Design and Implementation of a High-Performance Distributed Web Crawler

*Vladislav Shkapenyuk** *Torsten Suel*

CIS Department
Polytechnic University
Brooklyn, NY 11201

* *Currently at AT&T Research Labs, Florham Park*

Overview:

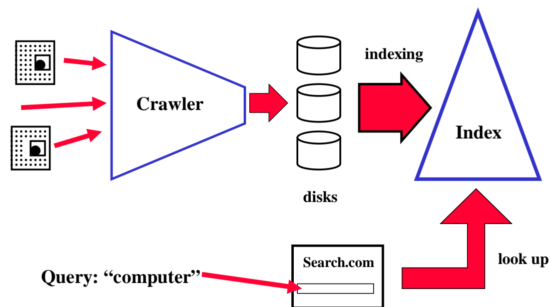
1. Introduction
2. PolyBot System Architecture
3. Data Structures and Performance
4. Experimental Results
5. Discussion and Open Problems

1. Introduction

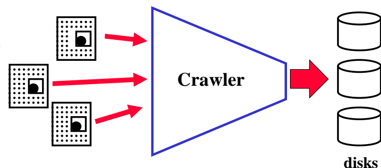
Web Crawler: *(also called spider or robot)*

- tool for data acquisition in search engines
- large engines need high-performance crawlers
- need to parallelize crawling task
- PolyBot: a parallel/distributed web crawler
- cluster vs. wide-area distributed

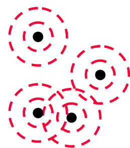
Basic structure of a search engine:



Crawler



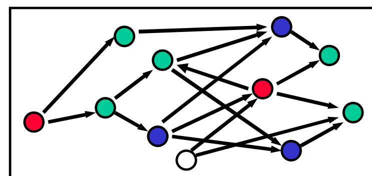
- fetches pages from the web
- starts at set of "seed pages"
- parses fetched pages for hyperlinks
- then follows those links (e.g., BFS)
- variations:
  - recrawling
  - focused crawling
  - random walks



Breadth-First Crawl:

- Basic idea:
  - start at a set of known URLs
  - explore in "concentric circles" around these URLs

start pages
distance-one pages
distance-two pages



- used by broad web search engines
- balances load between servers

Crawling Strategy and Download Rate:

- crawling strategy: “What page to download next?”
- download rate: “How many pages per second?”
- different scenarios require different strategies
- lots of recent work on crawling strategy
- little published work on optimizing download rate
  (main exception: Mercator from DEC/Compaq/HP?)
- somewhat separate issues
- building a slow crawler is (fairly) easy ...

Basic System Architecture

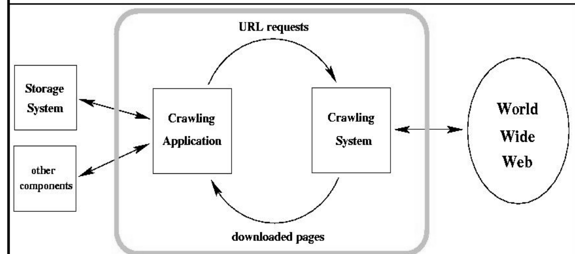


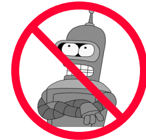
Figure 1: Basic two components of the crawler

- Application determines crawling strategy

System Requirements:

- flexibility (different crawling strategies)
- scalabilty (sustainable high performance at low cost)
- robustness
  (odd server content/behavior, crashes)
- crawling etiquette and speed control
  (robot exclusion, 30 second intervals, domain level throttling, speed control for other users)
- manageable and reconfigurable
  (interface for statistics and control, system setup)

Details: *(lots of ‘em)*



- robot exclusion
  - robots.txt file and meta tags
  - robot exclusion adds overhead
- handling filetypes
  (exclude some extensions, and use mime types)
- URL extensions and CGI scripts
  (to strip or not to strip? Ignore?)
- frames, imagemaps
- black holes (robot traps)
  (limit maximum depth of a site)
- different names for same site?
  (could check IP address, but no perfect solution)

Crawling courtesy

- minimize load on crawled server
- no more than one outstanding request per site
- better: wait 30 seconds between accesses to site
  (this number is not fixed)
- problems:
  - one server may have many sites
  - one site may be on many servers
  - 3 years to crawl a 3-million page site!
- give contact info for large crawls

