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

Ziv Bar-Yossef† Idit Keidar‡ Uri Schonfeld§

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.

1 Introduction

The DUST problem. The web is abundant with DUST: *Different URLs with Similar Text* [1, 2, 3, 4]. 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

*An extended abstract of this paper appeared at the 16th International World-Wide Web Conference, 2007.

†Department of Electrical Engineering, Technion, Haifa 32000, Israel. Google Haifa Engineering Center, Israel. Email: zivby@ee.technion.ac.il.

‡Department of Electrical Engineering, Technion, Haifa 32000, Israel. Email: idish@ee.technion.ac.il.

§Department of Computer Science, University of California Los Angeles, CA 90095, USA. Email: shuri@shuri.org.

`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. In particular, information about many redundant URLs can be represented succinctly in the form of a short list of rules. Once the rules are obtained, they can be used to avoid fetching duplicate pages altogether, including pages that were never crawled before. The rules can be obtained after crawling a small subset of a web site, or may be retained from a previous crawl of the same site. The latter is particularly useful in dynamic web sites, like blogs and news sites, where new pages are constantly added. Finally, the use of rules is robust to web site structure changes, since the rules can be validated anew before each crawl by fetching a small number of pages. For example, in a crawl of a small news site we examined, the number of URLs fetched would have been reduced by 26%.

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 indexing, and caching [1, 2], and increasing the accuracy of page metrics, like PageRank.

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. The fact DustBuster's input is a list of URLs rather than a collection of web pages significantly reduces its running time and storage requirements.

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 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 $\alpha \rightarrow \beta$ 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 $\alpha \rightarrow \beta$ 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

[1] Increasingly many web server logs are available nowadays to search engines via protocols like Google Sitemaps [5].

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 *valid* DUST rules, meaning their support includes many instances, URL pairs, whose associated documents are not similar. For example, in one site we found an invalid DUST rule, "`movie-forum`" → "`politics-forum`" whose instances included pairs of URLs of the form: `http://movie-forum.com/story_<num>` and `http://politics-forum.com/story_<num>`. In this case the URLs were associated with two unrelated stories that happen to share the same "story id" number. Another example is the rule "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 since the pairs of URLs are not associated with similar documents. 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 [6] 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, without considering the content itself, 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. The problem of finding a canonical set of URLs for a given URL list is NP-hard due to reducability to the minimum dominating set problem. Despite this, 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

2.1 Canonization rules

Global canonization rules. The most obvious and naive way DUST is being dealt with today is through standard canonization. URLs have a very standard structure [7]. The hostname may have many different aliases. Different hostnames may return the exact same site. For example adding a "`www`" to the base hostname often returns the same content. Choosing one hostname to identify each site is a standard way to canonize a URL. Other standard canonization techniques include replacing a "`//`" with a single "`/`" and removing the index.html suffix. However, site specific DUST rules cannot be detected using these simple rules.

Site-specific canonization rules. Another method for discovering DUST rules is to examine the web server configuration file and file system. In the configuration, file alias rules are defined. Each such rule allows a directory in the web server to be accessed using a different name, an alias. By parsing the web server configuration file [8] one could easily learn these rules. Further inspection of the file system may uncover symbolic links. These too have the exact same effect.

There are three main problems with using this technique. The main problem is that symbolic links and aliases are by no means the sole source of DUST rules. Other sources include parameters that do not affect the content, different dynamic files that produce the same content and many others. Our technique, therefore, can discover a wider range of DUST rules, regardless of cause. The second problem is that each configuration file is different according to the type and version of the web server. The third and final problem is that once the files are parsed and processed the rules discovered would have to be transferred to the search engine. Transferring these rules from the web server to the search engine may be possible in the future if new protocols are defined and adopted. The Google Sitemaps architecture [5] defines a protocol for web site managers to supply information about their sites to search engines. This type of protocol can be extended to enable the web server to send canonization rules to any party that wants such information. However, such an extension has not been adopted yet.

2.2 Detecting similarity between documents

The standard way of dealing with DUST is using document sketches [9, 10, 6, 11, 12, 13, 14, 15, 16, 17], 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.

2.3 Detecting mirror sites

One common source of near-duplicate content is mirroring. A number of previous works have dealt with automatic detection of mirror sites on the web. There are two basic approaches to mirror detection. The first method is based on the content itself [18, 1]. The documents are downloaded and processed. Mirrors are detected by processing the documents to detect the similarity between hosts. We will call these techniques *"bottom-up"*.

The second method uses meta information that may already be available such as a URL list, the IP number associated with the host names and other techniques. We will call these techniques *"top-down"* [19, 20, 21]. These techniques are closer to what we are doing in this paper.

In contrast to mirror detection, we deal with the complementary problem of detecting DUST within one site. Mirror detection may exploit syntactic analysis of URLs and limited sampling as we do. However, 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 focuses on comparing a given pair of sites, and only needs to determine *whether* they are mirrors.

2.4 Analysis of web server logs

Various commercial tools as well as papers are available on analyzing web server logs. We are not aware of any previous algorithm for automatically detecting DUST rules nor of any algorithm for harvesting information for search engines from web server logs. Some companies (e.g., [22, 23, 24]) have tools for analyzing web server logs, but their goals are very different from ours. This type of software usually provides such statistics as: popular keyword terms, entry pages (first page users hit), exit pages (pages users use to move to another site), information about visitor paths and many more.

