Distributed Web Crawling with AWS and Python

Steve Howard
Thumbtack, Inc

Outline

- Architecture of a high-performance web crawler
- Distributing the crawler
- Implementing with AWS and Python

September 26, 2001

SRC Research Report

173

High-Performance Web Crawling

Marc Najork
Allan Heydon

COMPAQ

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.compaq.com/SRC/

Mercator

- Marc Najork and Allan Heydon. 2001. *High-Performance Web Crawling*. Technical Report 173, Compaq Systems Research Center.
  - http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-173.pdf
- Also based on description of Mercator in *Introduction to Information Retrieval* by Manning et al., 2008.
  - http://www.amazon.com/Introduction-Information-Retrieval-Christopher-Manning/dp/0521865719

Goals

- **Scalable**
- Efficient
- Polite
- Robust

http://thesisready.com/tutorial/scalable-content-for-wordpress-and-you/

Goals

- Scalable
- **Efficient**
- Polite
- Robust

http://passionategreen.net/2011/07/shift-to-led-lights

Goals

- Scalable
- Efficient
- **Polite**
- Robust

http://www.jaunted.com/story/2006/10/6/45346/1566/travel/Polite+Destinations

Goals

- Scalable
- Efficient
- Polite
- **Robust**

http://onlyrealhiphop.blogspot.com/2008/05/robust-el-foto-grande-2007.html

Web crawler architecture

- URL Frontier
- Fetching
  - DNS cache, robots.txt cache, HTTP fetcher
- Processing
  - Link extraction
  - Parsing, indexing, searching, stats, etc.
- Duplicate URL Eliminator

URL Frontier

- At a high level, just a big queue
- Complicated by three factors
  - Scale: 100MMs of URLs = GBs of data
  - Politeness: don't overload any single host
    - "Weak politeness" and "strong politeness"
  - Priority: some fetches may be higher priority than others

