Detecting Quilted Web Pages at Scale

Marc Najork
Microsoft Research
1065 La Avenida
Mountain View, CA 94043, USA
najork@microsoft.com

ABSTRACT

Web-based advertising and electronic commerce, combined with the key role of search engines in driving visitors to ad-monetized and e-commerce web sites, has given rise to the phenomenon of web spam: web pages that are of little value to visitors, but that are created mainly to mislead search engines into driving traffic to target web sites. A large fraction of spam web pages is automatically generated, and some portion of these pages is generated by stitching together parts (sentences or paragraphs) of other web pages. This paper presents a scalable algorithm for detecting such "quilted" web pages. Previous work by the author and his collaborators introduced a sampling-based algorithm that was capable of detecting some, but by far not all quilted web pages in a collection. By contrast, the algorithm presented in this work identifies all quilted web pages, and it is scalable to very large corpora. We tested the algorithm on the half-billion page English-language subset of the ClueWeb09 collection, and evaluated its effectiveness in detecting web spam by manually inspecting small samples of the detected quilted pages. This manual inspection guided us in iteratively refining the algorithm to be more efficient in detecting real-world spam.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Clustering, Information Filtering

General Terms

Algorithm, Experimentation, Measurement

Keywords

Web spam detection, phrase-level duplication, scalable algorithms

SIGIR'12, August 12–16, 2012, Portland, Oregon, USA.

1. INTRODUCTION

Over the past 20 years, the World Wide Web has evolved from a way for scientists to exchange data and ideas to become the general population's medium of choice for information, communication, entertainment, and commercial transactions. Web-mediated electronic commerce in the United States is projected to reach close to $ 200 billion in 2011 [1]. Search engines such as Google and Bing form a key component of this web-based ecosystem, in that they enable users to locate content and services. The explosive rise in the popularity of the web and the volume of e-commerce, together with the key role of search engines, has given rise to the phenomenon of web spam: web content that is of little value to users, but created mainly to drive search engine traffic to particular target sites. Web spam creates a negative experience for search users, lowering the utility they derive from the search service. Consequently, web spam detection is an active area of research, both in search engines and in academia.

A substantial fraction of spam web pages is automatically generated, and some of that content is constructed out of all or parts of other web pages. In this paper, we introduce a technique for detecting all such web pages in a given collection, exhaustively and at large scale, using data-parallel infrastructure. We coin the term *quilted web page* to refer to a page that is "stitched" together out of "patches" taken from multiple other web pages. Figure 1 shows an example of such a quilted page (though arguably not spammy in nature) together with three of the source web pages that each donated a patch of words. The notion of quilted web page is congruent with our an earlier notion of *phrase-level replication* [9], but more constrained than the notion of *local text reuse* [15] introduced by Seo and Croft, which refers to portions of text shared between two documents, but does not insist on a large fraction of a document consisting of patches of multiple other documents.

The notion of textual similarity is at the heart of Information Retrieval, and there is a wide gamut of similarity-based problems, ranging from the retrieval problem (finding those documents in a collection that contain the – typically few – terms of a query) over local text reuse detection to the other extreme of near-duplicate detection (clustering a collection into classes of highly similar documents). The problem addressed in this paper is situated in the middle of the gamut (along with local text reuse detection), but it borrows ideas from the extreme end of the gamut, namely near-duplicate detection, and in particular *shingling*.

