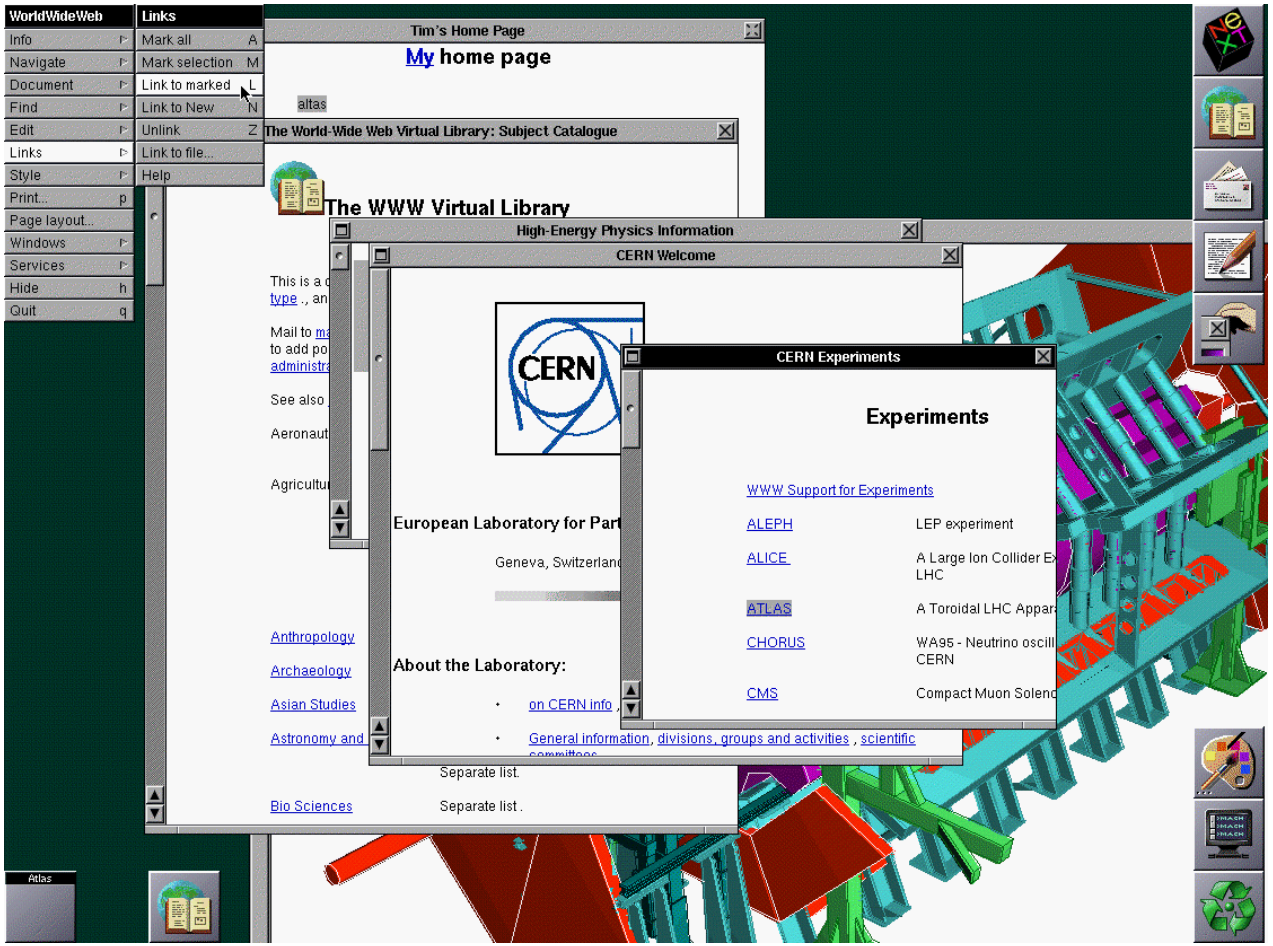
Detour

- 1990
  - CERN computer scientist **Robert Cailliau** joins Berners-Lee's vision and rewrites the proposal
  - Both present their idea at the European Conference on Hypertext Technology but find no vendors who support them
  - The name **World Wide Web** is born
  - By Christmas 1990, all **tools** for a working Web have been created by Berners-Lee:
    - HTML
    - HTTP
    - A Web server software: CERN httpd
    - A Website: http://info.cern.ch
    - A Web browser/editor: WorldWideWeb (runs only on NeXT)

The Beginnings of the Web

Detour

The first Web browser:



The Beginnings of the Web

Detour

- 1991
  - Nicola Pellow creates a **simple text browser** that could run on almost any computer
  - To encourage use within CERN, they put the **CERN telephone directory** on the Web, which previously was located on a mainframe
  - Berners-Lee announces the Web in the alt.hypertext newsgroup:
    "The WorldWideWeb (WWW) project aims to allow all links to be made to any information anywhere. [...] The WWW project was started to allow high energy physicists to share data, news, and documentation. We are very interested in spreading the web to other areas, and having gateway servers for other data. Collaborators welcome!"

The Beginnings of the Web

Detour

- 1993
  - The Web spreads around the world
  - The graphical Web browser **Mosaic** is developed by a team at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign; the team is led by the later founder of Netscape, **Marc Andreessen**
- 1994
  - Netscape is founded
  - Mosaic becomes the **Netscape Navigator**
  - The **World Wide Web Consortium (W3C)** is founded by Berners-Lee at the Massachusetts Institute of Technology with support from the Defense Advanced Research Projects Agency (DARPA) and the European Commission

Web Crawling

1. How the Web Works
2. **Web Crawling**

A Basic Crawler

- **A basic crawler (aka robot, bot, spider) consists of:**
  - A **queue** of URIs to be visited
  - A method to **retrieve** Web resources and process HTTP data
  - A **page parser** to extract links from retrieved resources
  - A **connection** to the search engine's **indexer**
- **The basic mode of operation:**
  1. Initialize the queue with URIs of known **seed pages**
  2. Take URI from queue
  3. Retrieve and parse page
  4. Extract URIs from page
  5. Add new URIs to queue
  6. GOTO (2)

Problem Size

- The Web is large: **60 billion pages** (more or less…)
- Let's assume we want to **crawl each page once a year**

- How many pages do we have to crawl **per second** then?
  - 60,000,000,000 pages per year
  - 5,000,000,000 pages per month
  - 166,666,667 pages per day
  - 6,944,444 pages per hour
  - 115,740 pages per minute
  - **1929 pages per second**
- Well, it seems like we need a **highly scalable** crawler…

Further Complications

- **Apart from scalability, there are further issues**

- How to detect **spam** pages?
- How to detect **duplicates** or pages already seen?
- How to avoid **spider traps?**
- We need many machines, how do we **distribute?**
- How to handle **latency** problems?
- How to limit the used **bandwidth?**
- How **deep** should we crawl sites?
- How to comply with the **site owner's** wishes?

