REMOVING DUST USING MULTIPLE ALIGNMENT OF SEQUENCES

KAIO WAGNER LIMA RODRIGUES

REMOVING DUST USING MULTIPLE ALIGNMENT OF SEQUENCES

Tese apresentada ao Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal do Amazonas como requisito parcial para a obtenção do grau de Doutor em Informática.

Orientador: Marco Antônio Pinheiro de Cristo

Manaus

Setembro de 2016

KAIO WAGNER LIMA RODRIGUES

REMOVING DUST USING MULTIPLE ALIGNMENT OF SEQUENCES

Thesis presented to the Graduate Program in Computer Science of the Universidade Federal do Amazonas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Marco Antônio Pinheiro de Cristo

Manaus

September 2016

Rodrigues, Kaio Wagner Lima

R696r Removing DUST using multiple alignment of sequences / Kaio Wagner Lima Rodrigues. — Manaus, 2016

xxv, 91 f. : il. ; 29cm

Tese (Doutorado em Informática) — Universidade Federal do Amazonas

Orientador: Marco Antônio Pinheiro de Cristo

1. Motores de busca. 2. Coleta. 3. Eliminação de Duplicatas. 4. Normalização de URLs. 5. Regras de Reescrita. I. Cristo, Marco Antônio Pinheiro de II. Universidade Federal do Amazonas III. Título.

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
INSTITUTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

UFAM

FOLHA DE APROVAÇÃO

"Removing DUST Using Multiple Alignment of Sequences"

KAIO WAGNER LIMA RODRIGUES

Tese de Doutorado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Marco Antonio Pinheiro de Cristo - PRESIDENTE

Profa. Eulanda Miranda dos Santos - MEMBRO INTERNO

Prof. Raimundo da Silva Barreto - MEMBRO INTERNO

Prof. Adriano Alonso Veloso - MEMBRO EXTERNO

Profa. Claudine Santos Badue Gonçalves - MEMBRO EXTERNO

Manaus, 21 de Setembro de 2016

Aos meus pais por toda dedicação, carinho e renúncias.

Acknowledgments

Em primeiro lugar, agradeço aos meus pais, Jonas e Luzdete, pelo amor, apoio e pelos sacrifícios realizados para que eu pudesse chegar até aqui. Agradeço por serem compreensivos e por terem me apoiado na escolha da minha carreira.

Agradeço à Elane Ferreira, meu amor, que esteve sempre ao meu lado com muita compreensão, ajudando a adoçar os momentos mais difíceis com suas palavras de conforto e incetivo.

Agradeço ao meu orientador, Marco Cristo, por ter me aceitado como seu aluno, pela preocupação, paciência, disponibilidade e conhecimento transmitido. Muito obrigado pelos direcionamentos e pelas horas de dedicação.

Meus agradecimentos também se estendem aos professores Altigran Soares e Edleno Silva de Moura, pelo grande auxílio e contribuições dados quando este trabalho ainda estava em seu estágio inicial.

Agradeço aos amigos Bruno Campos, André Carvalho, Marcio Palheta, Caio Daoud, Thiago Salles e todas as outras pessoas que participaram de forma direta ou indireta na conclusão desta tese.

Ao pessoal da secretaria do departamento, pela simpatia e disposição em ajudar. Em especial à Elienai Nogueira, pela amizade e ajuda desde que eu era calouro do curso de Ciência da Computação.

Agradeço também ao Governo do Estado do Amazonas por meio da Fundação de Amparo à Pesquisa (FAPEAM), pelo auxílio financeiro destinado a realização desta pesquisa.

Acima de tudo, agradeço a Deus por ter me concedido saúde, sabedoria, força e proteção neste momento tão importante da minha vida.

"Porque muitos são chamados,
mas poucos escolhidos."
(Mateus 22:14)

Resumo

Um grande número de URLs obtidas por coletores corresponde a páginas com conteúdo duplicado ou quase duplicado, conhecidas em Inglês pelo acrônimo DUST, que pode ser traduzido como Diferentes URLs com Texto Similar. DUST são prejudiciais para sistemas de busca porque ao serem coletadas, armazenadas e utilizadas, contribuem para o desperdício de recursos, a criação de rankings de baixa qualidade e, consequentemente, uma experiência pior para o usuário. Para lidar com este problema, muita pesquisa tem sido realizada com intuito de detectar e remover DUST antes mesmo de coletar as URLs. Para isso, esses métodos se baseiam no aprendizado de regras de normalização que transformam todas as URLs com conteúdo duplicado para uma mesma forma canônica. Tais regras podem ser então usadas por coletores com o intuito de reconhecer e ignorar DUST. Para isto, é necessário derivar, de forma eficiente, um conjunto mínimo de regras que alcance uma grande taxa de redução com baixa incidência de falsos-positivos. Como a maioria dos métodos propostos na literatura é baseada na análise de pares, a qualidade das regras é afetada pelo critério usado para selecionar os exemplos de pares e a disponibilidade de exemplos representativos no treino. Para evitar processar um número muito alto de exemplos, em geral, são aplicadas técnicas de amostragem ou a busca por DUST é limitada apenas a sites, o que impede a geração de regras que envolvam diferentes nomes de DNS. Como consequência, métodos atuais são muito sucetíveis a ruído e, em muitos casos, derivam regras muito específicas. Nesta tese, é proposta uma nova técnica para derivar regras, baseada em uma estratégia de alinhamento múltiplo de sequências. Em particular, mostramos que um alinhamento prévio das URLs com conteúdo duplicado contribui para uma melhor generalização, o que resulta na geração de regras mais efetivas. Através de experimentos em duas diferentes coleções extraídas da Web, observa-se que a técnica proposta, além de ser mais rápida, filtra um número maior de URLs duplicadas. Uma versão distribuída do método, baseada na arquitetura MapReduce, proporciona a possibilidade de escalabilidade para coleções com dimensões compatíveis com a Web.

Palavras-chave: Motores de busca, Coleta, Eliminação de Duplicatas, Normalização de URLs, Regras de re-escrita.

Abstract

A large number of URLs collected by web crawlers correspond to pages with duplicate or near-duplicate contents. These duplicate URLs, generically known as DUST (Different URLs with Similar Text), adversely impact search engines since crawling, storing and using such data imply waste of resources, the building of low quality rankings and poor user experiences. To deal with this problem, several studies have been proposed to detect and remove duplicate documents without fetching their contents. To accomplish this, the proposed methods learn normalization rules to transform all duplicate URLs into the same canonical form. This information can be used by crawlers to avoid fetching DUST. A challenging aspect of this strategy is to efficiently derive the minimum set of rules that achieve larger reductions with the smallest false positive rate. As most methods are based on pairwise analysis, the quality of the rules is affected by the criterion used to select the examples and the availability of representative examples in the training sets. To avoid processing large numbers of URLs, they employ techniques such as random sampling or by looking for DUST only within sites, preventing the generation of rules involving multiple DNS names. As a consequence of these issues, current methods are very susceptible to noise and, in many cases, derive rules that are very specific. In this thesis, we present a new approach to derive quality rules that take advantage of a multi-sequence alignment strategy. We demonstrate that a full multi-sequence alignment of URLs with duplicated content, before the generation of the rules, can lead to the deployment of very effective rules. Experimental results demonstrate that our approach achieved larger reductions in the number of duplicate URLs than our best baseline in two different web collections, in spite of being much faster. We also present a distributed version of our method, using the MapReduce framework, and demonstrate its scalability by evaluating it using a set of 7.37 million URLs.

Keywords: Search engines, Crawling, De-duplication, URL Normalization, Rewrite rules.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Context

Search engines are often faced with a number of challenging aspects to be tackled in order to maintain an efficient resource usage and guarantee their scalability. One such challenge is the presence of a large amount of URLs on the web that points to duplicate (or near-duplicate) content. Such syntactically different URLs linking to identical or closely similar content are called duplicate URLs or DUST (Different URLs with Similar Text [Bar-Yossef et al., 2006]). In fact, such URLs are quite common on the Web. As an example, in [Fetterly et al., 2003], the authors stated that, in a corpus of 20 billion fetched documents, around one-quarter are duplicates.

Duplicate URLs bring serious issues to the whole pipeline of a search engine, from crawling to result serving. Considering the crawling module of a web search engine, bandwidth and time are wasted to download (near-)duplicate content. Furthermore, politeness rules may be compromised, incurring in potentially aggressive requests. When indexing, memory is wasted when storing redundant information and the index itself becomes bloated and inefficient. Moreover, link analysis used, for example, to provide better download schedules, may be considerably affected, hampering the web search efficiency. Finally, the user experience may be considerably prejudiced due to the presentation of polluted results. Clearly, as such kind of URLs do not add any new information to the final search engine user, it just serves to negatively affect the search engine and should be avoided.

A standard way to eliminate duplicate pages from search engines consists to fetch the content of the URLs and then apply fingerprint methods to discard the similar ones [Broder, 1997; Broder et al., 1997; Charikar, 2002]. Although this approach can be accurately used to limit and diversify the set of search results, it requires the pages

to have been crawled. In other words, it still needs to download duplicate content and the precious bandwidth and storage cannot be saved. Thus, a better solution to this problem involves detecting (near-) duplicate content as early as possible in the web search pipeline (e.g., before fetching duplicate content).

In recent years, several authors have proposed strategies for detecting DUST that inspect only the URLs without fetching the corresponding page [Bar-Yossef et al., 2006; Dasgupta et al., 2008; Agarwal et al., 2009; Koppula et al., 2010; Lei et al., 2010]. The idea of these methods, known as URL-based de-duping, is mine crawl logs and use clusters of URLs referring to (near) duplicate content[1] to learn *normalization rules*. These rules (also known as DUST rules [Bar-Yossef et al., 2006] or rewrite rules [Dasgupta et al., 2008]) are able to transform duplicate URLs to a same canonical form. This information can be used by a web crawler to avoid fetching DUST, including ones that are found for the first time during the crawling.

1.2 Problem

The more general problem related to this thesis is to determine if two URLs, u_i and u_j, could be transformed one into another by means of only syntactical modifications between equivalent structural components of the URLs. For instance, in URLs u_i = `http://www.google.com/index.html`[2] and u_2 = `http://www.google.com/index`, substrings `index.htm` and `index` are equivalent structural components which indicate the site entry page. Thus, u_1 could be transformed into u_2 and vice versa, which indicates they point to the same content. Given URLs u_i and u_j, this problem is also equivalent to find a canonical representation $f(u)$, such that $f(u_i) = f(u_j)$ if u_i can be transformed into u_j and vice versa. Note that $f(u)$ does not need to be a valid URL.

In this work, we address the problem of determining the canonical representation $f(u)$. To determine which operations should be used to transform a URL into another, recent literature has focused on mining transformation rules from examples of duplicate URLs. For instance, from our previous example, we could extract rule `index.html` $\rightarrow$ `index`. Thus, the problem we address in this thesis can be described as follows. Let $\mathcal{U}$ be a set of URLs partitioned into n disjoint clusters D_1, D_2, ..., D_n. Each cluster D_i, from now on referred to as *dup-cluster*, groups URLs which point to the same content. Thus, from URLs $u_{i,1}$, $u_{i,2}$, ..., $u_{i,m}$, belonging to dup-cluster D_i, only one should be fetched as they are duplicates. Given these dup-clusters, we need to determine an

[1] Webmasters explicitly assist search engines with the creation of these clusters when they indicate which URLs are DUST by using the HTML element called canonical tag [Lei et al., 2010].

[2] Along this work, we use typographical font to indicate a literal or a sequence of literal characters.

effective set of transformation rules that, when applied to two URLs u_i and u_j, allow us to say whether the URLs can be transformed into each other, i.e., $f(u_i) = f(u_j)$. By an effective set of rules, we understand (i) as small as possible; (ii) precise, specially to avoid false positives and, consequently, consider new content as duplicate; and (iii) generic enough to remove the maximum of DUST.

1.3 Research Hypotheses and Questions

The main challenge for DUST detection methods is to derive general rules with a reasonable cost from the available training sets. As observed in [Lei et al., 2010], many methods derive rules from pairs of duplicate URLs. They take advantage of the general syntactical structure of a URL to find URL substrings (tokens) that could induce string substitution rules. Such rules can be complex, dealing with patterns such as tokens transpositions (e.g., the cases of `jogos` and `uol.com.br/` in URLs `http://jogos.uol.com.br/` and `http://uol.com.br/jogos`) and the recognition of parameters (e.g., although similar, URLs `http://uol.com.br/?id=5` and `http://uol.com.br/?id=7`, are likely different since the last token is a parameter which assumes different values).

As most methods are based on pairwise analysis, the quality of the rules is affected by the criterion used to select the pairs of examples and the availability of specific examples in the training sets. To avoid processing large numbers of URLs, they employ techniques such as random sampling or by looking for DUST only within sites, preventing the generation of rules involving multiple DNS names. As a consequence of these issues, current methods are very susceptible to noise and, in many cases, derive rules that are very specific.

Although prohibitive, the methods should learn general rules from more than two training examples, taking maximum advantage of them and without sacrificing the detection of DUST across different sites. To cope with this issue, we first observe that the general problem of finding rules to transform URL u_i into u_j is similar to the problem of sequence alignment, that is, finding an optimum set of edit operations able to transform a sequence of symbols into another. For instance, the rules to transform URL $u_i =$ `http://google.com/index.html` into $u_j =$ `http://www.google.com/index` can be derived from the best alignment between u_i and u_j, illustrated as follows:

```
http://    google.com/index.html
|||||||++++|||||||||||||||-----
http://www.google.com/index
```

Inspecting such alignment, u_i can be transformed into u_j by inserting characters `w`, `w`, `w`, and `.` after the second slash (`/`) and by removing `.`, `h`, `t`, `m`, and `l` at the end of u_i. If we consider, as Brill and Moore [2000], that the symbols to be aligned could be substrings instead of characters[3], this alignment suggests that u_i can be transformed into u_j by applying transformation rules for inserting substring `www.` after `/` and deleting substring `.html` at the end of an URL.

Similarly, the problem of finding a general set of rules to transform any URL within a dup-cluster to a canonical form can be viewed as similar to the problem of multiple alignment of sequences. The optimum alignment of several sequences finds similar and dissimilar tokens among all the strings according to cost functions that take into consideration specific properties of the domain of the problem. We illustrate this kind of alignment below:

```
http://       google.com/index.html
|||||||       ||||||||||||
http://   www.google.com/index
|||||||       ||||||||||||
http://mirror.google.com/index
```

where we included a new duplicated URL, from a different site[4], in the previous example. The alignment clearly indicates the special situations soon after the second slash and at the end of the URLs. We can now infer that the optional nature of `.html` is likely a general convention and not a site specific pattern.

[3]Brill and Moore studied the problem of finding the best alignment of two strings in the domain of spelling correction. They observed that, in many cases, typos can be better modeled as syllable transformations instead of character transformations. For instance, given two strings, `unfisical` and `unphysical`, the typo is more likely due to the fact that the user mistakes syllables `phy` and `fi` by each other than to the fact that she uses to mistype `f` by `ph` after `n`. They then proposed that a string should be viewed as a set of syllables such that the entire syllables should be taken as single tokens in the alignment.

[4]In this work, we use the hostname as a site identifier (cf. Section 2.3.2).

The problem of multiple sequence alignment has been largely studied, specially in the domain of molecular biology. In fact, the importance of such technique in biology has motivated many efforts concerning the proposition of optimized algorithms [Katoh et al., 2002; Blackshields et al., 2010]. In general, specific characteristics of the domain are explored to improve computational performance such that more sequences can be evaluated in less time.

Surprisingly, despite the similarity between the problems and the need of learning general rules from more than two URLs, no previous work in literature has proposed techniques based on multiple sequence alignment. As these methods find patterns involving all the available strings, it would be possible to find more general rules avoiding problems related to pairwise rule generation, and the problem related to finding rules across sites. And since that very efficient multiple sequence alignment algorithms have been proposed, this technique could be used as a feasible general approach to identify similarities and differences among all URLs. Based on these ideas, we formulate the first hypothesis in this thesis:

> **Hypothesis 1**: *The identification of similarities and differences among* all *URLs can be explored to determine fixed and mutable substrings in URLs. Such similarities and differences can be effectively identified by a multiple sequence alignment approach. Since the substrings are considered fixed or mutable based on the analysis of multiple URLs, more general normalization rules can be derived.*

We also observe that, in spite of a rule can be better generalized by the inspection of patterns among different sets of duplicate URLs, they are extracted from individual and independent dup-clusters. For instance, by inspecting URLs within a dup-cluster, it would be possible to observe that `.html` is optional after `index`. The reliability of such rule is higher if it is also observed in other dup-clusters. Anyway, the initial learning of the rules is clearly based on independent data sources. We could take advantage of this fact to mitigate the overall cost of performing full multiple sequence alignments. The intuition is that, as the training is based on a large number of such data sources, a highly parallelizable induction algorithm can be devised, where rules are learned in parallel on a per-dup-cluster basis and then confirmed in other dup-clusters. This leads us to the second hypothesis in this thesis:

> **Hypothesis 2**: *A fast rule induction algorithm can be devised by taking advantage of the organization of the duplicate URLs, used for training, in*

independent dup-clusters. Such algorithm should learn rules in parallel on a per-dup-cluster basis.

Given the previously presented hypotheses, some questions arise:

- How effective is a DUST detection approach based on a multiple sequence alignment when compared to traditional approaches? More specifically, how effective is it regarding rule precision, the number of rules generated, and DUST removal?
- Is it possible to take advantage of specific characteristics of the DUST detection problem to improve the computational performance of the alignement algorithm, as observed in other domains such as computational biology?
- How efficient is the approach when the rules are learned in parallel on a per-dup-cluster basis?

In this work, we intend to provide answers to such questions. The pursuit for these answers defined our objectives, described in next section.

1.4 Objectives

The general objective of the research described in this work is to propose a method for web-scale DUST detection to obtain a small and general set of normalization rules when compared with state-of-the-art methods. This objective translates into the following specific objectives:

- Development of a DUST detection method based on multiple sequence alignment. The set of rules delivered by the method is expected to be *small* enough to be used by robots which operate using no much memory. When compared to other state-of-the-art approaches, it is also expected to be at least as *precise* in avoiding false positives and more *generic*, such that more duplicate URLs are detected.
- Development of a multiple sequence alignment algorithm which takes advantage of specific characteristics of the DUST detection problem to induce rules *faster* than (i) a method based on traditional multiple sequence alignment and (ii) other state-of-the-art DUST detection approaches.
- Development of a parallel version of the training algorithm which takes advantage of the cluster of computers normally used in the environments where large

scale crawlings are performed. Such method should operate in web scale scenarios, inducing rules faster than any sequential method previously proposed with performance gains proportional to the number of available computing nodes.

1.5 Contributions

We show in this thesis that a full multi-sequence alignment of duplicate URLs, performed before rules are generated, can make the learning process more robust and less susceptible to noise when compared to previous work in the literature. Furthermore, our method is able to generate rules involving multiple DNS names and its parallel version is very efficient even when applied in large scale scenarios. The complexity of our sequential and parallel algorithms is proportional to the number of URLs to be aligned, unlike other methods where the complexity is proportional to the number of specific rules generated from all clusters, which can be unfeasible in practice.

The contributions made during this research are summarized as follows:

1. A DUST detection method based on a traditional multi-sequence alignment technique able to deliver general normalization rules involving multiple DNS names;
2. A comprehensive evaluation of the proposed method and its variations including comparison with state-of-the-art baselines;
3. Description of the entire crawling architecture the method will be part of;
4. A dataset composed of 3.86 million Brazilian web pages with duplicates annotated according to the canonical tag.
5. A sequential DUST detection method with a linear multi-sequence alignment approach;
6. A scalable and efficient parallel DUST detection method able to operate at large web scale scenarios;

The first two items above were firstly reported in the paper by [Rodrigues et al., 2013] entitled "Learning URL Normalization Rules Using Multiple Alignment of Sequences" and presented in the 20th International Symposium on String Processing and Information Retrieval (SPIRE 2013). Contributions 3 and 4 were included in a following work by [Rodrigues et al., 2015], published on the IEEE Transactions on Knowledge Data Engineering and entitled "Removing DUST Using Multiple Alignment of Sequences." The consolidated results of this thesis were submitted to the ACM Transactions on

Web (ACM ToW), entitled "A Highly-Scalable Algorithm For Generating URL Normalization Rules." This work, currently under review, mainly focuses on contributions 5 and 6.

1.6 Thesis Organization

This work is organized as follows. In Chapter 2 we provide basic concepts necessary to understand our work. In Chapter 3 we present a compilation of the works on the literature that are related to ours. In Chapter 4 we present *DUSTER*, a DUST detection method based on a traditional multi-sequence alignment technique. In Chapter 5 we present *DustLin*, a sequential DUST detection method with a linear multi-sequence alignment approach. We then present *DustLin-MR*, a scalable and efficient parallel DUST detection method. The experiments and the results are presented in Chapter 6. Finally, Chapter 7 concludes this thesis and presents future work.

Chapter 2

Background

This chapter introduces some basic concepts required for a better understanding of the method proposed in this thesis. We describe sequence alignment in Section 2.1. In Section 2.2, we describe search engines, their main components and how they are affected by the presence of DUST. In Section 2.3, we define Web concepts such as URL, web sites and canonical tags. In Section 2.4, we introduce notation and concepts related to regular expressions. Finally, in Section 2.5, we present the main problem addressed in this work, the detection of DUST.

2.1 Sequence Alignment

Sequence alignment is a fundamental procedure used in molecular biology where two or more biological sequences (DNA, RNA, or protein) are arranged such that similar regions are identified. These regions, common to most of the sequences in the group, reveal similarities which are consequence of structural, functional or evolutionary relationships between the sequences. Beside finding similar substrings, this information can be used to calculate the distance between sequences. In our context, we are interested in finding fixed and mutable structural parts of URLs.

