The Architecture and Implementation of an Extensible Web Crawler

Jonathan M. Hsieh, Steven D. Gribble, and Henry M. Levy
Department of Computer Science & Engineering
University of Washington, Seattle, WA, USA 98195
{jmhsieh,gribble,levy}@cs.washington.edu

Abstract

Many Web services operate their own Web crawlers to discover data of interest, despite the fact that large-scale, timely crawling is complex, operationally intensive, and expensive. In this paper, we introduce the extensible crawler, a service that crawls the Web on behalf of its many client applications. Clients inject filters into the extensible crawler; the crawler evaluates all received filters against each Web page, notifying clients of matches. As a result, the act of crawling the Web is decoupled from determining whether a page is of interest, shielding client applications from the burden of crawling the Web themselves.

This paper describes the architecture, implementation, and evaluation of our prototype extensible crawler, and also relates early experience from several crawler applications we have built. We focus on the challenges and trade-offs in the system, such as the design of a filter language that is simultaneously expressive and efficient to execute, the use of filter indexing to cheaply match a page against millions of filters, and the use of document and filter partitioning to scale our prototype implementation to high document throughput and large numbers of filters. We argue that the low-latency, high selectivity, and scalable nature of our system makes it a promising platform for taking advantage of emerging real-time streams of data, such as Facebook or Twitter feeds.

1 Introduction

Over the past decade, an astronomical amount of information has been published on the Web. As well, Web services such as Twitter, Facebook, and Digg reflect a growing trend to provide people and applications with access to real-time streams of information updates. Together, these two characteristics imply that the Web has become an exceptionally potent repository of programmatically accessible data. Some of the most provocative recent Web applications are those that gather and process large-scale Web data, such as virtual tourism [33], knowledge extraction [15], Web site trust assessment [24], and emerging trend detection [6].

New Web services that want to take advantage of Web-scale data face a high barrier to entry. Finding and accessing data of interest requires crawling the Web, and if a service is sensitive to quick access to newly published data, its Web crawl must operate continuously and focus on the most relevant subset of the Web. Unfortunately, massive-scale, timely web crawling is complex, operationally intensive, and expensive. Worse, for services that are only interested in specific subsets of Web data, crawling is wasteful, as most pages retrieved will not match their criteria of interest.

In this paper, we introduce the *extensible crawler*, a utility service that crawls the Web on behalf of its many client applications. An extensible crawler lets clients specify filters that are evaluated over each crawled Web page; if a page matches one of the filters specified by a client, the client is notified of the match. As a result, the act of crawling the Web is decoupled from the application-specific logic of determining if a page is of interest, shielding Web-crawler applications from the burden of crawling the Web themselves.

We anticipate two deployment modes for an extensible crawler. First, it can run as a service accessible remotely across the wide-area Internet. In this scenario, filter sets must be very highly selective, since the bandwidth between the extensible crawler and a client application is scarce and expensive. Second, it can run as a utility service [17] within cloud computing infrastructure such as Amazon's EC2 or Google's AppEngine. Filters can be much less selective in this scenario, since bandwidth between the extensible crawler and its clients is abundant, and the clients can pay to scale up the computation processing selected documents.

This paper describes our experience with the design, implementation, and evaluation of an extensible crawler, focusing on the challenges and trade-offs inherent in this class of system. For example, an extensible crawler's filter language must be sufficiently expressive to support interesting applications, but simultaneously, filters must be efficient to execute. A naïve implementation of an extensible crawler would require computational resources proportional to the number of filters it supports multiplied by its crawl rate; instead, our extensible crawler prototype uses standard indexing techniques to vastly reduce the cost of executing a large number of filters. To scale, an extensible crawler must be distributed across a cluster. Accordingly, the system must balance load (both filters and pages) appropriately across machines, otherwise an overloaded machine will limit the rate at which the entire system can process crawled pages. Finally, there must be appropriate mechanisms in place to allow web-crawler applications to update their filter sets frequently and efficiently.

We demonstrate that XCrawler, our early prototype system, is scalable across several dimensions: it can efficiently process tens of millions of concurrent filters while processing thousands of Web pages per second. XCrawler is also flexible. By construction, we show that its filter specification language facilitates a wide range of interesting web-crawler applications, including keyword-based notification, Web malware detection and defense, and copyright violation detection.

An extensible crawler bears similarities to several other systems, including streaming and parallel databases [1, 10, 11, 13, 14, 19], publish-subscribe systems [2, 9, 16, 27, 31], search engines and web crawlers [8, 12, 20, 21], and packet filters [23, 25, 30, 32, 34]. Our design borrows techniques from each, but we argue that the substantial differences in the workload, scale, and application requirements of extensible crawlers mandate many different design choices and optimizations. We compare XCrawler to related systems in the related work section (Section 5), and we provide an in-depth comparison to search engines in Section 2.1.

2 Overview

To better motivate the goals and requirements of extensible crawlers, we now describe a set of web-crawler applications that we have experimented with using our prototype system. Table 1 gives some order-of-magnitude estimates of the workload that we expect these applications would place on an extensible crawler if deployed at scale, including the total number of filters each application category would create and the selectivity of a client's filter set.

Keyword-based notification. Similar to Google Alerts, this application allows users to register keyword phrases of interest, and receive an event stream corresponding to Web pages containing those keywords. For example, users might upload a vanity filter ("Jonathan Hsieh"), or a filter to track a product or company ("palm pre"). This application must support a large number of users, each with a small and relatively slowly-changing filter set. Each filter should be highly selective, matching a very small fraction of Web pages.

	keyword notification	Web malware	copyright violation	Web research
# clients	~10^6	~10^2	~10^2	~10^3
# filters per client	~10^2	~10^6	~10^6	~10^2
fraction of pages that match for a client	~10^-5	~10^-4	~10^-6	~10^-3
total # filters	~10^8	~10^8	~10^8	~10^5

Table 1: **Web-crawler application workloads.** This table summarizes the approximate filter workloads we expect from four representative applications.

Web malware detection. This application uses a database of regular-expression-based signatures to identify malicious executables, JavaScript, or Web content. New malware signatures are injected daily, and clients require prompt notification when new malicious pages are discovered. This application must support a small number of clients (e.g., `McAfee`, `Google`, and `Symantec`), each with a large and moderately quickly changing filter set. Each filter should be highly selective; in aggregate, approximately roughly 1 in 1000 Web pages contain malicious content [26, 28].

Copyright violation detection. Similar to commercial offerings such as `attributor.com`, this application lets clients find Web pages containing content containing their intellectual property. A client, such as a news provider, maintains a large database of highly selective filters, such as key sentences from their news articles. New filters are injected into the system by a client whenever new content is published. This application must support a moderate number of clients, each with a large, selective, and potentially quickly changing filter set.

Web measurement research. This application permits scientists to perform large-scale measurements of the Web to characterize its content and dynamics. Individual research projects would inject filters to randomly sample Web pages (e.g., sample 1 in 1000 random pages as representative of the overall Web) or to select Web pages with particular features and tags relevant to the study (e.g., select Ajax-related JavaScript keywords in a study investigating the prevalence of Ajax on the Web). This application would support a modest number of clients with a moderately sized, slowly changing filter set.

