Do Not Crawl in the DUST: Different URLs with Similar Text*

Ziv Bar-Yossef†
Dept. of Electrical Engineering
Technion, Haifa 32000, Israel
Google Haifa Engineering Center, Israel
zivby@ee.technion.ac.il

Idit Keidar
Dept. of Electrical Engineering
Technion, Haifa 32000, Israel
idish@ee.technion.ac.il

Uri Schonfeld‡
Dept. of Computer Science
University of California
Los Angeles, CA 90095, USA
shuri@shuri.org

ABSTRACT

We consider the problem of DUST: Different URLs with Similar Text. Such duplicate URLs are prevalent in web sites, as web server software often uses aliases and redirections, and dynamically generates the same page from various different URL requests. We present a novel algorithm, *DustBuster*, for uncovering DUST; that is, for discovering rules that transform a given URL to others that are likely to have similar content. DustBuster mines DUST effectively from previous crawl logs or web server logs, *without* examining page contents. Verifying these rules via sampling requires fetching few actual web pages. Search engines can benefit from information about DUST to increase the effectiveness of crawling, reduce indexing overhead, and improve the quality of popularity statistics such as PageRank.

Categories and Subject Descriptors: H.3.3: Information Search and Retrieval.

General Terms: Algorithms.

Keywords: search engines, crawling, URL normalization, duplicate detection, anti-aliasing.

1. INTRODUCTION

The DUST problem. The web is abundant with DUST: *Different URLs with Similar Text* [17, 10, 20, 18]. For example, the URLs `http://google.com/news` and `http://news.google.com` return similar content. Adding a trailing slash or `/index.html` to either returns the same result. Many web sites define links, redirections, or aliases, such as allowing the tilde symbol ˜ to replace a string like `/people`. A single web server often has multiple DNS names, and any can be typed in the URL. As the above examples illustrate, DUST is typically not random, but rather stems from some general rules, which we call *DUST rules*, such as "˜" → "`/people`", or "`/index.html`" at the end of the URL can be omitted.

DUST rules are typically not universal. Many are artifacts of a particular web server implementation. For example, URLs of dynamically generated pages often include parameters; which parameters impact the page's content is up to the software that generates the pages. Some sites use their own conventions; for example, a forum site we studied allows accessing story number "num" both via the URL `http://domain/story?id=num` and via `http://domain/story_num`. Our study of the CNN web site has discovered that URLs of the form `http://cnn.com/money/whatever` get redirected to `http://money.cnn.com/whatever`. In this paper, we focus on mining DUST rules within a given web site. We are not aware of any previous work tackling this problem.

Standard techniques for avoiding DUST employ universal rules, such as adding `http://` or removing a trailing slash, in order to obtain some level of canonization. Additional DUST is found by comparing document sketches. However, this is conducted on a page by page basis, and all the pages must be fetched in order to employ this technique. By knowing DUST rules, one can reduce the overhead of this process.

Knowledge about DUST rules can be valuable for search engines for additional reasons: DUST rules allow for a *canonical* URL representation, thereby reducing overhead in crawling, indexing, and caching [17, 10], and increasing the accuracy of page metrics, like PageRank. For example, in one crawl we examined, the number of URLs fetched would have been reduced by 26%.

We focus on URLs with *similar* contents rather than identical ones, since different versions of the same document are not always identical; they tend to differ in insignificant ways, e.g., counters, dates, and advertisements. Likewise, some URL parameters impact only the way pages are displayed (fonts, image sizes, etc.) without altering their contents.

Detecting DUST from a URL list. Contrary to initial intuition, we show that it is possible to discover likely DUST rules without fetching a single web page. We present an algorithm, *DustBuster*, which discovers such likely rules from a list of URLs. Such a *URL list* can be obtained from many sources including a previous crawl or web server logs.[1] The rules are then verified (or refuted) by sampling a small number of actual web pages.

At first glance, it is not clear that a URL list can provide reliable information regarding DUST, as it does not include actual page contents. We show, however, how to use a URL

*A short version of this paper appeared as a poster at WWW2006 [21]. A full version, including proofs and more experimental results, is available as a technical report [2].

†Supported by the European Commission Marie Curie International Re-integration Grant and by Bank Hapoalim.

‡This work was done while the author was at the Technion.

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

[1]Increasingly many web server logs are available nowadays to search engines via protocols like Google Sitemaps [13].

list to discover two types of DUST rules: *substring substitutions*, which are similar to the "replace" function in editors, and *parameter substitutions*. A substring substitution rule $\alpha \to \beta$ replaces an occurrence of the string α in a URL by the string β. A parameter substitution rule replaces the value of a parameter in a URL by some default value. Thanks to the standard syntax of parameter usage in URLs, detecting parameter substitution rules is fairly straightforward. Most of our work therefore focuses on substring substitution rules.

DustBuster uses three heuristics, which together are very effective at detecting likely DUST rules and distinguishing them from invalid ones. The first heuristic leverages the observation that if a rule $\alpha \to \beta$ is common in a web site, then we can expect to find in the URL list multiple examples of pages accessed both ways. For example, in the site where `story?id=` can be replaced by `story_`, we are likely to see many different URL pairs that differ only in this substring; we say that such a pair of URLs is an *instance* of the rule "`story?id=`" → "`story_`". The set of all instances of a rule is called the rule's *support*. Our first attempt to uncover DUST is therefore to seek rules that have large support.

Nevertheless, some rules that have large support are not DUST rules. For example, in one site we found instances such as `http://movie-forum.com/story_100` and `http://politics-forum.com/story_100` which support the invalid rule "`movie-forum`" → "`politics-forum`". Another example is "1" → "2", which emanates from instances like `pic-1.jpg` and `pic-2.jpg`, `story_1` and `story_2`, and `lect1` and `lect2`, none of which are DUST. Our second and third heuristics address the challenge of eliminating such invalid rules. The second heuristic is based on the observation that invalid rules tend to flock together. For example in most instances of "1" → "2", one could also replace the "1" by other digits. We therefore ignore rules that come in large groups.

Further eliminating invalid rules requires calculating the fraction of DUST in the support of each rule. How could this be done without inspecting page content? Our third heuristic uses cues from the URL list to guess which instances are likely to be DUST and which are not. In case the URL list is produced from a previous crawl, we typically have document sketches [7] available for each URL in the list. These sketches can be used to estimate the similarity between documents and thus to eliminate rules whose support does not contain sufficiently many DUST pairs.

In case the URL list is produced from web server logs, document sketches are not available. The only cue about the contents of URLs in these logs is the sizes of these contents. We thus use the size field from the log to filter out instances (URL pairs) that have "mismatching" sizes. The difficulty with size-based filtering is that the size of a dynamic page can vary dramatically, e.g., when many users comment on an interesting story or when a web page is personalized. To account for such variability, we compare the ranges of sizes seen in all accesses to each page. When the size ranges of two URLs do not overlap, they are unlikely to be DUST.

Having discovered likely DUST rules, another challenge that needs to be addressed is eliminating redundant ones. For example, the rule "`http://site-name/story?id=`" → "`http://site-name/story_`" will be discovered, along with many consisting of substrings thereof, e.g., "`?id=`" → "`_`". However, before performing validations, it is not obvious which rule should be kept in such situations– the latter could be either valid in all cases, or invalid outside the *context* of the former. We are able to use support information from the URL list to remove many redundant likely DUST rules. We remove additional redundancies after performing some validations, and thus compile a succinct list of rules.