MUST-Have Features

- **Robustness**
  - **Golden rule:**
    For every crawling problem you can (or cannot) think of, there will be a Web page exhibiting this problem
  - Web pages, URLs, HTTP responses, and network traffic as such can be **malformed** and might **crash your software**
  - Therefore, use **very robust software**
  - "Very robust" usually means non-standard
  - Robustness also refers to the ability to **avoid spider traps**

MUST-Have Features

- **Politeness**
  - Web site owner's usually have to pay for their Web traffic
  - Do not generate unnecessarily high traffic!
  - Do not slow down other people's servers by "hammering," i.e., keep the number of requests per time unit low!
  - Obey explicit crawling policies set by site owners (e.g. robots.txt)!

Robot Exclusion Standard

- **The robot exclusion standard**
  - **Exclude some resources** from access by robots, and thus from indexing by search engines
  - Put a file named **robots.txt** in your domain's top-level directory (e.g. http://en.wikipedia.org/robots.txt), which specifies what resources crawlers are allowed to access
  - **Caution:** This "standard" is not a standard in the usual sense, it's purely advisory!
- **Examples:**
  - **Allow all robots to view all files:**

```
User-agent: *
Disallow:
```

Key components

Detour

- **User-agent:** Specifies the name of the robot or group of robots the rule applies to.
- **Disallow:** Indicates the URLs that should not be crawled or indexed by the specified user-agent.
- **Allow:** Overrides previous disallow rules to allow specific URLs for the specified user-agent.
- **Request-rate:** Sets the maximum number of requests per second a robot can make to the website.
- **Crawl-delay:** Specifies the minimum delay in seconds between successive requests from the same robot.
- **Sitemap:** Informs search engines about the location of the XML sitemap for the website.

Robot Exclusion Standard

Detour

- **Examples:**
  - **Keep all robots out:**
    ```
    User-agent: *
    Disallow: /
    ```
  - **Exclude certain resources:**
    ```
    User-agent: *
    Disallow: /cgi-bin/
    Disallow: /private/
    ```
  - **Exclude a specific bot:**
    ```
    User-agent: BadBot
    Disallow: /private/
    ```
  - **Limit the number of requests per second:**
    ```
    Request-rate: 1/5
    ```
  - **Recommend a visit time interval (in GMT):**
    ```
    Visit-time: 0600-0845
    ```

Robot Exclusion Standard

A look at http://www.wikipedia.org/robots.txt:

```
#
# robots.txt for http://www.wikipedia.org/ and friends
#
# Please note: There are a lot of pages on this site, and there are
# some misbehaved spiders out there that go _way_ too fast. If you're
# irresponsible, your access to the site may be blocked.
#

# advertising-related bots:
User-agent: Mediapartners-Google*
Disallow: /

# Wikipedia work bots:
User-agent: IsraBot
Disallow:
```

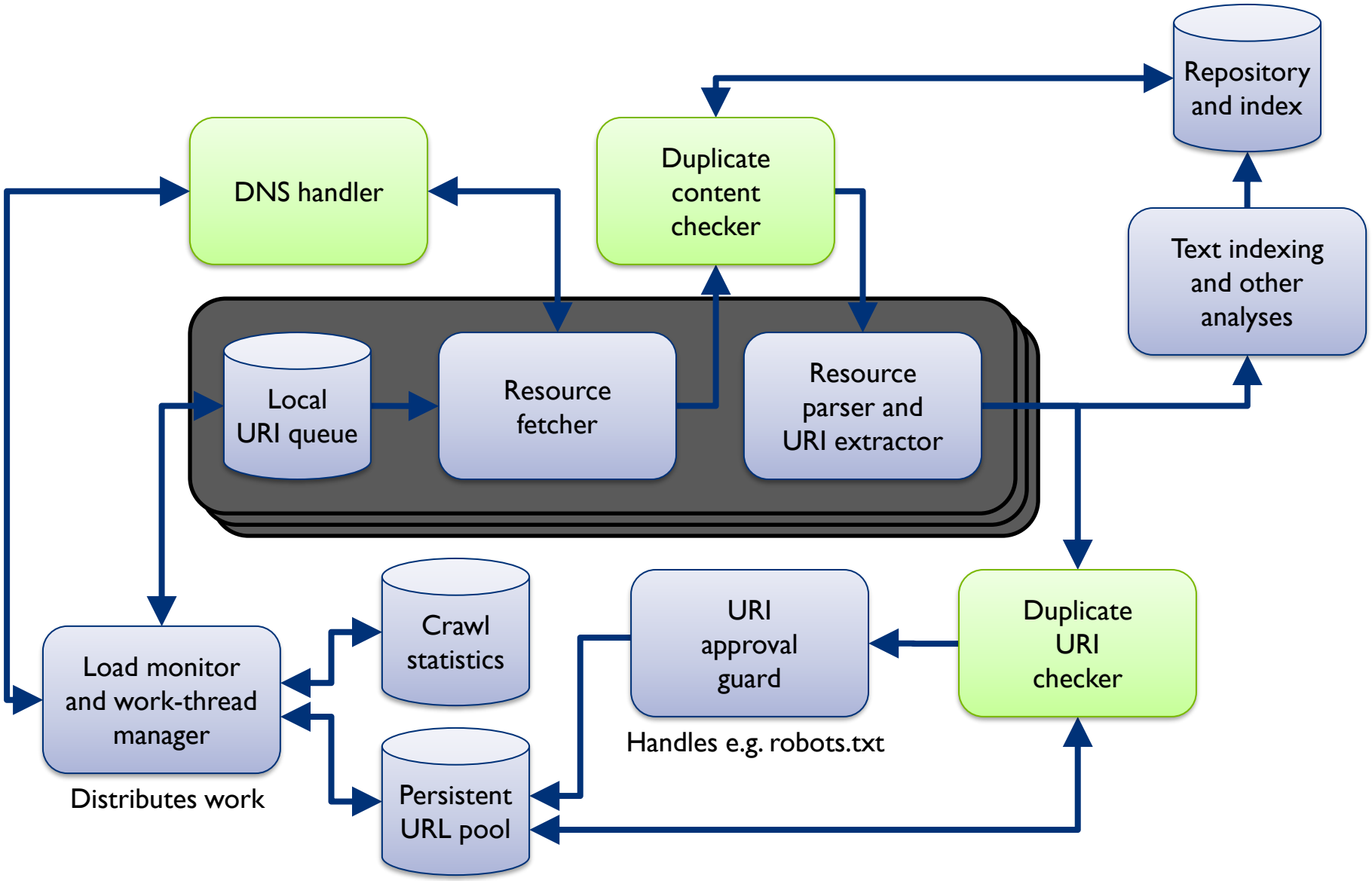
SHOULD-Have Features

- **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 and updating them more often than "useless" ones

SHOULD-Have Features

- **Freshness:**
  - The crawler should operate in **continuous mode,** i.e. it should obtain fresh copies of previously fetched pages
  - Crawl a page with a frequency that approximates the **rate of change** of that page
  - Be able to **update** a given set of pages **on demand,** e.g. if there is some current highly popular topic ("World Cup")
- **Extensible:**
  - Be able to cope with **new data formats, new protocols, …**
  - This amounts to having a **modular architecture**

Anatomy of a Large-Scale Crawler



The DNS Handler

- Fetching DNS information usually is slow due to **network latency** and the need to query many servers in parallel
- The DNS handler is a customized local DNS component
  - **Prefetches DNS information** that will be needed by some work-thread in the near future
  - Uses a **relaxed policy regarding DNS updates,** i.e., break the DNS standard to avoid unnecessary DNS queries

The Duplicate URI Checker

- **Task:**
  - Find out whether a given URI is contained in the URI pool
  - But: As quick as possible!