2.1.1 Pairwise Sequence Alignment

The alignment of two sequences, called *pairwise sequence alignment*, is the basic step for aligning an arbitrary number of sequences. This problem can typically be solved using dynamic programming to calculate all the subproblems involved in the process [Needleman and Wunsch, 1970]. We formally define this concept as given in Definition 1.

$$S = \begin{array}{c} \\ Y \\ G \\ G \\ A \\ T \\ C \\ G \\ A \end{array} \begin{array}{c} \begin{array}{cccccccccccc} X & G & A & A & T & T & C & A & G & T & T & A \end{array} \\ \begin{bmatrix} \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \nwarrow \mathbf{1} & \leftarrow 1 & \leftarrow 1 & \leftarrow 1 & \leftarrow 1 & \leftarrow 1 & \leftarrow 1 & \nwarrow 1 & \leftarrow 1 & \leftarrow 1 & \leftarrow 1 \\ 0 & \uparrow \mathbf{1} & \leftarrow \mathbf{1} & \leftarrow 1 & \leftarrow 1 & \leftarrow 1 & \leftarrow 1 & \leftarrow 1 & \nwarrow 2 & \leftarrow 2 & \leftarrow 2 & \leftarrow 2 \\ 0 & \uparrow 1 & \nwarrow 2 & \nwarrow \mathbf{2} & \leftarrow \mathbf{2} & \leftarrow 2 & \leftarrow 2 & \nwarrow 2 & \leftarrow 2 & \leftarrow 2 & \leftarrow 2 & \nwarrow 3 \\ 0 & \uparrow 1 & \uparrow 2 & \leftarrow 2 & \nwarrow 3 & \nwarrow \mathbf{3} & \leftarrow 3 & \leftarrow 3 & \leftarrow 3 & \nwarrow 3 & \nwarrow 3 & \leftarrow 3 \\ 0 & \uparrow 1 & \uparrow 2 & \leftarrow 2 & \uparrow 3 & \leftarrow 3 & \nwarrow \mathbf{4} & \leftarrow \mathbf{4} & \leftarrow 4 & \leftarrow 4 & \leftarrow 4 & \leftarrow 4 \\ 0 & \nwarrow 1 & \uparrow 2 & \leftarrow 2 & \uparrow 3 & \leftarrow 3 & \uparrow 4 & \leftarrow 4 & \nwarrow \mathbf{5} & \leftarrow \mathbf{5} & \leftarrow \mathbf{5} & \leftarrow 5 \\ 0 & \uparrow 1 & \nwarrow 2 & \nwarrow 3 & \leftarrow 3 & \leftarrow 3 & \uparrow 4 & \nwarrow 5 & \leftarrow 5 & \leftarrow 5 & \leftarrow 5 & \nwarrow \mathbf{6} \end{bmatrix} \end{array}$$

Figure 2.1: Example of alignment of two sequences, $X =$ **GAATTCAGTTA** and $Y =$ **GGATCGA**. We assume a simple scoring function, where $sf(X_i, Y_j) = 1$ if $X_i = Y_j$. The aligner first creates a matrix S with $M + 1$ columns and $N + 1$ rows (M and N correspond to the size of X and Y, respectively) initializing it with zeros. Then, the value of cells $S_{i,j}$, $i \neq 0$ and $j \neq 0$, is filled according to Equation 2.1, starting with $S_{1,1}$. For example, the value of the cell at $(1, 1)$ is given by $S_{1,1} =$ **MAX** $[S_{0,0} + 1, S_{1,0}, S_{0,1}] =$ **MAX** $[1, 0, 0] = 1$.

Definition 1 *(Pairwise Sequence Alignment) Let X and Y be two sequences of characters, and $|X|$ and $|Y|$ represent their respective lengths. The pairwise sequence alignment between them is a mapping of X and Y to other two sequences X' and Y' with the same characters and in the same order, with possible inserted spaces (also known as gaps) such that $|X'| = |Y'|$ and X'_i (Y'_i) is gap only if Y'_i (X'_i) is not.*

Given the sequences X and Y with m and n characters respectively, the alignment process can be described by using a matrix S of size $(m+1)\times(n+1)$ where the sequence X is placed along the top of the matrix and sequence Y is placed along the left side, so that S cells are filled as follows:

$$S_{i,j} = \begin{cases} 0 & if \text{ i=0 or j=0} \\ max \begin{pmatrix} S_{i-1,j-1} + sf(X_i, Y_j), \\ S_{i-1,j}, \\ S_{i,j-1} \end{pmatrix} & otherwise \end{cases} \tag{2.1}$$

where $sf(X_i, Y_j)$ is a scoring function that defines a similarity between the pairs of symbols (X_i, Y_j). This function gives points for matching tokens and penalties for any gap.

Once the alignment matrix shown in Equation 2.1 has been computed, the best alignment can be deduced from it by using a process called *traceback*. By establishing pointers during the solution of the subproblems, it is possible to reconstruct the optimal solution calculated by dynamic programming. This process works as follows. When the value of cell $S_{i,j}$ is computed, a pointer from $S_{i,j}$ is set to cell (a) $S_{i,j-1}$ if $S_{i,j} = S_{i,j-1}$; (b) $S_{i-1,j}$ if $S_{i,j} = S_{i-1,j}$; or (c) $S_{i-1,j-1}$ if $S_{i,j} = S_{i-1,j-1} + sf(X_i, Y_j)$. The traceback

step determines the alignment that result in the maximum score. As illustrated in Figure 2.1, the traceback process always begins at the bottom right cell. Each leftward arrow in the path indicates that a gap has to be inserted into X while an upward arrow indicates a gap to be inserted into Y. A diagonal arrow indicates the symbols from the two sequences are aligned and no gap should be inserted. The traceback is completed when the first top-left cell of the matrix is reached. The score in the last cell (bottom right) represents the alignment score for the best alignment. For the example illustrated in Figure 2.1, the maximum alignment score for the two test sequences is 6 and the best alignment is:

```
X' = G_AATTCAGTTA
     |  | || |  |
Y' = GG_A_TC_G__A
```

2.1.2 Multiple sequence alignment

Given $k > 2$ sequences $\mathcal{S} = \{S_1, S_2, ..., S_k\}$, a Multiple Sequence Alignment (MSA) of $\mathcal{S}$ can be considered a natural generalization of the pairwise alignment problem. Spaces are inserted at arbitrary positions in any of the k sequences to be aligned, so that the resulting sequences have the same size ℓ. The sequences can be arranged in k lines and ℓ columns, such that each element or gap of each sequence is in a single column.

Definition 2 *(Multiple Sequence Alignment) Let $\{S_1, S_2, S_3, ..., S_k\}$ be sequences of characters, and let $|S_i|$ represent the size of S_i. The Multiple Sequence Alignment among S_1 to S_k is a mapping of $\{S_1, S_2, ..., S_k\}$ to other sequences $\{S'_1, S'_2, ..., S'_k\}$ such that S'_i has the same characters of S_i in the same order with possibly the addition of spaces (also known as gaps) and $|S'_1| = |S'_2| = ... = |S'_k|$.*

As the multiple sequence alignment problem is known to be NP-hard, several heuristics have been proposed in literature. Although recent progress have focused in *iterative* and *consistency based* strategies[1], benchmarking on an alignment reference dataset indicates that methods which use progressive alignment perform reasonably

[1]Iterative algorithms are based on the idea that the solution to a given problem can be computed by already existing sub-optimal solution and each modification step is an iteration. Consistency-based algorithms take advantage of conservation across many sequences to provide a stronger signal for pairwise comparisons. For a comprehensive review of the available MSA methods we refer the reader to the surveys by [Notredame, 2002] and by [Wang et al., 2015]

well on a wide range of situations [Notredame, 2002]. It also shows that iterative and consistency based methods are more appropriate for sequences with long insertion/deletions. As we do not expect such long insertions/deletion in the URLs within the dup-clusters, we adopted the most popular heuristic known as *Progressive Alignment* ([Feng and Doolittle, 1987]) to align clusters of duplicate URLs.

In general lines, the method first performs the alignment between two previously selected sequences. Then a new sequence is chosen and aligned with the first alignment obtained or another pair of sequences is selected and aligned. This process is repeated until all sequences have been aligned, giving rise to the final multiple alignment. Thus, as most multiple alignment approaches, *progressive alignment* is comprised by selected pairwise alignments of the input sequences. In particular, it requires a quadratic number of sequences alignments, being unfeasible for a large number of sequences. However, this cost is affordable in our scenario because very large dup-clusters are rare and heuristics can be used to limit their size, such as done by [Koppula et al., 2010]. Therefore, this approach will be the base for DUSTER, our method described in Chapter 4.

The progressive alignment method uses a greedy policy in which once a space is inserted, it can not be removed for any subsequent alignment. Thus, all spaces are preserved until the final solution. The error rate introduced by the progressive alignment at each step tends to decrease if the most similar sequences are chosen, and increase if the most divergent sequences are chosen. Thus, determining the best order for the alignments is crucial. Ideally, the most similar sequences are aligned first, leaving to the end the most divergent ones, in order to reduce the error introduced by this heuristic solution.

Unlike biological sequences, URLs within dup-clusters are not so dissimilar to each other. In fact, it is very common that many of them are almost the same. This characteristic makes it possible to devise a more efficient multiple alignment algorithm which requires only a linear number of pairwise alignments, as we will show in Chapter 5.

2.2 Web Search Engines

Millions of users access the internet to freely read and publish content on the Web. As consequence, the Web is an environment hosting a vast volume of information. This amount of content makes it difficult locating information if we decide to start searching by using only navigation links of known pages. To cope with this problem, search engines were created and represent one of the key technologies to retrieve information

in web pages.

A search engine is a web application that returns the information required by the users on their queries. It needs to (i) fetch useful web documents, parsing them efficiently, (ii) build optimized data structures for various types of queries, (iii) process them almost instantaneously to (iv) present them in a clear and attractive way to the users. To accomplish these tasks, they count on a series of components that work together: (1) A crawler that is responsible to automatically find, download and store web pages; (2) an indexer that builds the inverted index, which is the main data structure used by the search engine and represents the crawled pages; (3) and a query handler that answers user queries using the index. In the next sections, we present each of these components.

2.2.1 Web Crawler

A Web Crawler is responsible to automatically find web pages, download them, and store them in a local repository. Since there are much more web pages in the web than available resources to store them in the data centers of any search company, this component is very important as it will define which pages should be collected. Thus, it determines the general quality, and usefulness, of the page collection to be built.

Figure 2.2 shows the flow of a basic sequential crawling process. The crawler maintains a list of unvisited URLs called *the frontier*. The process starts by initializing the frontier using a set of seed URLs that may be provided by a user or any other source. The URL frontier updates the URL repository with URLs that have been crawled. Each crawling loop involves picking up the next URL to crawl from the frontier, fetching and parsing the retrieved page to extract the outgoing URLs. Finally, the unvisited URLs are added to the frontier for future crawling and the downloaded page is stored in the local repository. This process is repeated until the stopping criteria of the web crawler is met, such as a certain number of downloaded pages.

Web crawlers should be secure and robust enough to deal with the problems encountered on the Web because they are the gateway to everything that comes from it. One of these problems is the existence of duplicate information on the Web. Duplicate content does not contribute new information to search results and causes a series of problems to crawlers, such as waste of precious time, bandwidth and disk storage. Therefore, a system for detecting DUST should work next to this component, because beyond avoiding redundant access to the same content via multiple URLs, the saved time could be spent fetching new pages and, consequently, increasing the effectiveness of the crawling.

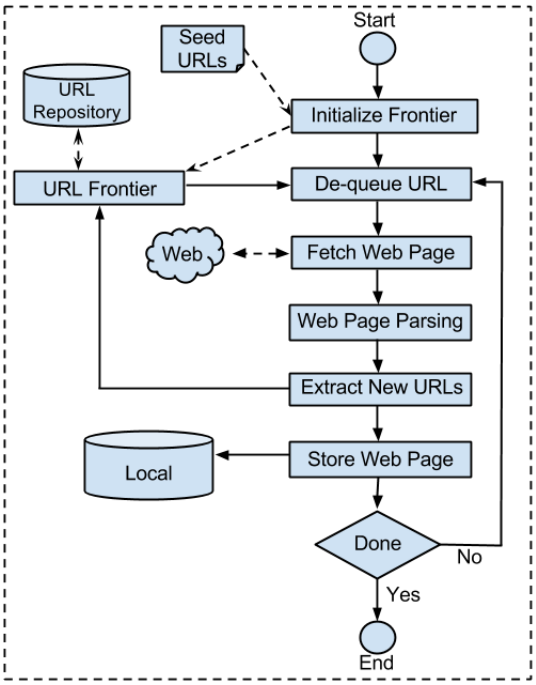

Figure 2.2: Crawling process in a typical web crawler. The crawling starts by initializing the frontier using a set of seed URLs. The URLs are de-queued from frontier for the downloading web pages. HTML parsing is performed on the downloaded page to extract new URLs. Finally, the downloaded web pages are stored in local repository.

2.2.2 Indexer

After crawling the pages, it is necessary to capture the information within them and arrange it in a format that allows very efficient retrieval. To accomplish this, search engines build an inverted index (which we refer to as *index*), which is the main data structure used by them to represent the crawled web pages. This task is accomplished by the indexer, which performs a HTML parsing on the pages, extracts their terms and adds on the data structures used during the query processing.

In large collections as Web, the large number of terms can increase the time necessary to process each query submitted to the search engine. One of the factors that contribute to the building an inefficient index is the presence of duplicate pages on the search engine local repository. Many resources, as machines and hard disks, are wasted during the indexing if search engines do not have knowledge about DUST. The elimination of this redundant information not only reduces costs, but also improves the quality and credibility of the service provided by the search engine.

2.2.3 Query Handler

When users need some information from the Web, they submit a query to search engines. This query is answered by using a component named as query handler that takes account of the constructed index to retrieve the most relevant pages according to the query. Documents retrieved are sorted by using algorithms that attempt to predict their relevance [Brin and Page, 1998; Bharat and Henzinger, 1998; Chakrabarti et al., 1998; Kleinberg, 1999].

The presence of DUST also affects the quality of answers displayed to users, since artificial information connectivity modifies the reputation of pages. Beyond directly affecting how well ranking algorithms work, duplicate pages do not contribute new information to search results and thus annoy users.

2.3 Web Concepts

In this section, we present some concepts related to the web, widely used through this thesis.

2.3.1 URL

A Uniform Resource Locator (URL) is a string that locates a unique resource on the Web. The existing architecture of WWW uses URL to address web pages. As we can see in Figure 2.3, we can identify 5 components in an URL: scheme, hostname, path, query, and fragment. Each component is separated by delimiters as slash ("/"), question mark ("?") and the number sign ("#").

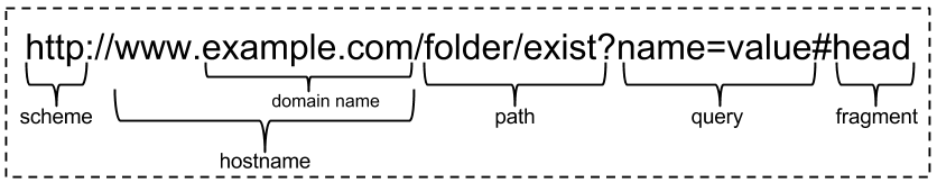

Figure 2.3: Example of the five URL components.

The scheme component contains a protocol that is used for communicating between a client and a web server. The hostname component contains the location of a web server. The hostname starts after the scheme component and finishes in first slash delimiter ("/") in the URL, if the path is not empty. The path component contains directories including a web page and a file name of the page. Each document is identified by its own path that reflects the directory structure of a web server. A directory

and a file are separated by the slash symbol ("/"). The query component contains parameter names and values. The query string starts with the question symbol ("?"). A parameter name and a parameter value are separated by the equal symbol ("="). A pair of parameter name and value is separated from each other by the ampersand symbol ("&"). The fragment component is used for indicating a particular part of a document. The fragment string starts with the sharp symbol ("#").

2.3.2 Web Site

The concept of web site is not clearly defined in the literature [da Costa Carvalho et al., 2007]. An approach adopted by works about site replica detection is using the hostname as an identifier for the site (e.g., `http://esporte.uol.com.br` and `http://economia.uol.com.br` belong to different sites). Despite this definition is not perfect, it has been adopted because it is simple to use and is a good balance between sets of pages of high granularity and low granularity. A page set of high granularity could be obtained assuming that each domain name is a different site (e.g., `http://esporte.uol.com.br` and `http://economia.uol.com.br` belong to same sites). In this thesis, we adopt the first approach, defined as follows:

Definition 3 *(WebSite) A web site is a set of pages that share the same hostname.*

2.3.3 Canonical Tags

In order to mitigate the problem of duplicate content, Google, Microsoft and Yahoo! proposed the introduction of a new HTML element which would allow webmasters to specify the "canonical" or "preferred" version of a web page. Such search engine optimization was formally described in RFC 6596, in April 2012, as Canonical Tag (or Canonical link) [Lei et al., 2010]. As an example, consider the following duplicate URLs that return the site entry page of a website:

- http://site.com
- http://site.com/
- http://www.site.com
- http://www.site.com/
- http://site.com/index.html
- http://www.site.com/index.html

To make it clear to search engines that the first URL is the preferred one, webmaster should add the canonical tag into the $< head >$ section of the others web page as follows:

```
<html>
<head>
<link rel="canonical" href="http://site.com" />
</head>
<body>
...
```

Note that webmasters explicitly assist search engines with the creation of dup-clusters when they indicate which URLs are DUST by using the canonical tags. We used this tag to create an annotated dataset composed of 3.86 million Brazilian web pages. This dataset, as well as the other dataset used in our work, is described in Section 6.1.

2.4 Regular Expressions

A regular expression (regex, for short) is a string that describes a pattern of characters, normally used to perform pattern-matching and search-and-replace functions on text. In this thesis, we use regular expressions, along with rules, to indicate how a URL could be transformed into a canonical representation. Regular expressions are also used to compose the canonical representation.

The most basic regular expression is a single literal character, that will match the first occurrence of that character on text. However, we usually need to do more than simply search for literal string pieces. In this work, in particular, we are interested in the following tasks: (a) match any character; (b) match any one of a series of patterns; (c) to treat a pattern as optional; (d) to treat a set of characters as a single symbol; (e) to indicate matches at the start and at the end of a URL; (f) to perform a match 0, 1, or more times.

The dot is one of the most commonly used metacharacters. The dot matches a single character, without caring what that character is. The vertical bar '|' character is used to match any one of a series of patterns, where the '|' character separates each pattern. The question mark makes the preceding token in the regular expression optional. Note that you can make several tokens optional by grouping them together using round brackets, and placing the question mark after the closing bracket.

Capturing groups is a way to treat multiple characters as a single unit. They are created by placing the characters to be grouped inside parentheses. The portion which matched the group will be saved in memory for later recall via backreferences. A backreference is specified as a backslash '\' followed by a digit indicating the number of the group to be recalled. As in the Perl language, you can use variables $1, $2, etc. to access the part of the string matched by the backreference.

Anchors are used to specify which part of the text should be matched. A match at the beginning of the pattern is indicated by an anchor caret ('^') while a '$' indicates a match at the end. Quantifiers specify how many instances of a character, group, or character class must be present in the input for a match to be found. The '*' quantifier matches the preceding element zero or more times. The '+' quantifier matches the preceding element one or more times. The '?' quantifier matches the preceding element zero or one time. Table 2.1 illustrates the meaning of various metacharacters by listing regexes and examples of strings that should be matched by them.

Regex	**Matches**
.at	Any three-character with "at", including "hat", "cat", and "bat".
[hc]at	"hat" and "cat".
^[hc]at	"hat" and "cat", but only at the beginning of the string.
[hc]at$	"hat" and "cat", but only at the end of the string.
cat|dog	"cat" or "dog".
[hc]?at	"hat", "cat", and "at".
[hc]*at	"hat", "cat", "hhat", "chat", "hcat", "cchchat", "at", and so on.
[hc]+at	"hat", "cat", "hhat", "chat", "hcat", "cchchat", and so on, but not "at".
([a-c])x$1x$1	"axaxa", "bxbxb" and "cxcxc".

Table 2.1: Meaning of various strings regexes.

2.5 DUST Problem

Syntactically different URLs locating similar content is a quite common phenomenon on the Web. Earlier works estimate that at least 30% of all content available on the Web is replicated content [Broder, 1997; Fetterly et al., 2003; Henzinger, 2006]. These duplicate URLs, generically known as DUST, occur due to many reasons beyond plagiarism. For instance, web servers often use aliases, symbolic links and redirections to make a website more user-friendly. In order to help their users' navigation, many web sites often have multiple DNS names that returns the same content (server aliasing) or multiple URLs to the same document on a server (URL aliasing). Moreover, some web

sites have a exact replica of their content (mirroring) in another web servers in order to either serve as backup or allow load balancing.

To make a web site more search engine friendly, webmasters use to place many different static and dynamic links to the same content. Moreover, they usually insert additional parameters (e.g., session-ids and cookie information) when they are designing the URL scheme of a web site to provide a personalized service. These parameters are used to track the user or session but they have no impact on the content of the page. Others parameters are irrelevant or superfluous in URLs, impacting only the way pages are displayed.

Others common reasons for the occurrence of duplicate content are: (i) web servers working on the Windows operating system are case insensitive and can be accessed with capitalized URL names as well as lower case names; (ii) dynamic parameters can appear in distinct positions at the URLs; (iii) different conventions are adopted (and recognized) for file extensions (e.g. htm or html); (iv) some URL elements are treated as optional, such as the trailing slash "/" and "www" in the hostname or webserver directory index.

In the following sections we define the problem of DUST, as addressed in this thesis.

2.5.1 Problem Definition

The problem addressed in this work is related to de-duplication of web pages. More specifically, it is related to the existence of syntactically different URLs linking to similar content. These URLs, generically known as DUST, usually have specific patterns that can be learned and used by URL-based de-duping methods. Along with this and next sections, we use the URLs in Table 2.2 as a running example.

The input to this problem consists of a set of URLs U (i.e. a *training set*) partitioned into groups of similar pages (referred to as *dup-cluster*) from one or more *sites*[2]. The strategy of the URL-based de-duplication methods is to learn, by mining these dup-clusters, rules that transform duplicate URLs to the same *canonical form*. In Table 2.2, $U = \{u_0, u_1, u_2, u_3, u_4, u_5\}$ is partitioned in dup-clusters C_1 and C_2. A possible canonical form of the URLs in C_1 and C_2 are given by n_1 and n_2, respectively. Note that the URLs of a same dup-cluster point to the same or similar content where URLs from different dup-clusters likely correspond to different content.

[2]A site (for instance, "britney.com.br") is an identification string that defines a realm of administrative autonomy, authority, or control on the Internet.

dup-cluster	URL
C_1	u_0 = http://britney.com.br/?id=5
	u_1 = http://britney.com.br/index.php?id=5
	u_2 = http://Britney.com.br/?id=5
	u_3 = http://www.britney.com.br/?id=5
	n_1 = http://www.britney.com.br/index.php?id=5
C_2	u_4 = http://britney.com.br/?id=7
	u_5 = http://Britney.com.br/index.php?id=7
	n_2 = http://www.britney.com.br/index.php?id=7

Table 2.2: Example of URLs to be de-duplicated and possible canonical forms. URLs of a same dup-cluster point to the same or similar content. URLs from different dup-clusters likely correspond to different content. Thus, in this example, whereas contents of u_0 and u_1 are the same, contents of u_0 and u_4 are different.

This process, called as URL normalization, identifies, at crawling time, whether two or more URLs are DUST without fetching their contents. As crawlers have resources constraints, the best methods are those that achieve larger reductions with smaller false positive rates using the minimum number of normalization rules. A normalization rule is a description of the conditions and operations necessary to transform a URL into a canonical form, as described in Definition 4.

Definition 4 *(NORMALIZATION RULE). A normalization rule can be defined as a tuple $r = (c, t)$, where c and t are regular expressions (regexes)*[3] *named context and transformation, respectively. Context c is a regex that matches a set of URLs that we refer to as the URLs affected by r. Transformation t describes which operations will be applied to the URLs affected by r to transform them into a canonical form. Let $\mathcal{S}$ be the set of sites of the URLs affected by r and $Hostname(u)$ be the site of a new URL u. We say that r is applicable to* u *if $Hostname(u) \in \mathcal{S}$.*

In Table 2.2, a rule to infer n_1 from the URLs in C_1 could be given by $r_1 = (c_1, t_1)$ where regexes c_1 and t_1 are given as:

c_1 = ^⋆://(www.)?(Britney|britney).⋆.⋆/(index.php)?\?⋆=⋆$
t_1 = $1://www.britney.$2.$3/index.php?$4=$5

In these regexes, any symbols other than '(', '|', ')', '$$i$$', '?' and '⋆' are literals. In order to use last question mark '?' as a literal in c_1, we escape it with a backslash

[3]In this work, regular expressions describe string patterns to be matched as well as inserted, deleted or modified.

