The Impact of Near-Duplicate Documents on Information Retrieval Evaluation

by

Hani Khoshdel Nikkhoo

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Near-duplicate documents can adversely affect the efficiency and effectiveness of search engines. Due to the pairwise nature of the comparisons required for near-duplicate detection, this process is extremely costly in terms of the time and processing power it requires. Despite the ubiquitous presence of near-duplicate detection algorithms in commercial search engines, their application and impact in research environments is not fully explored. The implementation of near-duplicate detection algorithms forces trade-offs between efficiency and effectiveness, entailing careful testing and measurement to ensure acceptable performance. In this thesis, we describe and evaluate a scalable implementation of a near-duplicate detection algorithm, based on standard shingling techniques, running under a MapReduce framework. We explore two different shingle sampling techniques and analyze their impact on the near-duplicate document detection process. In addition, we investigate the prevalence of near-duplicate documents in the runs submitted to the adhoc task of TREC 2009 web track.

Acknowledgements

I would like to thank Charles L.A. Clarke, my supervisor, for providing me the opportunity to work with him and the freedom to grow. His support was always felt and I am grateful for that.

Gordon V. Cormack and Mark D. Smucker accepted to read my thesis in a very short period of time. I must thank them both for their consideration and helpfulness.

Chrysanne DiMarco was the one who introduced me to my supervisor Charles L.A. Clarke. I would like to thank her as well.

I would also like to thank all the members of the PLG lab who were part of the amazing experience that I had in the School of Computer Science at the University of Waterloo from 2008 until 2010.

To My Family

Table of Contents

List of Tables

List of Figures

"The only true wisdom is in knowing you know nothing."

-Socrates

Chapter 1

Introduction

1.1 Motivation and Justification of the Research

The presence of near-duplicate documents (NDDs) adversely impacts both the *efficiency*[1] and the *effectiveness*[2] of information retrieval systems. The prevalence of NDDs is particularly high in Web search, but NDDs may appear in other contexts as well. Efficiency is adversely affected because NDDs increase the space needed to store the index and slow down response time[44]. The negative impact on effectiveness is due to the appearance of redundant information, which may exasperate users. Moreover, some recently proposed effectiveness measures explicitly penalize redundancy and reward novelty [28, 22]. In order to build high performance IR systems it is essential to appropriately identify and remove NDDs. Nonetheless, identifying NDDs has a much wider range of applications. Some of these applications are as following:

- Technical support document management

  Many companies like Hewlett-Packard have millions of technical support documents which

[1] By *efficiency*, we mean the conventional measures that are used for evaluating information retrieval systems. Efficiency is typically measured in terms of time (e.g. response time) [for further information see pages 8, 75, 468 of [18]]

[2] *Effectiveness* of information retrieval (IR) systems depends on the human judgment of relevance of the retrieved information. [for further information see pages 8, 67, 538, 584 of [18]]

are frequently merged and groomed. In this process it is very important to identify NDDs [38].

- Plagiarism detection
  Modern electronic technologies have made it extremely easy to plagiarize. In order to tackle this problem NDD detection mechanisms can be used (e.g. [74, 45]).

- Web crawling
  The drastic growth of the World Wide Web requires modern web crawlers to be more efficient. NDD detection algorithms are one of the means that can be used in this regard (e.g. [61, 64]).

- Digital libraries and electronic publishing
  Effectively organizing large digital libraries, which include several large electronically published collections and news archives with some overlap, requires NDD detection algorithms (e.g. [21]).

- Database cleaning
  In database systems an essential step for data cleaning and data integration is the identification of NDDs (e.g. [8]).

NDD detection and elimination is a standard practice for commercial web search, but outside the major search companies, the problem is not well analyzed. Potthast and Stein have meticulously described, in their taxonomy of NDD detection algorithms [67], that all NDD detection algorithms follow a general pattern. They divide each document (d) into chunks (c). This set of chunks (C_d) is then filtered by a selection heuristic and then the outcome of that filter is hashed and used for comparison of documents. As a result, these algorithms are mostly distinguishable from each other in terms of chunk creation, selection heuristic or hashing techniques.

Broder et al. [14, 17] used contiguous blocks of word sequences, called *shingles*, to detect near-duplicate web pages. Charikar [23] proposed a locality sensitive hashing scheme for comparing documents. Later, Henzinger [44] combined the algorithms of Broder et al. and Charikar to improve overall precision and recall. Recently, Qi Zhang et al. [87] suggested a new algorithm based on sequence matching which determines the location of duplicated parts in documents. Far

fewer researchers have investigated the impact of NDDs on search results. Bernstein and Zobel [6] studied redundant documents in the runs submitted to TREC 2004 terabyte track[3].

In this thesis, we build on the ideas in this prior work. Our efforts are shaped by our experiences with the TREC Web Track[4], which currently uses the 25TB ClueWeb09 collection[5]. This collection was crawled from the commercial Web in early 2009, and represents a reasonable snapshot of the Web at that time.

We describe our experience with a MapReduce [33] implementation of an algorithm for NDD detection, which is primarily based on Broder's technique [14]. For implementation purposes, we rely on the Hadoop[6] open source version of MapReduce and the Amazon Elastic MapReduce[7]. Due to the pairwise nature of the comparisons required for NDD detection, this process is extremely costly in terms of the time and processing power it requires. Inevitably a viable practical solution needs to be *highly scalable*. With careful optimization, our MapReduce implementation provides this property.

1.2 Statement of The Problem

In most IR tasks two documents are considered *similar* when there is some semantic *relevance* between them. But at the same time the two documents can be very different in their syntax. On the other hand, in early database research a very conservative definition is adopted for similarity. In that context, syntactically almost-identical documents are targeted[11, 17, 74]. Nonetheless, as Metzler et al. [63] and Hui Yang et al. [85] have pointed out, many applications require the detection of *intermediate level of similarity*. In this work, we focus on near-duplicate document detection as a form of intermediate level of similarity.

This thesis addresses the impact of near-duplicate documents (NDDs) on web search results. With the current size of the World Wide Web, which includes more than one trillion unique URLs

[3] `trec.nist.gov`
[4] `plg.uwaterloo.ca/~trecweb`
[5] `boston.lti.cs.cmu.edu/Data/clueweb09`
[6] `hadoop.apache.org`
[7] `aws.amazon.com`

[2], any viable solution for tackling this issue should be highly scalable in order to have practical applications.

1.3 Accomplished Tasks and Contributions

In order to find NDDs on the web, first we propose an NDD detection algorithm based on the MapReduce [33] framework. Our MapReduce algorithm is inspired by the shingling technique suggested by Broder et al. [12, 17, 16, 14]. For implementing the proposed solution, we use the Amazon Elastic MapReduce which uses Hadoop.

We verify our implementation by repeating one of the major experiments of Bernstein and Zobel on the GOV2 collection [6] and comparing our results with theirs. In order to improve the scalability of our solution, we examine two different shingle sampling techniques and study their average error, correlation, precision and recall with respect to previous results. We apply our improved algorithm to the runs submitted to adhoc task of the TREC 2009 web track, and study the prevalence of NDDs in these search results. Moreover, we study the source of those NDDs which belong to the ClueWeb09 collection.

1.4 Brief Outline of the Thesis

This thesis is organized as follows:

- *Chapter 1* includes a brief introduction about near-duplicate document detection and its applications, the statement of the problem addressed in the thesis and a description of the accomplished tasks. It also includes the outline of the thesis.
- *Chapter 2* provides a review of the literature of duplicate document detection algorithms. It also explains the TREC Web Track and two of the data collections used by TREC namely GOV2 and ClueWeb09. Then it describes the MapReduce framework and its open-source implementation named Hadoop. Furthermore, it describes Elastic MapReduce, Elastic Compute Cloud, Simple Storage Service and SimpleDB which are all part of the Amazon Web Services and have been used for the experiments run for this thesis. In the end, the

chapter provides the details of MapReduce cluster configuration and initializations that are used in the following chapters.

- *Chapter 3* presents the proposed MapReduce algorithm for finding NDDs and the details of its implementation. In addition, it includes the verification of the implementation by repeating one of the major experiments of Bernstein and Zobel on the GOV2 collection [6]. It also discusses two different shingle sampling techniques for facilitating large scale deployments of the algorithm: Hash-value-based sampling and Threshold-based sampling. Moreover, it investigates how the application of these two techniques affects the quality of the results by studying four different quantitative measures namely average error, correlation, recall and precision.

- *Chapter 4* mainly focuses on studying the prevalence of NDDs in the runs submitted to TREC 2009 Web Track. First, it studies the prevalence of NDDs in the collection of runs submitted for each of the TREC 2009 Web Track topics. Then, it studies the prevalence of duplicates in each of the runs submitted for each of the queries (i.e. run-topics). Finally, the chapter studies the sources of NDDs.

- *Chapter 5* summarizes the thesis and discusses the results and conclusions. It also proposes some future work.

Chapter 2

Background and Preliminaries

2.1 Duplicate Detection

In the early 1990s, Manber [59] proposed the first algorithm for near-duplicate file detection and developed a tool for it called sif . This algorithm was based on comparison of sequences of adjacent bytes and it was intended for applications in file management, file synchronization, data compression, and maybe even plagiarism detection.

Later, Heintze [43] suggested a more scalable document fingerprinting technique based on rare chunks of text. The technique is resilient to noise introduced by type conversion of documents (e.g. postscript to plain text).

Shivakumar and Garcia-Molina [74] developed the Stanford Copy Analysis Mechanism (SCAM) in 1996 to deal with the problem of plagiarism and illegal copying of documents. Their method exploits an inverted index of the text chunks.

Broder et al. [13, 17] suggested mathematical notions for document *resemblance* and *containment*. Their idea is based on splitting documents into several smaller chunks of text named shingles. By doing so, they reduce the issue of duplicate document detection to set intersection problems.

Chowdhury et al. [25] proposed the I-Match approach which works based on collection statistics. They show that their algorithm scales reasonably well in terms of the number of documents.

In addition, their algorithm works fine with documents of different sizes and is faster than the initial algorithm suggested by Broder et al.[17].

Charikar [23] proposed a locality sensitive hashing scheme for comparing documents. The hashing function is constructed based on the relationship of rounding algorithms for fractional solutions of Linear Programming problems and vector solutions of Semi-Definite Programming problems on the one hand, and hash functions for specific classes of objects on the other hand.

Winnowing is the algorithm proposed by Schleimer et al. [72] for document fingerprinting and duplicate detection. The algorithm describes an efficient procedure for sampling a small number of hashes of k-grams from each document for the purpose of document comparison. It defines a window of size w(user defined) to be w consecutive hashes of k-grams in a document. By choosing at least one hash from each window, it guarantees that at least part of any sufficiently long (i.e. $w + k - 1$) match is detected.

Fetterly et al. [36] expanded Broder's work by introducing the notion of megashingles. They found out that the clusters of NDDs on the web are fairly stable. In other words, two documents that are near-duplicates of one another are very likely to be near-duplicates in 10 weeks. This finding means that web crawlers can be fairly confident that two documents that are judged as near-duplicates will remain as near-duplicates in the near future and only one of them needs to be crawled.

Later, Henzinger [44] combined the algorithms of Broder et al. [17] and Charikar [23] to improve overall precision and recall. Furthermore, Henzinger's algorithm performs better than the other two algorithms in finding NDDs on the same site.

Bernstein and Zobel [7] presented SPEX which is a novel hash-based technique for detecting duplicate parts of documents. They also studied the redundant documents of TREC 2004 terabyte track [6]. Moreover, in another study [89] of NDDs they have tried to clarify the difficult to define concept of "duplicate" and they have highlighted a paradox of computer science research: "objective measurements of outcomes involves subjective choice of preferred measures and attempts to define measures can easily founder in circular reasoning".

Huffman et al. [48] focus only on the results of the same query. Therefore, the number of pairwise comparisons that they face is small. They use a machine learning technique for improving

the recall of Charikar's approach [23]. In addition, for some web pages they use extended fetching techniques to fill in the frames and execute JavaScript.

SpotSigs is the algorithm presented by Jonathan, Theobald et al. [50, 75] which focuses on finding NDDs in web archives of news sites. It functions based on spot signatures that are in favor of natural-language portions of the web pages over advertisements and navigational components of web sites.

Amit Agarwal et al. [1] suggest a technique which mines rules from URLs without considering the contents of web pages for NDD detection. They show that their machine learning technique can generalize rules and achieve reasonable performance at web-scale.

Hajishirzi et al. [41] represent documents as real-valued sparse k-gram vectors, where weights are learned for a specific similarity function (e.g. cosine similarity). NDDs are then detected via this similarity measure. They show that their method can be fine tuned for a particular domain.

Most recently, Qi Zhang et al. [87] have proposed the *PDC-MR* algorithm which detects NDDs through three MapReduce jobs. These three jobs include indexing, sentence duplication detection and sequence matching. Their approach identifies which parts of the documents are duplicates.

2.2 TREC Web Track

The Text REtrieval Conference (TREC) [1] is an annual conference organized by the National Institute of Standards and Technology (NIST) [2]. The purpose of the conference is to "support research within the information retrieval community by providing the infrastructure necessary for large-scale evaluation of retrieval methodologies"[3]. The conference consists of several different *Tracks* with different areas of concentration. The Web Track [4] focuses on exploring and evaluating Web retrieval technologies. Each Track usually consists of a number of tasks where each task targets a very specific research area. The Web Track in both 2009 and 2010 included two major tasks:

[1] `http://trec.nist.gov/`
[2] `http://www.nist.gov/`
[3] `http://trec.nist.gov/overview.html`
[4] `http://plg.uwaterloo.ca/~trecweb/`

- Adhoc Task

  The objective of this task is the examination of the performance of an IR system which is searching a static corpus. This task requires that given a specific previously-unseen query, the IR system returns a ranking of the documents in the collection in order of decreasing probability of relevance. In the Adhoc Task, we assume the probability of relevance of each document is independent of the probability of relevance of the other documents which have appeared before this document in the returned list by the IR system. For this task, the process of returning documents for a specific query should be completely automatic. No human intervention is allowed in any stage of the retrieval. In the judgment process, each returned document will be evaluated as either highly relevant, relevant or not relevant.

- Diversity Task

  The Diversity Task resembles the Adhoc Task with regard to the required returned results. Nonetheless, it will be judged in a different way. In this task, the probability of relevance of a specific document in the returned results is assumed to be dependent on the probability of relevance of the documents that have appeared before it in the list of documents returned by the IR system. The judgment of this task is based on measures that penalize redundancy and reward novelty [28, 22, 27].

2.3 TREC Data Collections

Different Tracks use different data collections for their experiments. The two major datasets that have been recently used by TREC are *ClueWeb09* and *GOV2*. Since these two data sets have been used extensively throughout this thesis, we will describe them in this part.

2.3.1 ClueWeb09

The ClueWeb09[5] dataset has been created by the Language Technologies Institute at Carnegie Mellon University. The targets that the creators of the dataset had in mind are as following [19]:

[5]The name *ClueWeb09* has been chosen by the creators of the collection because the Cluster Exploratory (CluE) program of the U.S. National Science Foundation has provided the resources and funding that was required to collect this data from the web in 2009.

- Approximating the Tier I[6] pages of the web
- Generating a dataset with a good coverage of languages other than English
- Creating a dataset that could be used in TREC and similar scientific explorations by researchers in the next 5-10 years

Several tracks of the TREC 2009 and TREC 2010 have adopted this dataset.

This dataset was crawled by an open source Nutch [7]crawler which had been customized at Carnegie Mellon University.

