# Learning URL Patterns for Webpage De-duplication

Hema Swetha Koppula
Yahoo! Labs
Bangalore, India
shema@yahoo-inc.com

Krishna P. Leela
Yahoo! Labs
Bangalore, India
krishna@yahoo-inc.com

Amit Agarwal *
Picsquare.com
Bangalore, India
amit@picsquare.com

Krishna Prasad Chitrapura
Yahoo! Labs
Bangalore, India
pkrishna@yahoo-inc.com

Sachin Garg
Yahoo! Labs
Bangalore, India
gsachin@yahoo-inc.com

Amit Sasturkar
Yahoo! Inc.
Sunnyvale, CA
asasturk@yahoo-inc.com

## ABSTRACT

Presence of duplicate documents in the World Wide Web adversely affects crawling, indexing and relevance, which are the core building blocks of web search. In this paper, we present a set of techniques to mine rules from URLs and utilize these rules for de-duplication using just URL strings without fetching the content explicitly. Our technique is composed of mining the crawl logs and utilizing clusters of similar pages to extract transformation rules, which are used to normalize URLs belonging to each cluster. Preserving each mined rule for de-duplication is not efficient due to the large number of such rules. We present a machine learning technique to generalize the set of rules, which reduces the resource footprint to be usable at web-scale. The rule extraction techniques are robust against web-site specific URL conventions. We compare the precision and scalability of our approach with recent efforts in using URLs for de-duplication. Experimental results demonstrate that our approach achieves 2 times more reduction in duplicates with only half the rules compared to the most recent previous approach. Scalability of the framework is demonstrated by performing a large scale evaluation on a set of 3 Billion URLs, implemented using the MapReduce framework.

## Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Search process; I.7.0 [**Document and Text Processing**]: General

## General Terms

Algorithms, Performance

## Keywords

Search engines, Webpage de-duplication, Site-specific delimiters, Page importance, Generalization, Decision trees, MapReduce

*The research described herein was conducted when the author was at Yahoo! Labs.


*WSDM'10,* February 4–6, 2010, New York City, New York, USA.
ACM 978-1-60558-889-6/10/02.

## 1. INTRODUCTION

Our focus in this paper is on efficient and large-scale de-duplication of documents on the WWW. Web pages which have the same content but are referenced by different URLs, are known to cause a host of problems. Crawler resources are wasted in fetching duplicate pages, indexing requires larger storage and relevance of results are diluted for a query.

Duplicate URLs are present due to many reasons such as:

- Making URLs search engine friendly, e.g., `http://en.wikipedia.org/wiki/Casino_Royale` and `http://en.wikipedia.org/?title=Casino_Royale`.
- Session-id or cookie information present in URLs, e.g., `sid` in `http://cs.stanford.edu/degrees/mscs/faq/index.php?sid=67873&cat=8` and `http://cs.stanford.edu/degrees/mscs/faq/index.php?sid=78813&cat=8`.
- Irrelevant or superfluous components in URLs, e.g., `http://www.amazon.com/Lord-Rings/dp/B000634DCW` and `http://www.amazon.com/dp/B000634DCW`.
- Removing/adding index files such as index.html and default.html by web servers.
- Webmasters at times, construct URL representations with custom delimiters, e.g., `http://catalog.ebay.com/The-Grudge_UPC_043396062603_W0QQ_fclsZ1QQ_pcatidZ1QQ_pidZ43973351QQ_tabZ2` and `http://catalog.ebay.com/The-Grudge_UPC_043396062603_W0?_fcls=1&_pcatid=1&_pid=43973351&_tab=2`

An estimate by [11] shows that approximately 29 percent of pages in the WWW are duplicates and the magnitude is increasing. Clearly, this prompts for an efficient solution that can perform de-duplication without fetching the content of the page. As duplicate URLs have specific patterns which can be utilized for de-duplication, in this paper we focus on the problem of de-duplication of web-pages using just URLs without fetching the content.

### 1.1 Related Work

Conventional methods to identify duplicate documents involved fingerprinting each document's content and group documents by defining a similarity on the fingerprints. Many

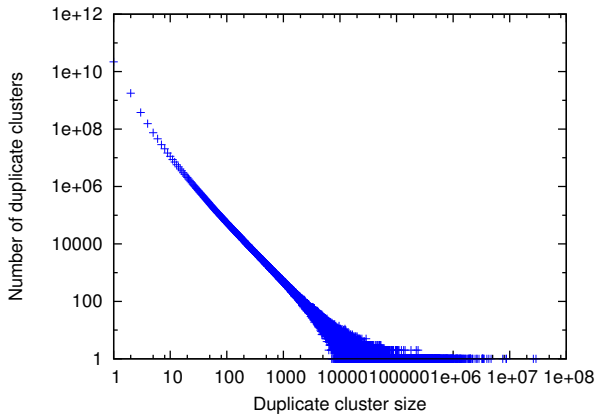


**Figure 1: Duplicate cluster distribution**

elegant and effective techniques using fingerprint based similarity for de-duplication have been devised [7, 8, 13, 17]. [13, 17] also emphasized the need for scale and presented results of experiments on large data sets. However, with the effectiveness comes the cost of fingerprinting and clustering of documents. Recently, more cost-effective approach of using just the URLs information for de-duplication has been proposed, first by Bar-Yossef et.al. [4] and extended by Dasgupta et.al. [9]. Since our technique is an extension of the work in [9], which, in turn extended the work in [4], we describe these two techniques in a little more detail.

Bar-Yossef et al. [4] call the problem, "DUST: Different URLs with Similar Text" and propose a technique to uncover URLs pointing to similar pages. The DUST algorithm focuses on discovering substring substitution rules, which are used to transform URLs of similar content to one canonical URL. DUST rules are learnt from URLs obtained from previous crawl logs or web server logs and each generated rule is annotated with a confidence measure. Heuristics are used to find likely string substitution rules from a given URL list. An example valid DUST rule generated from `http://www.wsdm-conference.org/2010/index.html` and `http://www.wsdm-conference.org/2010/` is "$/index.html\$$" $\rightarrow$ "$". String substitution rules thus obtained can potentially overlap, so they propose ways to eliminate redundancies and preserve either specific or generic rules. Example of overlapping rules are "$.com/story?id=$" $\rightarrow$ "$.com/story\_$" and "$story?id=$" $\rightarrow$ "$story\_$", where every instance of the former is an instance of the latter. Validation is performed on the final set of rules by fetching a small sample of web pages and comparing the page fingerprints. The final, validated rule set $R$ is used to canonicalize a given URL $u$ by repeatedly applying all the rules in $R$ to $u$ until $u$ remains unchanged or a threshold on the number of rules is reached.

Dasgupta et. al. [9] extended this formulation by considering a broader set of rule types which subsume the DUST rules. Different rule types which they consider are: DUST rules, session-id rules, irrelevant path components and complicate rewrites. Their algorithm learns rules from a cluster of URLs with similar page content (such a cluster is referred to as a *duplicate cluster* or a *dup cluster*). They generate rules for every URL pair in a duplicate cluster for all dup clusters. A Rule is generated from a source, target URL pair and is composed of context and transformation. "Context" represents the source URL and "transformation" is the steps required to transform the source URL to the target URL. Consider the following source and target URLs: `http://www.xyz.com/show.php?sid=71829` and `http://www.xyz.com/show.php?sid=17628`. Here, the Rule context $c$ is $c(k_1) = http$, $c(k_2) = www.xyz.com$, $c(k_3) = show.php$ and $c(k_{sid}) = 71829$. The Rule transformation $a$ is $a(k_i) = k_i$ for $i \in 1, 2, 3$ and $a(k_{sid}) = 17628$. The Rules obtained from all URL pairs are "generalized" to a smaller set using heuristics. These generalized Rules are applied to the URLs to canonicalize them.