'\'. The substrings between round brackets ("www." and "index.php") with a question mark after the closing bracket are optionals. The list of substrings between parentheses, separated by '|', are alternative options. Symbol '$\star$' indicates any sequence of characters. Symbol '$i' in t_1 is the i-th sequence matched by a '$\star$' symbol in c_1.[4]

Thus, regex c_1 will match URLs starting with any sequence of characters, followed by "://" and, optionally, "www.". The following sequence in the URL may be "Britney" or "britney", followed by two sequences of characters, both starting with dot. The sequence continues with a slash, optionally followed by "index.php". The last part of the URL starts with '?' and is followed by two sequences of characters separated by '='. Regex t_1 indicates that the canonical form of the URL matched by c_1 must start with the first matched substring, followed by "://www.britney.", followed by the second matched substring, and so on.

During normalization, when two URLs are converted to the same canonical form, the pair is called an *instance* of the rule. The set of all instances is referred to as the *support* of the rule. These and other key concepts are formally defined as follows[5]:

Definition 5 *(INSTANCE). Given a set of URLs U, an instance of a rule r is a unordered URL pair (x, y), $x \in U$, $y \in U$, which is transformed to the same canonical form after applying* r.

Definition 6 *(SUPPORT). The support of rule r, given a set of URLs U, denoted $supp(r, U)$, is the set of all instances of r, given U.*

The support of rule r_1, previously described, considering all the URLs in C_1 and C_2, is given by $S_{r_1} = \{(u_0, u_1), (u_0, u_2), (u_0, u_3), (u_1, u_2), (u_1, u_3), (u_2, u_3), (u_4, u_5)\}$.

Definition 7 *(FALSE-POSITIVE RATE). Let $FP(r, U)$ be the number of instances (x, y) in $supp(r, U)$ such that x and y* are not *DUST. The false-positive rate of r is denoted by $fpr(r, U) = FP(r, U)/|supp(r, U)|$.*

The primary goal of a URL-based de-duping method is, given a set of duplicate clusters $\mathcal{C}$, to find the set of rules that have high support and low false-positive rate, when considering the URLs in $\mathcal{C}$. The problem of detecting DUST can now be formally stated as follows:

[4]We adopted this syntax in this section to avoid a cumbersome notation. However, note that the symbol '$\star$' should be replaced by an appropriate regular expression such as $[a-zA-Z]+$ (alphabetic), $[0-9]+$ (numeric) etc.

[5]We adopt in this thesis the same definitions presented in [Bar-Yossef et al., 2006; Dasgupta et al., 2008]

Problem 1 *Given thresholds minSup, $fpr_{max} \geq 0$, and a set of URLs U partitioned in a set of dup-clusters $\mathcal{C}$, generate the set of rules R such that $\forall r \in R$, $|supp(r, U)| \geq minSup$ and $fpr(r, U) \leq fpr_{max}$.*

2.5.2 Evaluation Metrics

In this section we present the key metrics used to evaluate how effective is a set of rules regarding DUST detection. We used these metrics for measuring our techniques and compare them with the state-of-the-art approaches.

To estimate the percentage of URLs we can avoid fetching by using a set of normalization rules, we adopt the *Compression Rate* metric proposed by [Lei et al., 2010]. In particular, it measures the reduction ratio of the number of URLs after applying a set of rules R. It is defined as:

$$Compression\ Rate = \frac{N_{orig} - N_{norm}}{N_{orig}} \tag{2.2}$$

where N_{orig} and N_{norm} are the number of URLs before and after the normalization, respectively. This metric is also known as *Reduction Ratio* [Dasgupta et al., 2008] or *Discovered Redundancy* [Bar-Yossef et al., 2006].

The amount of DUST in a dataset is defined as the difference between the number of unique URLs and the number of dup-clusters. Thus, in order to compare the amount of DUST achieved by an evaluated method with the total amount of DUST in a dataset, we adapted the *Coverage* metric proposed by [Bar-Yossef et al., 2006]. It is defined as:

$$Coverage = 1 - \frac{(N_{norm} - C_{norm})}{(N_{orig} - C_{orig})} \tag{2.3}$$

where C_{orig} and C_{norm} are the number of dup-clusters before and after the normalization, respectively. This metric is also known as *dup-reduction rate* [Lei et al., 2010].

In normalization, it is possible that two non-duplicate URLs are incorrectly converted to the same canonical form. When two duplicate URLs are correctly converted to the same canonical form, such pair is called as *correct instance*. To estimate how accurate is the normalization process by applying a set of rules R, we use *Normalization Precision* metric proposed by [Koppula et al., 2010]. It is definied as:

$$Normalization\ Precision = \frac{N_{correct}}{N_{total}} \tag{2.4}$$

where $N_{correct}$ and N_{total} are the number of correct instances and total number of instances in the normalization, respectively.

The last measure we adopt in this work is the Average Reduction per Rule (AR/R) [Koppula et al., 2010]. This metric assesses how general is a set of rules (R). It is defined as:

$$AvgReductionPerRule = \frac{N_{orig} * Compression\ Rate}{|R|} \tag{2.5}$$

where $|R|$ is the size of the set of rules R.

2.6 Summary

In this chapter, we presented background information necessary to understand the remaining of this thesis. This included a description of search engines and how they are affected by DUST, sequence alignment, definitions for common web concepts such as URLs, regular expressions and the DUST problem. We also introduced notation to be used from now on. In next chapter, we will describe the works in literature most related to ours.

Chapter 3

Related Work

In this chapter, we introduce previous work in literature which addressed the problem of DUST detection. This problem was first approached as an issue regarding URL normalization. As such, it was first addressed by Internet standardization bodies. Along the time, with the proliferation of duplicate content, other approaches arised. In this section, we summarize such research efforts.

3.1 URL Normalization

The fact that pages do not have unique identifiers creates problems in almost every large scale software that deals with the web. Systems that deal with web content should be aware of the existence of duplicate information. Duplicates occur when two or more syntactically different URLs locate the same content. In order to obtain some level of normalization, standard Internet bodies defined a set of steps (or *universal rules* [Bar-Yossef et al., 2006]), known as Standard URL Normalization (SUN), to transform duplicate URLs into a canonical form. This process is named as *URL normalization*.

SUN focus on URL structure syntax and was designed to minimize false negative while strictly avoiding false positives. In other words, it never transforms non-DUST into a syntactically identical string.

Some of the steps used in typical URL normalization procedures, as described by [Berners-Lee et al., 1998], are presented in the following paragraphes.

Percent-Encoding Normalization. All unreserved characters can be encoded into a three-digit string. Percent symbol (%) should be located at the first position, and the last two digits are a hexadecimal number representing an ASCII code of the character under consideration (e.g. %2D for hyphen and %5F for underscore). The idea behind this normalization is perform URL encoding for theses commonly used charac-

ters. This would prevent the crawler from treating `http://example.com/~DUSTER` as a different URL from `http://example.com/%7DUSTER`. Hexadecimal digits within a percent-encoding triplet are case-insensitive and should be normalized to use uppercase letters for the digits A-F.

Case Normalization. Scheme and hostname components are case insensitive. During the normalization, all the letters in these components are changed into lowercase letters. This would prevent the crawler from treating `HTTP://example.com` as a different URL from `http://EXAMPLE.COM`.

Path Segment Normalization. Path components such as '.' ou '..' should be removed. For instance, `www.site.com/dir/../index.html` is reduced to `www.site.com/index.html`.

Remove Default Port Number. An URL with a default port number (80 for the HTTP protocol) and a URL without the port number represent the same page. For instance, http://example.com:80/ and http://example.com/ represent the same page. During Standard URL Normalization, the default port number is truncated from a URL.

Truncate the Fragment of URL. Fragments are used to reference a part of a page. However, the crawling of URLs that differ only on the anchors would result in repeated downloads of the same page. For instance, `http://example.com/page.html#chap1` and `http://example.com/page.html` represent the same page. During the normalization, the fragment in the URL should be truncated.

Add trailing "/" after the hostname. When path name is not present in the URL, it must be given as "/" when used as a request for a resource. This normalization rules prevents that URLs, such as `www.example.com` and `www.example.com/`, originate duplicates. During the normalization, if a path string is null then the path string is transformed into "/".

Several works were proposed to extend SUN aiming to reduce false negatives while allowing false positives at a limited level. [Lee et al., 2005] extended SUN by adding three new heuristic steps: (i) change the path component into lower case; (ii) to eliminate the last slash symbol component at the non-empty path component; and (iii) eliminate default pages (e.g. default.html, default.htm, index.html and index.htm). Beside theses three steps, they also proposed two evaluation metrics, *redundancy rate* and *coverage loss*. Their results indicate that their proposed steps were able to reduce duplicate URLs while allowing limited false positives. [Kim et al., 2006] did not propose any new steps but they presented a set of metrics to evaluate SUN: *URL consistency*, *URL applying rate*, *URL reduction rate* and *true positive rate*.

As observed by [Dasgupta et al., 2008], DUST is typically not random but rather

stems from *structured transformations* on URLs string. As result, simple rules for URL normalization do not cover most of the examples of DUST on the web and a proper understanding of the observed URL transformations would be necessary to generate more effective rules to detect DUST. Simple URL normalization is based on heuristics which will fail in many situations. This way, the research problem moved from simple URL normalization towards the more complex problem of DUST detection.

Currently, such research is classified in two main families of methods, according if they take into consideration the page content: content-based and URL-based. These approaches are described in the next sections.

3.2 Content-based DUST Detection

In content-based DUST detection, the similarity of two URLs is determined by comparing their contents. Thus, to infer if two distinct URLs correspond to duplicates, or near duplicates, it is necessary to fetch and inspect the whole content of their corresponding pages.

For instance, Soon *et al.* proposed to enhance SUN by incorporating semantically meaningful metadata of web pages (or URL signatures) [Soon and Lee, 2008a,b, 2010; Soon et al., 2012]. The metadata used are the body texts and page size of the web pages extracted during HTML parsing. They construct its URL signature by hashing or fingerprinting the body text using Message-Digest algorithm 5 [Rivest, 1992]. URLs which share identical signatures are considered duplicates in their scheme. Their experimental results show that their proposed method helps to further reduce redundant Web information in comparison with SUN.

Many other methods have been proposed in literature that explore different content syntactic and semantic evidence, such as shingles, text signatures, pair-wise similarities, sentence-wise similarities, and semantic graphs [Mao et al., 2011; Lei et al., 2010; Alsulami et al., 2012]. As such methods imply in a waste of resources, several URL-based methods have been proposed to determine duplicated URLs without examining the associated contents. This is the case of the methods we propose in this thesis. Thus, we describe such general approach in the next section. Regarding content-based methods, for a comprehensive review of the literature, we refer the reader to the surveys by [Kumar and Govindarajulu, 2009] and by [Alsulami et al., 2012].

3.3 URL-based DUST Detection

We now focus on URL-based methods including the ones that, as far as we know, reported the best results in the literature.

The first URL-based method proposed was DustBuster [Bar-Yossef et al., 2006]. In their work, the authors addressed the DUST detection problem as a problem of finding normalization rules able to transform a given URL to another likely to have similar content. The rules consist of substring substitutions learned from crawl logs or web logs. Rules are selected if (a) they have large support, (b) they do not come from large groups and (c) URLs matched by them have similar sketches or compatible sizes in the training log. Redundant rules are eliminated based on their support information. By evaluating their method in four websites, the authors found that up to 90% of the top ten rules were valid, 47% of the duplicate URLs were recognized and the crawl was reduced by up to 26%.

Since substitution rules were not able to capture many common duplicate URL transformations on the web, Dasgupta et. al. presented a new formalization of URL rewrite rules [Dasgupta et al., 2008]. The new formulation was expressive enough to capture all previous substitution rules as well as more general patterns, such as the presence of irrelevant substrings, complex URL token transpositions and session-id parameters. The authors use some heuristics to generalize the generated rules. In particular, they attempt to infer the false-positive rate of the rules in order to select the most precise ones. To accomplish this, they verify if the set of values that a certain URL component assumes is greater than a threshold value N, a heuristic which they call *fanout-N*. Their best results were obtained with $N = 10$. In this work, we refer to this method as $R_{\textit{fanout-10}}$. By applying the set of rules found by $R_{\textit{fanout-10}}$ to a number of large scale experiments on real data, the authors were able to reduce the number of duplicate URLs by 60%, whereas only substitution rules achieved 22% reduction.

The authors in [Agarwal et al., 2009] extended the work in [Dasgupta et al., 2008] to make their use feasible at web scale. They observed that the quadratic complexity of the rule extraction performed in [Dasgupta et al., 2008] is prohibitive for very large dup-clusters. Thus, they proposed a method for deriving rules from samples of URLs. In addition, they used a decision tree algorithm to learn a small number of higher precision rules to minimize the number of rules deployed to the crawler. The authors evaluated their method in a set of about 8 million URLs, achieving a de-duplication reduction rate of about 42% using the top 9% of precise rules (precision level above 80%).

In a subsequent paper, [Koppula et al., 2010] implemented their algorithm using

a distributed framework and extended the URL and rule representations to include two additional patterns: tokens inside path components and more complex irrelevant components. The authors used a very simple alignment heuristic to deal with irrelevant components. To show the scalability of their method, they evaluated it with 3 billion URLs. By comparing their method with $R_{fanout\text{-}10}$, they achieved two times more reduction using 56% of the rules. Unfortunately, the method proposed by [Koppula et al., 2010] is not publicly available and was not described with enough detail to be implemented.

The methods previously described use a bottom-up approach in which normalization rules are learned by inducing local duplicate pairs to more general forms. The main drawback of these strategies is the difficulty to induce general rules starting from pairs of duplicate URLs. The method in [Dasgupta et al., 2008], in particular, also adopts an additional requirement that the inputs should differ from each other at only one single token (which the authors in [Dasgupta et al., 2008] refer to as Single Key Requirement). Such issues usually break the rule-inducing algorithm somewhere avoiding the derivation of more general rules, as we can see in Figure 3.1. In this example, there are 8 URLs referring to duplicate content, where the values of tokens t_3 and t_5 should be normalized (or generalized) to '*'. By analysing pairs of rules, we note from pairs (u_5, u_6) and (u_7, u_8) that t_5 could be generalized. Note, however, that if some URLs are absent at the time of rule generation (eg: u_6), the process could not derive rule r_3. In addition to this limitation, most bottom-up strategies discard a lot of training examples that do not satisfy the Single Key Requirement. As we can see in Figure 3.2, URLs u_1, u_2 and u_7 are not leveraged in the learning process and, as a result, the learned rule cannot obtain the maximum compression expected for this and other clusters that follow the same pattern.

Due to these problems, authors in [Lei et al., 2010] argue that these previous approaches are computationally inefficient and very sensitive to noise. Thus, they propose a top-down approach in which statistics from the entire training data are calculated to help in the generalization of the rules, and a URL pattern tree (UPT) is built from clusters of duplicate URLs for a targeted website. According to the authors, using the UPT contributes to (a) a robust and reliable rule extraction, (b) accelerated learning since rules are directly summarized in UPT nodes and (c) the selection of more general rules due to the removal of conflicts and redundancy. They evaluated their approach in a collection with 70 million URLs and showed that their method was able to outperform $R_{fanout\text{-}10}$ achieving about twice the reduction using 46% of the rules and consuming half of the learning time. In this work, we refer to it as R_{tree}.

By aligning all URLs we can obtain a unique sequence representing the entire dup-

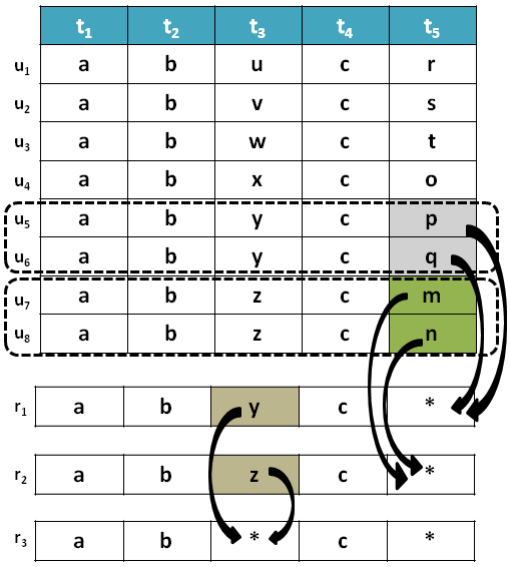

Figure 3.1: Examples of rules induced by pairwise bottom-up strategies. Example adapted from [Lei et al., 2010].

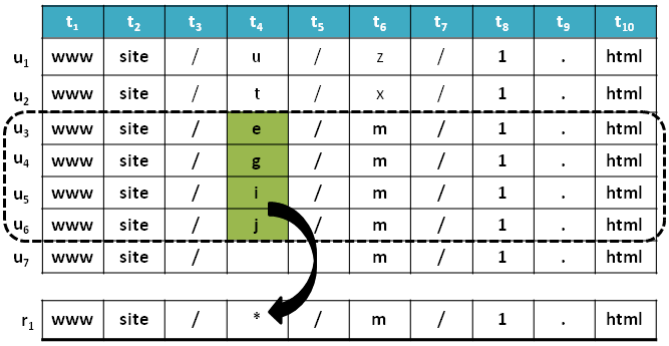

Figure 3.2: Example of rule induced by pairwise bottom-up strategy, in which URLs u_1, u_2 and u_7 are discarded from the learning process.

cluster, referred to as *consensus sequence*[1]. Thus, we can derive a general rule from the consensus sequence, even if some URLs are absent. Take, for instance, the scenarios described in Figure 3.1 and Figure 3.2. In Figure 3.1, if u_6 is absent, the consensus sequence obtained from the cluster is $\mathcal{CS}_1 = \langle a, b, \{u, v, w, x, y, z\}, c, \{r, s, t, o, p, m, n\}\rangle$. It is easy to derive r_3 from $\mathcal{CS}_1$ since it is clear that tokens t_3 and t_5 should be normalized as '*'. In Figure 3.2, the consensus sequence obtained from the cluster is $\mathcal{CS}_2 = \langle www, site, /, \{u, t, e, g, i, j, \lambda\}, \{/, \lambda\}, \{m, x, z\}, /, 1, ., html\rangle$, where λ is a gap. As before, the derivation of a rule involving all URLs from $\mathcal{CS}_2$ is straightforward since it is clear that tokens t_4 and t_5 are irrelevant, and t_4 should be normalized as '*'. Based on this idea, we will propose in this thesis new algorithms for the DUST detection problem. To

[1]A formal definition of consensus sequence is provided in section 4.1.2.

evaluate our proposed methods, we will compare it with methods $R_{fanout\text{-}10}$ and R_{tree}. $R_{fanout\text{-}10}$ is a traditional method used as benchmark in most of the previous works in literature whereas R_{tree}, as far as we know, is the one which reported the best results in literature.

3.4 Summary

In this chapter, we introduced previous work in literature, starting with approaches proposed to the problem of URL normalization. As observed, the research in this area evolved to address the more complex problem of DUST detection. Although many methods addressing such problem use content based techniques, several works have focused only on the URL string, as an attempt to avoid wasting resources fetching content. This is the case of the methods we propose in this thesis. Compared to other URL-based methods, our approach is original as it is the first based on a multi-sequence alignment. In the next chapters, we will describe three variants of our algorithm.

Chapter 4

DUSTER

DUST Detection with a Quadratic Multi-Sequence Alignment Algorithm

In this chapter we describe our first algorithm to detect DUST, DUSTER. It takes advantage of a traditional multi-sequence alignment strategy, performed over dup-clusters given as input. We start by describing how URLs are aligned. Then, each phase of the algorithm is presented, that is, the generation of candidate rules, their validation, and finally, the URL normalization. We also describe important intermediate steps as the classification of URL tokens and the conversion of the aligner output into transformation rules.

4.1 URL Alignment

In order to obtain a smaller and more general set of normalization rules, our method takes advantage of multiple sequence alignment. The strategy is to create the so called *consensus sequence* for each dup-cluster in the training set and extract the rules from them. We perform this task by aligning the URLs in each cluster and then generating the consensus sequences as a result of this alignment. In the following subsections, we show how to align two or more URLs and how to generate a consensus sequence for these dup-clusters. Before presenting our URL alignment approach, we first show how we represent URLs.

4.1.1 URL Tokenization

Unlike previous works that treat URLs as strings generated according to W3C grammar[1], we adopt a simpler representation. We consider a URL as a sequence of three types of tokens (URL tokens), as described by the EBNF-based[2] grammar G described below:

⟨*URL*⟩ ::= ⟨*token*⟩ { ⟨*token*⟩ }

⟨*token*⟩ ::= ⟨*alphabetic*⟩ | ⟨*number*⟩ | ⟨*punctuation*⟩

⟨*alphabetic*⟩ ::= ⟨*alpha*⟩ { ⟨*alpha*⟩ }

⟨*alpha*⟩ ::= '`a`'..'`z`' | '`A`'..'`Z`'

⟨*number*⟩ ::= ⟨*digit*⟩ { ⟨*digit*⟩ }

⟨*digit*⟩ ::= '`0`'..'`9`'

⟨*punctuation*⟩ = All remaining characters such as '`/`', '`:`', and '`.`'

Each URL to be aligned is initially parsed according to grammar G. This process, referred to as *tokenization*, decomposes the URL into a sequence of URL tokens. To facilitate URL alignment, each token extracted from a URL is represented as a singleton set.

For example, URL u = `http://ex.com/1.htm` is represented by the following sequence of 11 token sets:
S = ⟨{*http*}, {:}, {/}, {/}, {*ex*}, {.}, {*com*}, {/}, {1}, {.}, {*htm*}⟩

4.1.2 Pair-wise URL Alignment

The output of our alignment process is a sequence of sets, referred to as the *consensus sequence*, which is a way of representing the result of the alignment. The consensus sequence of n sequences is created by the union of the tokens in the corresponding positions of the n aligned sequences.

To help readers better understand the complete pair-wise URL alignment process and how we generate a consensus sequence, we illustrate it with an example.

[1] `http://www.w3.org/Addressing/URL/5_BNF.html`

[2] An EBNF (Extended Backus-Naur Form) specification is a set of derivation rules, written as <symbol> ::= *expression*, where <symbol> is a nonterminal, and *expression* consists of one or more sequences of symbols. Alternatives are separated by the symbol '|' and 0 or more repetitions of an expression are indicated by braces. Symbols that never appear on the left side are terminals. The '::=' means that the symbol on the left must be replaced with the expression on the right. Finally, the expression $s_1..s_2$ indicates a sequence of alternative symbols in the interval starting at symbol s_1 and finishing at symbol s_2.

To obtain a consensus sequence for two URLs u_i = `www.IRS.gov/foia/index.html` and u_j = `www.irs.ustreas.gov/foia`, we first obtain the token set sequences X and Y, associated with u_i and u_j respectively, with m and n tokens. X and Y are given by:

$\mathrm{X} = \langle\{www\}, \{.\}, \{IRS\}, \{.\}, \{gov\}, \{/\}, \{foia\}, \{/\}, \{index\}, \{.\}, \{html\}\rangle$.
$\mathrm{Y} = \langle\{www\}, \{.\}, \{irs\}, \{.\}, \{ustreas\}, \{.\}, \{gov\}, \{/\}, \{foia\}\rangle$.

Tokenized sequences X and Y are then aligned by inserting gaps, either into or at the ends of them. To determine where gaps should be inserted, matrix $S_{i,j}$ in Equation 2.1 has to be calculated. To accomplish this, a score function sf should be defined to measure the distance between the URL token sets. The scoring function we adopt, given by Equation 4.1, is the *Jaccard similarity coefficient* [Theobald et al., 2008] which is commonly used to measure the overlap between two sets. For two sets, it is denoted as the cardinality of their intersection divided by the cardinality of their union.

$$sf(X_i, Y_j) = \begin{cases} \frac{|X_i \cap Y_j|}{|X_i \cup Y_j|} & \text{if } \exists(x_i, y_j) \in X_i \times Y_j | \tau(x_i) = \tau(y_j) \\ -1 & \text{otherwise} \end{cases} \tag{4.1}$$

where $\tau : \mathcal{T} \rightarrow \{a, n, p\}$ is a function which maps a token set to its token type, $\mathcal{T}$ is the token space and $\{a, n, p\}$ are the token types (a for alphabetic, n for numeric, and p for punctuation).