The languages that this dataset covers and their portion of the dataset are as following [19] : English (50.0%), Chinese (17.0%), Spanish (7.7%), Japanese (5.8%), French (4.2%), German (3.8%), Arabic (3.7%), Portuguese (3.6%), Korean (2.1%), and Italian (2.1%). It is noteworthy that since the creators wanted to capture a good portion of both English and non-English pages, they have dedicated 50.0% of the collection to English and then they have divided the rest of the collection among the aforementioned nine languages proportional to number of Internet users who use web pages in that language. The statistics for this purpose has been obtained from Internet World Stats [8] . It is noteworthy that by reviewing the statistics, we have figured out that the creators of the collection have replaced Russian which is among the top 10 most popular web page languages, with Italian. We could not find a specific explanation for this act in any of the published documents about the collection.

For language identification the *TextCat* [9] language guesser software has been deployed. This language identification software works based on the n-gram-based text categorization algorithm proposed by Cavnar et al.[20].

The entire dataset consists of approximately one billion documents (web pages). The size of this collection is 25 TB uncompressed (5TB compressed). For researchers who are interested in working with a smaller subset of this dataset, the TREC 2009 has named the first approximately

[6]By *Tier I* web pages we mean the pages that have high page rank, significant search or click through activity.
[7]http://lucene.apache.org/nutch/
[8]The statistics that they have used can be found here: `http://www.internetworldstats.com/stats7.htm`
[9]`http://odur.let.rug.nl/vannoord/TextCat/`

```
WARC/0.18
WARC-Type:  response
WARC-Target-URI: http://karaoke.meetup.com/cities/us/tx/deer_park/
WARC-Warcinfo-ID: 9c391157-da1e-4d7b-92c5-afb0744e4972
WARC-Date:  2009-03-65T10:52:49-0800
WARC-Record-ID: <urn:uuid:0bcf9174-6f89-4845-8caa-bb12e3d25456>
WARC-TREC-ID: clueweb09-en0002-00-00000
Content-Type:  application/http;msgtype=response
WARC-Identified-Payload-Type:
Content-Length:  66104
```

Figure 2.1: WARC Header

50 million documents of the English corpus the *Category B* set. [10]

The dataset is organized into directories named $ClueWeb09\_ < language > \_ < segment\# >$ where $< language >$ is the language of pages for segment (e.g. English) and $< segment\# >$ is the segment number. Each of these directories includes approximately 50 million web pages in the form of a set of subdirectories named $< language >< directory\# >$, where $< language >$ is a 2-letter standard language identifier [11], and $< directory\# >$ is the sequence number for that language. Each these subdirectory contains up to 100 files named $< file\# > .warc.gz$ where $< file\# >$ is the sequence number of the file within its directory from "00.warc.gz" up to "99.warc.gz". Each file contains approximately 40,000 web pages in WARC file format . A sample WARC header has been shown in Figure 2.1 [12]. An uncompressed file requires about 1 GB of storage.

[10] For further information about this dataset and the publication describing it you can visit: `http://boston.lti.cs.cmu.edu/Data/clueweb09/`

Another useful site for this purpose is the wiki created for this dataset which is located at : `http://boston.lti.cs.cmu.edu/clueweb09/wiki/tiki-index.php?page=ClueWeb09%20Wiki`

[11] See Language Identifiers at `http://boston.lti.cs.cmu.edu/clueweb09/wiki/tiki-index.php?page=Dataset+Information` .

[12] For details about the WARC format see `http://www.digitalpreservation.gov/formats/fdd/fdd000236.shtml` .

2.3.2 GOV2

The GOV2 [13] collection is a crawl of a large portion of the publicly available *.gov* sites in early 2004. This includes html and text plus extracted text of pdf, word and postscript documents. The dataset includes roughly 25 million documents (approximately 426 GB).

2.4 MapReduce

As mentioned in Section 1.1, any practical solution for finding NDDs needs to be highly scalable. In order to provide this feature, we rely on MapReduce [33] in our work. MapReduce is a framework developed at Google, Inc. for processing large amounts of data through distributed processing. The main two features that this framework incorporates include:

- Parallelism
- Fault-tolerance

Dean and Ghemawat [33] have used the following basic Functional Programming principles for developing MapReduce [51, 9]:

- When a function is applied to a data structure, the data structure does not change, rather the result is stored in a new data structure.
- A function can be used as the argument of another function.

The two major functions that the MapReduce framework is built on include: *Map* and *Reduce* (usually called *fold* in Functional Programming).

Map

`map f lst`

Creates a new list by applying f to each element of the input list; returns output in order. This operation has been depicted in Figure 2.2(Adapted from [51, 9])

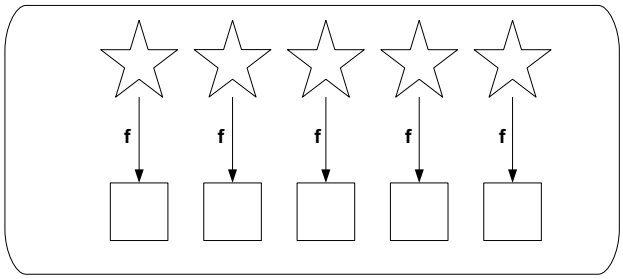


Figure 2.2: Map Operation

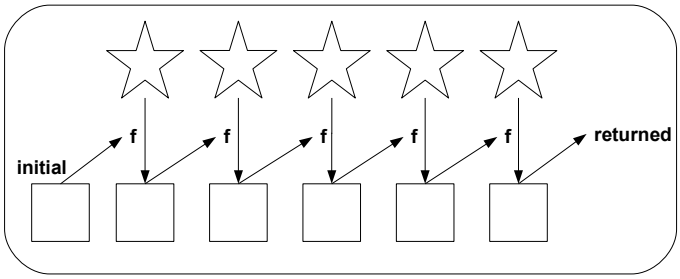


Figure 2.3: Reduce Operation

Reduce (fold)

`fold f` x_0 `lst`

Moves across a list, applying `f` to each element plus an accumulator. `f` returns the next accumulator value, which is combined with the next element of the list. Figure 2.3 describes this operation. (Adapted from [51, 9])

The MapReduce framework [33] reads its input in the form of $(key_i, value_i)$, applies a `map` functions to those pairs and creates the intermediate $(key_m, value_m)$ pairs, and finally -by using a `reduce` function- it merges all the intermediate values associated with the same key.

From a practical standpoint, the typical programmer mostly needs to deal with writing the `map` and the `reduce` functions. Then the MapReduce framework will automatically partition the input data, allocate the necessary distributed resources required for the map and reduce

[13] `http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm`

operations across a set of machines, take care of the possible faults in the system, handle inter-machine communications and produce the output.

MapReduce implementations usually use a cluster of commodity machines. The machines that execute the `map` function are called *mapper*s and the machines in charge of the `reduce` operation are called *reducer*s. Typically, there is a node called the *master* which is in charge of assigning the map and reduce tasks to different machines and checking on their progress by communicating with them periodically. The input data is split into M user defined splits. These splits are then distributed among mapper machines. The mapper performs the map operation and creates the intermediate $(key_m, value_m)$ pairs. The output of the mappers is then partitioned into R local regions. The location of these regions is then conveyed to reducers through the master. At each reducer first the intermediate $(key_m, value_m)$ pairs are sorted and reorganized by a *shuffle* process and then the actual reduce operation is carried out. The reducer writes the output of each reduce task into a different output file. The MapReduce framework provides fault tolerance by means of re-execution of failed tasks. In addition, the same task is usually assigned to multiple machines in order to avoid problems related to a single sluggish machine.[14]

2.4.1 Hadoop

Hadoop[15] is an open source Apache [16] software project for developing a highly scalable and distributed computing systems. The Hadoop project consists of a number of subprojects:

- Hadoop Common
- MapReduce
- Avro
- HBase
- Hive
- Pig
- HDFS
- ZooKeeper
- Chukwa
- Hive
- Mahout

For the purpose of this thesis we are mostly interested in the open source MapReduce project. It

[14] For a more comprehensive description of MapReduce please see the original MapReduce paper by Dean and Ghemawat [33]. In addition, Büttcher et al. have a good explanation and clarifying example on pages 498-503 of their book [18].

[15] `http://hadoop.apache.org/`

[16] `http://www.apache.org/`

is noteworthy that MapReduce is very much dependent on the Hadoop Distributed File System (HDFS) but the ordinary developer does not need to know about the functionality details of HDFS.

2.4.2 Amazon Web Services

Amazon Web Services (AWS) is an infrastructure service which allows users to use Amazon's[17] infrastructure based on their need. It provides plenty of *flexibility* for users to choose their own customized virtual machines, platforms, programming model, etc. In addition, users pay only for what they use. Hence, it is very *cost-effective* in comparison to other similar services for which users have to sign agreements for an extended period of time regardless of their actual usage.

Moreover, AWS allows users to save a lot of time by incorporating pre-installed and pre-configured services (e.g. Hadoop, Apache HTTP Server[18], etc.) in their applications instead of starting from scratch.

In general, AWS includes a variety of services for different purposes [19]. Among those services, the following four have been directly or indirectly used in this project:

- Amazon Elastic MapReduce (EMR)
- Amazon Elastic Compute Cloud (EC2)
- Simple Storage Service (S3)
- Amazon SimpleDB

We will explain in the subsequent sections how these services are related to this thesis.

[17] `http://www.amazon.com/`

[18] `http://httpd.apache.org/`

[19] For a comprehensive list of AWS services see the *Products* tab on `http://aws.amazon.com/`

Elastic MapReduce (EMR)

Amazon Elastic MapReduce (EMR) [20] is the service that provides users with the opportunity to run jobs based on the MapReduce framework [33]. It utilizes Hadoop[21] MapReduce which is an open source implementation of the Google MapReduce under the Apache Software Foundation as explained in Section 2.4.1. Currently, Amazon EMR supports Hadoop 0.18.3 [22].

Aligned with the objectives of the original MapReduce framework [33], Hadoop MapReduce is very suitable for performing data-intensive tasks and is *highly scalable*. That is our main motivation for using EMR as a tool for duplicate document detection.

In summary, in order to run a MapReduce job on Amazon EMR, the user is required to determine the following information:

- Job flow name
- Job flow type (e.g. Custom JAR, Streaming, etc.)
- MapReduce code location on S3 (e.g. JAR Location)
- Input and output locations on S3
- Number of EC2 instances
- Type of instances (e.g. m1.small, m2.xlarge) [23]
- Enable/Disable debugging and if applicable Amazon S3 Log path
- Enable/Disable Hadoop debugging

[20] `http://aws.amazon.com/elasticmapreduce/`

[21] "The name Hadoop is not an acronym; it's a made up name. The project creator, Doug Cutting, explains how the name came about:
The name my kid gave a stuffed elephant. Short, relatively easy to spell and pronounce, meaningless, and not used elsewhere: those are my naming criteria. Kids are good at generating such. Googol is a kid's term."[82]

[22] It is important to note the *version* of the Hadoop software installed on the cluster machines at the time of Java code development. Because the APIs used during development might be inconsistent with the Hadoop software installed on the cluster. Such inconsistencies cause errors that are extremely hard to catch on AWS.

[23] A complete list and description of the EC2 instances can be found here: `http://aws.amazon.com/ec2/instance-types/`

Amazon EMR uses EC2 instances for running the MapReduce jobs. Moreover, it reads the MapReduce code and its corresponding input files from S3. After accomplishing the MapReduce task, it writes the output and the corresponding log files to S3 as well. In addition, its monitoring system depends on SimpleDB. In order to provide a better understanding of the whole task process, we will describe EC2, S3 and SimpleDB in the following sections briefly.

Elastic Compute Cloud (EC2)

Amazon EC2 [24] is the service that allows users to utilize a resizable and elastic computing capacity in the cloud. It provides the user with the opportunity to launch a specified number[25] of virtual machines called Amazon EC2 instances. The instances can have a variety of configurations in terms of computing capacity, memory, storage, etc[26] . In addition, they can be configured according to pre-configured Amazon Machine Instances(AMI) which can contain certain configuration settings, applications, data, etc.

The user will be the `root`/`administrator` (depending on the selected AMI operating system) of the instance and is hence allowed to modify it as they wish. Instances can be `launch`ed or `terminate`d at any point of time based on the request of the user.

Network access and security of the instances is determined by the `Security Group` which needs to be specified at launch time. The security groups can be modified through the web interface of the AWS console. In order to access the EC2 instances, the user can use the predefined `SSH-only` security group at launch time; in addition the user will need a `Key Pair` which can again be acquired through the AWS console. The username required for SSH to ordinary Linux instances would be `root`. It is noteworthy that the username for EMR `master` instances is `hadoop` and `root` will not work for them. For EMR `slave` instances, by default SSH is disabled, but we have modified the `ElasticMapReduce-slave` security group to enable SSH for close monitoring of those instances. `Public DNS` address of the instance, that is provided by the AWS console, is also required for SSH communications.

[24] `http://aws.amazon.com/ec2/`

[25] The standard limit is 20 instances, but for the purpose of this project we contacted AWS and after requesting a use case, they agreed to increase my limit to 200 instances.

[26] For a complete list and description of AWS instances see `http://aws.amazon.com/ec2/instance-types/`

Simple Storage Service (S3)

Amazon S3 [27] allows users to upload, download and store data on Amazon's infrastructure and then use that data across the Internet and with other Amazon Web Services (e.g. EC2, EMR). Its design objective has been providing *high scalability*, *high availability* and *low latency* at commodity costs. In addition, it allows users to charge people who download the data that they make available [3].

Data is stored in S3 in the form of fundamental entities called `objects`. Each object consists of *object data* and *metadaata*. Users can store as many objects as they wish on S3 but each individual object cannot be larger than 5GB [28]. Objects are stored in containers that are called `buckets` by Amazon.Users are required to use an `Access Key` and a `Secret Key` to get full access to their data on S3. It is noteworthy that S3 does not support nested buckets. In other words, one cannot create a bucket within an existing bucket. In order to tackle this problem, many users and applications use naming conventions that include the "/" character in the name of objects within a bucket (e.g. innerbucket1/object1, innerbucket2/object2). That works fine as long as the application using the data is aware of the situation and has the means to handle it. Unfortunately, most other Amazon Web Services like EMR do not normally support object names that include the "/" character.

In this thesis we need to transfer the input data and the MapReduce code to S3 and eventually the output data will become available for retrieval on S3 as well. Due to this significant amount of interaction with S3, it is important to find the appropriate tool for data transfer to and from S3. In order to do so, I have examined the following three tools:

- S3Fox:

  This is a Firefox add-on[29]. Hence it can be used on any platform on which Firefox can be installed. It has the capability to download and upload from and to S3. Its interface resembles the common dual-pane FTP clients. Therefore, it is a good tool to start with. Nonetheless, when I used it for large data transfers over an extended period of time, I occasionally faced

[27] `http://aws.amazon.com/s3/`

[28] In December 2010 Amazon increased this limit to 5TB

[29] `https://addons.mozilla.org/en-US/firefox/addon/3247`

some failed transfers without being able to determine the cause. Some of the other features of S3Fox include support for: secure transfers (HTTPS), AWS import/export, multiple S3 accounts and synchronization of local and S3 folders.