On a data set size of 7.8 million URLs, they achieved 21% reduction ratio with precision greater than 95%. While considering all generated Rules on the same data set, DUST Rules achieved reduction of 22% and all the Rule types achieved reduction of 60%. This indicates that their Rule types are able to capture most of the learning. Although the Rules are effective, the overall complexity of Rule generation is $O(l_{max}^2 \sum_i C_i^2)$ where $l_{max}$ is the maximum number of key value pairs in a URL and $C_i$ is the size of the dup cluster $i$. As the distribution of dup cluster size over number of clusters is a power law distribution [11], it is not practically feasible to run this algorithm at Web scale. Dup clusters in our data set also follow a power-law distribution, as seen in Figure 1, where the total number of URL pairs runs into trillions.

In our previous work [3], we presented techniques to scale pair-wise Rule generation and introduced a decision tree based Rule generalization algorithm. In this paper, we build on that work by extending the URL and Rule representations and introduce algorithm for finding host specific delimiters. Together these set of techniques form a robust method for de-dulication of web pages using URL strings. While the URL and Rule representation in [9] is complete and covers most patterns, we consider two additional patterns due to their significance on the Web: Deep Token components in a URL and URL component alignment. These are described in later sections of the paper.

## 1.2 Contributions

Our contributions in this paper are four fold:

1. We extend the representation of URL and Rule presented in [9]. The extensions result in better utilization of the information encoded in the URLs to generate precise Rules with higher coverage.

2. We propose a technique for extracting host specific delimiters and tokens from URLs. We extend the pairwise Rule generation to perform source and target URL selection. We also introduce a machine learning based generalization technique for better precision of Rules. Collectively, these techniques form a robust solution to the de-duplication problem.

3. Since scale is a necessary dimension on the Web, we present MapReduce [10] adaptation of the proposed techniques.

4. We demonstrate via experimental comparison that the proposed techniques produce 2 times more reduction in duplicates with half the number of Rules compared to [9]. Finally, via large scale experimental evaluation on a 3-billion URL corpus, we show that the techniques are robust and scalable.

The rest of the paper is organized as follows. In Section 2 we formalize the problem and describe the representations of a URL and a Rule. Section 3 presents the algorithm with details of each technique. MapReduce adaptation of the techniques is presented in Section 4. We present detailed experimental evaluation in Section 5 and conclude in Section 6.

# 2. PROBLEM DEFINITION

In this section, we present the problem definition and the extensions to the URL and the Rule representations from [9].

Given a set of duplicate clusters and their corresponding URLs, can we learn Rules from URL strings which can identify duplicates? Can we utilize these learnt Rules for normalizing unseen duplicate URLs into a unique normalized URL? Our goal in this paper is to learn pair-wise Rules from pairs of URLs in a dup cluster. These pair-wise Rules are later generalized not only to reduce the number of Rules but also efficiently normalize unseen URLs. Pair-wise Rule generation and Rule Generalization techniques are described in Section 3.2. Applications such as crawlers can apply these generalized Rules on a given URL to generate a normalized URL. As these online applications have resource constraints, techniques which achieve larger reduction in duplicates with less false positives and smaller set of Rules are obviously better.

A URL is tokenized using standard delimiters and the primary components of the URL, namely protocol, hostname, path components and query-args [6] are extracted. A URL $u$ can be represented as a function from $K \rightarrow V$ where $K$ is composed of keys and $V$ is composed of values from both static path components and query-args. While query-arg keys inherit the key name from the query name, the path component keys $k_i$ are indexed with an integer $i$, where $i$ is the position index from the start of the URL with protocol corresponding to $i$ equals 1. For example, `http://en.wikipedia.org/wiki?title=Generalized_linear_model` is represented as $\{k_1 = http,\ k_2 = en.wikipedia.org,\ k_3 = wiki$ and $k_{title} = Generalized\_linear\_model\}$.

While the above URL representation captures most of the information encoded in a URL, we uncover some of the reasons for extending this representation. One such reason is the presence of host-specific delimiters in a URL, e.g., delimiters "[" and "]" in `http://www.simtel.net/accelerate.php[id]100019[SiteID]softwareoasis`. Substrings extracted from URL tokens through host-specific delimiters are defined as *DeepTokens*. Detection of host-specific delimiters is described in Section 3.1.2. We modify the key representation $k_i$ in URL to handle deep tokens by providing an additional index $j$ to the keys, where $j$ signifies the relative position of the deep token within the key $k_i$; this is represented as $k_{i.j}$. The above *simtel.net* URL with deep tokens is represented as $\{k_1 = http,\ k_2 = www.simtel.net,\ k_{3.1} = accelerate.php,\ k_{3.2} = [,\ k_{3.3} = id,\ k_{3.4} =],\ k_{3.5} = 100019,\ k_{3.6} = [,\ k_{3.7} = SiteID,\ k_{3.8} =],\ k_{3.9} = softwareoasis\}$. Here $k_{3.8}$ refers to the eighth deep token of the third static path component.

Irrelevant path components are easy to identify given a pair of URLs $u$ and $v$ from a dup cluster. However, it is not easy to generalize these irrelevant path component Rules across dup clusters. For example, consider the following two dup clusters: $Cluster_1$: $\{u_1$: `http://ag.arizona.edu/srnr/about/art/index.html`, $u_2$: `http://ag.arizona.edu/srnr/about/art`$\}$ and $Cluster_2$: $\{u_3$: `http://ag.arizona.edu/art/index.html`, $u_4$: `http://ag.arizona.edu/art`$\}$. Best generalized Rule for these dup clusters has context $c(k_1) = http$, $c(k_2) = ag.arizona.edu$, $c(k_{final}) = index.html$ and transformation $a(k_{final}) = \perp$ where $\perp$ denotes deletion of a key. Here due to misalignment of *index.html* across the two dup clusters, all the four specific Rules are retained by the rewrite approach [9]. As each dup cluster has different number of irrelevant path components, it is difficult to obtain the best generalization. To overcome this problem, we introduce a modified representation of URL which includes both the position index from the start and end of the URL.

Most of the web servers do not differentiate between different value conversions such as uppercase and lowercase. Similar to irrelevant path components, retaining specific value conversions in the Rules does not yield in good generalization. We extend our Rule representation to handle this scenario by allowing value conversion functions to be part of transformation. Complete definitions of URL and Rule are supplied in Definition 1 and 2 respectively.

*DEFINITION 1.* (URL) A *URL* $u$ is defined as function $u : K \rightarrow V \cup \{\perp\}$ where $K$ represents the set of all keys from the URL set and $V$ represents the set of all values. $K$ is represented as $\{k_{(x.i,y.j)}\}$ where $x$, $y$ represent the position index from the start and end of the URL respectively and $i$, $j$ represent the deep token index. A key not present in the URL is denoted by $\perp$.

