ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/221023992

A pattern tree-based approach to learning URL normalization rules

Conference Paper · January 2010

DOI: 10.1145/1772690.1772753 · Source: DBLP

CITATIONS
28

READS
460

6 authors, including:

Rui Cai
Microsoft
62 PUBLICATIONS **2,123** CITATIONS
SEE PROFILE

Xiaodong Fan
Microsoft
11 PUBLICATIONS **189** CITATIONS
SEE PROFILE

Lei Zhang
University of Pittsburgh
588 PUBLICATIONS **48,046** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project Image enhancement and restoration View project

Project Neural stem cells View project

All content following this page was uploaded by Rui Cai on 09 February 2015.

The user has requested enhancement of the downloaded file.

A Pattern Tree-based Approach to Learning URL Normalization Rules

Tao Lei[†][*], Rui Cai[‡], Jiang-Ming Yang[‡], Yan Ke[§], Xiaodong Fan[§], and Lei Zhang[‡]
[†]Dept. Computer Science, Peking Univ.
[‡]Microsoft Research Asia; [§]Microsoft Corporation
[†]asukalei@gmail.com, [‡§]{ruicai, jmyang, yanke, xiafan, leizhang}@microsoft.com

ABSTRACT

Duplicate URLs have brought serious troubles to the whole pipeline of a search engine, from crawling, indexing, to result serving. URL normalization is to transform duplicate URLs to a canonical form using a set of rewrite rules. Nowadays URL normalization has attracted significant attention as it is lightweight and can be flexibly integrated into both the online (e.g. crawling) and the offline (e.g. index compression) parts of a search engine. To deal with a large scale of websites, automatic approaches are highly desired to learn rewrite rules for various kinds of duplicate URLs. In this paper, we rethink the problem of URL normalization from a *global* perspective and propose a pattern tree-based approach, which is remarkably different from existing approaches. Most current approaches learn rewrite rules by iteratively inducing *local* duplicate pairs to more general forms, and inevitably suffer from noisy training data and are practically inefficient. Given a training set of URLs partitioned into duplicate clusters for a targeted website, we develop a simple yet efficient algorithm to automatically construct a URL pattern tree. With the pattern tree, the statistical information from all the training samples is leveraged to make the learning process more robust and reliable. The learning process is also accelerated as rules are directly summarized based on pattern tree nodes. In addition, from an engineering perspective, the pattern tree helps select deployable rules by removing conflicts and redundancies. An evaluation on more than 70 million duplicate URLs from 200 websites showed that the proposed approach achieves very promising performance, in terms of both de-duping effectiveness and computational efficiency.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*; I.5.2 [**Pattern Recognition**]: Design Methodology—*Pattern analysis*

General Terms

Algorithms, Performance, Experimentation.

Keywords

URL normalization, URL pattern, URL deduplication.

1. INTRODUCTION

Syntactically different URLs linking to identical or closely similar text content, called duplicate URLs or DUST [7], are quite common on the Web. The intention of creating duplicate URLs is to make a website more user-friendly and easier to maintain. For example, some websites have mirrors used to balance load or served as backups, e.g., http://us.php.net and http://de.php.net. To provide a personalized service, most web server software allows webmasters to insert additional parameters like *session_id* into URLs to identify a user connection, although such parameters have no impact on the content of the retrieved web page. Furthermore, to make the surfing experience smoother, website designers usually add abundant aliasing URLs as shortcuts to help users quickly navigate to target pages. At last, some industry conventions, e.g., adding trailing slash or removing directory indexes like default.asp, also lead to many duplicates [3]. As a result, duplicate URLs have become quite common and helpful to both webmasters and Internet users.

However, to search engines, duplicate URLs are kind of a nightmare. Actually, duplicate URLs have brought a series of troubles to the whole pipeline of a search engine, from the backend to the frontend. First, in crawling, precious bandwidth and time are wasted to download large quantities of duplicate pages. Then, in indexing, many machines and hard disks are required to store such redundant information; and consequently the constructed inverted index becomes bloated and inefficient. After that, in the link-based ranking algorithms like PageRank [8], pages cannot obtain appropriate scores as the Web's link graph has been disturbed by duplicate URLs. At last, in presenting, users cannot be satisfied when they see duplicate entities in the search results. According to our statistics on an untreated corpus of 20 billion documents, around one-quarter are duplicates. Consequently, most commercial search engines like Bing, Google, and Yahoo have paid much attention to remove duplicates. It is worth noting that Google even asked webmasters to explicitly mark duplicate links in their pages [1].

Besides leveraging the supports from webmasters, intensive research efforts have been devoted in recent years to pursue automatic de-duping solutions, from both academia [7, 10,12,18] and industry [4,13–16]. According to the methodologies, most of the current de-duping technologies can be classified into two categories: ***content-based de-duping*** and ***URL-based de-duping***.

Content-based approaches once dominated the de-duping research [10,12,14,15,18]. Just as the name implies, content-based methods compare the semantic content of two web

[*]This work was performed at Microsoft Research, Asia.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

pages and consider two pages having very similar content as duplicates. Content-based approaches are essentially a combination of information retrieval (IR) and database (DB) technologies. Most research efforts from IR area focus on extracting representative features to characterize a document (web page). The mainstream strategy of designing a feature is to generate one or several bit strings (called fingerprints [17]) based on the terms extracted from a document [14, 15]. The widely adopted features are simhash [11] and shingles [9, 10]. In contrast, researchers with DB background are more interested in developing appropriate similarity measurements and efficient algorithms to minimize the time cost of identifying near duplicates in a high dimensional space, such as the similarity join algorithm in [18] and the locality-sensitive hashing (LSH) technology in [5]. Detailed comparisons of some state-of-the-art content-based de-duping approaches can be found in [14]. Content-based approaches can accurately identify duplicates as they already know the content of web pages. However, from the industrial point of view, content-based de-duping is only an offline post-processing, as it still needs to download duplicates and cannot save the precious bandwidth and storage. A search engine also needs online de-duping in some preprocessing steps (e.g., normalizing discovered URLs before adding them to the crawling queue) where only URL strings are available. To fulfill such a requirement, URL-based de-duping has been attracting significant attention nowadays.

The main idea of URL-based de-duping is to learn some normalization rules (also called DUST rules [7] or rewrite rules [13]) for a website based on a training set. The training set can be automatically constructed by first sampling a few pages from a targeted website and then discovering duplicate groups based on their fingerprints. With the learnt normalization rules, we can tell whether two URLs are duplicates without examining the corresponding page content, and can further rewrite them to the same canonical form [3]. Therefore, URL-based de-duping has low computational cost and is easy to integrate. In this paper, we focus on developing an effective and efficient approach to learning URL normalization rules. Bar-Yossef et al. [7] are pioneers in the field of URL-based de-duping. In their work, URL normalization is considered as a "string substitution" problem. For example, http://us.php.net can be transformed to http://de.php.net using a substitution operation "us→de". The limitation of [7] is that they simply treat a URL as an ordinary string but ignore there is a syntax structure scheme strictly defined for URLs in the W3C standard [2]. Another important milestone work is from Dasgupta et al. [13], in which the most critical contribution is to introduce the syntactic structure information into URL normalization. As shown in [13], with the syntactic structure information, the algorithm can deal with more duplicate cases than string substitutions.