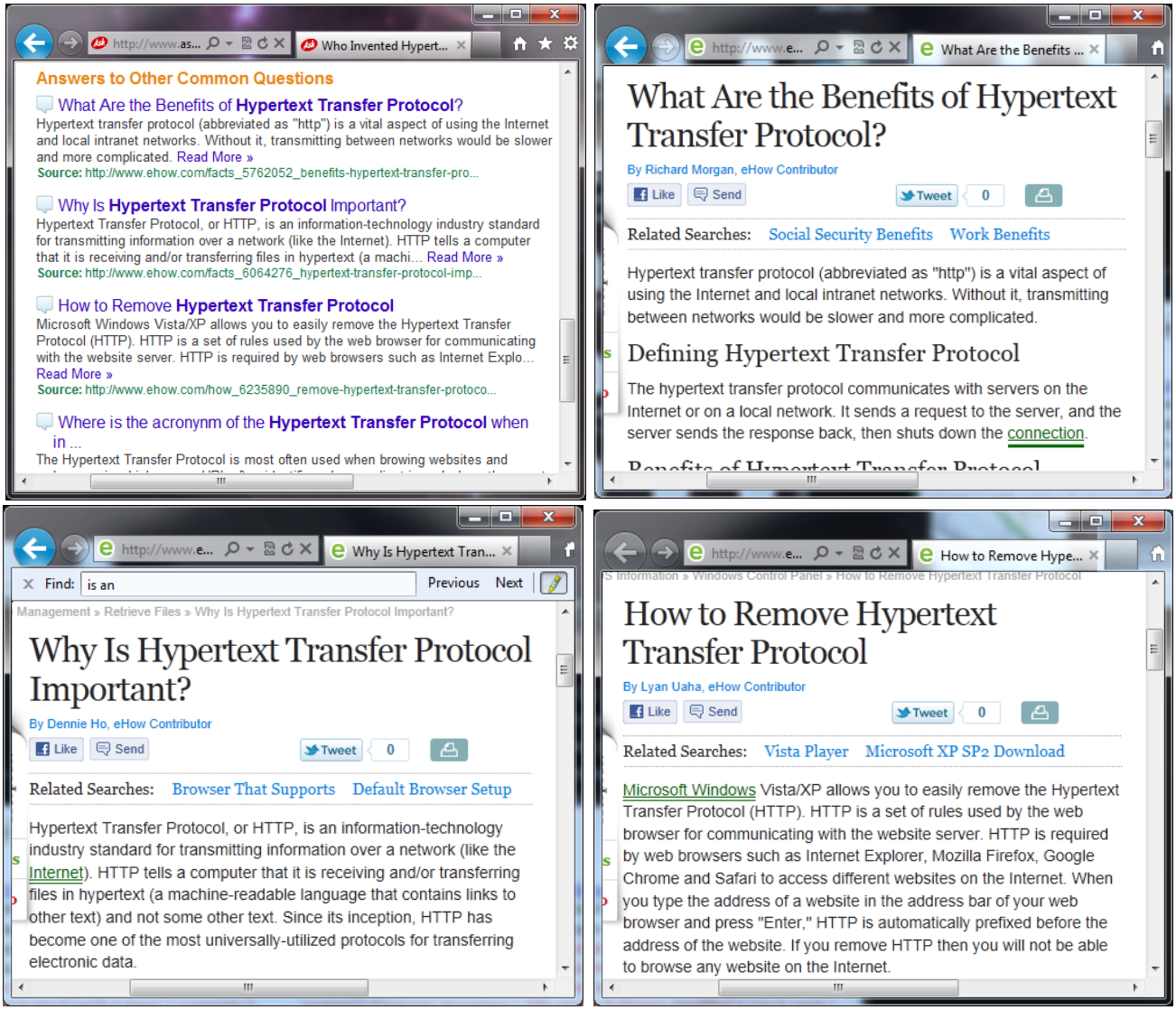

Figure 1: At the top left, a quilted web page, and around it, three of its source web pages.

The core idea of the seminal *shingling algorithm*, due to Broder *et al.*, is to segment each document into overlapping k-grams called *shingles* (drawing on the image of the overlapping shingles of a roof), to compact the shingles by hashing them, and to draw a consistent sample from the set of hashed shingles. In an early version of the algorithm [6], this was achieved by retaining all hash values h of a document where $h \bmod p = 0$ (with p fixed in advance), resulting in a variable-length vector of *shingleprints*. A refined version [5] employed randomly selected permutations and retained for each document those hash values that are minimal among all the document's hashes under a permutation, resulting in a fixed-sized shingleprint. Either scheme greatly reduces the amount of data required to represent each document, but neither addresses the time-complexity issues of document clustering, naive pair-wise comparison having quadratic complexity. A subsequent refinement (see [8] for a detailed description) combines groups of shingles into *megashingles* such that any two documents sharing a megashingle are with high probability near-duplicates of one another, thereby making it possible to cluster a set of documents into near-duplicate sets in linear time. Shingling is a purely *syntactic* technique; it does not presume any knowledge of the semantics of the document, and does not involve any language models. A competing approach to shingling, Charikar's locality-sensitive hashing scheme [7], is similarly syntactic in nature: Documents are tokenized through some extraneous mechanism (e.g. simple word-breaking or segmentation into overlapping k-grams), resulting in feature vectors that are viewed as points in a very high-dimensional space. A set of k hyperplanes in that high-dimensional space is fixed ahead of time. The algorithm determines for each point (document) and each hyperplane, what side of the hyperplane the point falls on, producing a bit per hyperplane and resulting in a k-bit hash value. Near-duplicate documents have, with high proba-

bility, highly similar hash values. The interested reader is referred to Henzinger's experimental comparison [10] of Broder's and Charikar's algorithms. Like these two algorithms, the algorithm described in this paper is purely syntactic. Moreover, it uses Broder's idea of segmenting each document into a sequence of overlapping k-grams.

There are multiple works addressing the middle portion of the gamut, detecting phrase-level replication. Fetterly, Manasse and Najork [9] attacked the problem by using core ideas of shingling – segmenting each document into overlapping k-grams and then performing a sequence of data reduction steps: hashing k-grams, collapsing near-duplicate documents (through shingling proper), eliminating popular k-grams, sampling the hashed k-grams of each document, and finally constructing sets of documents covering the surviving hashes of each document. Due to the sampling-based approach, their algorithm identifies some, but by no means all of the quilted pages in a collection, and some but not all of the "patch sources" of each quilted page. The algorithm described in this paper resembles our earlier attempt in many ways, but it is not probabilistic: given a collection, it will identify *all* quilted pages and *all* their source pages.

Baeza-Yates, Pereira and Ziviani [2] attacked the related problem of constructing a genealogical tree of web pages – deducing which web pages are derived from which other web pages – by segmenting pages into three-sentence shingles (as opposed to the more common work-level shingling), collapsing duplicate documents in a pre-processing stage, and finally deducing parent-child relationships between pages using a greedy algorithm. Their work is similar to ours in that it is not sampling-based, but differs in the choice of fingerprinting unit (three-sentence units vs. k-word units), the corpus size it was applied to (14 million vs. 503 million pages), and the underlying motivation.

Seo and Croft [15] describe an algorithm for detecting *local text reuse*, a document containing a patch of text taken from another document. Their algorithm (like both our earlier [9] and our current algorithm) is purely syntactic; documents are viewed as simple sequences of terms. It segments each document into non-overlapping sequences of terms using a technique called *hash-breaking*, and uses another technique called DCT (discrete-cosine transform) fingerprinting to make the algorithm less sensitive to small modifications of text. The algorithm outputs pairs of documents sharing a patch of text; unlike our algorithm it is not aimed at identifying quilted documents incorporating portions from multiple sources. Seo and Croft evaluated their algorithm empirically, and found precision and recall of text reuse to be good. It is difficult to ascertain the scalability of their approach. Their largest experiment processed a corpus of about 3 million blog posts; by comparison, the algorithm proposed in this paper was tested on a collection of over half a billion web pages.

More recently, Bendersky and Croft took [3] took another stab at the problem of detecting text reuse. Their approach is aimed at finding reuse in web-scale collections, and it assumes the existence of a retrieval system for that collection (say, a search engine). The algorithm does not aim to detect all instances of text reuse; rather, it takes a set of topical statements as input and aims to find all instances of text reuse related to these topics (leveraging the search engine to retrieve candidate documents for each topic). Another important difference to the previously described approaches is that theirs is not purely syntactic – it does involve language models, query likelihoods, and temporal information.