*DEFINITION 2.* (RULE) A Rule $r$ is defined as a function $r : C \rightarrow T$ where $C$ represents the context and $T$ represents the transformation of the URL. Context $C$ is a function $C : K \rightarrow V \cup \{*\}$ and transformation $T$ is a function $T : K \rightarrow V \cup \{\perp, K^{'}\}$ where $K^{'} = K \cup ValueConversions$ and $ValueConversions = \{Lowercase(K), Uppercase(K), Encode(K), Decode(K), ...\}$

## 2.1 Supporting Examples

In this section, we present some example dup clusters along with the pair-wise Rules generated from the URLs of these clusters. We also present the generalized Rules obtained by performing generalization on the pair-wise Rules.

**Alignment and Irrelevant Path Components**

Consider the previous example of *ag.arizona.edu* with two dup clusters, URLs from $Cluster_1$ can be represented as $u_1 = \{k_{(1,6)} = http,\ k_{(2,5)} = ag.arizona.edu,\ k_{(3,4)} = srnr,\ k_{(4,3)} = about,\ k_{(5,2)} = art,\ k_{(6,1)} = index.html\}$ and $u_2 = \{k_{(1,5)} = http,\ k_{(2,4)} = ag.arizona.edu,\ k_{(3,3)} = srnr,\ k_{(5,2)} = about,\ k_{(5,1)} = art\}$. Rule generated from URLs of $Cluster_1$ with $u_1$ as source URL and $u_2$ as target URL, has context $c(k_{(1,6)}) = http$, $c(k_{(2,5)}) = ag.arizona.edu$, $c(k_{(3,4)}) = srnr$, $c(k_{(4,3)}) = about$, $c(k_{(5,2)}) = art$, $c(k_{(6,1)}) = index.html$ and transformation $t(k_{(6,1)}) = \perp$. Rule generated from URLs of $Cluster_2$ with $u_3$ as source URL and $u_4$ as target URL, has context $c(k_{(1,4)}) = http$, $c(k_{(2,3)}) = ag.arizona.edu$, $c(k_{(3,2)}) = art$, $c(k_{(4,1)}) = index.html$ and transformation $t(k_{(4,1)}) = \perp$.

For the above dup clusters, the generalized Rule has context $c(k_{(1,-)}) = http$, $c(k_{(2,-)}) = ag.arizona.edu$, $c(k_{(-,1)}) = index.html$ and transformation $t(k_{(-,1)}) = \perp$. However, previous approaches will retain the two pair-wise Rules as this kind of generalization is not possible.

**Deep Token Components**

Consider the following two URLs from the same dup cluster: {$u_1$: `http://360.yahoo.com/friends-lttU7d6kIuGq`, $u_2$: `http://360.yahoo.com/friends-nMfcaJRPUSMQ`}. The deep tokenized URLs are as following: $u_1 = \{k_{(1,3)} = http, k_{(2,2)} = 360.yahoo.com, k_{(3.1,1.3)} = friends, k_{(3.2,1.2)} = -, k_{(3.3,1.1)} = lttU7d6kIuGq\}$ and $u_2 = \{k_{(1,3)} = http, k_{(2,2)} = 360.yahoo.com, k_{(3.1,1.3)} = friends, k_{(3.2,1.2)} = -, k_{(3.3,1.1)} = nMfcaJRPUSMQ\}$.

Pair-wise Rule generated with $u_1$ as source URL and $u_2$ as target URL, has context $c(k_{(1,3)}) = http$, $c(k_{(2,2)}) = 360.yahoo.com$, $c(k_{(3.1,1.3)}) = friends$, $c(k_{(3.2,1.2)}) = -$, $c(k_{(3.3,1.1)}) = nMfcaJRPUSMQ$ and transformation $t(k_{(3.3,1.1)}) = lttU7d6kIuGq$. Generalization performed on similar pair-wise Rules results in context $c(k_{(1,3)}) = http$, $c(k_{(2,2)}) = 360.yahoo.com$, $c(k_{(3,1)}) = friends - *$ and transformation $t(k_{(3,1)}) = friends - *$. If the URLs are not deep tokenized, the generalized Rule will have context $c(k_{(1,3)}) = http$, $c(k_{(2,2)}) = 360.yahoo.com$, $c(k_{(3,1)}) = *$ which can match URLs of different pattern resulting in high false positive rate.

**Value Conversions**

Consider the following URLs from the same dup cluster: {$u_1$: `http://allrecipes.com/Recipe/Smores/default.aspx`, $u_2$: `http://allrecipes.com/RECIPE/Smores/default.aspx`}. Rule generated from these URLs with $u_1$ as source URL and $u_2$ as target URL, has context $c(k_{(1,5)}) = http$, $c(k_{(2,4)}) = allrecipes.com$, $c(k_{(3,3)}) = Recipe$, $c(k_{(4,2)}) = Smores$, $c(k_{(5,1)}) = default.aspx$ and transformation $t(k_{(3,3)}) = RECIPE$. Due to value conversion functions, the Rule transformation will be $t(k_{(3,3)}) = Uppercase(k_{(3,3)})$.

# 3. ALGORITHM

In Section 3.1, we discuss two techniques for extracting tokens from URLs: basic tokenization and host specific (deep) tokenization. Section 3.2 covers two Rule generation algorithms: pair-wise Rule generation, which generates Rules specific to a URL pair and Rule generalization, which generalizes both the context and the transformation of pair-wise Rules.

## 3.1 URL Preprocessing

Tokenization is performed on URLs to generate a set of $< key, value >$ pairs as represented in Definition 1. In this section we present generic tokenization which is valid for all URLs across the Web and host-level tokenization which is valid for URLs of a particular host.

### 3.1.1 *Basic Tokenization*

Basic tokenization involves parsing the URL according to RFC 1738 [6] and extracting the tokens. We extract the protocol, hostname, path components and query-args from the URL using the standard delimiters specified in the RFC.

### 3.1.2 *Deep Tokenization*

Deep Tokenization involves learning host-specific delimiters given the set of URLs from a host. In contrast with efforts in literature for extracting semantic features from URLs [16, 5], we discuss a technique which extracts syntactic features from URLs. We employ an unsupervised technique to learn custom URL encodings used by webmasters. Our approach is influenced by the sequence based techniques in computational biology [12]. Definitions 3, 4 and 5 describe the core elements of the algorithm.


**Figure 2: Deep Tokenization Example**

Starting with the generic pattern $*$, the algorithm recursively detects more specific patterns matching the given set of tokens. This results in generation of a pattern tree as depicted in figure 2. For example, pattern *-Austria_*.html matches all tokens in figure 2. The algorithm refines this pattern into two specific patterns: ctattractions-*-Austria_*.html and profile-*-Austria_*.html each of which matches a subset of tokens. Patterns are refined recursively until more possible delimiters are detected. Once a leaf pattern occurs in the tree, it is used to extract deep tokens from corresponding tokens. In figure 2, the pattern ctattractions-*-Austria_*_attractions.html tokenizes the token ctattractions-17876002-Austria_Vienna_attractions.html into the following deep tokens: *ctattractions*, −, 17876002, −, *Austria*, _, *Vienna*, _, *attractions*, ., *html*. Thus deep tokenization of a token results in a sequence of deep tokens.

*DEFINITION 3.* (CANDIDATE ANCHOR) Candidate anchor is a token substring bounded by a delimiter where delimiter is any non-alphanumeric character or a unit change. Unit change indicates transition between lowercase alphabets, uppercase alphabets and numbers. Candidate anchor can be represented as a regex $([a-z]^i|[A-Z]^j|[0-9]^k)$ where $i$, $j$ and $k$ are integers greater than 0.