The resulting scoring/traceback matrix is shown in Figure 4.1, where URLs `www.IRS.gov/foia/index.html` and `www.irs.ustreas.gov/foia` are aligned.

S_{ij}	u_i	{www}	{.}	{irs}	{.}	{ustreas}	{.}	{gov}	{/}	{foia}
u_j	0	0	0	0	0	0	0	0	0	0
{www}	0	↖1	←1	←1	←1	←1	←1	←1	←1	←1
{.}	0	↑1	↖2	←2	↖2	←2	↖2	←2	←2	←2
{IRS}	0	↑1	↑2	↖3	←3	←3	←3	←3	←3	←3
{.}	0	↑1	↖2	↑3	↖4	←4	↖4	←4	←4	←4
{gov}	0	↑1	↑2	↑3	↑4	↖4	←4	↖5	←5	←5
{/}	0	↑1	↑2	↑3	↑4	←4	↖4	↑5	↖6	←6
{foia}	0	↑1	↑2	↑3	↑4	↖4	←4	↑5	↑6	↖7
{/}	0	↑1	↑2	↑3	↑4	←4	↖4	↑5	↖6	↑7
{index}	0	↑1	↑2	↑3	↑4	↖4	←4	↑5	↑6	↑7
{.}	0	↑1	↖2	↑3	↖4	←4	↖5	←5	↑6	↑7
{html}	0	↑1	↑2	↑3	↑4	↖4	↑5	↖5	↑6	↑7

Figure 4.1: Scoring/traceback matrix for the duplicate URLs u_i = `www.IRS.gov/foia/index.html` and u_j = `www.irs.ustreas.gov/foia`. Each leftward arrow in the path indicates that a gap has to be inserted into X while an upward arrow indicates a gap to be inserted into Y. A diagonal arrow indicates the symbols from the two sequences are aligned and no gap should be inserted. Larger scores indicate a better alignment.

At the end of the alignment, X and Y are transformed into sequences X' and Y' as showed in in Table 4.1. Note that besides the four gaps (λ indicates a gap) that were inserted at the end of the URL X, it was necessary to insert two gaps into the URL Y to align the similar tokens. X' and Y' have the same length so that every token is either a unique token or a gap in the other sequence.

X'	www	.	irs	.	ustreas	.	gov	/	foia	λ	λ	λ	λ
Y'	www	.	IRS	λ	λ	.	gov	/	foia	/	index	.	html
	1	2	3	4	5	6	7	8	9	10	11	12	13

Table 4.1: An illustration of the alignment between the duplicates URLs X = `www.irs.ustreas.gov/foia` and Y = `www.IRS.gov/foia/index.html`.

The final consensus sequence $\mathcal{CS}_{ij}$ for URLs u_i and u_j from Figure 4.1 is given by uniting the token sets of X' and Y' in Table 4.1.

$$\mathcal{CS}_{ij} = \langle \{www\}, \{.\}, \{irs, IRS\}, \{., \lambda\}, \{ustreas, \lambda\}, \{.\}, \{gov\}, \{/\}, \{foia\}, \{/, \lambda\}, \{index, \lambda\}, \{., \lambda\}, \{html, \lambda\} \rangle$$

Note that no token set T_i of $\mathcal{CS}_{ij}$ will be $\{\lambda\}$ since gaps are not aligned with other gaps by the pairwise algorithm. We formally define a consensus sequence as follows:

Definition 8 *(Consensus Sequence) Let* $\{X_1, X_2, ..., X_n\}$ *be a set of* n *tokenized and aligned URLs, such that* $|X_1| = |X_2| = ... = |X_n| = k$*, where* $|X_i|$ *is the number of tokens in* X_i*. Let* t_{Xi} *be the token of URL* X *at position* i*. A consensus sequence is a sequence of* k *token sets* $\langle T_1, ..., T_k \rangle$ *such that* $T_i = \cup_{\forall X}\{t_{Xi}\}$*.*

URL Alignment Complexity. When computing the value for a specific cell (i, j), only cells $(i-1, j-1)$, $(i, j-1)$ and $(i-1, j)$ are examined, along with the tokens within the token sets T_i and T_j. To fill a cell (i, j) it is necessary to estimate the overlap between token sets T_i and T_j, which takes $O(min(|T_i|, |T_j|))$, i.e., is linear in the number of elements of the token set with smaller cardinality. Thus, the dynamic programing table for computing the pair-wise URL alignment of two token set sequences X and Y can be computed in $O(|X||Y|J)$, where J is the cost to calculate the *Jaccard similarity coefficient* for each $|X| \times |Y|$ iteration.

4.1.3 Multiple URL Alignment

In this section, we show how to align a dup-cluster larger than two URLs. To accomplish this, we use the progressive alignment strategy presented in Feng and Doolittle [1987] which aligns the two most similar sequences at each iteration and infers a new

token set sequence from them. This process is repeated until all sequences have been aligned, resulting in the final multiple alignment (i.e. a consensus sequence).

Algorithm 1 MultipleURLAligment ($\mathcal{C}$)

Input: *A* dup-cluster $\mathcal{C} = \{u_1, ..., u_n\}$ with n duplicate URLs
Output: A tuple $\pi = (consensus, \mathcal{S})$.

1: Let $\mathcal{Q}$ be a priority queue in which tuples $\sigma = (X, Y, consensus_{XY}, scoring)$ are sorted in descending order according to the alignment *scoring*.
2: $\mathcal{S} = \emptyset$; $S_{equences} = \emptyset$; $A_{ligned} = \emptyset$;
3: **for all** pairs of distinct URLs u_1, u_2 in $\mathcal{C}$ **do**
4: $\mathcal{S} = \mathcal{S} \cup \{hostname(u1)\} \cup \{hostname(u2)\}$
5: $X = tokenize(u1)$
6: $Y = tokenize(u2)$
7: $S_{equences} = S_{equences} \cup \{X\} \cup \{Y\}$
8: $\sigma = pairWiseURLAlignment(X, Y)$
9: add σ to $\mathcal{Q}$
10: **end for**
11: **while** $\mathcal{Q}$ is not empty **do**
12: Pop the first tuple σ from $\mathcal{Q}$
13: **if** $\sigma.X \notin A_{ligned}$ and $\sigma.Y \notin A_{ligned}$ **then**
14: $A_{ligned} = A_{ligned} \cup \{\sigma.X\} \cup \{\sigma.Y\}$
15: $S_{equences} = S_{equences} - A_{ligned}$
16: **for all** sequences s in $S_{equences}$ **do**
17: $\epsilon = pairWiseURLAlignment(\sigma.consensus, s)$
18: add ϵ to $\mathcal{Q}$
19: **end for**
20: $S_{equences} = S_{equences} \cup \{\sigma.consensus\}$
21: **end if**
22: **end while**
23: Let s be the unique sequence in $S_{equences}$
24: **return** $\pi = (s, \mathcal{S})$

As previously mentioned in Section 2.1.2, the order in which we select examples to be aligned could influence the final result. The policy used to select the most similar sequences is based in Feng and Doolittle [1987] and is described in Algorithm 1.

First, a priority queue $\mathcal{Q}$ is created from all pairs of URLs in $\mathcal{C}$ (Lines 3-10). Each tuple in $\mathcal{Q}$ is composed of two sequences (X and Y), a consensus sequence obtained by aligning them, and an alignment score. Note that we extract the hostnames of all URLs within $\mathcal{C}$ and we keep them in the set $\mathcal{S}$. This set indicates the sites where the rule generated from $\mathcal{C}$ can be applied. By using $\mathcal{Q}$, it is possible to find the tuple σ with the most similar pair of URLs (Line 12). These two sequences are removed from the set of sequences to be aligned and added to the set of aligned sequences (Lines

14-15). We then align the consensus sequence in σ with all the remaining sequences to be aligned (Lines 16-19). At the end of this process, we have aligned all URLs in $\mathcal{C}$ and reduced them to a single sequence of token sets.

Analysis of the Algorithm. In this analysis, we consider the number of alignments between two sequences as the relevant cost measure to determine the running time. Thus, we count the number of times lines 8 and 17 are performed with n sequences provided as input. First, each pair of sequences has to be aligned (lines 3-10). This process requires $\frac{n(n-1)}{2}$ alignments involving all pairs provided as input. Then, progressive alignments of the sequences are carried out until the final multiple alignment is done (lines 11-22). This process takes $(n-1)$ iterations because is starts with n sequences and the two most similar are aligned at each iteration i. In addition, the algorithm requires further $(m-i-1)$ alignments between the new sequence and the others. Thus, in total, $\frac{(n^2-3n+2)}{2}$ alignments are performed. The overall cost function is given by $f(n) = n^2 - 2n + 1$. Therefore, the complexity of the algorithm is $O(P \times n^2)$, where P is the cost of the method *Pair-wise URL alignment*.

Note that, in practice, this cost is not prohibitive since very large dup-clusters are rare and heuristics can be used to limit their size, such as done by Koppula et al. [2010].

4.2 DUSTER Algorithm

In this section, we describe in detail our solution to avoid the presence of DUST in search engines. Figure 4.2 depicts the framework of our algorithm, DUSTER.

As we can see in this figure, once a new set of URLs is crawled, it is merged with the already known URLs to form a new set of known URLs. During crawling, the crawler is also able to identify examples of DUST by following canonical tags. As a result, a new set of known DUST is also available. This set can be still enriched by processes such as those based on content signature, followed by manual inspection. Given the final set of known DUST, DUSTER can use it to find and validate rules, by split it in training and validate sets. The resulting rules are then used to normalize the known URLs yielding a new (and reduced) set of URLs to be crawled. By using this set and the set of DUST rules, the crawler can gather additional URLs, closing the cycle.

The two main phases of DUSTER are the generation of candidate rules, where a multi-sequence alignment algorithm generates candidate rules from dup-clusters, and the rules validation, where DUSTER filters out candidates rules according to their

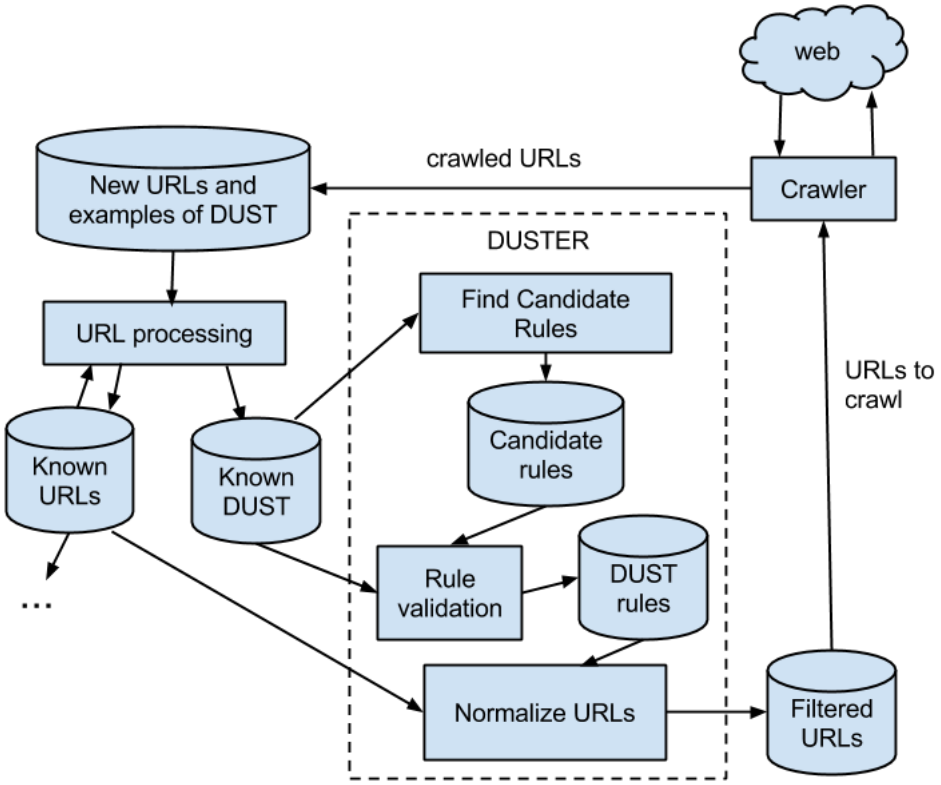

Figure 4.2: DUSTER framework. The dotted box highlights the components of the DUSTER algorithm.

performance in a validation set. These two phases are described in the next sections.

4.2.1 Candidate Rules Generation

This phase consists of two steps: (1) For each dup-cluster in the training set, we obtain a rule. To accomplish this, we first align a set of K URLs randomly selected within the cluster in order to obtain a consensus sequence to represent these URLs. A rule is then extracted from this consensus sequence. Note that if the cluster has less than K URLs, all of them are involved in the alignment process. We adopt this sampling strategy because, in general, it is not necessary to inspect all the training examples (i.e URLs) to get a general rule, as many patterns are observed with just a few number of examples. In that way, we also avoid the alignment of large dup-clusters which could be very expensive. Also note that, since the cluster size distribution is very skewed, our sampling strategy affects only the larger dup-clusters, that is, the minority of the dup-clusters. (2) From the generated rules, we discard the ones with frequency less than min_{freq}. Thus, very specific rules, with few occurrences in the training set, are

discarded.

Algorithm 2 GenerateCandidateRules ($\mathcal{TS}$)

Input: *T*raining Set $\mathcal{TS} = \{c_1, ..., c_n\}$ with n dup-clusters
Output: Set of m candidate rules $\mathcal{CR} = \{r_1, ..., r_m\}$
1: Create table $\mathcal{RT}$ ($context$, $transformation$, $\mathcal{S}$)
2: Create table $\mathcal{CRT}$ ($context$, $transformation$, $\mathcal{S}$)
3: **for all** dup-cluters $c_i \in \mathcal{TS}$ **do**
4: $\mathcal{A}$ = selectRandomlyURLs(c_i,K)
5: π = MultipleURLAligment($\mathcal{A}$)
6: r = generateRule (π.consensus)
7: add ($r.context$, $r.transformation$, $\pi.\mathcal{S}$) to $\mathcal{RT}$
8: **end for**
9: group tuples in $\mathcal{RT}$ into buckets by ($context$, $transformation$)
10: **for all** buckets B **do**
11: **if** ($|B| >= min_{freq}$) **then**
12: $\mathcal{S} = \emptyset$;
13: **for all** tuples $t \in B$ **do**
14: $\mathcal{S} = \mathcal{S} \cup t.\mathcal{S}$
15: **end for**
16: α = the first tuple in B
17: add ($\alpha.context$, $\alpha.transformation$, $\mathcal{S}$) to $\mathcal{CRT}$
18: **end if**
19: **end for**
20: **return** a set $\mathcal{CR}$ of rules created from $\mathcal{CRT}$

Algorithm 5 presents *GenerateCandidateRules* which takes a set of dup-clusters as input and generates a set of candidate rules as output. In lines 1 and 2, two tables are created: $\mathcal{RT}$ (Rules Table) which stores the candidate rules generated for each cluster, and $\mathcal{CRT}$ (Candidate Rules Table) which stores rules which exceed the frequency threshold min_{freq}. In lines 4-5, K URLs are randomly selected from c_i and aligned by algorithm *MultipleURLAligment* (see Algorithm 1). In lines 6 to 7, candidate rules are generated and added to table $\mathcal{CRT}$. The conversion of a consensus sequence to a normalization rule is described in details in Section 4.2.1.2. In Line 9, the rules are grouped into buckets according to their context and transformation (we sort table RT by the first and second attributes). In Lines 10-20, the algorithm enumerates all distinct rules generated by the first step. Note that two rules are considered the same if they have the same context and transformation. In Line 11, if the bucket size exceeds min_{freq}, the rule is considered as a *candidate*. In Lines 13-15, the rules with same context and transformation are unified by the union of the sets $\mathcal{S}$ of each tuple within the bucket. This strategy helps to reduce the final number of candidate rules

as the same rule can be applied to several sites. In Line 17, a candidate rule is added to table $\mathcal{CRT}$. In Line 20, a set of candidate rules is returned.

4.2.1.1 Token Set Classification

URL components play different roles when webmasters are designing the URL scheme of a website, e.g., some of them impact only the way pages are displayed (fonts, sizes, etc) and others are used only to identify a user connection without altering their content. Therefore, the goal of URL normalization is to distinguish URL components (or tokens) that impact page content from the ones with none or no relevant impact. In other words, it is necessary to determine the role each component plays and to infer which ones should be held, removed or generalized. For instance, tokens which express directories or document types should be held in the canonical form while tokens that denote parameter values should be generalized. After investigating a number of dup-clusters and their respective consensus sequences, we noted that the multiple alignment of URLs helps to define the importance of each component in the normalization process. In this way, given a consensus sequence $\mathcal{CS} = \langle T_1, T_2, ..., T_k \rangle$, inferred from a set of URLs $\mathcal{C}$, the token set T_i is classified as follows:

- **Irrelevant**: T_i is *irrelevant* if $\lambda \in T_i$, that is, some token of T_i was aligned with a gap during the multiple alignment process. These tokens are considered irrelevant, i.e., the page content is the same independently of their presence in the URL. For instance, tokens 11 to 13 of X' ("`index.html`") are irrelevant in example presented in Table 4.1. Such tokens should be removed from the canonical form.

- **Invariant**: T_i is *invariant* if $|T_i - \{\lambda\}| = 1$ (inside $\mathcal{C}$) and it is present in all URLs of $\mathcal{C}$. Such invariant tokens must be kept in the canonical form. This is the case of token $t_{X',9}$ = "`foia`" in Table 4.1.

- **Variant**: T_i is *variant* if $|T_i - \{\lambda\}| > 1$. Unlike from irrelevant tokens, we can not remove them from the URLs, it is necessary to choose one of them. However, regardless of the choice, the content of the page does not change. As examples of these tokens, we cite tokens denoting directories where files were copied redundantly, multiple domain names from the same website or session id lists used to identify users. This is the case of the tokens "irs" and "IRS" in Table 4.1.

4.2.1.2 Conversion of Consensus Sequences to Rules

A consensus sequence $\mathcal{CS}$ can be directly converted to a normalization rule. To accomplish this, we first divide the k token sets from $\mathcal{CS}$ into subsequences according to special URLs delimiters (e.g., '/', '?', '=', '&', '#', ';', ':', '.'). Next, each subsequence s_i is converted to regular expressions by using the token set classification (see more details in section 4.2.1.1). This process is described in the following paragraphs.

If all token sets in a subsequence s_i are invariant, we generalize them by adding a regular expression $\star$[3] to the rule context, and a backreference $N is included in rule transformation. Otherwise, each token set T_i in subsequence s_i is separately converted to a regular expression:

- If T_i is invariant, all tokens are generalized according to the type of its tokens in the rule context, and a *backreference* $N is included in rule transformation.
- If T_i is variant, all tokens from T_i are grouped inside parentheses separated by '|' in the rule context, and a randomly selected token from T_i is included in the rule transformation. An alternative is to include the group inside parentheses in both rule context and transformation.
- If T_i is irrelevant, all tokens from T_i are grouped between parentheses, separated by '|', with a question mark `?` placed after the closing bracket. The regular expression is included only in the rule context. As alternative, we can include the group in both rule context and transformation.

If a token set has more than a certain number of tokens ($Card_{set}$ threshold), we generalize this set converting it to a regular expression according to its token type: (i) Alphabetic: The group is substituted by the regular expression ([a-zA-Z]+); (ii) Numeric: The group is substituted by the regular expression ($[0-9]+$); and (iii) Punctuation: no generalization is done.

Finally, anchors '`^`' and '`$`' are included at the start and the end of the context, respectively. Thus, given a consensus sequence:

$$\mathcal{CS}_{ij} = \langle \{www\}, \{.\}, \{irs, IRS\}, \{., \lambda\}, \{ustreas, \lambda\}, \{.\}, \{gov\}, \{/\}, \{foia\}, \{/, \lambda\}, \{index, \lambda\}, \{., \lambda\}, \{html, \lambda\} \rangle$$

the subsequences are: $s_1 = www$, $s_2 = \{irs, IRS\}$, $s_3 = \{ustreas, \lambda\}$, $s_4 = gov$, $s_5 = foia$, $s_6 = \{\lambda, index\}$ and $s_7 = \{html, \lambda\}$.
The corresponding context and the transform pattern could be:

[3]Note that the symbol '$\star$' should be replaced by an appropriate regular expression.

c = ^⋆.(irs|IRS).(ustreas.)?⋆/⋆(/index.html)?$
t_1 = $1.irs.$2/$3 *or, alternatively,* t_2 = $1.(irs|IRS).ustreas.$2/$3/index.html

Note that the alternative transformation pattern t_2 will generate a canonical form with a regular expression '(irs|IRS)' and with irrelevant tokens (/index.html). This is an acceptable transformation as the canonical form does not need to be a valid URL.

4.2.2 Validating Candidate Rules

The goal of this phase is to consider as valid or refute the candidate rules generated in the previous phase. This filter selects the more effective rules by two pre-defined thresholds: false-positive rate (fpr_{max}) and minimum support (min_{supp}). If the false-positive rate is larger than fpr_{max} *or* the support of the rule is smaller than min_{supp}, the rule is discarded. Otherwise, the rule is added to the set of valid rules. Note that rules with small support values are not desirable anyway, because the reduction gained by applying them can be insignificant. Thus, the support value is indicative of the possible compression that a rule can achieve, whereas the false-positive rate corresponds to the precision of the rule in the task of DUST detection.

A web crawler can use the canonical identification obtained by the normalization rules to represent the content associated with a URL. However, a normalization rule can misclassify a non-DUST URL as DUST and, consequently, prevent the crawler of collecting new/useful content. Thus, the larger is fpr_{max}, the larger is the loss of useful content. On the other hand, a small value for fpr_{max} implies on low coverage, with more duplicate content being collected.

The solution for this trade-off depends on specific characteristics of the application scenarios. In a typical web-search scenario, in which a crawler has a very large set of URLs available to fetch, but it does not have enough resources (computer nodes and bandwidth) to crawl them, a higher false-positive threshold is advisable. Thus, even if the crawler does not fetch some unique URLs due to incorrect rules, the impact is reduced as it has enough URLs to fetch. However, if a crawler has enough resources to crawl the URLs, a lower false-positive threshold would prevent the loss of unique content due to the application of incorrect rules.

Algorithm 3 presents *ValidateRules* which takes a set of candidate rules as input and outputs a set of valid rules. It uses two thresholds to declare a rule as valid: fpr_{max} and min_{supp}. The algorithm calculates the support and false positive rate of each candidate rule by applying them to a validation set, given as input (Lines 3-30). It uses two tables during the validation phase: $\mathcal{CT}$ (Canonical Table) which stores the

Algorithm 3 ValidateRules ($\mathcal{VS}$, $\mathcal{CR}$, fpr_{max}, min_{supp})

Input: $\mathcal{VS}$: validation set, $\mathcal{CR}$: Set of n candidate rules, fpr_{max}: maximum false-positive rate that can be tolerated, min_{supp}: minimum number of instances required.