Another related line of work is that of Kolak and Schilit, who studied the problem of identifying quotations (short pieces of text excerpted from other documents) in a large collection of documents, namely the Google Books corpus. The underlying objective was to treat quotations as a form of "link" from the quoting to the quoted document. Like our past and present work, their algorithm borrows key ideas from shingling: documents are segmented into overlapping k-grams which are then hashed (core ideas of shingling), and hashes are added to an index to build up a collection of repeated text sequences. Their algorithm has been implemented in the MapReduce framework and appears to be highly scalable. While Kolak and Schilit's algorithm has many commonalities with the algorithm described in this paper, the two algorithms do compute different things: their algorithm identifies quotations (portions of text shared between documents) while ours identifies quilted web pages (web pages that consist largely of portions of other web pages).

Finally, it is worth pointing out that the problem of identifying quilted web pages is quite similar to that of identifying plagiarism. There exists a large body of work on plagiarism detection; two exemplary systems include COPS [4] and SCAM [16]. The interested reader is further referred to a study by Hoad and Zobel [11] comparing the effectiveness of various algorithmic approaches to plagiarism detection.

The remainder of the paper is structured as follows: Section 2 introduces a new algorithm for detecting quilted web pages, first by providing an intuitive explanation, then by framing the algorithm in a set-theoretic setting, and finally by sketching its implementation in DryadLINQ [17], a platform for data-parallel cluster computing that leverages LINQ, Microsoft's "language-integrated query" extension to the C# programming language. LINQ is based on list comprehensions, which in turn are closely related to set comprehensions, making the leap from specification to implementation quite small. Section 3 describes the experimental validation of our algorithm. It provides details on the data set and the computational infrastructure used in this study, and describes the runs we performed. We measured the effectiveness of the quilt detection algorithm in identifying spam web pages by manually inspecting a small sample of detected quilted pages (together with the source pages that provided the "patches" for each "quilt"). While (per definition) every detected page was indeed a quilt, only a small fraction of these pages was considered spam. Analyzing the nature of these false-positives guided us in refining our algorithm with several heuristics, which greatly reduced the rate of false positives. Finally, section 4 offers some concluding remarks and avenues for future research.

2. A SCALABLE QUILT DETECTION ALGORITHM

Our quilt detection algorithm takes a corpus of web pages as input, and outputs the set of all "quilted" web pages, together with the source web pages providing the textual "patches" that went into the quilt. Rather than considering only proper phrases as potential patches, we segment each n-word document into $n - k + 1$ overlapping k-grams. Discarding common k-grams that occur in more than m doc-

uments as well as unique ones that occur in only one document leaves us with a set of candidate *patch grams*. We consider a page to be *quilted* iff at least a θ fraction of its k-grams are patch grams, and if the minimum cover set of documents from which the patch grams are drawn contains at least c documents. So, our definition of quilted page involves the four parameters k, m, c and θ. The choice of these parameters impacts both the effectiveness (what fraction of quilted pages would be considered spam by human judges) and the efficiency (how long it takes to process the corpus) of the detection algorithm.

More formally, we consider a corpus D of documents (web pages). Each document $d \in D$ is a sequence of terms (words) $\langle t_1 \cdots t_n \rangle$, which can be grouped into $n-k+1$ overlapping k-grams, with $g_1 = \langle t_1 \cdots t_k \rangle$, $g_2 = \langle t_2 \cdots t_{k+1} \rangle$, etc. We write $grams_k(d)$ to denote the k-gram-set of document d, and $docs_k(g)$ to denote $\{d \in D : g \in grams_k(d)\}$, the set of documents containing k-gram g. We define the patch gram set of a document d (those k-grams of d that co-occur in at least one but fewer than m other documents) as follows:

$$pgrams_{k,m}(d) = \{g \in grams_k(d) : 1 < |docs_k(g)| \leq m\}$$

Obviously, $pgrams_{k,m}(d) \subseteq grams_k(d)$ for all d, k and m. We define the patch-fraction of a document d as the fraction of patch grams in the document's gram set:

$$patchfrac_{k,m}(d) = \frac{|pgrams_{k,m}(d)|}{|grams_k(d)|}$$

Moreover, we define $sources_{k,m}(d)$, the source documents contributing patch grams to a document d, as follows:

$$sources_{k,m}(d) = \text{minimal set } D' \subseteq D \text{ such that } pgrams_{k,m}(d) \subseteq \bigcup_{d' \in D'} pgrams_{k,m}(d')$$

In other words, $sources_{k,m}(d)$ is a (not necessarily unique) minimal set of documents whose combined patch grams cover all the patch grams of d. The set cover problem is NP-hard [12], but there is a well-known greedy approximation algorithm, detailed below. Using this notation, we define quilted web pages as follows:

Definition: A document d is said to be (k, m, c, θ)-*quilted* iff $patchfrac_{k,m}(d) \geq \theta$ and $|sources_{k,m}(d)| \geq c$.

Before describing the approximation algorithm for computing $sources_{k,m}(d)$, we first provide a few theorems concerning the role of the parameters in the definition of quiltedness:

Theorem 1 (Monotonicity in θ): A document that is (k, m, c, θ)-quilted is also (k, m, c, θ')-quilted for any $\theta' \leq \theta$.

Proof: Straightforward. For any document d, d is (k, m, c, θ)-quilted implies $|sources_{k,m}(d)| \geq c$ and $patchfrac_{k,m}(d) \geq \theta$, which implies $patchfrac_{k,m}(d) \geq \theta'$ since $\theta \geq \theta'$, which implies d is (k, m, c, θ')-quilted. □

Theorem 2 (Monotonicity in c): A document that is (k, m, c, θ)-quilted is also (k, m, c', θ)-quilted for any $c' \leq c$.

Proof: Straightforward. For any document d, d is (k, m, c, θ)-quilted implies $patchfrac_{k,m}(d) \geq \theta$ and $|sources_{k,m}(d)| \geq c$, which implies $|sources_{k,m}(d)| \geq c'$ since $c \geq c'$, which implies d is (k, m, c', θ)-quilted. □

Theorem 3 (Monotonicity in m): A document that is (k, m, c, θ)-quilted is also (k, m', c, θ)-quilted for any $m' \geq m$.