Canonization. Once the correct DUST rules are discovered, we exploit them for URL canonization. While the canonization problem is NP-hard in general, we have devised an efficient *canonization algorithm* that *typically* succeeds in transforming URLs to a site-specific canonical form.

Experimental results. We experiment with DustBuster on four web sites with very different characteristics. Two of our experiments use web server logs, and two use crawl outputs. We find that DustBuster can discover rules very effectively from moderate sized URL lists, with as little as 20,000 entries. Limited sampling is then used in order to validate or refute each rule.

Our experiments show that up to 90% of the top ten rules discovered by DustBuster *prior to the validation phase* are found to be valid, and in most sites 70% of the top 100 rules are valid. Furthermore, DUST rules discovered by DustBuster may account for 47% of the DUST in a web site and that using DustBuster can reduce a crawl by up to 26%.

Roadmap. The rest of this paper is organized as follows. Section 2 reviews related work. We formally define the DUST detection and canonization problems in Section 3. Section 4 presents the basic heuristics our algorithm uses. DustBuster and the canonization algorithm appear in Section 5. Section 6 presents experimental results. We end with some concluding remarks in Section 7.

2. RELATED WORK

The standard way of dealing with DUST is using document sketches [6, 11, 7, 22, 9, 16, 15], which are short summaries used to determine similarities among documents. To compute such a sketch, however, one needs to fetch and inspect the whole document. Our approach cannot replace document sketches, since it does not find DUST across sites or DUST that does not stem from rules. However, it is desirable to use our approach to complement document sketches in order to reduce the overhead of collecting redundant data. Moreover, since document sketches do not give rules, they cannot be used for URL canonization, which is important, e.g., to improve the accuracy of page popularity metrics.

One common source of DUST is mirroring. A number of previous works have dealt with automatic detection of mirror sites on the web [3, 4, 8, 19]. We deal with the complementary problem of detecting DUST within one site. A major challenge that site-specific DUST detection must address is efficiently *discovering* prospective rules out of a daunting number of possibilities (all possible substring substitutions). In contrast, mirror detection is given pairs of sites to compare, and only needs to determine *whether* they are mirrors.

Our problem may seem similar to mining association rules [1], yet the two problems differ substantially. Whereas the input of such mining algorithms consists of complete lists of items that belong together, our input includes individual items from different lists. The absence of complete lists renders techniques used therein inapplicable to our problem.

One way to view our work is as producing an Abstract Rewrite System (ARS) [5] for URL canonization via DUST rules. For ease of readability, we have chosen not to adopt the ARS terminology in this paper.

3. PROBLEM DEFINITION

URLs. We view URLs as strings over an alphabet Σ of tokens. Tokens are either alphanumeric strings or non-alphanumeric characters. In addition, we require every URL to start with the special token `^` and to end with the special token `$` (`^` and `$` are not included in Σ).

A URL u is *valid*, if its domain name resolves to a valid IP address and its contents can be fetched by accessing the corresponding web server (the http return code is not in the 4xx or 5xx series). If u is valid, we denote by $\mathrm{doc}(u)$ the returned document.

DUST. Two valid URLs u_1, u_2 are called DUST if their corresponding documents, $\mathrm{doc}(u_1)$ and $\mathrm{doc}(u_2)$, are "similar". To this end, any method of measuring the similarity between two documents can be used. For our implementation and experiments, we use the popular *shingling resemblance* measure due to Broder *et al.* [7].

DUST rules. We seek general *rules* for detecting when two URLs are DUST. A DUST rule ϕ is a relation over the space of URLs. ϕ may be many-to-many. Every pair of URLs belonging to ϕ is called an *instance* of ϕ. The *support* of ϕ, denoted $\mathrm{support}(\phi)$, is the collection of all its instances.

Our algorithm focuses primarily on detecting substring substitution rules. A *substring substitution rule* $\alpha \to \beta$ is specified by an ordered pair of strings (α, β) over the token alphabet Σ. (In addition, we allow these strings to simultaneously start with the token `^` and/or to simultaneously end with the token `$`.) Instances of substring substitution rules are defined as follows:

DEFINITION 3.1 (INSTANCE OF A RULE). *A pair u_1, u_2 of URLs is an* instance *of a substring substitution rule $\alpha \to \beta$, if there exist strings p, s s.t. $u_1 = p\alpha s$ and $u_2 = p\beta s$.*

For example, the pair of URLs `http://www.site.com/index.html` and `http://www.site.com` is an instance of the DUST rule "`/index.html$`" $\to$ "`$`".

The DUST problem. Our goal is to detect DUST and eliminate redundancies in a collection of URLs belonging to a given web site S. This is solved by a combination of two algorithms, one that discovers DUST rules from a URL list, and another that uses them in order to transform URLs to their canonical form.

A *URL list* is a list of records consisting of: (1) a URL; (2) the http return code; (3) the size of the returned document; and (4) the document's sketch. The last two fields are optional. This type of list can be obtained from web server logs or from a previous crawl. The URL list is a (non-random) sample of the URLs that belong to the web site.

For a given web site S, we denote by U_S the set of URLs that belong to S. A DUST rule ϕ is said to be *valid* w.r.t. S, if for each $u_1 \in U_S$ and for each u_2 s.t. (u_1, u_2) is an instance of ϕ, $u_2 \in U_S$ and (u_1, u_2) is DUST.

A *DUST rule detection algorithm* is given a list $\mathcal{L}$ of URLs from a web site S and outputs an ordered list of DUST rules. The algorithm may also fetch pages (which may or may not appear in the URL list). The ranking of rules represents the confidence of the algorithm in the validity of the rules.

Canonization. Let $\mathcal{R}$ be an ordered list of DUST rules that have been found to be valid w.r.t. some web site S. We would like to define what is a *canonization* of the URLs in U_S using the rules in $\mathcal{R}$. To this end, we assume that there is some standard way of applying every rule $\phi \in \mathcal{R}$, so that applying ϕ to any URL $u \in U_S$ results in a URL $\phi(u)$ that also belongs to U_S (this assumption holds true in all the data sets we experimented with).

The rules in $\mathcal{R}$ naturally induce a labeled graph $G_{\mathcal{R}}$ on U_S: there is an edge from u_1 to u_2 labeled by ϕ if and only if (u_1, u_2) is an instance of ϕ. Note that adjacent URLs in $G_{\mathcal{R}}$ correspond to similar documents. For the purpose of canonization, we assume that document dissimilarity respects at least a weak form of the triangle inequality, so that URLs connected by short paths in $G_{\mathcal{R}}$ are similar URLs too. Thus, if $G_{\mathcal{R}}$ has a bounded diameter (as it does in the data sets we encountered), then every two URLs connected by a path are similar. A canonization that maps every URL u to some URL that is reachable from u thus makes sense, because the original URL and its canonical form are guaranteed to be DUST.

A set of *canonical URLs* is a subset $CU_S \subseteq U_S$ that is reachable from every URL in U_S (equivalently, CU_S is a dominating set in the transitive closure of $G_{\mathcal{R}}$). A canonization is any mapping $C : U_S \to CU_S$ that maps every URL $u \in U_S$ to some canonical URL $C(u)$, which is reachable from u by a directed path. Our goal is to find a small set of canonical URLs and a corresponding canonization, which is efficiently computable.