Mercator URL Frontier

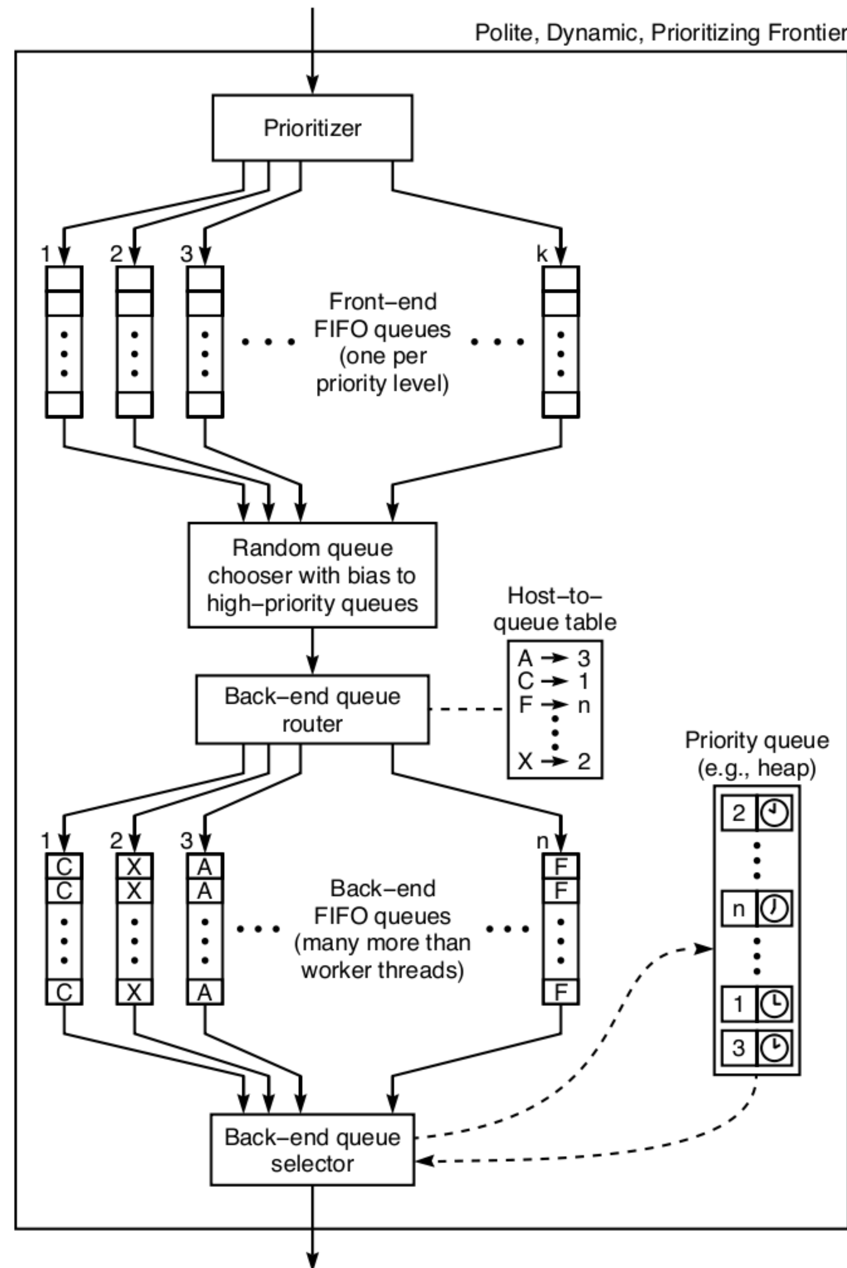


Figure 3: Our best URL frontier implementation

(from Najork 2001)

URL Frontier code

```
def ready_sites(self):
  while (self._host_heapq
        and self._host_heapq[0][0] <= time_now):
    can_crawl_at_time, host = heapq.heappop(self._host_heapq)
    queue = self._host_queue_assignments[host]
    self._num_enqueued -= 1
    yield queue.get()

def fill_host_queue(self, queue):
  while self._front_queue:
    fetch_task = self._front_queue.get()
    host = self._get_host(fetch_task.site)
    if host in self._host_queue_assignments:
      self._host_queue_assignments[host].put(fetch_task)
    else:
      self._unassigned_queues.remove(queue)
      self._host_queue_assignments[host] = queue
      queue.put(fetch_task)
      self._add_to_heap(host)
      return
```

URL Frontier: Disk-based Queues

- Split the contents into blocks
- Keep the head and tail blocks in memory, rest on disk
- When tail fills: flush to disk
- When head empties: load next block from disk
- I used http://code.activestate.com/recipes/501154-persistent-queue/

Fetching

- DNS cache: simple LRU + obey TTL
  - Choose randomly from multiple addresses
- robots.txt cache: SQLite DB with 24-hour expiration
  - Treat 5xx as blocked
- HTTP fetching: mostly straightforward
  - Limit download size
  - Avoid redirect loops and be polite

Processing

- Parse HTML
- Extract HREFs
- Domain-specific work -- generate output
- This is the most CPU-intensive piece

http://www.cnmeonline.com/

Duplicate URL Eliminator

- At a high level, just a big set of strings
- Challenge: scale
  - Billions of URLs
  - 1000s of QPS