- **Problems:**
  - Doing string comparisons with all pool URIs is too expensive
  - Even using index structures does not help much here since string operations as such are very expensive

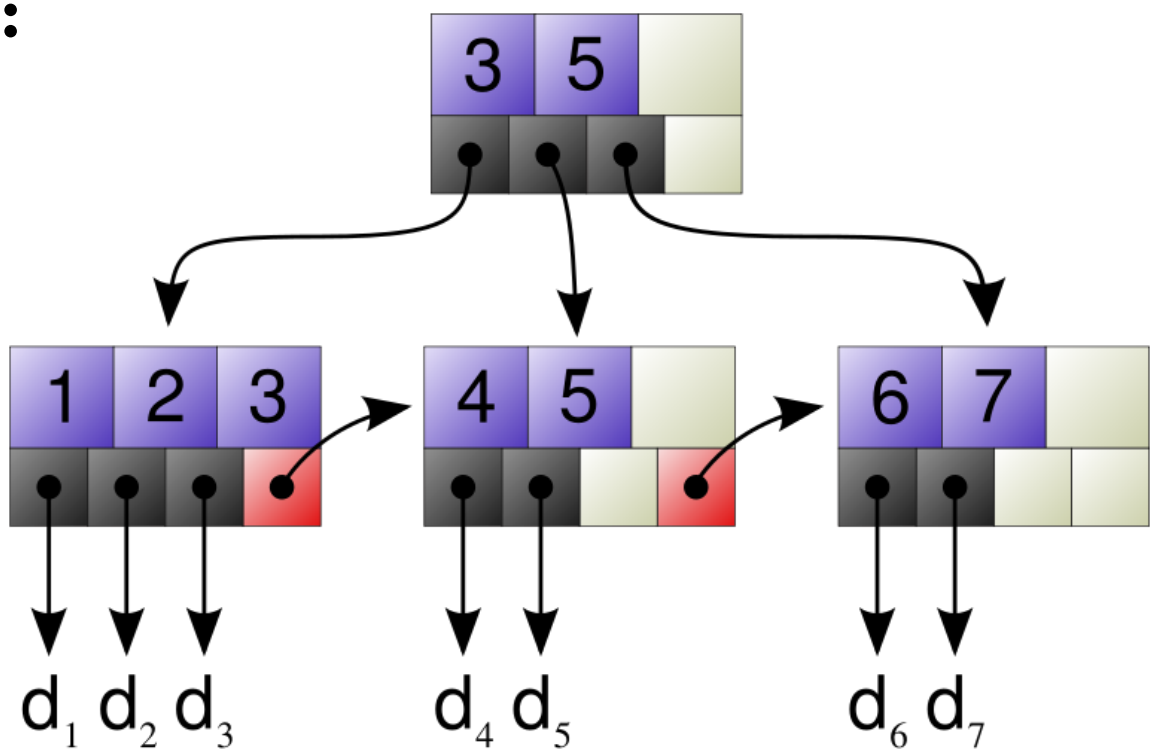
- **Solution:**
  - Use **fingerprints!**

The Duplicate URI Checker

- **Fingerprinting**
  - First, only use URIs in their normalized forms
    - This reduces the number of different URIs that must be handled
  - Then, for any normalized URI, compute its **hash value** (aka **fingerprint**) with respect to some **hash function**
  - A popular hash function is **MD5,** which can be computed quickly and yields a **128-bit fingerprint**
  - **Example of MD5:** http://www.ifis.cs.tu-bs.de becomes 75924e8d184c52dd9bc5b368361093a8 (hexadecimal)
- Now, build a **B-tree** (or hash table) of all fingerprints containing pointers to the original URIs

The Duplicate URI Checker

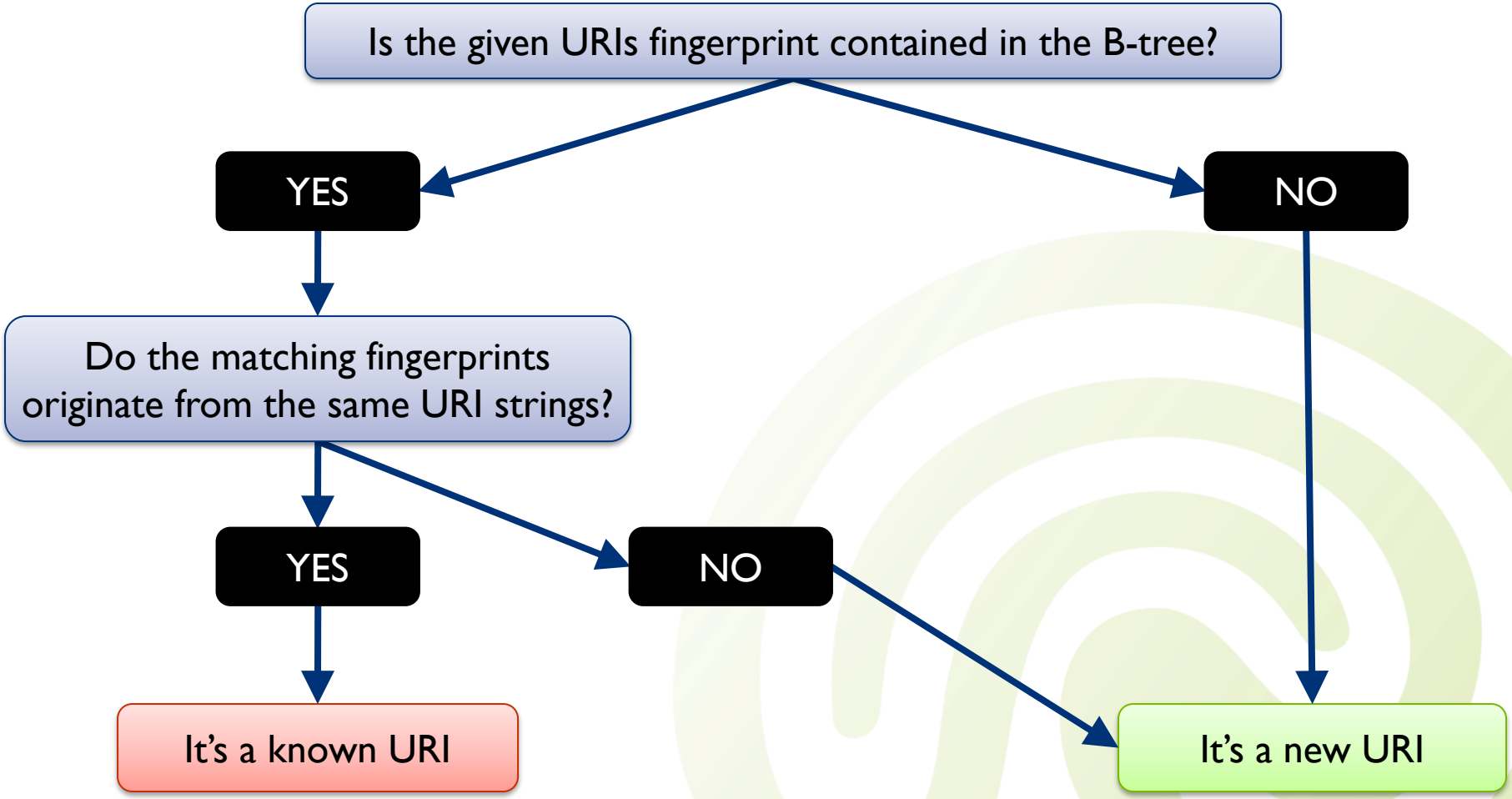
A B-tree:



- B-trees can be searched efficiently
- **Numerical comparisons** can be done quickly

The Duplicate URI Checker

The whole process:



The Duplicate URI Checker

- **Problem size?**
  - Let's say we have collected **1 billion URIs**
  - Each URI's fingerprint requires at least **16 bytes**
  - To store **1 billion URIs,** we need **about 15 GB of storage**
  - Plus much more space to store URI strings and metadata
- There are two options of storage:
  - A distributed **main memory** index
  - Put it on **disk**
- In both cases, it would be reasonable to **enforce some locality** by grouping URIs together that usually will be accessed in **quick succession**

The Duplicate URI Checker

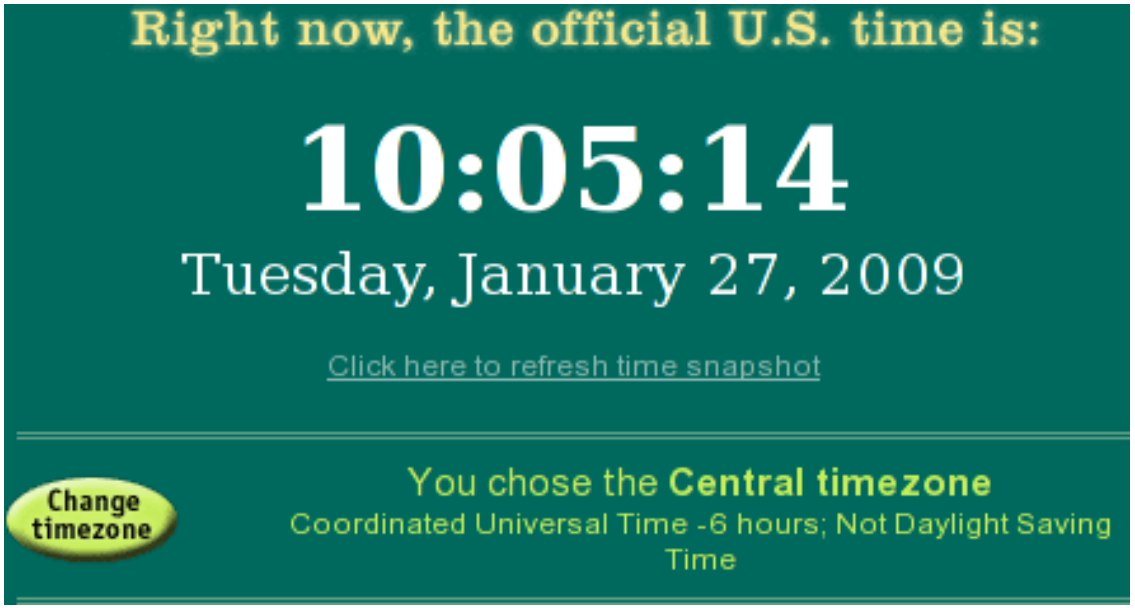
- **How to enforce locality?**
- **Observation:** URIs having the same hostname are usually accessed together in crawling
- **Idea:** Take **two fingerprints** per URI
  - One for the **hostname**
  - One for the **rest**
  - **Concatenate** both to form a URI's fingerprint
- Then, URIs of the **same hostname** are located in the **same sub-tree** of the index

`http://en.wikipedia.org/wiki/New_South_Wales`

`57c6caa6d66b0d64f9147075a219215f` `c4b591806d11a2ffa9b81c92348eeaf9`

The Duplicate Content Checker

- In principle, we could check for **duplicate content** in the same way as we did it for **duplicate URIs**
- **But what about this page?**



- Or, think of pages with ads that change on every visit

The Duplicate Content Checker

- This problem is called **near-duplicate detection**

- **First step:** Focus on content!
  - **Remove all styling information** from the Web resource
  - Convert the resource into a text-only view
  - Drop textual information like navigation structures
  - Drop images and dynamic content

The Duplicate Content Checker

Example:

Institute for Information Systems

The Institute for Information Systems at Technische Universität Braunschweig, Germany, focuses on research and teaching in the area of databases and information systems.

The Duplicate Content Checker

- After this step, the problem amounts to near-duplicate detection on **plain text documents** (word sequences)
- It can be solved using a technique called **shingling**
  - **Given:** A positive number *k* and a sequence of terms *d*
  - **Definition:** The ***k*-shingles of *d*** are the set of **all consecutive sequences of *k* terms in *d***
- Example:
  - *d* = "a rose is a rose is a rose"
  - *k* = 4 (a typical value used in the near-duplicate detection of Web pages)
  - The 4-shingles of *d* are:

    "a rose is a" "rose is a rose" "is a rose is"

The Duplicate Content Checker

- **Intuitive idea:** Two documents are near-duplicates if the two sets of shingles generated from them are nearly the same
- **A more precise definition:**
  Let *d* and *d'* be documents and let ***S*(*d*)** and ***S*(*d'*)** be their respective **sets of shingles**
- Remember the **Jaccard coefficient** from fuzzy retrieval
- We use it to measure the **overlap** between the sets:

$$J\Big(S(d), S(d')\Big) = \frac{|S(d) \cap S(d')|}{|S(d) \cup S(d')|}$$

- Define *d* and *d'* to be near-duplicates if J(…) is "large," e.g. larger than 0.9