Output: Set of n valid rules $\mathcal{VR} = \{r_1, ..., r_n\}$

```
1: Create table CT (canonical, url)
2: Create table RT (context, transformation, S, support)
3: for all candidate rules r in CR do
4:    Nsupp = 0; S_support = ∅; Nfpp = 0;
5:    Let U be a set of URLs from VS where r is applicable
6:    for all URLs u in U do
7:       canonical = normalizeURL (u, r)
8:       add (canonical, u) to CT
9:    end for
10:   group tuples in CT into buckets by (canonical)
11:   for all buckets B do
12:      if (|B| > 1) then
13:         for all pairs of distinct tuple t1,t2 ∈ B do
14:            Nsupp = Nsupp + 1
15:            S_support = S_support ∪ {(t1.url, t2.url)}
16:            if (t1.url and t2.url are not DUST) then
17:               Nfpp = Nfpp + 1
18:            end if
19:         end for
20:      end if
21:   end for
22:   if (Nsupp >= min_supp) then
23:      fpr = Nfpp/Nsupp
24:      if (fpr <= fpr_max) then
25:         add (r.context, r.transformation, S, S_support) to RT
26:      end if
27:   end if
28:   Clear table CT
29: end for
30: return a set of all rules in RT
```

URL and its corresponding canonical form, and $\mathcal{RT}$ (Rule Table) which stores the valid rules.

In the first step of this phase, the set of URLs possibly affected by the candidate rule r (Line 5) is built. URLs from $\mathcal{U}$ are normalized and added with its canonical form to table $\mathcal{CT}$ (Lines 6-9). URLs converted to the same canonical form are grouped into buckets in order to calculate the fraction of DUST in the support of the rule r (Lines 11-21). In Lines 13-19, the instances of r which are not DUST are counted to

calculate the rate of false positives of this rule (Lines 22-27). If rule r passes through pre-defined criteria, it is added to table $\mathcal{RT}$ (Line 25). Otherwise, it is discarded. In Line 30, the set of valid rules from $\mathcal{RT}$ is returned.

4.2.3 URL Normalization

Since our objective is to identify DUST at crawling time, the learned rules are incorporated into the crawler in order to avoid fetching more than one URL from the same canonical form. These rules remain valid for months and even years and are valid for new pages as well as old ones. Normalization rules can be learned from an off-line computation, and they can be deployed in conjunction with the crawler to de-duping URLs and ensuring that duplicate URLs are not even crawled.

Unlike some approaches, which convert a given URL to another that likely has the same content, in our work, we treat the output of the normalization rule r as a signature to represent all URLs that have very similar page content. Whenever a set of URLs is mapped by the rules to a specific canonical form, they are all subsequently represented by just one of them and the others are discarded without checking their content. Note that, in our method, the only situation in which a web crawler loses unique URL is when two or more URLs, that are not DUST, are converted to the same canonical form.

The normalization algorithm receives a URL u and a set of valid rules R. The idea behind the algorithm is simple: It normalizes u with the first rule *applicable* in R. In order to ensure the highest possible reduction, we sort the rules in R according to their support size. Our study demonstrate that the reduction achieved by using this algorithm is high.

4.3 Summary

In this chapter we presented DUSTER. This algorithm is an evolution of our first method based on a traditional multi-sequence alignment approach to improve the generalization of transformation rules. In our first idea, many rules could be extracted from a dup-cluster and much more processing among the dup-clusters was necessary. The URL alignment was basically the same. That first idea was published in the paper entitled "Learning URL Normalization Rules Using Multiple Alignment of Sequences" [Rodrigues et al., 2013] and presented in the 20th International Symposium on String Processing and Information Retrieval (SPIRE 2013).

The version of DUSTER presented in this chapter simplified the rule generation of the SPIRE-2013 method, by allowing the extraction of a single pattern for each dup-cluster, as described in the step of candidate rule generation. This new version performed much better, being faster and more effective in removing DUST. Because of this, we only reported it in this thesis. This algorithm was published on the IEEE Transactions on Knowledge Data Engineering, in an article entitled "Removing DUST Using Multiple Alignment of Sequences" [Rodrigues et al., 2015].

While very effective as a DUST detector, our method was based on straightforward quadratic alignment strategy which did not take advantage of particular characteristics of the domain. Further, no analysis was carried out on the rules in order to eliminate the redundant ones. To cope with these issues, we devised a new DUSTER version, which we present in the next chapter.

Chapter 5

DustLin

DUST Detection with a Linear Multi-Sequence Alignment Algorithm

In this chapter, we present *DustLin* and *DustLin-MR*, our algorithms for efficiently generating URL normalization rules from dup-clusters provided as input. We proposed a new heuristic to align a set of duplicate URLs by taking advantage of particular characteristics of the domain. We also introduced a technique to discard redundant rules, since they represent waste of resources.

5.1 Heuristic for Multiple URL Alignment

The alignment of sequences is a common procedure used to determine the similarity of two or more sequences. For instance, it is commonly performed in comparisons of biological sequences, whether DNA, RNA, or proteins. In contrast with pair-wise alignment, in Biology, the multiple alignment of sequences can reveal similarities and differences among groups of proteins, and find historical and evolutionary relationships between species. As multiple alignment of sequences is computationally expensive both with respect to time and memory, a large number of heuristics has been developed in order to accelerate and increase the precision of this task [Daugelaite et al., 2013].

One of the most popular heuristics to align multiple sequences is the progressive alignment Feng and Doolittle [1987], which uses a greedy strategy, where once a space is inserted, it can not be removed for any subsequent alignment. Thus, all the spaces are preserved until the final solution. The error rate introduced by the progressive

alignment at each step tends to decrease if the most similar sequences are chosen, and tends to increase if the most divergent sequences are chosen. Thus, determining the best order for the alignments is crucial. Ideally, the most similar sequences are aligned first, leaving the most divergent ones to the end. As a result, the error introduced by this heuristic solution is reduced. The complexity of this algorithm is $O(P \times n^2)$, where P is the cost of the *pair-wise alignment* and n is the number of sequences provided as input.

Algorithms for Web mining should take advantage of the specific characteristics of the data to be efficient. Unlike biological sequences, we observed that the URLs within dup-clusters are not so dissimilar to each other. It is very common that many URLs in a dup-cluster are almost the same. As a consequence, the order in which the alignments are made could have little to no impact on the quality of the generated rules. Thus, very simple alignment heuristics can be used at an expected low error, while achieving significant gains in running times.

To illustrate that point, Table 5.1 compares the similarity of URLs selected by using the progressive alignment heuristic and a simple approach which picks URL pairs at random. To accomplish this, we sampled one third of the dup-clusters from GOV2 and WBR10 datasets and estimated the average alignment score using both heuristic approaches. The results include the standard error considering a 95% confidence level.

Table 5.1: Average alignment score for GOV2 and WBR10 datasets by using progressive alignment heuristic and a random heuristic.

GOV2		**WBR10**	
Progressive	Random	Progressive	Random
90.46±0.09	89.87±0.10	83.42±0.14	82.46±0.15

As we can see, although differences are statistically significant, the average alignment score obtained by a progressive alignment is very close to the one obtained by selecting the URL pairs at random. In GOV2, the progressive alignment score was only 0.6% better than the one obtained by the linear heuristic. A similar result is observed in WBR10 with a gain of about 1%. These results clearly suggest that a method which selects URL pairs at random can obtain a set of rules as effective as the set obtained by adopting a progressive alignment. Based on this idea, we present method *DustLin* in next section.

5.2 DustLin Algorithm

DustLin is composed of three phases. In phase 1, URLs in the training set are aligned by using a linear heuristic in order to generate candidate rules. In phase 2, incorrect candidate rules are filtered out according to their performance in a validation set. In phase 3, redundant rules are removed from the deployable set of rules. Since phase 2 is the same phase of DUSTER, we only describe the phases 1 and 3 in the following sections. But, firstly we present our algorithm for the multiple alignment of duplicate URLs.

5.2.1 Linear Multiple URL Aligment Algorithm

In this section we present our linear algorithm that aligns duplicate URLs from dup-clusters by selecting pair of URLs at random. This approach is described in details in Algorithm 4.

Algorithm 4 LinearMultipleURLAligment $(\mathcal{C}, K)$

Input: *L*et be $\mathcal{C}$ an dup-cluster with n duplicate URLs
Output: A tuple π = (consensus sequence s, $\mathcal{S}$).

1: $u_1 \leftarrow$ randomly select an URL from $\mathcal{C}$
2: $s = tokenize(u_1)$
3: $\mathcal{S} = \{hostname(u_1)\}$
4: $\mathcal{S}_{elected} = \{u_1\}$
5: **for** $i = 2$ to K **do**
6: $\quad u_i \leftarrow$ randomly select an URL from $\mathcal{C} \setminus \mathcal{S}_{elected}$
7: $\quad x = tokenize(u_i)$
8: $\quad s = PairwiseURLAlignment(x, s)$
9: $\quad \mathcal{S} = \mathcal{S} \cup \{hostname(u_i)\}$
10: $\quad \mathcal{S}_{elected} = \mathcal{S}_{elected} \cup \{u_i\}$
11: **end for**
12: **return** $\pi = (s, \mathcal{S})$

Algorithm 4 presents *LinearMultipleURLAligment* which takes a dup-cluster $\mathcal{C}$ as input and produces a tuple $\pi = (s, \mathcal{S})$, where s is a consensus sequence associated with $\mathcal{C}$, and $\mathcal{S}$ is the set of sites within $\mathcal{C}$. In lines 1-2, *LinearMultipleURLAligment* randomly selects an URL u_1 from $\mathcal{C}$ and obtains a sequence s after tokenizing it. In lines 3-4, it initializes the set of sites $\mathcal{S}$ with the hostname of the first URL u_1 and initializes the set of selected URLs with u_1. In lines 6-8, it picks the next URL u_i, tokenizes it and aligns u_i with the current sequence s. In lines 9-10, it adds the hostname of u_i to $\mathcal{S}$ and adds u_i to the set of selected URLs. This process is repeated until K URLs from $\mathcal{C}$ have been aligned.

Unlike *MultipleURLAligment* (Algorithm 1), which requires a quadratic number of sequences alignments $O(K^2 \times P)$, *LinearMultipleURLAligment* requires only a linear number of pairwise alignments. Its complexity is $O(K \times P)$, where P is the cost of the method *PairwiseURLAlignment*. For this reason, we can devise a more efficient algorithm to generate rules from the training set.

5.2.2 Candidate Rules Generation

Given a training set $\mathcal{TS} = \{c_1, ..., c_n\}$ with n duplicate clusters, our first task is to quickly generate a candidate rule for each dup-cluster in $\mathcal{TS}$. To accomplish this, we first align the URLs within each dup-cluster to obtain a consensus sequence to represent this alignment. Then, we extract a candidate rule from this consensus sequence, that is able to normalize the entire dup-cluster. Preserving each discovered rule for de-duplication is not efficient due to large number of such rules. Thus, the second step in this phase is to discard the rules with frequency less than min_{freq} from the set of candidate rules. This way, very specific rules with few occurrences in the training set are discarded before the next phase.

Algorithm 5 GenerateCandidateRules ($\mathcal{TS}$)

Input: *T*raining Set $\mathcal{TS} = \{c_1, ..., c_n\}$ with n duplicate clusters
Output: Set of m candidate rules $\mathcal{CR} = \{r_1, ..., r_m\}$
1: Create table $\mathcal{RT}$ ($context$, $transformation$, $\mathcal{S}$)
2: Create table $\mathcal{CRT}$ ($context$, $transformation$, $\mathcal{S}$)
3: **for all** cluters $c_i \in \mathcal{TS}$ **do**
4: π = LinearMultipleURLAligment(c_i, K)
5: r = generateRule ($\pi.s$)
6: add ($r.context$, $r.transformation$, $\pi.\mathcal{S}$) to $\mathcal{RT}$
7: **end for**
8: group tuples in $\mathcal{RT}$ into buckets by ($context$, $transformation$)
9: **for all** buckets B **do**
10: **if** ($|B| >= min_{freq}$) **then**
11: $\mathcal{S} = \emptyset$;
12: **for all** tuples $t \in B$ **do**
13: $\mathcal{S} = \mathcal{S} \cup t.\mathcal{S}$
14: **end for**
15: α = the first tuple in B
16: add ($\alpha.context$, $\alpha.transformation$, $\mathcal{S}$) to $\mathcal{CRT}$
17: **end if**
18: **end for**
19: a set $\mathcal{CR}$ of rules created from $\mathcal{CRT}$
20: **return** $\mathcal{CR}$

The generation of candidate rules is described in Algorithm 5. This algorithm receives as input a training set ($\mathcal{TS}$) with n dup-clusters and outputs a set with m candidate rules, $m <= n$. In lines 1-2, it creates two tables: (1) $\mathcal{RT}$ (Rules Table) which stores rules generated for every dup-cluster in $\mathcal{TS}$; (2) $\mathcal{CRT}$ (Candidate Rules Table) which stores rules that exceed the frequency threshold min_{freq}. In line 4, K URLs from dup-cluster c_i are aligned by the algorithm *LinearMultipleURLAligment*. In lines 5-6, it generates rule r and adds it to $\mathcal{RT}$. In line 8, all rules are grouped into buckets according to their context and transformation. To accomplish this, table $\mathcal{RT}$ is sorted by the first and second attributes. In lines 9-18, the algorithm enumerates all distinct rules that exceed the frequency threshold min_{freq}. In lines 11-15, the rules with same context and transformation are unified. This strategy reduces the final number of candidate rules. In line 16, a candidate rule is added to table $\mathcal{CRT}$. Finally, in line 19, a set of candidate rules is returned.

5.2.3 Eliminating Redundant Rules

The output of the validation phase may include pairs of redundant rules. For example, when running on GOV2, our algorithm found some redundant rules as shown in Table 5.2. Note that, in the table, every instance in the support set of r_2 (S_{r_1}) also appears in S_{r_2}. It means that all cases covered by rule r_1 are also covered by r_2. Thus, we can say that the former rule is *refined* (is covered) by the latter. The notion of *refinement* was presented for the first time in Bar-Yossef et al. [2006] and we define it below:

Definition 9 *(REFINEMENT). A rule r_1 refines a rule r_2, if $support(r_1) \subseteq support(r_2)$.*

That is, r_1 refines r_2, if every instance (u_i, u_j) of r_1 is also an instance of r_2. In Algorithm 6, we explore this definition by adding a phase into *DustLin* in which redundant rules can be eliminated resulting into a smaller set of rules.

Table 5.2: Example of redundant rules.

Redundant Rules Found on GOV2.
$c_1 =$ ^⋆://(www.)?⋆.⋆/⋆/$(OGWDW\|ogwdw)$(000)?/⋆/⋆.⋆$ $t_1 =$ \$1://\$2.\$3/\$4/ogwdw/\$5/\$6.\$7 $S_{r_1} = \{(u_0, u_1), (u_0, u_2)\}$
$c_2 =$ ^⋆://(www.)?⋆.⋆/⋆/$(OGWDW\|ogwdw\|safewater)$(000)?/⋆/⋆.⋆$ $t_2 =$ \$1://\$2.\$3/\$4/ogwdw/\$5/\$6.\$7 $S_{r_2} = \{(u_0, u_1), (u_0, u_2), (u_1, u_2), (u_3, u_4\}$

Algorithm 6 EliminateRedundantRules ($\mathcal{VR}$)

Input: Set of m valid rules $\mathcal{VR} = \{r_1, ..., r_m\}$
Output: Set of m rules with no redundancy: $\mathcal{R} = \{r_1, ..., r_m\}$
1: $\mathcal{E} = \emptyset$;
2: **for all** pairs of distinct *rules* R_x,$R_y \in \mathcal{VR}$ **do**
3: **if** $R_x.support \subseteq R_y.support$ **then**
4: $\mathcal{E} = \mathcal{E} \cup \{R_x\}$
5: **end if**
6: **end for**
7: $\mathcal{R} = \emptyset$
8: **for all** rules $r \in \mathcal{VR}$ **do**
9: **if** $r \notin \mathcal{E}$ **then**
10: $\mathcal{R} = \mathcal{R} \cup \{r\}$
11: **end if**
12: **end for**
13: **return** a set of all rules in $\mathcal{R}$

Algorithm 6 takes a set of m valid rules and filters out the redundant ones. Firstly, in Lines 2-6, the algorithm checks the support of all pairs of distinct rules (R_x,R_y) from $\mathcal{VR}$. If the support of R_x is a subset of R_y then R_x is added to the set of eliminated rules $\mathcal{E}$. In Lines 8-12, a new set of deployable rules $\mathcal{R}$ is created by discarding the rules present in $\mathcal{E}$ from $\mathcal{VR}$.

5.3 DustLin-MR - A Parallel version of DustLin

In this section we present Dustlin-MR, a distributed version of our algorithm for generating normalization rules at Web scale. To that end, we also briefly overview MapReduce, a popular paradigm to deal with massive amounts of data.

5.3.1 MapReduce

The Web is the biggest and fastest growing data repository in the world. The estimated size of the indexable Web was at least 11.5 billion pages as of the end of January 2005 [Gulli and Signorini, 2005]. In 2011, its size was estimated between 50 and 100 billion pages and roughly doubling every eight months [Baeza-Yates and Ribeiro-Neto, 2011].

When dealing with web-size datasets (e.g. sets of dup-clusters), the costs of a sequential algorithms are not acceptable. The size of the dataset and the structures that support the solution will easily outgrow the storage capabilities of a single machine.

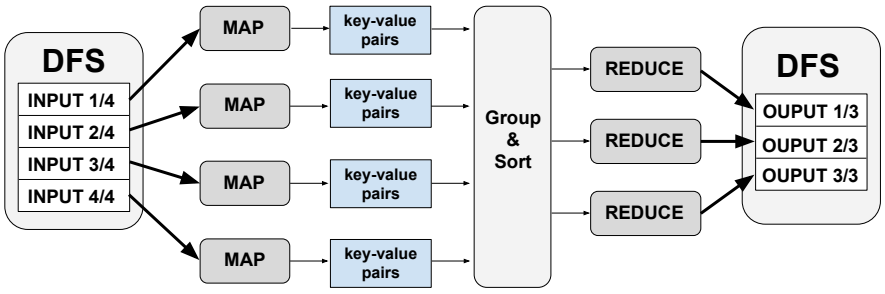

Figure 5.1: MapReduce computation data flow.

There are a variety of programming models and techniques to process data on large clusters [Jackson et al., 2015]. Among them, MapReduce is the most popular programming paradigm designed for the processing of large amounts of data by distributing work tasks over multiple machines in a shared-nothing cluster [Dean and Ghemawat, 2008][1]. The input data, represented as pairs $\langle key, value \rangle$ is initially partitioned across the nodes of the cluster and stored in a distributed file system (DFS).

The key concept behind MapReduce is inspired by the map and reduce primitives present in many functional languages. Map or reduce tasks are specified by the user and may run on different machines, allowing parallelism to be achieved. In MapReduce, map functions are applied in parallel on different partitions of the input data to compute a set of intermediate key/value pairs. The outputs from the map function are then automatically grouped by their key (e.g., $\langle key_i, list(value_i) \rangle$) and passed to the reduce function. Then reduce is applied to all values that shared the same key. The output of each *reduce* function ($\langle key_i, value_i \rangle$), is written to a distributed file in DFS. Figure 5.1 shows the data flow in a MapReduce computation.

MapReduce has gained popularity in commercial settings with many different implementations by Google [Dean and Ghemawat, 2008], Yahoo! and Microsoft [Isard et al., 2007]. In this work, we employ Hadoop[2], which is the only freely available MapReduce implementation and it was developed by Yahoo!.

5.3.2 DustLin-MR Algorithm

As previously observed, generating normalization rules at web scale is computationally demanding. In order to handle hundreds of millions of dup-clusters in time, we redesigned the first two phases of *DustLin* as a distributed algorithm based on the MapReduce framework. We refer to this method as *DustLin-MR*. It consists of two

[1] A shared-nothing cluster is a large commodity cluster interconnected by a local network.
[2] http://hadoop.apache.org

MapReduce jobs: *Rules Generation* and *Rules Validation*. The input of these jobs is represented by pairs (DID, $[u_1, u_2, ...]$), where DID is a dup-cluster ID and $[u_1, u_2, ...]$ is a list of duplicate URLs.

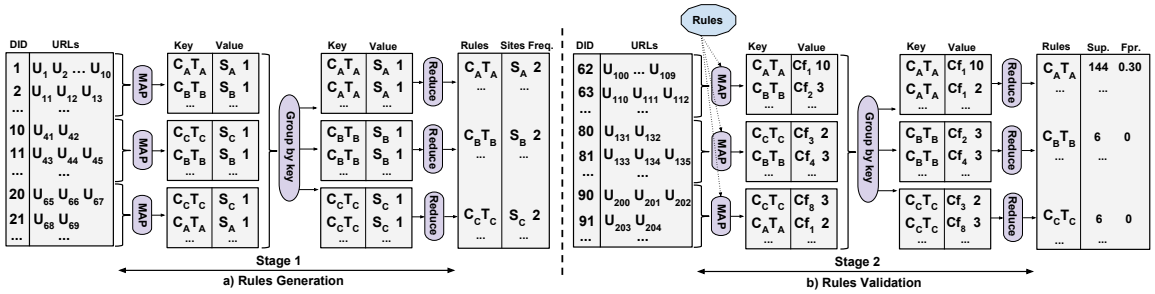

Figure 5.2: Example data flow of Stages 1 and 2.

5.3.2.1 Job 1: Rules Generation

The MapReduce job, called *Rules Generation*, receives as input a set of dup-clusters and outputs a list of candidate rules with frequency greater than min_{freq}. As we can see in Figure 5.2, this process is splitted into m map tasks, each of them operating on its own subset of dup-clusters. For each dup-cluster, the map function generates a rule and yields a pair $(C_iT_i, \langle \mathcal{S}_i, 1 \rangle)$, where C_iT_i represents the context and transformation of rule r_i. After being grouped and sorted, all the pairs belonging to each rule (i.e., which share the same key C_iT_i) are brought together. Subsequently, the *reduce* function sums up the total counting for each rule, unifies the rules according to their sites, and outputs $(C_iT_i, \langle \mathcal{S}_i, count \rangle)$ pairs, where $\mathcal{S}_i$ is the set of sites where r_i can be applied, and *count* is the total frequency for the rule i in the training set.

Figure 5.2(a) shows the data flow for a sample dataset. Take, for instance, rule C_AT_A, produced by two different map functions. After being grouped and sorted, these pairs are routed to the same reduction function to produce the output $(C_AT_A, \langle \mathcal{S}_A, 2 \rangle)$.

Algorithm 7 provides a pseudo-code implementation of map and reduce functions for this strategy. Input to this procedure *map* consists of pairs (DID, $[u_1, u_2, ...]$). For each dup-cluster, in Line 1, a rule for the dup-cluster DID is generated by calling *GenerateRule*. The procedure *reduce* takes the rule c_it_i and emits it, along with its corresponding frequency and set of sites. In Lines 3-6, the algorithm counts the frequency of the rule and unifies its sites. Finally, in Line 8, it emits the rule if the frequency is greater than min_{freq}.

