THE ARCHITECTURE AND IMPLEMENTATION OF AN EXTENSIBLE WEB CRAWLER

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

MOTIVATION

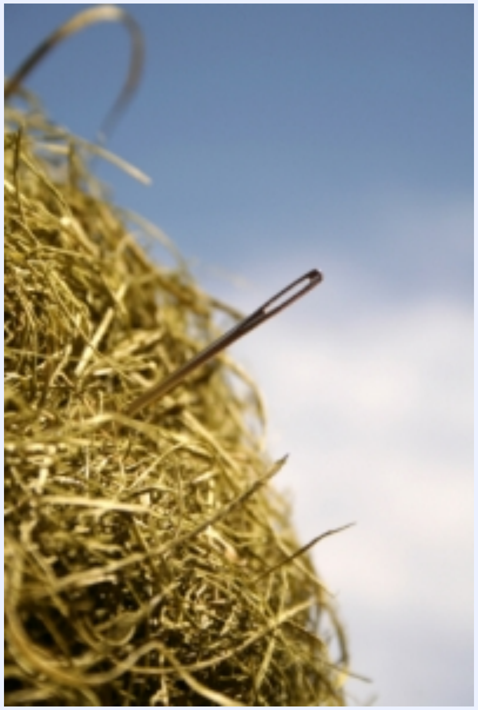
- The web is an ever-changing, interesting, and incredibly massive database of information
  - Google,7/25/08: 1 trillion unique URLS in index
- There are many crawler applications that scour the web to harvest data

TWO CATEGORIES OF CRAWLER APPS

- Crawl the entire web and use <u>all</u> of the content
- Crawl the entire web and use <u>a small subset</u> of the content

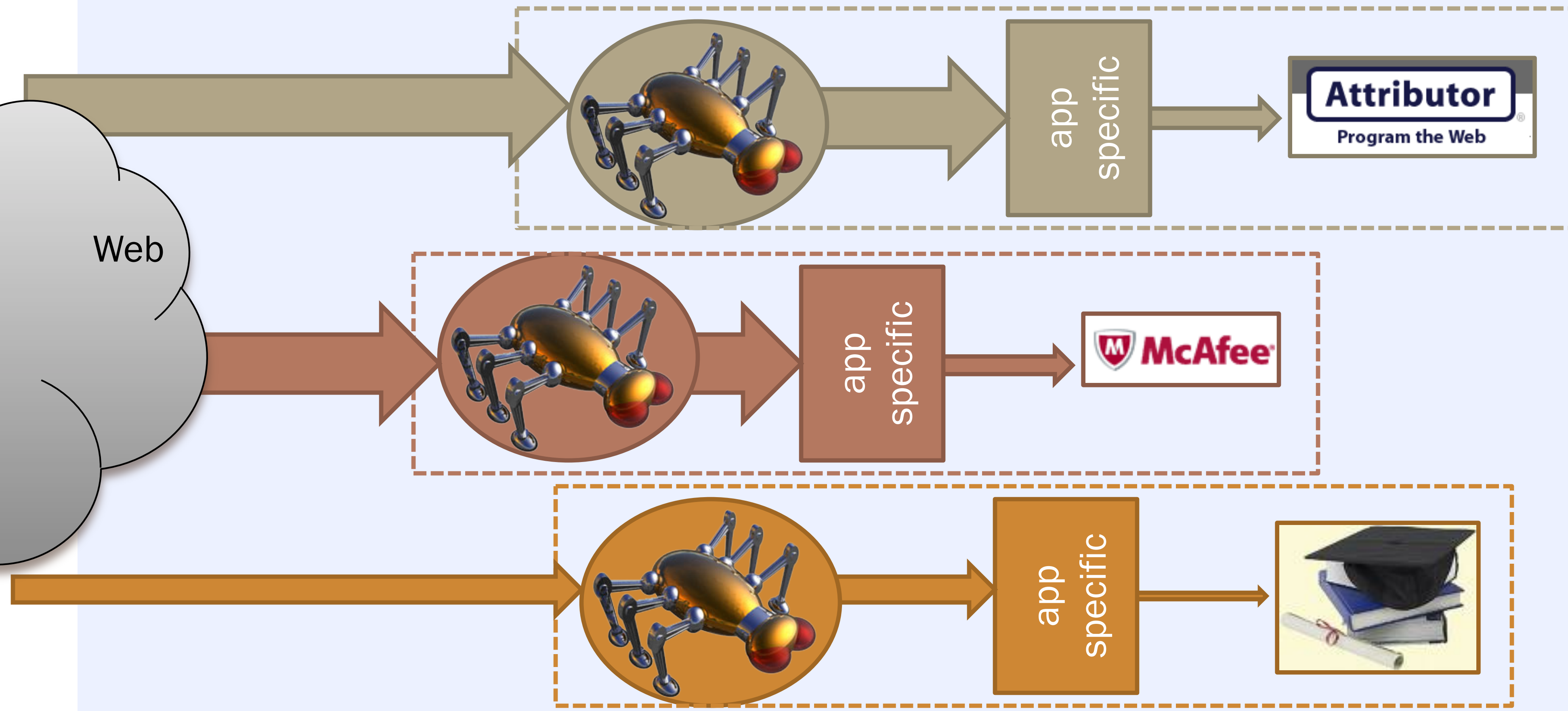
"NEEDLE IN A HAYSTACK" CRAWLER APPS

- Crawler Applications do two tasks:
  - Crawl the entire web
  - Application specific work
- Crawling at web scale is hard
  - Expensive
  - Operationally difficult
  - Discards most documents