Proof: Assume $m' \geq m$, and consider a (k, m, c, θ)-quilted document d, so $patchfrac_{k,m}(d) \geq \theta$ and $|sources_{k,m}(d)| \geq c$. It is easy to see that $pgrams_{k,m'}(d) \supseteq pgrams_{k,m}(d)$, which implies that $patchfrac_{k,m'}(d) \geq patchfrac_{k,m}(d) \geq \theta$. Furthermore, since $pgrams_{k,m'}(d) \supseteq pgrams_{k,m}(d)$, and since $sources_{k,m}(d)$ is the *minimal* set of documents whose patch-grams cover $pgrams_{k,m}(d)$, the minimality property implies that $|sources_{k,m'}(d)| \geq |sources_{k,m}(d)| \geq c$. Taken together, this implies that d is (k, m', c, θ)-quilted. □

The attentive reader will notice that there is no Theorem 4 – (k, m, c, θ)-quiltedness is *not* monotonic in k. It is fairly easy to see why: choosing a very small value of k may render most k-grams common, i.e. occurring in more than m documents, while choosing a very large value of k may render most k-grams unique, i.e. occurring in only a single document. At either extreme, the average document will tend to have a smaller patch gram set, and hence a lower patch fraction, thereby reducing the number of documents d for which $patchfrac_{k,m}(d) \geq \theta$.

As mentioned above, the set cover problem is NP-hard. We compute $sources_{k,m}(d)$, the source documents contributing patch-grams to a document d, through the following greedy approximation algorithm:

$C \leftarrow \{(d', g) : d' \in D \setminus \{d\} \wedge g \in grams_k(d) \cap grams_k(d') \wedge 1 < |docs_k(g)| \leq m\}$
$sources_{k,m}(d) \leftarrow \emptyset$
while $C \neq \emptyset$ do
  find a d' that maximizes $|\{g : (d', g) \in C\}|$
  $sources_{k,m}(d) \leftarrow sources_{k,m}(d) \cup \{d'\}$
  $C \leftarrow C \setminus \{(d'', g') \in C : g' \in \{g : (d', g) \in C\}\}$

It should be pointed out that Theorem 3 does not necessarily hold when $sources_{k,m}(d)$ is approximated rather than computed precisely. Nonetheless, even when using the above greedy approximation, we expect the set of quilted documents in a large collection to increase "mostly" monotonically as m increases.

We implemented the above set-theoretic definition of quiltedness using DryadLINQ [17], a data-parallel cluster computing platform that leverages LINQ, Microsoft's "language integrated query" extension to the C# programming language. LINQ is based on *list comprehensions*, a concept that originated in the functional programming community and that in turn is related to mathematical set notation (sometimes called set comprehensions). Because of the fairly close relationship between LINQ and set notations, it is reasonably straightforward to cast the above definitions in LINQ and C#. The entire implementation, shown in Figure 5 is just 74 lines of C# and LINQ, not counting utility libraries for hashing and mapping between textual and numeric ClueWeb09 document identifiers.

Functional programs in general and list comprehensions in particular can be efficiently evaluated in a data-parallel fashion, and DryadLINQ exploits that fact by executing any LINQ query on many machines in parallel. A DryadLINQ program is started on a single machine. Whenever the control flow encounters a LINQ query, DryadLINQ ships that query to available machines in a compute cluster, which

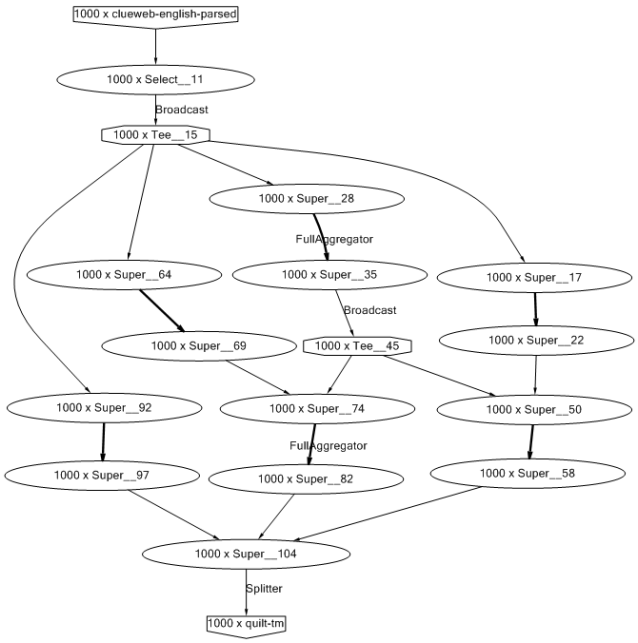

Figure 2: Data flow graph of the DryadLINQ implementation of our quilt detection algorithm.

also store the data (in our case, the corpus of documents) in a distributed fashion. Data is organized into *streams* segmented into multiple *partitions*, and the partitions of a stream are distributed across the cluster. A query is decomposed into multiple *stages*, each stage consisting of the maximal pipeline of operators (e.g. `Select`, `Where`, `GroupBy`, `Join`, `Partition` and `Merge`) that can be executed locally by reading and writing items in a streaming fashion. Each stage is executed in parallel, with the degree of parallelism determined by the partitioning of the input streams, and spread over the available machines in the cluster. A DryadLINQ computation can be characterized by a directed acyclic graph, with the nodes corresponding to stages and the edges corresponding to data flowing from one stage to the next. Figure 2 shows the data flow graph of the DryadLINQ implementation of our quilt detection algorithm. It contains 14 stages (represented as ovals), and completing each stage involves running 1000 processes distributed over the machines in the cluster. The processes in each stage are independent from each other and thus parallelizable; in practice, the DryadLINQ runtime runs just one such process per machine at any given time.