2.1 Comparison to a search engine

At first glance, one might consider implementing an extensible crawler as a layer on top of a conventional search engine. This strawman would periodically execute filters against the search engine, looking for new document matches and transmitting those to applications. On closer inspection, however, several fundamental differences between search engines and extensible crawlers, their workloads, and their performance requirements are evident, as summarized in Table 2. Because of

	search engine	extensible crawler
clients	millions of people	thousands of applications
documents	~trillion stored in a crawl database and periodically refreshed by crawler	arrive in a stream from crawler, processed on-the-fly and not stored
filters / queries	arrive in a stream from users, processed on-the-fly and not stored	~billion stored in a filter database and periodically updated by applications
indexing	index documents	index queries
latency	query response time is crucial; document refresh latency is less important	document processing time is crucial; filter update latency is less important
selectivity and query result ranking	queries might not be selective; result ranking is crucial for usability	filters are assumed to be selective; all matches are sent to applications
caching	in-memory cache of popular query results and "important" index subset	entire filter index is stored in memory; result caching is not relevant

Table 2: **Search engines vs. extensible crawlers.** This table summarizes key distinctions between the workload, performance, and scalability requirements of search engines and extensible crawlers.

these differences, we argue that there is an opportunity to design an extensible crawler that will scale more efficiently and better suit the needs of its applications than a search-engine-based implementation.

In many regards, an extensible crawler is an inversion of a search engine. A search engine crawls the Web to periodically update its stored index of Web documents, and receives a stream of Web queries that it processes against the document index on-the-fly. In contrast, an extensible crawler periodically updates its stored index of filters, and receives a stream of Web documents that it processes against the filter index on-the-fly. For a search engine, though it is important to reduce the time in between document index updates, it is crucial to minimize query response time. For an extensible crawler, it is important to be responsive in receiving filter updates from clients, but for "real-time Web" applications, it is more important to process crawled documents with low latency.

There are also differences in scale between these two systems. A search engine must store and index hundreds of billions, if not trillions, of Web documents, containing kilobytes or megabytes of data. On the other hand, an extensible crawler must store and index hundreds of millions, or billions, of filters; our expectation is that filters are small, perhaps dozens or hundreds of bytes. As a result, an extensible crawler must store and index four or five orders of magnitude less data than a search engine, and it is more likely to be able to afford to keep its entire index resident in memory.

Finally, there are important differences in the performance and result accuracy requirements of the two systems. A given search engine query might match millions of Web pages. To be usable, the search engine must rely heavily on page ranking to present the top matches to users. Filters for an extensible crawler are assumed to be more selective than search engine queries, but even if they are not, filters are executed against documents as they are crawled rather than against the enormous Web corpus gathered by a search engine. All matching pages found by an extensible crawler are communicated to a web-crawler application; result ranking is not relevant.

Traditional search engines and extensible crawlers are in some ways complementary, and they can co-exist. Our work focuses on quickly matching freshly crawled documents against a set of filters, however, many applications can benefit from being able to issue queries against a full, existing Web index in addition to filtering newly discovered content.

2.2 Architectural goals

Our extensible crawler architecture has been guided by several principles and system goals:

High Selectivity. The primary role of an extensible crawler is to reduce the number of web pages a web-crawler application must process by a substantial amount, while preserving pages in which the application might have interest. An extensible crawler can be thought of as a highly selective, programmable matching filter executing as a pipeline stage between a stock Web crawler and a web-crawler application.

Indexability. When possible, an extensible crawler should trade off CPU for memory to reduce the computational cost of supporting a large number of filters. In practice, this implies constructing an index over filters to support the efficient matching of a document against all filters. One implication of this is that the index must be kept up-to-date as the set of filters defined by web-crawler applications is updated. If this update rate is low or the indexing technique used supports incremental updates, keeping the index up-to-date should be efficient.

Favor Efficiency over Precision. There is generally a tradeoff between the precision of a filter and its efficient execution, and in these cases, an extensible crawler should favor efficient execution. For example, a filter language that supports regular expressions can be more precise than a filter language that supports only conjuncts of substrings, but it is simpler to build an efficient index over the latter. As we will discuss in Section 3.2.2, our XCrawler prototype implementation exposes a rich filter language to web-crawler applications, but uses *relaxation* to convert precise filters into less-precise, indexable versions, increasing its scalability at the cost of exposing false positive matches to the applications.

Low Latency. To support crawler-applications that depend on real-time Web content, an extensible crawler should be capable of processing Web pages with low latency. This goal suggests the extensible crawler should

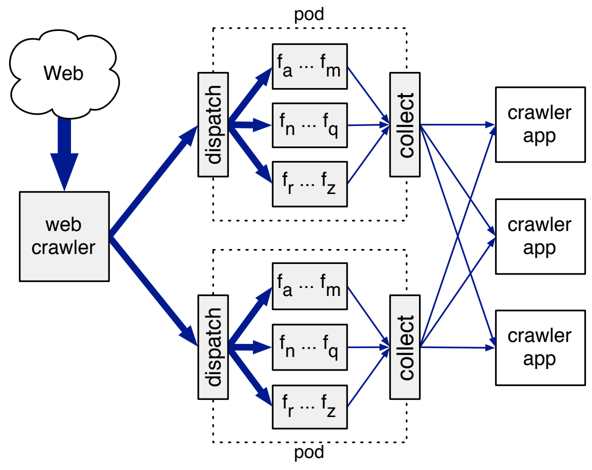

Figure 1: **Extensible crawler architecture.** This figure depicts the high-level architecture of an extensible crawler, including the flow of documents from the Web through the system.

be architected as a stage in a dataflow pipeline, rather than as a batch or map-reduce style computation.

Scalability. An extensible crawler should scale up to support high Web page processing rates and a very large number of filters. One of our specific goals is to handle a linear increase in document processing rate with a corresponding linear increase in machine resources.

2.3 System architecture

Figure 1 shows the high-level architecture of an extensible crawler. A conventional Web crawler is used to fetch a high rate stream of documents from the Web. Depending on the needs of the extensible crawler's applications, this crawl can be broad, focused, or both. For example, to provide applications with real-time information, the crawler might focus on real-time sources such as Twitter, Facebook, and popular news sites.

Web documents retrieved by the crawler are partitioned across **pods** for processing. A pod is a set of nodes that, in aggregate, contains all filters known to the system. Because documents are partitioned across pods, each document needs to be processed by a single pod; by increasing the number of pods within the system, the overall throughput of the system increases. **Document set partitioning** therefore facilitates the scaling up of the system's document processing rate.

Within each pod, the set of filters known to the extensible crawler is partitioned across the pod's nodes. **Filter set partitioning** is a form of sharding and it is used to address the memory or CPU limitations of an individual node. As more filters are added to the extensible crawler, additional nodes may need to be added to each pod, and the partitioning of filters across nodes might need adjustment. Because filters are partitioned across pod nodes, each document arriving at a pod needs to be distributed to each pod node for processing. Thus, the throughput of the pod is limited by the slowest node within the pod; this implies that load balancing of filters across pod nodes is crucially important to the overall system throughput.

Each node within the pod contains a subset of the system's filters. A naïve approach to processing a document on a node would involve looping over each filter on that node serially. Though this approach would work correctly, it would scale poorly as the number of filters grows. Instead, as we will discuss in Section 3.2, we trade memory for computation by using *filter indexing*, *relaxation*, and *staging* techniques; this allows us to evaluate a document against a node's full filter set with much faster than linear processing time.

If a document matches any filters on a node, the node notifies a match collector process running within the pod. The collector gathers all filters that match a given document and distributes match notifications to the appropriate web-crawler application clients.

Applications interact with the extensible crawler through two interfaces. They upload, delete, or modify filters in their filter sets with the filter management API. As well, they receive a stream of notification events corresponding to documents that match at least one of their filters through the notification API. We have considered but not yet experimented with other interfaces, such one for letting applications influence the pages that the web crawler visits.