*DEFINITION 4.* (PATTERN) Pattern is defined as a regex $(a_c|d|*)^n$ representing a set of tokens, where $a_c$ is a candidate anchor, d is a delimiter, $*$ matches any string and n is an integer greater than 0.

*DEFINITION 5.* (DEEP TOKEN) Deep token is the token substring which matches a regex group of a pattern.

The *DeepTokenize* algorithm takes a pattern and a set of tokens as input and generates the set of deep token sequences. It is a recursive algorithm with the input pattern set to $*$ in the first invocation. *FindCandidateAnchors* in line 1 generates a sequence of candidate anchors for each input token. Line 2 calls *SelectAnchors* which selects and returns a set of anchors from the set of candidate anchors, as illustrated in algorithm 2. If no anchors are returned, the current pattern cannot be further refined and hence forms the leaf pattern in the current path of the tree. *FindDeepTokens* in line 4 uses the current pattern to generate sequence of deep tokens from the corresponding input

**Algorithm 1** DeepTokenize

**Input:** Pattern and a set of tokens: $p$, $\{tok\}$
**Output:** Set of deep token sequences:$\{\langle tok_d \rangle\}$
1: $\{\langle a_c \rangle\} \Leftarrow$ FindCandidateAnchors( $\{tok\}$, $p$ )
2: $\{a\} \Leftarrow$ SelectAnchors( $\{\langle a_c \rangle\}$ )
3: **if** $\{a\}$ is empty **then**
4: $\{\langle tok_d \rangle\} \Leftarrow$ FindDeepTokens( $p$, $\{tok\}$ )
5: **else**
6: $\{(p_{child}, \{tok\})\} \Leftarrow$ GenerateChildPatterns( $p$, $\{a\}$, $\{tok\}$ )
7: **for all** $(p_{child}, \{tok\})$ **do**
8: DeepTokenize( $p_{child}$, $\{tok\}$ )
9: **end for**
10: **end if**

**Algorithm 2** SelectAnchors

**Input:** Set of candidate anchor sequences: $\{\langle a_c \rangle\}$
**Output:** Set of anchors: $\{a\}$
1: $\{(cluster, \{a_c\})\} \Leftarrow$ ClusterCandidateAnchors( $\{a_c\}$ )
2: **for all** $(cluster, \{a_c\})$ **do**
3: ComputeFeatures( $cluster$, $\{a_c\}$ )
4: **end for**
5: $\{cluster_{sel}\} \Leftarrow$ SelectClusters( $\{cluster\}$ )
6: **if** $\{cluster_{sel}\}$ is not empty **then**
7: **for all** $a_c \in cluster$ where $cluster \in cluster_{sel}$ **do**
8: $\{a\} = \{a\} \cup a_c$
9: **end for**
10: **end if**

tokens. If some anchors are returned by *SelectAnchors*, *GenerateChildPatterns* in line 6 uses the selected anchors to create child patterns from the current pattern. It returns a set of refined patterns, each with the set of tokens matching the refined pattern. Lines 7-9 recursively call *DeepTokenize* for each child pattern and their corresponding tokens.

Algorithm 2 illustrates how anchors are selected from candidate anchors. The goal here is to select anchors that are homogeneous, span most of the tokens and create least number of child patterns. *SelectAnchor* takes a set of candidate anchor sequences as input and generates a set of anchors. Line 1 clusters the candidate anchors. We use the following information for clustering candidate anchors: (1) Start Delimiter - delimiter at the start of candidate anchor; (2) End Delimiter - delimiter at the end of candidate anchor; (3) Candidate Anchor Type - numeric or alphabetic. Lines 2-4 compute the following features for each cluster: (1) Token Coverage - number of tokens the cluster covers; (2) Unique Candidate Anchor Count - number of unique candidate anchors in the cluster and (3) Candidate Anchor Frequency Variance - variance in the frequency of candidate anchors in the cluster. Line 5 selects clusters with high token coverage, low unique candidate anchor count and low candidate anchor frequency variance. For the example in figure 2, the set of clusters formed in the first iteration are: $\{Austria\}$, $\{html\}$, $\{ctattractions, profile\}$, $\{Vienna, Salzburg, Graz\}$, $\{attractions, Belvedere, Dommuseum, Eisriesenwelt\}$, and $\{17876002, 17826402, 17682302, 78587305, \cdot\}$. The clusters $\{Austria\}$ and $\{html\}$ are selected and the candidate anchors *Austria*, *html* are returned as the selected anchors. These are then used to refine the pattern $*$ into *-Austria_*.html. All tokens in the example satisfy this pattern. In the next iteration, the cluster $\{ctattractions, profile\}$ is selected, and the pattern *-Austria_*.html is refined into two patterns: ctattractions-*-Austria_*.html and profile-*-Austria_*.html, each applicable to a subset of tokens. These patterns are further refined using the corresponding token subsets as shown in the figure.

LEMMA 1. *Algorithm DeepTokenize can be implemented to run in time $O(\sum_i n_i * d_{max})$ for all hosts, where $n_i$ is the number of URLs in host $i$ and $d_{max}$ is the maximum depth of a pattern tree.*

PROOF. *SelectAnchors* requires $O(\sum_i \sum_j \sum_k c_{ijk})$ iterations where $c_{ijk}$ is the number of candidate anchors for value $k$ of key $j$ of host $i$. For any host $i$, $\sum_j \sum_k c_{ijk}$ is bounded by $n_i * l_{max} * k_{max}$ where $l_{max}$ is the maximum number of candidate anchors in a token and $k_{max}$ is the maximum number of keys. $l_{max}$ is bounded by the maximum length of a token, thus $l_{max} * k_{max}$ is bounded by the length of the URL which can be considered as constant. Thus the total complexity of *SelectAnchors* is $O(\sum_i n_i)$. Similarly, number of iterations required to create child patterns is $O(\sum_i n_i)$. Thus total complexity of *DeepTokenize* is $O(\sum_i n_i * d_{max})$ where $O(\sum_i n_i)$ is the complexity at each level of pattern tree. □

## 3.2 Rule Generation

### 3.2.1 *Pair-wise Rule Generation*

Given a set of dup clusters and corresponding URLs, *GenerateAllRules* algorithm selects pairs of duplicate URLs and generates pair-wise Rule. *GeneratePairwiseRule* consumes a URL pair $u, v$ and generates a pair-wise Rule which converts the source URL $u$ to the target URL $v$. At web scale, as the number of URLs and dup clusters are large in number, generating Rules for all URL pairs in a dup cluster for all dup clusters is not feasible. The number of URL pairs to be considered for generating all pair-wise Rules is $\sum_i C_i^2$ where $C_i$ is the size of dup cluster $i$. This number can run in trillions for large sized dup clusters, which are not uncommon due to instances of session-ids and irrelevant components in the URLs.

We propose optimizations to reduce the number of source and target URLs considered for pair-wise Rule generation. In the ideal scenario, we would have one canonicalized URL representing every dup cluster. However, this will not hold if we consider all target URLs for pair-wise Rule generation. Also, dup clusters exhibit the characteristic of having few URLs which are close to the normalized URL and it is prudent to use these URLs as the target URLs. Consider the following dup cluster: $u_1$: `http://www.youtube.com/watch?v=S9_4KMVtgw0&feature=channel`, $u_2$: `http://www.youtube.com/watch?v=S9_4KMVtgw0` and $u_3$:`http://www.youtube.com/watch?v=S9_4KMVtgw0&feature=channel&ytsession=WDxRmlqSR8olo8oVN`. For this cluster $u_2$ represents the ideal target URL.