Contributions:

- distributed architecture based on collection of services
  - separation of concerns
  - efficient interfaces
- I/O efficient techniques for URL handling
  - lazy URL -seen structure
  - manager data structures
- scheduling policies
  - manager scheduling and shuffling
- resulting system limited by network and parsing performane
- detailed description and how-to (limited experiments)

2. PolyBot System Architecture

Structure:

- **separation of crawling strategy and basic system**
- **collection of scalable distributed services**

  (DNS, downloading, scheduling, strategy)
- **for clusters and wide-area distributed**
- **optimized per-node performance**
- **no random disk accesses (no per-page access)**

Basic Architecture (ctd):

- application issues requests to manager
- manager does DNS and robot exclusion
- manager schedules URL on downloader
- downloader gets file and puts it on disk
- application is notified of new files
- application parses new files for hyperlinks
- application sends data to storage component (indexing done later)

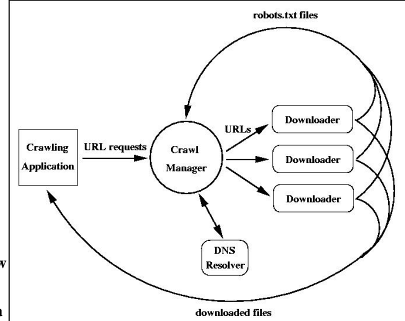


Figure 2: Small crawler configuration

System components:

- **downloader: optimized HTTP client written in Python (everything else in C++)**
- **DNS resolver: uses asynchronous DNS library**
- **manager uses Berkeley DB and STL for external and internal data structures**
- **manager does robot exclusion by generating requests to downloaders and parsing files**
- **application does parsing and handling of URLs (has this page already been downloaded?)**

Scaling the system:

- **small system on previous pages:**

  3-5 workstations and 250-400 pages/sec peak
- **can scale up by adding downloaders and DNS resolvers**
- **at 400-600 pages/sec, application becomes bottleneck**
- **at 8 downloaders manager becomes bottleneck**

  ➡ **need to replicate application and manager**
- **hash-based technique (Internet Archive crawler) partitions URLs and hosts among application parts**
- **data transfer batched and via file system (NFS)**

Scaling up:

- *20 machines*
- *1500 pages/s?*
- *depends on crawl strategy*
- *hash to nodes based on site (b/c robot ex)*

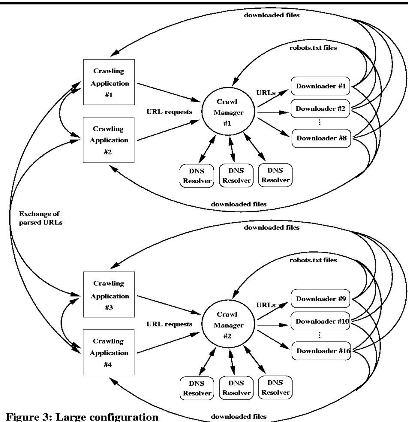


Figure 3: Large configuration

3. Data Structures and Techniques

Crawling Application

- **parsing using *pcre* library**
- **NFS eventually bottleneck**
- **URL-seen problem:**
  - need to check if file has been parsed or downloaded before
  - after 20 million pages, we have "seen" over 100 million URLs
  - each URL is 50 to 75 bytes on average
- **Options: compress URLs in main memory, or use disk**
  - prefix+huffman coding (DEC, JY01) or Bloom Filter (Archive)
  - disk access with caching (Mercator)
  - we use lazy/bulk operations on disk

• Implementation of URL-seen check:
- while less than a few million URLs seen, keep in main memory
- then write URLs to file in alphabetic, prefix-compressed order
- collect new URLs in memory and periodically reform bulk check by merging new URLs into the file on disk

• When is a newly a parsed URL downloaded?