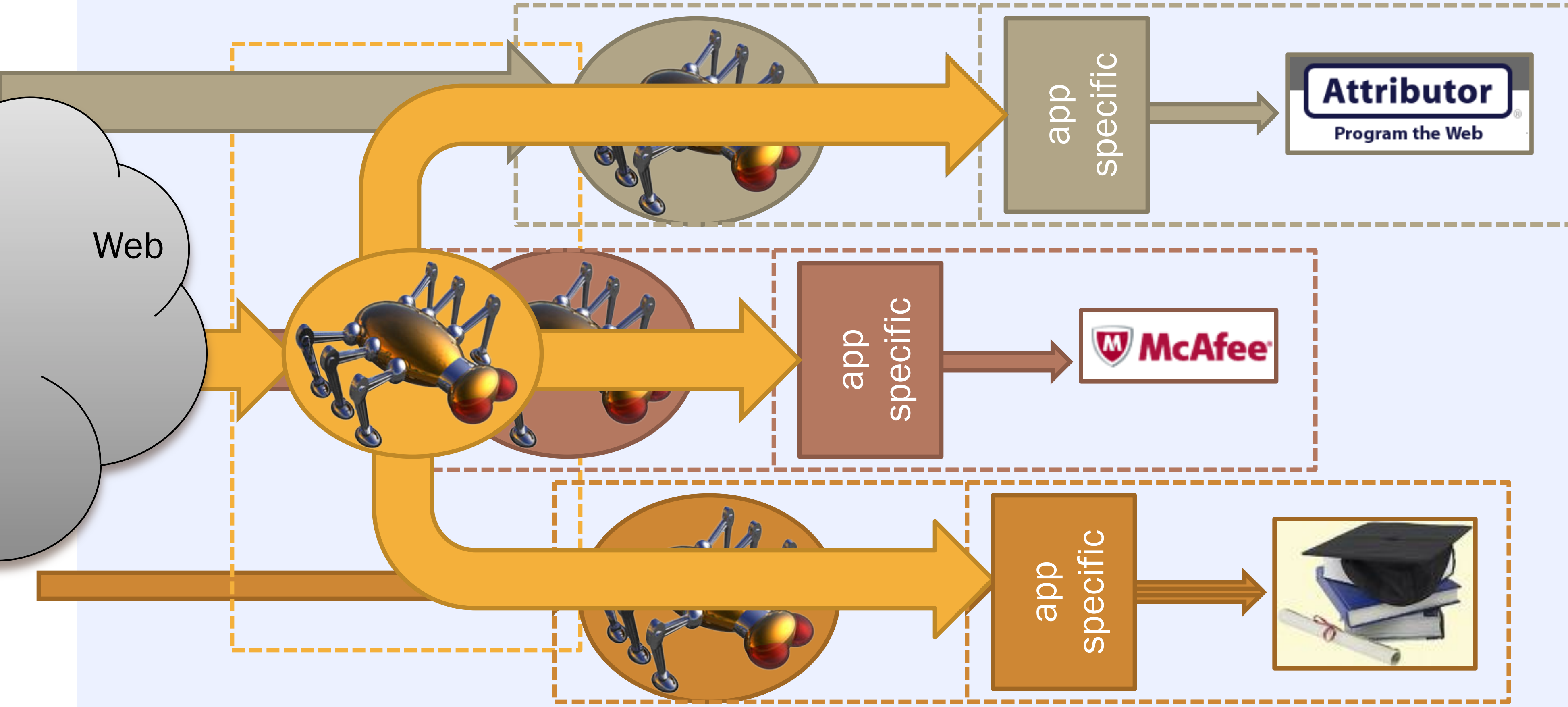
KEY INSIGHT

- Decouple the difficult crawling tasks from the application-specific tasks



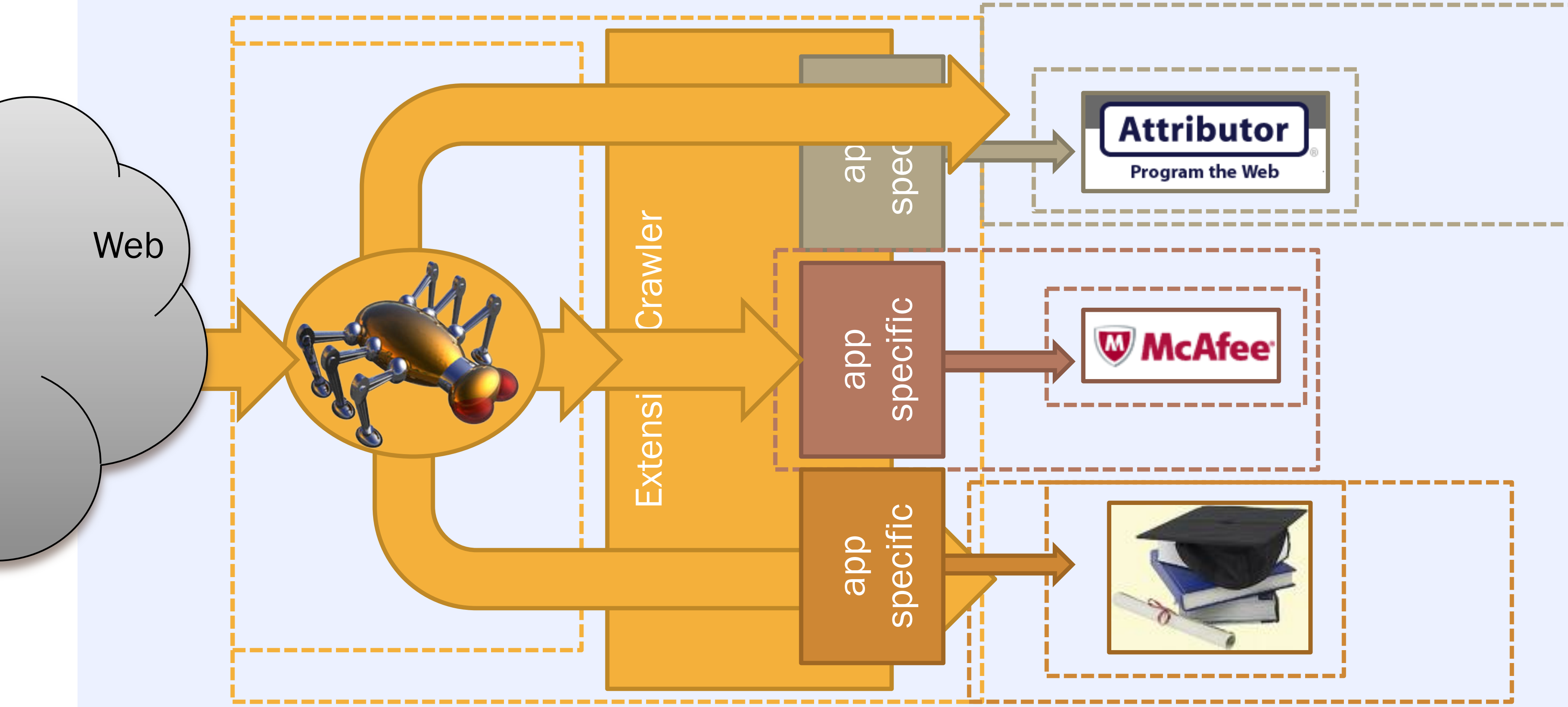
KEY INSIGHT

- Share the crawler as a common service
  - Still need to deliver the deluge



KEY INSIGHT

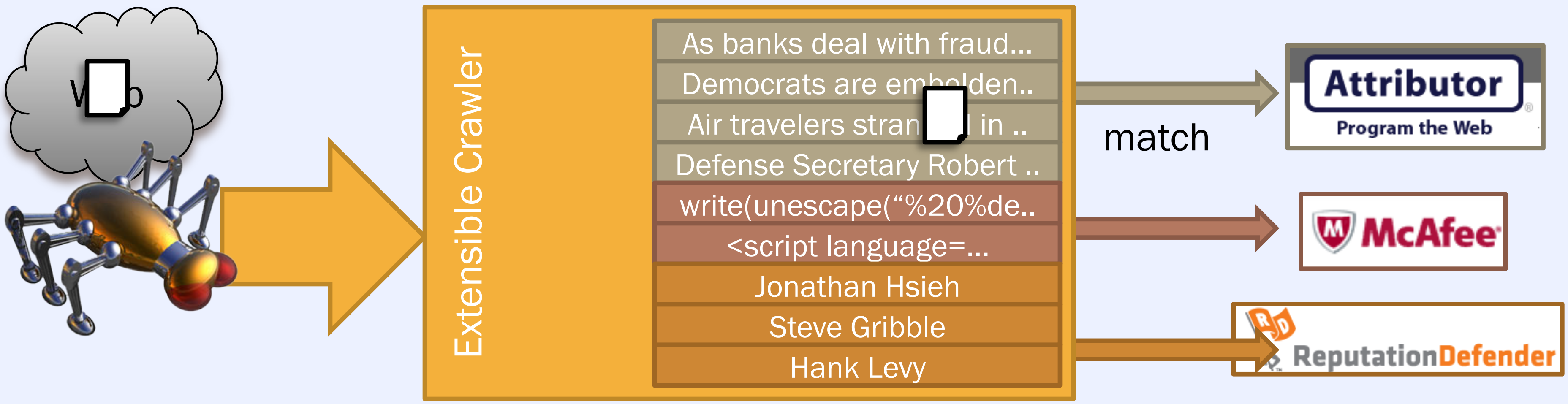
- Make filtering a shared resource
  - Only a small trickle of documents now!



THE EXTENSIBLE CRAWLER

- 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

THE EXTENSIBLE CRAWLER



- 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 <u>service</u> 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