3 XCrawler Design and Implementation

In this section, we describe the design and implementation of XCrawler, our prototype extensible crawler. XCrawler is implemented in Java and runs on a cluster of commodity multi-core x86 machines, connected by a gigabit switched network. Our primary optimization concern while building XCrawler was efficiently scaling to a large number of expressive filters.

In the rest of this section, we drill down into four aspects of XCrawler's design and implementation: the filter language it exposes to clients, how a node matches an incoming document against its filter set, how documents and filters are partitioned across pods and nodes, and how clients are notified about matches.

3.1 Filter language and document model

XCrawler's declarative filter language strikes a balance between expressiveness for the user and execution efficiency for the system. The filter language has four entities: attributes, operators, values, and expressions. There are two kinds of values: simple and composite. Simple values can be of several types, including byte sequences, strings, integers and boolean values. Composite values are tuples of values.

A document is tuple of attribute and values pairs. Attributes are named fields within a document; during crawling, each Web document is pre-processed to extract a static set of attributes and values. This set is passed to nodes and is referenced by filters during execution. Examples of a document's attribute-value pairs include its URL, the raw HTTP content retrieved by the crawler, certain HTTP headers like Content-Length or Content-Type, and if appropriate, structured text extracted from the raw content. To support sampling, we also provide a random number attribute whose per-document value is fixed at chosen when other attributes are extracted.

A user-provided filter is a predicate expression; if the expression evaluates to true against a document, then the filter matches the document. A predicate expression is either a boolean operator over a single document attribute, or a conjunct of predicate expressions. A boolean operator expression is an (attribute, operator, value) triple, and is represented in the form:

```
attribute.operator(value)
```

The filter language provides expensive operators such as substring and regular expression matching as well as simple operators like equalities and inequalities.

For example, a user could specify a search for the phrase "Barack Obama" in HTML files by specifying:

```
mimetype.equals("text/html") &
text.substring("Barack Obama")
```

Alternatively, the user could widen the set of acceptable documents by specifying a conjunction of multiple, less restrictive keyword substring filters.

```
mimetype.equals("text/html") &
text.substring("Barack") &
text.substring("Obama")
```

Though simple, this language is rich enough to support the applications outlined previously in Section 2. For example, our prototype Web malware detection application is implemented as a set of regular expression filters derived from the ClamAV virus and malware signature database.

3.2 Filter execution

When a newly crawled document is dispatched to a node, that node must match the document against its set of filters. As previously mentioned, a naïve approach to executing filters would be to iterate over them sequentially; unfortunately, the computational resources required for this approach would scale linearly with both the number of filters and the document crawl rate, which is severely limiting. Instead, we must find a way to optimize the execution of a set of filters.

To do this, we rely on three techniques. To maintain throughput while scaling up the number of filters on a node, we create memory-resident *indexes* for the attributes referenced by filters. Matching a document against an indexed filter set requires a small number of index lookups, rather than computation proportional to the number of filters. However, a high fidelity index might require too much memory, and constructing an efficient index over an attribute that supports a complex operator such as a regular expression might be intractable. In either case, we use *relaxation* to convert a filter into a form that is simpler or cheaper to index. For example, we can relax a regular expression filter into one that uses a conjunction of substring operators.

A relaxed filter is less precise than the full filter from which it was derived, potentially causing false positives. If the false positive rate is too high, we can feed the tentative matches from the index lookups into a second stage that executes filters precisely but at higher cost. By *staging* the execution of some filters, we regain higher precision while still controlling overall execution cost. However, if the false positive rate resulting from a relaxed filter is acceptably low, staging is not necessary, and all matches (including false positives) are sent to the client. Whether a false positive rate is acceptable depends on many factors, including the execution cost of staging in the extensible crawler, the bandwidth overhead of transmitting false positives to the client, and the cost to the client of handling false positives.

3.2.1 Indexing

Indexed filter execution requires the construction of an index for each attribute that a filter set references, and for each style of operator that is used on those attributes. For example, if a filter set uses a substring operator over the document body attribute, we build an Aho-Corasick multistring search trie [3] over the values specified by filters referencing that attribute. As another example, if a filter set uses numeric inequality operators over the document size attribute, we construct a binary search tree over the values specified by filters referencing that attribute.

Executing a document against a filter set requires looking up the document's attributes against all indexes to find potentially matching filters. For filters that contain a conjunction of predicate expressions, we could insert each expression into its appropriate index. Instead, we identify and index only the most selective predicate expression; if the filter survives this initial index lookup, we can either notify the client immediately and risk false positives or use staging (discussed in Section 3.2.3) to evaluate potential matches more precisely.

Creating indexes lets us execute a large number of filters efficiently. Figure 2 compares the number of nodes that would be required in our XCrawler prototype to sustain a crawl rate of 100,000 documents per second, using either naïve filter execution or filter execution with in-

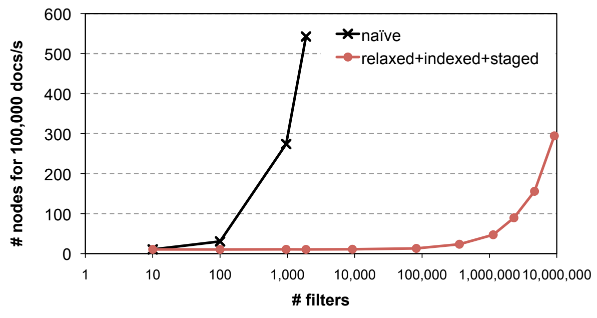

Figure 2: **Indexed filter execution.** This graph compares the number of nodes (machines) required for a crawl rate of 100,000 documents per second when using naïve filter execution and when using indexed filter execution, including relaxation and staging.

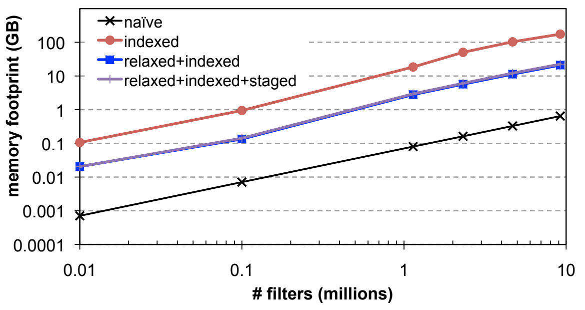

Figure 3: **Indexed and relaxed filter memory footprint.** This graph compares the memory footprint of substring filters when using four different execution strategies. The average filter length in the filter set was 130 bytes, and relaxation used 32 byte ngrams. Note that the *relaxed+indexed* and the *relaxed+indexed+staged* lines overlap on the graph.

dexing, relaxation, and staging enabled. The filter set used in this measurement are sentences extracted from Wikipedia articles; this emulates the workload of a copyright violation detection application. Our measurements were gathered on a small number of nodes, and projected upwards to larger numbers of nodes assuming linear scaleup in crawl rate with document set partitioning.

Our prototype runs on 8 core, 2 GHz Intel processors. When using indexing, relaxation, and staging, a node with 3GB of RAM is capable of storing approximately 400,000 filters of this workload, and can process documents at a rate of approximately 9,000 documents per second. To scale to 100,000 documents per second, we would need 12 pods, i.e., we must replicate the full filter set 12 times, and partition incoming documents across these replicas. To scale to 9,200,000 filters, we would need to partition the filter set across 24 machines with 3GB of RAM each. Thus, the final system configuration would have 12 pods, each with 24 nodes, for a total of 288 machines. If we installed more RAM on each machine, we would need commensurately fewer machines.

