IRLbot: Scaling to 6 Billion Pages and Beyond

Presented by Xiaoming Wang

Hsin-Tsang Lee, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov

Internet Research Lab
Computer Science Department
Texas A&M University

April 25, 2008
Agenda

• Introduction and challenges
• Background
• Overcoming the challenges
  – Scalability
  – Spam and reputation
  – Politeness
• Experiments
• Conclusion

we are here
Introduction

• WWW has evolved from a handful of pages to billions of diverse objects
  – In January 2008, Google reported indexing 30 billion documents and Yahoo 37 billion (see paper for details)

• Search engines consist of two fundamental parts
  – Web crawlers
  – Data miners

• Challenges
  – Scalability
  – Spam avoidance
  – Politeness

our focus in this paper
Challenges – Scalability

- Inherent tradeoff between scalability, performance, and resource usage
  - **Scalability**: number of pages $N$ the crawler can handle
  - **Performance**: speed $S$ at which the crawler discovers the web as a function of $N$
  - **Resource usage**: CPU and RAM requirements $\Sigma$ needed to download $N$ pages at average speed $S$

- Previous research can satisfy any two objectives (i.e., large slow crawls, fast small crawls, or fast large crawls with unlimited resources)

- **Our goal**: achieve large $N$ (trillions of pages) with fixed $S$ ($1000+$ pages/sec) and modest $\Sigma$ (single server)
Challenges – Spam Avoidance

• Experience shows that BFS eventually becomes trapped in useless content
  – The queue of pending URLs is filled with spam links and infinite auto-generated webs
  – The DNS resolver is overloaded with new hostnames that are dynamically created within a single domain
  – Crawler is “bogged down” in synchronized delay attacks of certain spammers

• Prior research has not attempted to avoid spam or even document its effect on the collected data

• **Our goal**: prevent low-ranked (spam) domains from impacting crawler performance
Challenges – Politeness

• Web crawlers often get in trouble with webmasters for slowing down their servers
  – Sometimes even reading the robots.txt file generates a complaint

• Per-website and per-IP hit limits are simple
  – However, even with spam avoidance, the entire RAM eventually gets filled with URLs from a small set of hosts (e.g., ebay, blogspot) and the crawler simply choke on its politeness

• Previous algorithms do not address this issue

• Our goal: crawl large legitimate sites without stalling
Agenda

• Introduction and challenges
• Background
• Overcoming the challenges
  – Scalability
  – Spam and reputation
  – Politeness
• Experiments
• Conclusion
Background – Crawler Objectives

• Ideal task is to start from a set of seed URLs $\Omega_0$ and eventually crawl the set of all pages $\Omega_\infty$.
• Crawler may reorder URLs to achieve some “good” coverage of useful pages $\Omega_U \subseteq \Omega_\infty$ in some finite amount of time.
• We call an algorithm non-scalable if it
  1) imposes hard limits on any of the following metrics
     • Max. # pages per host, #hosts per domain, #domains in the Internet, #pages in the crawl
  2) is unable to maintain crawling speed when these metrics become arbitrarily large.
• Scalability is an important objective.
**Background – Crawler Operation**

- For each URL $u$ in the queue $Q$ of pending pages, the crawler downloads $u$’s HTML page and extracts new URLs $u_1, u_2, \ldots, u_k$.
- For each $u_i$, the crawler verifies its uniqueness using `URLseen` and checks compliance with robots.txt using `RobotsCache`.
- It then adds the passing URLs to $Q$ and `URLseen`.
- Last, it updates `RobotsCache` if necessary.
- The crawler may also maintain a `DNScache` structure to reduce the load on the local DNS server.
Related Work

- Largest crawl with a disclosed implementation: 473M HTML pages
- Fastest: 816 pages/sec