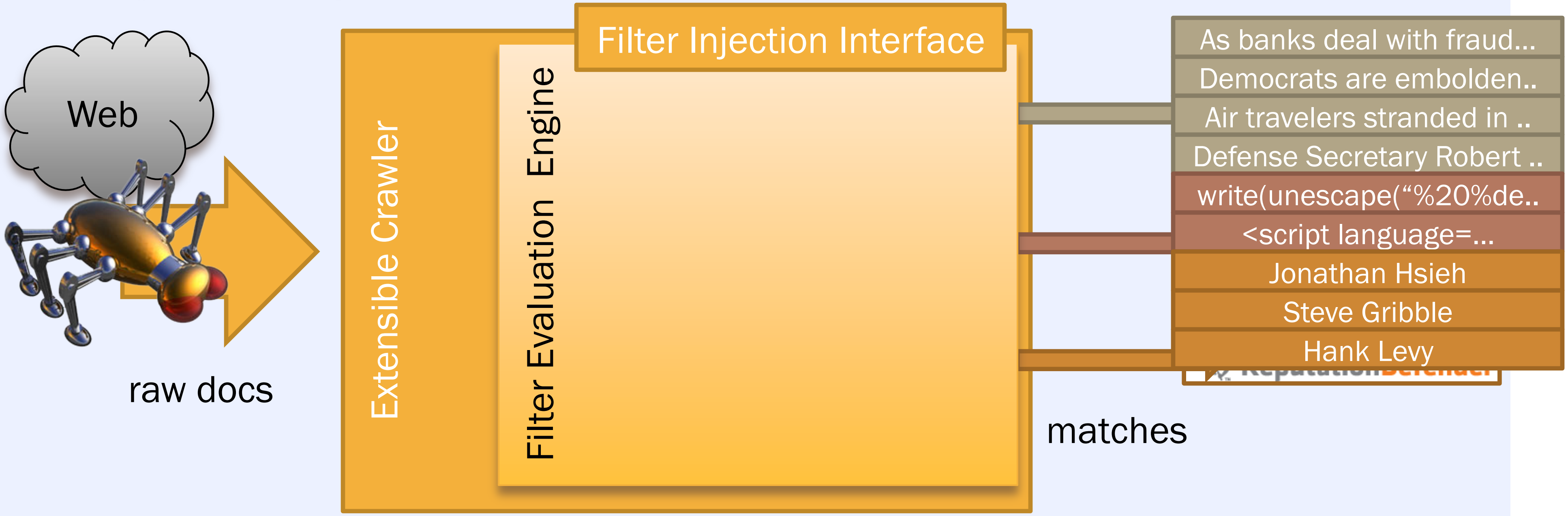
Motivation

ARCHITECTURE

Implementation and Evaluation

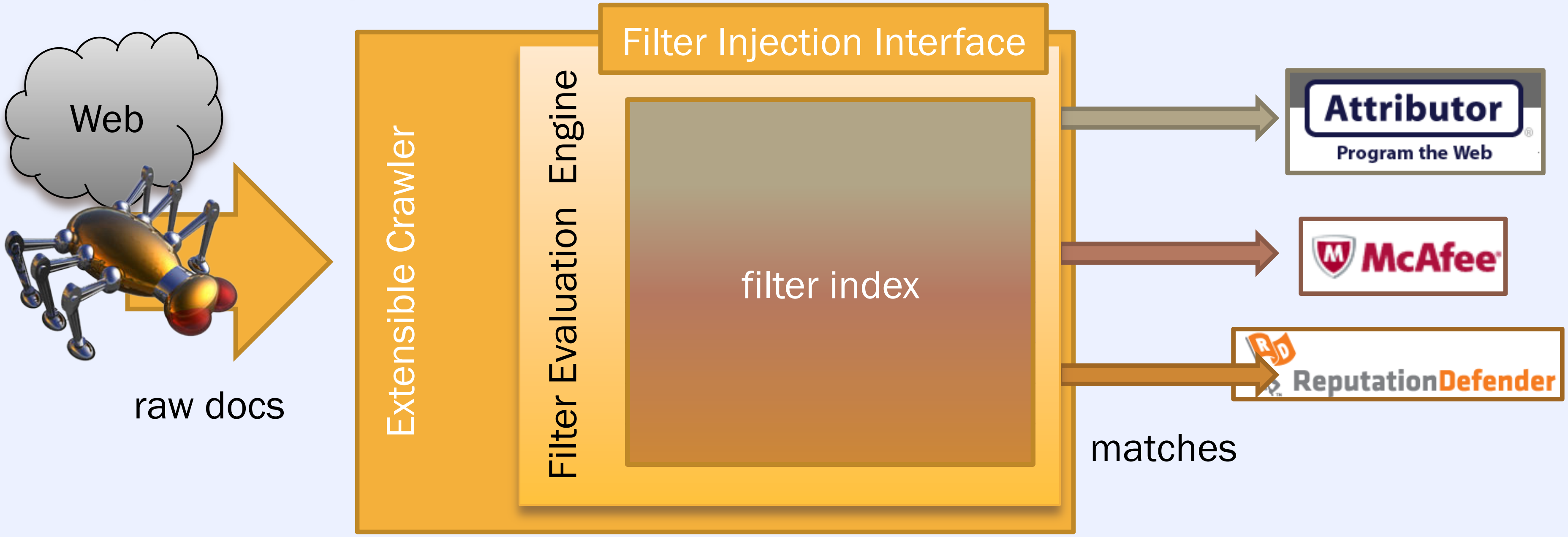
Conclusion

ARCHITECTURE HIGHLIGHTS



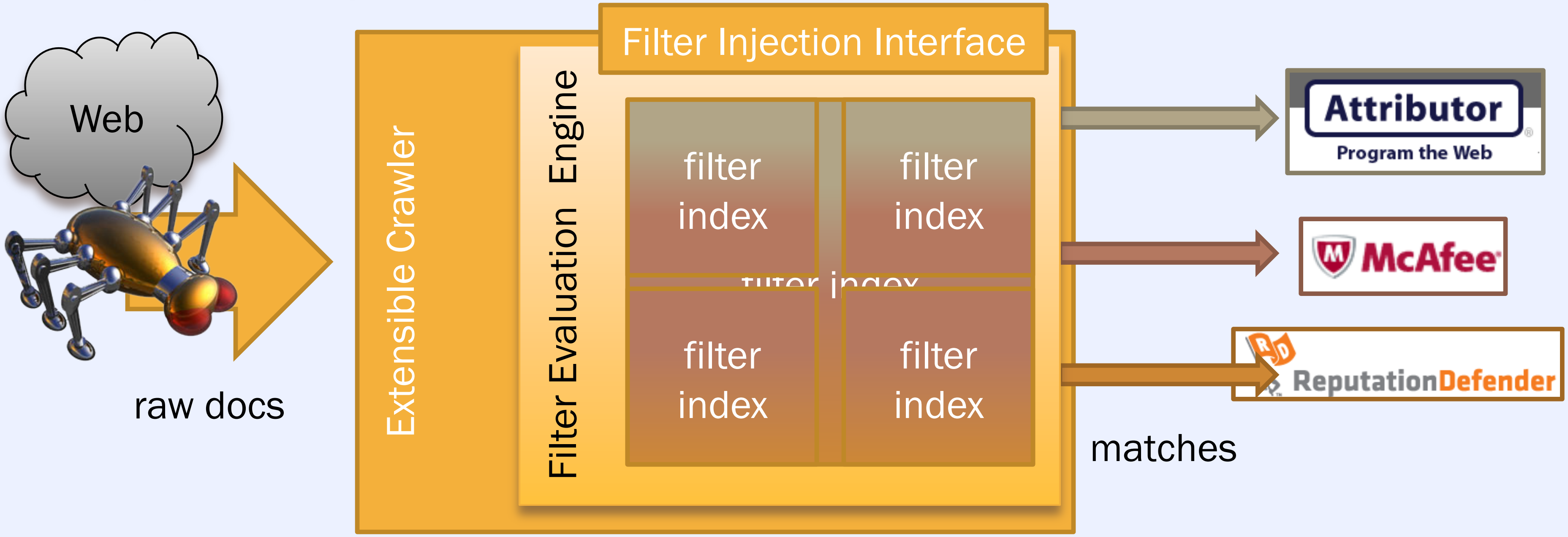
- Design Tradeoffs of Filter Language
- Efficient Filter Evaluation

ARCHITECTURE HIGHLIGHTS