Algorithm 7 Pseudo-code of Candidate Rules Generation in MapReduce

MAP (DID, $[u_1, u_2, ...]$)

1: r_i = GenerateRule($[u_1, u_2, ...]$)
2: $c_i t_i = r_i.context + r_i.transform$
3: EMIT($c_i t_i$, $\langle r_i.\mathcal{S}_i, 1\rangle$)

REDUCE ($c_i t_i, \{\langle \mathcal{S}_i, 1\rangle\}$)

1: $\mathcal{S} \leftarrow$ new SET
2: Freq = 0
3: **for all** $v \in \{\langle \mathcal{S}_i, 1\rangle\}$ **do**
4: $\quad \mathcal{S} = \mathcal{S}_i \cup v.first$
5: $\quad$ Freq = Freq + v.second
6: **end for**
7: **if** Freq >= min_{freq} **then**
8: $\quad EMIT(c_i t_i, \mathcal{S}, Freq)$
9: **end if**

5.3.2.2 Job 2: Rules Validation

The second MapReduce job, called *Rules Validation*, takes a set of dup-clusters as input and calculates the support and false-positive rate of each rule generated in the previous stage. It then outputs a list of rules with false-positive rate smaller than fpr_{max} and support greater than min_{supp}. Figure 5.2(b) shows the data flow for a sample dataset. As illustrated by that figure, before the map functions start running, an initialization function is called to load the set of candidate rules R produced in the first stage. In particular, we broadcast and load the set of rules R at each map function before the input data is consumed. The map functions then normalize the URLs from each dup-cluster by using the rules within the set R and yield $(C_i T_i, \langle Cf_n, nou\rangle)$ pairs. In these pairs, *nou* indicates the number of URLs transformed to canonical form Cf_n by the rule $C_i T_i$. After being grouped and sorted, these pairs are provided to the same reduction function which produces the output $(C_i T_i, \langle Nsupp_i, fpr_i\rangle)$, where $Nsupp_i$ and fpr_i are the support size and false-positive rate of rule r_i, respectively.

Note in Figure 5.2(b) that the rule $C_A T_A$ normalizes URLs from the dup-clusters 62 and 91 to the same canonical form Cf_1. The fpr of this rule is greater than zero as its support contains incorrect instances such as (U_{100}, U_{203}) and (U_{100}, U_{204}). Thereby, if the threshold of our algorithm is $fpr_{max} = 0$, this rule would be discarded from the deployable set of normalization rules.

Algorithm 8 provides a pseudo-code implementation of map and reduce functions for this job. In Line 1, the set of candidate rules $\mathcal{CR}$ is loaded before the reading of the input data. In Line 2, an associative array RC is created to store the number of URLs

(nou) converted to the same canonical form by every rule in $\mathcal{CR}$. Then, an auxiliary set of tuples T_{uples} is created in Line 3 to control the pairs $\langle C_iT_i, Cf\rangle$ inserted into RC. In Lines 4-12, the URLs u, from the dup-cluster DID, are processed: (a) all rules that match u are added into the set $R_{matched}$ (Line 5); (b) u is then normalized by using rules in $R_{matched}$ set. For each matched rule R_i, the pair $\langle C_iT_i, Cf_n\rangle$ is stored into RC, where Cf_n is the canonical form generated by the rule R_i (Lines 6-11). Finally, the algortihm emits a pair $(C_iT_i, \langle Cf, nou\rangle)$ for each element stored in RC (lines 13-16).

Algorithm 8 Pseudo-code of Validation Rules in MapReduce

MAP (DID, $[u_1, u_2, ...]$)
1: Load the set of candidate rules $\mathcal{CR}$
2: $RC \leftarrow$ new AssociativeArray
3: $T_{uples} \leftarrow$ new SET
4: **for all** URLs $u \in \{u_1, u_2, ...\}$ **do**
5: Let $R_{matched}$ be the set all rules from R that match u
6: **for all** rules $R_i \in R_{matched}$ **do**
7: Cf_n = NormalizeURL (u, R_i)
8: $C_iT_i = R_i.context + R_i.transform$
9: $T_{uples} = T_{uples} \cup \{\langle C_iT_i, Cf_n\rangle\}$
10: $RC[\langle C_iT_i, Cf\rangle]$++
11: **end for**
12: **end for**
13: **for all** tuples $t \in T_{uples}$ **do**
14: $C_iT_i = t.first$; $Cf_n = t.second$
15: $nou = RC[t]$
16: EMIT$(C_iT_i, \langle Cf_n, nou\rangle)$
17: **end for**

REDUCE $(C_iT_i, \{\langle Cf_n, nou\rangle\})$
1: $CA \leftarrow$ new AssociativeArray
2: $\mathcal{C} \leftarrow$ new SET with canonical forms
3: $Nci = 0$
4: **for all** $v \in \{\langle Cf_n, nou\rangle\}$ **do**
5: $Nci = Nci + \frac{nou^2 - nou}{2}$
6: $CA[Cf_n] = CA[Cf_n] + nou$
7: $\mathcal{C} = \mathcal{C} \cup \{Cf_n\}$
8: **end for**
9: Nsupp = 0
10: **for all** $c \in \mathcal{C}$ **do**
11: $count = CA[c]$
12: $Nsupp = Nsupp + \frac{count^2 - count}{2}$
13: **end for**
14: **if** $(Nsupp >= min_{supp})$ **then**
15: $fpr = 1.0 - Nci/Nsupp$
16: **if** $(fpr <= fpr_{max})$ **then**
17: EMIT$(C_iT_i, \langle Nsupp, fpr\rangle)$
18: **end if**
19: **end if**

In line 1 of the *reduce* function, an associative array CA is created to store the number of URLs transformed into each canonical form Cf_n. This array is used to calculate the support of current rule R_i. In Lines 4-8, the pairs $\langle Cf_n, nou\rangle$, obtained by rule R_i in map phase, are processed. The number of correct instances Nci in the support of the R_i is calculated (Line 5) and CA is updated (Line 6). The support of the current rule R_i is then calculated by using the array CA and the set $\mathcal{C}$ (Lines 10-13).

Finally, the algorithm emits the rule if the support $Nsupp$ is greater than min_{supp} and the fpr does not exceed fpr_{max} (lines 14-18).

5.4 Final Considerations

In this chapter, we presented *DustLin*, our method which takes advantage of specific characteristics of the DUST problem to generate normalization rules in a more efficient fashion. This chapter first described the linear heuristic to align sets of duplicate URLs and a technique to discard redundant rules, since they represent waste of resources. Whereas *DUSTER* required a quadratic number of sequences alignment, *DustLin* requires only a linear number of pairwise alignments. We then presented *DustLin-MR*, a distributed version of our algorithm for generating normalization rules. In spite of the efficiency of *DustLin*, at web scale, we have to handle hundred of millions of dup-clusters in time which motivate the study of a distributed solution. It was implemented as a 2-stage parallel algorithm using a MapReduce programming model, implemented in a Hadoop environment.

DustLin and *DustLin-MR* were described in a paper submitted to the ACM Transactions on the Web, entitled "A Highly-Scalable Algorithm For Generating URL Normalization Rules." This article is currently under review. In the next chapter, we present the experiments involving the three algorithms proposed in this thesis: *DUSTER*, *DustLin* and *DustLin-MR*.

Chapter 6

Experimental Evaluation

In this chapter, we report the experimental evaluation of the three algorithms proposed in this thesis: DUSTER, DustLin, and DusLin-MR. We first present our test datasets and then evaluate our methods, including by comparing them with state-of-the-art baselines in the tasks of generating normalization rules and detecting DUST.

6.1 Datasets

We use two document collections in our experiments: *GOV2* and *WBR10*. GOV2 dataset consists of a snapshot of the resources fetched from 25,205,179 individual documents from US government domains in 2004. According to the TREC track informationClarke et al. [2004], some duplicate documents have already been removed from GOV2. The GOV2 TREC dataset contains about 3.88 million duplicate URLs divided into about 1.43 million dup-clusters. These documents were grouped by creating a small fingerprint of their content and hashing the URLs with identical fingerprints into the same clusters.

WBR10 is a collection of over 150 million web pages crawled from the Brazilian domain using an actual Brazilian crawling system. This crawling was performed from September to October, 2010, with no restrictions regarding content duplication or quality. To identify groups of duplicate URLs in WBR10, we adopted the same approach used by the authors in Lei et al. [2010]. Thus, we scanned the collection to find out the web sites which explicitly indicate the canonical URLs in their pages. By doing this, we identified about 3.86 million duplicate documents in WBR10, for a total of about 1.11 million dup-clusters. Although WBR10 is six times larger than GOV2, it has almost the same amount of DUST identified. This was expected since webmasters are not obliged to identify canonical URLs.

Table 6.1: GOV2 and WBR10 characterization

DataSet	#Dup-Clusters	#URLs	Tokens Per URL	#Dup-Clusters Size <= 10	#Dup-Clusters Size >10	Dup-Clusters Size (Avg.)	Dup-Cluster Size (Max)
GOV2	1,432,034	3,876,604	27.20	1,424,491 (99.47%)	7,543 (0.53%)	2.71	53,132
WBR10	1,108,186	3,858,620	28.45	1,083,759 (97.80%)	24,427 (2.20%)	3.48	47,977

Table 6.2: Statistics of the sites within GOV2 and WBR10

DataSet	#Sites	#URLs in a Site (Min)	#URLs in a Site (Max)	#URLs in a Site (Avg.)	#Sites in a Dup-Cluster (Avg.)
GOV2	15,106	1	265,388	256.63	1.57
WBR10	39,094	1	57,472	98.70	1.62

Note that by using these two datasets, we are able to evaluate our method in the scenario where most of the DUST was identified (GOV2) as well as in a more realistic scenario, where only a small sample of DUST is known (WBR10). Information about these two datasets is summarized in Tables 6.1 and 6.2.

6.2 DUSTER Evaluation

In this section, we present our evaluation of DUSTER. We start with a study on the thresholds used by our method and then present the experiments we conducted to evaluate DUSTER, followed by a discussion on the obtained results.

6.2.1 Thresholds Study

The performance of DUSTER is associated with the choice of values for five thresholds: (1) the number of URLs to be aligned in each dup-cluster – K; (2) the maximum acceptable false positive rate – fpr_{max}; (3) the cardinality of a token set, necessary to it be transformed into a regular expression – $Card_{set}$; (4) the minimum frequency a rule should have to not be discarded – min_{freq}; and (5) the minimum support a rule should have to be validated – min_{supp}.

To better understand the impact of such thresholds, we randomly sampling 50% of the dup-clusters and divided them into three approximately equal-size sets. We calculated the results as the average obtained for these three sub-sets according to a 3-fold-cross-validation strategy [Mitchell, 1997]. We used the training set to generate the rules, the validation set to filter them, and the test set to evaluate them. The results of such study are presented in the following sections

6.2.1.1 Threshold K

As observed in Section 4.2.1, DUSTER randomly selects K URLs to be aligned in each dup-cluster. This is necessary to mitigate the quadratic cost of a multi-sequence alignment when applied to a very large dup-cluster. In this work we set $K = 10$ and the reason for that is twofold. First, as observed in Table 6.1, this value covers almost all dup-clusters. The number of dup-cluster with more than 10 URLs is about 2% in WBR10 and less than 1% in GOV2. These cases have little impact on the performance of the method regarding DUST detection. Second, as also observed by [Dasgupta et al., 2008], rules derived from these rare large dup-clusters are usually related to the generalization of session ids or the elimination of irrelevant path components (c.f. Appendix A). We randomly select K URLs from these dup-clusters because we believe it is not necessary inspect all the training examples to get general rules, as the pattern can be observed with just a few number of examples.

6.2.1.2 Threshold fpr_{max}

A web crawler can use the canonical identification obtained by the normalization rules to represent the content associated with a URL. However, a normalization rule can misclassify a non-DUST URL as DUST and, consequently, prevent the crawler of collecting new/useful content. Thus, the larger is fpr_{max}, the larger is the loss of useful content. On the other hand, a small value for fpr_{max} implies on low coverage, with more duplicate content being collected.

The solution for this trade-off depends on specific characteristics of the application scenarios. In a typical web-search scenario, in which a crawler has a very large set of URLs available to fetch, but it does not have enough resources (nodes and bandwidth) to crawl them, a higher false-positive threshold (e.g. $fpr_{max} \leq 10\%$) can be used. Thus, even if the crawler does not fetch some unique URLs due to incorrect rules, the impact is reduced as it has enough URLs to fetch. However, if a crawler has enough resources to crawl the URLs, a lower false-positive threshold (e.g. $fpr_{max} \leq 5\%$) would prevent the loss of unique content due to the applications of incorrect rules. In our experiments, we consider $fpr_{max} = 0$ level the most important one, as we are interested in estimating how effective are the best rules generated by DUSTER and baselines.

6.2.1.3 Threshold $Card_{set}$

Threshold $Card_{set}$ determines if token sets obtained from consensus sequences should be transformed into regular expressions according to its token type (c.f. Sec-

tion 4.2.1.2). The basic intuition is simple. Given a token set T_i and a $Card_{set}$ threshold value, we convert the token set T_i if $|T_i| \geq Card_{set}$. Table 6.3 presents the results obtained by using $Card_{set}$ values varying from 2 to 10, in GOV2 and WBR10 datasets. This range of values was chosen as it corresponds to the sizes that dup-clusters can assume using $K = 10$.

Table 6.3: $Card_{set}$ behavior for $fpr_{max} = 0$ in GOV2 and WBR10 datasets. CR stands for "Compression Rate"; NP stands for "Normalization Precision"; AR/R stands for "Average Reduction Per Rules".

$Card_{set}$	GOV2					WBR10				
	#Rules		DUST Detection			#Rules		DUST Detection		
	Candidates	Valid	CR (%)	NP (%)	AR/R	Candidates	Valid	CR (%)	NP (%)	AR/R
2	28,211	9,173	21.76	99.13	15.60	36,368	11,292	30.53	99.58	17.62
3	44,967	15,843	26.33	99.32	10.93	41,522	13,330	30.82	99.77	15.08
4	46,063	16,198	26.50	99.31	10.76	42,885	14,105	30.83	99.77	14.24
5	46,678	16,315	26.43	99.33	10.65	43,914	14,658	31.31	99.80	13.93
6	47,091	16,387	26.48	99.33	10.63	44,559	15,021	31.68	99.84	13.75
7	47,369	16,415	26.50	99.34	10.62	44,991	15,224	30.97	99.85	13.27
8	47,552	16,442	26.50	99.33	10.60	45,263	15,358	30.58	99.83	12.99
9	47,669	16,462	26.50	99.48	10.59	45,468	15,445	30.13	99.81	12.73
10	47,773	16,473	26.49	99.48	10.58	46,118	15,653	28.52	99.70	11.88

If we increase $Card_{set}$, we expect fewer token sets being generalized and, as a consequence, a larger set of rules is generated since less rules are unified. This behavior can be observed in Table 6.3, in which the larger is $Card_{set}$, the larger is the set of candidate/valid rules.

If we obtain a larger set of rules, we expect to achieve a larger compression of DUST. We can observe this in both collections when we increased $Card_{set}$ from 2 to 3. Note that Compression Rate (CR) increased from 21.76% to 26.33% and from 30.53% to 30.82% in GOV2 and WBR10, respectively. However, in WBR10, CR decreases as we increase $Card_{set}$ from 6 to 10. It happens because, in spite of being larger, the set of rules become less general as we increase $Card_{set}$. This behavior can be confirmed by AR/R (Average reduction per rule) which decreases as we increase the $Card_{set}$ threshold for both datasets. In general, the impact of the variation of $Card_{set}$ is lesser in GOV2 than in WBR10 as the dup-cluster sizes in GOV2 are smaller than in WBR10. For instance, note that, when we increased $Card_{set}$ from 3 to 10, only 630 additional valid rules were obtained in GOV2, whereas in WBR10, this number was 2,323.

Even if the normalization precision (NP) of the set of valid rules is greater than 99% for all values of $Card_{set}$ in both datasets, we conclude that the worst normalization precision was reached by the set of valid rules obtained with $Card_{set} = 2$. Therefore, we suggest the use of intermediate values for $Card_{set}$ because lower values lead to the loss of unique URLs while higher values, to small compression rates.

6.2.1.4 Threshold min_{freq}

This parameter determines when infrequent rules should be discarded from the set of candidate rules generated in the first phase of DUSTER. min_{freq} helps to remove very specific rules from the set of candidate rules. Given a rule r and its frequency $Freq_r$, DUSTER discards r if $Freq_r < min_{freq}$. For instance, if we set $min_{freq} = 1$, DUSTER does not discard any rule and all rules within the set of candidates are provided to the second phase of the method. However, if we set $min_{freq} = 10$, only rules that were generated from at least 10 different dup-clusters are retained as candidates. Therefore, in order to analyze its behavior, we vary min_{freq} from 1 to 10 and the results are summarized in Table 6.4.

Table 6.4: Parameter min_{freq} for $fpr_{max} = 0$ and $Card_{set} = 5$ in GOV2 and WBR10 datasets. CR stands for "Compression Rate"; NP stands for "Normalization Precision"; AR/R stands for "Average Reduction Per Rules"

min_{freq}	GOV2						WBR10					
	#Rules			DUST Detection			#Rules			DUST Detection		
	Candidates	Valid	Rate	CR (%)	NP (%)	AR/R	Candidates	Valid	Rate	CR (%)	NP (%)	AR/R
1	46,678	16,315	34.95	41.48	99.33	10.65	43,914	14,658	33.38	43.70	99.80	13.93
2	11,424	8,467	74.12	39.51	99.45	19.55	6,174	4,926	79.79	38.40	99.93	36.40
3	7,568	5,966	78.83	38.30	99.72	26.89	3,814	3,196	83.80	37.07	99.94	54.11
4	5,751	4,611	80.18	37.32	99.82	33.90	2,749	2,318	84.32	35.70	99.93	71.89
5	4,750	3,839	80.82	36.65	99.83	40.00	2,183	1,841	84.33	34.55	99.94	87.63
6	4,058	3,282	80.88	35.95	99.84	45.88	1,825	1,528	83.73	33.95	99.95	103.83
7	3,559	2,868	80.58	35.31	99.86	51.57	1,599	1,338	83.68	33.31	99.95	116.27
8	3,160	2,541	80.41	34.72	99.87	57.25	1,434	1,202	83.82	32.75	99.95	127.18
9	2,862	2,293	80.12	34.28	99.87	62.65	1,306	1,099	84.15	32.40	99.98	137.73
10	2,613	2,090	79.98	33.83	99.88	67.81	1,197	1,009	84.29	31.97	99.98	148.10

If we increase the min_{freq} threshold, most general and precise rules are kept in the final set rules. Note in Table 6.4 that, for all datasets., the average reduction per rule (AR/R) increases as we increase min_{freq} as well as the set of rules tend to be more precise. On the other hand, the number of candidate/valid rules decreases as we increase min_{freq} and, by extent, the compression achieved by DUSTER. This behavior is observed in both datasets. Note that, in average, about 34% of the rules are considered as valid when DUSTER does not discard any rules from the set of candidates ($min_{freq} = 1$). This rate increased to 74.12% and 79.79% when we increased the min_{freq} to 2 in GOV2 and WBR10 datasets, respectively.

6.2.1.5 Threshold min_{supp}

This parameter determines if rules are accepted or not in the DUSTER validation phase. Given a rule r, its support, and a validation set VS, DUSTER discards r if

$supp(r, VS) < min_{supp}$. To understand the behavior of min_{supp}, we vary its value from 1 to 10 and the results are summarized in Table 6.5.

Table 6.5: Parameter min_{supp}. CR stands for "Compression Rate"; NP stands for "Normalization Precision"; AR/R stands for "Average Reduction Per Rules"

min_{supp}	GOV2					WBR10				
	#Rules		DUST Detection			#Rules		DUST Detection		
	Candidates	Valid	CR (%)	NP (%)	AR/R	Candidates	Valid	CR (%)	NP (%)	AR/R
1	46,678	16,315	41.48	99.33	10.65	43,914	14,658	43.70	99.80	13.93
2	46,678	13,251	41.10	99.44	12.99	43,914	12,616	43.60	99.81	16.14
3	46,678	11,658	40.77	99.45	14.65	43,914	11,363	43.50	99.81	17.90
4	46,678	10,263	40.47	99.48	16.52	43,914	10,409	43.39	99.82	19.50
5	46,678	9,387	40.15	99.48	17.92	43,914	9,770	43.29	99.83	20.72
6	46,678	8,759	39.97	99.48	19.12	43,914	9,299	43.19	99.83	21.73
7	46,678	8,052	39.65	99.48	20.63	43,914	8,787	43.05	99.85	22.92
8	46,678	7,588	39.39	99.49	21.75	43,914	8,472	42.98	99.85	23.74
9	46,678	7,221	39.22	99.49	22.75	43,914	8,223	42.92	99.85	24.43
10	46,678	6,855	38.98	99.49	23.82	43,914	7,970	42.84	99.85	25.16

Based on the results given in Table 6.5, we can conclude that min_{supp} has a similar behavior to min_{freq}. For instance, the resulting set of valid rules tend to be smaller, more precise and more general as we increase min_{supp}. Further, the compression rate (CR) also decreases as we increase min_{supp}. Note that fpr_{max} and min_{supp} are threshold values used by $DUSTER$ and the baselines (cf. Problem 1).

6.2.2 Comparison with Previous Work

In this section we present a comparison between the results obtained by DUSTER with those obtained by two state-of-the-art methods. In particular, we compare the methods according to the number of rules they detected, the number of valid rules they selected, and their performance in DUST detection. We also study the results of applying the rules to better understand the methods that derive them.

As previously described, we adopted two different baseline methods for comparison: (1) The first is the work by Dasgupta et al Dasgupta et al. [2008], which we implemented using the *fanout* heuristic. For this task, we use a threshold value equal to 10, which is the same value adopted by the authors in their experiments. (2) Our second baseline is the method proposed in Lei et al. [2010], which we refer to as R_{tree}. R_{tree} builds the so called *pattern tree* for each target site. These two methods were chosen due to their performance in previous experiments, which indicate they represent the best options found in literature for de-duplicating URLs.

6.2.2.1 Empirical Methodology

In all the experiments, we calculated the metrics presented in Section 2.5.2 as the average obtained for three sub-sets of URLs according to a 3-fold-cross-validation strategy [Mitchell, 1997], as follows. We randomly divided the dup-clusters of each collection into three approximately equal-size sets. From each of these three subsets, the first one was retained as training set, the second one was retained as validation set, and the remaining as test set. We then performed 3 runs, rounding the sets such that a same URL was never used as test example in different runs.

The time complexity of our first baseline $R_{fanout\text{-}10}$ is at least $O(cn^2)$, where c is the number of dup-clusters in the training set and n is the average number of URLs in each dup-cluster. As we presented in Table 6.2, in practice, a dup-cluster can have tens of thousands URLs, such that the processing of the entire clusters is unfeasible. We thus decided firstly sample K URLs for each dup-cluster before providing them as input. We adopted this strategy for our method and all the baselines.

Based on our previous studies on parameters, the following thresholds were used in our experiments: $K = 10$ and $min_{supp} = 10$. Regarding DUSTER, we used $min_{freq} = 10$ and $Card_{set} = 5$.

6.2.2.2 Candidate Rules vs. Valid Rules

In this section we analyze the number of rules learned by the three methods after the training (candidate rules) and the number of the rules ready to be used in the test (valid rules). Note that a small number of valid rules is desirable since the crawler should have a small footprint. For $R_{fanout\text{-}10}$ and DUSTER, the valid rules consist of the rules that were not discarded in the validation. The rules considered invalid are automatically removed from the rule pool. For R_{tree}, the valid rules are the ones picked out in the selection phase.

Table 6.6: Number of candidates and valid rules generated by different methods in *GOV2* and *WBR10* ($fpr_{max} = 0$).