**Target Selection:** As target URLs are used for generating transformations, selecting better targets yield better transformations and compact Rules. Typically, dup clusters have a small set of URLs which are very close to the normalized URL and we capture these URLs by ranking and selecting top-k. This not only achieves significant reduction in the number of Rules but also makes the Rules coherent

**Algorithm 3** GeneratePairwiseRule

**Input:** Pair of URLs: $u$,$v$
**Output:** Rule: context $c$ and transformation $t$

1: Define context $c$ as $\forall k \in K : c(k) = u(k)$
2: Define transformation $t$ as follows
3: **for all** $k \in K$ **do**
4: **if** $\exists k^{'} \in K : u(k^{'}) \neq \perp \wedge v(k) = u(k^{'})$ **then**
5: $t(k) = k^{'}$ (Key Reference)
6: **else if** $\exists k^{'} \in K : u(k^{'}) \neq \perp \wedge v(k) = ValueConversion(u(k^{'}))$ **then**
7: $t(k) = ValueConversion(k^{'})$ (Value Conversion)
8: **else if** $u(k) \neq v(k)$ **then**
9: $t(k) = v(k)$ (Value Literal or Key Add/Delete)
10: **end if**
11: **end for**
12: return $r = (c, t)$.

**Algorithm 4** GenerateAllRules

**Input:** Set of URLs $U$
**Output:** Set of Pair-wise Rules $R$

1: Initialize $R = \emptyset$
2: **for all** Duplicate URLs $D \in U$ **do**
3: $rankURLsForTargetSelection(D)$
4: $T = getTargetURLs(D)$
5: $S = getSourceURLs(D)$
6: **for all** $(u, v) : u \in S, v \in T$ **do**
7: $r = GeneratePairwiseRule(u, v)$
8: $R = R \cup \{r\}$
9: **end for**
10: **end for**
11: return $R$

for generalization to perform better. Some of the characteristics of an ideal normalized URL include static type of the URL, shorter length of URL, minimum hop distance from the domain root and high number of in-links. As these characteristics closely match those in host-level page rank discussed in [15], we considered this approach for ranking target URLs. Since we use previous crawl logs for Rule generation, all of the above features are available for ranking already.

**Source Selection:** As source URLs are used for generating context, we retain URLs with high importance. Consider the case of crawler which normalizes URLs on-the-fly. As some URLs are seen more often than others due to the power-law distribution [14], generating Rules for high traffic URLs will result in high reduction in duplicates. URLs need to be ranked for source selection; however we cant reuse the rank computed for target selection as the characteristics for source selection is different from target selection. Ranking of source URLs can be based on PageRank or the On-Line Page Importance [18][2]. For our experiments, we used the page importance metric [2], which is computed at crawl time and is available in logs. We sampled ranked URLs from each duplicate cluster using stratified sampling. URLs are divided into equal sized buckets based on the page importance. URLs are sampled from each bucket proportional to the contribution of the bucket to the total importance of the dup cluster. In the absence of ranking information, sampling can be done based on other information from URLs such as number of distinct tokens in the URL. This criterion ensures that Rules are learnt for various patterns of URLs of a dup cluster.

Algorithm 4 presents *GenerateAllRules* which takes a set of URLs as input and generates a set of Rules as output. For each dup cluster, Line 3 of *GenerateAllRules* ranks URLs for target selection. From ranked URLs, line 4 selects top-k URLs as the target set $T$. Line 5 samples the remaining URLs $U$-$T$ to select source URL set $S$. Lines 6-9 generate pair-wise Rules from the URLs of $S$ and $T$ using the *GeneratePairwiseRule* algorithm. Line 1 of *GeneratePairwiseRule* algorithm sets the context of the Rule as the source URL. Lines 3-11 construct the transformation, which captures the difference in the source and target URLs for all keys in $K$. Lines 4-5 check if a value of key $k$ in source URL occurs with a different key $k'$ in the target, the target key reference is added to the transformation. Lines 6-7 check if the value is present in the target URL after applying the value conversion function on source key. If it is, the value conversion function of the target key is added to the transformation. If the above two conditions fail, lines 8-9 assign the value of target key $k$ to the transformation. This handles key additions, key deletions and value replacements.

LEMMA 2. *Algorithm* GenerateAllRules *can be implemented to run in time* $O(\sum_i C_i * log(C_i))$*, where* $C_i$ *is the size of dup cluster* $i$*.*

PROOF. Target selection and source selection requires sorting within a duplicate cluster, which can be performed in $O(\sum_i C_i * log(C_i))$. *GeneratePairwiseRule* is linear in terms of the number of tokens in source and target URLs. Complexity to generate Rules from $s_i$ source URLs and $t$ target URLs is $O(\sum_i s_i * t)$. Here $s_i$ is bounded by $C_i$ and $t$ is a constant. Thus the total complexity of *GenerateAllRules* is $O(\sum_i C_i * log(C_i))$ where complexity of generating Rules in a duplicate cluster is bounded by $O(\sum_i C_i)$. □

### 3.2.2 *Rule Generalization*

Although the number of pair-wise Rules generated through pair-wise Rule generation is linear in the number of URLs, these Rules can be further reduced by generalization. Generalization is required for the following reasons: (1) Pair-wise Rules if kept intact will not be applicable on unseen URLs unless the Rules are generalized to accommodate new values and (2) Applications such as crawlers which use Rules in an online mode require small footprint for storing the Rules.

Rule generalization captures generic patterns out of the pair-wise Rules and generalizes both the contexts and transformations. As target URLs are restricted by target selection, many sources map to the same target URLs making it feasible for context generalization. Previous efforts used heuristics to perform generalization, however these heuristics do not guarantee precision of the Rules. We use a Decision Tree [19] for context generalization. Advantages of Decision Tree over heuristics are the proven error bounds and the robustness of the technique as it has been used in multiple domains.

Context generalization involves constructing the decision tree with transformations as Classes/Targets. The key set $K$ is considered as attributes and the value set $V \cup \{*\}$ is considered as instances for the attributes. We construct a bottom-up decision tree where an attribute (or key $k$) is

**Algorithm 5** GeneralizePairwiseRules

**Input:** Pair-wise Rules: $R = \{< c, t >\}$
**Output:** Generalized Rules: $R_{gen} = \{< c_{gen}, t_{gen} >\}$
1: $Class \Leftarrow \{t\}$; $keySet \Leftarrow K$; $Nodes \Leftarrow Class$
2: **while** $keySet$ is not empty **do**
3: $\forall key \in keySet\ InfoGain(key) \Leftarrow Entropy(R) - \sum_{v \in \{c(key)\}} \frac{\sharp c(key)=v}{\sharp c(key)} Entropy(R \mid c(key) = v)$
4: $key_{sel} \Leftarrow$ select key with $max\ InfoGain$
5: new node set $P = \emptyset$
6: **for all** nodes $n \in Nodes$ **do**
7: $V = \{c_i(key_{sel})\}$; where $< c_i, t_j > \in \{< c, t >\} \wedge t_j \in \{Class(n)\}$
8: **if** $|V| > threshold_{|V|}$ **then**
9: $P = P \cup (< key_{sel}, * >)$
10: **else**
11: $P = P \cup (< key_{sel}, v >)$
12: **end if**
13: **end for**
14: merge nodes in $P$ with the same value $v$
15: $Nodes \Leftarrow P$
16: remove $key_{sel}$ from $keySet$
17: **end while**
18: $\{< c_{gen}, t >\} \Leftarrow$ all paths in the $DTree$
19: $\{< c_{gen}, t_{gen} >\} \Leftarrow$ GeneralizeTransformations( $\{< c_{gen}, t >\}$ )

selected at every iteration. Nodes at the current iteration are assigned a particular value $v$ of the selected attribute $k$. We assign $*$ (wildcard) to a node if there is no single value $v$ which holds majority. After constructing the decision tree, we traverse it top-down to generate the generalized context and corresponding transformation.