<table>
<thead>
<tr>
<th>Crawler</th>
<th>Year</th>
<th>Crawl size (HTML pages)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mercator-B [23]</td>
<td>2001</td>
<td>473M</td>
</tr>
<tr>
<td>Polybot [27]</td>
<td>2001</td>
<td>120M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Crawler</th>
<th>URLseen</th>
<th>RobotsCache</th>
<th>DNSCache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RAM</td>
<td>Disk</td>
<td>RAM</td>
</tr>
<tr>
<td>Mercator-B [23]</td>
<td>LRU</td>
<td>batch</td>
<td>LRU</td>
</tr>
<tr>
<td>Polybot [27]</td>
<td>tree</td>
<td>batch</td>
<td>database</td>
</tr>
</tbody>
</table>
Agenda

- Introduction and challenges
- Background
- Overcoming the challenges
  - Scalability
  - Spam and reputation
  - Politeness
- Experiments
- Conclusion
**Scalability**

- One of the bottlenecks in web crawling is the disk I/O in checking URLseen.
- A 6B page crawl requires verification of 394B URLs:
  - Disk structures must be optimized to support the crawling rate.
- Our solution is called **Disk Repository with Update Management (DRUM)**.
- The purpose of DRUM is to allow for efficient storage of large collections of `<key, value>` pairs:
  - Key is a unique identifier of some data (e.g., URL hash).
  - Value is arbitrary information attached to keys (e.g., URL text).
Scalability 2

• Three operations are supported
  – **Check**: verifies uniqueness of a key
  – **Update**: overrides the value if the key exists and otherwise adds a new entry
  – **Check + update**: performs both check and update in one pass through the disk cache

• Unlike prior methods, DRUM is based on disk **bucket sort** rather than variations of merge/insertion sort
  – For certain values of RAM size, DRUM achieves a linear number of disk reads rather than quadratic as in prior work
  – See paper for details
Scalability 3

- Overhead metric \( \omega \): # of bytes written to/read from disk during uniqueness checks of \( lN \) URLs
  - \( l \) is the average # of links per downloaded page
  - \( N \) is the # of crawled pages

- Assume that the average URL length is \( b \) bytes and \( R \) is the size of RAM allocated to URL checks

- Then \( \omega \) can be split into a product two terms: \( \alpha \times blN \)
  - \( blN \) is the number of bytes in all parsed URLs
  - \( \alpha \) is the number of times they are written to/read from disk

- Various methods differ in the first term only
  - Metric \( \alpha \) may be a constant or a linear function of \( N \)
Scalability 4

• **Theorem 1**: The overhead of URLseen batch disk check is \( \omega = \alpha blN \) bytes, where for Mercator:

\[
\alpha = \frac{2(2UH + pHlN)(H + P)}{bR} + 2 + p
\]

Both have \( \alpha \sim \frac{N}{R} \)

- and for Polybot:

\[
\alpha = \frac{2(2Ubq + pbqlN)(b + 4P)}{bR} + p
\]

• **Theorem 2**: DRUM’s URLseen overhead is \( \omega = \alpha blN \) bytes, where:

\[
\alpha = \frac{8M(H + P)(2UH + pHlN)}{bR^2} + 2 + p + \frac{2H}{p}
\]

Notice that \( \alpha \sim \frac{N}{R^2} \)
Scalability 5

• For certain values of $R$, DRUM achieves $N/R^2 \approx 0$ and thus $\alpha$ remains almost constant as $N$ increases.