Even when including the additional cost of staging, indexed execution can provide several orders of magnitude better scaling characteristics than naïve execution as the number of filters grows. Note that the CPU is the bottleneck resource for execution in both cases, although with staged indexing, staging causes the CPUs to be primarily occupied with processing false positives from relaxed filters.

3.2.2 Relaxation

We potentially encounter two problems when using indexing: the memory footprint of indexes might be excessive, and it might be infeasible to index attributes or operators such as regular expressions or conjuncts. To cope with either problem, we use relaxation to convert a filter into a form that is less accurate but indexable.

As one example, consider copyright violation detection filters that contain sentences that should be searched for as substrings within documents. Instead of searching for the full sentence, filters can be relaxed to search for an ngram extracted from the sentence (e.g., a 16 byte character fragment). This would significantly reduce the size of the in-memory index.

There are many possible ngram relaxations for a specific string; the ideal relaxation would be just as selective as the full sentence, returning no false positives. Intuitively, shorter ngrams will tend to be less selective but more memory efficient. Less intuitively, different fragments extracted from the same string might have different selectivity. Consider the string `<a href="http://zyzzyva.com">`, and two possible 8-byte relaxations `<a href=` and `/zyzzyva`: the former would be much less selective than the latter. Given this, our prototype gathers run-time statistics on the hit rate of relaxed substring operations, identifies relaxations that have anomalously high hit rates, and selects alternative relaxations for them. If we cannot find a low hit rate relaxation, we ultimately reject the filter.

Relaxation also allows us to index operations that are not directly or efficiently indexable. Conjuncts are not directly indexable, but can be relaxed by picking a selective indexable subexpression. A match of this subexpression is not as precise as the full conjunction, but can eliminate a large portion of true negatives. Similarly, regular expressions could hypothetically be indexed by combining their automata, but combined automata tend to have exponentially large state requirements or high computational requirements [12, 23, 32]. Instead, if we can identify substrings that the regular expression implies must occur in an accepted document, we can relax the regular expression into a less selective but indexable substring. If a suitably selective substring cannot be identified from a given regular expression, that filter can be rejected when

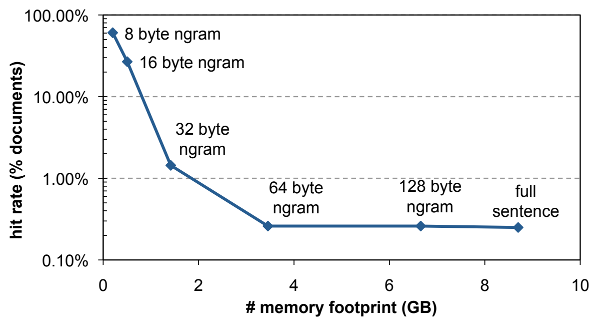

Figure 4: **Filter relaxation trade-off.** This graph illustrates the trade-off between memory footprint and false positive rates when using different degrees of relaxation.

the client application attempts to upload it.

Figure 3 compares the memory footprint of naïve, indexed, relaxed+indexed, and relaxed+indexed+staged filter execution. The filter set used in this measurement is the same as in Figure 2, namely sentences extracted from Wikipedia articles, averaging 130 characters in length. Relaxation consists of selecting a random 32 character substring from a sentence. The figure demonstrates that indexing imposes a large memory overhead relative to naïve execution, but that relaxation can substantially reduce this overhead.

Relaxation potentially introduces false positives. Figure 4 illustrates the trade-off between the memory footprint of filter execution and the hit rate, as the degree of relaxation used varies. With no relaxation, an indexed filter set of 400,000 Wikipedia sentences averaging 130 characters in length requires 8.7GB of memory and has a hit rate of 0.25% of Web documents. When relaxing these filters to 64 byte ngrams, the memory footprint is reduced to 3.5GB and the hit rate marginally climbs to 0.26% of documents. More aggressive relaxation causes a substantial increase in false positives. With 32 byte ngrams, the memory footprint is just 1.4GB, but the hit rate grows to 1.44% of documents: nearly four out of five hits are false positives.

3.2.3 Staging

If a relaxed filter causes too many false positives, we can use staging to eliminate them at the cost of additional computation. More specifically, if a filter is marked for staging, any document that matches the relaxed version of the filter (a *partial hit*) is subsequently executed against the full version of that filter. Thus, the first stage of filter execution consists of index lookups, while the second stage of execution iterates through the partial hits identified by the first stage.

The second stage of execution does not benefit from indexing or relaxation. Accordingly, if the partial hit rate in the first stage is too high, the second stage of execution has the potential to dominate computation time and limit the throughput of the system. As well, any filter that is staged requires the full version of the filter to be stored in memory. Staging eliminates false positives, but has both a computational and memory cost.

3.3 Partitioning

As with most cluster-based services, the extensible crawler achieves cost-efficient scaling by partitioning its work across inexpensive commodity machines. Our workload consists of two components: documents that continuously arrive from the crawler and filters that are periodically uploaded or updated by client applications. To scale, the extensible crawler must find an intelligent partitioning of both documents and filters across machines.

3.3.1 Document set partitioning

Our first strategy, which we call *document set partitioning*, is used to increase the overall throughput of the extensible crawler. As previously described, we define a **pod** as a set of nodes that, in aggregate, contains all filters known to the system. Thus, each pod contains all information necessary to process a document against a filter set. To increase the throughput of the system, we can add a pod, essentially replicating the configuration of existing pods onto a new set of machines.

Incoming documents are partitioned across pods, and consequently, each document must be routed to a single pod. Since each document is processed independently of others, no interaction between pods is necessary in the common case. Document set partitioning thus leads to an embarrassingly parallel workload, and linear scalability. Our implementation monitors the load of each pod, periodically adjusting the fraction of incoming documents directed to each pod to alleviate hot spots.

3.3.2 Filter set partitioning

Our second strategy, which we call *filter set partitioning*, is used to address the memory and CPU limitations of an individual node within a pod. Filter set partitioning is analogous to sharding, declustering, and horizontal partitioning. Since indexing operations are memory intensive, any given node can only index a bounded number of filters. Thus, as we scale up the number of filters in the system, we are forced to partition filters across the nodes within a pod.

Our system supports complex filters composed of a conjunction of predicate expressions. In principle, we could decompose filters into predicates, and partition predicates across nodes. In practice, our implementation uses the simpler approach of partitioning entire filters. As such, a document that arrives at a node can be fully

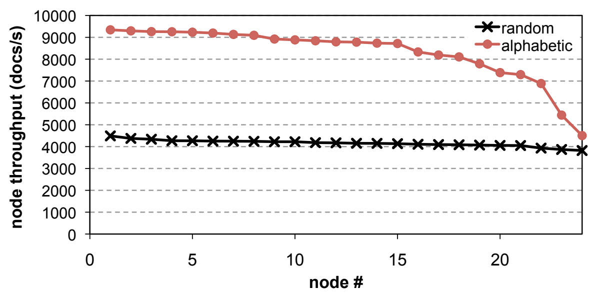

Figure 5: **Node throughput.** This (sorted) graph shows the maximum throughput each node within a pod is capable of sustaining under two different policies: random filter placement, and alphabetic filter placement.