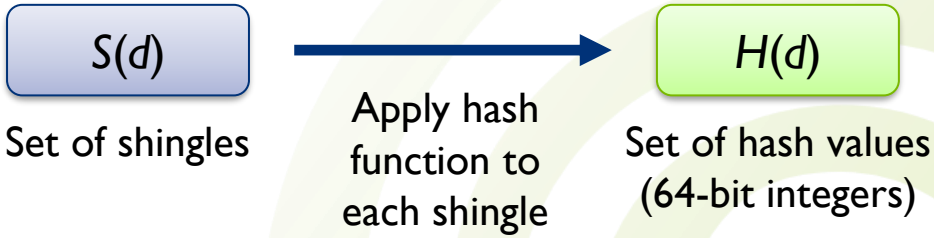
The Duplicate Content Checker

$$J\left(S(d), S(d')\right) = \frac{|S(d) \cap S(d')|}{|S(d) \cup S(d')|}$$

- Computing the value of $J(S(d), S(d'))$ directly is easy
- **Complexity is O(n log n)**
  - Sort each set of shingles
  - Find intersection and union by merging the two sorted lists
- However, the **typical situation** is different:
  - We already have a large document collection
  - We want to check whether a new document is a near-duplicate
  - Compare the new document with all existing ones?
    - Too expensive, **we need some clever indexing technique…**

The Duplicate Content Checker

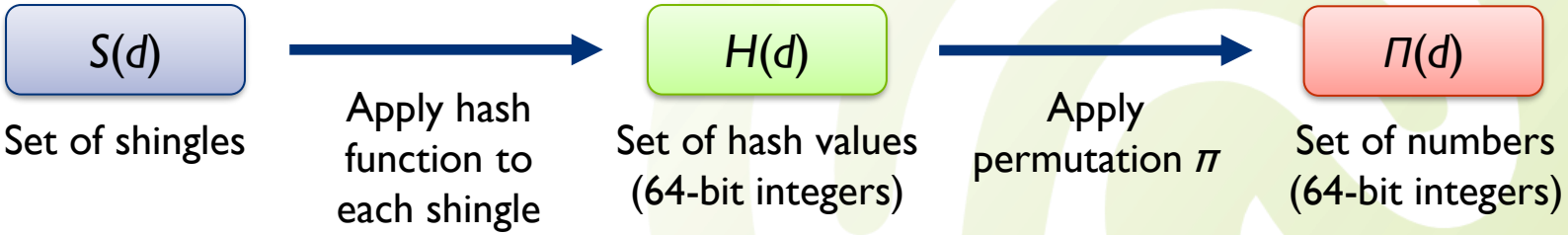
- A very clever indexing technique (to be discussed later) relies on a **randomized approximation algorithm** for computing $J(S(d), S(d'))$
- To explain this algorithm, we need the following:
  - Map every shingle into a **hash value** over a large space, say the space of all **64-bit integers**
  - Let $\boldsymbol{H(d)}$ be the **set of hash values** derived from $S(d)$



  - Then, it is $\boldsymbol{J(S(d), S(d')) \approx J(H(d), H(d'))}$

The Duplicate Content Checker

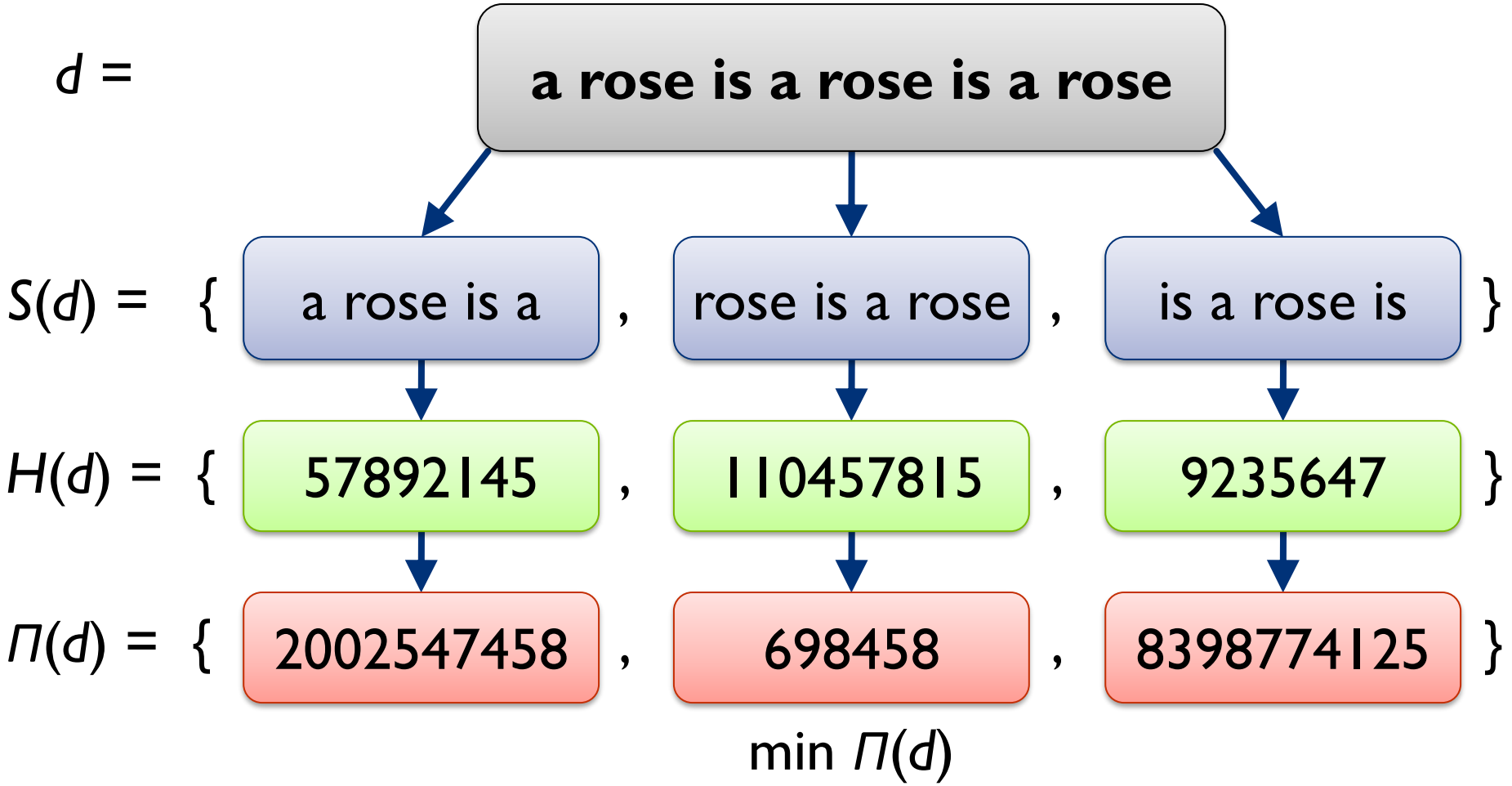
- Let π be a **random permutation** on the set of all 64-bit integers, i.e. π is a **one-to-one function** that **maps any 64-bit integer to some 64-bit integer**
  - The simplest permutation is the **identity mapping** (every 64-bit number is mapped to itself)
  - Another example of a permutation is $\pi(x) = (x + 1) \bmod 2^{64}$
  - Here, "random" means **chosen at random** according to the **uniform distribution** over the set of all permutations on the set of all 64-bit integers
- When applying a single permutation π to each hash value in $H(d)$, we get a new **set of 64-bit numbers $\Pi(d)$**



- Furthermore, let **min($\Pi(d)$)** be the **smallest number in $\Pi(d)$**

The Duplicate Content Checker

Illustration:



The Duplicate Content Checker

- Then, the following is true, for any documents *d* and *d'*:

$$J\left(H(d), H(d')\right) = \Pr\left(\min\left(\Pi(d)\right) = \min\left(\Pi(d')\right)\right)$$

  - Intuitive meaning: The overlap between the sets of shingles (measured by the Jaccard coefficient) is same as the probability that their corresponding **hash sets** have the **same smallest number when permuted randomly**

- This identity will allow us to build an indexing schema that supports efficient near-duplicate detection for new documents

- How to prove this identity?