3. EXPERIMENTAL VALIDATION

The experiments described in this section were performed on a cluster of about 220 computers, each running the Windows Server 2003 64-bit operating system. Each machine had two dual-core AMD Opteron 2218 HE CPUs clocked at 2.6 GHz, 16 GB of DDR2 RAM, four 750 GB SATA drives, and a 1 Gbit/sec Ethernet NIC. Groups of about 25 computers were connected to a Blade RackSwitch G8052 48-port full-crossbar local switch; and the nine local switches were interconnected through a Blade RackSwitch G8264 switch, with 30 Gbit/sec full-duplex inter-switch bandwidth achieved via port aggregation.

For our experiments, we used the English-language subset of ClueWeb09 [14]. The collection was assembled through a web crawl conducted in 2009. The full collection consists of over a billion web pages, while the English-language subset is comprised of 503,898,901 pages. We segmented each page into individual words by embedding the Bing HTML parser into DryadLINQ and performing the parsing and word-breaking on our compute cluster. We subsequently performed several runs of the DryadLINQ implementation of our quilt detection algorithm, exploring various choices of the parameters k, m, c and θ. Typical runs took between 18 and 48 hours depending on parameter combinations as well as job competition. Since our compute cluster is shared among multiple users and a job's individual processes are routinely aborted by the cluster scheduler to accommodate other users, the aforementioned running times should only be viewed as upper bounds.

k	m	c	θ	# quilted	# sources
2	50	4	0.5	562,450	1114.78
3	50	4	0.5	4,865,307	27.75
4	50	4	0.5	33,904,401	21.58
5	50	4	0.5	58,002,718	22.35
6	50	4	0.5	58,526,481	16.49
7	50	4	0.5	52,926,434	12.14
8	50	4	0.5	47,158,399	9.78
9	50	4	0.5	42,278,205	8.53
10	50	4	0.5	38,479,113	7.82
5	50	2	0.5	82,832,986	16.37
5	50	3	0.5	68,103,596	19.48
5	50	4	0.5	58,002,718	22.35
5	50	5	0.5	50,251,757	25.18
5	50	6	0.5	44,206,465	27.94
5	50	7	0.5	39,444,093	30.59
5	50	8	0.5	35,611,760	33.12
5	50	9	0.5	32,453,725	35.57
5	50	10	0.5	29,797,202	37.94
5	10	4	0.5	16,924,347	15.16
5	20	4	0.5	31,492,573	17.87
5	30	4	0.5	42,267,988	19.85
5	40	4	0.5	50,849,031	21.32
5	50	4	0.5	58,002,718	22.35
5	60	4	0.5	64,145,058	23.16
5	70	4	0.5	69,434,726	23.84
5	80	4	0.5	74,062,259	24.41
5	50	4	0.4	93,141,485	28.68
5	50	4	0.5	58,002,718	22.35
5	50	4	0.6	33,040,377	16.79
5	50	4	0.7	15,700,171	12.71
5	50	4	0.8	4,650,478	9.38
5	50	4	0.9	439,044	6.95
5	50	4	1.0	209	4.75

Table 1: Percentage of (k, m, c, θ)-quilted pages in ClueWeb09 and average number of source pages, for various choices of k, m, c and θ.

The output of each quilt detection run is a stream containing the ClueWeb09 document IDs of all detected quilted

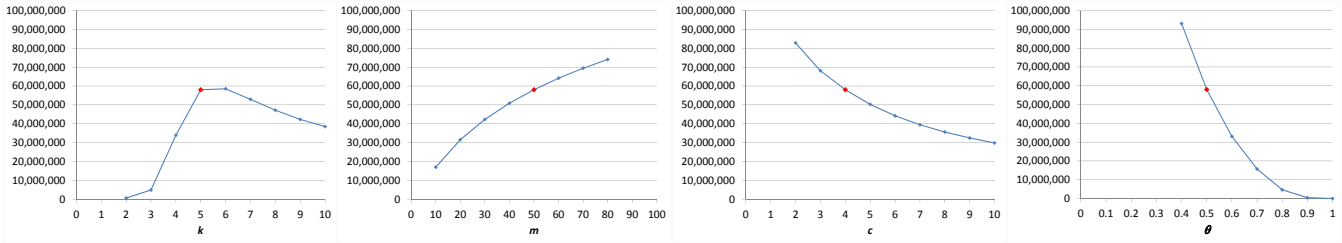

Figure 3: Number of quilted pages dependent on parameters setting. All plots have the common, highlighted pivot point $(k, m, c, \theta) = (5, 50, 4, 0.5)$, and each plots sweeps the parameter space in one dimension around that pivot point. From left to right, the parameters k, m, c and θ are varied.

pages, together with a minimal[1] set of source documents. We picked parameter combination $(k, m, c, \theta) = (5, 50, 4, 0.5)$ as a pivot point and varied the parameters around that pivot point, one dimension at a time. Table 1 shows the number of quilted pages in our corpus and the average number of source documents for each of the parameter combinations we explored. Figure 3 plots the number of quilted pages for each parameter sweep, giving credence to Theorems 1–3 stating the monotinicity in θ, c and m, as well as illustrating the non-monotonicity in k. The pivot point is highlighted in each plot.

In order to evaluate the effectiveness of our quilt detection algorithm at identifying spam web pages, we drew a small sample from the set of detected quilts, and extracted the words of each quilted page and their source documents from the ClueWeb09 corpus (again leveraging DryadLINQ). We then manually inspected each quilt and its sources and labeled them as *positive* (i.e. spam) or *negative* (not spam). Figure 4 shows a screen shot of the tool we used to perform this labeling.

Table 2 shows the results of labeling five of the runs described in Table 1,the first being the pivot point and the other four extreme choices of k, m, c, and θ. For each run, we sampled 100 (k, m, c, θ)-quilted pages and assessed them using the judging tool.

k	m	c	θ	spam
5	50	4	0.5	27%
10	50	4	0.5	22%
5	10	4	0.5	17%
5	50	10	0.5	19%
5	50	4	1.0	27%

Table 2: Percentage of sampled (k, m, c, θ)-quilted pages that are labeled as spam, for five choices of k, m, c and θ.