To the best of our knowledge, most URL-based de-duping approaches, including [7] and [13], adopt the same ***bottom-up pairwise strategy*** to learning rules. That is, although their algorithm details are different, the fundamental process is the same – starting from pairs of duplicate URLs and iteratively inducing them to more general forms. However, such a strategy has several limitations in practice:

– It is sensitive to noise and is easy to break before reaching the final stage. As the strategy starts from original URLs, random noise in individual URLs usually mislead the algorithms to generate wrong rules. Such mistakes are possibly accumulated in the bottom-up process and may break the induction somewhere.

– It cannot take full advantage of all the training samples. To avoid noise influence, most algorithms adopt very strict assumptions to filter the training data. As a result, a part of training samples cannot be involved in the algorithms.

– It is computationally inefficient to do pairwise induction. The time complexity is at least $O(cn^2)$ where c is the number of duplicate groups in training set and n is the average number of URLs in each duplicate group. In practice, a duplicate group can have tens of thousands URLs[1]. The cost becomes extremely heavy and unacceptable.

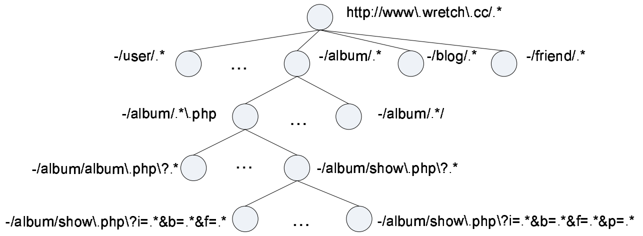


Figure 1: An illustration of the pattern tree (simplified for a clear view) for URLs from www.wretch.cc.

In this paper, we rethink the URL normalization problem from a ***global perspective*** and try to leverage as much information as possible from the whole training set. It should be noticed that webmasters usually have some principles when designing the URL scheme of a website. In their minds, different URL components by design take different functions, e.g., some components denote semantic categories and some others record browsing parameters. Actually, the goal of URL normalization is to find out which components are irrelevant to the page content. If the design principles are known, it becomes relatively easier and more confident to tell which components should be normalized. In comparison, the bottom-up pairwise strategy is from a local perspective as individual duplicate pairs cannot tell design principles.

After investigating a substantial number of duplicate URLs, we found the design principles of a website can be revealed by a pattern tree to some extent. A pattern tree is a group of hierarchically organized URL patterns. Each node (a pattern) on a pattern tree represents a group of URLs sharing the same syntax structure formation; and the pattern of a parent node characterizes all its children. In this way, a pattern tree actually provides a statistic to the syntax scheme of a website. More specifically, in a pattern tree, salient URL tokens which express directories, functions, and document types are usually explicitly represented by some nodes; while trivial tokens which denote parameter values are easy to be generalized to some regular expressions[2]. Figure 1 shows a part of the pattern tree of www.wretch.cc, from which the typical patterns and their relationships are clearly illustrated. From Figure 1, it is noticed the salient tokens like "album" and "show.php" are reserved by some nodes; and trivial tokens such as the values of the parameter "b" are generalized to "b=.*" in the bottom nodes.

Since a normalization rule is to transform a group of URLs having the same formation to another (canonical) formation,

[1] As shown in Table 1 in Section 6.1, some duplicate groups in the experiments have more than 300 thousand URLs.

[2] In this paper, we simply denote all the regular expressions with the wildcard character '*'; while in implementation more complex regular expressions are supported.

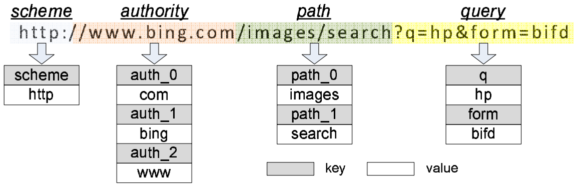


Figure 2: An illustration of the syntax structure of a URL based on which it can be described as a set of *key-value* pairs.

it can be represented by *a mapping between two nodes on the pattern tree* as a tree node (a URL pattern) indeed represents a set of URLs sharing the same formation. Based on this observation, in this paper, we propose a pattern tree-based approach to learning URL normalization rules. First, we present an algorithm to automatically reconstruct a pattern tree given a collection of URLs. Second, tree nodes having similar duplicate distributions are discovered, based on which a set of candidate rules are generated. At last, we develop a graph-based strategy to select deployable rules from the candidate rules. In our approach, as the learning process directly starts from the tree nodes instead of URLs, the computational cost is lower as the number of tree nodes is much smaller than that of original duplicate URLs. Moreover, a pattern tree leverages all the training samples in the statistics, and is also robust to random noise in individual URLs. A large-scale evaluation on more than 70 million duplicate URLs from 200 websites showed our approach can achieve significant improvements in comparison with one state-of-the-art method, for both de-duping effectiveness and computational efficiency.

The rest of this paper is organized as follows. In Section 2 we explain our motivations with a concrete example; and clarify some basic concepts in Section 3. The overview of the proposed approach is introduced in Section 4; and the algorithm details are described in Section 5. Experimental evaluations are reported in Section 6; and in the last section we draw conclusions and point out some future directions.

2. BACKGROUND AND MOTIVATIONS

In this part, we explain the detailed motivations of this work. With a concrete example, we show how the bottom-up pairwise strategy works and when it fails; and explain how a pattern tree can help to improve the performance.

URL Scheme and the Key-Value Representation.

To help readers better understand the following algorithms and our motivations, an example URL and its syntax structure are shown in Figure 2. According to [2], a URL can be first decomposed into some generic URI components (e.g., the *scheme*, *authority*, *path* and *query* in Figure 2) by a set of general delimiters like ':', '/', and '?', where the components before the delimiter '?' are called static part and the rest is called dynamic part. Each component can be further broken down into several sub-sections by another set of delimiters such as '&' and '='. Following the terminology in [13], a URL can be easily represented by a series of key-value pairs based on the decomposition. For example, the "path_0-images" and "q-hp" are two *key-value* pairs in Figure 2. Here, for the static part of a URL, the keys are pre-defined by the corresponding components such as "auth_0" and "path_0"; and for the dynamic part the keys are the tokens before the delimiter '='.

How the Bottom-up Pairwise Strategy Works.

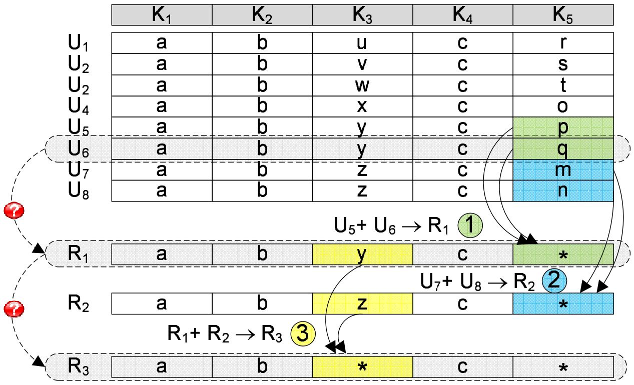
Based on the key-value representation, Dasgupta et al. proposed an effective algorithm in [13] to discover the mapping relationships among different keys. In this way, keys can provide perfect contextual constraints, and can help create more general normalization rules. For example, Figure 3 shows a set of duplicate URLs and their key-value representations, where their values of K_3 and K_5 are different and should be generalized in the normalization. With the DUST algorithm in [7], it can only generate substitution rules such as "p→q" and "m→n", which are too trivial to be deployed. In comparison, Dasgupta's algorithm can successfully generalize these URLs in a hierarchical progress. As shown in Figure 3, in the step (1) and (2), two raw rules R_1 and R_2 are first generated according to the duplicate pairs (U_5, U_6) and (U_7, U_8); R_1 and R_2 are further combined in the step (3) to create the ideal rule R_3, i.e., for URLs satisfying the contextual constraint "value(K_1)=a & value(K_2)=b & value(K_4)=c", their values under K_3 and K_5 should be normalized to the wildcard '*'.