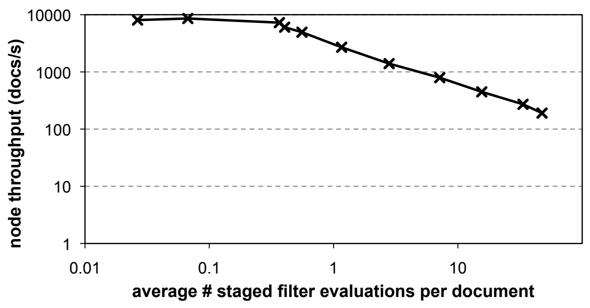

Figure 6: **Staged evaluations vs. throughput.** This graph shows the effect of increasing partial hit rate of a filter set within a node on the maximum document throughput that node is capable of processing.

evaluated against each filter on that node without requiring any cross-node interactions.

Since a document must be evaluated against all filters known by the system, each document arriving at a pod must be transmitted to and evaluated by each node within the pod. Because of this, the document throughput that a pod can sustain is limited by the throughput of the slowest node within the pod.

Two issues substantially affect node and pod throughput. First, a filter partitioning policy that is aware of the indexing algorithms used by nodes can tune the placement of filters to drive up the efficiency and throughput of all nodes. Second, some filters are more expensive to process than others. Particularly expensive filters can induce load imbalances across nodes, driving down overall pod throughput.

Figure 5 illustrates these effects. Using the same Wikipedia workload as before, this graph illustrates the maximum document throughput that each node within a pod of 24 machines is capable of sustaining, under two different filter set partitioning policies. The first policy, *random*, randomly places each filter on a node, while the second policy, *alphabetic*, sorts the substring filters alphabetically by their most selective ngram relaxation. By sorting alphabetically, the second policy causes ngrams that share prefixes to end up on the same node, improving both the memory and computation efficiency of the Aho-Corasick index. The random policy achieves good load balancing but suffers from lower average throughput than alphabetic. Alphabetic exhibits higher average throughput but suffers from load imbalance. From our measurements using the Wikipedia filter set, a 5 million filter index using random placement requires 13.9GB of memory, while a 5 million filter index using alphabetic placement requires 12.1GB, a reduction of 13%.

In Figure 6, we measure the relationship between the number of naïve evaluations that must be executed per document when using staged relaxation and the throughput a node can sustain. As the number of naïve executions increases, throughput begins to drop, until eventually the node spends most of its time performing these evaluations. In practice, the number of naïve evaluations can increase for two reasons. First, a given relaxation can be shared by many independent filters. If the relaxation matches, all associated filters must be executed fully. Second, a given relaxation might match a larger-than-usual fraction of incoming documents. In Section 4.2, we further quantify these two effects and propose strategies for mitigating them.

Given all of these issues, our implementation takes the following strategy to partition filters. Initially, an index-aware partitioning is chosen; for example, alphabetically-sorted prefix grouping is used for substring filters. Filters are packed onto nodes until memory is exhausted. Over time, the load of nodes within each pod is monitored. If load imbalances appear within a pod, then groups of filters are moved from slow nodes to faster nodes to rectify the imbalance. Note that moving filters from one node to another requires the indexes on both nodes to be recomputed. The expense of doing this bounds how often we can afford to rebalance.

A final consideration is that the filter set of an extensible crawler changes over time as new filters are added and existing filters are modified or removed. We currently take a simple approach to dealing with filter set changes: newly added or modified filters are accumulate in "overflow" nodes within each pod and are initially executed naïvely without the benefit of indexing. We then take a generational approach to re-indexing and re-partitioning filters: newly added and modified filters that appear to be stable are periodically incorporated into the non-overflow nodes.

3.4 Additional implementation details

The data path of the extensible crawler starts as documents are dispatched from a web crawler into our system and ends as matching documents are collected from workers and transmitted to client crawler applications (see Figure 1). Our current prototype does not fully explore the design and implementation issues of either the dispatching or collection components.

In our experiments, we use the open source Nutch spider to crawl the web, but we modified it to store documents locally within each crawler node's filesytem rather than storing them within a distributed Hadoop filesystem. We implemented a parallel dispatcher that runs on each crawler node. Each dispatcher process partitions documents across pods, replicates documents across pod nodes, and uses backpressure from nodes to decide the rate at which documents are sent to each pod. Each pod node keeps local statistics about filter matching rates, annotates matching documents with a list of filters that matched, and forwards matching documents to one of a static set of collection nodes.

An interesting configuration problem concerns balancing the CPU, memory, and network capacities of nodes within the system. We ensure that all nodes within a pod are homogeneous. As well, we have provisioned each node to ensure that the network capacity of nodes is not a system bottleneck. Doing so required provisioning each filter processing node with two 1-gigabit NICs. To take advantage of multiple cores, our filter processing nodes use two threads per core to process documents against indexes concurrently. As well, we use one thread per NIC to pull documents from the network and place them in a queue to be dispatched to filter processing threads. We can add additional memory to each node until the cost of additional memory becomes prohibitive. Currently, our filter processing nodes have 3GB of RAM, allowing each of them to store approximately a half-million filters.

Within the extensible crawler itself, all data flows through memory; no disk operations are required. Most memory is dedicated to filter index structures, but some memory is used to queue documents for processing and to store temporary data generated when matching a document against an index or an individual filter.

We have not yet explored fault tolerance issues. Our prototype currently ignores individual node failures and does not attempt to detect or recover from network or switch failures. If a filter node fails in our current implementation, documents arriving at the associated pod will fail to be matched against filters that resided on that node. Note that our overall application semantics are best effort: we do not (yet) make any guarantees to client applications about when any specific web page is crawled. We anticipate that this will simplify fault tolerance issues, since it is difficult for clients to distinguish between failures in our system and the case that a page has not yet been crawled. Adding fault tolerance and strengthening our service guarantees is a potentially challenging future engineering topic, but we do not anticipate needing to invent fundamentally novel mechanisms.

3.5 Future considerations

There are several interesting design and implementation avenues for the extensible crawler. Though they are beyond the scope of this paper, it is worth briefly mentioning a few of them. Our system currently only indexes textual documents; in the future, it would be interesting to consider the impact of richer media types (such as images, videos, or flash content) on the design of the filter language and on our indexing and execution strategy. We currently consider the crawler itself to be a black box, but given that clients already specify content of interest to them, it might be beneficial to allow clients to focus the crawler on certain areas of the Web of particular interest. Finally, we could imagine integrating other streams of information into our system besides documents gathered from a Web crawler, such as real-time "firehoses" produced by systems such as Twitter.

4 Evaluation

In this section, we describe experiments that explore the performance of the extensible crawler, we investigate the effect of different filter partitioning policies. As well, we demonstrate the need to identify and reject non-selective filters. Finally, we present early experience with three prototype Web crawler applications.

All of our experiments are run on a cluster of 8-core, 2GHz Intel Xeon machines with 3GB of RAM, dual gigabit NICs, and a 500 GB 7200-RPM Barracuda ES SATA hard drive. Our systems are configured to run 32bit Linux kernel version 2.6.22.9-91.fc7, and to use Sun's 23 bit JVM version 1.6.0_12 in server mode. Unless stated otherwise, the filter workload for our experiments consists of 9,204,600 unique sentences extracted from Wikipedia; experiments with relaxation and staging used 32 byte prefix ngrams extracted from the filter sentences.

For our performance oriented experiments, we gathered a 3,349,044 Web document crawl set on August 24th, 2008 using the Nutch crawler and pages from the DMOZ open directory project as our crawl seed. So that our experiments were repeatable, when testing the performance of the extensible crawler we used on a custom tool to stream this document set at high throughput, rather than re-crawling the Web. Of the 3,349,044 documents in our crawl set, 2,682,590 contained textual content, including HTML and PDF files; the rest contain binary content, including images and executables. Our extensible crawler prototype does not yet notify wide-area clients about matching documents; instead, we gather statistic about document matches, but drop the matching documents instead of transmitting them.