Finding the minimum size set of canonical URLs is intractable, due to the NP-hardness of the minimum dominating set problem (cf. [12]). Fortunately, our empirical study indicates that for typical collections of DUST rules found in web sites, efficient canonization is possible. Thus, although we cannot design an algorithm that always obtains an optimal canonization, we will seek one that maps URLs to a *small* set of canonical URLs, and *always* terminates in polynomial time.

Metrics. We use three measures to evaluate DUST detection and canonization. The first measure is *precision*—the fraction of valid rules among the rules reported by the DUST detection algorithm. The second, and most important, measure is the *discovered redundancy*—the amount of redundancy eliminated in a crawl. It is defined as the difference between the number of unique URLs in the crawl before and after canonization, divided by the former.

The third measure is *coverage*: given a large collection of URLs that includes DUST, what percentage of the duplicate URLs is detected. The number of duplicate URLs in a given URL list is defined as the difference between the number of unique URLs and the number of unique document sketches. Since we do not have access to the entire web site, we measure the achieved coverage within the URL list. We count the number of duplicate URLs in the list before and after canonization, and the difference between them divided by the former is the coverage.

One of the standard measures of information retrieval is *recall*. In our case, recall would measure what percent of all correct DUST rules is discovered. However, it is clearly impossible to construct a complete list of all valid rules to compare against, and therefore, recall is not directly measurable in our case.

4. BASIC HEURISTICS

Our algorithm for extracting likely string substitution rules from the URL list uses three heuristics: the *large support heuristic*, the *small buckets heuristic*, and the *similarity like-*

liness heuristic. Our empirical results provide evidence that these heuristics are effective on web-sites of varying scopes and characteristics.

Large support heuristic.

> **Large Support Heuristic**
> *The support of a valid DUST rule is large.*

For example, if a rule "`index.html$`" $\rightarrow$ "`$`" is valid, we should expect many instances witnessing to this effect, e.g., `www.site.com/d1/index.html` and `www.site.com/d1/`, and `www.site.com/d3/index.html` and `www.site.com/d3/`. We would thus like to discover rules of large support. Note that valid rules of small support are not very interesting anyway, because the savings gained by applying them are negligible.

Finding the support of a rule on the web site requires knowing all the URLs associated with the site. Since the only data at our disposal is the URL list, which is unlikely to be complete, the best we can do is compute the support of rules *in this URL list*. That is, for each rule ϕ, we can find the number of instances (u_1, u_2) of ϕ, for which both u_1 and u_2 appear in the URL list. We call these instances the *support of ϕ in the URL list* and denote them by $\text{support}_{\mathcal{L}}(\phi)$. If the URL list is long enough, we expect this support to be representative of the overall support of the rule on the site.

Note that since $|\text{support}_{\mathcal{L}}(\alpha \rightarrow \beta)| = |\text{support}_{\mathcal{L}}(\beta \rightarrow \alpha)|$, for every α and β, our algorithm cannot know whether both rules are valid or just one of them is. It therefore outputs the pair α, β instead. Finding which of the two directions is valid is left to the final phase of DustBuster.

Given a URL list $\mathcal{L}$, how do we compute the size of the support of every possible rule? To this end, we introduce a new characterization of the support size. Consider a substring α of a URL $u = p\alpha s$. We call the pair (p, s) the *envelope* of α in u. For example, if $u =$`http://www.site.com/index.html` and $\alpha =$"`index`", then the envelope of α in u is the pair of strings "`^http://www.site.com/`" and "`.html$`". By Definition 3.1, a pair of URLs (u_1, u_2) is an instance of a substitution rule $\alpha \rightarrow \beta$ if and only if there exists a shared envelope (p, s) so that $u_1 = p\alpha s$ and $u_2 = p\beta s$.

For a string α, denote by $E_{\mathcal{L}}(\alpha)$ the set of envelopes of α in URLs that satisfy the following conditions: (1) these URLs appear in the URL list $\mathcal{L}$; and (2) the URLs have α as a substring. If α occurs in a URL u several times, then u contributes as many envelopes to $E_{\mathcal{L}}(\alpha)$ as the number of occurrences of α in u. The following theorem, proven in the full draft of the paper, shows that under certain conditions, $|E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)|$ equals $|\text{support}_{\mathcal{L}}(\alpha \rightarrow \beta)|$. As we shall see later, this gives rise to an efficient procedure for computing support size, since we can compute the envelope sets of each substring α separately, and then by join and sort operations find the pairs of substrings whose envelope sets have large intersections.

THEOREM 4.1. *Let $\alpha \neq \beta$ be two non-empty and non-semiperiodic strings. Then,*

$$|\text{support}_{\mathcal{L}}(\alpha \rightarrow \beta)| = |E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)|.$$

A string α is *semiperiodic*, if it can be written as $\alpha = \gamma^k \gamma'$ for some string γ, where $|\alpha| > |\gamma|$, $k \geq 1$, γ^k is the string obtained by concatenating k copies of the string γ, and γ' is a (possibly empty) prefix of γ [14]. If α is not semiperiodic, it is *non-semiperiodic*. For example, the strings "`/////`" and "`a.a.a`" are semiperiodic, while the strings "`a.a.b`" and "`%////`" are not.

Unfortunately, the theorem does not hold for rules where one of the strings is either semiperiodic or empty. For example, let α be the semiperiodic string "`a.a`" and $\beta =$ "`a`". Let $u_1 =$ `http://a.a.a/` and let $u_2 =$ `http://a.a/`. There are *two* ways in which we can substitute α with β in u_1 and obtain u_2. Similarly, let γ be "`a.`" and δ be the empty string. There are two ways in which we can substitute γ with δ in u_1 to obtain u_2. This means that the instance (u_1, u_2) will be associated with two envelopes in $E_{\mathcal{L}}(\gamma) \cap E_{\mathcal{L}}(\delta)$ and with two envelopes in $E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$ and not just one. Thus, when α or β are semiperiodic or empty, $|E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)|$ can overestimate the support size. On the other hand, such examples are quite rare, and in practice we expect a minimal gap between $|E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)|$ and the support size.

Small buckets heuristic. While most valid DUST rules have large support, the converse is not necessarily true: there can be rules with large support that are not valid. One class of such rules is substitutions among numbered items, e.g., (`lect1.ps`,`lect2.ps`), (`lect1.ps`,`lect3.ps`), and so on.

We would like to somehow filter out the rules with "misleading" support. The support for a rule $\alpha \rightarrow \beta$ can be thought of as a collection of recommendations, where each envelope $(p, s) \in E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$ represents a single recommendation. Consider an envelope (p, s) that is willing to give a recommendation to any rule, for example "`^http://`" $\rightarrow$ "`^`". Naturally its recommendations lose their value. This type of support only leads to many invalid rules being considered. This is the intuitive motivation for the following heuristic to separate the valid DUST rules from invalid ones.

If an envelope (p, s) belongs to many envelope sets $E_{\mathcal{L}}(\alpha_1)$, $E_{\mathcal{L}}(\alpha_2)$,..., $E_{\mathcal{L}}(\alpha_k)$, then it contributes to the intersections $E_{\mathcal{L}}(\alpha_i) \cap E_{\mathcal{L}}(\alpha_j)$, for all $1 \leq i \neq j \leq k$. The substrings $\alpha_1, \alpha_2, \ldots, \alpha_k$ constitute what we call a *bucket*. That is, for a given envelope (p, s), *bucket*(p, s) is the set of all substrings α s.t. $p\alpha s \in \mathcal{L}$. An envelope pertaining to a large bucket supports many rules.