• Overhead $\alpha$ for dynamically scaling disk size:

<table>
<thead>
<tr>
<th>$N$</th>
<th>$R = 4$ GB</th>
<th>$R = 8$ GB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mercator-B</td>
<td>DRUM</td>
</tr>
<tr>
<td>800M</td>
<td>4.48</td>
<td>2.30</td>
</tr>
<tr>
<td>8B</td>
<td>25</td>
<td>2.7</td>
</tr>
<tr>
<td>80B</td>
<td>231</td>
<td>3.3</td>
</tr>
<tr>
<td>800B</td>
<td>2,290</td>
<td>3.3</td>
</tr>
<tr>
<td>8T</td>
<td>22,887</td>
<td>8.1</td>
</tr>
</tbody>
</table>
Scalability

- Theorem 3: Maximum download rate (in pages/s) supported by the disk portion of URL uniqueness checks is

\[ S = \frac{W}{\alpha b l} \]

- For IRLbot, we set disk speed \( W = 101.25 \text{ MB/s} \) and RAM size \( R = 8 \text{ GB} \)

- Assuming \( N = 8 \) trillion pages, DRUM yields a sustained download rate of \( S = 4,192 \text{ pages/s} \)
  - 10 DRUM servers and 10-gb/s link could give 100 billion pages per month

- For \( N = 8T \), Mercator achieves an average rate of only 1.4 pages/s and Polybot 0.2 pages/s
IRLbot Organization

- **crawling threads**
  - new URLs
  - check + update

- **DRUM URLseen**
  - unique URLs

- **robots & DNS threads**
  - update
  - unable to check

- **robots download queue \( Q_D \)**
  - unique hostnames

- **robots request queue \( Q_E \)**
  - hostnames
  - check + update

- **DRUM RobotsRequested**
  - URLs

- **DRUM RobotsCache**
  - check

- **ready queue \( Q \)**

- **pass robots**
  - fail robots

- **robots-check queue \( Q_R \)**
  - pass budget

- **STAR budget check**

- **BEAST budget enforcement**
Agenda

• Introduction and challenges
• Background
• Overcoming the challenges
  - Scalability
  - Spam and reputation
  - Politeness
• Experiments
• Conclusion
Spam and Reputation

• In our experience, BFS is a poor technique in the presence of spam farms and infinite webs

• Quickly branching sites (1000+ links per page) are potential traps
  – Dominate the queue after 3 levels of BFS with $10^9$ pages
    (exact impact depends on the seed URLs and crawls size)

• Simply restricting the branching factor or the maximum number of pages hôsts per domain is not a viable solution
  – A number of legitimate sites contain over 100 million pages
    and over 10 million virtual hosts
  – Yahoo reports 1.2B objects within its own domain
Spam and Reputation 2

• Computing traditional PageRank for each page could be prohibitively expensive in large crawls
  – In our case, over 41B pages in the webgraph

• However, the goal here is not to classify each page, but rather to understand which domains should be allowed to massively branch

• Spam can be effectively deterred by budgeting the number of allowed pages per pay-level domain (PLD)
  – PLDs are domains that one must pay for at some registrar

• PLD reputation is determined by its in-degree from resources that spammers must pay for, which in our case are other PLDs
Spam and Reputation 3

- This solution we call Spam Tracking and Avoidance through Reputation (STAR)
- Each PLD $x$ has budget $B_x$ that represents the number of pages that are allowed to pass from $x$ (including all subdomains) to crawling threads every $T$ time units
- Each PLD $x$ starts with a default budget $B_0$, which is then dynamically adjusted as its in-degree $d_x$ changes over time
  - Diverse strategies can be achieved by varying the adjustment function (e.g., more preference for popular/unpopular domains, equal preference, linear/non-linear scaling of $B_x$)
- DRUM is used to store PLD budgets and aggregate PLD-PLD link information
Operation of STAR

crawling threads

new URLs

check + update

DRUM URLseen

check + update

unique URLs

DRUM PLDindegree

PLD links

update

STARD

URLs & budgets

robots-check queue $Q_R$

pass budget

BEAST budget enforcement
**Agenda**

- Introduction and challenges
- Background
- Overcoming the challenges
  - Scalability
  - Spam and reputation
  - Politeness
- Experiments
- Conclusion
Politeness

• Prior work has only enforced a certain per-host access delay $\tau_h$ seconds
  – Easy to crash servers that co-locate 1000s of virtual hosts
  – Thus, a per-IP limit $\tau_s$ is needed as well

