Overview

1. Size of the web
2. Duplication detection

Outline

1. Size of the web
2. Duplication detection

Size of the web: Who cares?

- Media
- Users
  - They may switch to the search engine that has the best coverage of the web.
  - Users (sometimes) care about recall. If we underestimate the size of the web, search engine results may have low recall.
- Search engine designers (how many pages do I need to be able to handle?)
- Crawler designers (which policy will crawl close to N pages?)
What is the size of the web? Any guesses?

Simple method for determining a lower bound

- OR-query of frequent words in a number of languages
- According to this query: Size of web $\geq 21,450,000,000$ on 2007.07.07
- Big if: Page counts of google search results are correct. (Generally, they are just rough estimates.)
- But this is just a lower bound, based on one search engine.
- How can we do better?

Size of the web: Issues

- The “dynamic” web is infinite.
  - Any sum of two numbers is its own dynamic page on Google. (Example: “2+4”)
  - Many other dynamic sites generating infinite number of pages
- The static web contains duplicates – each “equivalence class” should only be counted once.
- Some servers are seldom connected.
  - Example: Your laptop
  - Is it part of the web?

“Search engine index contains $N$ pages”: Issues

- Can I claim a page is in the index if I only index the first 4000 bytes?
- Can I claim a page is in the index if I only index anchor text pointing to the page?
  - There used to be (and still are?) billions of pages that are only indexed by anchor text.
How can we estimate the size of the web?

Sampling methods

- Random queries
- Random searches
- Random IP addresses
- Random walks

Variant: Estimate relative sizes of indexes

There are significant differences between indexes of different search engines.
- Different engines have different preferences.
  - max url depth, max count/host, anti-spam rules, priority rules etc.
- Different engines index different things under the same URL.
  - anchor text, frames, meta-keywords, size of prefix etc.

Relative Size from Overlap
[Bharat & Broder, 98]

Sample URLs randomly from A and vice versa

\[ A \cap B = \frac{1}{2} \times \text{Size A} \]

\[ A \cap B = \frac{1}{6} \times \text{Size B} \]

\[ \text{Size A} \times \text{Size B} = \frac{1}{2} \times \frac{1}{6} \]

Each test involves: (i) Sampling (ii) Checking
Sampling URLs

- Ideal strategy: Generate a random URL
- Problem: Random URLs are hard to find (and sampling distribution should reflect “user interest”)
- Approach 1: Random walks / IP addresses
  - In theory: might give us a true estimate of the size of the web (as opposed to just relative sizes of indexex)
- Approach 2: Generate a random URL contained in a given engine
  - Suffices for accurate estimation of relative size

Random URLs from random queries

- Idea: Use vocabulary of the web for query generation
- Vocabulary can be generated from web crawl
- Use conjunctive queries $w_1 \text{ AND } w_2$
  - Example: vocalists AND rsi
- Get result set of one hundred URLs from the source engine
- Choose a random URL from the result set
- This sampling method induces a weight $W(p)$ for each page $p$.
- Method was used by Bharat and Broder (1998).

Checking if a page is in the index

- Either: Search for URL if the engine supports this
- Or: Create a query that will find doc $d$ with high probability
  - Download doc, extract words
  - Use 8 low frequency word as AND query
  - Call this a strong query for $d$
  - Run query
  - Check if $d$ is in result set
- Problems
  - Near duplicates
  - Redirects
  - Engine time-outs

Computing Relative Sizes and Total Coverage [BB98]

- $a = \text{AltaVista}$, $e = \text{Excite}$, $h = \text{HotBot}$, $i = \text{Infoseek}$

\[
\begin{align*}
\text{Overlap terms:} & \\
\epsilon_1 & = f_{eh} * a - f_{ha} * h \\
\epsilon_2 & = f_{el} * a - f_{la} * i \\
\epsilon_3 & = f_{ae} * a - f_{ea} * e \\
\epsilon_4 & = f_{hi} * h - f_{ih} * i \\
\epsilon_5 & = f_{he} * h - f_{eh} * e \\
\epsilon_6 & = f_{el} * e - f_{le} * i \\
\end{align*}
\]