- `aws` Command-line Tool:

  This is a command line tool developed by Timothy Kay[30]. In addition to S3 tools it provides an extensive command set for dealing with EC2 instances. This tool is very popular in the Amazon Developer Community[31]. It works on both Linux and Windows platforms. After installation, it provides a set of commands (e.g. `s3mkdir`, `s3put`, `s3ls`, etc.) which will facilitate data transfer to S3. Although, it is a rather stable and reliable tool, it does not provide any statistics about the progress and rate of transfer. Hence, it might not be very suitable for determining slow and probably unsuccessful transfers. Nevertheless, it is a very useful tool for dealing with S3 and EC2 in the same environment.

- `s3cmd` Linux Command:

  `s3cmd` is a command-line tool for managing data on S3. It is included in the latest versions of all the major Linux distributions. For older Linux versions and distributions, it is available through the standard update commands (e.g. `yum`(Fedora family), `apt-get` (Ubuntu family)). After installation it can be configured by running the `s3cmd --configure` command. This tool has the capability to protect your files from reading by unauthorized persons while in transfer to S3 by an encrypted password. It also supports HTTPS. It is noteworthy that HTTPS is slower than plain HTTP and cannot be used if you are behind a proxy. `s3cmd` also can operate from behind a proxy server. Overall, this is a very stable and reliable tool for Linux environments. It is very flexible and can be smoothly integrated into Bash or Perl scripts. Moreover, it shows the progress and transfer rate of the transfer in almost real-time. It also automatically retries the failed transfers up to three times. We found this tool more helpful than the other tools and used it throughout our work with AWS.

[30] `http://timkay.com/aws/`

[31] `http://developer.amazonwebservices.com/`

SimpleDB

Amazon SimpleDB [32] is a non-relational data store that has been designed by Amazon for running real-time queries on structured data. This service has been designed to work closely with EC2 and S3. In addition, it provides high availability and scalability[4].

In order to enable the Hadoop Debugging Service on EMR, it is required that the user activates this service on AWS. Therefore, for the purpose of this thesis, we have activated this service on our AWS account in order to become capable of activating Hadoop Debugging during the course of our experiments. Other than that, this service has not been directly used by us in our experiments.

2.4.3 Code Development and Karmasphere

For developing MapReduce code that is executable on Amazon EMR, one can use a variety of languages (e.g C++, Java, Perl, Hive, etc.). We first started to develop C++ code for our experiments, but later, due to some modifications that needed to make on Hadoop classes, we switched to Java. Hence, the final code developed for the experiments is in Java.

During our experience with C++, we had to compile our code on an arbitrary machine and then upload the compiled version to S3 for deployment on EMR. In this process, it is crucial to check the computer architecture of EC2 instance platform that will be used for code deployment and match it with the platform that we use for compiling the code. For instance, if we are planning to deploy our code on 32-bit EC2 instances (e.g. m1.small), we should compile the code on a similar 32-bit platform otherwise the MapReduce code will not work properly at run time. The same story holds for 64-bit platforms. This error would be extremely hard to catch since the error report of EMR will not include anything relevant to the aforementioned problem. In addition, this issue has not been addressed in Amazon documents to the best of our knowledge.

For MapReduce code development with Java, any Java development tool capable of creating a JAR file would be suitable. But then, we would have to upload the JAR file to S3 manually and then use the AWS `console` to run the MapReduce Job. This process will usually repeat many times during debugging of the code and as a result will become very mundane. There are certain

[32] `http://aws.amazon.com/simpledb/`

Integrated Development Environments (IDEs) that are capable of deploying the MapReduce code on Amazon EMR automatically. *Karmasphere Studio* is one of them.

Karmasphere Studio[33] for Hadoop is a MapReduce development environment which has been built on top of NetBeans[34]. It has a number of very useful features which would facilitate MapReduce job deployments on Amazon EMR:

- It allows the user to set up the EMR cluster properties (e.g. number and type of EC2 instances) and job properties (e.g. local JAR path) once and for all. Therefore, the user will not need to enter those pieces of information before each run at the time of debugging.
- It lets the user to prototype the MapReduce job locally without the need of a real cluster. In our personal experience, this prototyping feature works only for very small input files and for very straight forward MapReduce task. With the structure of our large input files we were not able to take advantage of this feature very much. Nonetheless, it is a perfect tool for demonstrating MapReduce operation for benchmark examples like the Word Count example suggested in [33].
- It provides a monitoring console very similar to the web-based AWS console provided by Amazon. Therefore, you will not need to login to the web-based console to monitor your job. This capability makes Karmasphere rather self-sufficient.

2.4.4 Monitoring Tools

In general, when we run EMR jobs there are two levels of monitoring required. One is the monitoring of the whole MapReduce operation on Hadoop (e.g. the progress of Mappers); the other is monitoring individual EC2 instances(e.g. their memory usage). There are two ways for performing the monitoring in these two levels:

- Using the web-based console
  AWS gives us the opportunity to monitor both the MapReduce operation and each individual EC2 instance. The problem with this approach is that the web-based monitoring system

[33] `http://www.karmasphere.com/products/`
[34] *NetBeans* is an IDE developed by Sun Microsystems(owned by Oracle): `http://netbeans.org/`

is time delayed and at the same time it usually takes a couple of minutes before the Hadoop MapReduce monitor starts showing information to the user. We have experienced delays, sometimes up to 25 minutes, before we could see the first traces of monitoring information. Moreover, by using the Amazon EC2 monitoring for individual instances, you are basically paying for information that you can get for free by using `SSH`.

- Using command-line tools

  The best monitoring tool that we could find for the MapReduce operation is accessible by connecting to the Master node via `SSH` and then running the following command:
  `lynx http://localhost:9100/`

  This will give you access to an interface with the capability to minor the most detailed aspects of the entire MapReduce operation (e.g. the amount of bytes that each Mapper has temporarily written on local disk). The data is almost real-time. The only drawback is that `lynx` does not update the page automatically; so, the user will need to hit `Ctrl+R` in order to see the most recent version of the monitoring data. In addition, for diagnosing MapReduce Step failures, the user can check the files under `/mnt/var/log/hadoop/steps` on the Master node. This log file can be very helpful in finding general Step faults (e.g. S3 connection problems, JAR file problem).

  Individual EC2 instances can be monitored by connecting to them via `SSH` and then running usual Linux monitoring tools (e.g. `top`). They only issue that requires special attention is that, EC2 instance operating as `slave` nodes are not accessible by default EMR configuration. The user needs to modify the corresponding security group and open the `SSH port` as described before.

2.4.5 MapReduce Operations in Hadoop

In this part, we will briefly explain how the MapReduce operations suggested in [33] are actually carried out in Hadoop and how we have modified them for the purpose of this project. The explanation focuses on the details required for MapReduce code development.

As shown in Figure 2.4, first the input file set is split into several smaller pieces called `FileSplit`s. After splitting the files, Hadoop processes the `FileSplit`s according to the `RecordReader`

it gets from the specified `InputFormat`. The `RecordReader` is the module which determines how each individual `FileSplit` is read by Mappers.

Due to the structure of the input files in our experiments, the input file set cannot be split arbitrarily by Hadoop. Hence, we have overridden the method in charge of splitting the input files to avoid splitting. Moreover, since the standard `InputFormat`s (e.g. `TextInputFormat`) offered by Hadoop APIs cannot distinguish WARC documents within a WARC file, we have developed the `WARCInputFormat` which not only does not split the input files but also is capable of extracting the WARC documents from the WARC input files and passing them to Mappers one by one. It is noteworthy that since the WARC input files are each over 1GB in size and include roughly 35000 WARC documents, seek time will be unfavorably affected. Nevertheless, this is the most effective way that we could find to perform this job overall.

As it is not necessary for the `InputFormat` to generate both meaningful *keys* and *values*, the `WARCInputFormat` returns *null* (`NullWritable`)[35] as its key and a WARC document as its value.

The key-value pairs generated by the `RecordReader` are then passed to the Mapper. The Mapper performs whatever operation it is supposed to perform on the input pair. Then it calls the `OutputCollector.collect` with the output key-value pair. It is noteworthy that all the output keys should have the same type. Likewise, the output values should all be from the same type. This is due the fact that the Map output is written into a `SequenceFile` which has per-file type information; hence all the records must be from the same type. The Map output is then partitioned by a `Partitioner`. In this thesis, the default `HashPartitioner` has been used which utilizes the hash code function on the key of the output pairs.

When Reduce tasks start, their input is scattered across all the map nodes. Therefore, first they must be copied to the local file system of the Reducer through a *copy phase*. Then all the gathered files will be appended into one file in an *append phase*. Afterwards, in a *sort phase* all the pairs with the same key will become contiguous. This will facilitate the reduce operation. Then the file will be read sequentially by the Reducer through an `Iterator`. One output file will be created for each executed reduce task.[36] (This part has been adapted for this thesis based on [82] and [5])

[35] This is a Hadoop variable type with no data.

[36] Since a *Combiner* has not been used in this project, it has not been mentioned here.

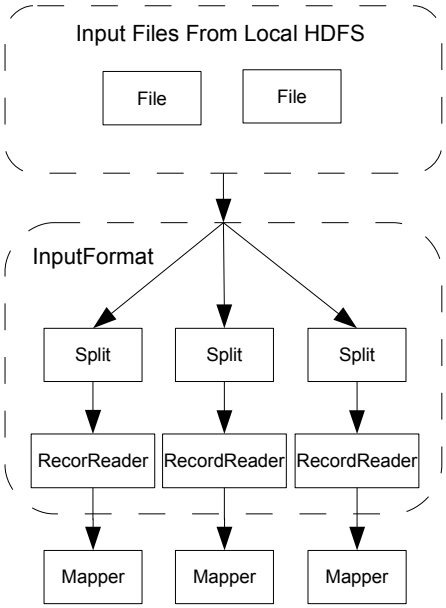


Figure 2.4: InputFormat

In the following subsections, we will describe the cluster configuration and optimizations that we have utilized in the experiments. Furthermore, we will provide some statistics about the initialization of MapReduce tasks on Amazon EMR.

2.4.6 Cluster Configuration

For experiments dealing with a few gigabytes of data, a small cluster of 1 Master node and 2-3 Slave nodes has been used. The Master node is a `c1.medium` Amazon EC2 instance while the Slave nodes are `m2.4xlarge` instances.

For experiment dealing with large amounts of data , a much larger cluster has been used. This cluster consists of 1 `m1.xlarge` instance with 4 virtual cores as the Master node and up to 80 `m2.4xlarge` instances with 8 virtual cores on each node as Slave nodes.

In order to configure the aforementioned Amazon EMR clusters, the Hadoop configuration API has been used. Different components in Hadoop can be configured by this API. An instance of the `JobConf` class represents the configuration properties and the values that are used for the

MapReduce operations on the Hadoop cluster. Following are some of the major properties that have been modified from their default values for the purpose of the experiments in this project.

- `mapred.task.timeout`

  The value of this property represents the number of milliseconds before tasks (e.g. a specific Mapper in a MapReduce operation) will be terminated if it does not update its status string. The default value for this property is 10 minutes. It is very important for the MapReduce code developer to have a rough estimate of the timing of the Map and Reduce tasks; because if you set this value too low, the task (e.g. the Mapper or the Reducer) can be killed before accomplishing its mission, and if you set this value too high if something goes wrong, it will take a long time before the Master kills that task. This can lead to an increase of operational costs on cloud computing services like Amazon EMR where you pay per hour of usage. The other important point which needs to be taken into account is that a few Reducers, will start operating way before the Map operation is completely over. Since they should wait for the Mappers to be completely done before they could report that they themselves are done, those Reducers will operate much longer than the other reducers. This fact should be considered when setting this threshold. For MapReduce# 1 explained in Section 3.1.1, this value needs to bet set to higher than default values while for the other MapReduce operation the default value will work fine.

- `mapred.child.java.opts`

  This property specifies the Java options for the task tracker child processes. By using this property, we can set the maximum heap size that each JVM [37] can use. Depending on the Java code, increasing the heap space might be very helpful. For our experiment, we have set this property equal to `-Xmx3500m` which means that each JVM has up to 3.5GB of heap space to use. It is important to note that the maximum heap space multiplied by the number of virtual cores on each virtual node should be less than total amount of RAM available for that virtual node, otherwise JVMs will not be launched and the MapReduce task will fail.

- `io.sort.mb`

[37] JVM stands for Java Virtual Machine . For details see [56].

The value of this property indicates the total amount of buffer memory that will be used for sorting files in megabytes. If the buffer exceeds a certain percentage of this amount (by default 80%), it will start spilling to temporary files on disk. This phenomenon will make the whole MapReduce process tremendously slow. It is important to set this property appropriately to avoid excessive spilling. I have set this value to 1500.

- `fs.inmemory.size.mb`

  This is the maximum amount of memory (in MB) allocated for the in-memory file-system which is used to merge the output of Mappers at the Reduces. I have set it to 1600.

- `io.sort.factor`

  This property specifies the number of streams that are merged at once while sorting files. We have set this property to 100. It should be noted that this value also indicates the number of open file handles. Therefore it should not be set to a very large number, otherwise it will affect the performance of the system.

- `io.file.buffer.size`

  This is the size of the read and write buffers that are used for `SequenceFiles` usually used for writing the out of Mappers. I have set it to 131072.

- `tasktracker.http.threads`

  In order to set the number of worker threads used for map output fetching through the http server this property should be used. I have set this value to 50.

- `mapred.reduce.parallel.copies`

  This is the number of parallel transfers run by reduce during the copy phase. This property has been set to 50 for our experiments.

- `mapred.output.compress`

  This property indicates whether the final output should be compressed. We have set its value to `true`. Generally speaking, when working with large data sets like ClueWeb09, it is a good idea to work with compressed files because the transfer and storage rates will decrease. This comes at the price for CPU usage at the time of writing final outputs.

- `mapred.compress.map.output`

  It will indicate whether the intermediate key-value pairs that are generated by Mappers should be compressed. This will be helpful when the intermediate records are large in size, otherwise it will hinder the MapReduce process. Therefore, for MapReduce# 1 (see Section 3.1.1) we have set it to `true` and for the other MapReduce operation we have set it to `false`.

We have made these configurations based on the recommendation of [82], the recommended configuration for the sort1400[38] benchmark, and trial and error in our experiments.

2.4.7 Cluster Initialization

Usually from the time that AWS receives our request to launch a MapReduce task until the MapReduce task actually starts, there is a few minutes of gap. This is the amount of time that it takes AWS to lunch the required EC2 instances and start the Hadoop cluster. Figure 2.5 shows a snapshot of the length of this gap during different times of the day. We have recorded this time for 185 different experiments in April and May 2010, on average this process takes 3 minutes and 12 seconds for standard Amazon EMR jobs with less than or equal to 20 instances (standard limit).

2.5 Summary

In this chapter, we first described some of the major works done by various researchers for the purpose of duplicate document detection. Then we described TREC Web Track, its tasks and two of the data collections that have been used by TREC in the recent years, namely ClueWeb09 and GOV2.

Since the NDD detection algorithm which will be proposed in the next Chapter is based on MapReduce, we described the MapReduce framework and Hadoop which is its open source implementation.

