High–Performance Crawling: The State Of BUbiNG

Sebastiano Vigna
(with Paolo Boldi, Andrea Marino and Massimo Santini)

Why a new crawler?

- Not so many open-source crawlers
- Not so configurable
- Not so extensible
- Not distributed
- NIH

Previous work

- Mercator (Najork *et al.*)
- UbiCrawler (Boldi *et al.*)
- IRLBot (WWW 2008)
- Heritrix (Internet Archive)
- Nutch (based on Hadoop)
- Bixo (based on Hadoop)
- Surprisingly little performance data

Challenges

- Use massive memory and multiple cores efficiently (does not work on a mobile phone)
- Fill bandwidth in spite of politeness (both at host and IP level)
- Stoppable/restartable
- Completely configurable
- Extensible will little effort (no recompilation)

Crawler Behaviour

- Simple text key/value file
- By design, all properties must be specified (code is not responsible for defaults)
- For instance: `maxUrls=500M`
- Or: `urlCacheSize=128Mi`
- Time units, SI multipliers, NIST multipliers

Crawling Phases

- Totally generic approach
- Each phase (schedule, fetch, parse, follow, store) has an associated *filter*
- Filters can be specified using Boolean formulae (with short-circuit semantics)
- Atoms are Java classes instantiated by reflection using a natural syntax
- Atoms can be applied to URLs or responses (adaptation is automatic)

Crawling Phases

- Many useful ready-made atoms
- From easy ones: `HostEndsWith`
- To extremely sophisticated ones: `DuplicateSegmentsLessThan` finds URLs with repeated segments like `/a/a/a` or `/a/b/a/b/a/b` using suffix arrays (10x faster than regular expressions)
- To content-based: `IsProbablyBinary()`

Typical cases

- Parsing:
  `( ContentTypeStartsWith(text/) or PathEndsWithOneOf(.html,.htm,.txt) ) and not IsProbablyBinary()`
- Scheduling: `(SchemeEquals(http) or SchemeEquals(https)) and HostEndsWith(.it) and not PathEndsWithOneOf(.axd,.xls,.rar,.... )`

The Workbench

- Crawling happens by picking elements from the *workbench*
- First, each host (and related state) is stored in a *visit state*, which contains a FIFO queue of URLs to be visited
- Each visit state has a next-fetch time that is the first instant in time in which it is possible (by politeness) to fetch a URL from the host

Entries

- Visit states are grouped by IP address in *workbench entries*
- Each entry contains a queue of visit states prioritized by next-fetch
- Moreover every entry has a next-fetch that is the first instant in time in which it is possible (by politeness) to fetch a URL from the IP address

Priority of entries

- Each entry is stored in the *workbench*, which is a queue of entries prioritized by the maximum between the next-fetch of the entry and the minimum next-fetch of associated hosts
- Thus, if there is an entry with a ready visit state, there is an entry with a ready visit state at the top of the workbench

High Parallelism

- We use massively multiple (like 1000) threads
- Every thread handles a request and is I/O bound
- Parallel threads parse and store pages
- Slow data structures are sandwiched between *wait-free* queues

Handling Queues

- The workbench is actually an abstraction
- The FIFO queues of URLs grow exponentially
- They must be stored partially on disk
- The goal is to maintain wide the *front* of the crawl
- We set a required front and increase it each time a fetching thread waits

The Sieve

- The *sieve* is the basic data structure behind the crawl
- Is a FIFO queue partially stored on disk from which elements are dequeued just once
- We use an implementation similar to that of Mercator
- Alternatives such as DRUM do not preserve the breadth-first visit order
- Common mistake: Bloom filters

Fully Distributed

- We use JGroups to set up a view on a set of agents
- Hosts are assigned to agent using consistent hashing
- URLs for which an agent is not responsible are quickly delivered to the right agent
- We use JAI4J, a thin layer over JGroups that handles job assignment.

Near-Duplicates

- We detect (presently) near-duplicates using a MurmurHash3 fingerprint of a stripped page (stored in a Bloom filter)
- The stripping includes eliminating almost all tag attributes and numbers from text
- We are going to experiment with more sophisticated methods like SimHash
- Suggestions for heuristics are welcome
- We would like to have a test collection

Parsing

- You can specify any number of parsers
- Every parsers implements `Filter`, and the first parser that accept a response will parse it
- Fallback binary parser
- Presently the HTML parser just parses text
- In the future: JavaScript partial evaluation

Storing

- You can specify in the configuration file a `Store` implementation
- The basic one generates a compressed Warc file (with parallel compression)
- We plan on having one storing into HBase
- Data can even be just streamed somewhere else
- The implementation knows whether we estimate that the page is a duplicate

Future

- Reduce object creation (too much garbage collection)
- Experiment with HTTP KeepAlive
- Choice between synchronous and asynchronous Apache HTTP clients
- Do first real-world large crawls
- Distribute the crawler to selected location for testing before public release
- Write a manual!