- Arbritrarily, let $a = 1$. $\text{We have 6 equations and 3 unknowns.}$
- Solve for $e, h$ and $i$ to minimize $\sum \epsilon_i^2$
- Compute engine overlaps.
- Re-normalize so that the total joint coverage is 100%
**Random searches**

- Choose random searches extracted from a search engine log (Lawrence & Giles 97)
- Use only queries with small result sets
- For each random query: compute ratio size($r_1$)/size($r_2$) of the two result sets
- Average over random searches

---

**Advantages & disadvantages**

- Statistically sound under the induced weight.
- Biases induced by random query
  - Query Bias: Favors content-rich pages in the language(s) of the lexicon
  - Ranking Bias: Solution: Use conjunctive queries & fetch all
  - Checking Bias: Duplicates, impoverished pages omitted
  - Document or query restriction bias: engine might not deal properly with 8 words conjunctive query
  - Malicious Bias: Sabotage by engine
  - Operational Problems: Time-outs, failures, engine inconsistencies, index modification.

- **Advantage**
  - Might be a better reflection of the human perception of coverage

- **Issues**
  - Samples are correlated with source of log (unfair advantage for originating search engine)
  - Duplicates
  - Technical statistical problems (must have non-zero results, ratio average not statistically sound)

---

**Random searches [Lawr98, Lawr99]**

- 575 & 1050 queries from the NEC RI employee logs
- 6 Engines in 1998, 11 in 1999
- Implementation:
  - Restricted to queries with < 600 results in total
  - Counted URLs from each engine after verifying query match
  - Computed size ratio & overlap for individual queries
  - Estimated index size ratio & overlap by averaging over all queries
Random IP addresses [ONei97, Lawr99]

- [Lawr99] exhaustively crawled 2500 servers and extrapolated
- Estimated size of the web to be 800 million

Advantages and disadvantages

**Advantages**
- Can, in theory, estimate the size of the accessible web (as opposed to the (relative) size of an index)
- Clean statistics
- Independent of crawling strategies

**Disadvantages**
- Many hosts share one IP (→ oversampling)
- Hosts with large web sites don’t get more weight than hosts with small web sites (→ possible undersampling)
- Sensitive to spam (multiple IPs for same spam server)
- Again, duplicates

Random IP addresses [Lawrence & Giles ‘99]

- Generate random IP addresses
- Find a web server at the given address
  - If there’s one
- Collect all pages from server.
  
  [Link](http://digitalarchive.oclc.org/da/ViewObject.jsp?objid=000003447)
**Random walks**

[Henzinger et al. WWW9]

- View the Web as a directed graph
- Build a random walk on this graph
  - Includes various “jump” rules back to visited sites
    - Does not get stuck in spider traps!
    - Can follow all links!
  - Converges to a stationary distribution
    - Must assume graph is finite and independent of the walk.
    - Conditions are not satisfied (cookie crumbs, flooding)
    - Time to convergence not really known
- Sample from stationary distribution of walk
- Use the “strong query” method to check coverage by SE

**Advantages & disadvantages**

- **Advantages**
  - “Statistically clean” method at least in theory!
  - Could work even for infinite web (assuming convergence) under certain metrics.
- **Disadvantages**
  - List of seeds is a problem.
  - Practical approximation might not be valid.
  - Non-uniform distribution
    - Subject to link spamming

**Dependence on seed list**

- How well connected is the graph? [Broder et al., WWW9]

**Conclusion**

- Many different approaches to web size estimation.
- None is perfect.
- The problem has gotten much harder.
- There hasn’t been a good study for a couple of years.
- Great topic for a thesis!
Outline

1. Size of the web
2. Duplication detection

How to detect exact duplicates?