When the Bottom-up Pairwise Strategy Fails.

However, as only a few inputs (URLs or raw rules) are compared in each induction, in [13] there is a strict requirement – *the inputs should differ with each other at only one single key* – to ensure the induction is safe. A similar requirement can also be found in [7]. Such strict requirements usually break the algorithms somewhere. For example, in Figure 3 if one of $U_5 \sim U_8$ (e.g. U_6) is absent, some raw rules like R_1 and the final R_3 cannot be generated. In other words, some "critical" training samples cannot be missing to ensure the performance of these bottom-up pairwise algorithms. However, in real applications, URLs usually have more complex *key-value* structures than the toy case in Figure 3. Consequently, we need more "critical" samples to cover all the possible key-value combinations. It is practically impossible to prepare such a complete training set. On the other hand, these algorithms discard a lot of training samples as they don't satisfy the "only-one-key-different" requirement. In Figure 3, 50% samples ($U_1 \sim U_4$) are not leveraged in the learning process although they indeed reveal the information that the values under K_3 and K_5 are quite diverse and possibly to be generalized for normalization.

How a Pattern Tree Helps.

First, a pattern tree can fully leverage the statistical information of the whole training set. In this way, the learning process becomes more reliable and robust, and no longer suffers from random noise. For the example in Figure 3, even some URLs are absent, it is still possible to construct a tree node with the pattern "K_1=a&K_2=b&K_3=.*&K_4=c&K_5=.*" by considering all the other URLs (including $U_1 \sim U_4$). Second, with a pattern tree, the learning efficiency can be significantly accelerated by summarizing rules directly based on the tree nodes. For example, in Figure 3 we no longer need R_1 and R_2 and can directly generate R_3 based on the above pattern. An additional benefit from the pattern tree is to help detect and resolve the conflicts of rules. From an engineering perspective, the generated rules cannot be directly deployed as some rules may be redundant and some rules may conflict with each other. Selecting deployable rules is an important step as unqualified rules will degrade the performance and even crash the system. Unfortunately, this problem was not addressed in previous work [7, 13, 16]. A



	K_1	K_2	K_3	K_4	K_5
U_1	a	b	u	c	r
U_2	a	b	v	c	s
U_2	a	b	w	c	t
U_4	a	b	x	c	o
U_5	a	b	y	c	p
U_6	a	b	y	c	q
U_7	a	b	z	c	m
U_8	a	b	z	c	n
R_1	a	b	y	c	*
R_2	a	b	z	c	*
R_3	a	b	*	c	*

Figure 3: An illustration of how the traditional bottom-up pairwise strategy works. There are 8 duplicate URLs $U_1 \sim U_8$, from which it is clear the values of K_3 and K_5 should be normalized to '*'. Following the steps (1) to (3), three rules R_1, R_2, and R_3 are sequentially generated where R_3 is the ideal one. However, if some URLs like U_6 are absent, the bottom-up process is easy to break down and cannot produce R_1 and R_3, as denoted by the shadowed rows and corresponding dashed arrows. Moreover, this strategy cannot leverage $U_1 \sim U_4$ in induction.

pattern tree can also help on this problem. For example, the "ancestor–descendant" relationships between tree nodes make it easy to identify redundant rules. More details can be found in Section 5.3.

3. PROBLEM FORMULATION

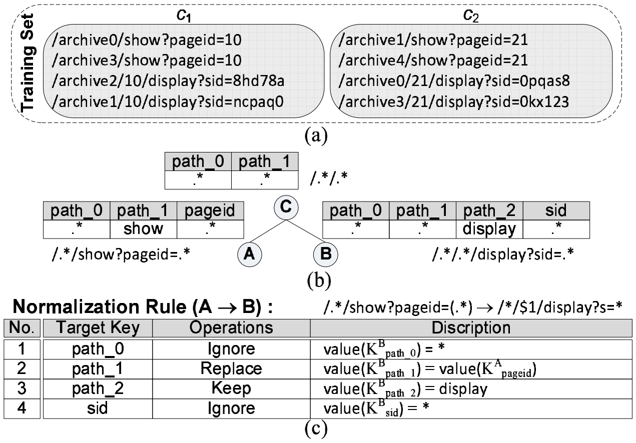
To make a clear presentation and facilitate the following discussions, we first define some concepts used in this paper.

Duplicate Clusters and Training Set. A duplicate cluster is a group of URLs having very similar page content; and a training set is a collection of duplicate clusters. In this paper, we number the duplicate clusters in sequence, denoted as $\{c_1, c_2, \ldots, c_n\}$, to distinguish different duplicates, as shown in Figure 4 (a).

URL Pattern and Pattern Tree. Similar with the formation of a URL, a URL pattern is also described with a group of key-value pairs, and the values can be regular expressions. A pattern tree is a hierarchical structure where the pattern of a parent node can characterize all its children. Figure 4 (b) shows the pattern tree discovered from the training set in Figure 4 (a), where nodes A and B are siblings and node C is their parent. For each node, both the key-value pairs and the regular expression of the corresponding URL pattern are illustrated.

Rewrite Operation and Normalization Rule. Figure 4 (c) shows a normalization rule which coverts URLs of the source pattern A to the format of the target pattern B in Figure 4 (b). A normalization rule is eventually a set of rewrite operations for every URL key. Each rewrite operation works on one target key and sets its value accordingly. There are three kinds of operations defined in our approach:

- **Ignore** is to generalize the values on the target key to the wildcard '*' as its value doesn't affect the page content. Such a key is also called "irrelevant path components" in [13]. Mirror version and session ID are typical examples of this case, as shown with the 1^{st} and 4^{th} operations in Figure 4 (c).



Training Set

c_1
/archive0/show?pageid=10
/archive3/show?pageid=10
/archive2/10/display?sid=8hd78a
/archive1/10/display?sid=ncpaq0

c_2
/archive1/show?pageid=21
/archive4/show?pageid=21
/archive0/21/display?sid=0pqas8
/archive3/21/display?sid=0kx123

(a)

path_0	path_1
.*	.*

/.*/.*

path_0	path_1	pageid
.*	show	.*

/.*/show?pageid=.*

path_0	path_1	path_2	sid
.*	.*	display	.*

/.*/.*/display?sid=.*

(b)

Normalization Rule (A → B) : /.*/show?pageid=(.*) → /*/$1/display?s=*

No.	Target Key	Operations	Discription
1	path_0	Ignore	value($K^B_{path\_0}$) = *
2	path_1	Replace	value($K^B_{path\_1}$) = value(K^A_{pageid})
3	path_2	Keep	value($K^B_{path\_2}$) = display
4	sid	Ignore	value(K^B_{sid}) = *

(c)

Figure 4: An illustration of the definitions of (a) *Duplicate Cluster* and *Training Set*, (b) *URL Pattern* and *Pattern Tree*, and (c) *Rewrite Operation* and *Normalization Rule*.

