THE ARCHITECTURE AND IMPLEMENTATION OF AN EXTENSIBLE WEB CRAWLER

Jonathan Hsieh
Steve Gribble
Hank Levy
University of Washington
NSDI ’10, San Jose, CA
MOTIVATION

- The web is an ever-changing, interesting, and incredibly massive database of information
  + Google, 7/25/08: 1 trillion unique URLs in index
- There are many **crawler applications** that scour the web to harvest data
TWO CATEGORIES OF CRAWLER APPS

 Crawl the entire web and use **all** of the content

 Crawl the entire web and use **a small subset** of the content

- Google
- Bing
- McAfee
- Reputation Defender
- Attributor
“NEEDLE IN A HAYSTACK” CRAWLER APPS

- Crawler Applications do two tasks:
  - Crawl the entire web
  - Application specific work
- Crawling at web scale is hard
  - Expensive
  - Operationally difficult
  - Discards most documents
Decouple the difficult crawling tasks from the application-specific tasks
Share the crawler as a common service
Still need to deliver the deluge
**KEY INSIGHT**

- Make filtering a shared resource
- Only a small trickle of documents now!
Client uses filter language to inject filters
The crawler harvests webpages and dispatches documents
A filter engine evaluates documents
Document matches are collected by crawler apps
Client uses filter language to inject filters
The crawler harvests webpages and dispatches documents
A filter engine evaluates documents
Document matches are collected by crawler apps
ARCHITECTURAL GOALS

- The extensible crawler is a **service** that must be:
  - **Flexible**
    - Support a diverse set of crawler applications
    - Expressive filter language for complex web data
  - **Scalable**
    - large filter sets (10’s millions-billions)
      - efficient filter execution
    - high document throughput (100k docs/s)
      - commodity cluster architecture
  - **Low Latency**
    - support real-time applications
SEARCH ENGINE VS EXTENSIBLE CRAWLER

× Search engine
  + Millions of humans constantly enter one query at a time
    × Queries are keywords
    × Query latency important
    × Return only the top-ranked subset of matches
  + Process a stream of queries against a document index

× Extensible crawler
  + Hundreds of programs periodically enter millions of filters
    × Filters are conjuncts of expressions.
    × Doc latency important
    × Returns all matches
  + Process a stream of documents against a filter index
Design Tradeoffs of Filter Language

Efficient Filter Evaluation
Design Tradeoffs of Filter Language
Efficient Filter Evaluation
Design Tradeoffs of Filter Language
Efficient Filter Evaluation
Achieving Scale with Commodity Clusters
FILTER LANGUAGE

- The filter language needs to be expressive
  - Support a wide variety of apps
  - Web data is complex, largely unstructured
- Examples:
  - `substring("Jonathan Hsieh")`
  - `regex("Jonathan.{1,20}Hsieh")`
  - `substring("Jonathan") AND substring("Hsieh")`
FILTER ENGINE TRANSFORMS AND EXECUTES FILTERS

- Efficient
  + indexing and evaluation

- Expressive
  + support complex data and diverse apps

- Accurate
  + we promise 100% recall
  + we permit false positives (less than 100% precision) to gain efficiency
inject filters
for \( D = \) next document
   for each \( F \) in set of filters
      if \( F \) accepts \( D \)
         forward to collector
      else
         drop

- One pass per document per filter
  + Work = \# documents * \# filters
- Not cost efficient
Indexing filters.
  + Trade memory for CPU
  + Execute all filters simultaneously for less than linear cost.
  + Compile cost is amortized because filters change infrequently

Single pass per document
**Example: Indexing**

- Execution of many substrings
  - One pass per filter
- Execution of Aho-Corasick DFA in one pass
  - One pass for all filters

As banks de...
Air travelers ...

compile into Aho Corasick DFA
Document partitioning

- Every document must be evaluated by a pod
- Pods are independent
- Document workload is embarrassingly parallel

Filter set partitioning

- Every document must evaluated by every machine in a pod
- Constrained by slowest node in a pod
IMPLEMENTATION AND EVALUATION

- Worker execution optimization
  - Relaxing and Staging filters

- Pod filter partitioning strategies
  - Random vs Sorted

- Prototype crawler applications
RELAXING FILTERS

- Indexing is not always efficient
- **Relax** filters to a less precise version
  - False positives now possible
  - Trade accuracy for reduced resource requirements

**substring**(“General Motors said on Wednesday”) that it had a positive cash flow of $1 billion in the six months after emerging from bankruptcy protection”.

universe of all possible documents

relaxed matches

exact matches
Relaxing introduces false positives

- A relaxed filter may accept too many documents

Solution: Optional second phase called staging

- If a relaxed filter matches in first stage, only execute its full filter in second stage
- Clean up false positives if cheap enough
EXAMPLE: RELAXING FILTERS

```javascript
regex('`<script language="javascript"> eval (unescape("%66%75%6e%63%74%69%6f%6e%20%. {4}%28%.{4}%29%76%61%72%20`)
```
EXAMPLE: RELAXING FILTERS