Duplicate documents

- The web is full of duplicated content
- Strict duplicate detection = exact match
  - Not as common
- But many, many cases of near duplicates
  - E.g., Last modified date the only difference

Duplicate/Near-Duplicate Detection

- **Duplication**: Exact match can be detected with fingerprints
- **Near-Duplication**: Approximate match
  - Overview
    - Compute syntactic similarity with an edit-distance measure
    - Use similarity threshold to detect near-duplicates
    - E.g., Similarity > 80% => Documents are “near duplicates”
    - Not transitive though sometimes used transitively
Computing Similarity

- Features:
  - Segments of a document (natural or artificial breakpoints)
  - Shingles (Word N-Grams)
  - a rose is a rose is a rose →
    a_rose_is_a
    rose_is_a_rose
    is_a_rose_is
  - Similarity Measure between two docs (= sets of shingles)
    - Set intersection [Brod98]
      (Specifically, Size_of_Intersection / Size_of_Union)
    - Jaccard measure

Shingles + Set Intersection

- Computing exact set intersection of shingles between all pairs of documents is expensive/intractable
  - Approximate using a cleverly chosen subset of shingles from each (a sketch)
  - Estimate (size_of_intersection / size_of_union) based on a short sketch

Sketch of a document

- Create a “sketch vector” (of size ~200) for each document
  - Documents that share ≥ t (say 80%)
    corresponding vector elements are near duplicates
  - For doc D, sketch[D][i] is as follows:
    - Let f map all shingles in the universe to 0..2^m
      (e.g., f = fingerprinting)
    - Let πᵢ be a random permutation on 0..2^m
    - Pick MIN {πᵢ(f(s))} over all shingles s in D

Computing Sketch[i] for Doc1

- Start with 64-bit f(shingles)
- Permute on the number line with πᵢ
- Pick the min value
Test if \( \text{Doc1.Sketch[i]} = \text{Doc2.Sketch[i]} \)

- **Document 1**
- **Document 2**

Are these equal?

Test for 200 random permutations: \( \pi_1, \pi_2, \ldots, \pi_{200} \)

### However...

- **Document 1**
- **Document 2**

A = B iff the shingle with the MIN value in the union of Doc1 and Doc2 is common to both (i.e., lies in the intersection)

This happens with probability:

\[
\frac{\text{Size of intersection}}{\text{Size of union}}
\]

### Key Observation

- For columns \( C_i, C_j \), four types of rows:

\[
\begin{array}{cc}
C_i & C_j \\
A & 1 & 1 \\
B & 1 & 0 \\
C & 0 & 1 \\
D & 0 & 0 \\
\end{array}
\]

- Overload notation: \( A = \# \) of rows of type A

### Set Similarity

- **Set Similarity** (Jaccard measure)
  
  \[\text{Jaccard}(C_i, C_j) = \frac{|C_i \cap C_j|}{|C_i \cup C_j|}\]

- View sets as columns of a matrix; one row for each element in the universe. \( a_{ij} = 1 \) indicates presence of item \( i \) in set \( j \)

- Example

\[
\begin{array}{cc}
c_1 & c_2 \\
0 & 1 \\
1 & 0 \\
1 & 1 \\
0 & 0 \\
1 & 1 \\
0 & 1 \\
\end{array}
\]

\[\text{Jaccard}(C_1, C_2) = 2/5 = 0.4\]
“Min” Hashing

- Randomly permute rows
- Hash \( h(C_i) = \text{index of first row with 1 in column } C_i \)
- **Surprising Property**
  \[ P \left[ h(C_i) = h(C_j) \right] = \text{Jaccard}(C_i, C_j) \]
- Why?
  - Both are \( A/(A+B+C) \)
  - Look down columns \( C_i, C_j \) until first non-Type-D row
  - \( h(C_i) = h(C_j) \iff \text{type A row} \)

### Estimating \( P[h(C_1) = h(C_2)] \)

- For a particular permutation, the value of \( h(C_1) = h(C_2) \) is either 0 or 1.
- The average of values of \( h(C_1) = h(C_2) \) over many permutations is an estimate of \( P[h(C_1) = h(C_2)] \).
- \( P[h(C_1) = h(C_2)] = E_h[h(C_1) = h(C_2)] \).
- We estimate \( P[h(C_1) = h(C_2)] \) by computing the average of \( h(C_1) = h(C_2) \) for the 200 permutations that correspond to the elements of the sketch vector.