[38] sort of 14TB of data on 1400 nodes

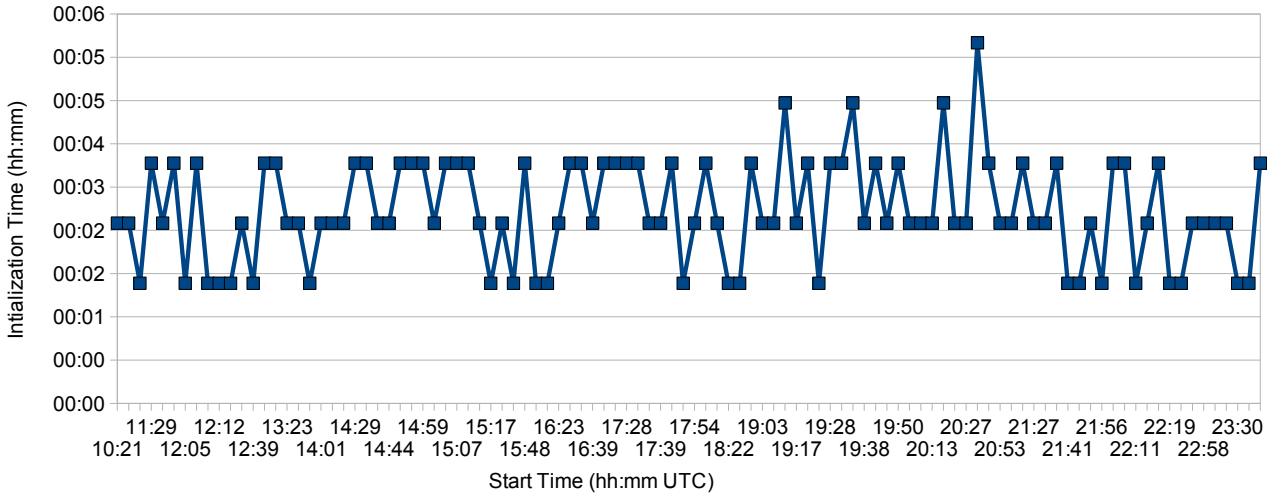


Figure 2.5: Cluster Initialization Time

In addition, we described the Amazon Web Services which we used for our experiments in this thesis. The descriptions include detailed practical aspects and challenges that were faced during the course of this thesis.

Chapter 3

Near-Duplicate Document Detection

3.1 Near-Duplicate Document Detection

Like many implementations, our NDD implementation is based on the shingling technique proposed by Broder et al. [12, 13, 17, 16, 14, 15], which suggests that in order to determine whether two documents d_1 and d_2 are syntactically near-duplicates, the following steps need to be taken:

- Since we are interested in investigating the partial similarity between documents, first the string representing each of the documents (e.g. d_1 and d_2) should be broken into several substrings. In order to do so, we pass the documents through a normalizing filter which basically removes all the formatting, style, punctuation, capitalization, HTML tags, etc. from the documents. Then we divide the normalized text into contiguous chunks of text with a specific length. These chunks are called *shingles*. At this point each document will be represented by a set of shingles.

- Second, the set of shingles representing each of the documents is converted to a set of fingerprints of that document by using a fingerprinting function. This function should satisfy the following two properties:

$$f(\alpha) \neq f(\beta) \Longrightarrow \alpha \neq \beta \tag{3.1}$$

$$Probability((f(\alpha) = f(\beta))|(\alpha \neq \beta)) << 1 \tag{3.2}$$

A good choice for such a fingerprinting function is a near collision-free hash function. One choice for such a hash function are the hash functions that are commonly used in cryptography (e.g. MD5, SHA, etc). The problem with this category of hash functions is that they are computationally expensive and relatively slow. Rabin's fingerprinting method [69, 12] is a very appropriate choice because it satisfies both of the aforementioned properties and is very fast and inexpensive in terms of its required calculations. By applying Rabin's fingerprinting method to the shingle set of each of the documents (e.g d_1 and d_2), we will get the set of fingerprints of those documents (namely d_{f1} and d_{f2}). These sets are called *fingerprints* of the original documents d_1 and d_2.

- Third, the resemblance rate (RR) between d_1 and d_2 is defined by the Jaccard similarity coefficient:

$$RR(d_1, d_2) = \frac{|d_{f1} \cap d_{f2}|}{|d_{f1} \cup d_{f2}|} \tag{3.3}$$

RR is a number between 0 and 1. When this value is close to 1 it means that the documents are roughly the same and when it is close to 0 it means that the two documents are quite distinct.

Despite all the ambiguities and disagreements among researchers with regard to the appropriate NDD detection algorithm, there is a consensus on the fact that with the current growth in the size of data collections and the Web, any practical algorithm needs to be *highly scalable*. This is the cause which has motivated us to use the Elastic MapReduce service of Amazon Web Services (AWS) in order to develop an NDD detection solution based on the MapReduce framework [33].

3.1.1 The MapReduce Solution

Our solution exploits the MapReduce framework [33] in order to calculate the resemblance rate for each pair of documents in a specific collection of web documents. The overview of the whole operation is depicted in Figure 3.1. It consists of two Map-Reduce operations.

The first MapReduce operation is in charge of determining the pairs of documents that have a specific shingle in common. The second MapReduce operation calculates the resemblance rate (RR) for all pairs of documents with at least one shingle in common based on Equation 3.3. Following is a more detailed explanation about these two MapReduce operations.

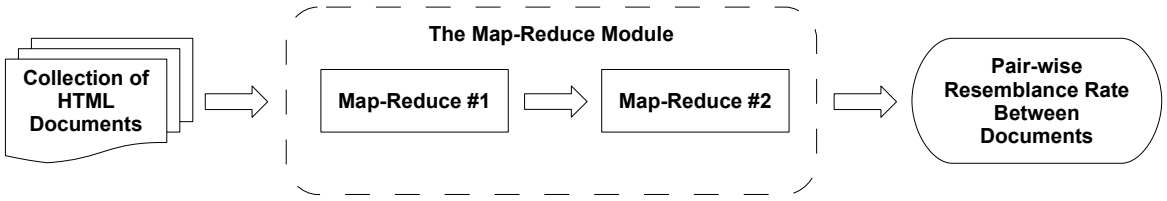


Figure 3.1: Overview of The MapReduce Operation

Map-Reduce#1

In Map-Reduce#1, the Mapper reads the input files. The input files are assumed to include thousands of HTML documents where each document has a unique identifier (ID) in its header.For each document in each of the input files it performs the following actions:

- It extracts the ID of the document from the additional header and then removes the additional header.
- It normalizes the remaining HTML document by converting all its characters to lower case and then removing the following items:
  - HTML header
  - scripts (e.g. $< script.../script >$)
  - styles (e.g. $< style.../style >$)
  - tags (e.g. $< head >$)
  - special characters (e.g. $\ $)
  - end of line and space characters (e.g. \n, \t)
  - major punctuation marks (e.g. :, ;, !)
- The normalized document is then split into many shingles. The shingles start at the beginning of each word and are n[1] characters long. Hence, n is the length of the shingles. It is noteworthy that Hung-Chi Chang et al. [21] have shown that if we choose a small n we will gain a higher recall but at the same time the computational expense of the process

[1] in our experiments $n = 64$ unless stated otherwise.

will increase because there will be much more pairs of documents with that small shingle in common. Furthermore, they have shown that a larger n will lead to higher precision but less robustness. Hence it is important to choose a number which is neither too small nor too large.

- It then employs the Rabin hash function [69, 12] to convert each of the shingles to a hash value.
- Finally, it emits a $< key, value >$ pair where the *key* is *shingle.hash* (Rabin hash of the shingle) and the *value* is *DocID-size* (documents ID appended by the number of shingle hashes that exist in that document).

According to the MapReduce framework conventions [33], then all the pairs with the same key, namely *shingle.hash*, will go to the same Reducer. The Reducer will then perform as following:

- For each key *shingle.hash*, it will create the set of all its values.
  (e.g. $\{DocID_a - size_a, DocID_b - size_b, ...\}$).
- It then calculates the 2-subsets of this set.
  (e.g. $\{DocID_a - size_a, DocID_b - size_b\}$, ...).
- Finally for each of the 2-subsets it emits an output with appended IDs as the key and 1 as the value (e.g. $< DocID_a - size_a : DocID_b - size_b, 1 >$).

Conceptually, this means that for each hash value, Map-Reduce#1 will generate the set of all document pairs that have that specific shingle in common. The pseudo-code of the operation performed by Map-Reduce#1 can be found in Figure 3.2.

Map-Reduce#2

This MapReduce task is in charge of calculating the actual resemblance rate between the pairs of documents. It receives the output of MapReduce #1 as the input of the Mapper.

The Mapper outputs the pairs as they are. The MapReduce framework will then send all the equal pairs to the same Reducer. At this stage, each pair means a common hash between the documents that its key consists of.

```
1: class MAPPER
2:   method MAP(Null, Collection Files)
3:     for all Files of the Collection do
6:       for each Document of the File do
7:         ▷ Divide the content of the document into shingles
8:         ▷ Hash the shingles
9:         for all shingle ∈ Document do
10:          ▷ Emit hash value of shingles
11:          EMIT (shingle.hash,DocID-size)
```

```
1: class REDUCER
2:   method REDUCE (shingle.hash,DocID-size)
3:     for all shingle.hash ∈ Document do
4:       ▷ Emit document pairs that include the shingle
5:       EMIT (DocID$_a$-Size$_a$:DocID$_b$-Size$_b$, 1)
```

Figure 3.2: Overview of MapReduce #1

The Reducer then performs the following operations:

- Counts the total number of

$$< DocID_a - Size_a : DocID_b - Size_b, 1 > \tag{3.4}$$

  pairs. This count would be the total number of hashes that these two documents have in common.

- Extracts the size of documents from the key-value pair 3.4.

- Calculates the resemblance rate according to Equation 3.3 as explained in Section. 3.1.

- Outputs

$$< DocID_a - Size_a : DocID_b - Size_b, resemblance(DocID_a, DocID_b) > \tag{3.5}$$

The pseudo-code of this operation can be found in Figure 3.3.

```
1: class REDUCER
2:   method REDUCE (DocID_a-Size_a:DocID_b-Size_b,1)
3:     for each DocID_a-Size_a:DocID_b-Size_b do
4:       ▷ Count the number of common shingles for a document pair
5:       Count (DocID_a-Size_a:DocID_b-Size_b,1)
6:       ▷ Emit resemblance rate for document pairs with common shingles
7:       EMIT (DocID_a-Size_a:DocID_b-Size_b, α)
```

Figure 3.3: Overview of MapReduce #2

3.2 Validation

In order to validate our implementation, we re-ran one of the major experiments conducted by Bernstein and Zobel [6]. They explore syntactic techniques (e.g. document fingerprinting) for detecting content similarity. By applying their technique on the GOV2 corpus, they reported a high degree of redundancy. Furthermore, they have conducted a user study to confirm that their metrics were accurately identifying resemblance of content.

Bernstein and Zobel [6] report that 16.6% of all relevant documents in the runs submitted to TREC 2004 terabyte track were redundant. In this section of the paper, we will apply our NDD algorithm to the same data.

3.2.1 TREC 2004 Terabyte Track

The main task in the terabyte track of TREC 2004 was an adhoc retrieval task. Participants submitted search results for a list of specific topics. The search was conducted over the GOV2 collection[2]. GOV2 is a TREC test collection which includes 25 million documents and is 426GB in size. It is a crawl of `.gov` sites conducted in early 2004. Bernstein and Zobel used the relevant documents[3] in these runs for their experiments, and we use the same set of documents for our validation experiments.

[2] `ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm`

[3] `trec.nist.gov/data/terabyte/04/04.qrels.12-Nov-04`

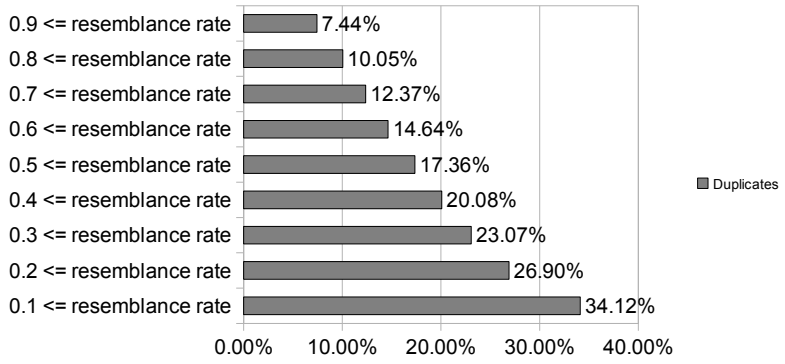


Figure 3.4: NDDs in Relevant Documents of TREC 2004 Terabyte Track

3.2.2 Experiment

For this experiment we choose a shingle size of 64 bytes. A new shingles is created at the beginning of each word. We consider any two documents with an RR[4] greater than or equal to 0.5 as NDDs.

The results for this experiment are shown in Figure 3.4. The x-axis in this figure represents the cumulative percentage of NDDs and the y-axis represents the resemblance rate. For instance, the first bar from the top indicates that 7.44% of the relevant documents submitted to TREC 2004 Terabyte Track are near-duplicates of some other documents with a resemblance rate of greater than or equal to 0.9 .

As we can see, 17.36% of the documents are detected as NDDs with respect to the aforementioned NDD definition, namely documents with $RR \geq 0.5$. This is very close to the 16.6% that Bernstein and Zobel report [6]. However, this is only a rough estimate because some of the documents detected by our algorithm might not have been detected as redundant documents in their approach and vice versa. We contacted them to obtain the list of the documents that they have considered redundant for a more precise comparison. Unfortunately they were not able to provide that data.

[4]As defined in Section 3.1.

3.3 Enhancements

The experiment in Section 3.2.2 indicates that our algorithm provides reasonable performance. Nonetheless, since we create a shingle at the start of each word, for a large text collection we will essentially have to deal with billions of shingles. These shingles will generate a very large set of $< key, value >$ pairs in the MapReduce process. This phenomenon will hinder the whole process significantly, particularly MapReduce#1 described in Section 3.1.1.

In order to tackle this issue, we examine the following two categories of shingle sampling techniques:

- Hash-value-based
- Threshold-based

In the subsequent sections we explain these techniques in detail. Furthermore, we study how the application of these two techniques affect the quality of the results. For this purpose we use four different quantitative measures namely average error, correlation, recall and precision.

3.3.1 Hash-value-based Shingle Sampling

As mentioned before, the main purpose of sampling shingles is reducing the total number of shingles that we need to deal with in the whole MapReduce operation described in Section 3.1.1. The shingles created by MapReduce#1 (described in Figure 3.2) can be eliminated by applying the `mod` operator on their Rabin hash values, as suggested by Broder et al. [14, 17]. In order to investigate how this technique affects our MapReduce implementation, we run an experiment in which the `Mapper` in MapReduce#1 eliminates a certain amount of the shingles by using the `mod` operator.

Table 3.1 summarizes the results of this experiment. It describes how the percentage of documents with different resemblance rates (RRs) varies by keeping a specific percentage of shingles mentioned in the first row. In order to generate the data mentioned in this table, we ran a series of experiments. First, we kept all the shingles and calculated the RR measure based on that (the 100% column). Then, in the subsequent experiments we kept only a specific portion

Table 3.1: Impact of Hash-value-based Shingle Sampling on NDD Detection

∼ % of Shingles Kept	100%	50%	25%	12.5%	6.25%	3.13%	1.56%	0.78%	0.39%	0.20%
0.9 ≤ RR	7.44%	7.44%	7.42%	7.53%	7.67%	7.79%	8.08%	8.42%	9.00%	8.51%
0.8 ≤ RR	10.05%	10.09%	10.13%	10.14%	10.37%	10.54%	10.92%	10.78%	10.66%	9.59%
0.7 ≤ RR	12.37%	12.43%	12.49%	12.80%	12.75%	12.81%	13.26%	12.75%	12.38%	10.73%
0.6 ≤ RR	14.64%	14.67%	14.71%	14.97%	14.92%	15.15%	15.54%	15.01%	14.74%	12.41%
0.5 ≤ RR	17.36%	17.30%	17.37%	17.53%	17.80%	17.88%	18.03%	17.28%	16.57%	13.83%
0.4 ≤ RR	20.08%	20.07%	20.19%	20.42%	20.38%	20.38%	20.35%	19.37%	18.16%	14.73%
0.3 ≤ RR	23.07%	23.06%	23.10%	23.16%	23.09%	23.35%	23.23%	21.76%	19.94%	15.76%
0.2 ≤ RR	26.90%	26.81%	26.84%	26.87%	26.98%	27.14%	26.92%	24.97%	22.16%	17.24%
0.1 ≤ RR	34.12%	34.05%	34.22%	34.38%	34.20%	34.67%	33.55%	29.84%	25.42%	19.57%

(i.e 50%, 25%, etc) of the whole shingles by eliminating the rest of the shingles based on their hash values. Then we calculated the RR values based on the portion of the shingles that were kept (i.e 50% column, 25% column, etc.). The values in the table cells, represent the percentage [5] of NDDs that were detected by considering a specific threshold for NDD definition (i.e. $0.9 \leq$ RR, etc.) and a specific portion of the shingles (i.e. 100%).