Among the five runs, the parameter combination we chose as the pivot point performed best. It is not clear how significant the observed differences in effectiveness are. The sample size is relatively small, but more importantly, our spam judgments are truly that: judgment calls. It was often quite hard to determine whether a given quilted page was spam or not. For one, a substantial fraction of web pages in the ClueWeb09 collection no longer exists in the wild; and often enough, the domain of the page either no longer exists or has been taken over by a "domain parking" service. In such cases, we had to judge the page solely based on its textual content, without considering embedded images or other pages in the same domain – two important cues in making spam judgments. Moreover, there is a fine line between low-quality content with commercial intent and outright spam. We might (somewhat arbitrarily) decide that an Amazon reseller engaging in keyword stuffing is pursuing a legitimate business, but a blog that is "reviewing" miracle patent medicines and monetizing this through a referral program is spam. It is hard to draw the line, and hard to do so consistently.

We were disappointed by the low percentage of quilted pages detected by our algorithm that would actually be considered spam by a human observer. By examining the many false positives, it became quickly clear that our algorithm identified the home pages of many blogs as quilted, with the full articles of the blog posts being the source documents contributing fragments to the blog home page. This observation guided us to add the following simple heuristic to our quilt detection algorithm: each source document should reside on a different web server than the quilted document. More formally, the initialization of C in the algorithm given in Section 2 is replaced by:

$$C \leftarrow \{(d', g) : d' \in D \setminus \{d\} \wedge g \in grams_k(d) \cap grams_k(d') \wedge 1 < |docs_k(g)| \leq m \wedge server(d) \neq server(d')\}$$

We refer to pages satisfying this refined definition of quiltedness as *foreign-sourced* quilted pages. There is some ambiguity as to what constitutes a web server: it could be identified by a symbolic host name, a paid domain name, an IP address, or even a domain registrar record. In our experiments, we treated paid domain names (such as `microsoft.com` or `cam.ac.uk`) as the identifiers of servers, and we insisted on source pages to originate from a different domain than the quilted page.

We ran this variant of our quilt detection algorithm on the same data set as before, with the parameters $(k, m, c, \theta) = (5, 50, 4, 0.5)$ that we had chosen as a pivot point. This run detected 20,576,548 quilted pages (a proper subset of about 35% of the pages detected by the unmodified quilt detection algorithm), each one incorporating an average of 28.26 foreign source pages. We labeled 100 pages drawn uniformly at random from the set of foreign-sourced quilted pages, and judged 34 of them as spam. This significantly increase in the fraction of quilted pages judged as spam provides some

[1]Or rather, approximately minimal, due to the greedy set cover approximation algorithm.

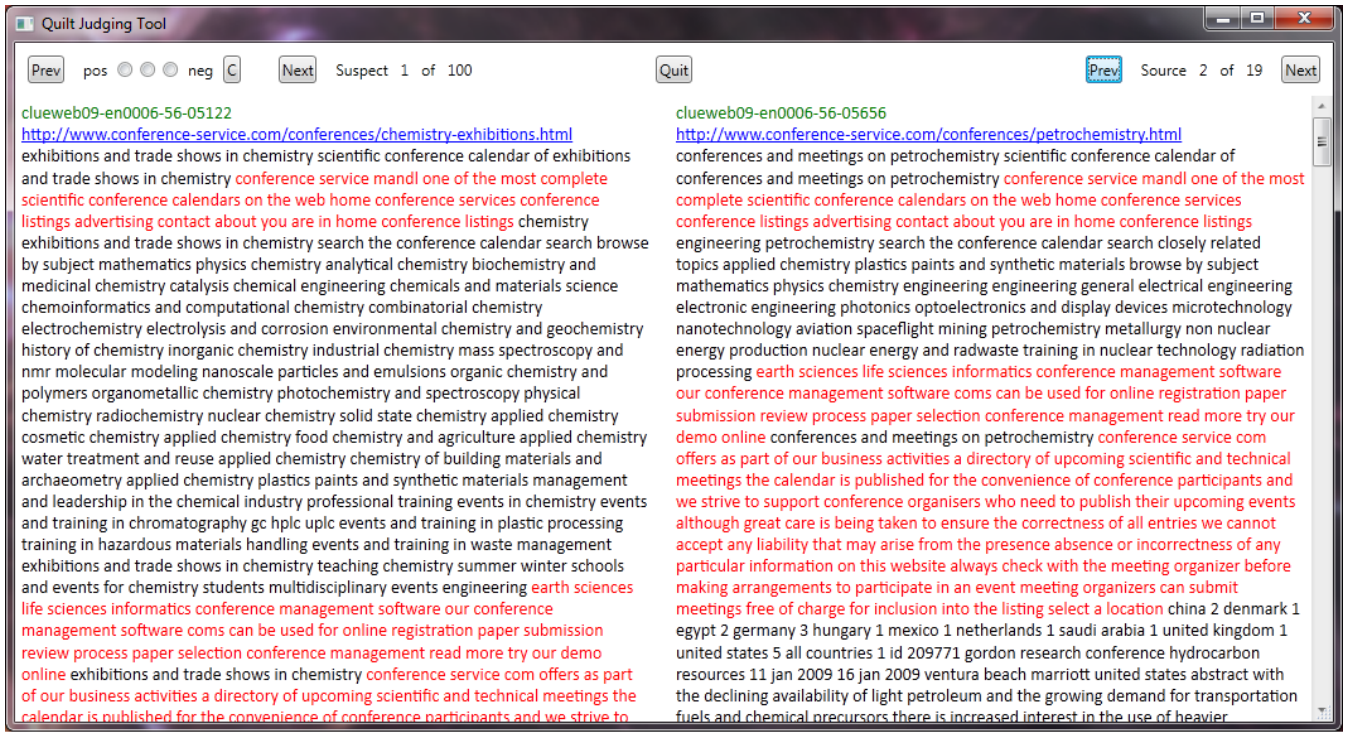

Figure 4: Tool used to view each quilt and its sources and label the quilt as spam or not spam.

evidence that the heuristic of requiring each source page to originate from a different server than the quilted page is indeed effective at surfacing spam.