### Example

<table>
<thead>
<tr>
<th>Signatures</th>
<th>( S_1 )</th>
<th>( S_2 )</th>
<th>( S_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perm 1 = (12345)</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Perm 2 = (54321)</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Perm 3 = (34512)</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( C_1 )</th>
<th>( C_2 )</th>
<th>( C_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( R_2 )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( R_3 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( R_4 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( R_5 )</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Implementation Trick

- Permuting rows even once is prohibitive
- **Row Hashing**
  - Pick \( P \) hash functions \( h_k : \{1, \ldots, n\} \rightarrow \{1, \ldots, O(n)\} \)
  - Ordering under \( h_k \) gives random row permutation
- **One-pass Implementation**
  - For each \( C_i \) and \( h_k \), keep “slot” for min-hash value
  - Initialize all slot\((C_i, h_k)\) to infinity
  - Scan rows in arbitrary order looking for 1’s
    - Suppose row \( R_j \) has 1 in column \( C_i \)
    - For each \( h_k \),
      - if \( h_k(j) < \text{slot}(C_i, h_k) \), then \( \text{slot}(C_i, h_k) \leftarrow h_k(j) \)
Example

<table>
<thead>
<tr>
<th></th>
<th>C₁</th>
<th>C₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R₂</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R₃</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R₄</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R₅</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- \( h(x) = x \mod 5 \)
- \( g(x) = (2x + 1) \mod 5 \)

Example

<table>
<thead>
<tr>
<th></th>
<th>C₁</th>
<th>C₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R₂</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R₃</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R₄</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R₅</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>C₁</th>
<th>C₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R₂</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R₃</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R₄</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R₅</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- \( h(x) = x \mod 5 \)
- \( g(x) = (2x + 1) \mod 5 \)

<table>
<thead>
<tr>
<th></th>
<th>C₁</th>
<th>C₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R₂</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R₃</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R₄</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R₅</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>C₁</th>
<th>C₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R₂</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R₃</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R₄</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R₅</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- \( h(x) = x \mod 5 \)
- \( g(x) = (2x + 1) \mod 5 \)
<table>
<thead>
<tr>
<th>C₁</th>
<th>C₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>1</td>
</tr>
<tr>
<td>R₂</td>
<td>0</td>
</tr>
<tr>
<td>R₃</td>
<td>1</td>
</tr>
<tr>
<td>R₄</td>
<td>1</td>
</tr>
<tr>
<td>R₅</td>
<td>0</td>
</tr>
</tbody>
</table>

- \( h(x) = x \mod 5 \)
- \( g(x) = (2x + 1) \mod 5 \)

\[
\begin{array}{|c|c|c|c|}
\hline
C₁ slots & C₂ slots \\
\hline
h(1) = 1 & 1 & 1 & - - \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
C₁ slots & C₂ slots \\
\hline
h(1) = 1 & 1 & 1 & - - \\
\hline
\end{array}
\]
### Efficient near-duplicate detection

Now we have an extremely efficient method for estimating a Jaccard coefficient for a single pair of two documents. But we still have to estimate $N^2$ coefficients where $N$ is the number of web pages. Still intractable

One solution: locality sensitive hashing (LSH)
Another solution: sorting (Henzinger 2006)

<table>
<thead>
<tr>
<th></th>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$C_1$ slots</th>
<th>$C_2$ slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$h(1)$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$g(1)$</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R_2$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$h(2)$</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$g(2)$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R_3$</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>$h(3)$</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$g(3)$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$h(4)$</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$g(4)$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R_5$</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>$h(5)$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$g(5)$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Resources

- IIR 19
- Phelps & Wilensky, Robust hyperlinks & locations, 2002.
- Bar-Yossef & Gurevich, Random sampling from a search engine’s index, WWW 2006.