2.5 Mining association rules

Our problem may seem similar to mining association rules [25], 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.

2.6 Abstract Rewrite System

One way to view our work is as producing an Abstract Rewrite System (ARS) [26] 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, every URL is prepended with the special token ^ and is appended 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 *Jaccard similarity coefficient* measure [27], which can be estimated using shingles, or document sketches due to Broder *et al.* [6].

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.

We discuss two types of DUST rules: substring substitutions and parameter substitutions. *Parameter substitution rules* either replace the value of a certain parameter appearing in the URL with a default value, or omit this parameter from the URL altogether. 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.

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 \$.) In Section 5.4 we will see that *applying* a substring substitution rule is simply done by replacing the first occurrence of the substring. However, at this point, instances of substring substitution rules are simply 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$`" → "`$`". This rule demonstrates that substring substition rules can be used to remove "irrelevant" segments of the URL, segments that if removed from a valid URL result in another valid URL with similar associated content.

The two types of rules we discuss in this paper are in no way complete. Indeed, it is easy to think of additional types of DUST rules that are not covered by the types of rules we present here. However, our experiments over real-world data show that the rules we explore are highly effective at uncovering DUST. It would be interesting to explore more types of DUST rules and compare their effectiveness in future work.

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.

The input of the first algorithm is a list of URLs, (typically from the same web site), and its output is a list of dust rules corresponding to these URLs. The *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. he URL list is a sample of the URLs that belong to the web site but is not required to be a *random* sample. For example, web server logs might be biased towards popular URLs.

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 ordering 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 $\mathcal{R}$. This definition is made somewhat more difficult by the fact that a DUST rule may map a URL to multiple URLs. For example, the URL `http://a.b.a.com/` is mapped by the rule "a." $\rightarrow$ "" to both `http://a.b.com/` and `http://b.a.com/`. To this end, we define a standard way of applying each rule. For example, in the case of substring substitions, we only replace the first occurrence of the string.

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 $\phi(u_1) = u_2$. Note that adjacent URLs in $G_{\mathcal{R}}$ correspond to similar documents. Further note that due to our rule validation process (see Section 5.4), R cannot contain both a rule and its inverse. Nevertheless, the graph G_R may still contain cycles.

For the purpose of canonization, we assume that document dissimilarity empirically respects at least a weak form of the triangle inequality, so that URLs that are connected by short paths in $G_{\mathcal{R}}$ are similar 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 the *reverse graph* of $G_{\mathcal{R}}$, the graph obtained by reversing the direction of all edges. A canonization is any mapping $C : U_S \rightarrow CU_S$ 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. [28]). 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. Therefore, recall is not directly measurable in our case, and is replaced by coverage.

The coverage measure is equivalent to recall over the set of duplicate URLs.

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 likeliness 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 at least one 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, where the URLs 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 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 [29]. 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. Rules where one of the strings is empty are actually very rare since another rule with additional context would be detected instead. For example, instead of detecting the rule "`/`" $\rightarrow$ "" we would detect the rule "`/$`" $\rightarrow$ "`$`" that would only remove the last slash. For the semiperiodic case, let us consider the following 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 $E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$ and with two envelopes in $E_{\mathcal{L}}(\gamma) \cap E_{\mathcal{L}}(\delta)$ 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.

Proof of Theorem 4.1 To prove the identity, we will show a 1-1 mapping from $\text{support}_{\mathcal{L}}(\alpha \rightarrow \beta)$ onto $E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$. Let (u_1, u_2) be any instance of the rule $\alpha \rightarrow \beta$ that occurs in $\mathcal{L}$. By Definition 3.1, there exists an envelope (p, s) so that $u_1 = p\alpha s$ and $u_2 = p\beta s$. Note that $(p, s) \in E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$, hence we define our mapping as: $f_{\alpha,\beta}(u_1, u_2) = (p, s)$. The main challenge is to prove that $f_{\alpha,\beta}$ is a well defined function; that is, $f_{\alpha,\beta}$ maps every instance (u_1, u_2) to a single pair (p, s). This is captured by the following lemma whose proof is provided below:

Lemma 4.2 *Let $\alpha \neq \beta$ be two distinct, non-empty, and non-semiperiodic strings. Then, there cannot be two distinct pairs $(p_1, s_1) \neq (p_2, s_2)$ s.t. $p_1\alpha s_1 = p_2\alpha s_2$ and $p_1\beta s_1 = p_2\beta s_2$.*

We are left to show that f is 1-1 and onto. Take any two instances (u_1, u_2),(v_1, v_2) of (α, β), and suppose that $f_{\alpha,\beta}(u_1, u_2) = f_{\alpha,\beta}(v_1, v_2) = (p, s)$. This means that $u_1 = p\alpha s = v_1$ and $u_2 = p\beta s = v_2$. Hence, necessarily $(u_1, u_2) = (v_1, v_2)$, implying f is 1-1. Take now any envelope $(p, s) \in E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$. By definition, there exist URLs $u_1, u_2 \in \mathcal{L}$, so that $u_1 = p\alpha s$ and $u_2 = p\beta s$. By Definition 3.1, (u_1, u_2) is an instance of the rule $\alpha \rightarrow \beta$, and thus $f_{\alpha,\beta}(u_1, u_2) = (p, s)$.