None of the pages in our sample were of the blatant machine-generated variety we observed during our 2005 study, suggesting that spammers have become more sophisticated. However, our algorithm was very effective in surfacing web sites that appropriated semantically coherent parts of other web sites (say, a short essay) and used this content as "bait" to drive content to their own site, which was typically not topically related to the essay. Another pattern we observed repeatedly was "cookie-cutter" web sites, where apparently independent entities would offer essentially the same digital content, or sell the same products.

About two-thirds of the foreign-sourced quilted pages identified by our technique were not judged as spam. Within this set, we observed two notable patterns:

1. Publicly available documents (such as technical documents, standards, song lyrics, government notices, and biographic entries) are replicated on many sites, and most users would consider this content to be useful.

2. Some organizations (for example, Hartnell College) utilize multiple domains (`hartnell.edu` and `hartnell.cc.ca.us`), and serve similar pages on these domains.

The second class of false-positives accounted for about one-fifth of all false-positives. Spot-checking a few of them showed that most of these domains resolve to the same canonical IP address. For example, the symbolic host names `hartnell.edu` and `hartnell.cc.ca.us` resolve to `198.189.134.200`; and the symbolic host names `efinancialcareers.com`, `efinancialcareers.ie`, `efinancialcareers.be` resolve to `74.115.248.70`. Using IP addresses to represent web servers fits well into the data-parallel implementation of our algorithm. The alternative approach of leveraging the apparent textual similarity between these domain names, while feasible, is harder to shoe-horn into the DryadLINQ implementation. We implemented the former approach and tested it on the same data set as before, with the parameters $(k, m, c, \theta) = (5, 50, 4, 0.5)$ that we had chosen as a pivot point. This run detected 19,729,026 quilted pages, each one incorporating an average of 29.22 source pages hosted on web servers with different IP addresses than that of the quilted page. In other words, using this stricter notion of differing web servers has little impact on how many pages are detected as quilted. Again, we sampled 100 pages from this run and manually inspected them, finding 26 of them to be spam, which is noticeably lower than the 34% spam yield uncovered by the previous run. We attribute it to the imprecise nature of the assessment process.

In the course of assessing 100 samples each from 7 runs, we found that many quilted pages drew most of their patch-grams from just a few source pages, while most of the source pages contributed just one or a few patch grams. It would be interesting to modify the quilt detection algorithm (and the very definition of quiltedness) so as to ensure that each source page contributes at least a fraction of ϕ of the patch-grams in the quilted page. We leave this as an avenue for future work.

4. CONCLUSION

This paper builds on our earlier work on detecting phrase-level duplication on the web [9]. We give a formal definition of what constitutes a *quilted web page* (a web page that is stitched together out of textual patches taken from other web pages) and prove a few properties of that definition; we

provide an algorithm for exhaustively detecting all quilted web pages in a given collection; we describe a data-parallel implementation of the algorithm that scales to very large collections; we evaluate the effectiveness of the algorithm in detecting web spam by applying it (with various parameterizations) to the half-billion page English-language subset of the ClueWeb09 collection [14] and judging samples of the detected quilted pages as to whether or not they indeed constitute spam; and we suggest a few heuristics to improve effectiveness.

A major area for future work will be to combine the syntactic approach of our algorithm with semantic (and domain-specific) techniques for spam detection. While the syntactic technique described in this paper (by virtue of its exhaustiveness) is fully effective at detecting all quilted web pages, the task of effectively identifying spam must ultimately incorporate semantic information. Our algorithm (correctly) identities news aggregation web sites to contain quilted pages (pages that collate opening paragraphs of news articles from other sources); but such news aggregation pages are not spam – users derive a real value from them. A potential semantic signal that distinguishes news aggregation from spam is *attribution* – a link from each excerpt to the underlying article.

The core ideas described in this paper are applicable not only to spam suppression in search engines, but also (and probably more so) to detecting plagiarism in web-scale corpora. Here again, semantic information (such as proper attribution) will be needed in addition to the purely syntactic approach of our technique.

Acknowledgments

This work grew out of a 2004 collaboration with Mark Manasse and Dennis Fetterly, and benefited from our many discussions. Dennis played a crucial role in the work described in this paper, by word-breaking the ClueWeb09 collection using the Bing HTML parser, supplying me with IP addresses for the URLs in the ClueWeb09 collection, providing me with a judging tool to adapt for this project, and patiently answering my many questions related to DryadLINQ. Thanks to the entire DryadLINQ team for building a system to make cluster-scale computing easy and fun, and particularly to Yuan Yu for being super-responsive in fixing bugs. Finally, thanks to Jamie Callan and his team at CMU for compiling the ClueWeb09 collection, which has arguably become the reference data set for reproducible web corpus research. I can hardly wait for ClueWeb12!

5. REFERENCES

[1] Jacqueline Anderson, Reineke Reitsma, Patti F. Evans, Samantha Y. Jaddou. Understanding Online Shopper Behaviors, US 2011. *Forrester Research*, 2011.

[2] Ricardo Baeza-Yates, Álvaro Pereira, and Nivio Ziviani. Genealogical Trees on the Web: A Search Engine User Perspective. In *17th International World Wide Web Conference*, 2008.

[3] Michael Bendersky and W. Bruce Croft. Finding Text Reuse on the Web. In *2nd ACM International Conference on Web Search and Data Mining*, 2009.

[4] Sergey Brin, James Davis, and Hector Garcia-Molina. Copy Detection Mechanisms for Digital Documents. In *ACM SIGMOD International Conference on Management of Data*, 1995.

[5] Andrei Z. Broder. On the Resemblance and Containment of Documents. In *Compression and Complexity of Sequences*, 1997.