> **Small Buckets Heuristic**
> *Much of the support of valid DUST substring substitution rules is likely to belong to small buckets.*

Similarity likeliness heuristic. The above two heuristics use the URL strings alone to detect DUST. In order to raise the precision of the algorithm, we use a third heuristic that better captures the "similarity dimension", by providing hints as to which instances are likely to be similar.

> **Similarity Likeliness Heuristic**
> *The likely similar support of a valid DUST rule is large.*

We show below that using cues from the URL list we can determine which URL pairs in the support of a rule are likely to have similar content, i.e., are *likely similar*, and which are not. The likely similar support, rather than the complete support, is used to determine whether a rule is valid or not. For example, in a forum web site we examined, the URL list included two sets of URLs `http://politics.domain/story_num` and `http://movies.domain/story_num` with different numbers. The support of the invalid rule "`http://`

politics.domain" → "http://movies.domain" was large, yet since the corresponding stories were very different, the likely similar support of the rule was found to be small.

How do we use the URL list to estimate similarity between documents? The simplest case is that the URL list includes a document sketch, such as the shingles of Broder *et al.* [7], for each URL. Such sketches are typically available when the URL list is the output of a previous crawl of the web site. When available, documents sketches are used to indicate which URL pairs are likely similar.

When the URL list is taken from web server logs, documents sketches are not available. In this case we use document sizes (document sizes are usually given by web server software). We determine two documents to be similar if their sizes "match". Size matching, however, turns out to be quite intricate, because the same document may have very different sizes when inspected at different points of time or by different users. This is especially true when dealing with forums or blogging web sites. Therefore, if two URLs have different "size" values in the URL list, we cannot immediately infer that these URLs are not DUST. Instead, for each unique URL, we track all its occurrences in the URL list, and keep the minimum and the maximum size values encountered. We denote the interval between these two numbers by I_u. A pair of URLs, u_1 and u_2, in the support are considered likely similar if the intervals I_{u_1} and I_{u_2} overlap. Our experiments show that this heuristic is very effective in improving the precision of our algorithm, often increasing precision by a factor of two.

5. DUSTBUSTER

In this section we describe DustBuster—our algorithm for discovering site-specific DUST rules. DustBuster has four phases. The first phase uses the URL list alone to generate a short list of *likely* DUST rules. The second phase removes redundancies from this list. The next phase generates likely parameter substitution rules and is discussed in the full draft of the paper. The last phase validates or refutes each of the rules in the list, by fetching a small sample of pages.

5.1 Detecting likely DUST rules

Our strategy for discovering likely DUST rules is the following: we compute the size of the support of each rule that has at least one instance in the URL list, and output the rules whose support exceeds some threshold MS. Based on Theorem 4.1, we compute the size of the support of a rule $\alpha \to \beta$ as the size of the set $E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$. That is roughly what our algorithm does, but with three reservations:

(1) Based on the small buckets heuristic, we avoid considering certain rules by ignoring large buckets in the computation of envelope set intersections. Buckets bigger than some threshold T are called *overflowing*, and all envelopes pertaining to them are denoted collectively by O and are not included in the envelope sets.

(2) Based on the similarity likeliness heuristic, we filter support by estimating the likelihood of two documents being similar. We eliminate rules by filtering out instances whose associated documents are unlikely to be similar in content. That is, for a given instance $u_1 = p\alpha s$ and $u_2 = p\beta s$, the envelope (p, s) is disqualified if u_1 and u_2 are found unlikely to be similar using the tests introduced in Section 4. These techniques are provided as a boolean function LikelySimilar which returns false only if the documents of the two input URLs are unlikely to be similar. The set of all disqualified envelopes is then denoted $D_{\alpha,\beta}$.

(3) In practice, substitutions of long substrings are rare. Hence, our algorithm considers substrings of length at most S, for some given parameter S.

To conclude, our algorithm computes for every two substrings α, β that appear in the URL list and whose length is at most S, the size of the set $(E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)) \setminus (O \cup D_{\alpha,\beta})$.

1: **Function** DetectLikelyRules(URLList $\mathcal{L}$)
2: create table ST (substring, prefix, suffix, size_range/doc_sketch)
3: create table IT (substring1, substring2)
4: create table RT (substring1, substring2, support_size)
5: **for** each record $r \in \mathcal{L}$ **do**
6:   **for** $\ell = 0$ to S **do**
7:     **for** each substring α of r.url of length ℓ **do**
8:       p := prefix of r.url preceding α
9:       s := suffix of r.url succeeding α
10:       add (α, p, s, r.size_range/r.doc_sketch) to ST
11: group tuples in ST into *buckets* by (prefix,suffix)
12: **for** each bucket B **do**
13:   **if** (|B| = 1 OR |B| > T) continue
14:   **for** each pair of distinct tuples $t_1, t_2 \in$ B **do**
15:     **if** (LikelySimilar(t_1, t_2))
16:       add (t_1.substring, t_2.substring) to IT
17: group tuples in IT into *rule_supports* by (substring1,substring2)
18: **for** each rule_support R **do**
19:   t := first tuple in R
20:   add tuple (t.substring1, t.substring2, |R|) to RT
21: sort RT by support_size
22: return all rules in RT whose support size is $\geq MS$

Figure 1: Discovering likely DUST rules.

Our algorithm for discovering likely DUST rules is described in Figure 1. The algorithm gets as input the URL list $\mathcal{L}$. We assume the URL list has been pre-processed so that: (1) only unique URLs have been kept; (2) all the URLs have been tokenized and include the preceding ^ and succeeding $; (3) all records corresponding to errors (http return codes in the 4xx and 5xx series) have been filtered out; (4) for each URL, the corresponding document sketch or size range has been recorded.

The algorithm uses three tables: a substring table ST, an instance table IT, and a rule table RT. Their attributes are listed in Figure 1. In principle, the tables can be stored in any database structure; our implementation uses text files.

In lines 5–10, the algorithm scans the URL list, and records all substrings of lengths 0 to S of the URLs in the list. For each such substring α, a tuple is added to the substring table ST. This tuple consists of the substring α, as well as its envelope (p, s), and either the URL's document sketch or its size range. The substrings are then grouped into buckets by their envelopes (line 11). Our implementation does this by sorting the file holding the ST table by the second and third attributes. Note that two substrings α, β appear in the bucket of (p, s) if and only if $(p, s) \in E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$.

In lines 12–16, the algorithm enumerates the envelopes found. An envelope (p, s) contributes 1 to the intersection of the envelope sets $E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$, for every α, β that appear in its bucket. Thus, if the bucket has only a single entry, we know (p, s) does not contribute any instance to any rule, and thus can be tossed away. If the bucket is overflowing (its size exceeds T), then (p, s) is also ignored (line 13).

In lines 14–16, the algorithm enumerates all the pairs (α, β) of substrings that belong to the bucket of (p, s). If it seems likely that the documents associated with the URLs

$p\alpha s$ and $p\beta s$ are similar (through size or document sketch matching) (line 15), (α, β) is added to the instance table IT (line 16).