- **Replace** is to set the value of the target key to the value of the source key. For example, the 2^{nd} operation in Figure 4 (c) sets the value of the key "path_1" of B according to the value of the key "pageid" of A, i.e., value($K^B_{path\_1}$) = value(K^A_{pageid}).
- **Keep** is to retain the original value of the target key. For example, the value "display" of the target key "path_2" of B doesn't change in the 3^{rd} operation.

Compression Rate, Dup-Reduction Rate, and False-Positive Rate are the measurements mostly used to evaluate the de-duping performance [7,13]. The *compression rate* and *dup-reduction rate* are defined as:

$$compression\ rate = 1 - N_{norm}/N_{ori} \qquad (1)$$

$$dup\text{-}reduction\ rate = 1 - \frac{1 - C_{norm}/N_{norm}}{1 - C_{ori}/N_{ori}} \qquad (2)$$

where N_{ori} and N_{norm} are the numbers of URLs before and after the normalization, respectively. Supposing there are C_{ori} duplicate clusters in the N_{ori} URLs, the original *duplicate rate* is $1 - C_{ori}/N_{ori}$. After the normalization, supposing there are C_{norm} clusters, the new *duplicate rate* is $1 - C_{norm}/N_{norm}$.

In normalization, it is still possible that two non-duplicate URLs are mistakenly rewritten to the same canonical format. Such two URLs are called a *false-positive pair*. By contrast, any two URLs that are normalized to the same URL are called a *support pair*. The false-positive rate (*fpr*) is to measure how accurate the normalization (either by applying one rule or a set of rules) is:

$$fpr = N_{fpp}/N_{supp} \qquad (3)$$

where N_{fpp} and N_{supp} are the numbers of *false-positive pairs* and *support pairs* in the normalization, respectively.

4. FRAMEWORK OVERVIEW

The overview of the proposed approach is shown in Figure 5, which consists of (a) pattern tree construction, (b) candidate rule generation, and (c) deployable rule selection.

As we have discussed in Section 2, a pattern tree can fully leverage the statistical information of the whole training set and help summarize duplicate URL patterns. Given a collection of training samples (i.e. URLs), the first component is

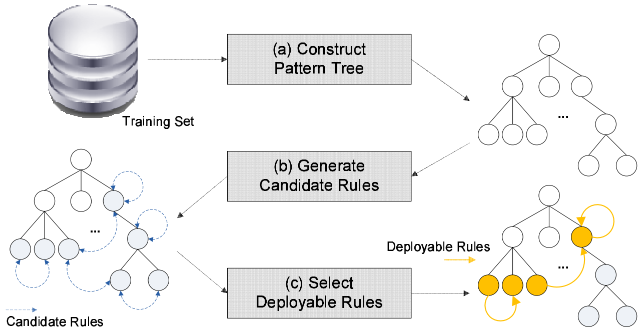

Training Set
(a) Construct Pattern Tree
(b) Generate Candidate Rules
(c) Select Deployable Rules
Deployable Rules
Candidate Rules
...

Figure 5: The flowchart of the proposed approach, which consists of three parts: (a) pattern tree construction, (b) candidate rules generation, and (c) deployable rules selection.

to construct a pattern tree based on the analysis of URL syntax structures. Each URL is first decomposed into a group of *key-value* pairs, as shown in Figure 2. The distribution of *values* under each *key* is then evaluated, based on which the pattern tree is construct through an iterative splitting process. Starting from the root node which contains all the training samples, members of a node is split into sub-groups according to the key whose value distribution has the smallest entropy. In this way, each node on the pattern tree has a set of member URLs; and the members of the parent node are the union of members of all its children nodes.

Based on the constructed pattern tree, we estimate how the duplicate URLs distribute on each tree node. If there is a node whose member URLs are highly duplicated with the members of another node, there should be a normalization rule to transform the URLs from one node to the other. In the second component, we present an inverted-index like structure to identify possible duplicate nodes with linear time complexity. After that, for every two duplicate nodes we generate one candidate rule, in which the rewrite operations are constructed by discovering the mapping relationships among the keys of the two nodes. Finally, a quick proofing is carried out to remove unqualified candidates with high false-positive rates. This is a necessary step as the learning process only considers *positive observations* (where URLs are duplicate) but ignore *negative observations*. As a result, some generated rules are biased by the positive cases and cause false-alarms on the negative ones.

The last component is to select deployable rules from qualified candidates. The goal is to identify an optimal subset of rules to balance the de-duping capability and the running stability, which is theoretically an NP-hard problem [13]. After investigating a plenty number of rules from different websites, we summarize several typical cases which lead to redundant and conflict rules. Based on these observations, we develop a graph-based method to incorporate both the populations of tree nodes and the link relationships created by normalization rules. Experimental results demonstrate that the proposed method is practically effective.

5. ALGORITHM DETAILS

In this section there are three subsections, each of which describes the algorithm details of one step in Figure 5.

5.1 Pattern Tree Construction

Constructing the pattern tree for a set of URLs is not a trivial problem. First, traditional pattern discover algorithms like [6] tend to find one or more common sub-strings from a group of strings, but cannot help build the hierarchical structure (e.g. Figure 1). In addition, the syntactic structure of URLs is not leveraged in these algorithms. Second, hierarchical clustering algorithms might be helpful to construct pattern tree; while it is still hard to find an appropriate similarity measurement to properly considering the syntax structure information. What's worse is calculating the similarity of every URL pair could be a huge task when the training set is large. In short, we need to seek a practical way for building a pattern tree.

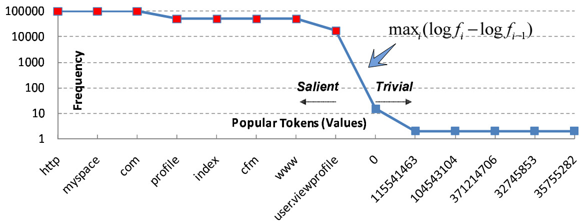


Figure 6: An illustration of the most popular tokens (or values) in 100,000 URLs from myspace.com. From the frequency curve, it is clear that the maximum decline point can help distinguish *salient values* from *trivial* ones.

Algorithm 1 Build-Pattern-Tree. Given a URL group U and the set of keys that have been processed K_{done}, the algorithm returns a tree node t for URLs in U.

```
Create a new node $t$
Generate a regular expression for $t$ to describe URLs in $U$
Calculate entropy $H(k)$ for each key in $U$
if $\exists k \notin K_{done}$ then
  Let $k^* \notin K_{done}$ be the key that minimizes $H(k)$
  $V_{k^*} = \emptyset$
  for all URL $u \in U$ do
    if $u(k^*)$ is a trivial value then
      $V_{k^*} = (V_{k^*} \cup$ "$*$"$)$
    else
      $V_{k^*} = (V_{k^*} \cup u(k^*))$
    end if
  end for
  if all $u(k^*)$ are trivial values then
    return the node $t$
  end if
  Split $U$ into sub-groups $U_1, \ldots, U_t$ according to $V_{k^*}$
  for all subgroup $U_i$ do
    $ch = \mathbf{BuildPatternTree}(U_i, K_{done} \cup \{k^*\})$
    add $ch$ to $t$ as a child node
  end for
end if
return the node $t$
```

In our observation, different URL components (or keys) usually have different functions and play different roles. In general, keys denoting directories, functions, and document types have only a few values, which should be explicitly recorded by the pattern tree. By contrast, keys denoting parameters such as user names and product IDs have quite diverse values, which should be generalized in the pattern tree. According to this observation, we propose a top-down recursive split process. Starting at the root node (which contains all the URLs), the tree and sub-trees are constructed by continually splitting the URLs into subgroups, as shown in Algorithm 1. We follow two criteria in the algorithm: 1) in each iteration, the key with the fewest values is selected, based on which the URLs are split into sub-groups; and 2) stop the iteration and generate a tree node if values of the selected key seldom appear in other URLs.