• Low-ranked PLDs ($B_x = B_0$)
  – Keep $\tau_h = 40$ seconds and $\tau_s = 1$ second

• High-ranked domains ($B_x > B_0$)
  – Both $\tau_h$ and $\tau_s$ are scaled proportional to $B_x$ to crawl URLs no slower than the rate at which they are admitted into RAM

• By controlling the coupling between PLD budgets and their crawl rate, we can avoid memory backlog
Politeness 2

• To admit URLs into RAM we have a method called Budget Enforcement with Anti-Spam Tactics (BEAST)

• BEAST does not discard URLs, but rather delays their download until it knows more about the PLD they belong to

• A naïve implementation is to maintain two queues
  – $Q$ contains URLs that passed the budget check
  – $Q_F$ contains those that failed

• After $Q$ is emptied, $Q_F$ is read and again split into two queues – $Q$ and $Q_F$
Politeness 3

- **Theorem 4**: Lowest disk I/O speed (in bytes/s) that allows the naïve budget-enforcement approach to download $N$ pages at fixed rate $S$ is:

  \[ \lambda = 2Sb(L - 1)\alpha_N \]

  where

  \[ \alpha_N = \max \left( 1, \frac{N}{E[B_x]V} \right) \]

- This theorem shows that $\lambda \sim \alpha_N = \Theta(N)$

- For IRLbot, $\lambda = 3.8$ MB/s for $N = 100$ million, $\lambda = 83$ MB/s for $N = 8$ billion, and $\lambda = 826$ MB/s for $N = 80$ billion
Politeness 4

• The correct implementation of BEAST rechecks $Q_F$ at exponentially increasing intervals

• Suppose the crawler works with $j$ queues $Q_1, \ldots, Q_j$
  - Old URLs are read from $Q_1$ and sent to robots check and later to the crawling threads
  - New URLs are written to $Q_2, \ldots, Q_j$ based on their remaining budget (\(B_x\) URLs per queue)

• After $Q_1$ is emptied, the crawler moves to reading $Q_2$ and spreads new URLs between $Q_3, \ldots, Q_j, Q_1$

• After it finally empties $Q_j$, the crawler re-scans $Q_F$ and splits it into $j$ additional queues $Q_{j+1}, \ldots, Q_{2j}$
  - URLs violating the budget of $Q_{2j}$ are placed into new $Q_F$
**Politeness 5**

- **Theorem 5**: Lowest disk I/O speed (in bytes/s) that allows BEAST to download $N$ pages at fixed rate $S$ is:

  $$\lambda = 2Sb \left[ \frac{2\alpha_N}{1 + \alpha_N} (L - 1) + 1 \right] \leq 2Sb(2L - 1)$$

- For $N \to \infty$, disk speed $\lambda \to 2Sb(2L - 1) = \text{constant}$
  - It is roughly four times the speed needed to write all unique URLs to disk as they are discovered during the crawl

- For the example used in Theorem 4, BEAST requires $\lambda \leq 8.2$ MB/s regardless of crawl size $N$
Operation of BEAST

- Crawling threads
- New URLs
- DRUM URLseen
- Unique URLs
- STAR budget check
- Queue shuffler
- URLS & budgets
- Robots-check queue $Q_R$
- Pass budget
- $Q_1$, $Q_2$, ..., $Q_i$, $Q_F$
- BEAST
Agenda

• Introduction and challenges
• Background
• Overcoming the challenges
  – Scalability
  – Spam and reputation
  – Politeness
• Experiments
• Conclusion

we are here
Experiments – Summary

- One quad-CPU AMD Opteron 2.6 GHz sever with 16 GB RAM, 24-disk RAID-5, and 1-gb/s link
- Active crawling period of 41 days in summer 2007
- IRLbot attempted 7.6 billion connections and received 7.4 billion valid HTTP replies
  - 6.3 billion responses with status code 200 and content-type text/html (964M errors and redirects, 92M non-HTML)
- Average download rate 319 mb/s (1,789 pages/s)
- Crawler received 143 TB of data (254 GB of robots.txt files) and sent 1.8 TB of HTTP requests
  - After decompression, 161 TB of HTML code went through the parser
Experiments – Summary 2