Algorithm 5 *GeneralizePairwiseRules* takes a set of pair-wise Rules and generates generalized Rules. Line 1 sets the transformations as the classes for the decision tree. Conditional entropy of each key is computed in lines 3-4 and the key with minimum conditional entropy is selected. Lines 6-13 assign values to nodes which is decided by the number of values the key takes in the selected node. Line 14 merges the set of nodes with the same value. These new set of nodes are added to the decision tree in line 15 and the selected key is removed from the set of keys in line 16. This process is repeated until all the keys are exhausted for tree construction.

We can see an example of generalization using one type of duplicates from www.imdb.com: $Cluster1$: { `http://www.imdb.com/title/tt0810900/photogallery`, `http://www.imdb.com/title/tt0810900/mediaindex` }, $Cluster2$: { `http://www.imdb.com/title/tt0053198/photogallery`, `http://www.imdb.com/title/tt0053198/mediaindex` }. The pair-wise Rule from $Cluster1$ $R_1$ is $c(k_{(1,5)}) = http$, $c(k_{(2,4)}) = www.imdb.com$, $c(k_{(3,3)}) = title$, $c(k_{(4.1,2.2)}) = tt$, $c(k_{(4.2,2.1)}) = 0810900$, $c(k_{(5,1)}) = photogallery$, $t(k_{(5,1)}) = mediaindex$. The pair-wise Rule from $Cluster2$ $R_2$ is $c(k_{(1,5)}) = http$, $c(k_{(2,4)}) = www.imdb.com$, $c(k_{(3,3)}) = title$, $c(k_{(4.1,2.2)}) = tt$, $c(k_{(4.2,2.1)}) = 0053198$, $c(k_{(5,1)}) = photogallery$, $t(k_{(5,1)}) = mediaindex$. The pair-wise Rules generated from similar dup clusters have the same transformation but different values of $k_{(4.2,2.1)}$ in the context. During the tree construction, the transformation $t(k_{(5,1)}) = mediaindex$ is taken as the Class, and at each iteration of algorithm 5, the key with maximum InfoGain is selected. In this example, $k_{(4.2,2.1)}$ will be selected first due to its higher InfoGain compared to other keys. Since no single value of this key takes majority, the key is assigned wildcard ($*$). The rest of the keys take only one value for the above dup clusters. This gives rise to the generalized Rule: $c(k_{(1,5)}) = http$, $c(k_{(2,4)}) = www.imdb.com$, $c(k_{(3,3)}) = title$, $c(k_{(4.1,2.2)}) = tt$, $c(k_{(4.2,2.1)}) = *$, $c(k_{(5,1)}) = photogallery$, $t(k_{(5,1)}) = mediaindex$.

While context generalization is done through decision tree, we also perform transformation generalization by considering all transformations corresponding to the same context. If a key is generalized to take wildcard ($*$) in the context and the same key takes multiple values in the transformation, then the value is replaced by wildcard in the transformation. The motivation for doing this generalization is, if the generalized key in the context can take any value ($*$), the key is irrelevant for any matched URL, and hence the transformation can also have a wildcard for that key.

LEMMA 3. *Algorithm GeneralizePairwiseRules can be implemented to run in time $O(\sum_i k_i^2 * r_i)$, where $k_i$ is the number of keys and $r_i$ is the number of Rules in host $i$.*

PROOF. Let $v_i$ be the max number of values of a key in host $i$. Entropy calculation at each iteration requires $O(k_i * v_i)$. Total time required for entropy calculation for all iterations is $O(\sum_i k_i^2 * v_i)$ as $k_i$ is the maximum number of iterations possible. Value assignment to nodes for all iterations is $O(\sum_i k_i * r_i)$. Since $v_i$ is bounded by $r_i$, total complexity of *GeneralizePairwiseRules* is $O(\sum_i k_i^2 * r_i)$. □

# 4. SCALABILITY

So far, we have described the algorithms for Deep Tokenization, Pair-wise Rule generation and Rule generalization, which form our set of proposed techniques for deduplication using URLs. In this section, we adapt these algorithms to the MapReduce paradigm. We present Map and Reduce functions for different stages of Rule generation techniques.

**Deep Tokenization**
**Stage I**: Generates deep tokenized key value pairs. $< key_i, val_{ij} >$ represents key value pair in a URL and $< key_{d_i}, val_{d_{ij}} >$ represents deep tokenized key value pair. $URL_{id}$ is hash of URL string. Reduce step calls algorithm 1.

$$\begin{aligned} Map: &\quad Host, URL \rightarrow Host, key_i, < URL_{id}, val_{ij} > \\ Red: &\quad Host, key_i, \{< URL_{id}, val_{ij} >\} \rightarrow \\ &\quad \{< Host, URL_{id}, key_{d_i}, val_{d_{ij}} >\} \end{aligned}$$

**Stage II**: Associates deep tokenized key, value pairs to the original URL and constructs deep tokenized URLs ($URL_{dt}$).

$$\begin{aligned} Map: &\quad Host, URL_{id}, key_{d_i}, val_{d_{ij}} \rightarrow \\ &\quad Host, URL_{id}, < key_{d_i}, val_{d_{ij}} > \\ Red: &\quad Host, URL_{id}, \{< key_{d_i}, val_{d_{ij}} >\} \rightarrow \\ &\quad Host, URL_{dt} \end{aligned}$$

**Pair-wise Rule Generation**
Generates pair-wise Rules from URL pairs of a duplicate cluster. $dupC$ stands for dup cluster id and $source_{rank}$ and $target_{rank}$ stand for source and target selection rank. $c$ and

$t$ stand for context and transformation of the pair-wise Rule.

$$Map: \quad Host, dupC, URL_{dt}, source_{rank}, target_{rank} \rightarrow Host, dupC, < URL_{dt}, source_{rank}, target_{rank} >$$
$$Red: \quad Host, dupC, \{< URL_{dt}, source_{rank}, target_{rank} >\} \rightarrow \{< Host, dupC, c, t >\}$$

**Rule Generalization**

**Stage I**: Generates frequency for each $< c_{key_i}, c_{val_{ij}}, t >$ where $c_{key_i}, c_{val_{ij}}$ represents key value pair of a Rule context.

$$Map: \quad c, t \rightarrow Host, t, \{< c_{key_i}, c_{val_{ij}} >\}$$
$$Red: \quad Host, t, \{< c_{key_i}, c_{val_{ij}} >\} \rightarrow \{< Host, t, c_{key_i}, c_{val_{ij}}, freq >\}$$

**Stage II**: Generalizes contexts using algorithm 5. $c_{gen}$ stands for generalized context.

$$Map: \quad Host, t, < c_{key_i}, c_{val_{ij}}, freq > \rightarrow Host, < t, c_{key_i}, c_{val_{ij}}, freq >$$
$$Red: \quad Host, \{< t, c_{key_i}, c_{val_{ij}}, freq >\} \rightarrow \{< Host, c_{gen}, t >\}$$

**Stage III**: Generalizes transformations corresponding to $c_{gen}$. $t_{gen}$ stands for generalized transformation.

$$Map: \quad Host, c_{gen}, t \rightarrow Host, c_{gen}, < t >$$
$$Red: \quad Host, c_{gen}, \{< t >\} \rightarrow \{< Host, c_{gen}, t_{gen} >\}$$

# 5. EXPERIMENTAL EVALUATION

In this section, we present the experimental setup and the key metrics used for measuring the performance of our techniques. We also use these metrics to compare our work with one of the previous approaches. We demonstrate the practical feasibility of our techniques for web-scale by evaluating them on a large data set.