The number of times a pair (α, β) has been added to the instance table is exactly the size of the set $(E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)) \setminus (O \cup D_{\alpha,\beta})$, which is our estimated support for the rules $\alpha \to \beta$ and $\beta \to \alpha$. Hence, all that is left to do is compute these counts and sort the pairs by their count (lines 17–22). The algorithm's output is an ordered list of pairs. Each pair representing two likely DUST rules (one in each direction). Only rules whose support is large enough (bigger than MS) are kept in the list.

The full draft of the paper contains the following analysis:

PROPOSITION 5.1. *Let n be the number of records in the URL list and let m be the average length (in tokens) of URLs in the URL list. The above algorithm runs in $\tilde{O}(mnST^2)$ time and uses $O(mnST^2)$ storage space, where $\tilde{O}$ suppresses logarithmic factors.*

Note that mn is the input length and S and T are usually small constants, independent of m and n. Hence, the algorithm's running time and space are only (quasi-) linear.

5.2 Eliminating redundant rules

By design, the output of the above algorithm includes many overlapping pairs. For example, when running on a forum site, our algorithm finds the pair ("`.co.il/story?id=`", "`.co.il/story_`"), as well as numerous pairs of substrings of these, such as ("`story?id=`", "`story_`"). Note that every instance of the former pair is also an instance of the latter. We thus say that the former *refines* the latter. It is desirable to eliminate redundancies prior to attempting to validate the rules, in order to reduce the cost of validation. However, when one likely DUST rule refines another, it is not obvious which should be kept. In some cases, the broader rule is always true, and all the rules that refine it are redundant. In other cases, the broader rule is only valid in specific *contexts* identified by the refining ones.

In some cases, we can use information from the URL list in order to deduce that a pair is redundant. When two pairs have exactly the same support in the URL list, this gives a strong indication that the latter, seemingly more general rule, is valid only in the context specified by the former rule. We can thus eliminate the latter rule from the list.

We next discuss in more detail the notion of *refinement* and show how to use it to eliminate redundant rules.

DEFINITION 5.2 (REFINEMENT). *A rule ϕ* refines *a rule ψ, if* $\text{support}(\phi) \subseteq \text{support}(\psi)$.

That is, ϕ refines ψ, if every instance (u_1, u_2) of ϕ is also an instance of ψ. Testing refinement for substitution rules turns out to be easy, as captured in the following lemma (proven in the full draft of the paper):

LEMMA 5.3. *A substitution rule $\alpha' \to \beta'$ refines a substitution rule $\alpha \to \beta$ if and only if there exists an envelope (γ, δ) s.t. $\alpha' = \gamma\alpha\delta$ and $\beta' = \gamma\beta\delta$.*

The characterization given by the above lemma immediately yields an efficient algorithm for deciding whether a substitution rule $\alpha' \to \beta'$ refines a substitution rule $\alpha \to \beta$: we simply check that α is a substring of α', replace α by β, and check whether the outcome is β'. If α has multiple occurrences in α', we check all of them. Note that our algorithm's input is a list of pairs rather than rules, where each pair represents two rules. When considering two pairs (α, β) and (α', β'), we check refinement in both directions.

Now, suppose a rule $\alpha' \to \beta'$ was found to refine a rule $\alpha \to \beta$. Then, $\text{support}(\alpha' \to \beta') \subseteq \text{support}(\alpha \to \beta)$, implying that also $\text{support}_{\mathcal{L}}(\alpha' \to \beta') \subseteq \text{support}_{\mathcal{L}}(\alpha \to \beta)$. Hence, if $|\,\text{support}_{\mathcal{L}}(\alpha' \to \beta')| = |\,\text{support}_{\mathcal{L}}(\alpha \to \beta)|$, then $\text{support}_{\mathcal{L}}(\alpha' \to \beta') = \text{support}_{\mathcal{L}}(\alpha \to \beta)$. If the URL list is sufficiently representative of the web site, this gives an indication that every instance of the refined rule $\alpha \to \beta$ that occurs on the web site is also an instance of the refinement $\alpha' \to \beta'$. We choose to keep only the refinement $\alpha' \to \beta'$, because it gives the full context of the substitution.

One small obstacle to using the above approach is the following. In the first phase of our algorithm, we do not compute the exact size of the support $|\,\text{support}_{\mathcal{L}}(\alpha \to \beta)|$, but rather calculate the quantity $|(E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)) \setminus (O \cup D_{\alpha,\beta})|$. It is possible that $\alpha' \to \beta'$ refines $\alpha \to \beta$ and $\text{support}_{\mathcal{L}}(\alpha' \to \beta') = \text{support}_{\mathcal{L}}(\alpha \to \beta)$, yet $|(E_{\mathcal{L}}(\alpha') \cap E_{\mathcal{L}}(\beta')) \setminus (O \cup D_{\alpha',\beta'})| < |(E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)) \setminus (O \cup D_{\alpha,\beta})|$.

In practice, if the supports are identical, the difference between the calculated support sizes should be small. We thus eliminate the refined rule, even if its calculated support size is slightly above the calculated support size of the refining rule. However, to increase the effectiveness of this phase, we run the first phase of the algorithm twice, once with a lower overflow threshold T_{low} and once with a higher overflow threshold T_{high}. While the support calculated using the lower threshold is more effective in filtering out invalid rules, the support calculated using the higher threshold is more effective in eliminating redundant rules.

The algorithm for eliminating refined rules from the list appears in Figure 2. The algorithm gets as input a list of pairs, representing likely rules, sorted by their calculated support size. It uses three tunable parameters: (1) the *maximum relative deficiency, MRD*, (2) the *maximum absolute deficiency, MAD*; and (3) the *maximum window size, MW*. MRD and MAD determine the maximum difference allowed between the calculated support sizes of the refining rule and the refined rule, when we eliminate the refined rule. MW determines how far down the list we look for refinements.

```
1: Function EliminateRedundancies(pairs_list R)
2:   for i = 1 to |R| do
3:     if (already eliminated R[i]) continue
4:     for j = 1 to min(MW, |R| − i) do
5:       if (R[i].size − R[i + j].size >
                      max(MRD · R[i].size, MAD))  break
6:       if (R[i] refines R[i + j])
7:         eliminate R[i + j]
8:       else if (R[i + j] refines R[i]) then
9:         eliminate R[i]
10:         break
11:  return R
```

Figure 2: Eliminating redundant rules.

The algorithm scans the list from top to bottom. For each rule $\mathcal{R}[i]$, which has not been eliminated yet, the algorithm scans a "window" of rules below $\mathcal{R}[i]$. Suppose s is the calculated size of the support of $\mathcal{R}[i]$. The window size is chosen so that (1) it never exceeds MW (line 4); and (2) the difference between s and the calculated support size of the lowest rule in the window is at most the maximum between $MRD \cdot s$ and MAD (line 5). Now, if $\mathcal{R}[i]$ refines a rule $\mathcal{R}[j]$ in the window, the refined rule $\mathcal{R}[j]$ is eliminated (line 7), while if some rule $\mathcal{R}[j]$ in the window refines $\mathcal{R}[i]$, $\mathcal{R}[i]$ is eliminated (line 9).

It is easy to verify that the running time of the algorithm is at most $|\mathcal{R}| \cdot MW$. In our experiments, this algorithm reduces the set of rules by over 90%.