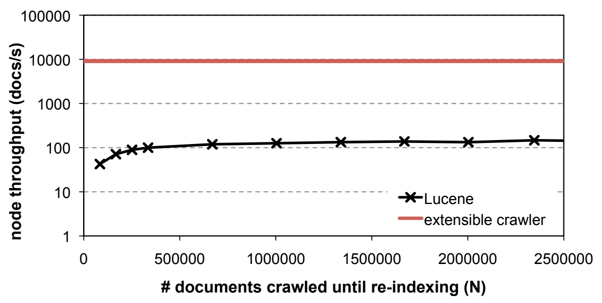

Figure 7: **Lucene vs. extensible crawler.** This graph compares the document processing rates of a single-node extensible crawler and a single-node Lucene search engine. The x-axis displays the number of documents crawled between reconstructions of the Lucene index. Note that the y-axis is logarithmically scaled.

4.1 Nutch vs. the extensible crawler

In Section 2.1, we described architectural, workload, and expected performance differences between the extensible crawler and an alternative implementation of the service based on a conventional search engine. To demonstrate these differences quantitatively, we ran a series of experiments directly comparing the performance of our prototype to an alternative implementation based on the Lucene search engine, version 2.1.0 [7].

The construction of a Lucene-based search index is typically performed as part of a Nutch map-reduce pipeline that crawls web pages, stores them in the HDFS distributed filesystem, builds and stores an index in HDFS, and then services queries by reading index entries from HDFS. To make the comparison of Lucene to our prototype more fair, we eliminated overheads introduced by HDFS and map-reduce by modifying the system to store crawled pages and indexes in nodes' local filesystems. Similarly, to eliminate variation introduced by the wide-area Internet, we spooled our pre-crawled Web page data set to Lucene's indexer or to the extensible crawler over the network.

The search engine implementation works by periodically constructing an index based on the N most recently crawled web pages; after constructing the index and partitioning it across nodes, each node evaluates the full filter set against its index fragment. The implementation uses one thread per core to evaluate filters. By increasing N, the implementation indexes less frequently, reducing overhead, but suffers from a larger latency between the downloading of a page by the crawler and the evaluation of the filter set against that page. In contrast, the extensible crawler implementation constructs and index over its filters once, and then continuously evaluates pages against that index.

In Figure 7, we compare the single node throughput of the two implementations, with 400,000 filters from the

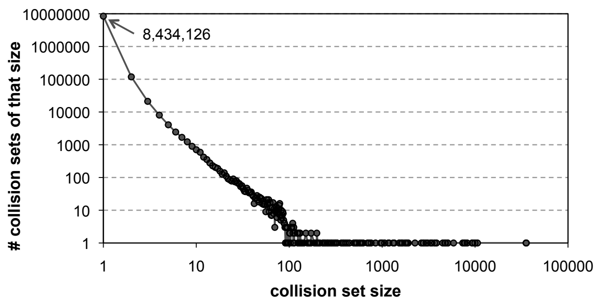

Figure 8: **Collision set size.** This histogram shows the distribution of collision set sizes when using 32-byte ngrams over the Wikipedia filter set.

Wikipedia workload. Note that the x-axis corresponds to N, but this parameter only applies to the Lucene crawler. The extensible crawler implementation has nearly two orders of magnitude better performance than Lucene; this is primarily due to the fact that Lucene must service queries against its disk-based document index, while the extensible crawler's filter index is served out of memory. As well, the Lucene implementation is only able to achieve asymptotic performance if it indexes batches of $N > 1,000,000$ documents.

Our head-to-head comparison is admittedly still unfair, since Lucene was not optimized for fast, incremental, memory-based indexing. Also, we could conceivably bridge the gap between the two implementations by using SSD drives instead of spinning platters to store and serve Lucene indexes. However, our comparison serves to demonstrate some of the design tensions between conventional search engines and extensible crawlers.

4.2 Filter partitioning and blacklisting

As mentioned in Section 3.3.2, two different aspects of a filter set contribute to load imbalances between otherwise identical machines: first, a specific relaxation might be shared by many different filters, causing a partial hit to result in commensurately many naïve filter executions, and second, a given relaxation might match a large number of documents, also causing a large number of naïve filter executions. We now quantify these effects.

We call the a set of filters that share an identical relaxation a *collision set*. A collision set of size 1 implies the associated filter's relaxation is unique, while a collision set of size N implies that N filters share a specific relaxation. In Figure 8, we show the distribution of collision set sizes when using a 32-byte prefix relaxation of the Wikipedia filter set. The majority of filters (8,434,126 out of 9,204,600) have a unique relaxation, but some relaxations collide with many filters. For example, the largest collision set size was 35,585 filters. These filters all shared the prefix "The median income for a household in the"; this sentence is used in many Wikipedia

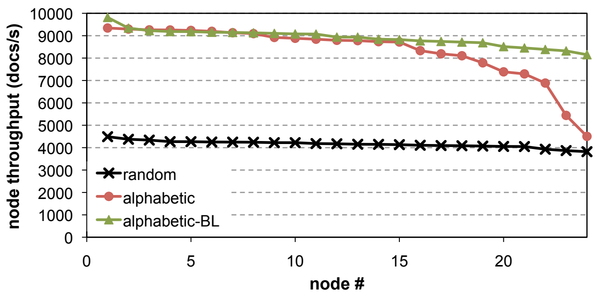

Figure 9: **Node throughput.** This (sorted) graph shows the maximum throughput each node within a pod is capable of sustaining under three different policies: random filter placement, alphabetic filter placement, and alphabetic filter placement with blacklisting.

articles describing income in cities, counties, and other population centers. If a document matches against this relaxation, the extensible crawler would need to naïvely execute all 35,585 filters.

Along a similar vein, some filter relaxations will match a larger-than-usual fraction of documents. One notably egregious example from our Web malware detection application was a filter whose relaxation contained was the 32-character sequence `<meta http-equiv="Content-Type">`. Unsurprisingly, a very large fraction of Web pages contain this substring!

To deal with these two sources of load imbalance, our implementation blacklists specific relaxations. If our implementation notices a collision set containing more than 100 filters, we blacklist the associated relaxation, and compute alternate relaxations for those filters. In the case of our Wikipedia filter set, this required modifying the relaxation of only 145 filters. As well, if our implementation notices that a particular relaxation has an abnormally high document partial hit rate, that "hot" relaxation is blacklisted and new filter relaxations are chosen.

Blacklisting rectifies these two sources of load imbalance. Figure 9 revisits the experiment previously illustrated in Figure 5, but with a new line that shows the effect of blacklisting on the distribution of document processing throughputs across the nodes in our cluster. Without blacklisting, alphabetic filter placement demonstrates significant imbalance. With blacklisting, the majority of the load imbalance is removed, and the slowest node is only 17% slower than the fastest node.

4.3 Experiences with Web crawler applications

To date, we have prototyped three Web crawler applications: vanity alters that detect pages containing a user's name, a copyright detection application that finds Web objects that match ClamAV's malware signature database, and a copyright violation detection service that looks for pages containing copies of Reuters or

		Vanity alerts	Copyright violation	Malware detection
# filters		10,622	251,647	3,128
relaxation only	doc hit rate	68.98%	0.664%	45.38%
	false +ves per doc	15.76	0.0386	0.851
	throughput (docs/s)	7,244	8,535	8,534
	# machines needed	13.80	11.72	11.72
with relaxation and staging	doc hit rate	13.1%	0.016%	0.009%
	throughput (docs/s)	592	8,229	6,354
	# machines needed	168.92	12.15	15.74

Table 3: **Web crawler application features and performance.** This table summarizes the high-level workload and performance features of our three prototype Web crawler applications.