The first criterion describes the value distribution of a given key k in the form of entropy, as:

$$H(k) = \sum_{i=1}^{V} -\frac{v_i}{N} \log \frac{v_i}{N} \quad (4)$$

where V is the number of values under this key, v_i is the frequency of the i^{th} value, and N is the number of total URLs in this iteration. In each iteration, the key with the smallest entropy is chosen; and the URLs are split into sub-groups according to their values of the selected key. The second criterion is to decide whether the iteration should be stopped. As we prefer general patterns and a clean tree, we just keep *salient values* in the patterns and generalize *trivial values* with regular expressions. The salient/trivial values are determined based on the statistics of their frequencies. Figure 6 shows the most popular values extracted from 100,000 URLs of the website myspace.com. From Figure 6, it is noticed there is a distinct decline of the frequency curve. Values before the decline such as "index" and "cfm" are called *salient values*; while values after the decline are called *trivial values*. In our implementation, values are sorted in descending order according to their frequencies; and the position of the decline is determined by finding the maximal drop rate on the frequency curve, as:

$$pos_{decline} = \max_i(\log f_i - \log f_{i-1}) \quad (5)$$

where f_i is the appearance frequency of the i^{th} value.

5.2 Candidate Rules Generation

As introduced in Section 4, candidate rules are generated based on the duplicate nodes on the pattern tree. For every two duplicate nodes, one raw candidate rule is created by constructing the corresponding rewrite operations. At last, a proofing is conducted to remove unqualified candidates with high false-positive rates.

5.2.1 Identify Possible Duplicate Nodes

As we have explained in Section 4, duplicate nodes on the pattern tree are pairs of two nodes sharing enough common duplicate URLs. To better assess the duplicate ratio of two nodes s and t, we define a quantitative measurement called *overlap*. We say a duplicate cluster c_i in the training set is common to s and t, if $s \cap c_i \neq \emptyset$ and $t \cap c_i \neq \emptyset$, then the *overlap* of s and t is defined based on all their common clusters c_i:

$$overlap(s,t) = \frac{\sum_{c_i} |\{u \mid u \in c_i \ and \ (u \in s \ or \ u \in t)\}|}{|s| + |t|} \quad (6)$$

The *overlap* sufficiently reflects how many common duplicates there are between s and t, thus a larger *overlap* provides us more confidence that there exists a rule between them. In addition, we consider the special case that the two duplicate nodes are the same, i.e., $s = t$. For such a case we use the *duplicate rate* defined in Section 3 instead of the *overlap* (as $overlap(s,s) = 1$) above, to ensure the node s still covers enough duplicates.

Based on the definition of (6), to find out possible duplicate nodes, a straightforward way is to check every pair of nodes on the pattern tree. The time cost is $O(cm^2)$, where m is the number of tree nodes and c is the average number of duplicate clusters in each node. Although the cost is much cheaper than checking original pairs of duplicate URLs, it is still time consuming when the tree has a large number of nodes. In practice, we find most tree node pairs do not have any common duplicates. This motivates us to accelerate the process with an inverted index-like structure. As shown in Algorithm 2, entries of the index structure are duplicate clusters; and the members of each entry are the tree nodes having the corresponding duplicate cluster. With this index structure, for a tree node s, we can identify all the nodes which share at least one common duplicate cluster with s in linear time. In this way, the time cost decreases to $O(cm)$. To reduce the efforts of the later steps, we also utilize a threshold o_{min} in Algorithm 2 to prune those duplicate nodes with too small overlap scores. In practice, o_{min} is empirically set to 0.5.

Algorithm 2 Identify-Duplicate-Nodes. Given the pattern tree T and all duplicate clusters $c_1, c_2, \ldots$, this algorithm returns a set of possible duplicate nodes $\mathfrak{D}$

```
$\mathfrak{D} = \emptyset$
Initialize an empty index list $\mathfrak{L}$
for all duplicate clusters $c_i$ do
    $\mathfrak{L}(c_i) = \{t \mid t \in T, t \cap c_i \neq \emptyset\}$
end for
for all duplicate clusters $c_i$ do
    for all $(s,t) \in \mathfrak{L}(c_i) \times \mathfrak{L}(c_i)$ do
        if $(s,t)$ has already been checked then
            continue
        end if
        if $overlap(s,t) \geq o_{min}$ then
            $\mathfrak{D} = \mathfrak{D} \cup \{(s,t)\}$
        end if
    end for
end for
return $\mathfrak{D}$
```

Algorithm 3 Find-Key-Key-Mappings. Given two duplicate nodes (s,t), this algorithm returns a set of key-key pairs M

```
$M = \emptyset$
$\sigma(k,k')$ = rate of common values shared by $k$ and $k'$
Let $K(s)$, $K(t)$ be the sets of keys in $s,t$.
for all $k' \in K(t)$ do
    Let $k \in K(s)$ be the key that maximizes $\sigma(k,k')$
    if $\sigma(k,k') > \sigma_{min}$ then
        $M = M \cup \{(k,k')\}$
    end if
end for
return M
```

5.2.2 Construct Rewrite Operations

As introduced in Section 3, give two identified nodes s and t which share a large portion of duplicate URLs, the process of creating a normalization rule $s \to t$ is to construct a series of rewrite operations for every URL key to transform the URL pattern of the node s to the format of t. To determine the operations and their types (keep/replace/ignore), the necessary information to know is the mapping relationships from the keys of s to the keys of t. More precisely, there is a mapping between two keys if they have very similar value distributions. Keys in a mapping relationship possibly trigger a "replace" operation; while keys without any mappings are more likely to take the "ignore" or "keep" operations. Algorithm 3 shows the pseudo codes to discover the mappings. We simply estimate the rate of common values shared by two keys and adopt a threshold σ_{min} to filter out mappings having lower possibilities. In implementation, σ_{min} is empirically set to 0.5. Based on the discovered key-key mapping relationships, the rewrite operations for every key in the target node t are determined as follows:

- ▷ **Keep** is the simplest one. If one key of t has a concrete value (not a regular expression), such as "path_2" of node B in Figure 4 (b) which has one unique value "display", one **keep** operation is created for this key. In normalization, we just directly fill the key with the related value.
- ▷ **Replace** is based on a key-key mapping between s and t. In general, two keys in such a mapping have the same function (e.g., denote product-ID or search query); and different values of these keys usually lead to different page content. For example, in Figure 4 (b), keys "pageid" of node A and "path_1" of node B both describe the ID of the page. These two keys sharing the same values like "10" and "21" will be identified as a key-key mapping in Algorithm 3. Consequently, a **replace** operation is created for "path_1" associated with the mapping key "pageid" from the source pattern. In normalization, we just copy the value of the source key (e.g., "pageid") to the target key (e.g., "path_1").
- ▷ **Ignore** is to say the values of a key do not affect (or are *irrelevant* to) the page content. There are two situations to generate an ignore operation:
  - – First, for a key of t, we create an **ignore** operation if the key has multiple values (denoted by "*") but no mapping relationship. No mapping means the values of this key never appear in s, which suggests the key is irrelevant to the page content. For example, the key "sid" of node B in Figure 4 (b) is ignored.
  - – The other case is to revise a **replace** operation. It should be noticed that "have a mapping relationship" is just a necessary condition but not a sufficient condition to "create a replace operation". In other words, a replace operation must rely on a mapping relationship but not all the mappings will lead to replace operations. For example, in Figure 4 (c), the two "path_0" keys from node A and node B share the same values (e.g., archive0 and archive3), and can pass the criterion in Algorithm 3 to set up a mapping. However, the values under these keys are actually used to describe the archive mirrors and have no impact to the page content. To identify such a "fake" replace, a test process is added to examine whether the key's values affect the page content. If too many URLs with different values on this key are duplicate, the **replace** operation is revised to an **ignore** operation accordingly.