□

In order to prove Lemma 4.2, we show the following basic property of semiperiodic strings:

Lemma 4.3 *Let $\alpha \neq \beta$ be two distinct and non-empty strings. If β is both a suffix and a prefix of α, then α must be semiperiodic.*

Proof Since β is both a prefix and a suffix of α and since $\alpha \neq \beta$, there exist two non-empty strings β_0 and β_2 s.t.

$$\alpha = \beta_0\beta = \beta\beta_2.$$

Let $k = \lfloor \frac{|\alpha|}{|\beta_0|} \rfloor$. Note that $k \geq 1$, as $|\alpha| \geq |\beta_0|$. We will show by induction on k that $\alpha = \beta_0^k\beta_0'$, where β_0' is a possibly empty prefix of β_0.

The induction base is $k = 1$. In this case, $|\beta_0| \geq \frac{|\alpha|}{2}$, and as $\alpha = \beta_0\beta$, $|\beta| \leq |\beta_0|$. Since $\beta_0\beta = \beta\beta_2$, it follows that β is a prefix of β_0. Thus, define $\beta_0' = \beta$ and we have:

$$\alpha = \beta_0\beta = \beta_0\beta_0'.$$

Assume now that the statement holds for $\lfloor \frac{|\alpha|}{|\beta_0|} \rfloor = k-1 \geq 1$ and let us show correctness for $\lfloor \frac{|\alpha|}{|\beta_0|} \rfloor = k$. As $\alpha = \beta_0\beta$, then $\lfloor \frac{|\beta|}{|\beta_0|} \rfloor = k-1 \geq 1$. Hence, $|\beta| \geq |\beta_0|$ and thus β_0 must be a prefix of β (recall that $\beta_0\beta = \beta\beta_2$). Let us write β as:

$$\beta = \beta_0\beta_1.$$

We thus have the following two representations of α:

$$\alpha = \beta_0\beta = \beta_0\beta_0\beta_1 \text{ and } \alpha = \beta\beta_2 = \beta_0\beta_1\beta_2.$$

We conclude that:

$$\beta = \beta_1\beta_2.$$

We thus found a string (β_1), which is both a prefix and a suffix of β. We therefore conclude from the induction hypothesis that

$$\beta = \beta_0^{k-1}\beta_0',$$

for some prefix β_0' of β_0. Hence,

$$\alpha = \beta_0\beta = \beta_0^k\beta_0'.$$

□

We can now prove Lemma 4.2:

Proof of Lemma 4.2 Suppose, by contradiction, that there exist two different pairs (p_1, s_1) and (p_2, s_2) so that:

$$u_1 = p_1\alpha s_1 = p_2\alpha s_2 \tag{1}$$

$$u_2 = p_1\beta s_1 = p_2\beta s_2. \tag{2}$$

These equations together with the fact that $(p_1, s_1) \neq (p_2, s_2)$ imply that both $p_1 \neq p_2$ and $s_1 \neq s_2$. Thus, by Equations (1) and (2), one of p_1 and p_2 is a proper prefix of the other. Suppose, for example, that p_1 is a proper prefix of p_2 (the other case is identical). It follows that s_2 is a proper suffix of s_1.

Let us assume, without loss of generality, that $|\alpha| \geq |\beta|$. There are two cases to consider:

Case (i): Both $p_1\alpha$ and $p_1\beta$ are prefixes of p_2. This implies that also $\alpha s_2, \beta s_2$ are suffixes of s_1.

Case (ii): At least one of $p_1\alpha$ and $p_1\beta$ is not a prefix of p_2.

Case (i): In this case, p_2 can be expressed as:

$$p_2 = p_1\alpha p_2' = p_1\beta p_2'',$$

for some strings p_2' and p_2''. Thus, since $|\alpha| \geq |\beta|$, β is a prefix of α. Similarly:

$$s_1 = s_1'\alpha s_2 = s_1''\beta s_2.$$

Thus, β is also a suffix of α. According to Lemma 4.3, α is therefore semiperiodic. A contradiction.

Case (ii): If $p_1\alpha$ is not a prefix of p_2, α can be written as $\alpha = \alpha_0\alpha_1 = \alpha_1\alpha_2$, for some non-empty strings $\alpha_0, \alpha_1, \alpha_2$, as shown in Figure 1. By Lemma 4.3, α is semiperiodic. A contradiction. The case that $p_1\beta$ is not a prefix of p_2 is handled in a similar manner.

□

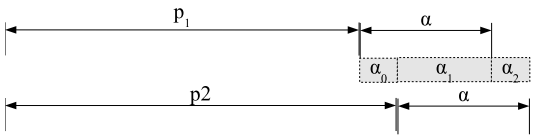


Figure 1: Breakdown of α in Lemma 4.2.