Wikipedia articles. Table 3 summarizes the high-level features of these applications and their filter workloads; we now discuss each in turn, relating additional details and anecdotes.

4.3.1 Vanity alerts

For our vanity filter application, we authored 10,622 filters based on names of university faculty and students. Our filters were constructed as regular expressions of the form `''first.{1,20}last''`, i.e., the user's first name followed by their last name, with the constraint of no more than 20 characters separating the two parts. The filters were first relaxed into a conjunct of substring 32-grams, and from there the longest substring conjunct was selected as the final relaxation of the filter.

This filter set matched against 13.1% of documents crawled. This application had a modest number of filters, but its filter set nonetheless matched against a large fraction of Web pages, violating our assumption that a crawler application should have highly selective filters. Moreover, when using relaxed filters without, there were many additional false partial hits (an average of 15.76 per document, and an overall document hit rate of 69%). Most false hits were due to names that are contained in commonly found words, such as Tran, Chang, or Park.

The lack of high selectivity of its filter set leads us to conclude that this application is not a good candidate for the extensible crawler. If millions of users were to use this service, most Web pages would likely match and need to be delivered to the crawler application.

4.3.2 Copyright violation detection

For our second application, we prototyped a copyright violation detection service. We evaluated this application by constructing a set of 251,647 filters based on 30,534 AP and Reuters news articles appearing between July and October of 2008. Each filter was a single sentence extracted from an article, but we extracted multiple filters from each article. We evaluated the resulting filter set against a crawl of 3.68 million pages.

Overall, 590 crawled documents (0.016%) matched against the AP/Reuters filter set, and 619 filters (0.028% of the filter set) were responsible for these matches. We manually determined that most matching documents that matched were original news articles or blogger pages that quoted sections of articles with attribution. We did find some sites that appeared to contain unauthorized copies of entire news stories, and some sites that plagiarized news stories by integrating the story body but replacing the author's byline.

If a document hit, it tended to hit against a single filter (50% of document hits were for a single filter). A smaller number of documents hit against many sentences (13% of documents matched against more than 6 filters). Documents that matched against many filters tended to contain full copies of the original news story, while documents that match a single sentence tended to contain boilerplate prose, such as a specific author's byline, legal disclosures, or common phrases such as "The officials spoke on condition of anonymity because they weren't authorized to release the information."

4.3.3 Web malware detection

The final application we prototyped was Web malware detection. We extracted 3,128 text-centric regular expressions from the February 20, 2009 release of the ClamAV open-source malware signature database. Because many of these signatures were designed to match malicious JavaScript or HTML, some of their relaxations contain commonly occurring substrings, such as `<a href=``http://''`. As a result, blacklisting was a particularly important optimization for this workload; the system was always successful at finding suitably selective relaxations of each filter.

Overall, this filter set matched 342 pages from the same crawl of 3.68 million pages, with an overall rate of 0.009%. The majority of hits (229) were for two similar signatures that capture obfuscated JavaScript code that emits an iframe in the parent page. We examined all of the pages that matched this signature; in each case, the iframe contained links to other pages that are known to contain malicious scripts. Most of the matching pages appeared to be legitimate business sites that had been compromised. We also found several pages that matched a ClamAV signature designed to detect Web bugs.

In addition to genuinely malicious Web pages, we found a handful of pages that were application-level false positives, i.e., they correctly matched a ClamAV filter, but the page did not contain the intended attack. Some of these application-level false positives contained blog entries discussing virulent spam, and the virulent spam itself was represented in the ClamAV database.

5 Related work

The extensible crawler is related to several classes of systems: Web crawlers and search engines, publish-subscribe systems, packet filtering engines, parallel and streaming databases, and scalable Internet content syndication protocols. We discuss each in turn.

Web crawlers and search engines. The engineering issues of high-throughput Web crawlers are complex but well understood [21]. Modern Web crawlers can retrieve thousands of Web pages per second per machine. Our work leverages existing crawlers, treating them as a black box from which we obtain a high throughput document stream. The Mercator project explored the design of an extensible crawler [20], though Mercator's notion of extensibility is different than ours: Mercator has well-defined APIs that simplify the job of adding new modules that extend the crawler's set of network protocols or type-specific document processors. Our extensible crawler permits remote third parties to dynamically insert new filters into the crawling pipeline.

Modern Web search engines require complex engineering, but the basic architecture of a scalable search engine has been understood for more than decade [8]. Our extensible crawler is similar to a search engine, but inverted, in that we index queries rather than documents. As well, we focus on in-memory indexing for throughput. Cho and Rajagopalan described a technique for supporting fast indexing of regular expressions by reducing them to ngrams [12]; our notion of filter relaxation is a generalization of their approach.

Though the service is now discontinued, Amazon.com offered programmatic search access to a 300TB archive containing 4 billion pages crawled by Alexa Internet [5] and updated daily. By default, access was restricted to queries over a fixed set of search fields, however, customers could pay to re-index the full data set over custom fields. In contrast, the extensible crawler permits customers to write custom filters over any attribute supported by our document extractors, and since we index filters rather than pages, our filters are evaluated in real-time, at the moment a page is crawled.

Publish-subscribe systems. The extensible crawler can be thought of as a content-based publish-subscribe system [22] designed and optimized for a real-time Web crawling workload. Content-based pub-sub systems have been explored at depth, including in the Gryphon [2], Siena [9], Elvin [31], and Le Subscribe [16, 27] projects. Many of these projects explore the trade-off between filter expressiveness and evaluation efficiency, though most have a wide-area, distributed event notification context in mind. Le Subscribe is perhaps closest to our own system; their language is also a conjunction of predicates, and like us, they index predicates in main-memory for scalable, efficient evaluation. In contrast to these previous projects, our work explores in depth the partitioning of documents and filters across machines, the suitability of our expression language for Web crawling applications, the impact of disproportionately high hit rate filters, and evaluates several prototype applications.

Web-based syndication protocols, such as RSS and Atom, permit Web clients to poll servers to receive feeds of new articles or document elements. Cloud-based aggregation and push notification services such as rssCloud and PubSubHubbub allow clients to register interest in feeds and receive notifications when updates occur, relieving servers from pull-induced overload. These services are roughly equivalent to channel-based pub-sub systems, whereas the extensible crawler is more equivalent to a content-based system.

The Google alerts system [18] allows users to specify standing search queries to be evaluated against Google's search index. Google alerts periodically emails users newly discovered search results relevant to their queries. Alerts uses two different approaches to gather new results: it periodically re-executes queries against the search engine and filters previously returned results, and it continually matches incoming documents against the body of standing user queries. This second approach has similarities to the extensible crawler, though details of Google alert's architecture, workload, performance, and scalability have not been publicly disclosed, preventing an in-depth technical comparison.

Cobra [29] perhaps most similar to our system. Cobra is a distributed system that crawls RSS feeds, evaluates articles against user-supplied filters, and uses reflectors to distributed matching articles to interested users. Both Cobra and the extensible crawler benefit from a filter language design to facilitate indexing. Cobra focused on issues of distribution, provisioning, and network-aware clustering, whereas our work focuses on a single-cluster implementation, efficiency through filter relaxation and staging, and scalability through document and filter set partitioning. As well, Cobra was oriented towards scalable search and aggregation of Web feeds, whereas the extensible crawler provides a platform for more widely varied crawling applications, such as malware and copyright violation detection.