DataSet	Method	Candidates	Valid	Rate
GOV2	$R_{fanout\text{-}10}$	12,249	6,517	53.20%
	R_{tree}	5,523	1,756	31.79%
	$DUSTER$	4,411	3,072	69.64%
WBR10	$R_{fanout\text{-}10}$	10,501	6,793	64.68%
	R_{tree}	13,542	3,103	22.91%
	$DUSTER$	1,980	1,550	78.28%

In Table 6.6, we compare, for $fpr_{max} = 0$, how many candidate rules are generated

and, out of them, how many are valid. These results show that even though DUSTER generates the smallest number of candidate rules, it has the highest rate of valid rules.

6.2.2.3 DUST Detection

In this section we present a comparison between DUSTER and the baseline methods regarding the task of DUST detection for the GOV2 and WBR10 datasets. Table 6.7 shows, for each fpr_{max} level and method, the number of applied/valid rules (A/V), along with its respective compression rate (CR), the coverage of the rules, the normalization precision (NP) and the average reduction per rule (AR/R).

The performance of DUSTER was far superior when compared to the baselines at all fpr_{max} levels experimented. As we are interested in estimate how effective are the best rules generated by the methods, we consider $fpr_{max} = 0$ level the most important one, since it includes rules that did not fail in any of the test URLs in the validation set. At this level, DUSTER was able to reduce the amount of URLs crawled in 24.07% in GOV2, while the best baseline ($R_{fanout10}$) achieved only 16.04%. In WBR10, DUSTER was able to reduce 27.59%, while the best baseline ($R_{fanout10}$) achieved only 16.94%. These results show that DUSTER obtained a gain in the process of identifying duplicate URLs of 50% in GOV2 and about 63% in WBR10, by applying almost two times less rules than $R_{fanout10}$ in GOV2 and three times in WBR10. Thus, besides achieving a higher compression rate, the rules generated by DUSTER are more effective than the ones generated by $R_{fanout10}$.

The next measure we use to evaluate DUSTER against the baselines is *Coverage*, i.e., the amount of DUST discovered by the applied rules compared to the total amount of DUST in a dataset. As we can see in Table 6.7, the rules generated by DUSTER can cover more DUST than those generated by $R_{fanout10}$ and R_{tree} at all false-positive levels in both collections. For instance, for $fpr_{max} = 0\%$ in GOV2, the coverage achieved by DUSTER was about 42% while our best baseline, $R_{fanout10}$, achieved about 28%. In WBR10 for $fpr_{max} \leq 5\%$, DUSTER covered about 60% whereas $R_{fanout10}$ covered about 29%. Note that there is almost no gain in coverage for $fpr_{max} \geq 5\%$ for all methods. This happens because only a small number of additional rules are used as fpr_{max} increases.

Table 6.7: Results obtained at each false-positive rate by $R_{fanout\text{-}10}$, R_{tree} and $DUSTER$ for *GOV2* and *WBR10* datasets. Column A/V stands for "number of rules Applied/Valid"; CR stands for "Compression Rate"; *NP* stands for "Normalization Precision" and AR/R stands for "Average Reduction per Rule".

fpr_{max}	$R_{fanout\text{-}10}$					R_{tree}					$DUSTER$				
	A/V	CR (%)	Coverage (%)	NP (%)	AR/R	A/V	CR (%)	Coverage (%)	NP (%)	AR/R	A/V	CR (%)	Coverage (%)	NP (%)	AR/R
GOV2															
= 0%	4146/6517	16.04	27.82	99.69	27.80	506/1756	7.04	12.16	98.48	50.56	2206/3072	24.07	41.73	99.91	88.35
≤ 5%	5624/9280	20.09	34.87	99.39	24.40	498/1803	6.93	11.97	98.19	48.68	2186/3588	31.96	55.34	99.10	100.43
≤ 10%	5655/9421	20.41	35.38	98.95	24.43	513/1839	7.07	12.22	97.37	48.57	2214/3745	33.93	58.66	98.33	102.14
≤ 15%	5877/9752	20.99	36.36	98.72	24.26	518/1866	7.23	12.47	96.85	48.88	2228/3813	34.17	59.06	98.11	101.03
≤ 20%	5978/9891	21.01	36.41	98.56	23.95	530/1920	7.48	12.88	95.46	49.42	2233/3839	34.23	59.18	98.05	110.56
WBR10															
= 0%	4220/6793	16.94	27.87	92.58	23.47	1747/3103	8.08	13.28	94.55	24.54	1386/1550	27.59	45.40	99.97	167.58
≤ 5%	4233/6872	17.53	28.84	93.13	24.01	1767/3158	8.42	13.84	94.45	25.12	1266/1694	36.14	59.46	99.80	200.79
≤ 10%	4254/6901	17.55	28.89	93.12	23.95	1794/3214	8.52	14.00	94.20	24.98	1277/1707	36.28	59.70	99.73	200.08
≤ 15%	4263/6912	17.58	28.93	93.09	23.94	1807/3247	8.60	14.12	93.75	24.95	1285/1717	36.37	59.84	99.71	199.46
≤ 20%	4274/6931	17.60	28.96	93.08	23.90	1831/3296	8.70	14.28	92.95	24.86	1296/1728	36.44	59.95	99.70	198.46

We note that R_{tree} presented the worst performance among the methods we implemented. Such a weak performance was due to the fact that it was designed to conduct normalization within websites, being unable to generate rules involving multiple sites. As we presented in Table 6.2, for each dup-cluster in our collections, there are duplicate URLs coming from more than one website. Therefore, the necessity of splitting the URLs according to each target site can partially explain its weak performance. Further, in their experiments, each site has in average 352,106 URLs. As we also presented in Table 6.2, GOV2 has in average only 257 URLs and WBR10 has only 99 URLs per site. It needs more training examples than we were able to provide in our collections. In sum, as observed by the authors, their algorithm was designed for normalization within websites with enough training data.

In both collections, DUSTER generates a small number of rules which are more generic. This generality is clearly associated with the URL alignment since the rules are directly extracted from the aligned sequences. In general, DUSTER was quite effective and is a viable alternative for solving the DUST detection problem. When considering other false-positive levels experimented, again DUSTER was able to outperform the baselines. For instance, when considering a $fpr_{max} \leq 20\%$ on WBR10 dataset, DUSTER reduced the number of crawled URLs in 36.44% of the original set of URLs, two times more than the best baseline, that reduced only 17.60%. In GOV2, for $fpr_{max} \leq 20\%$, DUSTER reduced 34.23% of URLs, while the best baseline $R_{fanout10}$ reduced only 21.01%.

6.2.2.4 Rules Evaluation

In order to better understand the performance of the methods, we analyze the performance of the *applied rules* by each method in collections GOV2 and WBR10, respectively. For this purpose, we consider only rules that were able to convert at least one pair of URLs to the same canonical form. Table 6.7 shows, for each method, the false-positive threshold rate used in the validation phase (fpr_{max}), the average number of rules effectively applied among the ones that were considered as valid (A/V), the normalization precision (column NP) obtained in task of DUST detection and the average of URLs reduced per applied rule (AR/R).

By comparing the fpr_{max} used to select the deployed rules and the real precision reached, we note that $R_{fanout10}$ and R_{tree} were better in the validation set than in the test set. In particular in WBR10, for $fpr_{max} = 0\%$, $R_{fanout10}$ and R_{tree} presented worse precision in the test than in the validation for $fpr_{max} = 0\%$. It implies that these rules presented a satisfactory performance during validation, but they did not

repeat this performance during test. The results presented in Table 6.7 suggest that R_{tree} experienced moderate overfitting and $R_{fanout10}$ experienced strong overfitting. For both collections, DUSTER presented the largest precision rates in the test (99.91% in GOV2 and 99.97% in WBR10).

Table 6.7 also gives the average reduction per rule (AR/R) for different levels of fpr_{max}. As shown, the AR/R for our method is much higher than $R_{fanout10}$ and R_{tree} at all false-positive levels. For instance, for $fpr_{max} = 0\%$ in GOV2, DUSTER achieved AR/R = 88.35 whereas $R_{fanout10}$ and R_{tree} achieved 27.80 and 50.56, respectively. Although R_{tree} was worse than $R_{fanout10}$ regarding the compression rate in GOV2, its AR/R was higher in all of the false-positive levels. R_{tree} seemed not be able to output general rules for both collections. Its rules present high average reduction per rule in GOV2 and low average reduction per rule in WBR10. Regarding $R_{fanout10}$, its bad performance is clearly related to the specificity of its rules. Note that it has the largest set of deployed (valid) rules among the methods. To make matters worse, its rules have normalization precision lower than DUSTER rules.

Table 6.8 provides examples of valid rules generated by the methods, URLs[1] in training and test sets, and canonical forms. URLs in the training sets are shown organized into dup-clusters D_1 to D_4. Valid rules obtained by algorithms $R_{fanout\text{-}10}$, R_{tree} and DUSTER are presented as pairs of context and transformation expressions. Finally, test URLs are presented as pairs of duplicates (for instance, u_{13} and u_{14} are duplicates) followed by their corresponding canonical forms obtained by $R_{fanout\text{-}10}$ (n_{rf}) and DUSTER (n_d). We note in this table that DUSTER used only two rules, (c_4, t_4) and (c_5, t_5), to canonize all the test URLs. $R_{fanout\text{-}10}$ and R_{tree} were not able to canonize URL pairs (u_{13}, u_{14}) and (u_{15}, u_{16}) because they found very specific rules, (c_0, t_0) and (c_3, t_3). As R_{tree} is not able to find rules for URLs from different domains, it failed to find appropriate rules for the pairs (u_{17}, u_{18}) and (u_{19}, u_{20}). Like DUSTER, $R_{fanout\text{-}10}$ correctly canonized these pairs but it has to use two different rules, (c_1, t_1) and (c_2, t_2), whereas DUSTER uses only one (c_5, t_5).

6.3 DustLin Evaluation

In this section we investigate how efficient is DustLin to generate a set of deployable rules. We compare to DUSTER, which takes advantage of a traditional multiple sequence alignment algorithm, and $R_{fanout\text{-}10}$, a bottom-up pairwise strategy which is one

[1]Due to space constraints, URL strings were slightly changed. For instance, 'a.l.g/m/r' corresponds to 'altruistic.lbl.gov/mirrors/redhat' in the real dataset.

Training URLs
dup-cluster D_1
u_0 = `http://a.l.g/m/r/6.2/ja/dochts/formats/pdf/smb-ht.pdf`
u_1 = `http://a.l.g/m/r/6.2/en/dochts/formats/pdf/smb-ht.pdf`
u_2 = `http://a.l.g/m/r/6.2/fr/dochts/formats/pdf/smb-ht.pdf`
dup-cluster D_2
u_3 = `http://a.l.g/m/r/6.2/ja/dochts/formats/pdf/Tips-ht.pdf`
u_4 = `http://a.l.g/m/r/6.2/en/dochts/formats/pdf/Tips-ht.pdf`
u_5 = `http://a.l.g/m/r/6.2/fr/dochts/formats/pdf/Tips-ht.pdf`
dup-cluster D_3
u_6 = `http://comprar.vlume.com.br/cpm-22/`
u_7 = `http://www.vlume.com.br/cpm-22/`
u_8 = `http://www.vlumi.com.br/cpm-22/`
dup-cluster D_4
u_{10} = `http://comprar.vlume.com.br/d-black/`
u_{11} = `http://www.vlumi.com.br/d-black/`
u_{12} = `http://www.vlume.com.br/d-black/`
Valid Rules (learned from training URLs)
$R_{fanout\text{-}10}$
c_0 = http://a.l.g/m/r/6.2/⋆/dochts/formats/pdf/⋆
t_0 = http://a.l.g/m/r/6.2/ja/dochts/formats/pdf/$2
c_1 = http://www.vlumi.com.br/⋆/
t_1 = http://comprar.vlume.com.br/$1/
c_2 = http://www.vlume.com.br/⋆/
t_2 = http://comprar.vlume.com.br/$1/
R_{tree}
c_3 = http://a.l.g/m/r/6.2/⋆/dochts/formats/pdf/⋆
t_3 = http://a.l.g/m/r/6.2/en/dochts/formats/pdf/$2
DUSTER
c_4 = ^⋆://⋆.⋆.⋆/⋆/⋆.⋆/(en|fr|ja)/⋆/⋆/⋆/⋆/⋆.⋆$
t_4 = $1://$2.$3.$4/$5/$6/$7.$8/en/$9/$10/$11/$12.$13
c_5 = ^⋆://(comprar|www).(vlume|vlumi).⋆.⋆/⋆/$
t_5 = $1://comprar.vlume.$2.$3/$4/
Test Set
u_{13} = `http://a.l.g/m/r/6.2/en/dochts/loc/be/be-ht-1.html`
u_{14} = `http://a.l.g/m/r/6.2/fr/dochts/loc/be/be-ht-1.html`
n_d = `http://a.l.g/m/r/6.2/en/dochts/loc/be/be-ht-1.html`
u_{15} = `http://a.l.g/m/r/7.2/fr/dochts/trans/es/isc.html`
u_{16} = `http://a.l.g/m/r/7.2/en/dochts/trans/es/isc.html`
n_d = `http://a.l.g/m/r/7.2/en/dochts/trans/es/isc.html`
u_{17} = `http://www.vlume.com.br/banda-h8/`
u_{18} = `http://www.vlumi.com.br/banda-h8/`
n_d = `http://comprar.vlume.com.br/banda-h8/`
n_{rf} = `http://comprar.vlume.com.br/banda-h8/`
u_{19} = `http://comprar.vlume.com.br/jack7/`
u_{20} = `http://www.vlume.com.br/jack7/`
n_d = `http://comprar.vlume.com.br/jack7/`
n_{rf} = `http://comprar.vlume.com.br/jack7/`

Table 6.8: Examples of URLs in training and test sets, rules derived from the training set and canonical forms (in bold face) obtained by DUSTER (n_d) and $R_{fanout\text{-}10}$ (n_{rf}).

of the state-of-the-art DUST detection method. Due to the weak performance of R_{tree} in our previous experiments, we decided to omit its results in this section.

All the experiments presented in this section were conducted on a single machine (i7 intel 920 processor running on 2.5GHz) with a Linux operating system (standard Ubuntu distribution 14.10). The time of execution is the sum of *sys* and *user* figures provided by Linux command *time*. The speeds that are reported are calculated by averaging the execution time of three runs.

6.3.1 Comparison with Previous Work

We start by evaluating the impact of using the linear heuristic presented in Section 5.1 to generate rules from a set of dup-clusters. For comparison, Table 6.9 presents the running times and the number of rules learned by DUSTER and DustLin. The table reports, for each method, rules learned after the training (candidate rules) and after validation (valid rules). As we note, when considering running time, DustLin remarkably outperforms DUSTER. For instance, in GOV2, DustLin was 7.58 times faster than DUSTER to generate rules. In WRB10, DustLin was 12.68 times faster than DUSTER. Note that almost the same proportion of generated rules were also valid in both datasets. This indicates that DustLin generates a quite close number of rules as compared with DUSTER, but spending much less time.

Table 6.9: Number of candidates and valid rules generated by DUSTER and DustLin in *GOV2* and *WBR10* datasets.

DataSet	Method	Candidates	Valid	Rate	Running Time (s)
GOV2	DUSTER	4,411	3,072	69.64%	1,296
	DustLin	4,421	3,087	69.83%	171
WBR10	DUSTER	1,980	1,550	78.28%	1,864
	DustLin	1,953	1,557	79.72%	147

Table 6.10 presents the results of using a phase to eliminate redundant rules from the set of rules generated by DustLin (cf. Section 5.2.3). In this table, we show the number of valid rules, the number of rules after the filtering was applied, and the time spent in this phase. As we can see, the removal of redundant rules resulted in a

Table 6.10: Rate of redundant rules filtered out after the execution of third phase of our method in *GOV2* and *WBR10*. Running time in seconds.

DataSet	Valid	No Redundancy	Rate	Running Time (s)
GOV2	3,087	2,461	-20.28%	7
WBR10	1,557	1,420	-8.80%	45

elimination rate of about 20% of the total number of rules learned from GOV2. This process took about 7 seconds to be carried out. In WBR10, a smaller reduction rate was obtained (about 9%) in a filtering which took about 45 seconds. Anyway, such result clearly shows the importance of filtering out redundant rules, thus avoiding the generation of a large number of rules which would not be useful for the DUST detection.

Table 6.11: Compression and number of rules obtained by $R_{fanout\text{-}10}$, DUSTER and DustLin for *GOV2* and *WBR10* datasets. Column CR stands for "Compression Rate"; AR/R stands for "Average Reduction per Rule"; and *NP* stands for "Normalization Precision".

DataSet	Method	Rules	Applied	% of Rules	CR (%)	Coverage (%)	NP (%)	AR/R	Time (s)
GOV2	$R_{fanout\text{-}10}$	6,517	4,146	63.62%	16.04	27.82	99.69	27.80	418
	DUSTER	3,072	2,206	71.81%	24.07	41.73	99.91	88.35	1,296
	DustLin	2,461	2,219	90.17%	24.29	42.12	99.91	111.31	178
WBR10	$R_{fanout\text{-}10}$	6,793	4,220	62.12%	16.94	23.47	92.58	27.87	472
	DUSTER	1,550	1,386	89.42%	27.59	45.40	99.97	167.58	1,864
	DustLin	1,420	1,391	97.96%	27.73	45.63	99.98	183.88	192

Table 6.11 shows a comprehensive comparison between DustLin and the baseline methods regarding DUST detection. From Table 6.11, it is clear that DustLin is more efficient than $R_{fanout\text{-}10}$ and DUSTER. Note that the running time of DustLin is the sum of time spent to generate the set of valid rules and time to eliminate redundant ones. In GOV2 DustLin took roughly 178 seconds to finish, while $R_{fanout\text{-}10}$ and DUSTER spent 418 and 1,296 seconds, respectively. In WBR10, the total running time of DustLin was about 192 seconds against 472 seconds for $R_{fanout\text{-}10}$ and 1,864 seconds for $DUSTER$. This demonstrates that our linear heuristic for multiple URL alignment can improve the computational efficiency of the process of generating rules.

In the other hand, we need to investigate the impact on quality of the rules generated by DustLin. In particular, we need to estimate how effective is its set of rules in comparison of $R_{fanout\text{-}10}$ and DUSTER. Table 6.11 shows the number of rules and the compression rate of each method. We observe that DustLin has the smallest and more useful set of deployable rules in both collections. For instance, DustLin was able to reduce the amount of URLs in 24.29% from GOV2 whereas $R_{fanout\text{-}10}$ and $DUSTER$ reduced 16.04% and 24.07%, respectively. These results show that DustLin is able to generate a set of rules as effective as the ones obtained by $DUSTER$, but spending much less time. To achieve this reduction, DustLin applied 90.17% of its rules, while $R_{fanout\text{-}10}$ and $DUSTER$ applied 63.62% and 71.81%, respectively. This demonstrate that the technique to remove the redundant rules was able to discard rules that in fact were not applied in practice.

Another interesting result showed in Table 6.11 is AvgReductionPerRule (AR/R) that helps to estimate how general a set of rules is. As shown, the average reduction per rule for DustLin is higher than the $R_{fanout\text{-}10}$ and DUSTER in both datasets. For instance, in GOV2, DustLin reduced 111.31 URLs per rule whereas $R_{fanout\text{-}10}$ and DUSTER reduced 27.80 and 88.35 per rule, respectively. These results were expected, as DustLin reduced the number of rules in relation to DUSTER and maintained almost the same Compression Rate (CR).

In sum, DustLin achieved very close compression rate as DUSTER, without loss in precision in both datasets. In contrast, DustLin was faster and generated less rules than DUSTER. Thus, our proposed linear alignment heuristic was able to accelerate the generation of candidate rules and the technique to remove the redundant rules was able to discard rules that which would no be useful for the DUST detection.

6.3.2 DustLin: Sampling vs. Entire Cluster

In section 4.2.1, we argued that we should avoid the alignment of very large dup-clusters as it could be very expensive. Thus, we adopted a sampling strategy in order to mitigate the cost of a multi-sequence alignment when applied to these dup-clusters. According to [Dasgupta et al., 2008], the rule types learned from dup-clusters of size greater than 10 URLs involve at most session ids and irrelevant paths components. Therefore, we chose to randomly select 10 URLs to be aligned in each dup-cluster for two reasons. First, this value covers almost all dup-clusters. Second, we believe that is not necessary to inspect all training examples to learn the general rule. Note that this strategy affects only the minority of the dup-clusters and these cases could have a negligible impact on the performance regarding DUST detection.

Although DUSTER has to been clearly superior to the baselines, we could not compare its results by aligning a sample of URLs against the entire cluster due to its quadratic complexity. However, as DustLin has linear complexity on the number of URLS to be aligned, we decided to investigate the impact of aligning the entire cluster instead of a sample of URLs.

Table 6.12: DustLin: Sampling vs. Entire Cluster in GOV2 and WBR10 datasets.

DataSet	Method	Rules	CR (%)	Coverage (%)	NP (%)	AR/R	Time (s)
GOV2	Sampling	3,079	20.96	33.33	99.84	87.83	157
	Entire	3,080	20.95	33.31	99.84	87.74	330
WBR10	Sampling	1,475	20.97	29.42	99.98	183.07	145
	Entire	1,483	20.94	29.39	99.98	181.92	687

Table 6.12 shows a comparison when DustLin aligns (a) a sample of 10 URLs from the dup-clusters, and aligns (b) all URLs within them. As we expected, the sampling strategy mitigated the costs of DustLin in both datasets. For instance, in WBR10 the sampling strategy accelerated the process of generating rules in 4.74 times. In GOV2, DustLin was about 2 times faster. From Table 6.12, it is clear that rules are as effective as the ones generated by aligning the entire dup-clusters. Note that the reduction was almost the same in both collections. These results confirm our initial idea, and URL sampling is a viable solution to the problem of aligning very large dup-clusters.

6.4 DustLin-MR Evaluation

We now investigate how efficient is the MapReduce version of *DustLin*. We present experiments using our MapReduce strategy to assess its scalability, in terms of hardware and input data. To understand the performance of parallel algorithms we need to measure absolute running time as well as relative speedup and scaleup [DeWitt and Gray, 1992].

Testing System. To create a shared-nothing cluster, we make use of Amazon Web Services (AWS) [AWS, 2006], a set of cloud computing services provided by Amazon. Amazon Elastic MapReduce (Amazon EMR) is an Amazon Web Service which offers a hosted Hadoop framework using Elastic Compute Cloud (EC2) for computation and Simple Storage Service (S3) for input and output data storage. We ran experiments on a 16-nodes cluster. Each node was configurated with one Intel Xeon processor E52670 with 8 cores, 30GB of RAM, and two 80GB hard disks. In total, the cluster consisted of 128 cores and 32 disks. We used an extra node for running the master daemons to manage the Hadoop jobs and the Hadoop distributed file system. Each node was running Hadoop 0.20.1.

Datasets and Methodology. We evaluated DustLin-MR with the same datasets GOV2 and WBR10. As we are only interested in studying how efficiently our parallel algorithm can generate the normalization rules, we use 100% of the dup-clusters of each dataset for training and randomly chose about 33% for validation. To evaluate our algorithm in a larger dataset, we decided to combine GOV2 and WBR10 to create a new dataset GOV2+WBR10. This new dataset contains 7.74 million duplicate URLs divided into about 2.54 million dup-clusters.