The Duplicate Content Checker

- Given:
  - $S(d)$ and $S(d')$: The sets of shingles
  - $H(d)$ and $H(d')$: The sets of corresponding hash values
  - π: A **random** permutation on the set of 64-bit integers
    - Random = Chosen uniformly over the set of all permutations on the 64-bit numbers
  - $\Pi(d)$ and $\Pi(d')$: The result of applying π to $H(d)$ and $H(d')$
  - $\min(\Pi(d))$ and $\min(\Pi(d'))$: The minima in $\Pi(d)$ and $\Pi(d')$
- Note that π, $\Pi(d)$, $\Pi(d')$, $\min(\Pi(d))$, and $\min(\Pi(d'))$ are **random**
- **We have to prove the following:**

$$J\left(H(d), H(d')\right) = \Pr\left(\min\left(\Pi(d)\right) = \min\left(\Pi(d')\right)\right)$$

The Duplicate Content Checker

We have to prove the following:

$$J\left(H(d), H(d')\right) = \Pr\left(\min\left(\Pi(d)\right) = \min\left(\Pi(d')\right)\right)$$

Proof:

- First, represent the **sets** $H(d)$ and $H(d')$ as **bit strings** of length 2^{64}, where the i-th bit is set if number i is contained in $H(d)$ or $H(d')$, respectively

Example: $H(d) = \{1, 2, 4, \ldots, 2^{64} - 2, 2^{64} - 1\}$ $H(d') = \{0, 2, 4, 5, \ldots, 2^{64} - 1\}$

	0	1	2	3	4	5	…	$2^{64}-3$	$2^{64}-2$	$2^{64}-1$
$H(d)$	0	1	1	0	1	0	…	0	1	1
$H(d')$	1	0	1	0	1	1	…	0	0	1

- The permutation π corresponds to a **random swapping of columns,** resulting in **bit strings $\Pi(d)$ and $\Pi(d')$**

	0	1	2	3	4	5	…	$2^{64}-3$	$2^{64}-2$	$2^{64}-1$
$\Pi(d)$	0	1	0	1	0	1	…	1	1	0
$\Pi(d')$	0	1	1	0	1	1	…	1	0	0

The Duplicate Content Checker

$$J\left(H(d), H(d')\right) = \Pr\left(\min\left(\Pi(d)\right) = \min\left(\Pi(d')\right)\right)$$

	0	1	2	3	4	5	…	$2^{64}-3$	$2^{64}-2$	$2^{64}-1$
$\Pi(d)$	0	1	0	1	0	1	…	1	1	0
$\Pi(d')$	0	1	1	0	1	1	…	1	0	0

Proof (continued):

- min($\Pi(d)$) and min($\Pi(d')$) are the **positions of the first "1" columns**
- Pr(min($\Pi(d)$) = min($\Pi(d')$)) is the probability that $\Pi(d)$ and $\Pi(d')$ have their **first "1" column at the same position**

- What's the probability that both $\Pi(d)$ and $\Pi(d')$ have their first "1" column at the same position?
  - Since "0–0" columns can be ignored, it's the same as the probability that the first non-"0–0" column is a "1–1" column
- Therefore:
  - Pr(min($\Pi(d)$) = min($\Pi(d')$)) = Pr(the first non-"0–0" column is a "1–1" column)

The Duplicate Content Checker

$$J\left(H(d), H(d')\right) = \Pr\left(\min\left(\Pi(d)\right) = \min\left(\Pi(d')\right)\right)$$

Pr(min(*Π*(*d*)) = min(*Π*(*d'*))) = Pr(the first non-"0–0" column is a "1–1" column)

	0	1	2	3	4	5	…	$2^{64}-3$	$2^{64}-2$	$2^{64}-1$
Π(*d*)	0	1	0	1	0	1	…	1	1	0
Π(*d'*)	0	1	1	0	1	1	…	1	0	0

Proof (continued):

- What's Pr(the first non-"0–0" column is a "1–1" column)?
  - Since *Π* is uniformly distributed over all permutations of columns:

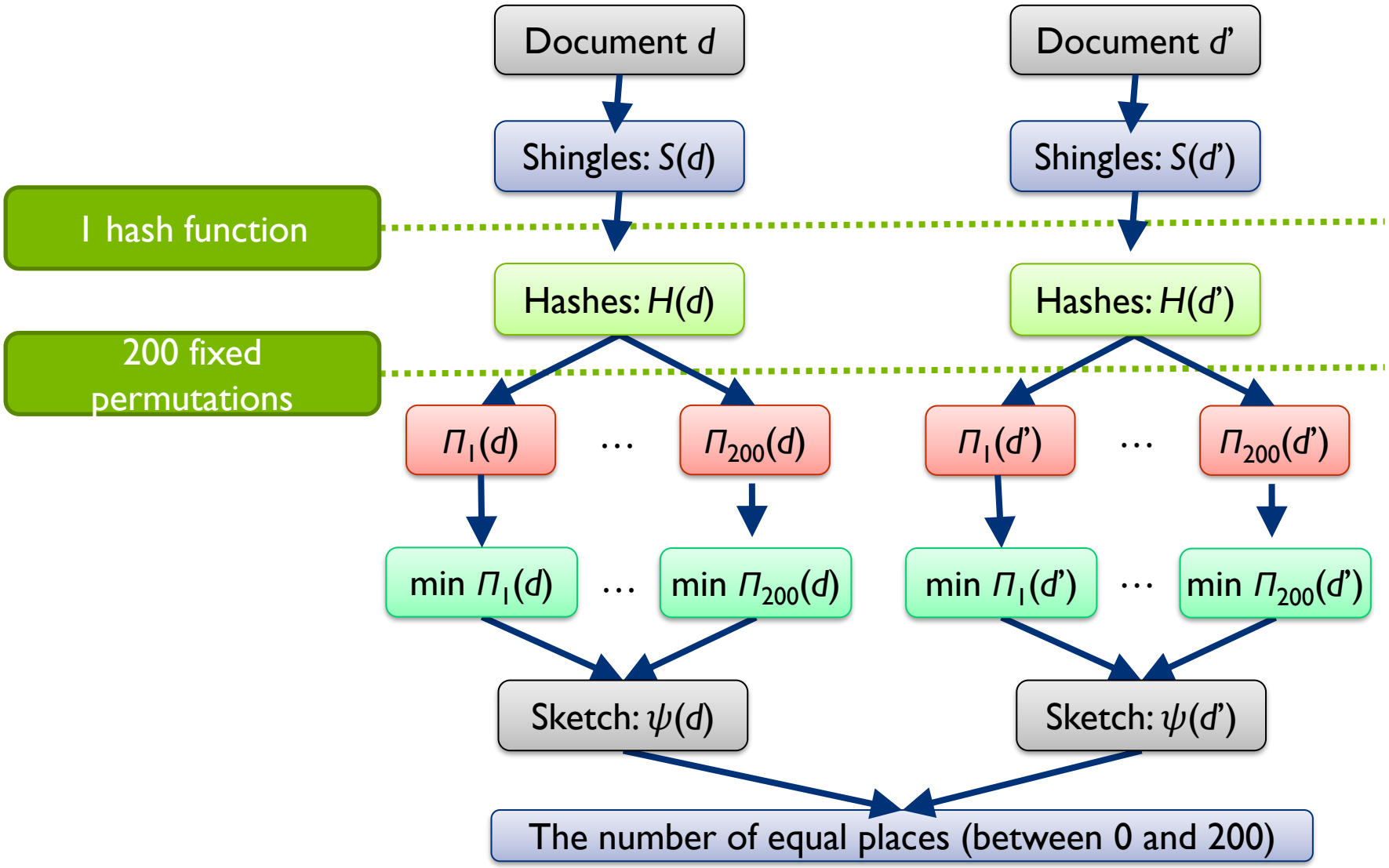
$$\Pr(\text{the first non-"0–0" column is a "1–1" column}) = \frac{\text{number of "1–1" columns}}{\text{number of non-"0–0" columns}}$$