`substring('<script language="javascript"> eval(unescape("%66%75%6e%63%74%69%6f%6e %20%'))
AND substring('%28%')
AND substring('%29%7b%76%61%72%20')`

- Relaxing a malware regular expression
  + Relax regex into a conjunct of substrings
EXAMPLE: RELAXING FILTERS

```javascript
substring('<script language="javascript">eval(unescape("%66%75%6e%63%74%69%6f%6e%20")
AND substring('%28%')
AND substring('%29%7b%76%61%72%20')
```
substring(`<script language="javascript">`)
eval(unescape("%66%75%6e%63%74%69%6f%6e
%20%’))

- Relaxing a malware regular expression
  - Relax regex into a conjunct of substrings
  - Relax conjunct into a single term
substring(‘<script language="javascript"> eval(unescape("%66%75%6e%63%74%69%6f%6e %20%’))

- Relaxing a malware regular expression
  - Relax regex into a conjunct of substrings
  - Relax conjunct into a single term
  - Relax long substring into short substring
EXAMPLE: RELAXING FILTERS

```javascript
substring('&lt;script language=&quot;javascript&quot;&gt;
  eval(unescape('%66%75%6e%63%74%69%6f%6e%20%27'))
```

- Relaxing a malware regular expression
  - Relax regex into a conjunct of substrings
  - Relax conjunct into a single term
  - Relax long substring into short substring
EXAMPLE: RELAXING FILTERS

```
substring('<script language="javascript">
eval(unescape('%66%75%6e%63%74%69%6f%6e
%20%'))--
```

- Relaxing a malware regular expression
  - Relax regex into a conjunct of substrings
  - Relax conjunct into a single term
  - Relax long substring into short substring
  - Select relaxations carefully!
EXAMPLE: RELAXING FILTERS

```javascript
substring('`<script language="javascript"> eval(unescape('%66%75%6e%63%74%69%6f%6e %20%'))
```

- Relaxing a malware regular expression
  - Relax regex into a conjunct of substrings
  - Relax conjunct into a single term
  - Relax long substring into short substring
  - Select relaxations carefully!
Relaxing a malware regular expression

- Relax regex into a conjunct of substrings
- Relax conjunct into a single term
- Relax long substring into short substring
- Select relaxations carefully!

Example: Relaxing Filters

```
substrings('75%6e%63%74%69%6f%6e%20%')
```
- Naïve filter execution is not cost effective
- **Index** filters to use memory instead of CPU
  - Each machine does more work
- Indexing is very memory intensive.
- Relax filters for less memory consumption
  - Order of magnitude less memory used
  - Order of magnitude more filters on a worker
FILTER SET PARTITIONING

- Indexes for large filter sets are too big for a single machine
  - Partition filters and build indexes on subsets

- Different strategies affect pod performance
  - Random: cheap and quick
  - Sorted: sharing efficiencies

random partitioning
- As banks deal with fraud...
- Defense Secretary Robert...

sorted partitioning (alpha)
- Democrats are embolden..
- Air travelers stranded in..
- As banks deal with fraud...
- Defense Secretary Robert..
- Democrats are embolden..
Random filter partitioning has low throughput variance

Sorted partitioning (alphabetizing) improves most nodes’ throughput, but has high variance.

Compensate for variance by blacklisting troublesome filters

Most machines not at capacity!

max alpha + blacklist throughput

max alpha throughput

max random throughput
PROTOTYPE CRAWLER APPLICATIONS

- Copyright Violation/Plagiarism
  + Sentences from Wikipedia, AP, and Reuters articles

- Web Malware Detection
  + Regexes from ClamAV web malware signatures

- Vanity/Online Identity Service
  + Regexes generated from names in a university directory
Applications tested against 3.68M web documents
Gathered by Nutch 0.9 crawler and seeded by DMOZ

<table>
<thead>
<tr>
<th># filters</th>
<th>Copyright</th>
<th>Malware</th>
<th>Identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relaxed-only</td>
<td>251,657</td>
<td>3,128</td>
<td>10,622</td>
</tr>
<tr>
<td>Throughput (docs/s)</td>
<td>8,535</td>
<td>8,534</td>
<td>7,244</td>
</tr>
<tr>
<td>Doc Hit Rate</td>
<td>0.664%</td>
<td>45.4%</td>
<td>69.0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relax+staged</th>
<th>Copyright</th>
<th>Malware</th>
<th>Identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput (docs/s)</td>
<td>8,229</td>
<td>6,354</td>
<td>592</td>
</tr>
<tr>
<td>Doc Hit Rate</td>
<td>0.016%</td>
<td>0.009%</td>
<td>13.1%</td>
</tr>
<tr>
<td># machines for 100k docs/s</td>
<td>12.2</td>
<td>15.7</td>
<td>169</td>
</tr>
</tbody>
</table>
Motivation
Architecture
Implementation and Evaluation

CONCLUSION
CONCLUSIONS

- We introduced the **service**, the architecture, and the implementation of the **extensible crawler**
  + **Flexible filter language** for efficiently filtering complex web data
  + **Scalable and cost-efficient** on commodity clusters architecture
  + **Low latency** to support real-time web applications