Mercator DUE

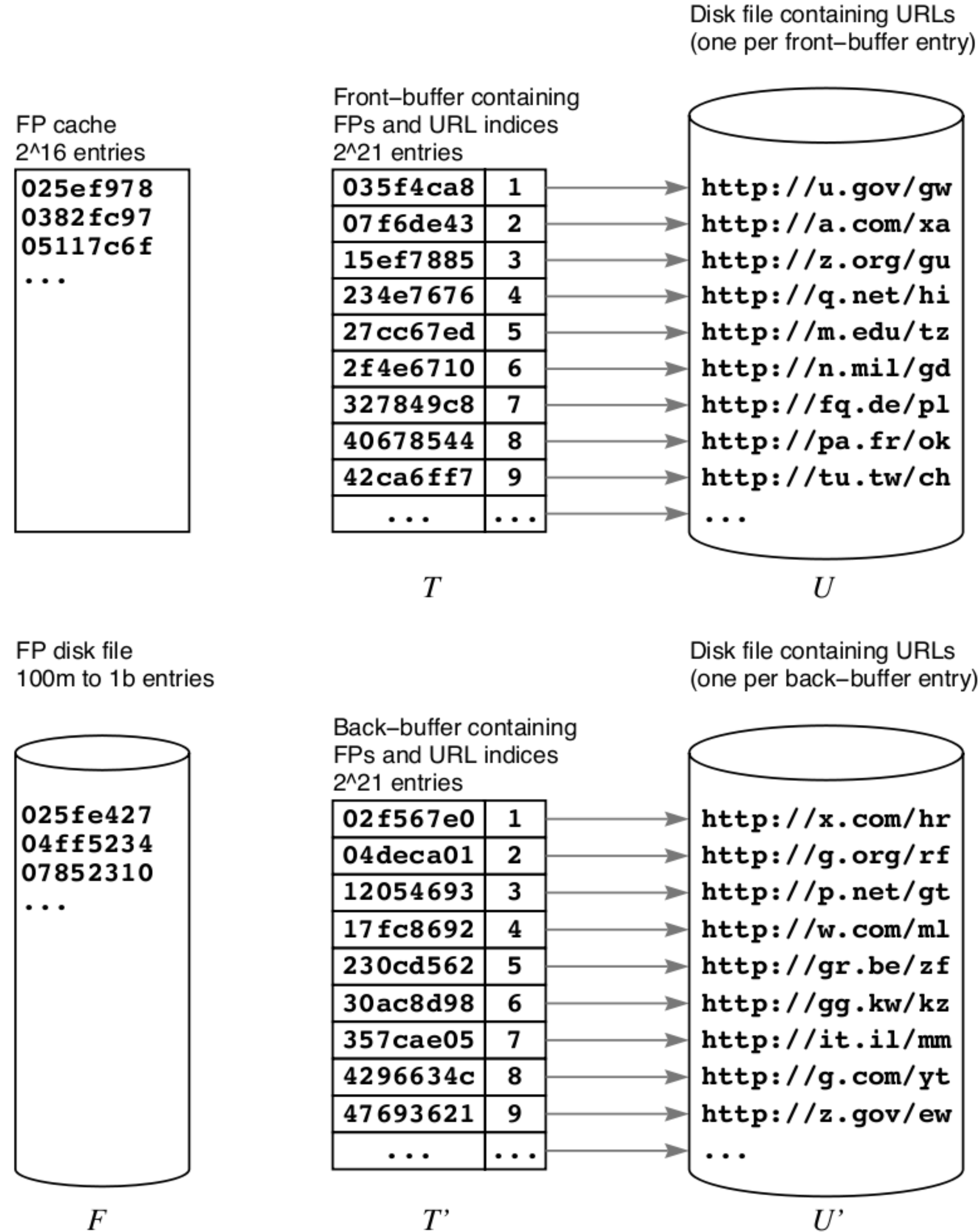


Figure 4: Our most efficient disk-based DUE implementation

(from Najork 2001)

DUE code

```
def put(self, url):
  hash_value = hash(url)
  if hash_value in self._lru_cache:
    return False

  self._lru_cache.add(hash_value)
  with self._front_buffer_lock:
    if hash_value in self._front_buffer:
      return False
    self._front_file.write(url + '\n')
    self._front_buffer[hash_value] = len(self._front_buffer)
  return True
```

DUE code

```
def get_new_urls(self):
  with self._front_buffer_lock:
    back_buffer = self._copy_front_buffer()
    back_file = self._fs.rename(self._FRONT_NAME, self._BACK_NAME)
    self._front_file = self._fs.open(self._FRONT_NAME)
    self._front_buffer = {}

  back_buffer.sort(key=operator.attrgetter('hash_value'))
  with self._fs.open(self._NEW_URL_HASHES) as new_hashes_stream:
    with self._fs.open(self._URL_HASHES) as old_hashes_stream:
      for hash_value, hash_from_file, buffer_entry in merge(
          old_hashes, back_buffer):
        new_hashes.write(hash_value)
        if hash_from_file is None:
          buffer_entry.is_new = True

  self._fs.rename(self._NEW_URL_HASHES, self._URL_HASHES)

  back_buffer.sort(key=operator.attrgetter('ordinal'))
  for entry, url in itertools.izip(back_buffer, back_file):
    if entry.is_new:
      self.new_urls.append(url)
```

Distributing Mercator

- Shard URLs by domain name
- Each host is completely self-contained *except* for cross-domain links
- After link extraction:
  - Cross-domain links get sent off to the appropriate hosts (in batches)
  - Hosts receiving links insert them directly into the DUE
  - 80% of links are relative according to Najork 2001
    - Observed ~10 links sent per page

Distributing Mercator

- Scaling up and down freely
  - Choose number of shards much greater than maximum number of hosts expected
  - Each shard has
    - DUE fingerprint file
    - URL frontier snapshot
  - Assign shards over hosts at startup
    - No load balancing right now