  - This is exactly the definition of the Jaccard coefficient!
    - QED

The Duplicate Content Checker

- **That's great!**
  - We can **estimate the overlap** between $H(d)$ and $H(d')$ by applying random permutations and comparing the minima
    - Estimate $\Pr(\min(\Pi(d)) = \min(\Pi(d')))$ by drawing random samples
    - The literature says that **200 is a good number** of random permutations/samples to use in practice
  - Therefore, let $\pi_1, \pi_2, \ldots, \pi_{200}$ be a fixed(!) set of permutations, which has been generated randomly
  - Let $\boldsymbol{\psi(d) = (\min(\pi_1(d)), \min(\pi_2(d)), \ldots, \min(\pi_{200}(d)))}$
  - $\psi(d)$ is called the **sketch** of d
  - Then, the Jaccard coefficient of $H(d)$ and $H(d')$ can be estimated by counting the number of places in which $\psi(d)$ and $\psi(d')$ agree
  - Since $J(H(d), H(d'))$ and $J(S(d), S(d'))$ usually are very similar, we finally arrived at a method for **estimating $J(S(d), S(d'))$**

The Duplicate Content Checker



Divide it by 200 and get an approximation of J(S(d), S(d'))

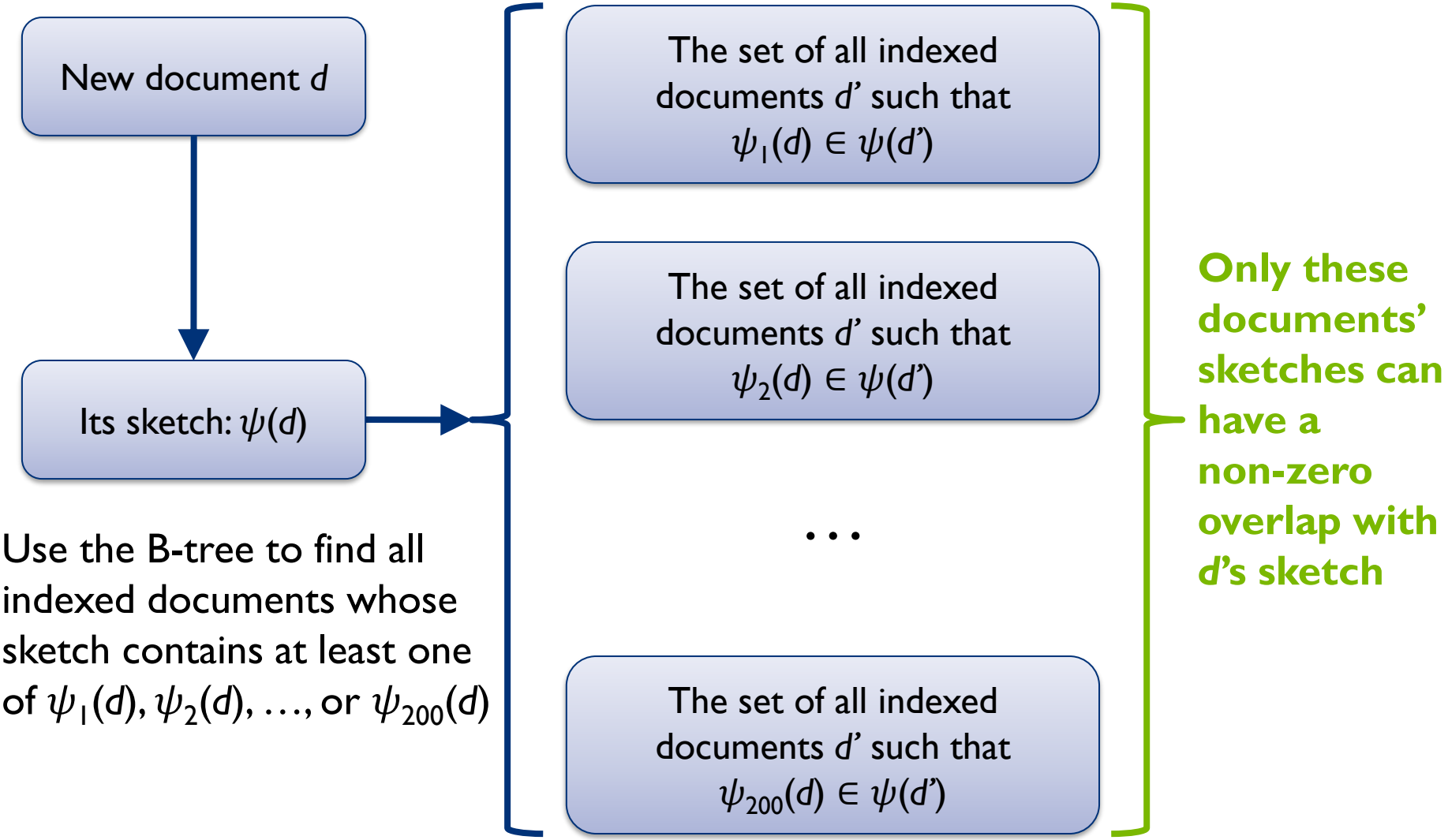
The Duplicate Content Checker

- **Now back to our initial problem:**
  - Given:
    - A large collection of documents (and their pre-computed sketches)
    - A new document d_{new}
  - Near-duplicates of d_{new} can be found by computing the sketch of d_{new} and comparing to the sketches of all existing docs
  - This is much faster than computing shingles and their overlap
- **But:**
  - **Finding near-duplicates is still quite expensive** if we have to compare the sketch of every new document to all the sketches of the documents that already have been indexed
  - Linear complexity in the size of the index… Bad!