In order to see the trends in Table3.1, we have visualized it in Figure 3.5. Interestingly, we see that almost the same results that we get by considering 100% of the shingles can be achieved by considering only 1.56% of the shingles. Even by retaining only 0.78% of the total number of shingles, the percentage of NDDs detected (i.e. $0.5 <= RR$) decreases by only 0.08%.

In terms of processing time, the whole MapReduce operation (including MapReduce#1 and MapReduce#2) took 19 minutes when we were considering 100% of the shingles. This time includes two Amazon cluster initializations discussed in Section 2.4.7. Hence, we can say that the whole process itself took roughly 12.5 minutes. This amount of time decreased to only 1.5 minutes when we considered only 1.56% of the shingles.

Thus far, we showed that the *amount* of detected NDDs by considering only a small portion of the shingles will be almost the same as when we do not eliminate any shingles. But are these two sets of NDDs the same NDDs? This is the question that we will answer next.

In order to make sure that the NDDs detected by considering only a specific percentage of

[5] In terms of the total number of relevant documents in the runs submitted to TREC 2004 terabyte track

shingles are the same as the NDDs detected without shingle elimination, we consider the following four quantitative measures:

- **Average error** is defined as

$$\frac{\sum_{i=1}^{n} |\alpha_i - \beta_i|}{n} \tag{3.6}$$

where α_i is the RR calculated for the i^{th} pair of documents based on considering all their shingles and β_i is the RR for the same pair when only a sampled set of shingles is used for RR calculation. n is the total number of document pairs. Figure 3.6(a) and Table 3.2 both show that the smaller the percentage of shingles we use for NDD detection the higher the error rate. This is what we intuitively expect. As we can see by considering only 1.56% of the total number of shingles we have an error rate of 0.1053, which may be acceptable in terabyte and petabyte data collections.

- **Correlation** is calculated based on the following formula:

$$\frac{n \sum \alpha_i.\beta_i - \sum \alpha_i \sum \beta_i}{\sqrt{n \sum \alpha_i^2 - (\sum \alpha_i)^2}\sqrt{n \sum \beta_i^2 - (\sum \beta_i)^2}} \tag{3.7}$$

where α_i and β_i are the same values defined in the previous section for average error. A correlation of +1 is ideal and describes the case of a perfect linear relationship. As we can see in 3.6(b) and Table 3.2, correlation drops significantly when we keep less than 1.56% of the shingles.

- **Recall** is defined as

$$\frac{|Dup \cap Res|}{|Dup|} \tag{3.8}$$

where *Res* represents the set of NDDs that are detected by the current sampling method while *Dup* represents the set of documents that we detected in Section 3.2.2 as NDDs. In other words, we consider the results of Section 3.2.2 as the ground truth and we compare the results of the sampling experiments with it.

- **Precision** is defined as

$$\frac{|Dup \cap Res|}{|Res|} \tag{3.9}$$

Dup and *Res* are as defined for recall. Again it is noteworthy that we are calculating the precision in comparison the experiment where we do not eliminate any shingles. In other

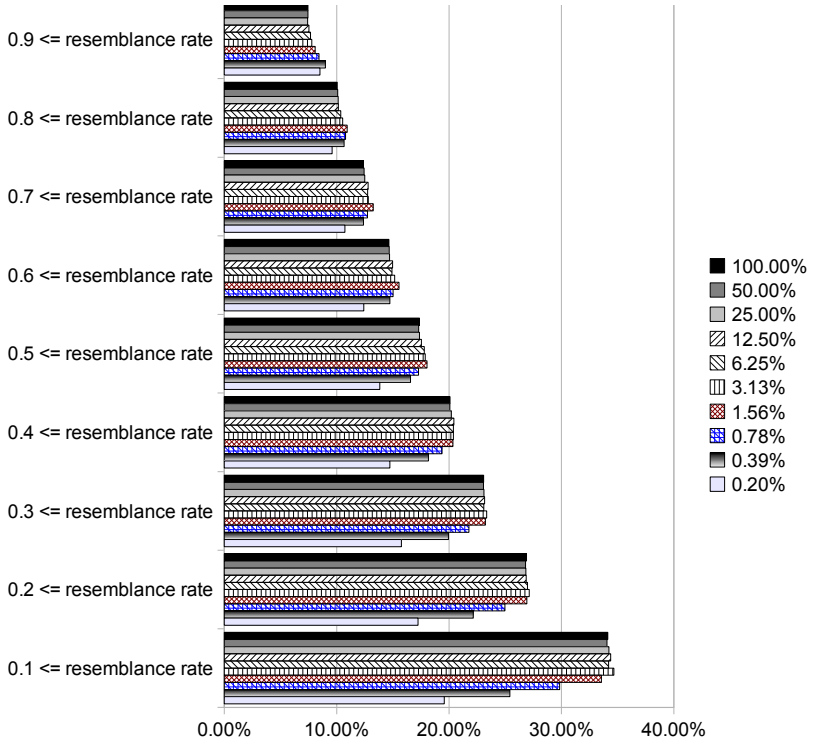


Figure 3.5: Impact of Hash-value-based Shingle Sampling on Duplicate Document Detection

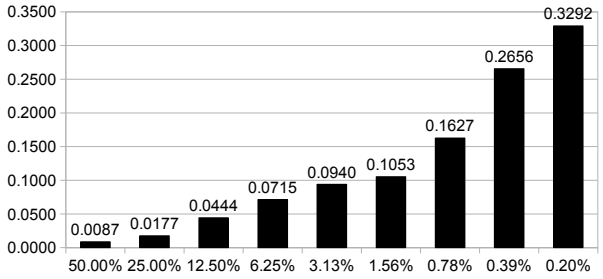
words, we consider the results of Section 3.2.2 as the ground truth and we compare the results of the sampling experiments with it.

The value of these four quantitative measures of performance have been calculated for all the hash-value-based shingle sampling rates (i.e. 50%, 25%, etc.) and are included in Table 3.2.

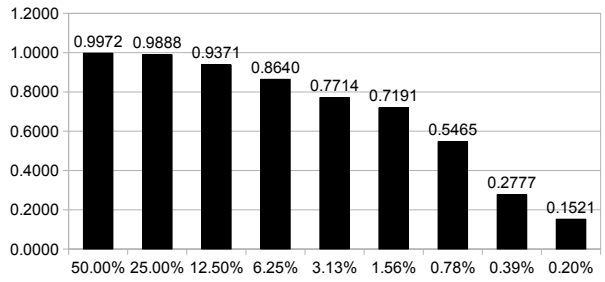
As we can see, hash-value-based shingle sampling adversely affects the performance of NDD detection algorithm. Nevertheless, even by keeping only 1.56% of the shingles, we gain a reasonably good performance. The trends have been depicted in Figure 3.6.

3.3.2 Threshold-based Sampling

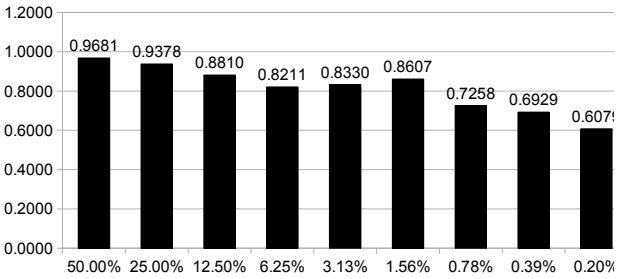
Another way of reducing the number of shingles, that we need to deal with during the MapReduce operation described in Section 3.1.1 , is threshold-based sampling. Very common shingles that are repeated more than a specific threshold in the whole set of shingles may not be useful when it



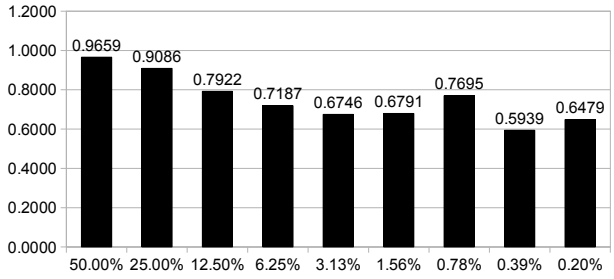
(a) Average Error



(b) Correlation



(c) Recall



(d) Precision

Figure 3.6: *Average Error*, *Correlation*, *Recall* and *Precision* of Hash-value-based Sampling

Table 3.2: Performance Analysis of Hash-value-based Shingle Sampling

∼ % of Shingles Kept	Average Error	Correlation	Recall	Precision
50.00%	0.0087	0.9972	0.9681	0.9659
25.00%	0.0177	0.9888	0.9378	0.9086
12.50%	0.0444	0.9371	0.8810	0.7922
6.25%	0.0715	0.8640	0.8211	0.7187
3.13%	0.0940	0.7714	0.8330	0.6746
1.56%	0.1053	0.7191	0.8607	0.6791
0.78%	0.1627	0.5465	0.7258	0.7695
0.39%	0.2656	0.2777	0.6929	0.5939
0.20%	0.3292	0.1521	0.6079	0.6479

comes to detecting near-duplicate content . These shingles are usually standard phrases (e.g policy statements, boilerplate, etc.)[6] that are not the right chunks of text for comparing documents.We call this threshold the *shingle commonality threshold (SCT)*. We hypothesize that the elimination of these shingles will not have a significant impact on the NDD detection algorithm.

In order to see how this threshold affects the MapRecuce operation (described in Section 3.1.1) and NDD detection, we conducted a series of experiments. For these experiments, we modify the `Reducer` in MapReduce#1 in order to accommodate the SCT. The results of these experiments are summarized in Table 3.3. The first row indicates the level of the SCT[7], while the first column represents the RR value. In order to generate the data mentioned in this table, we ran a series of experiments. First, we kept all the shingles and calculated the RR measure based on that (the $SCT = \infty$ column). Then, in the subsequent experiments we kept only a specific portion (i.e $SCT = 100$, $SCT = 90$, etc.) of the whole shingles by eliminating the rest of the shingles based on their frequency of occurrence. Then we calculated the RR values based on the portion of the shingles that were kept (i.e $SCT = 100$ column, $SCT = 90$ column, etc.).The values in the table

[6]For instance, the following are two 64-character shingles that are very common in the collection of all relevant documents in runs submitted to TREC 2004 terabyte track:

- "internet sites should not be constructed as an endorsement of the"
- "sorry you need a javascript capable browser to get the best from "

[7]SCT=∞ means that no threshold was set.

Table 3.3: Impact of the Shingle Commonality Threshold on NDD Detection

SCT	∞	**100**	**90**	**80**	**70**	**60**	**50**	**40**	**30**	**20**	**10**
$0.9 \leq$ RR	7.44%	6.33%	6.31%	6.00%	5.89%	5.81%	5.80%	5.77%	5.68%	5.24%	4.31%
$0.8 \leq$ RR	10.05%	9.29%	9.27%	8.81%	8.61%	8.59%	8.58%	8.55%	8.50%	8.01%	6.48%
$0.7 \leq$ RR	12.37%	12.00%	11.98%	11.69%	11.16%	11.08%	11.06%	10.98%	10.93%	10.36%	8.54%
$0.6 \leq$ RR	14.64%	14.26%	14.25%	14.03%	13.77%	13.34%	13.32%	13.12%	13.07%	12.43%	10.57%
$0.5 \leq$ RR	17.36%	16.62%	16.61%	16.44%	16.32%	15.34%	15.31%	14.94%	14.88%	14.27%	12.05%
$0.4 \leq$ RR	20.08%	19.08%	19.06%	18.89%	18.69%	17.87%	17.82%	17.13%	17.00%	16.22%	13.94%
$0.3 \leq$ RR	23.07%	21.90%	21.88%	21.64%	21.36%	20.72%	20.69%	19.93%	19.52%	18.50%	16.10%
$0.2 \leq$ RR	26.90%	25.54%	25.53%	25.30%	24.94%	24.16%	24.16%	23.51%	23.22%	22.12%	19.26%
$0.1 \leq$ RR	34.12%	32.65%	32.64%	32.44%	32.03%	31.51%	31.08%	30.19%	29.84%	28.34%	25.00%

cells, represent the percentage [8] of NDDs that were detected by considering a specific threshold for NDD definition (i.e. $0.9 \leq$ RR, etc.) and a specific SCT (i.e. 100).

As we can see by decreasing this threshold to small values (e.g. 10), the percentage of documents with an RR value greater than or equal to the values mentioned in the first column decrease significantly. Setting this threshold to small values will lead to the elimination of a large portion of the total shingles. This trend has been shown in Figure 3.7.

As we did for the hash-value-based shingle sampling, we need to verify that the NDDs detected by using the SCT are the same as NDDs detected without it. In order to do so, we use the four measures introduced in Section 3.3.1: average error, correlation, recall and precision. The summary of this verification can be found in Table 3.4.

The Average Error column of Table 3.4 indicates that decreasing the value of SCT will cause an increase in the value of the average error which is not desirable. Nevertheless, we can see that SCT values higher than 70 will cause a negligibly low average error. The average error trends based on different SCT values are depicted in Figure 3.8(a).

In addition, the Correlation column of Table 3.4 demonstrates that lower SCT values cause a lower correlation of RR values. But as we can see the correlation trends in Figure 3.8(b) the changes of the correlation values are not as visible as the changes of average error depicted in Figure 3.8(a).