5.3 Validating DUST rules

So far, the algorithm has generated likely rules from the URL list alone, without fetching even a single page from the web site. Fetching a small number of pages for validating or refuting these rules is necessary for two reasons. First, it can significantly improve the final precision of the algorithm. Second, the first two phases of DustBuster, which discover likely substring substitution rules, cannot distinguish between the two directions of a rule. The discovery of the pair (α, β) can represent both $\alpha \to \beta$ and $\beta \to \alpha$. This does not mean that in reality both rules are valid or invalid simultaneously. It is often the case that only one of the directions is valid; for example, in many sites removing the substring `index.html` is always valid, whereas adding one is not. Only by attempting to fetch actual page contents we can tell which direction is valid, if any.

The validation phase of DustBuster therefore fetches a small sample of web pages from the web site in order to check the validity of the rules generated in the previous phases. The validation of a single rule is presented in Figure 3. The algorithm is given as input a likely rule R and a list of URLs from the web site and decides whether the rule is valid. It uses two parameters: the *validation count,* N (how many samples to use in order to validate each rule), and the *refutation threshold,* ϵ (the minimum fraction of counterexamples to a rule required to declare the rule invalid).

1:**Function** ValidateRule(R, $\mathcal{L}$)
2: positive := 0
3: negative := 0
4: **while** (positive $< (1-\epsilon)N$ AND negative $< \epsilon N$) **do**
5: u := a random URL from $\mathcal{L}$ on which applying R results in a different URL
6: v := outcome of application of R to u
7: fetch u and v
8: **if** (fetch u failed) continue
9: **if** (fetch v failed OR DocSketch(u) $\neq$ DocSketch(v))
10 negative := negative + 1
11: **else**
12: positive := positive + 1
13: **if** (negative $\geq \epsilon N$)
14: return FALSE
15: return TRUE

Figure 3: Validating a single likely rule.

In order to determine whether a rule is valid, the algorithm repeatedly chooses random URLs from the given test URL list until hitting a URL on which applying the rule results in a different URL (line 5). The algorithm then applies the rule to the random URL u, resulting in a new URL v. The algorithm then fetches u and v. Using document sketches, such as the shingling technique of Broder *et al.* [7], the algorithm tests whether u and v are similar. If they are, the algorithm accounts for u as a positive example attesting to the validity of the rule. If v cannot be fetched, or they are not similar, then it is accounted as a negative example (lines 9–12). The testing is stopped when either the number of negative examples surpasses the refutation threshold or when the number of positive examples is large enough to guarantee the number of negative examples will not surpass the threshold.

One could ask why we declare a rule valid even if we find (a small number of) counterexamples to it. There are several reasons: (1) the document sketch comparison test sometimes makes mistakes, since it has an inherent false negative probability; (2) dynamic pages sometimes change significantly between successive probes (even if the probes are made at short intervals); and (3) the fetching of a URL may sometimes fail at some point in the middle, after part of the page has been fetched. By choosing a refutation threshold smaller than one, we can account for such situations.

Figure 4 shows the algorithm for validating a list of likely DUST rules. Its input consists of a list of pairs representing likely substring transformations, $(\mathcal{R}[i].\alpha, \mathcal{R}[i].\beta)$, and a test URL list $\mathcal{L}$.

For a pair of substrings (α, β), we use the notation $\alpha > \beta$ to denote that either $|\alpha| > |\beta|$ or $|\alpha| = |\beta|$ and α succeeds β in the lexicographical order. In this case, we say that the rule $\alpha \to \beta$ *shrinks* the URL. We give precedence to shrinking substitutions. Therefore, given a pair (α, β), if $\alpha > \beta$, we first try to validate the rule $\alpha \to \beta$. If this rule is valid, we ignore the rule in the other direction since, even if this rule turns out to be valid as well, using this rule during canonization is only likely to create cycles, i.e., rules that can be applied an infinite number of times because they cancel out each others' changes. If the shrinking rule is invalid, though, we do attempt to validate the opposite direction, so as not to lose a valid rule. Whenever one of the directions of (α, β) is found to be valid, we remove from the list all pairs refining (α, β)– once a broader rule is deemed valid, there is no longer a need for refinements thereof. By eliminating these rules prior to validating them, we reduce the number of pages we fetch. We assume that each pair in $\mathcal{R}$ is ordered so that $\mathcal{R}[i].\alpha > \mathcal{R}[i].\beta$.

Function Validate(rules_list $\mathcal{R}$, test_URLList $\mathcal{L}$)
create an empty list of rules LR
for i = 1 to $|\mathcal{R}|$ **do**
for j = 1 to i - 1 **do**
if ($\mathcal{R}[j]$ was not eliminated AND $\mathcal{R}[i]$ refines $\mathcal{R}[j]$)
eliminate $\mathcal{R}[i]$ from the list
break
if ($\mathcal{R}[i]$ was eliminated)
continue
if (ValidateRule($\mathcal{R}[i].\alpha \to \mathcal{R}[i].\beta$, $\mathcal{L}$))
add $\mathcal{R}[i].\alpha \to \mathcal{R}[i].\beta$ to LR
else if (ValidateRule($\mathcal{R}[i].\beta \to \mathcal{R}[i].\alpha$, $\mathcal{L}$))
add $\mathcal{R}[i].\beta \to \mathcal{R}[i].\alpha$ to LR
else
eliminate $\mathcal{R}[i]$ from the list
return LR

Figure 4: Validating likely rules.

The running time of the algorithm is at most $O(|\mathcal{R}|^2 + N|\mathcal{R}|)$. Since the list is assumed to be rather short, this running time is manageable. The number of pages fetched is $O(N|\mathcal{R}|)$ in the worst-case, but much smaller in practice, since we eliminate many redundant rules after validating rules they refine.

5.4 URL canonization

The canonization algorithm receives a URL u and a list of valid DUST rules $\mathcal{R}$. The idea behind this algorithm is very simple: it repeatedly applies to u all the rules in $\mathcal{R}$, until there is an iteration in which u is unchanged, or until a predetermined maximum number of iterations has been reached. For details see the full draft of the paper.

As the general canonization problem is hard, we cannot expect this polynomial time algorithm to always produce a minimal canonization. Nevertheless, our empirical study shows that the savings obtained using this algorithm are high. We believe that this common case success stems from two features. First, our policy of choosing shrinking rules whenever possible typically eliminates cycles. Second, our elimination of refinements of valid rules leaves a small set of rules, most of which do no affect each other.

6. EXPERIMENTAL RESULTS

Experiment setup. We experiment with DustBuster on four web sites: a dynamic forum site, an academic site (www.ee.technion.ac.il), a large news site (cnn.com) and a smaller news site (nydailynews.com). In the forum site, page contents are highly dynamic, as users continuously add comments. The site supports multiple domain names. Most of the site's pages are generated by the same software. The news sites are similar in their structure to many other news sites on the web. The large news site has a more complex structure, and it makes use of several sub-domains as well as URL redirections. The academic site is the most diverse: It includes both static pages and dynamic software-generated content. Moreover, individual pages and directories on the site are constructed and maintained by a large number of users (faculty members, lab managers, etc.)

In the academic and forum sites, we detect likely DUST rules from web server logs, whereas in the news sites, we detect likely DUST rules from a crawl log. Table 1 depicts the sizes of the logs used. In the crawl logs each URL appears once, while in the web server logs the same URL may appear multiple times. In the validation phase, we use random entries from additional logs, different from those used to detect the rules. The canonization algorithm is tested on yet another log, different from the ones used to detect and validate the rules.