5.2.3 Remove Unqualified Candidates

When generating a candidate rule, only duplicate samples from the two duplicate nodes s and t are taken into account. From the perspective of learning, only positive cases are observed but negative cases (non-duplicate URLs being rewritten to the same canonical format) are missed. Therefore, it is likely to produce "over fitting" rules which work properly on only a few positive samples but cannot be applied to more others. In other words, such unqualified rules will lead to many false-positive cases in practice. This is also discussed in [13] where a "false-positive filter" is proposed to clean these unqualified rules. Similarly, in our approach we examine each generated candidate rule by running it on the training set. If the false-positive rate is larger than a pre-defined threshold fpr_{max}, the rule is unqualified and is removed from the candidate set. In the experiments, we will show how the threshold affects the performance of the whole approach.

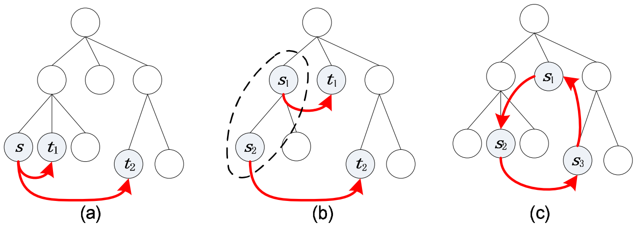


Figure 7: An illustration of three typical redundant and conflict cases: (a) two rules share the same source node; (b) one rule covers another; and (c) several rules construct a circle.

5.3 Deployable Rules Selection

As we have explained in the Introduction, the candidate rules cannot be directly deployed as some of them are redundant and conflict with each other. The redundant rules will degrade the computational efficiency and the conflict can undermine the performance or even crash the system. After investigating a plenty of candidate rules from various websites, we summarize three typical redundant and conflict cases, as shown in Figure 7:

(a) ***Two rules share the same source node.*** As shown in Figure 7 (a), there are two rules which transform node s to nodes t_1 and t_2, respectively. This doesn't make sense in practice as there should be only one canonical form for a URL. Moreover, two transformations also bring additional computational burden to the normalization system.

(b) ***One rule covers another.*** This is another kind of redundant rules. As shown in Figure 7 (b), as node s_1 is the ancestor of node s_2 (which means the pattern of s_1 is more general than that of s_2), URLs belonging to s_2 can also be normalized by the rule acting on s_1. Similar to the above case, this leads to multiple canonical targets (t_1 and t_2) and unnecessary computing work.

(c) ***Several rules construct a circle.*** This is a conflict case. As shown in Figure 7 (c), the nodes $s_1 \sim s_3$ are connected by three rules where the source of one rule is the target of another one. Such a conflict will crash the system as it leads to an endless loop in computing.

How to select an optimal sub-set of rules for deployment is not a trivial problem - the running stability and de-duping capability should be carefully considered. One straightforward solution is to remove the redundancies and conflicts with some ad-hoc strategies:

- – For case (a), keep only one rule for each node. The rule with the smallest false-positive rate is reserved and others are discarded.
- – For case (b), keep the rules acting on ancestor nodes and discard the rules on descendant nodes; or first apply the rules with smaller false-positive rates.
- – For case (c), remove the rule with minimum support to break the circle. Here, the support is defined as the number of URLs can be normalized by the rule.

Such an empirical solution does ensure the system stability but cannot guarantee the de-duping performance. We will demonstrate this in the experiments. Essentially, the above straightforward solution is designed from a local perspective and just addresses the rules in conflict. However, considering the goal of URL normalization, the key problem is how to select the canonical formats (i.e. choosing nodes as the *destinations* of the normalization), rather than judge which rules are with higher qualities.

In our observation, a good destination node should be general and popular. Being general means a node should have a clean format and can represent a variety of samples. Therefore, we can use the number of URLs covered by a node (called volume of a node) to approximate how general it is - the larger the number, the more general the node. Being popular means a node attracts more other nodes to be transformed to its format. Selecting popular nodes as destinations can maximize the compression performance. To estimate the popularity of a node, we can accumulate the volumes of all the nodes which can be normalized to the given node by some candidate rules. In this way, we can treat the volume of a node as a kind of *energy*, which can flow to other nodes through some normalization rules. Besides the rules, links from descendant nodes to their ancestors provide another way for energy transfer, as a descendant can always be "normalized" to its ancestors. When the flow of energy reaches to a stable status, nodes with higher energy are more likely to be selected as the destination nodes. This is very similar to the graph propagation adopted in the PageRank algorithm [8].

In detail, we construct a weighted directed graph $G =< V, E, w >$, where V is the set of vertices and E is the set of edges. One vertex v_i represents one tree node appearing in the candidate rules; and there exists an edge $e_{ij} \in E$ if there is a rule or a descendant-ancestor link from v_i to v_j. The weight w_{ij} of the edge e_{ij} is set to $1 - fpr_{ij}$ if the edge is constructed based on a rule r_{ij}; otherwise w_{ij} is simply set as 1 for a descendant-ancestor link. In this way, a natural random walk can be derived on the graph with the transition probability matrix $P = \{ p(i,j) \}$ defined by:

$$p(i,j) = \frac{w_{ij}}{outdegree(v_i)} \tag{7}$$

for all $e_{ij} \in E$, and 0 otherwise. Then each row of P is normalized to $\sum_j p(i,j) = 1$. Given the initial distribution $\pi^0 = (\pi_0^0, \ldots, \pi_n^0)^T$ where $\pi_i^0 = volumn(v_i)$, the final stable distribution can be estimated by $\pi^{stable} = (P^T)^\infty \pi^0$.

Once the destination nodes are determined, removing redundancies and conflicts becomes relatively easy, as shown in Figure 8. Figure 8 (a) shows a pattern tree with several candidate rules, from which we can find typical redundant and conflict cases as shown in Figure 7. Figure 8 (b) shows how nodes are connected to construct a graph, based on which the destination nodes are selected, as those in shadow in Figure 8 (c). Then we just need to remove those rules which start from the destinations. Figure 8 (d) and (e) also illustrate how to concatenate several chaining rules to one direct rule, which make the normalization more efficient.

6. EVALUATION AND DISCUSSION

In this section, we report the experimental results of applying our pattern tree-based approach on URL normalization. Both the evaluations of the three components and the overall performance are discussed. In comparison with state-of-the-art methods, our approach remarkably improves both the du-duping effectiveness and the computation efficiency.