**Metrics:** We define and use the following metrics for our experiments:

1. *Coverage* of a Rule is the number of URLs the Rule applies to, denoted by $r_{cov}$
2. *Precision* is a Rule level metric. If $r_{cov}$ is the coverage of Rule $r$ and $f$ is the number of URL pairs $(u, v) \ni r(u) = v$ and $u$ and $v$ are not in the same dup cluster, precision of $r$ is $\frac{r_{cov} - f}{r_{cov}} * 100$.
3. *ReductionRatio* is a metric for set of Rules. It is the percentage reduction in the number of URLs after transforming the URLs with a set of Rules. It is defined as $\frac{|U_{orig}| - |U_{norm}|}{|U_{orig}|}$ where $U_{orig}$ is the original URL set and $U_{norm}$ is the normalized URL set.
4. *AvgReductionPerRule* is a metric for set of Rules. This demonstrates the average reduction per Rule and is defined as $\frac{|U_{orig}| * ReductionRatio}{|R|}$ where $R$ is the set of Rules.

**Data Sets** We considered a small data set and a large data set, both consisting of dup clusters having size of at least 2. Small data set is the data set presented in [9]. Large data set consisting of billions of records is obtained from crawl logs of a commercial search engine. Data set characteristics such as number of URLs, hosts and dup clusters are presented in Table 1.

| Data set | # URLs | # Dup Clusters | # Hosts |
|---|---|---|---|
| Small | 7.87M | 1.83M | 356 |
| Large | 2.97B | 604M | 1M |

**Table 1: Data set characteristics**

| Data set | Pair-wise Rule gen | Rule generalization | Min coverage filtering |
|---|---|---|---|
| Small | 12.37M | 93.23K | 38.83K |
| Large | 728.26M | 13.54M | 8.29M |

**Table 2: Number of Rules after each Rule gen stage**

While our false positive rate is computed on dup clusters having size of at least 2, we observed that it is not straightforward to include single dup cluster URLs (not duplicates). As our approach consider URLs from crawl logs, it is not a practical assumption to consider existence of all possible duplicates. This means that it is not feasible to calculate false positive rate for single dup cluster URLs.

As shown in the table, small data set is composed of 7.87 Million URLs and large data set is composed of 2.97 Billion URLs. For both the data sets, we performed 50-50 test-train split by randomly assigning each dup cluster to either training set or test set. Section 5.1 and section 5.2 show the experimental results on small data set and large data set respectively.

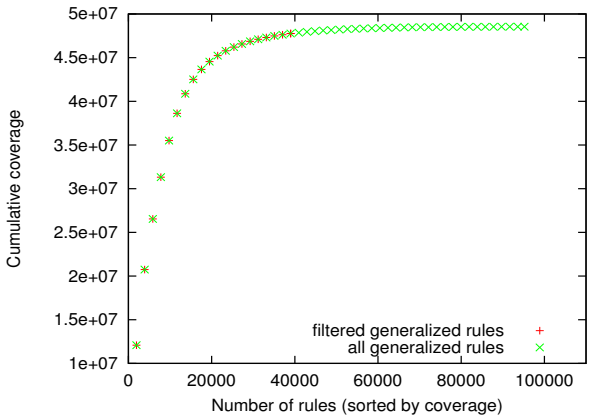


**Figure 3: Rule coverage distribution**

## 5.1 Small data set

In this section we present the results of our approach on small data set and compare these metrics with previous approach [9]. As our techniques extend the rewrite technique, we compare our work with the rewrite technique.

During our evaluation, we observed that there are large number of generalized Rules which have much less coverage. Figure 3 shows number of Rules vs. cumulative coverage for all generalized Rules and filtered set of generalized Rules (filtered on coverage >= 10). As seen from the figure, the cumulative coverage of Rules with coverage < 10 is not significant. Cumulative coverage in the graph is more than

| Precision threshold | Rewrite approach | | | Our approach | | |
|---|---|---|---|---|---|---|
| | Num Rules | % of Rules | Reduction Ratio(%) | Num Rules | % of Rules | Reduction Ratio(%) |
| precision = 1 | 1149 | 3.67 | 3 | 649 | 1.67 | 6 |
| >= .95 | 21894 | 69.81 | 16 | 1707 | 4.40 | 26 |
| >= .9 | 21938 | 69.95 | 17 | 2375 | 6.12 | 33 |
| >= .80 | 22105 | 70.48 | 18 | 3547 | 9.13 | 42 |
| All | 31363 | 100 | 43 | 38830 | 100 | 69 |

**Table 3: Metrics comparison with URL rewrite approach for small data set**

the number of URLs as more than one Rule applies to a URL. Table 2 gives the number of Rules after each step of Rule generation. Rule generalization which includes both context and transformation generalization reduces the number of Rules by 99.25% and filtering based on min coverage further reduces these Rules by 58.35%.

In Table 3 we list the reduction ratio and the number of Rules for different levels of precision. For comparison, we have considered a fan-out threshold of 30 for the previous approach as this value of fan-out is effective in learning Rules that have higher *AvgReductionPerRule* [9]. It can be seen that our approach achieves higher reduction ratios with lesser number of Rules. At 100% precision, our techniques generate 649 Rules which achieve a reduction of 6% while the previous approach generates 1149 Rules which achieve a reduction of 3%: previous approach requires double the number of Rules to achieve half the reduction as ours. Higher reduction ratio with lesser number of Rules holds for all precision levels above 80%.

| Precision threshold | Min Coverage | Num Rules | Avg Reduction per Rule | Reduction Ratio (%) |
|---|---|---|---|---|
| =1 | 10 | 649 | 748.17 | 6.2 |
| | 100 | 296 | 1429.94 | 5.37 |
| | 1000 | 126 | 2614.76 | 4.18 |
| >=0.95 | 10 | 1707 | 1218.78 | 26.42 |
| | 100 | 1105 | 1816.75 | 25.5 |
| | 1000 | 523 | 3292.46 | 21.88 |
| >= 0.9 | 10 | 2375 | 1093.51 | 32.98 |
| | 100 | 1527 | 1643.91 | 31.89 |
| | 1000 | 662 | 3208.83 | 26.99 |
| >= 0.8 | 10 | 3547 | 922.08 | 41.53 |
| | 100 | 2214 | 1431.22 | 40.26 |
| | 1000 | 894 | 2938.29 | 33.38 |

**Table 4: *AvgReductionPerRule* for different precision levels and coverage thresholds for small data set**

Table 4 gives the *AvgReductionPerRule* for different min coverage thresholds. As shown, the average reduction per Rule increases as we increase the minimum coverage threshold for all precision levels. Average reduction per Rule for our techniques is much higher than the previous approach: avg reduction per Rule for 100% and 95% precision levels are 748.17 and 1218.78 from our techniques compared to 205.6 and 57.5 from previous approach. For high precision levels, as we increase the min coverage threshold, the number of Rules reduce drastically with out much drop in the reduction ratio. For precision level of 1, by increasing the min coverage threshold from 10 to 1000, the number of Rules reduce by 5 fold with only 2% decrease in reduction ratio. For precision levels higher than 0.9, by increasing min coverage threshold from 10 to 1000, we achieve 3 to 5 fold reduction in the number of Rules with marginal decrease in the reduction ratio: 2% to 6%. Based on precision requirements and resource constraints, search engines can easily tune min coverage threshold with out much loss in achievable reduction.