Web Site	Log Size	Unique URLs
Forum Site	38,816	15,608
Academic Site	344,266	17,742
Large News Site	11,883	11,883
Small News Site	9,456	9,456

Table 1: Log sizes.

Parameter settings. The following DustBuster parameters were carefully chosen in all our experiments. Our empirical results suggest that these settings lead to good results (see more details in the full draft of the paper). The maximum substring length, S, was set to 35 tokens. The maximum bucket size used for detecting DUST rules, T_{low}, was set to 6, and the maximum bucket size used for eliminating redundant rules, T_{high}, was set to 11. In the elimination of redundant rules, we allowed a relative deficiency, MRD, of up to 5%, and an absolute deficiency, MAD, of 1. The maximum window size, MW, was set to 1100 rules. The value of MS, the minimum support size, was set to 3. The algorithm uses a validation count, N, of 100 and a refutation threshold, ϵ, of 5%-10%. Finally, the canonization uses a maximum of 10 iterations. Shingling [7] is used in the validation phase to determine similarity between documents.

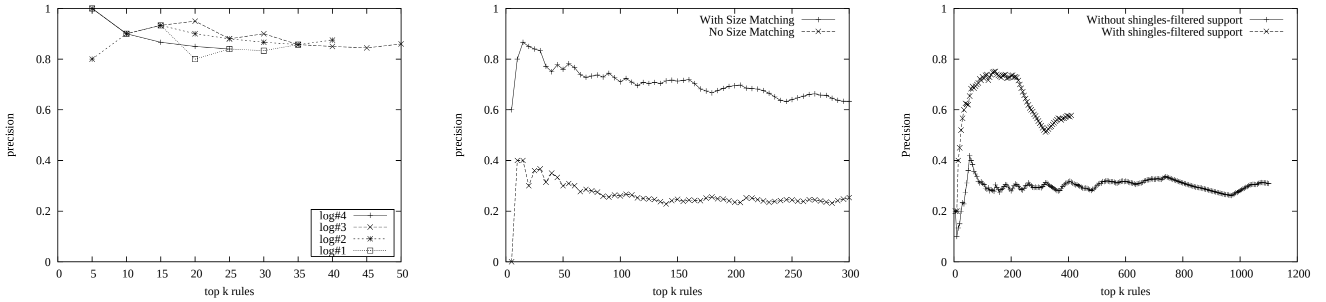
Detecting likely DUST rules and eliminating redundant ones. DustBuster's first phase scans the log and detects a very long list of likely DUST rules. Subsequently, the redundancy elimination phase dramatically shortens this list. Table 2 shows the sizes of the lists before and after redundancy elimination. It can be seen that in all of our experiments, over 90% of the rules in the original list have been eliminated.

Web Site	Rules Detected	Rules Remaining after 2nd Phase
Forum Site	402	37 (9.2%)
Academic Site	26,899	2,041 (7.6%)
Large News Site	12,144	1,243 (9.76%)
Small News Site	4,220	96 (2.3%)

Table 2: Rule elimination in second phase.

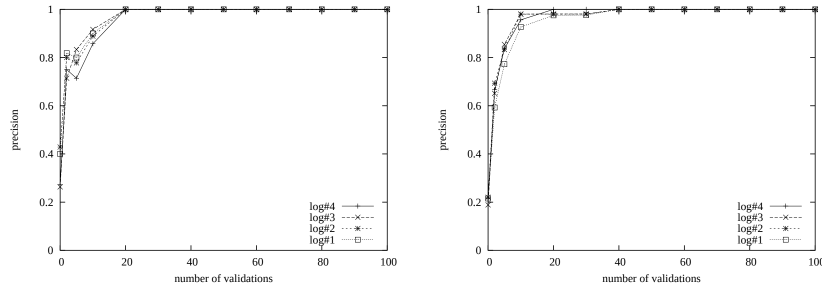
In Figure 5, we examine the precision level in the short list of likely rules produced at the end of these two phases in three of the sites. Recall that no page contents are fetched in these phases. As this list is ordered by likeliness, we examine the *precision@k*; that is, for each top k rules in this list, the curves show which percentage of them are later deemed valid (by DustBuster's validation phase) in at least one direction. We observe that, quite surprisingly, when similarity-based filtering is used, DustBuster's detection phase achieves a very high precision rate even though it does not fetch even a single page. In the forum site, out of the 40–50 detected rules, over 80% are indeed valid. In the academic site, over 60% of the 300–350 detected rules are valid, and of the top 100 detected rules, over 80% are valid. In the large news sites, 74% of the top 200 rules are valid.

This high precision is achieved, to a large extent, thanks to the similarity-based filtering (size matching or shingle matching), as shown in Figures 5(b) and 5(c). The log includes invalid rules. For example, the forum site includes multiple domains, and the stories in each domain are different. Thus, although we find many pairs `http://domain1/story_num` and `http://domain2/story_num` with the same num, these represent different stories. Similarly, the academic site has pairs like `http://site/course1/lect-num.ppt` and `http://site/course2/lect-num.ppt`, although the lectures are different. Such invalid rules are not detected, since stories/lectures typically vary in size. Figure 5(b) illustrates the impact of size matching in the academic site. We see that when size matching is not employed, the precision drops by around 50%. Thus, size matching reduces the number of accesses needed for validation. Nevertheless, size matching has its limitations– valid rules (such as "`ps`" $\rightarrow$ "`pdf`") are missed at the price of increasing precision. Figure 5(c) shows similar results for the large news site. When we do not use shingles-filtered support, the precision at the top 200 drops to 40%. Shingles-based filtering reduces the list of likely rules by roughly 70%. Most of the filtered rules turned out to be indeed invalid.



(a) Forum, 4 different logs. (b) Academic site, impact of size matching. (c) Large news site, impact of shingle matching, 4 shingles used.

Figure 5: Precision@k of likely DUST rules detected in DustBuster's first two phases *without* fetching actual content.



(a) Forum, 4 different logs. (b) Academic, 4 different logs.

Figure 6: Precision among rules that DustBuster attempted to validate vs. number of validations used (N).

Validation. We now study how many validations are needed in order to declare that a rule is valid; that is, we study what the parameter N in Figure 4 should be set to. To this end, we run DustBuster with values of N ranging from 0 to 100, and check which percentage of the rules found to be valid with each value of N are also found valid when N=100. The results from conducting this experiment on the likely DUST rules found in 4 logs from the forum site and 4 from the academic site are shown in Figure 6 (similar results were obtained for the other sites). In all these experiments, 100% precision is reached after 40 validations. Moreover, results obtained in different logs are consistent with each other.

At the end of the validation phase, DustBuster outputs a list of valid substring substitution rules without redundancies. Table 3 shows the number of valid rules detected on each of the sites. The list of 7 rules found in one of the logs in the forum site is depicted in Figure 7 below. These 7 rules or refinements thereof appear in the outputs produced using each of the studied logs. Some studied logs include 1–3 additional rules, which are insignificant (have very small support). Similar consistency is observed in the academic site outputs. We conclude that the most significant DUST rules can be adequately detected using a fairly small log with roughly 15,000 unique URLs.