- Design Tradeoffs of Filter Language
- Efficient Filter Evaluation

ARCHITECTURE HIGHLIGHTS



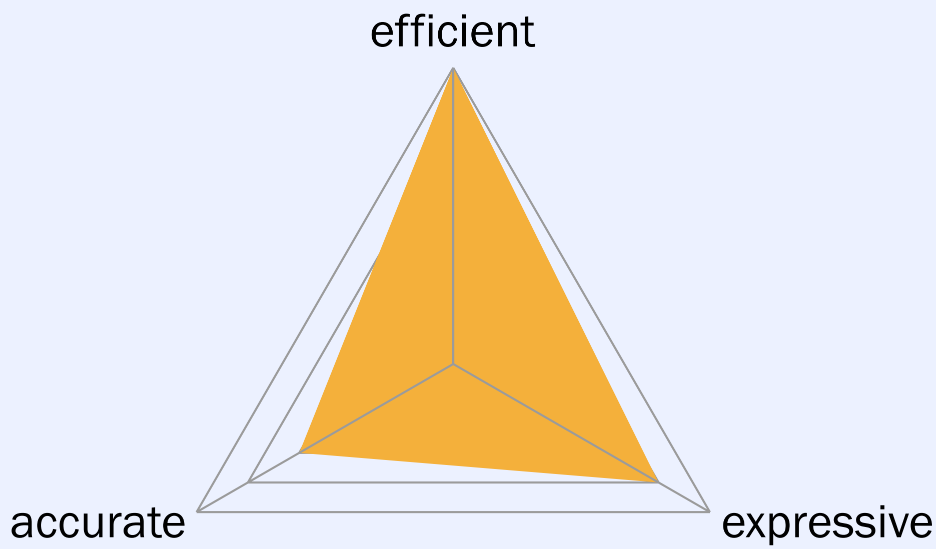
- Design Tradeoffs of Filter Language
- Efficient Filter Evaluation
- Achieving Scale with Commodity Clusters

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”)