[8] In terms of the total number of relevant documents in the runs submitted to TREC 2004 terabyte track

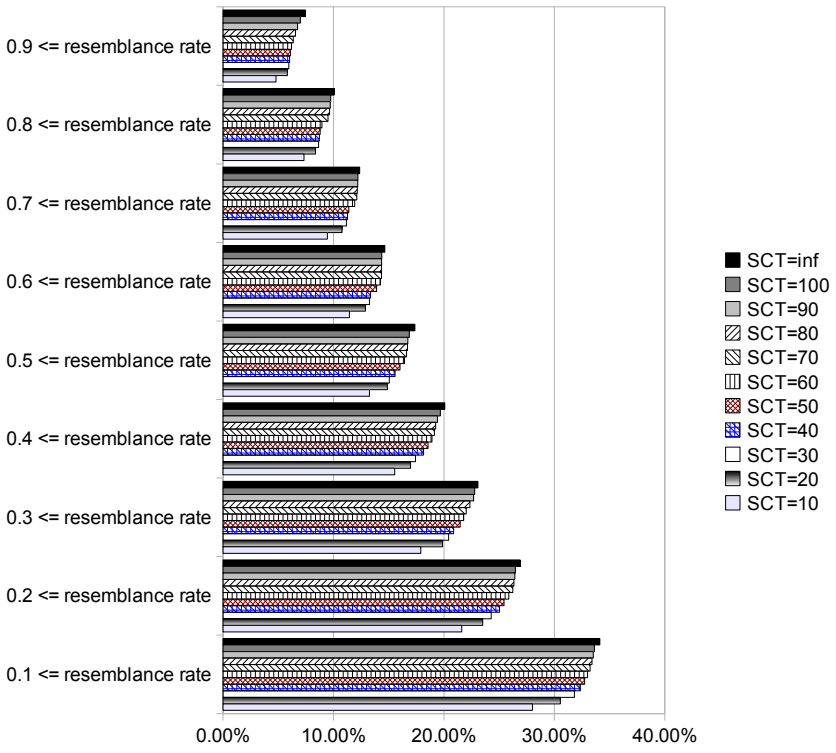


Figure 3.7: Impact of the Shingle Commonality Threshold on Duplicate Document Detection

Table 3.4: Performance Analysis of Threshold-based Shingle Sampling

SCT	**Average Error**	**Correlation**	**Recall**	**Precision**
100	0.0433	0.8511	0.9127	1.0000
90	0.0443	0.8484	0.9117	1.0000
80	0.0508	0.8273	0.9014	1.0000
70	0.0628	0.7756	0.8982	1.0000
60	0.1333	0.7639	0.7310	1.0000
50	0.1366	0.7611	0.7287	1.0000
40	0.2054	0.7767	0.6321	1.0000
30	0.2126	0.7730	0.6261	1.0000
20	0.2582	0.7559	0.5563	1.0000
10	0.3922	0.6212	0.3706	1.0000

Recall in Table 3.4 is calculated based on Equation 3.8. In other words, it is calculated against what we consider the ground truth here (i.e. $SCT = \infty$). It is shown that recall decreases by decreasing the SCT value. However, it remains higher than 90% for SCT values greater than or equal to 70. This trend is depicted in Figure 3.8(c).

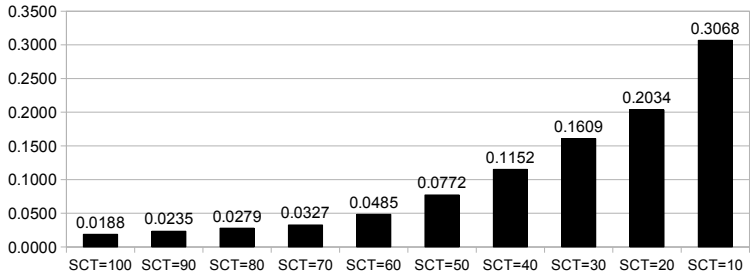
Precision is not affected by the Threshold-based shingling techniques. This is shown in the Precision column of Table 3.4. It is noteworthy, that precision is calculated based on Equation 3.9. In other words, precision is calculated based on the assumption that $SCT = \infty$ is the ground truth. This assumption has been validated in Section 3.2.

Impact on The MapReduce Operation

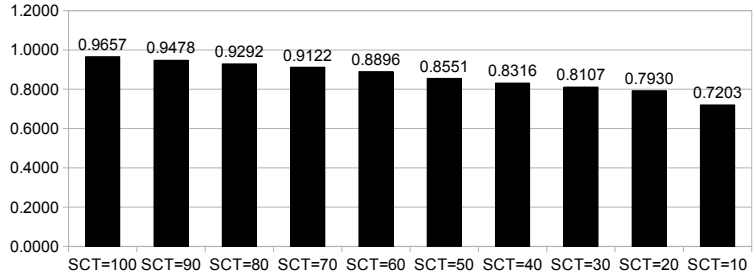
Thus far, we have studied how Threshold-based shingle sampling affects the quality of NDD detection. Nevertheless, there is another question that we need to answer at this point. How does Threshold-based shingle sampling facilitate the MapReduce operation described in Section 3.1.1.

We hypothesize that the SCT level will cause the number of the output pairs of MapReduce#1 in our algorithm to vary substantially. In order to investigate this issue, we run a series of experiments. In these experiments we vary the SCT value (e.g. 100, 90, etc.) and count the number of output $(key, value)$ pairs that Map-Reduce#1, explained in Section 3.1.1, produces. Then we compare these counts with the number of output $(key, value)$ pairs that Map-Reduce#1 produces when $SCT = \infty$.

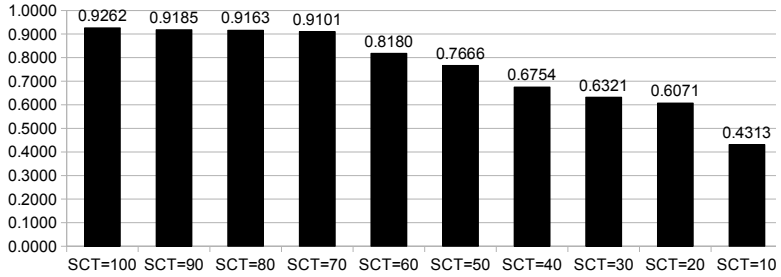
The result of these experiments is summarized in Figure 3.9. In this figure the y-axis represents the level of the SCT while the x-axis shows the percentage of reduction in the number of output pairs produced by MapReduce#1 in our algorithm. As we can see, the lower the level of SCT, the more pairs will be eliminated. The elimination of pairs will facilitate the MapReduce operation and speed up the NDD detection process. By considering Table 3.4 and Figure 3.9 at the same time, we can see that an SCT level of 70 although does not affect the NDD detection performance significantly, it does reduce the total number of pairs generated by MapReduce#1 about 25%.



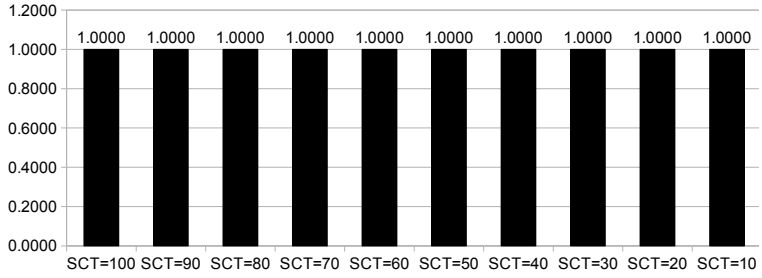
(a) Average Error (SCT)



(b) Correlation (SCT)



(c) Recall (SCT)



(d) Precision (SCT)

Figure 3.8: *Average Error*, *Correlation*, *Recall* and *Precision* of Threshold-based Sampling

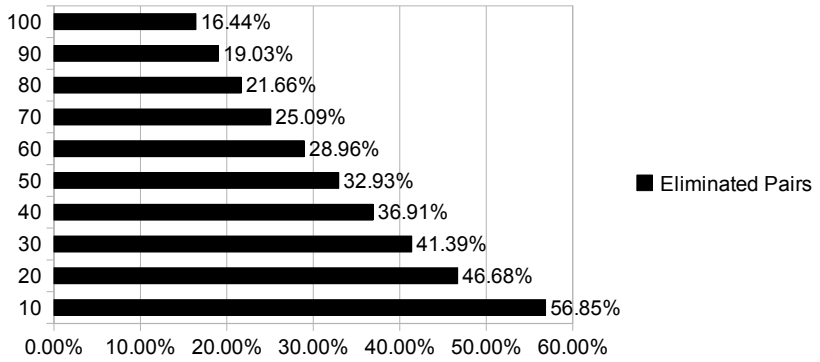


Figure 3.9: Impact of The Shingle Commonality Threshold on The Output Pairs of MR#1

3.4 Summary

In this chapter, we first introduced our NDD MapReduce solution which functions based on the shingling techniques proposed by Broder et. al [12, 13, 17, 16, 14, 15] . Our MapReduce solution consists of two MapReduce operations which calculate the Jaccord similarity measure for each pair of the documents of the input collection. The usage of MapReduce provides us with the benefit of highly desirable features like high scalability and fault tolerance.

We validated our solution by re-running one of the major experiments run by Bernstein and Zobel [6] on the relevant documents in the runs submitted to TREC 2004 treabyte track. We showed that our algorithm is capable of finding the same amount of NDDs in that collection as Bernstein and Zobel found.

Furthermore, we introduced two enhancement techniques for the MapReduce solution and studied their impact on the performance of the NDD detection.

Chapter 4

Experimental Results

4.1 Impact of NDD on Search Results

In this section we study the impact of the NDDs on search results. As previously mentioned, we conducted this work in the context of the TREC Web Track. In this section, we consider the runs submitted to the ad hoc task of TREC 2009 web track.

4.1.1 TREC 2009 Web Track

The goal of the TREC Web Track is to evaluate web retrieval technologies over the ClueWeb09 dataset.The ClueWeb09 data set includes 1 billion web pages in ten different languages and was created by the Language Technologies Institute at Carnegie Mellon University.The TREC Web Track includes two tasks: a traditional ad hoc task and a new diversity task.We consider the runs submitted to the TREC 2009 ad hoc task for our experiments in this section of the paper.

4.1.2 NDDs in Topic Collections

The participants in the TREC 2009 Web Track were required to return a list of 1000 documents from the ClueWeb09 dataset for 50 different query topics [1]. If we consider all the submitted

[1]See Appendix A for a complete list of the query topics.

results for each of the query topics as one small collection of documents, we will have 50 small collections. The prevalence of NDDs in these topic collections has been shown in Figures 4.1, 4.2, 4.3 and 4.4.

As we can see in these Figures, by considering a threshold of $RR \geq 0.50$, for all the query topics the prevalence of NDDs is over 20%. The average prevalence of NDDs in Topic Collections has been depicted in Figure 4.5.

4.1.3 NDDs in The Submitted Runs

One of the other issues that we would like to investigate is the prevalence of NDDs per query topic in each of the runs submitted by the participants of the TREC 2009 Web Track.

We call the results returned in each run for a specific query a *run-topic*. Our goal here is to study the prevalence of NDDs in the run-topics. In order to do, so we consider the top n documents in each run-topic. The prevalence of NDDs for different values of n is summarized in Table 4.1.

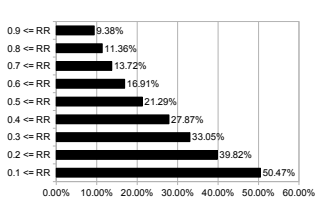
In Table 4.1, the first column indicates the cumulative percentage of NDDs by considering a threshold of $RR \geq 0.5$ for the definition of NDDs. The first row indicates the value of n for a particular experiment. Where n is the number of top documents considered for a each run-topic. Hence, each cell of the table indicates the percentage of run-topics that have a specific percentage of NDDs (indicated by the first column value) by considering the top n results of the run-topics.

The trends of Table 4.1 are depicted in Figure where the x-axis represents the percentage of NDDs and the y-axis represents the percentage of run-topics.

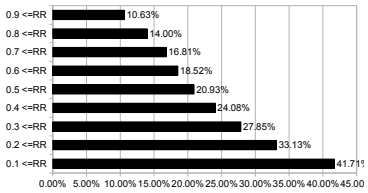
4.2 Sources of NDDs

A manual review of the document pairs detected as NDDs reveals the following as the two major sources of NDDs in the topic runs:

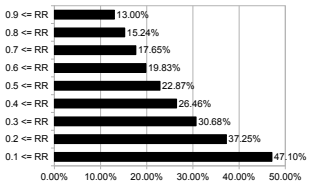
- URL variations: Many documents that are either exactly the same or near-duplicates with different but usually similar URLs. Similar URLs usually vary in either a prefix



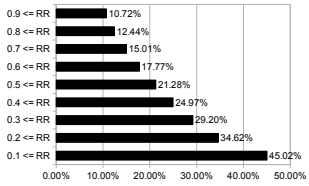
(a) wt09-1



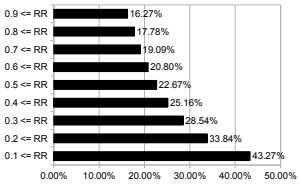
(b) wt09-2



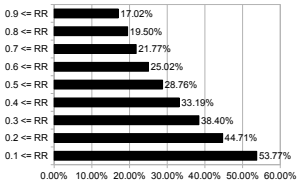
(c) wt09-3



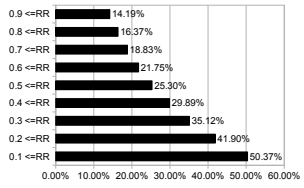
(d) wt09-4



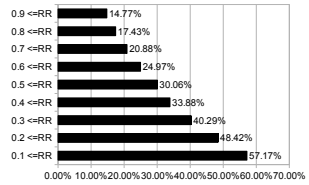
(e) wt09-5



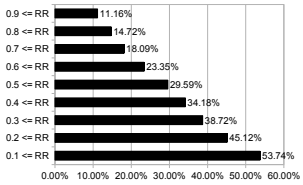
(f) wt09-6



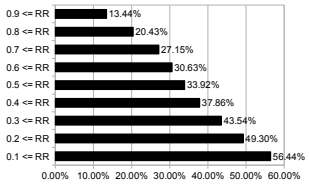
(g) wt09-7



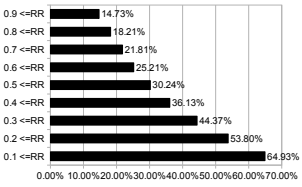
(h) wt09-8



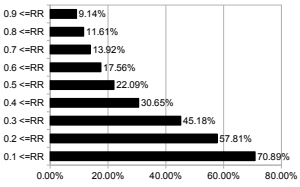
(i) wt09-9



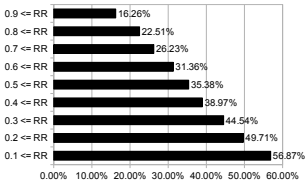
(j) wt09-10



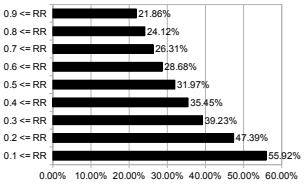
(k) wt09-11



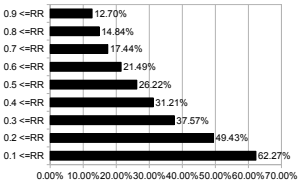
(l) wt09-12



(m) wt09-13

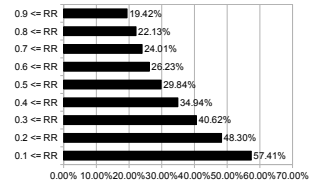


(n) wt09-14

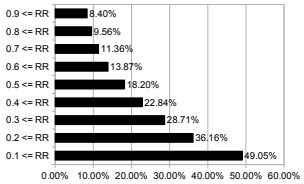


(o) wt09-15

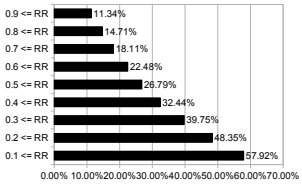
Figure 4.1: NDDs in Topic Collections (Part 1)