Coverage. We now turn our attention to coverage, or the percentage of duplicate URLs discovered by DustBuster, in the academic site. When multiple URLs have the same document sketch, all but one of them are considered *duplicates*. In order to study the coverage achieved by DustBuster, we use two different logs from the same site: a *training log* and a *test log*. We run DustBuster on the training log in order to learn DUST rules and we then apply these rules on the test log. We count what fraction of the duplicates in the test log are covered by the detected DUST rules. We detect duplicates in the test log by fetching the contents of all of its URLs and computing their document sketches. Figure 8 classifies these duplicates. As the figure shows, 47.1% of the duplicates in the test log are eliminated by DustBuster's canonization algorithm using rules discovered on another log. The rest of the DUST can be divided among several categories: (1) duplicate images and icons; (2) replicated documents (e.g., papers co-authored by multiple faculty members and whose copies appear on each of their web pages); (3) "soft errors",

Web Site	Valid Rules Detected
Forum Site	7
Academic Site	52
Large News Site	62
Small News Site	5

Table 3: The number of rules found to be valid.

1 ".co.il/story_" → ".co.il/story?id="
2 "\&LastView=\&Close=" → ""
3 ".php3?" → "?"
4 ".il/story_" → ".il/story.php3?id="
5 "\&NewOnly=1\&tvqz=2" → "\&NewOnly=1"
6 ".co.il/thread_" → ".co.il/thread?rep="
7 "http://www.../story_" → "http://www.../story?id="

Figure 7: The valid rules detected in the forum site.

i.e., pages with no meaningful content, such as error message pages, empty search results pages, etc.

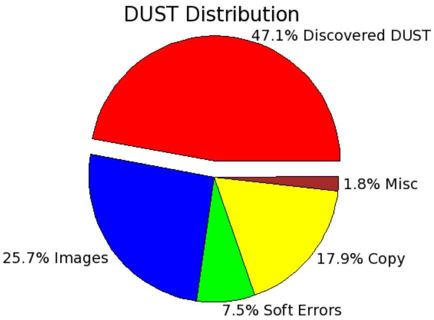


Figure 8: DUST classification, academic.

Savings in crawl size. The next measure we use to evaluate the effectiveness of the method is the discovered redundancy, i.e., the percent of the URLs we can avoid fetching in a crawl by using the DUST rules to canonize the URLs. To this end, we performed a full crawl of the academic site, and recorded in a list all the URLs fetched. We performed canonization on this list using DUST rules learned from the crawl, and counted the number of unique URLs before (U_b) and after (U_a) canonization. The discovered redundancy is then given by $\frac{U_b - U_a}{U_b}$. We found this redundancy to be 18% (see Table 4), meaning that the crawl could have been reduced by that amount. In the two news sites, the DUST rules were learned from the crawl logs and we measured the reduction that can be achieved in the next crawl. By setting a slightly more relaxed refutation threshold ($\epsilon = 10\%$), we obtained reductions of 26% and 6%, respectively. In the case of the forum site, we used four logs to detect DUST rules, and used these rules to reduce a fifth log. The reduction achieved in this case was 4.7%.

Web Site	Reduction Achieved
Academic Site	18%
Small News Site	26%
Large News Site	6%
Forum Site(using logs)	4.7%

Table 4: Reductions in crawl size.

7. CONCLUSIONS

We have introduced the problem of mining site-specific DUST rules. Knowing about such rules can be very useful for search engines: It can reduce crawling overhead by up to 26% and thus increase crawl efficiency. It can also reduce indexing overhead. Moreover, knowledge of DUST rules is essential for canonizing URL names, and canonical names are very important for statistical analysis of URL popularity based on PageRank or traffic. We presented DustBuster, an algorithm for mining DUST very effectively from a URL list. The URL list can either be obtained from a web server log or a crawl of the site.

Acknowledgments. We thank Tal Cohen and the forum site team, and Greg Pendler and the `http://ee.technion.ac.il` admins for providing us with access to web logs and for technical assistance. We thank Israel Cidon, Yoram Moses, and Avigdor Gal for their insightful input.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th VLDB*, pages 487–499, 1994.

[2] Z. Bar-Yossef, I. Keidar, and U. Schonfeld. Do not crawl in the DUST: different URLs with similar text. Technical Report CCIT Report #601, Dept. Electrical Engineering, Technion, 2006.

[3] K. Bharat and A. Z. Broder. Mirror, Mirror on the Web: A Study of Host Pairs with Replicated Content. *Computer Networks*, 31(11–16):1579–1590, 1999.

[4] K. Bharat, A. Z. Broder, J. Dean, and M. R. Henzinger. A comparison of techniques to find mirrored hosts on the WWW. *IEEE Data Engin. Bull.*, 23(4):21–26, 2000.

[5] M. Bognar. A survey on abstract rewriting. Available online at: `www.di.ubi.pt/~desousa/1998-1999/logica/mb.ps`, 1995.

[6] S. Brin, J. Davis, and H. Garcia-Molina. Copy Detection Mechanisms for Digital Documents. In *Proc. 14th SIGMOD*, pages 398–409, 1995.

[7] A. Z. Broder, S. C. Glassman, and M. S. Manasse. Syntactic clustering of the web. In *Proc. 6th WWW*, pages 1157–1166, 1997.

[8] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding replicated web collections. In *Proc. 19th SIGMOD*, pages 355–366, 2000.

[9] E. Di Iorio, M. Diligenti, M. Gori, M. Maggini, and A. Pucci. Detecting Near-replicas on the Web by Content and Hyperlink Analysis. In *Proc. 11th WWW*, 2003.

[10] F. Douglis, A. Feldman, B. Krishnamurthy, and J. Mogul. Rate of change and other metrics: a live study of the world wide web. In *Proc. 1st USITS*, 1997.

[11] H. Garcia-Molina, L. Gravano, and N. Shivakumar. dscam: Finding document copies across multiple databases. In *Proc. 4th PDIS*, pages 68–79, 1996.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[13] Google Inc. Google sitemaps. `http://sitemaps.google.com`.

[14] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and COmputational Biology.* Cambridge University Press, 1997.

[15] T. C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *J. Amer. Soc. Infor. Sci. Tech.*, 54(3):203–215, 2003.

[16] N. Jain, M. Dahlin, and R. Tewari. Using bloom filters to refine web search results. In *Proc. 7th WebDB*, pages 25–30, 2005.

[17] T. Kelly and J. C. Mogul. Aliasing on the world wide web: prevalence and performance implications. In *Proc. 11th WWW*, pages 281–292, 2002.

[18] S. J. Kim, H. S. Jeong, and S. H. Lee. Reliable evaluations of URL normalization. In *Proc. 4th ICCSA*, pages 609–617, 2006.

[19] H. Liang. A URL-String-Based Algorithm for Finding WWW Mirror Host. Master's thesis, Auburn University, 2001.

[20] F. McCown and M. L. Nelson. Evaluation of crawling policies for a web-repository crawler. In *Proc. 17th HYPERTEXT*, pages 157–168, 2006.

[21] U. Schonfeld, Z. Bar-Yossef, and I. Keidar. Do not crawl in the DUST: different URLs with similar text. In *Proc. 15th WWW*, pages 1015–1016, 2006.

[22] N. Shivakumar and H. Garcia-Molina. Finding Near-Replicas of Documents and Servers on the Web. In *Proc. 1st WebDB*, pages 204–212, 1998.