6.1 Experiment Setup

As introduced in Section 1, some websites supporting Google's suggestion explicitly indicate the canonical URLs in their pages. This naturally provides a high quality dataset to evaluate the performance of URL normalization algorithms.

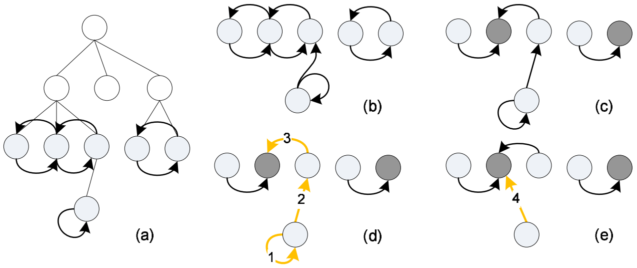


Figure 8: An illustration of how to select deployable rules: (a) several candidate rules on a pattern tree; (b) construct a graph with candidate rules and descendant-ancestor links; (c) select the nodes with the largest popularity as targets (nodes in shadow) and remove edges starting from these nodes; and (d) and (e) further concatenate the chaining rules (1+2+3) to one direct rule 4.

Table 1: Statistics of the experimental dataset.

	Min	Max	Average
#URL in a website	24,147	3,000,000	352,106
#Dup-cluster in a website	158	976,862	92,365
#URL in a dup-cluster	2	314,465	3.8

Table 2: Statistics of the constructed pattern-trees.

	Min	Max	Average
#Node on a tree	8	634	105
#Height of a tree	3	16	6
#Running Time (in seconds)	<1	257	22

In our experiment, we scanned a corpus of 20 billion pages and found out the top 200 websites supporting Google's convention. There are around 70 million URLs identified from the 200 websites, the statistics are shown in Table 1. The average duplicate ratio of the whole dataset is about 63.36%. For each website, we randomly select 20% samples (but no more than 0.1 million) as training set. The testing is carried out on the whole dataset.

In the experiment, we selected the method in [13] for comparison. To the best of our knowledge, this method is the most state-of-the-art approach which significantly outperforms other previous ones. According to the instructions in [13], we implemented their algorithm with the "fan-out" strategy, which is also the strategy utilized in their experiments. Actually, the "fan-out" is a threshold controlling the resolution of the rules. The smaller the threshold, the subtler the normalization rules, as well as the larger the number of total generated rules.

All the experiments were run on an x64 machine with 2.33GHz Intel®Xeon®CPU and 8G RAM. The operation system is Microsoft Windows 2003 Service Pack 2. All the programs were implemented with C# and .Net Framework 3.5 Service Pack 1.

6.2 Component Evaluations

In this subsection, we introduce some detailed information of the evaluations for the three components, respectively.

6.2.1 Pattern Trees

Pattern tree construction is the fundamental step of the proposed algorithm. Some statistics of the constructed trees are listed in Table 2. From Table 2, it is clear that the number of tree nodes (URL patterns) is much smaller than the

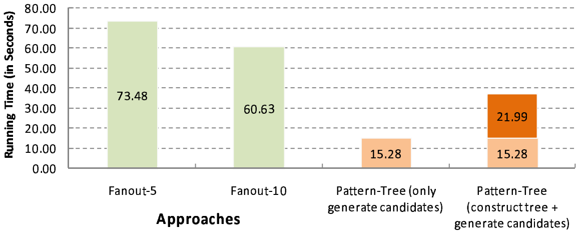


Figure 9: Comparisons of the average time (in seconds) to generate candidate rules for a website.

Table 3: Number of raw and qualified candidates generated by different methods for all the 200 websites ($fpr_{max} = 0.05\%$).

	#Raw Rule	#Qualified Rule	Rate
$R_{fanout\text{-}5}$	52,577	19,754	37.6%
$R_{fanout\text{-}10}$	36,149	13,813	38.2%
R_{tree}	14,912	6,373	42.7%

number of original URLs. Even for the most complicated case, the number of nodes is just over 600. Averagely, a pattern-tree just contains around 100 nodes. Moreover, the average height of the trees is just 6, which means the constructed trees are with relatively balanced structure.

We also evaluated the running time of this component. If the tree construction algorithm is not efficient enough, the computational advantage of the whole approach cannot be guaranteed. In practice, the computational cost depends on both the size of the dataset and the complexity of the URL syntax. From Table 2, in general we need around 22 seconds to build a pattern tree (based on around 70,000 URLs for each website). Totally, it took about 1.22 hours to process all the 200 websites. This time cost is considerable. As will be shown in the following discussion, it is still much cheaper in comparison with the traditional bottom-up pairwise strategies.

6.2.2 Candidate Rules

To provide a comprehensive comparison, we tried two different "fan-out" thresholds, $fan\text{-}out = 5$ and $fan\text{-}out = 10$, for the algorithm in [13]. The generated rule sets are called $R_{fanout\text{-}5}$ and $R_{fanout\text{-}10}$, respectively. For our pattern tree-based approach, the generated rule set is called R_{tree}.

First, the computational costs to generate the three rule sets are compared and shown in Figure 9. From Figure 9, it is clear our approach is more efficient than both $fanout$-5 and $fanout$-10. Averagely, our approach only needs around 15.3 seconds to generate candidate rules for one website; while $fanout$-5 and $fanout$-10 need 73.5 and 60.6 seconds, respectively. Even including the time cost of pattern tree construction, our approach totally needs 37.3 seconds to deal with one website and is still much faster than others. This demonstrates our proposed approach can remarkably improve the computational efficiency in comparison of the bottom-up pairwise strategy.

Second, we compare how many candidates can be generated using different methods, and how many of them are qualified. Table 3 shows the results with the threshold fpr_{max} set to 0.05. It is noticed the pattern tree-based approach generates quite less candidates but has higher qualified rate. To provide a more complete analysis, we vary fpr_{max} from 0 to 0.1 with an interval 0.01, and from 0.1 to 0.9 with an interval 0.1. The qualified rates of the three methods are shown in Figure 10. Again, our method shows better qualified rate in most cases.

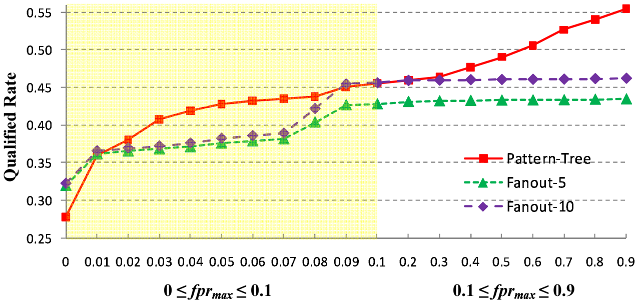


Figure 10: Comparisons of qualified rates with various fpr_{max}.

Table 4: Comparisons of the deployable rules selected from different rule sets using different selection strategies.

	#Candidates	#Deployable	%Compression
$R_{fanout\text{-}5} + S_{naive}$	19,754	10,433	18.8%
$R_{fanout\text{-}10} + S_{naive}$	13,813	7,434	16.0%
$R_{tree} + S_{naive}$	6,373	2,069	26.3%
$R_{tree} + S_{graph}$	6,373	1,171	34.5%

6.2.3 Deployable Rules

As there is no clear explanation of how the deployable rules are selected in [13], in the experiments we utilized the straightforward strategy (S_{naive}) presented in Section 5.3 to select the deployable rules from the candidate set $R_{fanout\text{-}5}$ and $R_{fanout\text{-}10}$. For comparison, we applied both the straightforward strategy and the graph-based strategy (S_{graph}) on our candidate set R_{tree}.