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 $E_{\mathcal{L}}(\alpha_1)$, $E_{\mathcal{L}}(\alpha_2)$,$\ldots$,$E_{\mathcal{L}}(\alpha_k)$, 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.* [6], 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 highly dynamic sites like forum 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 to be similar if the intervals I_{u_1} and I_{u_2} overlap. Note however, that in the case of size matching it is more accurate to say that the URLs are unlikely to be similar if their intervals do not overlap. For this reason the size matching heuristic is effective in filtering support but not as a measure of how similar two URLs are. 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. 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 $\alpha \rightarrow \beta$ 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 tokens, 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 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 ∈ L do
6:    for ℓ = 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 t1,t2 ∈ B do
15:     if (LikelySimilar(t1, t2))
16:       add (t1.substring, t2.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 ≥ MS
```

Figure 2: Discovering likely DUST rules.

Our algorithm for discovering likely DUST rules is described in Figure 2. 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 2. 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 $\alpha \rightarrow \beta$ and $\beta \rightarrow \alpha$. 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.

Complexity analysis. 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. We assume tokens are of constant length. The size of the URL list is then $O(mn)$ bits. We use $\tilde{O}()$ to suppress factors that are logarithmic in n,m,S, and T, which are typically negligible.

The computation has two major bottlenecks. The first is filling in the instance table IT (lines 12–16). The elements of the ST table, $O(mnS)$ are split into buckets of size $O(T)$ which results in $O(\frac{mnS}{T})$ buckets in ST. The algorithm then enumerates all the buckets, and for each of them enumerates each pair of substrings in the bucket. This computation could possibly face a quadratic blowup. Yet, since overflowing buckets are ignored, then this step takes only $O(\frac{mnS}{T}T^2) = O(mnST)$ time. The second bottleneck is sorting the URL list and the intermediate tables. Since all the intermediate tables are of size at most $O(mnST)$, the sorting can be carried in $\tilde{O}(mnST)$ time and $O(mnST)$ external storage space. By using an efficient external storage sort utility, we can keep the main memory complexity $\tilde{O}(1)$ rather than linear. The algorithm does not fetch any pages.

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 5.1 (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:

Lemma 5.2 *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* $\beta' = \gamma\beta\delta$.

Proof We prove derivation in both directions. Assume, initially, that there exists an envelope (γ, δ) s.t. $\alpha' = \gamma\alpha\delta$ and $\beta' = \gamma\beta\delta$. We need to show that in this case $\alpha' \to \beta'$ refines $\alpha \to \beta$. Take, then, any instance (u_1, u_2) of $\alpha' \to \beta'$. By Definition 3.1, there exists an envelope (p', s') s.t. $u_1 = p'\alpha's'$ and $u_2 = p'\beta's'$. Hence, if we define $p = p'\gamma$ and $s = \delta s'$, then we have that $u_1 = p\alpha s$ and $u_2 = p\beta s$. Using again Definition 3.1, we conclude that (u_1, u_2) is also an instance of $\alpha \to \beta$. This proves the first direction.

For the second direction, assume that $\alpha' \to \beta'$ refines $\alpha \to \beta$. Assume that none of $\alpha, \beta, \alpha', \beta'$ starts with ^ or ends with \$. (The extension to $\alpha, \beta, \alpha', \beta'$ that can start with ^ or end with \$ is easy, but requires some technicalities, that would harm the clarity of this proof.) Define $u_1 = \text{^}\alpha'\$$ and $u_2 = \text{^}\beta'\$$. By Definition 3.1, (u_1, u_2) is an instance of $\alpha' \to \beta'$. Due to refinement, it is also an instance of $\alpha \to \beta$. Hence, there exist p, s s.t. $u_1 = p\alpha s$ and $u_2 = p\beta s$. Hence, $p\alpha s = \text{^}\alpha'\$$ and $p\beta s = \text{^}\beta'\$$. Since α, β do not start with ^ and do not end with \$, then p must start with ^ and s must end with \$. Define then γ to be the string p excluding the leading ^, and define δ to be the string s excluding the trailing \$. We thus have: $\alpha' = \gamma\alpha\delta$ and $\beta' = \gamma\beta\delta$, as needed.

□

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, support$(\alpha' \to \beta') \subseteq$ support$(\alpha \to \beta)$, implying that also support$_{\mathcal{L}}(\alpha' \to \beta') \subseteq$ support$_{\mathcal{L}}(\alpha \to \beta)$. Hence, if $|\text{support}_{\mathcal{L}}(\alpha' \to \beta')| = |\text{support}_{\mathcal{L}}(\alpha \to \beta)|$, then support$_{\mathcal{L}}(\alpha' \to \beta') =$ 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 support$_{\mathcal{L}}(\alpha' \to \beta') =$ support$_{\mathcal{L}}(\alpha \to \beta)$, yet $|(E_{\mathcal{L}}(\alpha') \cap E_{\mathcal{L}}(\beta')) \setminus (O \cup D_{\alpha',\beta'})| < |(E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)) \setminus (O \cup D_{\alpha,\beta})|$.

How could this happen? Consider some envelope $(p,s) \in E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$, and let $(u_1,u_2) = (p\alpha s, p\beta s)$ be the corresponding instance of $\alpha \to \beta$. Since support$_{\mathcal{L}}(\alpha' \to \beta') =$ support$_{\mathcal{L}}(\alpha \to \beta)$, then (u_1,u_2) is also an instance of $\alpha' \to \beta'$. Therefore, there exists some envelope $(p',s') \in E_{\mathcal{L}}(\alpha') \cap E_{\mathcal{L}}(\beta')$ s.t. $u_1 = p'\alpha's'$ and $u_2 = p'\beta's'$.

α is a substring of α' and β is a substring of β', thus p' must be a prefix of p and s' must be a suffix of s. This implies that any URL that contributes a substring γ to the bucket of (p,s) will also contribute a substring γ' to the bucket of (p',s'), unless γ' exceeds the maximum substring length S. In principle, then, we should expect the bucket of (p',s') to be larger than the bucket of (p,s). If the maximum bucket size T happens to be exactly between the sizes of the two buckets, then (p',s') overflows while (p,s) does not. In this case, the first phase of DustBuster will account for (u_1,u_2) in the computation of the support of $\alpha \to \beta$ but not in the computation of the support of $\alpha' \to \beta'$, incurring a difference between the two.

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 3. 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.

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