## 5.2 Large data set

In this section, we present results of our approach on large data set and demonstrate the following: (1) Our Rule generation techniques are efficient at web-scale; (2) Metrics achieved on large data set are similar to those of small data set where we achieve high reduction ratios for all precision levels and (3) Rules generated from our techniques are of high impact and applications using these Rules will be scalable due to the small footprint of Rules. As it is not practically feasible to run [9] on billions of URLs, we don't have comparison numbers for large data set.

We ran Rule generation on a custom Hadoop [1] cluster of 100 nodes. It took 5 hr 24 min to generate the Rules. Increasing the number of nodes to 200 linearly decreased the computation time to 2 hr 40 min.

| Precision threshold | Num Rules | % of Rules | Reduction Ratio (%) |
|---|---|---|---|
| precision=1 | 439919 | 5.30 | 7.63 |
| >= .95 | 810611 | 9.77 | 29.02 |
| >= .9 | 1065641 | 12.85 | 33.59 |
| >= .8 | 1421145 | 17.14 | 38.82 |
| All | 8293763 | 100.00 | 60.36 |

**Table 5: *Precision* and *ReductionRatio* tradeoff for large data set**

Table 5 gives the precision reduction ratio tradeoff for different precision levels. For precision of 1, we achieve reduction ratio of 7.63 with only 5.3% of the total Rules. For precision levels higher than 0.9, we achieve a reduction ratio of 33.59 with only 12.85% of Rules which implies that we generate not only high precision Rules but also high impact Rules.

Table 6 gives the average reduction per Rule for different precision levels and minimum coverage thresholds. Similar to results from small data set, we see an increase in *AvgReductionPerRule* with increase in min coverage threshold with significant reduction in the number of Rules. For precision levels higher than 0.9, by increasing min coverage threshold from 10 to 1000, we achieve 10 to 27 fold reduction in the number of Rules with marginal decrease in reduction ratio: 4% to 9%.

| Precision threshold | Min Coverage | Num Rules | Avg Reduction per Rule | Reduction Ratio (%) |
|---|---|---|---|---|
| =1 | 10 | 439919 | 515.11 | 7.63 |
| | 100 | 123417 | 1549.96 | 6.44 |
| | 1000 | 16065 | 7621.71 | 4.12 |
| >=0.95 | 10 | 810611 | 1062.91 | 29.02 |
| | 100 | 381894 | 2133.12 | 27.42 |
| | 1000 | 82406 | 7688.66 | 21.33 |
| >= 0.9 | 10 | 1065641 | 935.96 | 33.59 |
| | 100 | 499926 | 1882.48 | 31.68 |
| | 1000 | 104977 | 6858.4 | 24.24 |
| >= 0.8 | 10 | 1421145 | 810.88 | 38.82 |
| | 100 | 669454 | 1621.74 | 36.55 |
| | 1000 | 135210 | 6034.98 | 27.47 |

**Table 6: *AvgReductionPerRule* for different precision levels and coverage thresholds for large data set**

One interesting observation from tables 4, 6 is that we achieve better performance at precision 0.95 not only in terms of the reduction ratios but also in terms of increase in average reduction per Rule. Reduction ratios increased by 20.22 and 21.39 for 0.95 compared to precision level 1. For various high precision levels, as the number of Rules we generate is very less for billions of URLs, our techniques are practical for Web scale systems. Also our system is easily tunable by precision and min coverage threshold depending on resource constraints.

# 6. CONCLUSIONS

In this paper, we presented a set of scalable and robust techniques for de-duplication of URLs. Our techniques scale to web due to feasible computational complexity and easy adaptability to the MapReduce paradigm. We presented basic and deep tokenization of URLs to extract all possible tokens from URLs which are mined by our Rule generation techniques for generating normalization Rules. We presented a novel Rule generation technique which uses efficient ranking methodologies for reducing the number of pair-wise Rules. Pair-wise Rules thus generated are consumed by the decision tree algorithm to generate highly precise generalized Rules.

We evaluated the effectiveness of our techniques on two data sets. We compared our results with previous approaches and showed that our approach significantly outperforms them on many key metrics. We also evaluated our techniques on multi billion URL corpus and showed that we achieve not only high reduction ratios but also high average reduction per Rule. Our system is very practical as it is configurable for different precision levels and coverage thresholds to achieve high reduction ratios.

Source selection approach which we have presented prioritizes head and torso traffic and we would like to explore the feasibility of Rule generation for the rest of tail traffic. Dup clusters which are the ground truth for generating Rules includes false positives due to the approximate similarity measures. We would like to explore ways of handling these in a robust fashion. Generalization is performed separately for source and target and we would like to explore the feasibility of generalizing both in an iterative fashion.

# 7. REFERENCES


[1] Hadoop: Open source implementation of mapreduce. `http://lucene.apache.org/hadoop/`.

[2] S. Abiteboul, M. Preda, and G. Cobena. Adaptive on-line page importance computation. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 280–290, May 2003.

[3] A. Agarwal, H. S. Koppula, K. P. Leela, K. P. Chitrapura, S. Garg, P. K. GM, C. Haty, A. Roy, and A. Sasturkar. Url normalization for de-duplication of web pages. In *CIKM '09: Proceedings of the 18th ACM conference on Information and knowledge management*, pages 1987–1990, November 2009.

[4] Z. Bar-Yossef, I. Keidar, and U. Schonfeld. Do not crawl in the dust: different urls with similar text. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 111–120, May 2007.

[5] E. Baykan, M. Henzinger, L. Marian, and I. Weber. Purely url-based topic classification. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 1109–1110, April 2009.

[6] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (url), 1994.

[7] A. Broder. On the resemblance and containment of documents. In *SEQUENCES '97: Proceedings of the Compression and Complexity of Sequences 1997*, page 21, June 1997.

[8] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, May 2002.

[9] A. Dasgupta, R. Kumar, and A. Sasturkar. De-duping urls via rewrite rules. In *KDD '08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 186–194, August 2008.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, December 2004.

[11] D. Fetterly, M. Manasse, and M. Najork. On the evolution of clusters of near-duplicate web pages. In *LA-WEB '03: Proc. of the First Conference on Latin American Web Congress*, page 37, November 2003.

[12] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, 1997.

[13] M. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 284–291, August 2006.

[14] M. Henzinger and S. Lawrence. Extracting knowledge from the world wide web. *Proceedings of the National Academy of Sciences*, 101:5186–5191, April 2004.

[15] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub. Exploiting the block structure of the web for computing pagerank. Technical report, Stanford University, 2003.

[16] M.-Y. Kan and H. O. N. Thi. Fast webpage classification using url features. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 325–326, November 2005.

[17] G. S. Manku, A. Jain, and A. D. Sarma. Detecting near-duplicates for web crawling. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 141–150, May 2007.

[18] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, November 1999.

[19] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.