LANGUAGE TRADEOFFS

- 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



NAÏVE FILTER EVALUATION

```
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

INDEXED FILTER EVALUATION

```
index and inject filters
for D = next document
   if filterIndex accepts D
      forward to collector
   else
      drop
```

- 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

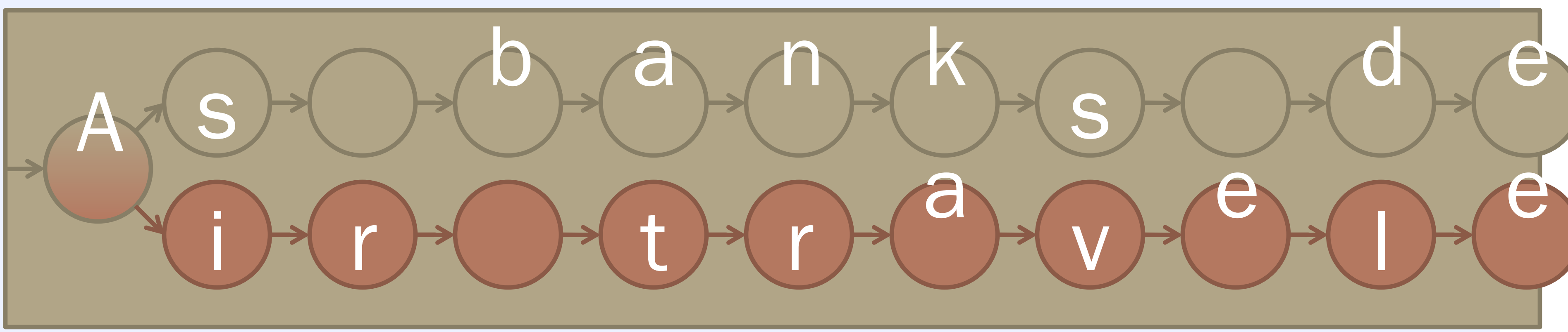
EXAMPLE: INDEXING

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

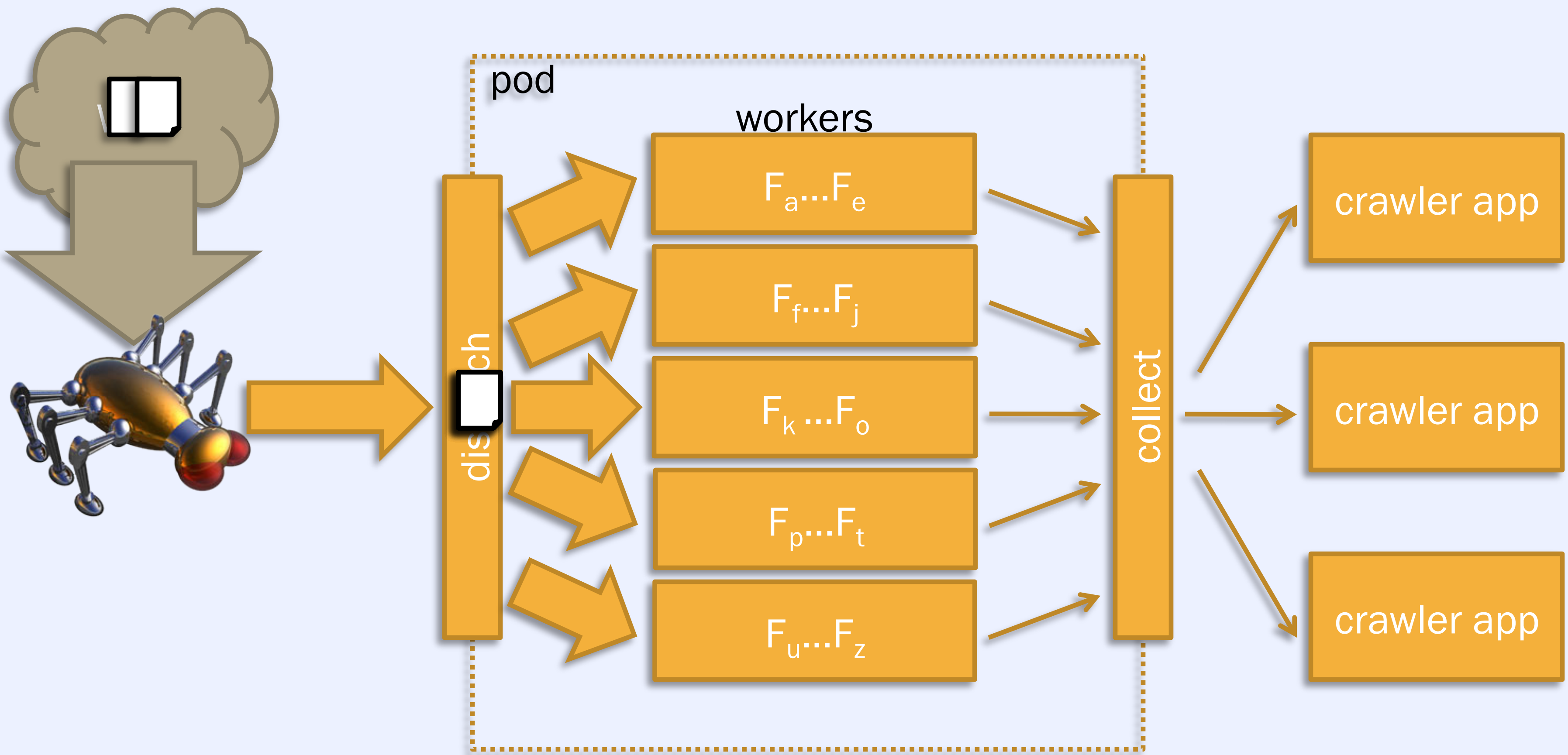
As banks de...

Air travelers ...

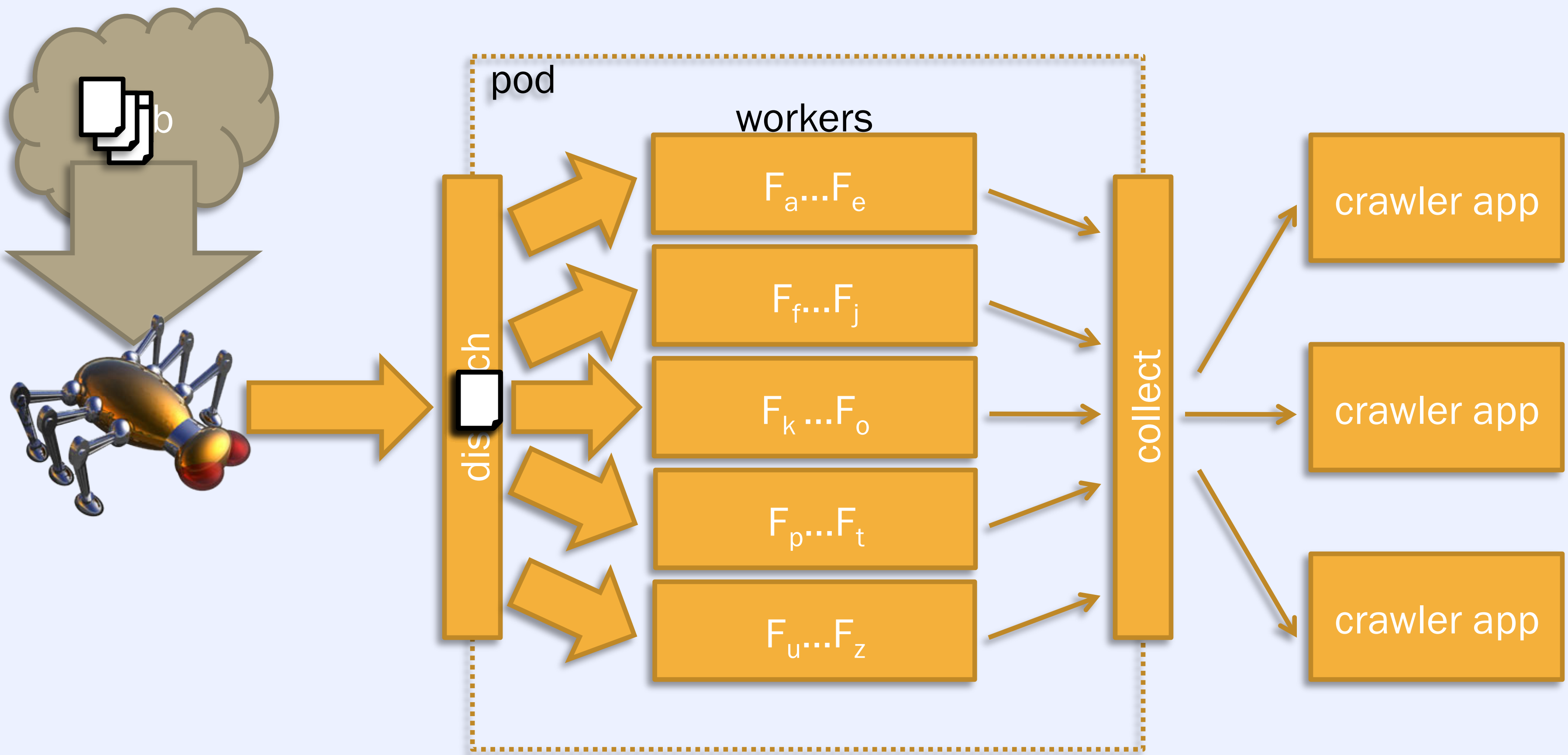
compile into