```
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
14:  return R
```

Figure 3: Eliminating redundant rules.

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 Parameter substitutions

Inline parameters in URLs typically comply with a standard format. In many sites, an inline parameter name is preceded by the "?" or "&" characters and followed by the "=" character and the parameter's value, which is followed by either the end of the URL or another "&" character. We can therefore employ a simple regular expression search on URLs in the URL list in order to detect popular parameters, along with multiple examples of values for each parameter. Having detected the parameters, we check for each one whether replacing its value with an arbitrary one is a valid DUST rule. To this end, we exploit the ST table computed by DustBuster (see Figure 2), after it has been sorted and divided into buckets. We seek buckets whose prefix attribute ends with the desired parameter name, and then compare the document sketches or size ranges of the relevant URLs pertaining to such buckets.

For each parameter, p, we choose some value of the parameter , v_p, and add two rules to the list of likely rules: the first, replaces the value of the parameter p with v_p, the second rule, omits the parameter altogether. Due to the simplicity of this algorithm, its detailed presentation is omitted. The next section describes the validation phase which will drop the DUST rules which do not generate valid URLs or URLs with similar content.

5.4 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 $\alpha \rightarrow \beta$ and $\beta \rightarrow \alpha$. 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 4. 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, L)
2:  positive := 0
3:  negative := 0
4:  while (positive < (1 − ε)N AND negative < εN) do
5:    u := a random URL from 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) ≠ DocSketch(v))
10      negative := negative + 1
11:   else
12:     positive := positive + 1
13:   if (negative ≥ εN )
14:     return FALSE
15:  return TRUE
```

Figure 4: Validating a single likely rule.

REMARK. The application of R to u (line 6) may result in several different URLs. For example, there are several ways of replacing the string "people" with the string "users" in the URL `http://people.domain.com/people`, resulting in the URLs `http://users.domain.com/people`, `http://people.domain.com/users`, and `http://users.domain.com/users`. Our policy is to select one standard way of applying a rule. For example, in the case of substring substitutions, we simply replace the first occurrence of the substring.

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.* [6], 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) counterex-

amples 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.

Each parameter substitution rule is validated using the code in Figure 4. The validation of substring substitutions is more complex, as it needs to address directions and refinements.

Figure 5 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 $\alpha \rightarrow \beta$ *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 $\alpha \rightarrow \beta$. 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 $\mathcal{R}[i].\alpha > \mathcal{R}[i].\beta$.

```
1:Function Validate(rules_list R, test_URLList L)
2   create an empty list of rules LR
3:  for i = 1 to |R| do
4:    for j = 1 to i - 1 do
5:      if (R[j] was not eliminated AND R[i] refines R[j])
6:        eliminate R[i] from the list
7:        break
8:    if (R[i] was eliminated)
9:      continue
10:   if  (ValidateRule(R[i].α → R[i].β, L))
11:     add R[i].α → R[i].β to LR
12:   else if  (ValidateRule(R[i].β → R[i].α, L))
13:     add R[i].β → R[i].α to LR
14:   else
15:     eliminate R[i] from the list
16: return LR
```

Figure 5: 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.

Application for URL canonization. Finally, we explain how the discovered DUST rules may be used for canonization of a URL list. Our canonization algorithm is described in Figure 6. The algorithm receives a URL u and a list of valid DUST rules, $\mathcal{R}$. The idea behind this algorithm is very simple: in each iteration, each rule in $\mathcal{R}$ in turn is repeatedly applied to u up to *MA* times, until the rule does not change the URL; this process is repeated up to *MA* times, until there is an iteration in which u is unchanged (lines 6–7).

```
1:Function Canonize(URL u, rules_list R)
2:  for k = 1 to MA do
3:    prev := u
3:    for i = 1 to |R| do
4:      for j = 1 to MA do
5:        u := A URL obtained by applying R[i] to u
6:    if (prev = u)
7:      break
8:  output u
```

Figure 6: Canonization algorithm.

If a rule can be applied more than once (e.g., because the same substring appears multiple times in the URL), then each iteration in lines 4-5 applies it in the first place in the URL where it is applicable. As long as the number of occurrences of the replaced substring in the URL does not exceed MA, the algorithm replaces all of them.