Packet filters and NIDS. Packet filters and network intrusion detection systems (NIDS) have similar challenges as the extensible crawler: both classes of systems must process a large number of filters over a high bandwidth stream of unstructured data with low latency. The BSD packet filter allowed control-flow graph filters to be compiled down to an abstract filtering machine, and executed safely and efficiently in an OS kernel [25]. Packet filtering systems have also confronted the problem of efficiently supporting more expressive filters, while preventing state space explosion when representing large filter sets as DFAs or NFAs [23, 32]. Like an extensible crawler, packet filtering systems suffer from the problem of normalizing documents content before matching against filters [30], and of providing additional execution context so that byte-stream filters can take advantage of higher-level semantic information [34]. Our system can benefit from the many recent advances in this class of system, though our applications require orders of magnitude more filters and therefore a more scalable implementation. As well, our application domain is more robust against false positives.

Databases and SDIs. The extensible crawler shares some design considerations, optimizations, and implementation techniques with parallel databases such as Bubba [13] and Gamma [14], in particular our need to partition filters (queries) and documents (records) over machines, and our focus on high selectivity as a path to efficiency. Our workload tends to require many more concurrent filters, but does not provide the same expressiveness as SQL queries. We also have commonalities with streaming database systems and continuous query processors [1, 10, 11], in that both systems execute standing queries against an infinite stream of data. However, streaming database systems tend to focus on semantic issues of queries over limited time windows, particularly when considering joins and aggregation queries, while we focus on scalability and Web crawling applications.

Many databases support the notion of triggers that fire when matching records are added to the database. Prior work has examined indexing techniques for efficiently supporting a large number of such triggers [19].

Selective Dissemination of Information (SDI) systems [35], including those that provide scalable, efficient filtering of XML documents [4], share our goal of executing a large number filters over semi-structured documents, and rely on the same insight of indexing queries to match against individual documents. These systems tend to have more complex indexing schemes, but have not yet been targeted at the scale, throughput, or application domain of the extensible crawler.

6 Conclusions

This paper described the design, prototype implementation, and evaluation of the *extensible crawler*, a service that crawls the Web on behalf of its many client applications. Clients extend the crawler by injecting filters that identify pages of interest to them. The crawler continuously fetches a stream of pages from the Web, simultaneously executes all clients' filters against that stream, and returns to each client those pages selected by its filter set.

An extensible crawler provides several benefits. It relieves clients of the need to operate and manage their own private crawler, greatly reducing a client's bandwidth and computational needs when locating pages of interest. It is efficient in terms of Internet resources: a crawler queries a single stream of Web pages on behalf of many clients. It also has the potential for crawling highly dynamic Web pages or real-time sources of information, notifying clients quickly when new or interesting content appears.

The evaluation of XCrawler, our early prototype system, focused on scaling issues with respect to its number of filters and crawl rate. Using techniques from related work, we showed how we can support rich, expressive filters using relaxation and staging techniques. As well, we used microbenchmarks and experiments with application workloads to quantify the impact of load balancing policies and confirm the practicality of our ideas. Overall, we believe that the low-latency, high selectivity, and scalable nature of our system makes it a promising platform for many applications.

Acknowledgments

We thank Tanya Bragin for her early contributions, Raphael Hoffman for his help in gathering the Wikipedia filter set, and Paul Gauthier and David Richardson for their helpful feedback on drafts of this paper. We also thank our anonymous reviews and our shepherd, Stefan Savage, for their guidance. This work was supported in part by an NDSEG Fellowship, by the National Science Foundation under grants CNS-0132817, CNS-0430477 and CNS-0627367, by the Torode Family Endowed Career Development Professorship, by the Wissna-Slivka Chair, and by gifts from Nortel Networks and Intel Corporation.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '99)*, Atlanta, GA, 1999.

[3] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[4] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*, San Francisco, CA, 2000.

[5] Amazon Web Services. Alexa web search platform. `http://www.amazon.com/b?ie=UTF8&node=16265721`.

[6] E. Amitay, D. Carmel, M. Herscovici, R. Lempel, and A. Soffer. Trend detection through temporal link analysis. *Journal of the American Society for Information Science and Technology*, 55(14):1270–1281, December 2004.

[7] Apache Software Foundation. Apache lucene. `http://lucene.apache.org/`.

[8] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on the World Wide Web (WWW7)*, Brisbane, Australia, 1998.

[9] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '00)*, Portland, OR, 2000.

[10] J. Chen, D. J. Dewitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, May 2000.

[11] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR '03)*, Asilomar, CA, January 2003.

[12] J. Cho and S. Rajagopalan. A fast regular expression indexing engine. In *Proceedings of the 18th International Conference on Data Engineering (ICDE '02)*, San Jose, CA, February 2002.

[13] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. *ACM SIGMOD Record*, 17(3):99–108, 1988.

[14] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[15] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Web-scale information extraction in KnowItAll. In *Proceedings of the 13th International Conference on the World Wide Web*, New York, NY, May 2004.

[16] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA, May 2001.

[17] R. Geambasu, S. D. Gribble, and H. M. Levy. CloudViews: Communal data sharing in public clouds. In *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud '09)*, San Diego, CA, June 1999.

[18] Google. Google alerts. `http://www.google.com/alerts`.

[19] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. Park, and A. Vernon. Scalable trigger processing. In *Proceedings of the 15th International Conference on Data Engineering*, Sydney, Austrialia, March 1999.

[20] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.

[21] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov. IRLbot: Scaling to 6 billion pages and beyond. In *Proceedings of the 17th International World Wide Web Conference*, Beijing, China, April 2008.

[22] Y. Liu and B. Plale. Survey of publish subscribe event systems. Technical Report TR574, Indiana University, May 2003.

[23] A. Majumder, R. Rastogi, and S. Vanama. Scalable regular expression matching on data streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Vancouver, Canada, June 2008.

[24] McAfee, Inc. McAfee SiteAdvisor. `http://www.siteadvisor.com/`.

[25] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference*, San Diego, California, January 1993.

[26] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A crawler-based study of spyware on the Web. In *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS '06)*, San Diego, CA, February 2006.

[27] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *Proceedings of the 7th International Conference on Cooperative Information Systems (CoopIS 2000)*, Eilat, Israel, September 2000.

[28] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[29] I. Rose, R. Murty, P. Pietzuch, J. Ledlie, M. Roussopoulos, and M. Welsh. Cobra: Content-based filtering and aggregation of blogs and RSS feeds. In *Proceedings of the 4th USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2007)*, Cambridge, MA, April 2007.

[30] S. Rubin, S. Jha, and B. P. Miller. Protomatching network traffic for high throughput network intrusion detection. In *Proceedings of the 13th ACM conference on Computer and Communications Security (CCS '06)*, October 2006.

[31] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with Elvin4. In *Proceedings of the 2000 AUUG Annual Conference (AUUG2K)*, Canberra, Australia, June 2000.

[32] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (Oakland '08)*, Oakland, CA, May 2008.

[33] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: Exploring photo collections in 3D. In *Proceedings of the 33rd International Conference an Exhibition on Computer Graphics and Interactive Techniques (SIGGRAPH '06)*, Boston, MA, July 2006.

[34] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the 10th ACM conference on Computer and Communications Security (CCS '03)*, October 2003.

[35] T. W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Transactions on Database Systems*, 19(2):332–364, June 1994.