Table 4 shows the results of using different strategies on different rule sets. From Table 4, it is clear that the straightforward strategy selects more candidates as deployable rules. In comparison, the graph-based strategy only selects 1,171 rules for all the 200 websites in total. This means on average there are only around 6 rules deployed for each website; while the number of using "$R_{fanout\text{-}10}+S_{naive}$" is larger than 50. However, in Table 4 the compression rate of "$R_{tree} + S_{graph}$" is the best, although it has the smallest number of deployable rules. This indicates the graph-based strategy can successfully resolve conflicts and select those most effective rules for deployment. More comparisons will be introduced in the following subsection.

6.3 Overall Performance

In this subsection, we evaluate the overall performance (i.e., the de-duping ability) using different *methods*, *strategies*, and *thresholds*. We adopt the two standard criteria introduced in Section 3 - *compression rate* and *dup-reduction rate* - to measure the performance. Moreover, considering the numbers of URLs from various websites are of different scales, we utilize both the micro- and macro- averages of the two criteria for evaluation, to avoid that some big websites dominant the performance. The micro-average takes all the URLs from various websites as a whole and calculates the criteria; while the macro-average first calculates the criteria for each website independently, and then adds them up and builds the averaged overall performance. The macro-average gets rid of the bias from big websites and shows how the approaches can deal with different duplicate cases. The micro-average provides a fair approximation to the per-

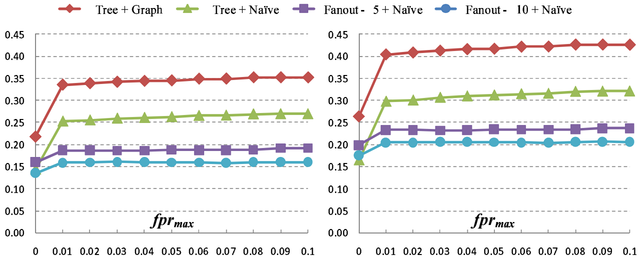


Figure 11: Comparisons of the micro-average (a) and macro-average (b) of the *compression rates* using different methods, strategies, and thresholds.

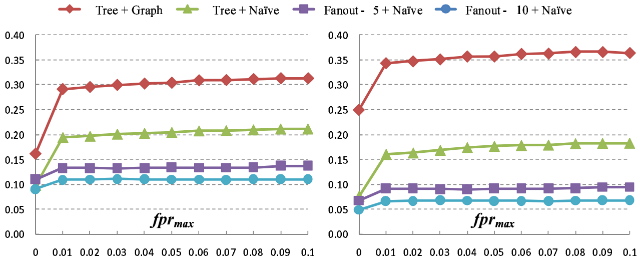


Figure 12: Comparisons of the micro-average (a) and macro-average (b) of the *dup-reduction rates* using different methods, strategies, and thresholds.

formance in real applications, in which websites are indeed of different scales.

For the *methods* and *strategies*, we adopt the same four combinations in Table 4. For the threshold fpr_{max}, we just vary it from 0 to 0.1 with an interval 0.01, because a large false-positive rate has little impact to practical applications.

Figure 11 shows the micro-average and macro-average of the *compression rates*. Notably, the rules generated by the proposed pattern tree-based approach can compress more redundant duplicates than those rules produced by $fanout$-5 and $fanout$-10. In addition, the graph-based strategy can select more effective rules than the straightforward strategy by comparing the compression rates of "Tree+Graph" and "Tree+Naive". In the best case, our approach can remove around 35% URLs (around 24.5 million URLs from all the 70 millions in the dataset). If these rules are integrated into a crawler, it can save considerable bandwidth and storage. Figure 12 shows the micro-average and macro-average of the *dup-reduction rates*. Similar to those observed in Figure 11, the pattern-tree-based approach removes more duplicates than $fanout$-5 and $fanout$-10, and the graph-based strategy exceeds the naive (straightforward) strategy.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we present a pattern tree-based approach to learning URL normalization rules. The main contribution of this work is to rethink the problem from a global perspective and leverage the statistical information from the entire training set. We first construct a pattern tree based on the training set, and then generate normalization rules via identifying duplicate nodes on the pattern tree. With the statistical information, the learning process is more robust to both the noise and incompleteness of the training samples. The computational cost is also low as rules are directly induced on patterns, rather than on every duplicate URL pair. Moreover, practical deployment issues have been discussed in this paper and a graph-based strategy has been proposed to select a subset of deployable rules for normalization. Experiment evaluations on 70M URLs show very promising results. Compared to the state-of-the-art approaches, our system can greatly reduce duplicates with much little false-positives. Besides, our system significantly reduces the running time of the learning process by around 50%.

There is still room to improve the propose method. First, the pattern tree construction algorithm should be further accelerated as it is the current computational bottleneck. Second, the current algorithms fully trust the duplicate information labeled in the training set. However, in practice, auto-generated training sets may contain some fake duplicate information caused by, for example, too relaxed parameters in the shingle print algorithms. How to deal with such fake duplicates is another goal of our future work.

8. REFERENCES

[1] Google webmaster central blog: specify your canonical. http://googlewebmastercentral.blogspot.com/2009/02/specify-your-canonical.html.

[2] Uniform Resource Identifier (URI): Generic Syntax. RFC 3986. http://tools.ietf.org/html/rfc3986.

[3] URL Normalization. http://en.wikipedia.org/wiki/URL_normalization.

[4] A. Agarwal, H. S. Koppula, K. P. Leela, K. P. Chitrapura, S. Garg, and P. K. GM. URL normalization for de-duplication of web pages. In *Proc. CIKM*, pages 1987–1990, 2009.

[5] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[6] D. Angluin. Finding patterns common to a set of strings. In *SOTC*, pages 130–141, 1979.

[7] Z. Bar-Yossef, I. Keidar, and U. Schonfeld. Do not crawl in the dust: different URLs with similar text. In *WWW*, pages 111–120, 2007.

[8] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1–7):107–117, 1998.

[9] A. Broder, S. C. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the Web. *Computer Networks*, 29(8–13):1157–1166, 1997.

[10] A. C. Carvalho, E. S. Moura, A. S. Silva, K. Berlt, and A. Bezerra. A cost-effective method for detecting web site replicas on search engine databases. *Data Knowl. Eng.*, 62(3):421–437, 2007.

[11] M. Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. SOTC*, pages 380–388, 2002.

[12] A. Chowdhury, O. Frieder, D. A. Grossman, and M. C. McCabe. Collection statistics for fast duplicate document detection. *TOIS*, 20(2):171–191, 2002.

[13] A. Dasgupta, R. Kumar, and A. Sasturkar. De-duping URLs via rewrite rules. In *KDD*, pages 186–194, 2008.

[14] M. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, pages 284–291, 2006.

[15] G. S. Manku, A. Jain, and A. D. Sarma. Detecting near-duplicates for web crawling. In *Proc. WWW*, pages 141–150, 2007.

[16] M. Najork. Systems and methods for inferring uniform resource locator (URL) normalization rules. US Patent Application Publication, 2006/0218143, Microsoft Corporation, 2006.

[17] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[18] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *Proc. WWW*, pages 131–140, 2008.