Mercator in Python

- Original Mercator uses threads; in Python we must use processes
  - `multiprocessing` is your friend, especially `Queue`
    - But beware the feeder thread! For our purposes, `Connection` + `Lock` does fine.
- Isolated processes for each major component
  - Fetching (uses threads)
  - Processing (actually a pool)
  - DUE flushing
  - Checkpointing
- One master process coordinates everything asynchronously
  - Has input/output queue pair for each subprocess (or subprocess pool)
  - `select.select` is your friend
  - Very few concurrency worries
- Don't rely on COW in Python

Mercator on AWS

- EC2 lets us scale out flexibly
  - Probably want at least an m1.large instance, depends on processing tasks
  - Use instance ephemeral storage for DUEs
  - Spot instances are OK
- S3 stores snapshots and crawl results
  - DUE snapshots and crawl results can be uploaded asynchronously
  - URL frontier snapshots are currently uploaded synchronously
  - At startup each instance downloads + reads snapshots
- SQS holds seed URLs
  - One queue for each domain shard
  - Script puts seed URLs into the appropriate queues
  - Crawler polls SQS for new seeds as needed and adds them to DUE
  - Cross-domain links can be sent to appropriate SQS queues

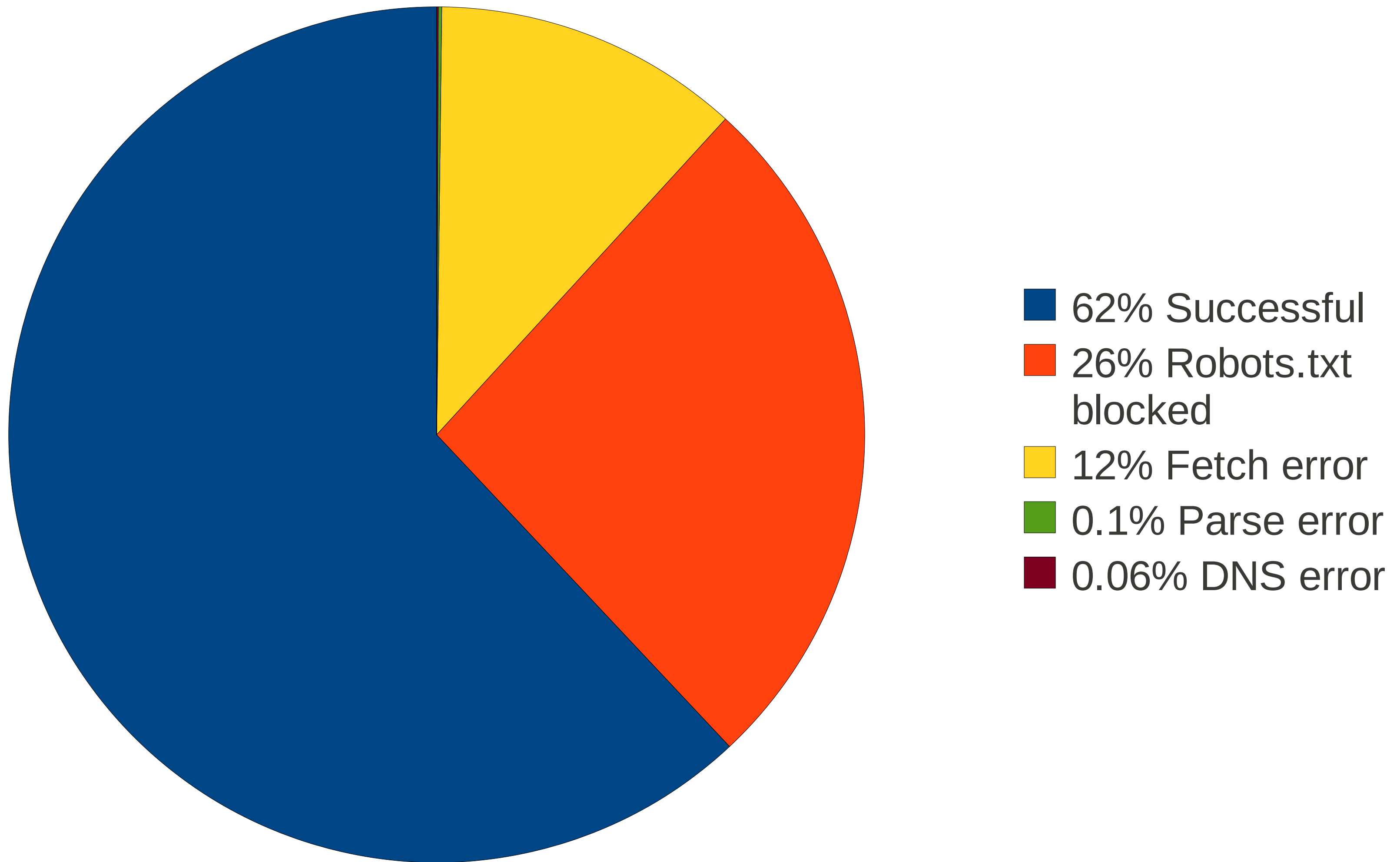
Performance

- For each instance
  - ~100 URLs processed per second = ~8.6MM/day, 250MM/month
  - Billions of URLs in DUE
  - ~1.4MM URLs/$ with c1.xlarge instances
- Have run up to 64 instances without problems

Example crawl

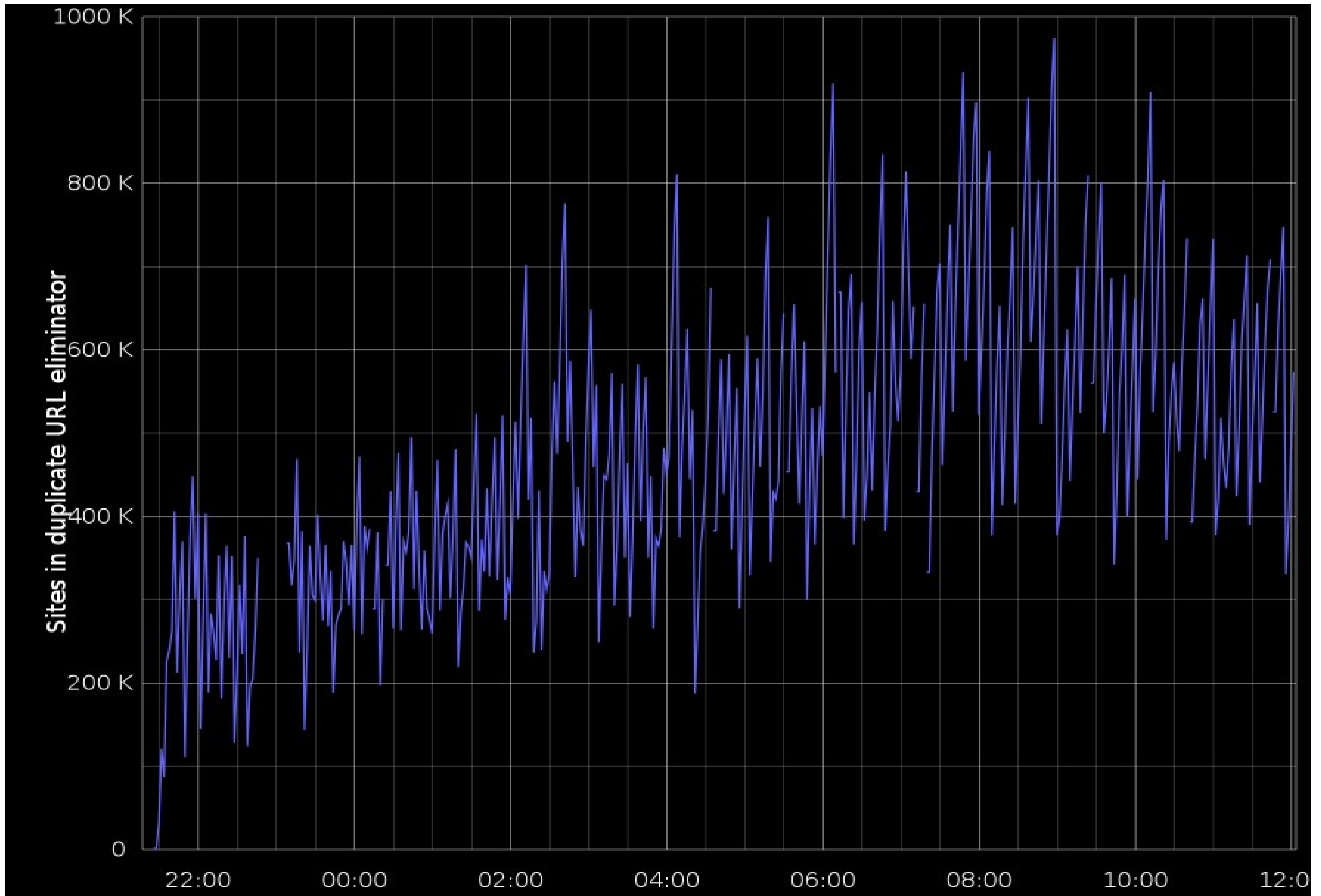
- Start with `local.yahoo.com`
- Four machines over ~16 hours
- 17MM URLs processed
  - 12MM fetched
  - 93MM unique URLs seen
  - Average ~8 new links per page