We limit the number of iterations of the algorithm and applications of a rule to the parameter *MA*, because otherwise the algorithm could have entered an infinite loop (if the graph $G_{\mathcal{R}}$ contains cycles). Since *MA* is a constant, chosen independently of the number of rules, the algorithm's running time is *linear* in the number of rules. Recall that the general canonization problem is hard, so we cannot expect this algorithm to always produce a minimum size canonization. Nevertheless, our empirical study shows that the savings obtained using this algorithm are high.

We believe that the algorithm's 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 not affect each other.

6 Experimental Results

Experiment setup. We experiment with DustBuster on four web sites: a dynamic forum site[2], 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 and 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. Finally, the academic site is the most diverse:

[2]The webmaster who gave us access to the logs asked us not to specify the name of the site.

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 set of logs, 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

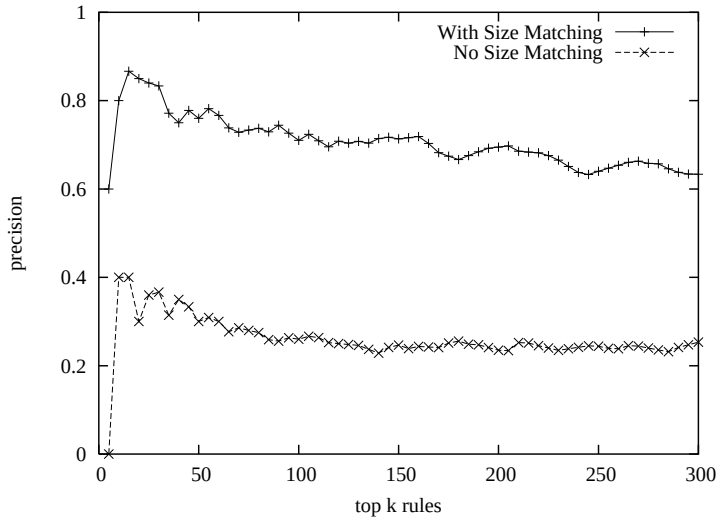
Table 1: Log sizes.

Parameter settings. The following DustBuster parameters were carefully chosen in all our experiments. Our empirical results suggest that these settings are robust across data sets, as they work in the 4 very different representative sites we experimented with. 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 [6] is used in the validation phase to determine similarity between documents.

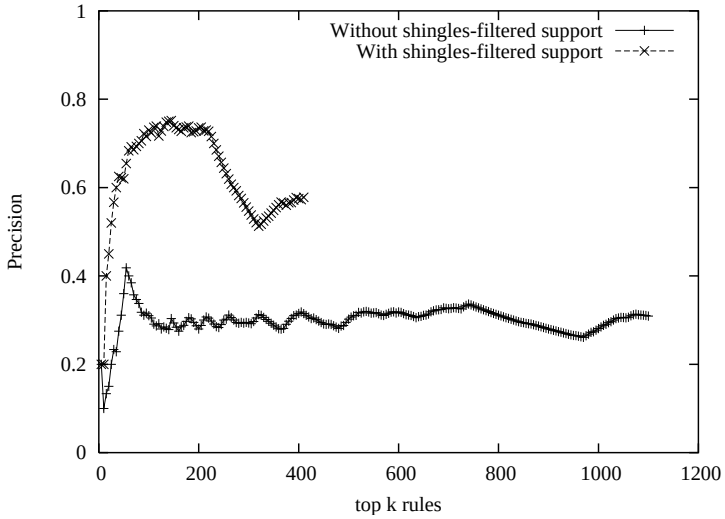
Detecting likely DUST rules and eliminating redundant ones. DustBuster's first phase scans the log and detects a very long list of likely DUST rules. Subsequently, the redundancy elimination phase dramatically shortens this list. Table 2 shows the sizes of the lists before and after redundancy elimination.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

For example, in the largest log in the academic site, 26,899 likely rules were detected in the first phase, and only 2041 (8%) remained after the second; in a smaller log 10,848 rules were detected, of which only 354 (3%) were not eliminated. In the large news site 12,144 were detected, 1243 remained after the second phase. In the forum site, much fewer likely rules were detected, e.g., in one log 402 rules were found, of which 37 (9%) remained. We believe that the smaller number of rules is a result of the forum site being more uniformly structured than the academic one, as most of its pages are generated by the same web server software.

In Figure 7, 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



(a) Academic site, impact of size matching.



(b) Large news site, impact of shingle matching, 4 shingles used.

Figure 7: Precision@k of likely DUST rules detected in DustBuster's first two phases *without* fetching actual content.

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.

DustBuster's validation phase) in at least one direction. We observe that 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. Figure 8 shows the results for four web server logs of 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.