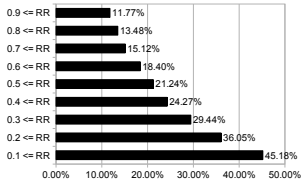
(a) wt09-16



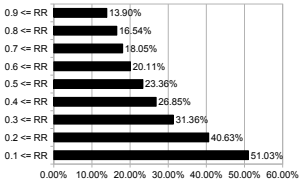
(b) wt09-17



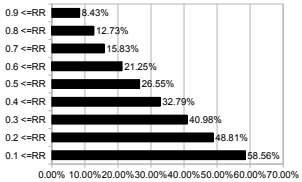
(c) wt09-18



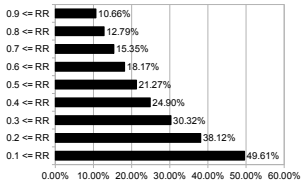
(d) wt09-19



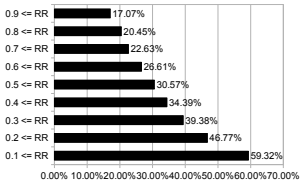
(e) wt09-20



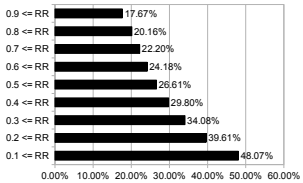
(f) wt09-21



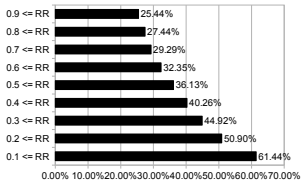
(g) wt09-22



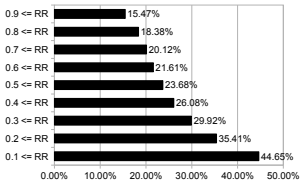
(h) wt09-23



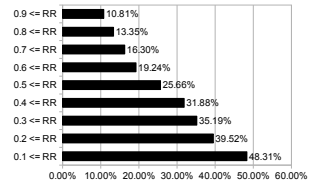
(i) wt09-24



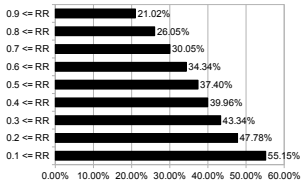
(j) wt09-25



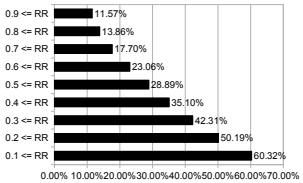
(k) wt09-26



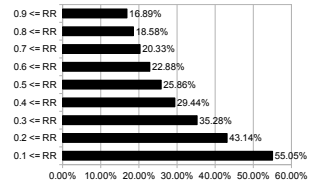
(l) wt09-27



(m) wt09-28

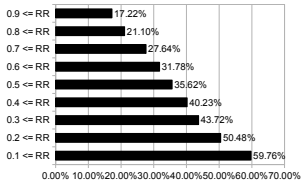


(n) wt09-29

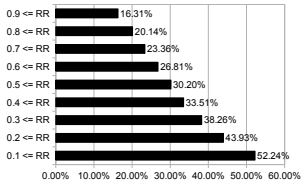


(o) wt09-30

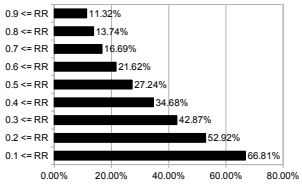
Figure 4.2: NDDs in Topic Collections (Part 2)



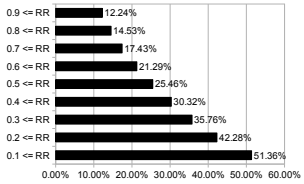
(a) wt09-31



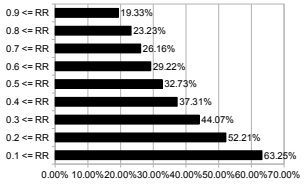
(b) wt09-32



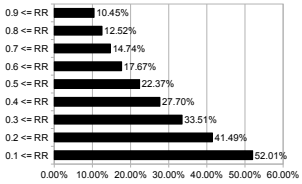
(c) wt09-33



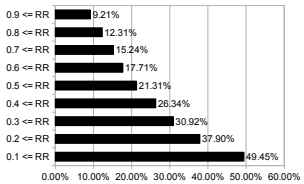
(d) wt09-34



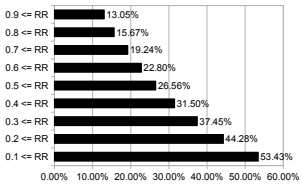
(e) wt09-35



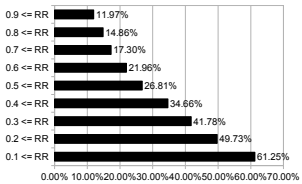
(f) wt09-36



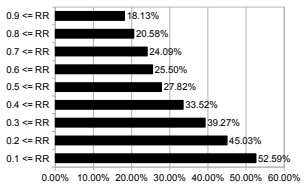
(g) wt09-37



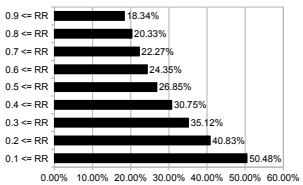
(h) wt09-38



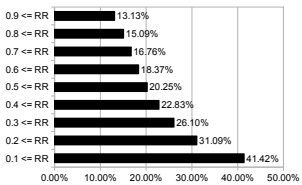
(i) wt09-39



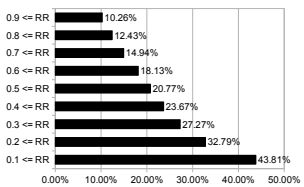
(j) wt09-40



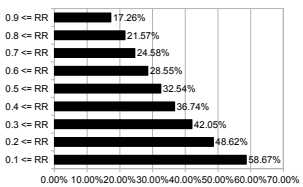
(k) wt09-41



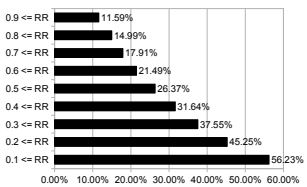
(l) wt09-42



(m) wt09-43

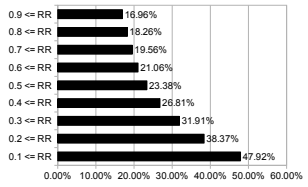


(n) wt09-44

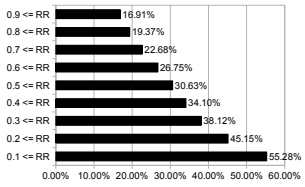


(o) wt09-45

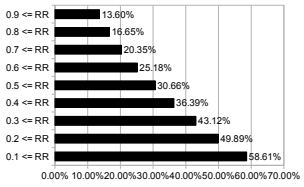
Figure 4.3: NDDs in Topic Collections (Part 3)



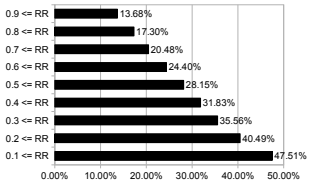
(a) wt09-46



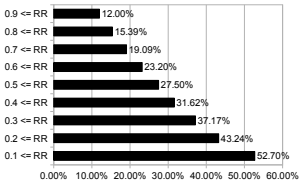
(b) wt09-47



(c) wt09-48



(d) wt09-49



(e) wt09-50

Figure 4.4: NDDs in Topic Collections (Part 4)

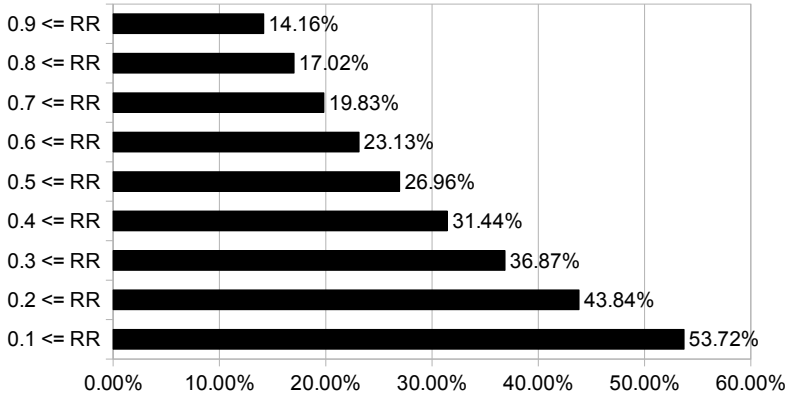


Figure 4.5: Average Prevalence of NDDs in Topic Collections

Table 4.1: Prevalence of Duplicates in TREC 2009 Run-Topics

Cumulative % of NDDs	% of Run-Topics Considering the Top n Results					
	n=10	**n=20**	**n=50**	**n=100**	**n=200**	**n=500**
5.00%	-	67.46%	73.80%	86.00%	92.25%	95.92%
10.00%	45.41%	49.13%	57.77%	66.79%	75.35%	84.65%
15.00%	45.41%	37.46%	42.11%	50.17%	57.15%	66.62%
20.00%	27.10%	30.00%	34.85%	37.32%	41.38%	48.68%
25.00%	27.10%	23.89%	26.85%	27.35%	29.86%	34.99%
30.00%	19.46%	20.17%	21.55%	21.24%	22.76%	24.17%
35.00%	19.46%	17.32%	16.62%	16.28%	17.89%	16.42%
40.00%	14.62%	14.28%	13.83%	13.21%	13.66%	13.07%
45.00%	14.62%	12.56%	11.15%	11.01%	10.56%	10.28%
50.00%	11.69%	11.07%	9.92%	9.24%	8.17%	7.66%
55.00%	11.69%	9.69%	7.86%	7.49%	6.45%	5.52%
60.00%	8.85%	8.34%	7.04%	5.86%	4.99%	3.86%
65.00%	8.85%	6.85%	5.46%	4.59%	3.46%	2.70%
70.00%	6.11%	5.75%	4.28%	3.44%	2.45%	1.61%
75.00%	6.11%	4.39%	3.01%	2.42%	1.61%	0.96%
80.00%	3.66%	3.52%	2.37%	1.55%	0.93%	0.65%
85.00%	3.66%	2.45%	1.80%	0.90%	0.59%	0.28%
90.00%	1.92%	1.69%	1.30%	0.51%	0.25%	0.17%
95.00%	1.92%	0.76%	0.62%	0.17%	0.08%	0.08%

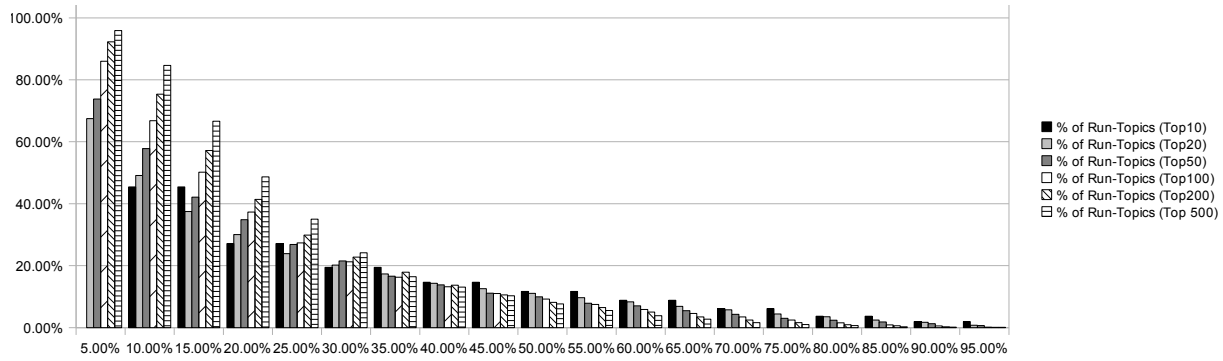


Figure 4.6: Prevalence of Duplicates in TREC 2009 Run-Topics

(e.g. www) or a postfix (e.g. /index.htm). For instance, in the ClueWeb09 collection two documents `clueweb09-en0032-85-03856` and `clueweb09-en0052-46-04406` are detected as near duplicate documents by our algorithm. The URL of the former document is `http://mirror-pole.com/apif_web/` while the URL of the latter document is `http://www.mirror-pole.com/apif_web/index.htm`. However, there are instances where the URLs are completely different (e.g. mirror sites, plagiarized content).

- Editorial variations: This includes page updates, variation in advertisement and forum like web sites. In particular, there are many updates of Wikipedia pages. For instance, in the ClueWeb09 collection the two documents with IDs `clueweb09-enwp03-44-01481` and `clueweb09-enwp02-01-19647` are near-duplicate Wikipedia documents with subtle editorial differences.

Chapter 5

Concluding Remarks

5.1 Conclusions

This thesis considers the undesirable prevalence of NDDs in information retrieval systems.We first examined a scalable implementation of a NDD detection algorithm based on the MapReduce framework.

In order to validate our implementation, we reproduced one of the main experiments reported by Bernstein and Zobel [6].The comparison of the results indicates that our algorithm is capable of producing reasonable performance.We examined two different techniques for enhancing the scalability of our algorithm.

In addition, we conducted a careful study on the impact of these enhancements on the quality of our NDD detection algorithm. For this purpose, we report four different measures - namely average error, correlation, recall and precision - for these two enhancements.

Finally, we applied our algorithm to the runs submitted to the ad hoc task of the TREC 2009 Web Track. As part of this study, we report the prevalence of NDDs in the top 10, 20, 50, 100 and 500 results returned by each of the participants in this experiment for each of the query topics. Our results show that 45.41% of all the run-topics submitted to the ad hoc task of TREC 2009 web track included more than one NDD in their top 10 returned results and 19.46% of all the run-topics included more than three NDDs in their top 10 results. Comprehensive statistics

were provided in Table 4.1. In particular and based on our results, we recommend to all TREC web track participants to use an explicit NDD detection algorithm in their retrieval process. This will favorably affect their effectiveness measures such as the $\alpha - nDCG$.

In general, we strongly believe that our study can be used by IR systems to improve their effectiveness in terms of some recently proposed effectiveness measures that explicitly penalize redundancy and reward novelty[28, 22].

5.2 Future Work

As future work, we are planning to apply our enhanced MapReduce algorithm to the English part of the ClueWeb09 collection ($\sim$12.5 TB, $\sim$ 503 million documents) and create a taxonomy for NDDs found in this collection. Since ClueWeb09 is a reasonable snapshot of the web, we expect our taxonomy to be a reasonable categorization of all the different types of NDDs on the web.

APPENDICES

Appendix A

TREC 2009 Web Track Topics

wt09-1:obama family tree
wt09-2:french lick resort and casino
wt09-3:getting organized
wt09-4:toilet
wt09-5:mitchell college
wt09-6:kcs
wt09-7:air travel information
wt09-8:appraisals
wt09-9:used car parts
wt09-10:cheap internet
wt09-11:gmat prep classes
wt09-12:djs
wt09-13:map
wt09-14:dinosaurs
wt09-15:espn sports
wt09-16:arizona game and fish
wt09-17:poker tournaments
wt09-18:wedding budget calculator
wt09-19:the current
wt09-20:defender
wt09-21:volvo
wt09-22:rick warren
wt09-23:yahoo
wt09-24:diversity
wt09-25:euclid
wt09-26:lower heart rate
wt09-27:starbucks
wt09-28:inuyasha
wt09-29:ps 2 games
wt09-30:diabetes education
wt09-31:atari
wt09-32:website design hosting
wt09-33:elliptical trainer
wt09-34:cell phones
wt09-35:hoboken
wt09-36:gps
wt09-37:pampered chef
wt09-38:dogs for adoption
wt09-39:disneyland hotel
wt09-40:michworks
wt09-41:orange county convention center
wt09-42:the music man
wt09-43:the secret garden
wt09-44:map of the united states
wt09-45:solar panels
wt09-46:alexian brothers hospital
wt09-47:indexed annuity
wt09-48:wilson antenna
wt09-49:flame designs
wt09-50:dog heat

Bibliography

[1] A. Agarwal, H.S. Koppula, K.P. Leela, K.P. Chitrapura, S. Garg, P.K. GM, C. Haty, A. Roy, and A. Sasturkar. URL normalization for de-duplication of web pages. In *Proceeding of the 18th ACM conference on Information and knowledge management*, pages 1987–1990. ACM, 2009.