Crawl data

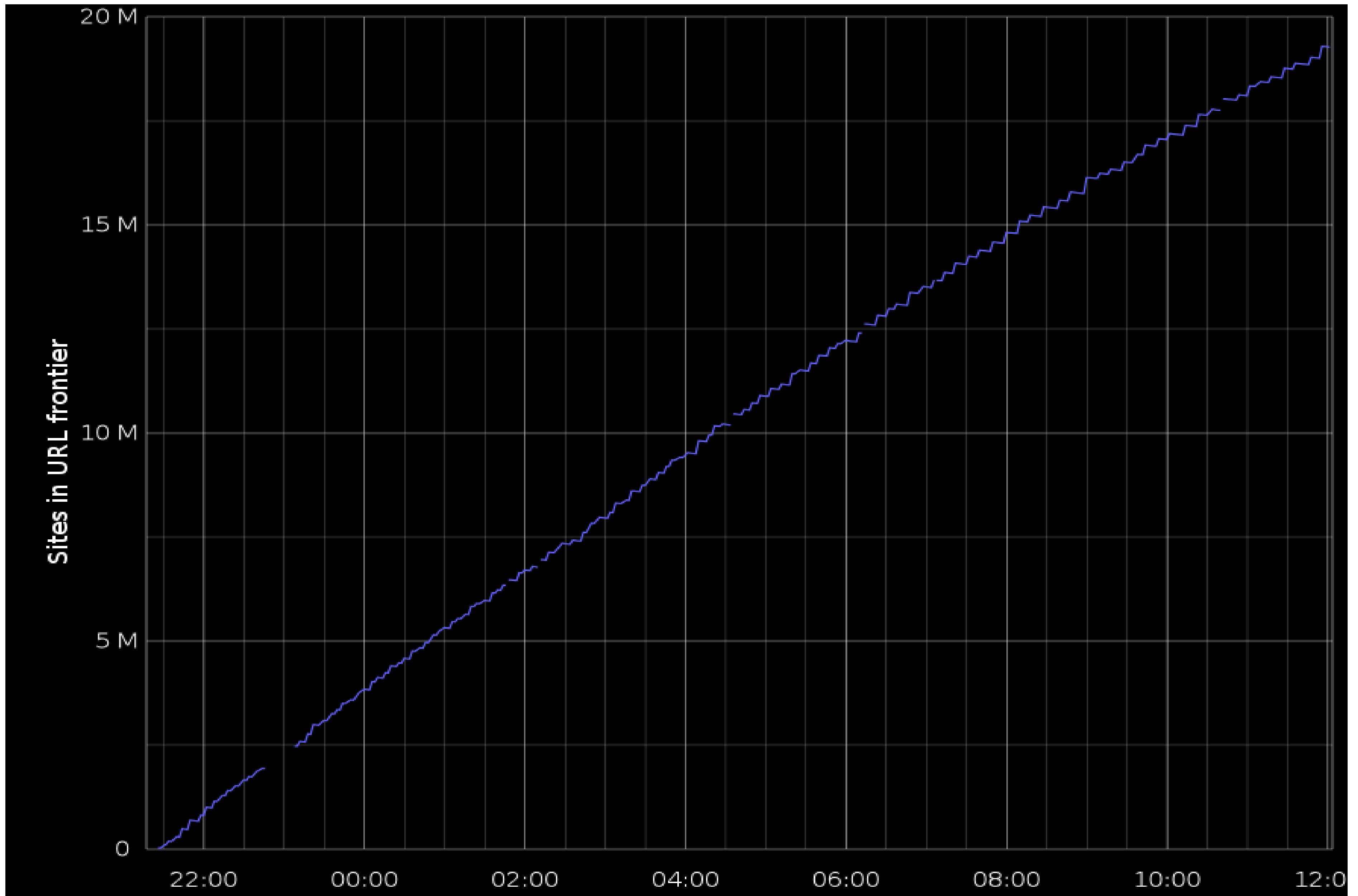


Graphs!

Graphs!



Graphs!



Thumbtack is hiring!

- Come talk to us!
- thumbtack.com/jobs
- steve@thumbtack.com

Questions?