[6] Andrei Z. Broder, Steven C. Glassmann, Mark S. Manasse, and Geoffrey Zweig. Syntactic Clustering of the Web. In *6th International World Wide Web Conference*, 1997.

[7] Moses S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *34th Annual ACM Symposium on Theory of Computing*, 2002.

[8] Dennis Fetterly, Mark Manasse and Marc Najork. On the Evolution of Clusters of Near-Duplicate Web Pages. *1st Latin American Web Congress*, 2003.

[9] Dennis Fetterly, Mark Manasse and Marc Najork. Detecting Phrase-Level Duplication on the World Wide Web. In *28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2005.

[10] Monika Henzinger. Finding Near-Duplicate Web Pages: A Large-Scale Evaluation of Algorithms. In *29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2006.

[11] Timothy C. Hoad and Justin Zobel. Methods for Identifying Versioned and Plagiarised Documents. *Journal of the American Society for Information Science and Technology*, **54**(3):203–215, 2003.

[12] Richard M. Karp. Reducibility Among Combinatorial Problems. In *Symposium on the Complexity of Computer Computations*, 1972.

[13] Okan Kolak and Bill N. Schilit. Generating Links by Mining Quotations. In *19th ACM Conference on Hypertext and Hypermedia*, 2008.

[14] Lemur Project. The ClueWeb09 DataSet. Online at `http://lemurproject.org/clueweb09/`

[15] Jangwon Seo and W. Bruce Croft. Local Text Reuse Detection. In *31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2008.

[16] Narayanan Shivakumar and Hector Garcia-Molina. SCAM: A Copy Detection Mechanism for Digital Documents. In *2nd Annual Conference on the Theory and Practice of Digital Libraries*, 1995.

[17] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gundar, and Jon Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

```
public class Doc : IEquatable<Doc> {
  public string url;
  public string docId;
  public string[] words;
  public override int GetHashCode() {
    return docId.GetHashCode();
  }
  public bool Equals(Doc other) {
    return docId.Equals(other.docId, StringComparison.Ordinal);
  }
}
public class QuiltFinder {
  public static ulong[] HashPatches(string[] words, int k) {
    var res = new ulong[Math.Max(0, words.Length - k + 1)];
    var sb = new StringBuilder();
    for (int i = 0; i < res.Length; i++) {
      for (int j = 0; j < k; j++) {
        if (j > 0) sb.Append(" ");
        sb.Append(words[i + j]);
      }
      res[i] = Hash(sb.ToString());
      sb.Clear();
    }
    return res;
  }
  public static IEnumerable<Pair<ulong,Pair<ulong,ulong>>> MakeDocGramDocTriple(ulong gram, ulong[] ids) {
    for (int i = 0; i < ids.Length; i++) {
      for (int j = 0; j < ids.Length; j++) {
        if (i != j) {
          yield return new Pair<ulong, Pair<ulong, ulong>>(ids[i], new Pair<ulong, ulong>(gram, ids[j]));
        }
      }
    }
  }
  public static ulong[] MinSourceDocSet(List<Pair<ulong, ulong>> gramIds) {
    List<ulong> res = new List<ulong>();
    while (gramIds.Count > 0) {
      var bestDoc = gramIds.OrderBy(x => x.Value)
                           .GroupBy(x => x.Value, x => x.Key, (k, g) => new Pair<ulong, int>(k, g.Count()))
                           .OrderByDescending(x => x.Value)
                           .First()
                           .Key;
      res.Add(bestDoc);
      var coveredGrams = gramIds.Where(x => x.Value == bestDoc).Select(x => x.Key).ToList();
      var dischargedGramDocs = gramIds.Join(coveredGrams, x => x.Key, y => y, (x,y) => x).ToList();
      gramIds = gramIds.Except(dischargedGramDocs).ToList();
    }
    return res.ToArray();
  }
  public static void FindQuilts(string inputStream, string outputStream, int k, int m, int c, double theta) {
    var docToGrams = PartitionedTable.Get<InitialParsedDocWithLinks>(inputStream)
                                     .Select(x => new { id = DocIdToULong(x.docId), grams = HashPatches(x.words, k) });
    var gramsPerDoc = docToGrams.Select(x => new { id = x.id, count = (double)x.grams.Length });
    var patchGrams = docToGrams.SelectMany(x => x.grams.Distinct())
                               .GroupBy(x => x, x => true, (gram, group) => new {gram = gram, count = group.Count()})
                               .Where(x => x.count > 1 && x.count <= m)
                               .Select(x => x.gram);
    var patchyDocs = docToGrams.SelectMany(x => x.grams.Select(y => new { id = x.id, gram = y }))
                               .Join(patchGrams, x => x.gram, y => y, (x, y) => x)
                               .GroupBy(x => x.id, x => x.gram, (id, group) => new { id = id, count = (double)group.Count() })
                               .Join(gramsPerDoc, x => x.id, y => y.id, (x, y) => new { id = x.id, frac = x.count / y.count })
                               .Where(x => x.frac >= theta)
                               .Select(x => x.id);
    var quilts = docToGrams.SelectMany(x => x.grams.Select(y => new { gram = y, id = x.id }))
                           .Join(patchGrams, x => x.gram, y => y, (x, y) => x)
                           .GroupBy(x => x.gram, x => x.id, (gram, group) => new { gram = gram, ids = group.Distinct().ToArray() })
                           .SelectMany(x => MakeDocGramDocTriple(x.gram, x.ids))
                           .Join(patchyDocs, x => x.Key, y => y, (x,y) => x)
                           .GroupBy(x => x.Key, x => x.Value, (id, group) => new { id = id, ids = MinSourceDocSet(group.ToList()) })
                           .Where(x => x.ids.Length >= c)
                           .Select(x => new Pair<string,string[]>(ULongToDocId(x.id), x.ids.Select(y => ULongToDocId(y)).ToArray()))
                           .ToPartitionedTable(outputStream);
  }
}
```

Figure 5: A DryadLINQ implementation of the quilt detection algorithm