Aho Corasick DFA



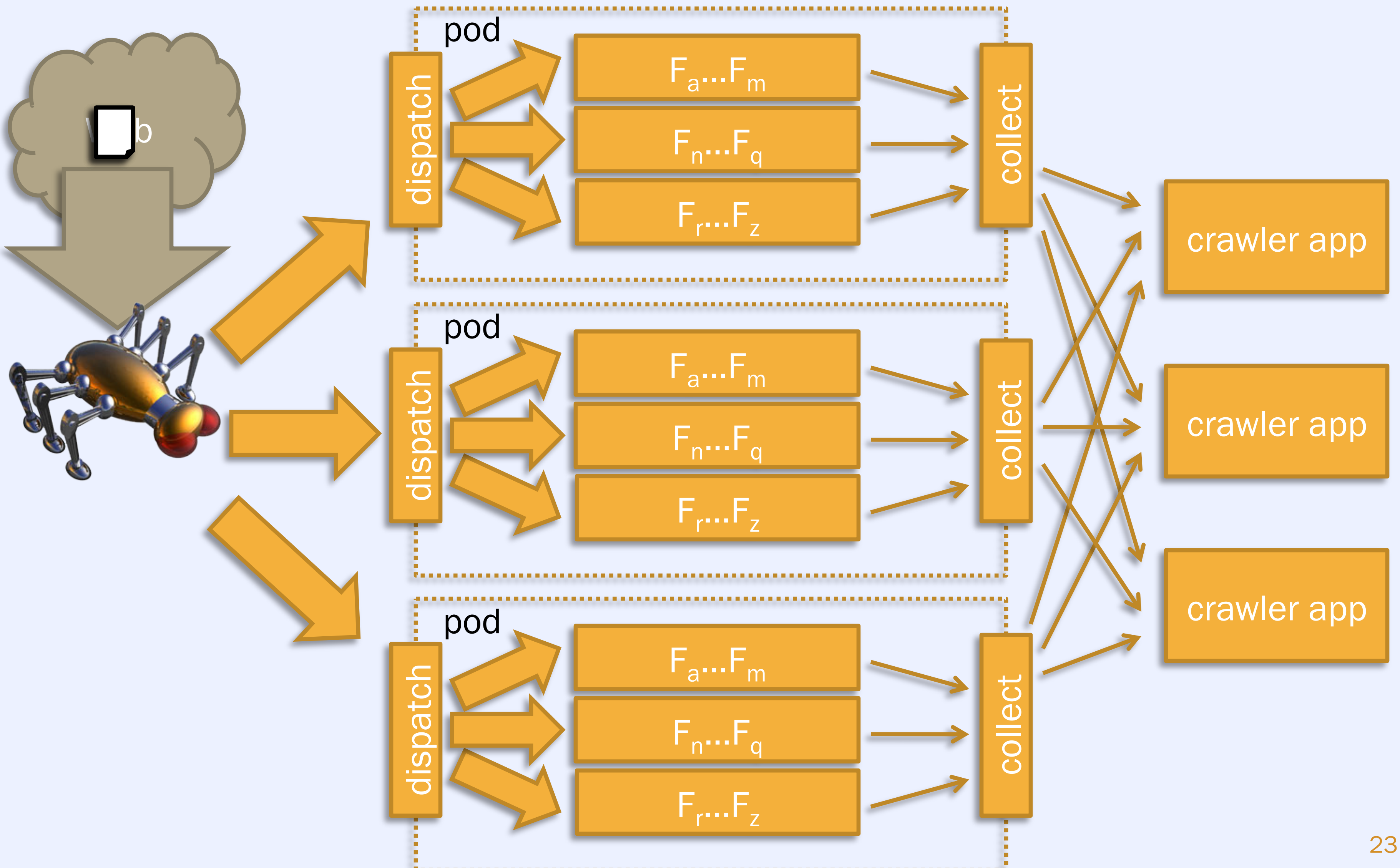
SCALING FILTERS



SCALING FILTERS



SCALING DOCUMENT THROUGHPUT



DISTRIBUTING WORK ACROSS MACHINES

- 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

IMPLEMENTATION AND EVALUATION

- Worker execution optimization
  - Relaxing and Staging filters
- Pod filter partitioning strategies
  - Random vs Sorted
- Prototype crawler applications

RELAXING FILTERS

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").~~

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

STAGING FILTERS

- 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