• Reordering request stream
- want to space ot requests from same subdomain
- needed due to load on small domains and due to security tools
- sort URLs with hostname reversed (e.g., com.amazon.www), and then "unshuffle" the stream ➡ provable load balance

Crawling Manager

• large stream of incoming URL request files

• goal: schedule URLs roughly in the order that they come, while observing time-out rule (30 seconds) and maintaining high speed

• must do DNS and robot excl. "right before"download

• keep requests on disk as long as possible!
- otherwise, structures grow too large after few million pages (performance killer)

Manager Data Structures:

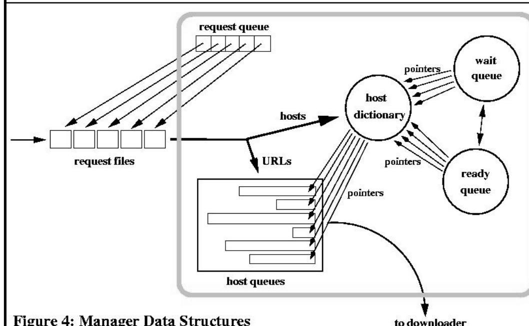


Figure 4: Manager Data Structures

• when to insert new URLs into internal structures?

URL Loading Policy

• read new request file from disk whenever less than x hosts in ready queue

• choose x > speed * timeout *(e.g., 100 pages/s * 30s)*

• # of current host data structures is
x + speed * timeout + n_down + n_transit
which is usually < 2x

• nice behavior for BDB caching policy

• performs reordering only when necessary!

4. Experimental Results

• crawl of 120 million pages over 19 days
161 million HTTP request
16 million robots.txt requests
138 million successful non-robots requests
17 million HTTP errors (401, 403, 404 etc)
121 million pages retrieved

• slow during day, fast at night

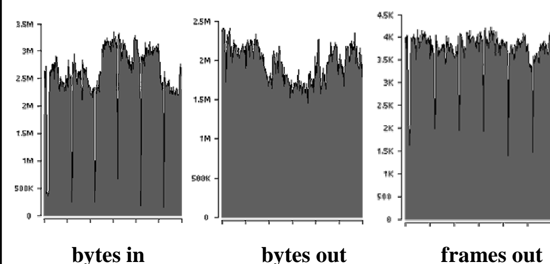
• peak about 300 pages/s over T3

• many downtimes due to attacks, crashes, revisions

• "slow tail" of requests at the end (4 days)

• lots of things happen

Experimental Results ctd.



bytes in bytes out frames out

Poly T3 connection over 24 hours of 5/28/01
(courtesy of AppliedTheory)

Experimental Results ctd.

- sustaining performance:
  - will find out when data structures hit disk
  - I/O-efficiency vital
- speed control tricky
  - vary number of connections based on feedback
  - also upper bound on connections
  - complicated interactions in system
  - not clear what we should want
- other configuration: 140 pages/sec sustained on 2 Ultra10 with 60GB EIDE and 1GB/768MB
- similar for Linux on Intel

More Detailed Evaluation (to be done)

- Problems
  - cannot get commercial crawlers
  - need simulation systen to find system bottlenecks
  - often not much of a tradeoff (get it right!)
- Example: manager data structures
  - with our loading policy, manager can feed several downloaders
  - naïve policy: disk access per page
- parallel communication overhead
  - low for limited number of nodes (URL exchange)
  - wide-area distributed: where do yu want the data?
  - more relevant for highly distributed systems

5. Discussion and Open Problems

Related work

- Mercator (Heydon/Najork from DEC/Compaq)
  - used in altaVista
  - centralized system (2-CPU Alpha with RAID disks)
  - URL-seen test by fast disk access and caching
  - one thread per HTTP connection
  - completely in Java, with pluggable components
- Atrax: very recent distributed extension to Mercator
  - combines several Mercators
  - URL hashing, and off-line URL check (as we do)

Related work (ctd.)

- early Internet Archive crawler (circa 96)
  - uses hashing to partition URLs between crawlers
  - bloom filter for "URL seen" structure
- early Google crawler (1998)
- P2P crawlers (grub.org and others)
- Cho/Garcia-Molina (WWW 2002)
  - study of overhead/quality tradeoff in parallel crawlers
  - difference: we scale services separately, and focus on single-node performance
  - in our experience, parallel overhead low

Open Problems:

- Measuring and tuning peak performance
  - need simulation environment
  - eventually reduces to parsing and network
  - to be improved: space, fault-tolerance (Xactions?)
- Highly Distributed crawling
  - highly distributed (e.g., grub.org) ? (maybe)
  - bybrid? (different services)
  - few high-performance sites? (several Universities)
- Recrawling and focused crawling strategies
  - what strategies?
  - how to express?
  - how to implement?