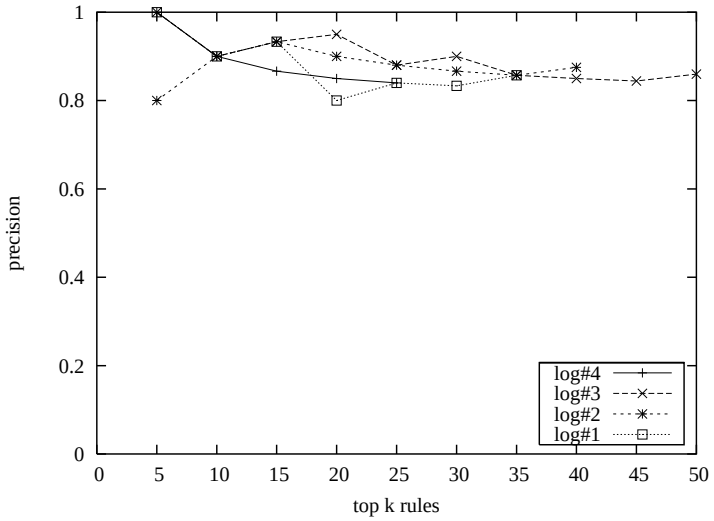
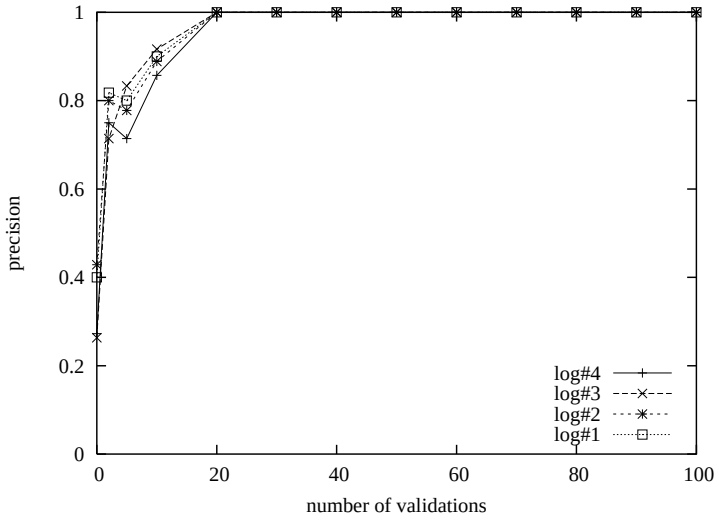


Figure 8: Forum site showing 4 different logs, precision@k of likely DUST rules detected in DustBuster's first two phases *without* fetching actual content.

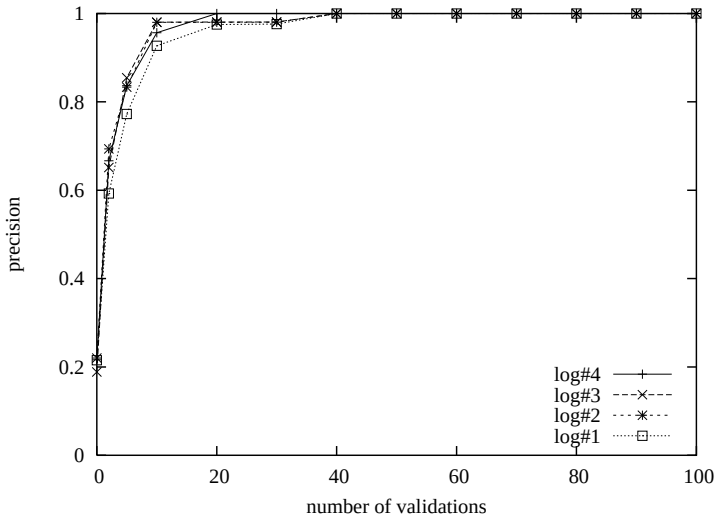
This high precision is achieved, to a large extent, thanks to the similarity-based filtering (size matching or shingle matching), as shown in Figures 7(a) and 7(b). The log includes invalid rules. For example, the forum site includes multiple domains, and the stories in each domain are different. Thus, although we find many pairs of the form `http://domain1/story_num` and `http://domain2/story_num` with the same story number, these URLs represent different stories. Similarly, the academic site has URL pairs of the form `http://site/course1/lect-num.ppt` and `http://site/course2/lect-num.ppt`, although the lectures are different. These URL pairs are instances of invalid rules, rules that are are not detected thanks to size matching. Figure 7(a) 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`" → "`pdf`") are missed at the price of increasing precision. Figure 7(b) 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.

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 5 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 9 (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.

In these graphs, we only consider rules that DustBuster attempts to validate. Since many valid rules are removed (in line 6 of Figure 5) after rules that they refine are deemed valid, the



(a) Forum site, 4 different logs.



(b) Academic site, 4 different logs.

Figure 9: Precision among rules that DustBuster attempted to validate vs. number of validations used (N).

percentage of valid rules among those that DustBuster attempts to validate is much smaller than the percentage of valid rules in the original list.

Our aggressive elimination of redundant rules reduces the number of rules we need to validate. For example, on one of the logs in the forum site, the validation phase was initiated with 28 pairs representing 56 likely rules (in both directions). Of these, only 19 were checked, and the rest were removed because they or their counterparts in the opposite direction were deemed valid either directly or since they refined valid rules. We conclude that the number of actual pages that need to be fetched in order to validate the rules is very small.

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 using one of the logs in the forum site is depicted in Figure 10 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.

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 10: The valid rules detected in the forum site.

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

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

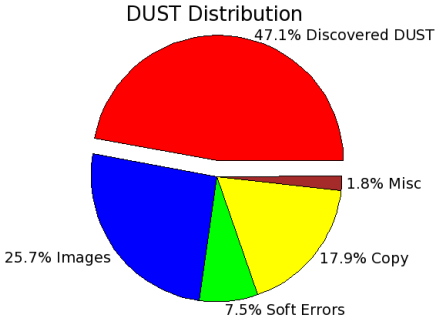


Figure 11: DUST classification, academic site.

we measured the reduction that can be achieved in the next crawl. By setting a slightly more relaxed refutation threshold ($\varepsilon = 10\%$), we obtained a reduction of 26% in the small news site and 6% in the large one. 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%. In all these experiments, the training and testing was done on logs of similar size.