```
regex('<script language="javascript"> eval
(unescape("%66%75%6e%63%74%69%6f%6e%20%.
{4}%28%.{4}%29%7b%76%61%72%20')
```

- Relaxing a malware regular expression

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

```
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
  - Relax conjunct into a single term

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

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

EXAMPLE: RELAXING FILTERS

substring('<script language="javasc~~ript">~~
~~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

```
substring('<script language="javasc~~ript">
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

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

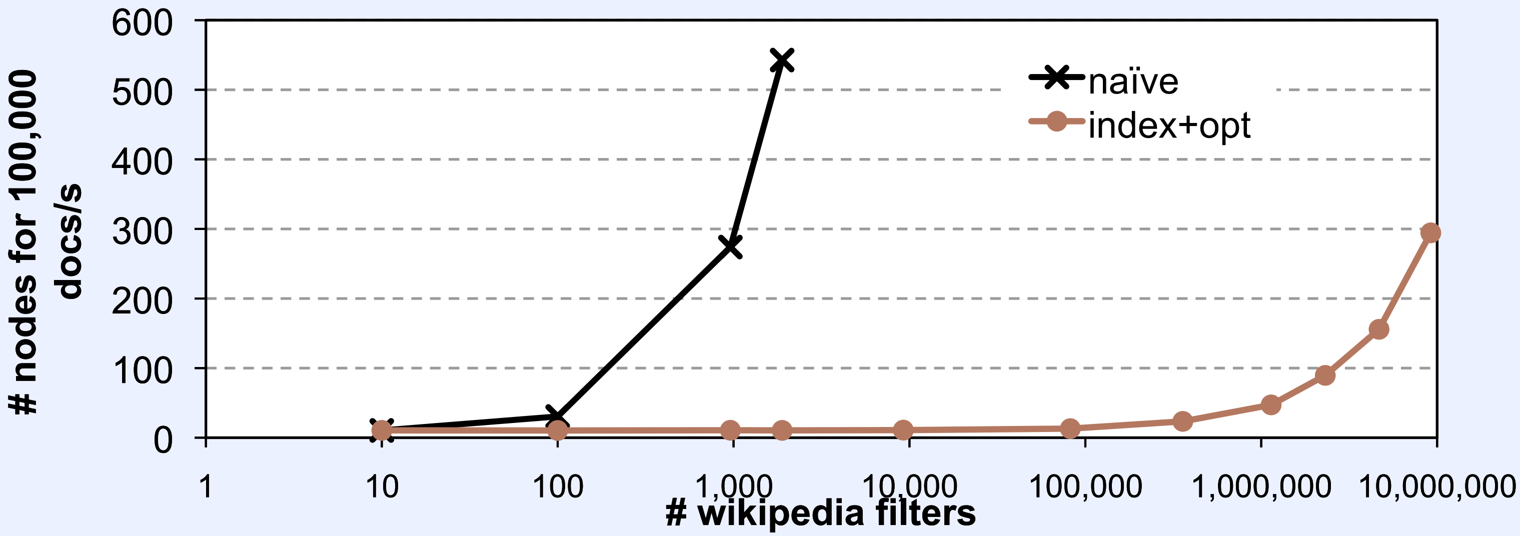
- Relaxing a malware regular expression
  - Relax regex into a conjunct of substrings
  - Relax conjunct into a single term
  - Relax long substring into short substring
  - Select relaxations carefully!

EXAMPLE: RELAXING FILTERS