The Duplicate Content Checker

- Again, there is a trick:
  - For each indexed document d and each entry $\psi_i(d)$ of its sketch $\psi(d)$, create a **pair $(\psi_i(d)$, id(d))**, where id(d) is d's document id
  - If n is the number of documents, we get **$200 \cdot n$ pairs** in total
  - Finally, create a **B-tree index** that is sorted by the $\psi_i(d)$s
  - Then, for each new document d, we can scan through its sketch and look up all other documents having at least one number in common—only these have to checked in detail…

The Duplicate Content Checker



Use the B-tree to find all indexed documents whose sketch contains at least one of $\psi_1(d), \psi_2(d), \ldots,$ or $\psi_{200}(d)$

The Duplicate Content Checker

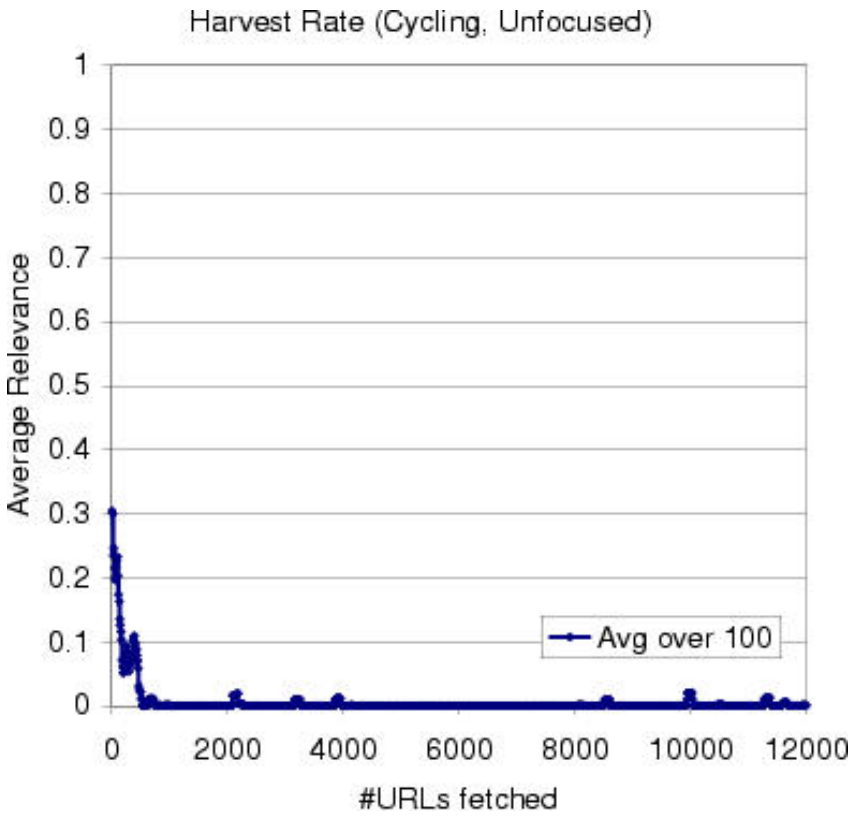
- **Extension:**
  - If we consider two documents to be near-duplicates if their sketches have at least *m* matching places, we restrict our search to all documents in the B-tree which have at least *m* numbers in common
  - The set of all these documents can be found by intersecting the sets of documents having at least 1 number in common

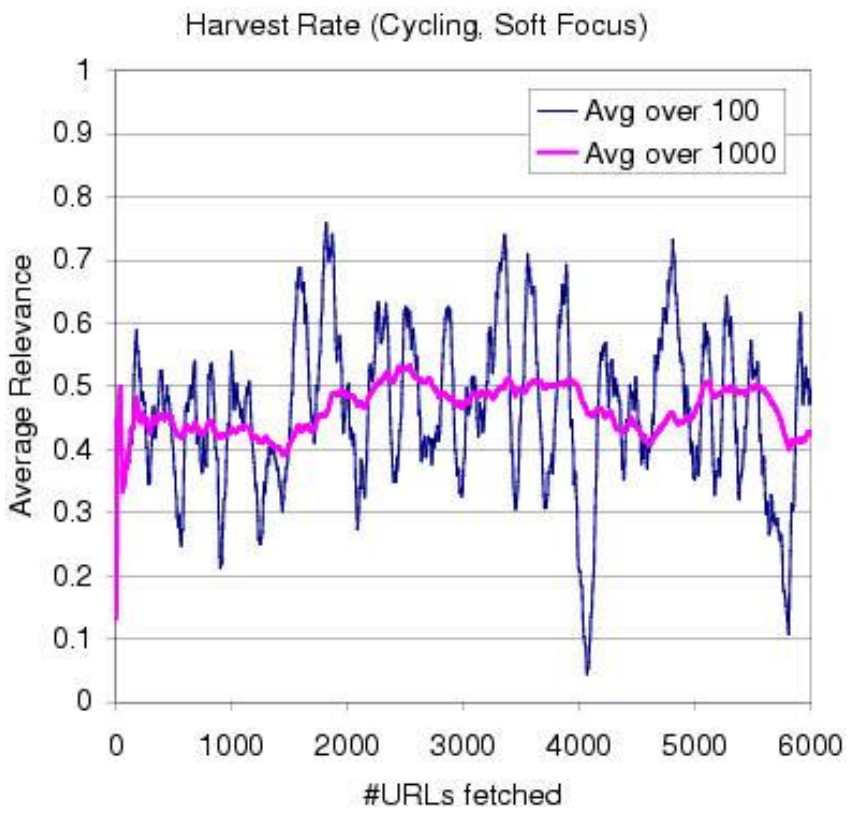
Focused Crawling

- Now, assume that you own a Web search engine that focuses on a **specific topic,** e.g. sports
  - Then, it would be reasonable to do some kind of **"focused crawling"** to avoid crawling unrelated pages
- How to do it?
  - Train a **classifier** that is able to detect whether a web page is about the relevant topic
  - Start crawling with a hand-crafted set of highly on-topic pages
  - When crawling, only follow out-links of on-topic pages
- Possible extension:
  - For any yet unseen page, estimate the probability that this page is on-topic using a clever model
  - Do the crawl in order of descending probabilities

Focused Crawling

Comparison to unfocused crawling:





Next Lecture

- Exploiting the Web graph for ranking
  - HITS
  - PageRank