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 server logs and for technical assistance. We thank Israel Cidon, Yoram Moses, and Avigdor Gal for their insightful input. We thank all our reviewers, both from the WWW 2007 conference and the TWEB journal, for their detailed and constructive suggestions.

References

[1] T. Kelly and J. C. Mogul, "Aliasing on the world wide web: prevalence and performance implications," in *the Proceedings of the 11th International World Wide Web Conference*

(WWW), pp. 281–292, 2002.

[2] F. Douglis, A. Feldman, B. Krishnamurthy, and J. Mogul, "Rate of change and other metrics: a live study of the world wide web," in *the Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems (USITS)*, 1997.

[3] F. McCown and M. L. Nelson, "Evaluation of crawling policies for a web-repository crawler," in *the Proceedings of the 17th ACM Conference on Hypertext and Hypermedia (HYPERTEXT)*, pp. 157–168, 2006.

[4] S. J. Kim, H. S. Jeong, and S. H. Lee, "Reliable evaluations of URL normalization," in *the Proceedings of the 4th International Conference on Computational Science and Its Applications (ICCSA)*, pp. 609–617, 2006.

[5] Google Inc., "Google sitemaps." `http://sitemaps.google.com`.

[6] A. Z. Broder, S. C. Glassman, and M. S. Manasse, "Syntactic clustering of the web," in *the Proceedings of the 6th International World Wide Web Conference (WWW)*, pp. 1157–1166, 1997.

[7] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform resource identifiers (URI): Generic syntax." `http://www.ietf.org/rfc/rfc2396.txt`.

[8] "Apache http server version 2.2 configuration files." `http://httpd.apache.org/docs/2.2/configuring.html`.

[9] S. Brin, J. Davis, and H. Garcia-Molina, "Copy Detection Mechanisms for Digital Documents," in *the Proceedings of the 14th Special Interest Group on Management of Data (SIGMOD)*, pp. 398–409, 1995.

[10] H. Garcia-Molina, L. Gravano, and N. Shivakumar, "dscam: Finding document copies across multiple databases," in *the Proceedings of the 4th International Conference on Parallel and Distributed Information Systems (PDIS)*, pp. 68–79, 1996.

[11] N. Shivakumar and H. Garcia-Molina, "Finding Near-Replicas of Documents and Servers on the Web," in *the Proceedings of the 1st International Workshop on the Web and Databases (WebDB)*, pp. 204–212, 1998.

[12] E. Di Iorio, M. Diligenti, M. Gori, M. Maggini, and A. Pucci, "Detecting Near-replicas on the Web by Content and Hyperlink Analysis," in *the Proceedings of the 11th International World Wide Web Conference (WWW)*, 2003.

[13] N. Jain, M. Dahlin, and R. Tewari, "Using bloom filters to refine web search results," in *the Proceedings of the 7th International Workshop on the Web and Databases (WebDB)*, pp. 25–30, 2005.

[14] T. C. Hoad and J. Zobel, "Methods for identifying versioned and plagiarized documents," *Journal of the American Society for Information Science and Technology*, vol. 54, no. 3, pp. 203–215, 2003.

[15] K. Monostori, R. A. Finkel, A. B. Zaslavsky, G. Hodász, and M. Pataki, "Comparison of overlap detection techniques," in *the Proceedings of the 10th International Conference on Complex Systems (ICCS)*, pp. 51–60, 2002.

[16] R. A. Finkel, A. B. Zaslavsky, K. Monostori, and H. W. Schmidt, "Signature extraction for overlap detection in documents," in *the Proceedings of the 25th Australasian Computer Science Conference (ACSC)*, pp. 59–64, 2002.

[17] J. Zobel and A. Moffat, "Exploring the similarity space," *SIGIR Forum*, vol. 32, no. 1, pp. 18–34, 1998.

[18] J. Cho, N. Shivakumar, and H. Garcia-Molina, "Finding replicated web collections," in *the Proceedings of the 19th Special Interest Group on Management of Data (SIGMOD)*, pp. 355–366, 2000.

[19] K. Bharat and A. Z. Broder, "Mirror, Mirror on the Web: A Study of Host Pairs with Replicated Content," *Computer Networks*, vol. 31, no. 11–16, pp. 1579–1590, 1999.

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

[21] K. Bharat, A. Z. Broder, J. Dean, and M. R. Henzinger, "A comparison of techniques to find mirrored hosts on the WWW," *Journal of the American Society for Information Science*, vol. 51, no. 12, pp. 1114–1122, 2000.

[22] WebLog Expert. `http://www.weblogexpert.com/`.

[23] StatCounter. `http://www.statcounter.com/`.

[24] Analog. `http://www.analog.cx/`.

[25] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *the Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pp. 487–499, 1994.

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

[27] P. Jaccard, "Jaccard, P. 1908. Nouvelles recherches sur la distribution florale," vol. 44, pp. 223–270, 1908.

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

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