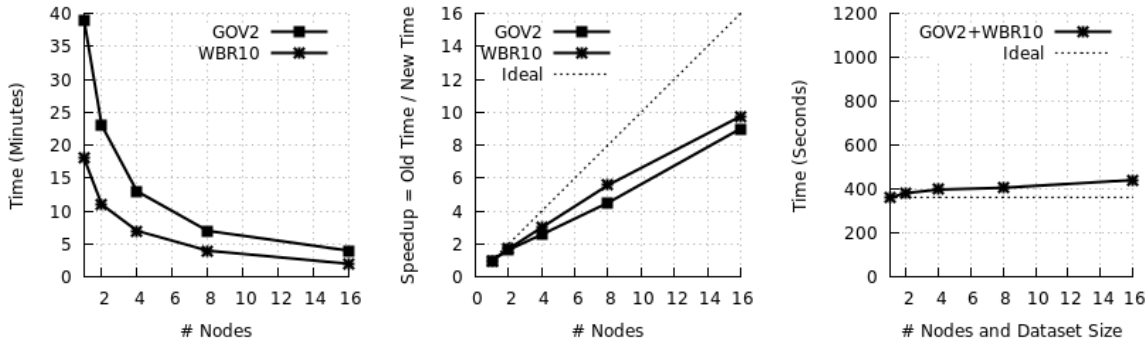

Figure 6.1: a) Running time for GOV2 and WBR10 on different cluster sizes; b) speedup of GOV2 and WBR10 as more nodes are added; c) running time of each stage in GOV2 + WBR10 dataset increased proportionally with the increase of the cluster size.

6.4.1 Running Time

Figure 6.1(a) shows the running time in both GOV2 and WBR10 datasets, using clusters with 1 to 16 nodes (8 to 128 cores). We plot the running time for DustLin-MR as we increase cluster size. Note that, as expected, the running time decreases as more machines are added. Table 6.13 shows the running time for each stage. The running time of the Rules Generation (RG) stage is very similar in both the collections since they have similar amount of dup-clusters. However, note that Validation Rules (VR) stage spent 30 minutes in GOV2 against 10 minutes in WBR10. It happened because the number of candidate rules generated from GOV2 is 9,870 against 4,920 from WBR10.

DataSet	**Stage**	**# Nodes**				
		1	**2**	**4**	**8**	**16**
GOV2	RG	9 min	5 min	3 min	2 min	1 min
	VR	30 min	18 min	10 min	5 min	3 min
WBR10	RG	8 min	5 min	3 min	2 min	1 min
	VR	10 min	6 min	4 min	2 min	1 min

Table 6.13: Running time of each stage for GOV2 and WBR10 collections on different cluster sizes.

6.4.2 Speedup

Speedup S_m is defined as $S_m = \frac{T_1}{T_m}$, where m is the number of machines, T_1 is the running time of the algorithm on a single machine, and T_m is the running time in parallel, using m machines. In order to evaluate the speedup of the algorithm, we keep the dataset size constant and increase the number of nodes in the cluster. Thus, if

the cluster has twice as many nodes and the data size does not change, the approach should be about twice as fast. This is known as linear speedup (where $S_m = m$) and is the ideal scenario for parallel processing. However, linear speedup is rare in practice because of communication and synchronization overheads and the growing influence of small sequential sections of code as the number of nodes increases (which is known as Amdahl's law [Amdahl, 1967]).

Figure 6.1(b) shows the speedup achieved in both datasets by DustLin-MR. The ideal speedup is indicated by a thin dotted black line. As expected, we observe in this figure that our method scales sublinearly as we increase the number of nodes allocated to generate the normalization rules. However, at least up to 16 nodes, the performance increases steadily as the number of nodes increases. This result shows that our method can satisfactorily scale on large cluster of machines, which is essential for web-scale collections.

6.4.3 Scaleup

Another important performance measure is scaleup, which captures the scalability of a parallel algorithm to handle larger datasets when more computing nodes are made available. In order to evaluate the scaleup of the proposed approach we increased the dataset size and the cluster size together by the same factor. A perfect scaleup could be achieved if the running time remained constant.

Figure 6.1(c) shows the running time for GOV2+WBR10 dataset, increased from 1 to 16 times (from about 160k dup-clusters to 2.54 million), on a cluster with 1 to 16 nodes, respectively. We can see that our method scales well as the size of the input data and the cluster size are increased. Table 6.14 shows the running time (in seconds) for each stage. Note that DustLin-MR shows only a small performance degradation with the increase in the system scale (scaleup). This means that with p times more computers, the algorithm can process a collection p times higher at almost the same time. For instance, DustLin-MR spent about 6 minutes to generate normalization rules from 160k dup-clusters whereas it spent 7.35 minutes to generate rules from 2.4 million dup-clusters.

DataSet	**Stage**	**# Nodes/ Dataset Size**				
		1/x1	2/x2	4/x4	8/x8	16/x16
GOV2 + WBR10	RG	130 s	136 s	140 s	142 s	154 s
	VR	232 s	246 s	258 s	264 s	287 s

Table 6.14: Running time of each stage in GOV2 + WBR10 dataset increased proportionally with the increase of the cluster size.

6.5 Summary

In this chapter, we evaluated algorithms DUSTER, DustLin, and DusLin-MR. To this, we presented the datasets used in such evaluation, provided studies on (1) the thresholds; (2) size, precision, and generality of the rules generated by the methods; (3) computational efficiency of the methods in sequential and parallel scenarios; and (4) how well the methods perform compared to previous state-of-the-art methods in literature. Based on these evaluations, we observed that the multi-sequence alignment led to better sets of rules. We also demonstrated that the proposed sequential and parallel methods are very efficient. In the next chapter, we present our concluding remarks and perspectives for future research.

Chapter 7

Conclusions and Future Work

7.1 Conclusion Remarks

In this thesis, we presented DUSTER, a new method to address the DUST problem, that is, the detection of distinct URLs that correspond to pages with duplicate or near-duplicate content. DUSTER learns normalization rules that are very precise in converting distinct URLs which refer the same content to a common canonical form, making it easy to detect them.

To achieve this, DUSTER applies a strategy based on a traditional multi-sequence alignment of training URLs with duplicate content. By taken advantage from the fact that the URLs in a dup-cluster are very similar, we proposed DustLin, a linear version of the multi-sequence alignment algorithm. The method performs much faster without loss in precision in two datasets. It is also able to deliver a small set of rules since redundant rules are removed.

We also proposed a parallel version of DustLin, the DustLin-MR. It was implemented as a 2-stage parallel algorithm using a MapReduce programming model, implemented in a Hadoop environment. In particular, we studied the computational efficiency of the proposed approach and have shown the feasibility of distributing the task of generating normalization rules across a cluster of machines. This implementation should support upcoming large-scale corpora.

This research was motivated by some questions for which we now provide answers in the following paragraphs.

How effective is a DUST detection approach based on a multiple sequence alignment when compared to traditional approaches? More specifically, how effective is it regarding rule generalization, the number of rules generated, and DUST removal?

As demonstrated in our experiments, accurate and general normalization rules can be obtained by performing a traditional multi-sequence alignment of duplicate URLs. Our algorithm is configurable for different false-positive rates to achieve high compression rates. To summarize our results, we achieved compression rates of up to 28% using the best rules generated by our algorithm. If we restrict the rules to the ones that have higher false-positive rates, it can still achieve compression of up to 36%. If integrated into a web crawler, these rules can save considerable bandwidth and storage.

We evaluated how effective is DUSTER on two different datasets. We compared the results with state-of-the-art approaches and we showed that our method can greatly reduce the number of duplicates with few false-positives. We evaluated DUSTER in a set of duplicate URLs extracted from the TREC GOV2 collection. DUSTER achieved a compression gain of about 50% over our best baseline. For that, it used a set of rules 52.86% smaller. When evaluating using a Brazilian web dataset, we obtained a gain in compression of up to 62.87% over the same baseline. This time, DUSTER used a set of rules 77.18% smaller. We also showed that our algorithm achieves not only high compression rates but also high average reduction per rule. For instance, our average reduction per rule in GOV2 and WBR10 are 88.35 and 167.58, respectively, compared to 27.80 and 23.47 from our best baseline.

Is it possible to take advantage of specific characteristics of the DUST detection problem to improve the alignment computational performance?

As previously mentioned, we proposed an algorithm called Dustlin which adopts a linear alignment strategy based on the fact that URLs in dup-cluster are very similar. In comparison with DUSTER method, DustLin remarkably improved the computational efficiency of the learning process. Further, DustLin delivers a set of rules about 20% smaller since it removes the redundant rules. Experimental results demonstrated that DustLin achieved similar reduction rates to DUSTER without loss in precision. In comparison with a state-of-the-art method, DustLin significantly reduced the running time of the learning process by a factor of about 57% and 59% in GOV2 and WBR10 datasets, respectively.

How efficient is the approach when the rules are learned in parallel on a per-dupcluster basis?

To investigate the parallel learning of rules on a per-dupcluster basis, we proposed Dustlin-MR. We studied the scalability of DustLin-MR by running experiments in a 16-node cluster. As we expected, the running time decreases as more machines are added. For instance, DustLin-MR spent 39 minutes in GOV2 for 1 node and 4 minutes using

16 machines. In WBR10, DustLin-MR spent 18 minutes for 1 node and 2 minutes with 16 nodes. We observed that our method scales sublinearly as we increase the number of nodes allocated to generate normalization rules.

We also increased the dataset size and the cluster size together by the same factor in order to investigated the scalability of the method to handle larger datasets when more computing nodes are made available. DustLin-MR spent about 6 minutes to generate rules from 160k dup-clusters whereas it spent 7.35 minutes to generate rules from 2.4 million of dup-clusters. These results show that our method can satisfactorily scale on large cluster of machines and is viable to support upcoming large-scale corpora.

7.2 Limitations of this work

During this research, we have faced some difficulties and, as a consequence, our results present some limitations. Among them, we cite:

- All available reference collections used in DUST literature have few annotated documents. While some works have adopted web-scale datasets, these were conducted by companies using proprietary data. Such resources were not made available. This issue impacted more on the experiments with the parallel algorithm, since large datasets are necessary to a detailed study on scalability.

- Similarly to the described in the previous item, the only parallel algorithm for DUST detection in literature [Koppula et al., 2010] was neither made available for download nor described in enough details to allow us to implement it.

- The annotation used in GOV dataset presents the following issues. Duplicate pages were not manually identified. For each page was assigned a hash value, obtained using the MD-5 algorithm [Clarke et al., 2004]. If two pages presented the same hash value, they were considered duplicates. This approach is problematic since (i) many near duplicates or (ii) duplicates not synchronized on time will not be labeled as DUST. As consequence, we expect a higher false negative rate.

- The annotation used in WBR10 dataset also presents issues. Duplicates were selected using the canonical tag. In spite of there is no study about the coverage of the canonical tags in any collection, we do not expect a large coverage as its use is optional. As consequence, many pairs of duplicates were probably not included in the dataset due to missing canonical tags.

7.3 Future work

As observed, all available reference collections in literature are very small compared to real datasets. As such, it is necessary to build larger datasets to be used as future reference collections. A promising evanue towards this end is to gather the common crawl datasets [Foundation, 2011] and identify its duplicate URLs using canonical tags. Common crawl is the current largest available web collection, with billions of pages crawled along the last seven years. Thus, as future work, we intend to evaluate our methods using such data.

Another issue with reference collections is the reliability of the data. Methods based on signature or canonical tags can lead to false positives in the training datasets. As future work, we intend to study the impact of these false positives, since that the current algorithms trust on such information. More specifically, we intend to carry out a detailed error analysis to investigate possible strategies to minimize this problem such as, for instance, the identification and removal of outliers.

We also intend to study the crawling overhead resulting from incorporating DUST detection in a real search engine architecture. Aspects to be considered include, for instance, impact of false positive rate in search results and which rule updating policy should be adopted.

We finally intend to adopt noise models [Jurafsky and Martin, 2009] to the problem of DUST detection. Noise models have been successfully applied in domains such as sequence alignment [Powell et al., 2004] and spelling correction [Brill and Moore, 2000]. For instance, in spelling correction, the incorrect input can be seen as a corrupted version of the word that the user intended to type. The corruption is associated with noise introduced in the process due to physical factors such as motor control of fingers, phonetic similarities between words, lack of vocabulary familiarity, etc [Hanada et al., 2016]. The likelihood of these factors can be estimated from examples of errors observed in a general corpus. Likewise, we can see a duplicate URL u as a noised version of another URL, u'. The transformation of u into u' results from sets of sub-string transformations due to unknown noise factors. These factors could be estimated from the dup-clusters. We believe this would be a promising research direction because we would derive the likelihood of each component transformation directly from real data.

Bibliography

Agarwal, A., Koppula, H. S., Leela, K. P., Chitrapura, K. P., Garg, S., GM, P. K., Haty, C., Roy, A., and Sasturkar, A. (2009). Url normalization for de-duplication of web pages. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 1987--1990, New York, NY, USA. ACM.

Alsulami, B. S., Abulkhair, M. F., and Eassa, F. E. (2012). Near duplicate document detection survey. *the Proceedings of International Journal of Computer Science and Communications Networks*, 2(2):147--151. ISSN 2249-5789.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. pages 483--485.

AWS (2006). Amazon Web Services.

Baeza-Yates, R. and Ribeiro-Neto, B. (2011). In *Modern Information Retrieval: The Concepts and Technology Behind Search*. Addison Wesley, 2 edition.

Bar-Yossef, Z., Keidar, I., and Schonfeld, U. (2006). Do not crawl in the dust: different urls with similar text. In *Proceedings of the 15th international conference on World Wide Web*, pages 1015--1016. ACM.

Berners-Lee, T., Fielding, R., and Masinter, L. (1998). RFC 2396: Uniform resource identifiers (URI): Generic syntax. RFC 2396.

Bharat, K. and Henzinger, M. R. (1998). Improved algorithms for topic distillation in a hyperlinked environment. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 104--111. ACM.

Blackshields, G., Sievers, F., Shi, W., Wilm, A., and Higgins, D. G. (2010). Sequence embedding for fast construction of guide trees for multiple sequence alignment. *Algorithms Mol Biol*, 5:21. ISSN 1748-7188.

Brill, E. and Moore, R. C. (2000). An improved error model for noisy channel spelling correction. In *Proc. ACL'00*, pages 286--293.

Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1):107--117.

Broder, A. Z. (1997). On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21--29. IEEE.

Broder, A. Z., Glassman, S. C., Manasse, M. S., and Zweig, G. (1997). Syntactic clustering of the web. In *Selected Papers from the Sixth International Conference on World Wide Web*, pages 1157--1166, Essex, UK. Elsevier Science Publishers Ltd.

Chakrabarti, S., Dom, B., Raghavan, P., Rajagopalan, S., Gibson, D., and Kleinberg, J. (1998). Automatic resource compilation by analyzing hyperlink structure and associated text. *Computer Networks and ISDN Systems*, 30(1):65--74.

Charikar, M. S. (2002). Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 380--388, New York, NY, USA. ACM.

Clarke, C. L. A., Craswell, N., and Soboroff, I. (2004). Overview of the trec 2004 terabyte track. In Voorhees, E. M. and Buckland, L. P., editors, *TREC*, volume Special Publication 500-261. National Institute of Standards and Technology (NIST).

da Costa Carvalho, A. L., de Moura, E. S., da Silva, A. S., Berlt, K., and Bezerra, A. (2007). A cost-effective method for detecting web site replicas on search engine databases. volume 62, pages 421--437. Elsevier.

Dasgupta, A., Kumar, R., and Sasturkar, A. (2008). De-duping urls via rewrite rules. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, pages 186--194, New York, NY, USA. ACM.

Daugelaite, J., O'Driscoll, A., and Sleator, R. D. (2013). An overview of multiple sequence alignments and cloud computing in bioinformatics. *ISRN Biomathematics*, 2013.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107--113.

DeWitt, D. and Gray, J. (1992). Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85--98.

Feng, D. F. and Doolittle, R. F. (1987). Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of molecular evolution*, 25(4):351--360. ISSN 0022-2844.

Fetterly, D., Manasse, M., and Najork, M. (2003). On the evolution of clusters of near-duplicate web pages. In *Proceedings of the First Conference on Latin American Web Congress*, LA-WEB '03, pages 37--, Washington, DC, USA. IEEE Computer Society.

Foundation, C. C. (2011). Common Crawl Dataset. `http://www.commoncrawl.org/`. [Online; accessed 01-August-2016].

Gulli, A. and Signorini, A. (2005). The indexable web is more than 11.5 billion pages. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 902--903. ACM.

Hanada, R., da Graca Pimentel, M., Cristo, M., and Lores, F. A. (2016). Effective spelling correction for eye-based typing using domain-specific information about error distribution. In *ACM CIKM'16*, New York, NY, USA. ACM.

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

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. (2007). Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59--72. ACM.

Jackson, J. C., Vijayakumar, V., Quadir, M. A., and Bharathi, C. (2015). Survey on programming models and environments for cluster, cloud, and grid computing that defends big data. *Procedia Computer Science*, 50:517--523.

Jurafsky, D. and Martin, J. H. (2009). Spelling Correction and the Noisy Channel. The Spelling Correction Task. In *Speech and Language Processing, 2nd Ed.* Prentice-Hall.

Katoh, K., Misawa, K., Kuma, K., and Miyata, T. (2002). MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform. *Nucleic Acids Research*, 30(14):3059–3066.

Kim, S. J., Jeong, H. S., and Lee, S. H. (2006). Reliable evaluations of url normalization. In *Computational Science and Its Applications-ICCSA 2006*, pages 609--617. Springer.

Kleinberg, J. M. (1999). Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604--632.

Koppula, H. S., Leela, K. P., Agarwal, A., Chitrapura, K. P., Garg, S., and Sasturkar, A. (2010). Learning url patterns for webpage de-duplication. In *Proceedings of the third ACM international conference on Web search and data mining*, WSDM '10, pages 381--390, New York, NY, USA. ACM.

Kumar, J. P. and Govindarajulu, P. (2009). Duplicate and near duplicate documents detection: A review. *European Journal of Scientific Research*, 32:514–527.

Lee, S. H., Kim, S. J., and Hong, S. H. (2005). On url normalization. In *Computational Science and Its Applications–ICCSA 2005*, pages 1076--1085. Springer.

Lei, T., Cai, R., Yang, J.-M., Ke, Y., Fan, X., and Zhang, L. (2010). A pattern tree-based approach to learning url normalization rules. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 611--620, New York, NY, USA. ACM.

Mao, X., Liu, X., Di, N., Li, X., and Yan, H. (2011). Sizespotsigs: an effective deduplicate algorithm considering the size of page content. In *Proceedings of the 15th Pacific-Asia conference on Advances in knowledge discovery and data mining - Volume Part I*, PAKDD'11, pages 537--548, Berlin, Heidelberg. Springer-Verlag.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition. ISBN 0070428077, 9780070428072.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443--453.

Notredame, C. (2002). Recent progress in multiple sequence alignment: a survey. *Pharmacogenomics*, 3(1):131--144.

Powell, D. R., Allison, L., and Dix, T. I. (2004). Modelling-alignment for non-random sequences. In *Proc. Australian J. Conf. Adv. Artificial Intelligence*, pages 203--214. Springer-Verlag.

Rivest, R. (1992). The md5 message-digest algorithm.

Rodrigues, K. W. L., Cristo, M., de Moura, E. S., and da Silva, A. S. (2013). Learning url normalization rules using multiple alignment of sequences. In Kurland, O.,

Lewenstein, M., and Porat, E., editors, *SPIRE*, volume 8214 of *Lecture Notes in Computer Science*, pages 197–205. Springer.

Rodrigues, K. W. L., Cristo, M., de Moura, E. S., and da Silva, A. S. (2015). Removing DUST using multiple alignment of sequences. *IEEE Trans. Knowl. Data Eng.*, 27(8):2261--2274.

Soon, L.-K., Ku, Y.-E., and Lee, S. H. (2012). Web crawler with url signature a performance study. In *Data Mining and Optimization (DMO), 2012 4th Conference on*, pages 127--130. IEEE.

Soon, L.-K. and Lee, S. (2010). Reducing redundantweb crawling using url signatures. In *Web-Based Information Technologies and Distributed Systems*, volume 2 of *Atlantis Ambient and Pervasive Intelligence*, pages 115–140. Atlantis Press.

Soon, L.-K. and Lee, S. H. (2008a). Enhancing url normalization using metadata of web pages. In *Computer and Electrical Engineering, 2008. ICCEE 2008. International Conference on*, pages 331--335. IEEE.

Soon, L.-K. and Lee, S. H. (2008b). Identifying equivalent urls using url signatures. In *Signal Image Technology and Internet Based Systems, 2008. SITIS'08. IEEE International Conference on*, pages 203--210. IEEE.

Theobald, M., Siddharth, J., and Paepcke, A. (2008). Spotsigs: robust and efficient near duplicate detection in large web collections. In *In SIGIR08: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 563--570. ACM.

Wang, X.-D., Liu, J.-X., Xu, Y., and Zhang, J. (2015). A survey of multiple sequence alignment techniques. In *International Conference on Intelligent Computing*, pages 529--538. Springer.

Appendix A

Supporting Examples

In this section we show a variety of DUST examples with their respective canonical forms.

A.1 URLs Search Engine Friendly

In order to make URLs more search engine friendly, webmasters create statics URLs for all dynamics URLs in their web sites. For example, the following two URLs, dynamic and static, point to the same content:

```
https://pt.wikipedia.org/?title=Flamengo
https://pt.wikipedia.org/wiki/Flamengo
```

If we consider a canonical form as a regular expression, a possible canonical form for the two URLs above could be:

```
https://pt.wikipedia.org/(\?)?(title|wiki)(=|/)Flamengo
```

A.2 Content-Neutral Parameters

Session-ids. Some parameters in dynamics URLs are considered as content-neutral, i.e. if we put any value in these type of parameters, the returned content is the same. Suppose the content-neutral parameter **id** in the following set of URLs:

```
https://example.com/index.php?id=5123
https://example.com/index.php?id=1253
https://example.com/index.php?id=2351
```

```
https://example.com/index.php?id=5213
```

A canonical form for the all URLs above is presented as follows:

```
https://example.com/index.php?id=*
```

Note that the canonical form has a symbol '*' at the end. In this case, the last '*' could be a regular expression such as $[0-9]+$. Alternatively, we could put a random value for the parameter **id** (e.g. 5123).

Irrelevant Path Components. Some components in the static part of the URLs can be considered as content-neutral. In other words, we can say that they are like session-ids that occur in the static part of the URLs. Considering the following set of duplicate URLs:

```
http://www.amazon.com/LordRings/dp/B00003CWT6
http://www.amazon.com/LordOfRings/dp/B00003CWT6
http://www.amazon.com/LordOfTheRings/dp/B00003CWT6
http://www.amazon.com/AnyString/dp/B00003CWT6
```

The canonical form that represents all URLs above is presented as follows:

```
http://www.amazon.com/*/dp/B00003CWT6
```

A.3 Tokens Transpositions

A common type of DUST occurs when the parameters of dynamic URLs are placed in distinct positions as, for example:

```
https://ex.com/show.php?prod=skirt&color=blue&size=p
https://ex.com/show.php?prod=skirt&size=p&color=blue
```

The canonical form that represents the URLs above is:

```
https://ex.com/show.php?prod=skirt&(color|size)=(blue|p)&(size|color)=(p|blue)
```

A.4 Redirecting Subdomain Folder to Subdomain URL

Another type of DUST occurs when webmasters redirect their subdomains's folder to their subdomain URL:

```
http://domain.com/subdomain/y
http://subdomain.domain.com/y
```

A possible regular expression that represents the canonical form to the URLs above can be written as follow:

```
https://(subdomain.)?domain.com/(subdomain)?/y
```