Crawl rate (pages/s)

Receiving rate (mb/s)
Experiments – Summary 3

- IRLbot parsed out 394 billion links
  - Removing invalid URLs, this translates to 59 links/page

- URLseen contained 41 billion unique URLs
  - On average, 6.5 unique links per crawled page
  - Pages hosted by 641 million websites

- Discovered 89 million PLDs
  - PLD-PLD graph contained 1.8B edges

- Received responses from 117 million sites
  - Belonged to 33M PLDs
  - Were hosted on 4.2 million IPs
Experiments – Uniqueness Probability

- The probability of uniqueness $p$ stabilized around 0.11 after the first billion pages were downloaded
  - $p$ was bounded away from 0 even at $N = 6.3$ billion

- We certainly know there are $\geq 41$ billion pages
  - The fraction of them with useful content and the number of additional pages not seen by the crawler are a mystery at this stage
Experiments – Effectiveness of STAR

- Top 10K ranked domains were given budget $B_x$ linearly interpolated between 10 and 10K
  - All other PLDs received the default budget 10

- Figure shows that IRLbot succeeded at correlating PLD bandwidth allocation with their in-degree

- Manual examination reveals only 14 spam sites in the top 1000 PLDs
# Experiments – Top Ranked PLDs

<table>
<thead>
<tr>
<th>Rank</th>
<th>Domain</th>
<th>In-degree</th>
<th>PageRank</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>microsoft.com</td>
<td>2,948,085</td>
<td>9</td>
<td>37,755</td>
</tr>
<tr>
<td>2</td>
<td>google.com</td>
<td>2,224,297</td>
<td>10</td>
<td>18,878</td>
</tr>
<tr>
<td>3</td>
<td>yahoo.com</td>
<td>1,998,266</td>
<td>9</td>
<td>70,143</td>
</tr>
<tr>
<td>4</td>
<td>adobe.com</td>
<td>1,287,798</td>
<td>10</td>
<td>13,160</td>
</tr>
<tr>
<td>5</td>
<td>blogspot.com</td>
<td>1,195,991</td>
<td>9</td>
<td>347,613</td>
</tr>
<tr>
<td>7</td>
<td>wikipedia.org</td>
<td>1,032,881</td>
<td>8</td>
<td>76,322</td>
</tr>
<tr>
<td>6</td>
<td>w3.org</td>
<td>933,720</td>
<td>10</td>
<td>9,817</td>
</tr>
<tr>
<td>8</td>
<td>geocities.com</td>
<td>932,987</td>
<td>8</td>
<td>26,673</td>
</tr>
<tr>
<td>9</td>
<td>msn.com</td>
<td>804,494</td>
<td>8</td>
<td>10,802</td>
</tr>
<tr>
<td>10</td>
<td>amazon.com</td>
<td>745,763</td>
<td>9</td>
<td>13,157</td>
</tr>
</tbody>
</table>
Agenda

- Introduction and challenges
- Background
- Overcoming the challenges
  - Scalability
  - Spam and reputation
  - Politeness
- Experiments
- Conclusion
Conclusion

• This paper tackled the issue of scaling web crawlers to billions and even trillions of pages
  – Single server with constant CPU, disk, and memory speed

• We identified several bottlenecks in building an efficient large-scale crawler and presented our solution to these problems
  – Low-overhead disk-based data structures
  – Non-BFS crawling order
  – Real-time reputation to guide the crawling rate

• Future work
  – Refining reputation algorithms, accessing their performance
  – Mining the collected data