[2] Jesse Alpert and Nissan Hajaj. We knew the web was big... In *The Official Google Blog: http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html*, 2008.

[3] Amazon Web Services LLC. *Amazon Simple Storage Service: Getting Started Guide*, 2009.

[4] Amazon Web Services LLC. *Amazon SimpleDB: Getting Started Guide*, 2010.

[5] Apache. How map and reduce operations are actually carried out. In *HadoopMapReduce-Hadoop Wiki*, 2010.

[6] Yaniv Bernstein and Justin Zobel. Redundant documents and search effectiveness. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 736–743, New York, NY, USA, 2005. ACM.

[7] Yaniv Bernstein and Justin Zobel. Accurate discovery of co-derivative documents via duplicate text detection. *Information Systems*, 31(7):595–609, November 2006.

[8] Mikhail Bilenko and Raymond J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 39–48, New York, NY, USA, 2003. ACM.

[9] Christophe Bisciglia, Aaron Kimball, and Sierra Michels-Slettvet. Mapreduce theory and implementation. Google, Inc, Distributed Computing Seminar, Summer 2007.

[10] N. Bourbakis, W. Meng, Z. Wu, J. Salerno, and S. Borek. Removal of redundancy in documents retrieved from different resources. In *Tools with Artificial Intelligence, 1998. Proceedings. Tenth IEEE International Conference on*, pages 112–119, Nov 1998.

[11] Sergey Brin, James Davis, and Héctor García-Molina. Copy detection mechanisms for digital documents. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 398–409, New York, NY, USA, 1995. ACM.

[12] Andrei Z. Broder. Some applications of rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[13] Andrei Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29, June 1997.

[14] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *COM '00: Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 1–10, London, UK, 2000. Springer-Verlag.

[15] Andrei Z. Broder. Algorithms for duplicate documents. *www.cs.princeton.edu/courses/archive/spr05/cos598E/bib/Princeton.pdf*, 2005.

[16] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 327–336, New York, NY, USA, 1998. ACM.

[17] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8-13):1157–1166, 1997.

[18] Stefan Büttcher, Charles L. A. Clarke, and Gordon V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press, Cambridge, Massachussetts, September 2010.

[19] Jamie Callan, Mark Hoy, Changkuk Yoo, and Le Zhao. Web08 − pr dataset. Technical report, Language Technologies Institute, Carnegie Mellon University, `http://boston.lti.cs.cmu.edu/Data/web08-bst/web08-bst-Sep26-08.pdf`, September 2008.

[20] William B. Cavnar and John M. Trenkle. N-gram-based text categorization. In *In Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, 1994.

[21] Hung-Chi Chang and Jenq-Haur Wang. Organizing news archives by near-duplicate copy detection in digital libraries. In *Asian Digital Libraries. Looking Back 10 Years and Forging New Frontiers*, volume 4822 of *Lecture Notes in Computer Science*, pages 410–419. Springer Berlin / Heidelberg, 2007.

[22] Olivier Chapelle, Donald Metlzer, Ya Zhang, and Pierre Grinspan. Expected reciprocal rank for graded relevance. In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, pages 621–630, New York, NY, USA, 2009. ACM.

[23] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, New York, NY, USA, 2002. ACM.

[24] A. Chowdhury. Duplicate data detection. *Retrieved from* *http://ir.iit.edu/~abdur/Research/Duplicate.html*, 2004.

[25] Abdur Chowdhury, Ophir Frieder, David Grossman, and Mary Catherine McCabe. Collection statistics for fast duplicate document detection. *ACM Transactions on Information Systems*, 20(2):171–191, 2002.

[26] Peter Christen. Febrl -: an open source data cleaning, deduplication and record linkage system with a graphical user interface. In *KDD '08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1065–1068, New York, NY, USA, 2008. ACM.

[27] Charles L. Clarke, Maheedhar Kolla, and Olga Vechtomova. An effectiveness measure for ambiguous and underspecified queries. In *Proceedings of the 2nd International Conference*

on Theory of Information Retrieval: Advances in Information Retrieval Theory, ICTIR '09, pages 188–199, Berlin, Heidelberg, 2009. Springer-Verlag.

[28] Charles L.A. Clarke, Maheedhar Kolla, Gordon V. Cormack, Olga Vechtomova, Azin Ashkan, Stefan Büttcher, and Ian MacKinnon. Novelty and diversity in information retrieval evaluation. In *SIGIR '08: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 659–666, New York, NY, USA, 2008. ACM.

[29] Jack G. Conrad, Xi S. Guo, and Cindy P. Schriber. Online duplicate document detection: signature reliability in a dynamic retrieval environment. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 443–452, New York, NY, USA, 2003. ACM.

[30] Jack G. Conrad and Cindy P. Schriber. Constructing a text corpus for inexact duplicate detection. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 582–583, New York, NY, USA, 2004. ACM.

[31] Jack G. Conrad and Cindy P. Schriber. Managing déjà vu: Collection building for the identification of nonidentical duplicate documents. *Journal of American Society for Inforfation Science and Technology*, 57(7):921–932, 2006.

[32] Anirban Dasgupta, Ravi Kumar, and Amit Sasturkar. De-duping urls via rewrite rules. In *KDD '08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 186–194, New York, NY, USA, 2008. ACM.

[33] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 137–150, 2004.

[34] G.A. Di Lucca, M. Di Penta, and A.R. Fasolino. An approach to identify duplicated web pages. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pages 481–486, 2002.

[35] Mohamed Elhadi and Amjad Al-Tobi. Duplicate detection in documents and webpages using improved longest common subsequence and documents syntactical structures. In *Fourth International Conference on Computer Sciences and Convergence Information Technology, 2009. ICCIT '09.*, pages 679 – 684, Seoul, Korea, November 2009.

[36] Dennis Fetterly, Mark Manasse, and Marc Najork. On the evolution of clusters of near-duplicate web pages. In *LA-WEB '03: Proceedings of the First Conference on Latin American Web Congress*, page 37, Washington, DC, USA, 2003. IEEE Computer Society.

[37] Dennis Fetterly, Mark Manasse, and Marc Najork. Detecting phrase-level duplication on the world wide web. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 170–177, New York, NY, USA, 2005. ACM.

[38] George Forman, Kave Eshghi, and Stephane Chiocchetti. Finding similar files in large document repositories. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, KDD '05, pages 394–400, New York, NY, USA, 2005. ACM.

[39] T.F. Frandsen. Letters to the editor: simplifying the reversed duplicate removal procedure. *Journal of the American Society for Information Science and Technology*, 54(3):275–276, 2003.

[40] Caichun Gong, Yulan Huang, Xueqi Cheng, and Shuo Bai. Detecting near-duplicates in large-scale short text databases. In *PAKDD'08: Proceedings of the 12th Pacific-Asia conference on Advances in knowledge discovery and data mining*, pages 877–883, Berlin, Heidelberg, 2008. Springer-Verlag.

[41] Hannaneh Hajishirzi, Wen-tau Yih, and Aleksander Kolcz. Adaptive near-duplicate detection via similarity learning. In *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, pages 419–426, New York, NY, USA, 2010. ACM.

[42] Rob Hall, Charles Sutton, and Andrew McCallum. Unsupervised deduplication using cross-field dependencies. In *KDD '08: Proceeding of the 14th ACM SIGKDD international confer-*

ence on Knowledge discovery and data mining, pages 310–317, New York, NY, USA, 2008. ACM.

[43] Nevin Heintze. Scalable document fingerprinting. In *Proceedings of USENIX Workshop on Electronic Commerce*, pages 191–200, 1996.

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

[45] T.C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *Journal of the American Society for Information Science and Technology*, 54(3):203–215, 2003.

[46] Yan Hu, Wei Li, Ying Qiu, and Wei Wu. Research of duplicate record cleaning technology based on a reformative keywords matching algorithm. In *E-Business and Information System Security, 2009. EBISS '09. International Conference on*, pages 1–5, May 2009.

[47] Lian'en Huang, Lei Wang, and Xiaoming Li. Achieving both high precision and high recall in near-duplicate detection. In *CIKM '08: Proceeding of the 17th ACM conference on Information and knowledge management*, pages 63–72, New York, NY, USA, 2008. ACM.

[48] Scott Huffman, April Lehman, Alexei Stolboushkin, Howard Wong-Toi, Fan Yang, and Hein Roehrig. Multiple-signal duplicate detection for search evaluation. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 223–230, New York, NY, USA, 2007. ACM.

[49] Dmitry I. Ignatov and Sergei O. Kuznetsov. Frequent itemset mining for clustering near duplicate web documents. In *ICCS '09: Proceedings of the 17th International Conference on Conceptual Structures*, pages 185–200, Berlin, Heidelberg, 2009. Springer-Verlag.

[50] S. Jonathan and A. Paepcke. SpotSigs: Near Duplicate Detection in Web Page Collections. 2007.

[51] Aaron Kimball. Cluster computing and mapreduce lecture 2. `http://www.youtube.com/watch?v=-vD6PUdf3Js`, Google Inc., Google Code University, Summer 2007.

[52] Aleksander Kolcz and Abdur Chowdhury. Lexicon randomization for near-duplicate detection with i-match. *The Journal of Supercomputing*, 45(3):255–276, 2008.

[53] J.P. Kumar and P. Govindarajulu. Duplicate and Near Duplicate Documents Detection: A Review. *European Journal of Scientific Research*, 32(4):514–527, 2009.

[54] Luís Leit ao, Pável Calado, and Melanie Weis. Structure-based inference of xml similarity for fuzzy duplicate detection. In *CIKM '07: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 293–302, New York, NY, USA, 2007. ACM.

[55] W. Li, J.Y. Liu, and C. Wang. Web document duplicate removal algorithm based on keyword sequences. In *Natural Language Processing and Knowledge Engineering, 2005. IEEE NLP-KE'05. Proceedings of 2005 IEEE International Conference on*, pages 511–516, 2005.

[56] Tim Lindholm and Frank Yellin. *The* Java^{TM} *Virtual Machine Specification*. Prentice Hall, 2nd edition, April 1999.

[57] Daniel P. Lopresti. Models and algorithms for duplicate document detection. In *ICDAR '99: Proceedings of the Fifth International Conference on Document Analysis and Recognition*, page 297, Washington, DC, USA, 1999. IEEE Computer Society.

[58] Daniel P. Lopresti. A comparison of text-based methods for detecting duplication in scanned document databases. *Information Retrrieval*, 4(2):153–173, 2001.

[59] Udi Manber. Finding similar files in a large file system. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 2–2, Berkeley, CA, USA, 1994. USENIX Association.

[60] Federica Mandreoli, Riccardo Martoglia, and Paolo Tiberio. A document comparison scheme for secure duplicate detection. *International Journal on Digital Libraries*, 4(3):223–244, 2004.

[61] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 141–150, New York, NY, USA, 2007. ACM.

[62] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Duplicate detection in click streams. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 12–21, New York, NY, USA, 2005. ACM.

[63] Donald Metzler, Yaniv Bernstein, W. Bruce Croft, Alistair Moffat, and Justin Zobel. Similarity measures for tracking information flow. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, CIKM '05, pages 517–524, New York, NY, USA, 2005. ACM.

[64] VA Narayana, P. Premchand, and A. Govardhan. A Novel and Efficient Approach For Near Duplicate Page Detection in Web Crawling. In *IEEE International Advance Computing Conference*, 2009.

[65] G. Niklas Norén, Roland Orre, and Andrew Bate. A hit-miss model for duplicate detection in the who drug safety database. In *KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 459–468, New York, NY, USA, 2005. ACM.

[66] A. Pereira Jr, R. Baeza-Yates, and N. Ziviani. Where and how duplicates occur in the web. In *Proc. 4th Latin American Web Congress*, pages 127–134. Citeseer, 2006.

[67] M. Potthast and B. Stein. New Issues in Near-duplicate Detection. *Data Analysis. In: Machine Learning and Applications. Springer, Heidelberg*, 200, 2007.

[68] Martin Potthast. Wikipedia in the pocket: indexing technology for near-duplicate detection and high similarity search. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 909–909, New York, NY, USA, 2007. ACM.

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

[70] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 499–510, May 2007.

[71] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 269–278, New York, NY, USA, 2002. ACM.

[72] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, New York, NY, USA, 2003. ACM.

[73] Jangwon Seo and W. Bruce Croft. Local text reuse detection. In *SIGIR '08: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 571–578, New York, NY, USA, 2008. ACM.

[74] Narayanan Shivakumar and Hector Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *DL '96: Proceedings of the first ACM international conference on Digital libraries*, pages 160–168, New York, NY, USA, 1996. ACM.

[75] Martin Theobald, Jonathan Siddharth, and Andreas Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *SIGIR '08: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 563–570, New York, NY, USA, 2008. ACM.

[76] Z. Tian, H. Lu, W. Ji, A. Zhou, and Z. Tian. An n-gram-based approach for detecting approximately duplicate database records. *International Journal on Digital Libraries*, 3(4):325–331, 2002.

[77] R. Vallur and R.S. Chandrasekhar. An adaptive algorithm for detection of duplicate records. In *TENCON 2003. Conference on Convergent Technologies for Asia-Pacific Region*, volume 1, pages 424–427 Vol.1, Oct. 2003.

[78] Jenq-Haur Wang and Hung-Chi Chang. Exploiting Sentence-Level Features for Near-Duplicate Document Detection. *Information Retrieval Technology*, pages 205–217, 2009.

[79] M. Wang and D. Liu. The Research of Web Page De-duplication Based on Web Pages Reshipment Statement. In *2009 First International Workshop on Database Technology and Applications*, pages 271–274. IEEE, 2009.

[80] M. Weis and F. Naumann. Detecting duplicates in complex xml data. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, pages 109–109, April 2006.

[81] Yunhe Weng, Lei Li, and Yixin Zhong. Semantic keywords-based duplicated web pages removing. In *Natural Language Processing and Knowledge Engineering, 2008. NLP-KE '08. International Conference on*, pages 1–7, Oct. 2008.

[82] Tom White. *Hadoop: The Definitive Guide*. O'Reilly—Yahoo Press, first edition, June 2009.

[83] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 131–140, New York, NY, USA, 2008. ACM.

[84] Mansheng Xiao, Youshi Liu, and Xiaoqi Zhou. A property optimization method in support of approximately duplicated records detecting. In *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*, volume 3, pages 118–122, Nov. 2009.

[85] Hui Yang and Jamie Callan. Near-duplicate detection by instance-level constrained clustering. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 421–428, New York, NY, USA, 2006. ACM.

[86] Shaozhi Ye, Ji-Rong Wen, and Wei-Ying Ma. A systematic study on parameter correlations in large-scale duplicate document detection. *Knowledge and Information Systems*, 14(2):217–232, 2008.

[87] Qi Zhang, Yue Zhang, Haomin Yu, and Xuanjing Huang. Efficient partial-duplicate detection based on sequence matching. In *SIGIR '10: Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 675–682, New York, NY, USA, 2010. ACM.

[88] Zhigang Zhang, Weijia Jia, and Xiaoming Li. Enhancing Duplicate Collection Detection Through Replica Boundary Discovery. *Advances in Knowledge Discovery and Data Mining*, pages 361–370, 2006.

[89] Justin Zobel and Yaniv Bernstein. The case of the duplicate documents measurement, search, and science. In *Frontiers of WWW Research and Development-APWeb 2006*, pages 26–39. Springer, 2006.

Index