```
substring('75%6e%63%74%69%6f%6e%20%')
```

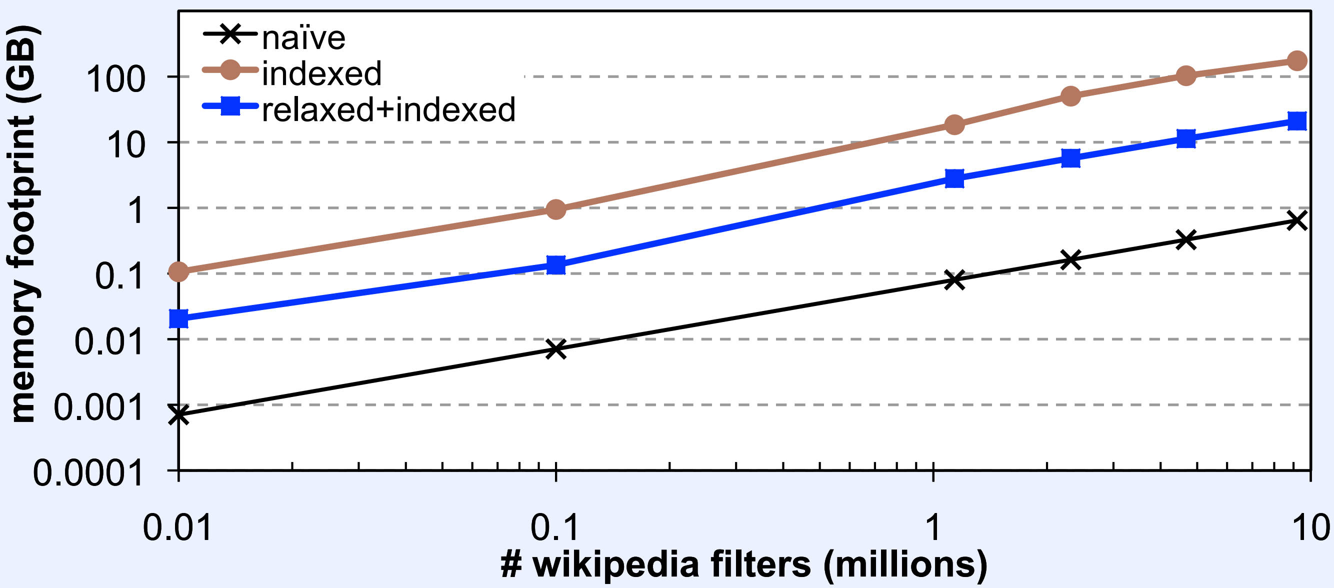
- Relaxing a malware regular expression
  - Relax regex into a conjunct of substrings
  - Relax conjunct into a single term
  - Relax long substring into short substring
  - Select relaxations carefully!

IMPACT OF INDEXED FILTER EXECUTION



- Naïve filter execution is not cost effective
- **Index** filters to use memory instead of CPU
  - Each machines does more work

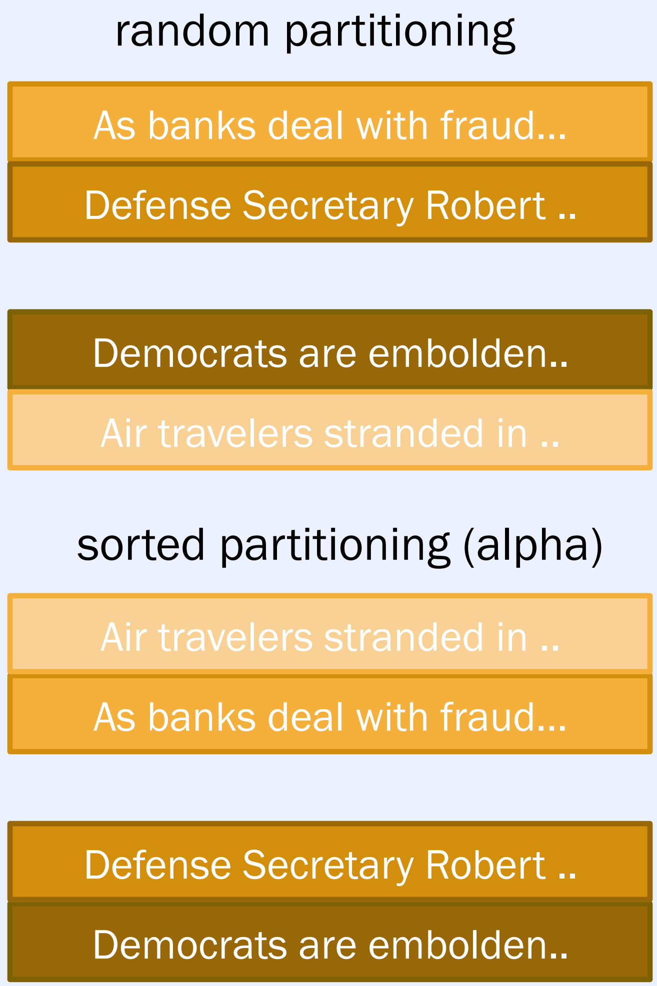
INDEXED FILTER MEMORY USAGE



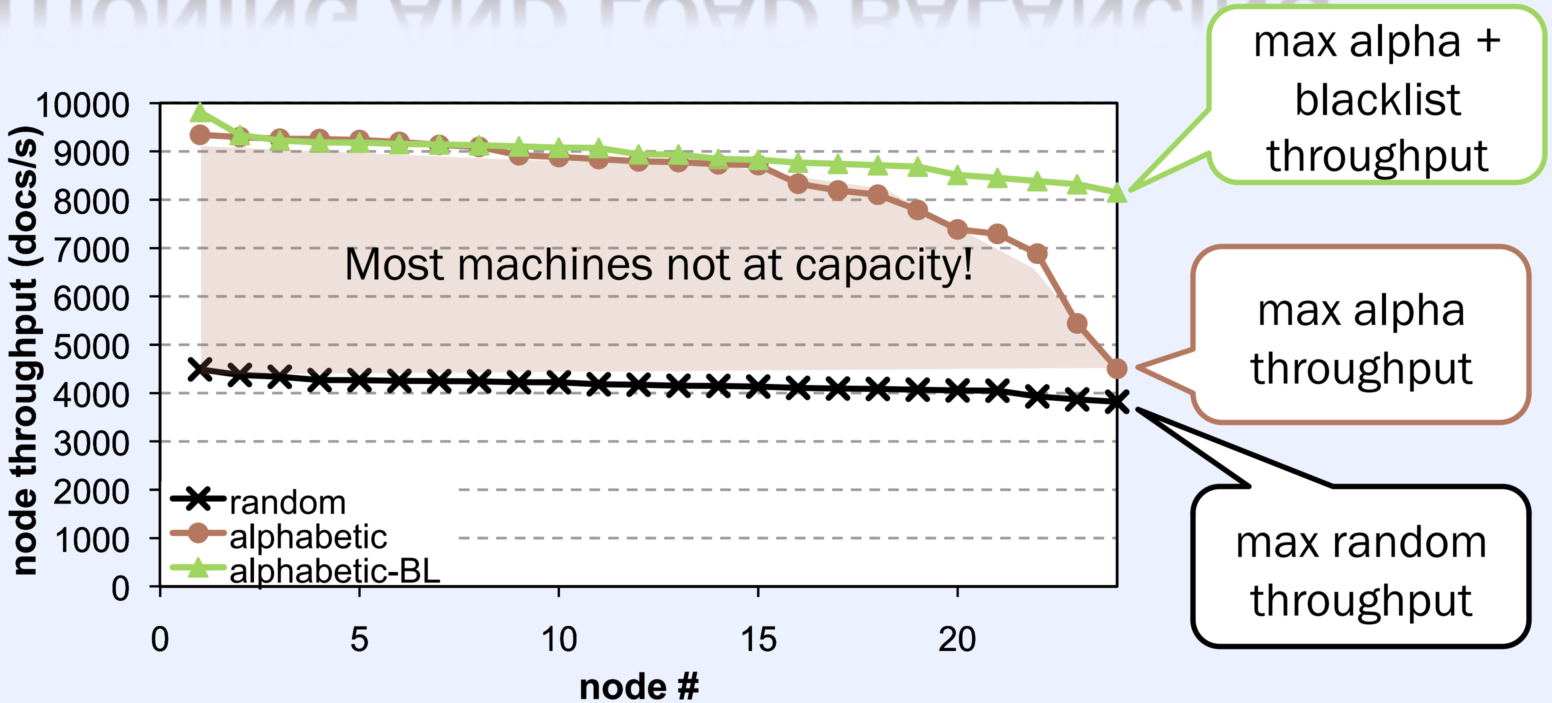
- Indexing is very memory intensive.
- **Relax** filters for less memory consumption
  - Order of magnitude less memory used
  - Order of magnitude more filters on a worker

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 efficiences



PARTITIONING AND LOAD BALANCING



- 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

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 RESULTS

- Applications tested against 3.68M web documents
  - Gathered by Nutch 0.9 crawler and seeded by DMOZ

		Copyright	Malware	Identity
# filters		251,657	3,128	10,622
Relaxed-only	Doc Hit Rate	0.664%	45.4%	69.0%
	Throughput (docs/s)	8,535	8,534	7,244
Relax+staged	Doc Hit Rate	0.016%	0.009%	13.1%
	Throughput (docs/s)	8,229	6,354	592
	# machines for 100k docs/s	12.2	15.7	169

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

THANKS!