The Pennsylvania State University

The Graduate School

HIGH PERFORMANCE RECORD LINKAGE

A Dissertation in

Computer Science and Engineering

by

Hung-sik Kim

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

August 2010

Abstract

In current world, the immense size of a data set makes problems in finding similar/identitcal data. In addition, the dirtiness of data, i.e. typos, missing/tilting information, and additional noises usually occurred by careless editing or entry mistakes, makes further difficulty to identify entity-belongs. Therefore, we focus on the faster detection of data referring the same real-world entity from a large size data set under the error prone environments, while the high accuracy of detection is maintained. In this thesis, we study high-performance linkage algorithms using four different applications. First, we introduce the image linkage algorithm to find near-duplicate images with similar characteristics by bridging two seemingly unrelated fields – Multimedia Information Retrieval and Biology. Under this idea, we study how various image features and gene sequence generation methods affect the accuracy and performance of detecting near-duplicate images. Second, we develop the video linkage algorithm using record linkage methods to detect copied videos from a large multi-media database or sites such as YouTube and Yahoo Videos. The utilization of video characteristics is reflected to the hierarchical structure of the proposed algorithms. In addition, the uses of pipe-lined linkage structures accelerate the speed further. Third, the parallel linkage algorithm, the parallelization of the data linkage frame, is introduced, when slow but optimal sequential linkage frames occur where iterative matching operations apply to clean and merge dirty sets. Any data matching functions can be adapted to the proposed parallel framework because a data linkage function is considered as a black box in the parallel scheme. Finally, we introduce a hashed linkage structure based on the locality sensitive hashing (LSH) algorithm. By remedying the poverty of a basic LSH structure to suit linkage problems, the proposed hashing structure reduces the precessing time tremendously comparing to the conventional LSH structures.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am heartily thankful to my advisor, Dr. Dongwon Lee, for his encouragement, guidance, patience, and support during my time in graduate school. I also owe my deepest gratitude to my parents, Mr. Changsoo Kim and Mrs. Soonki Kim, who encourage me during all my life. In addition, I would like to thank to my wife, Minam Kim, and my kids, Dongha, Minha, and Tayha Kim. Without my family, this thesis would not have been possible. I would also like to thank other committee members, Dr. Padma Raghavan, Dr. Bhuvan Urgaonkar, Dr. Ae Ja Yee, and Dr. Raj Acharya, for their guidance and suggestions during my research.

Lastly, I offer my regards and blessings to all others who supported me during my graduate school.

Chapter 1

Introduction

Large-scale data repositories often suffer from duplicate entities that have different identifiers, but refer to the same real-world object. To make matters worse, demand for large-scale data handling in diverse data formats makes this problem even harder in terms of accuracy as well as searching time for a query data. In addition, both cleaning a large-scale dirty data set and merging heterogeneous large-scale data sets are required tremendous processing time, because a record linkage between two data sets demands quadratic number of comparisons without any modification. For example, when all elements in a set with the size of N is compared together, we need ${}_NC_2$ pair-wise comparisons. For two different sets, $|A| = m$ and $|B| = n$, then the record linkage process requires $O(mn)$ time complexity. In this thesis, the improvement of performance to find records referring the same real-world object is mainly aimed. In order to surmount a tedious running time, we suggest four novel record linkage applications in various aspects.

Record linkage algorithm is usually located in two different steps. One is the *match* step, and the other is *merge* step. The *match* process is generally known as finding the similar (or same) record using proper *matching* function, while the *merge* process is not known in detail in terms of how to merge two clone-like records. Therefore, some applications [5, 64, 14] only focus on the generic algorithms for *match* step, but others [68, 10] also consider *merge* step even though the *merge* function is not fully studied.

In this thesis, we study high-performance record linkage algorithms in four different applications to enhance processing time as well as accuracy.

First, the Image Linkage algorithm is developed to find near-duplicate images with similar characteristics found in Multimedia Information Retrieval (MIR). Toward this effort, we propose a novel idea by bridging two seemingly unrelated fields – *MIR* and *Biology*. That is, we propose to use the popular gene sequence alignment algorithm in Biology, i.e., BLAST, in detecting near-duplicate images. Under the new idea, we study how various image features and gene sequence generation methods (using gene alphabets such as `A`, `C`, `G`, and `T` in DNA sequences) affect the accuracy and performance of detecting near-duplicate images. We also investigate the impact of customized scoring matrices in BLAST, originally optimized for Biological data. The output of Image Linkage solution is a set of near-duplicate images for a query image.

Second, the Video Linkage algorithm – the copied video detection using a record linkage method – is shown to find copied videos for a query video from large video collections such as YouTube or Yahoo video. The issue on copied video detection (CVD) is not on merging or cleaning a data set, but on finding copied videos to address the copyright protection problem. Thus, Video Linkage focuses on how to make an efficient *match* algorithm. In Video Linkage algorithm, we utilize video feature and a video structure to enhance performance. In addition, video pre-filtering and pipe-lining also improve a query processing time drastically. The output of Video Linkage solution is a set of copied videos for a query video.

Third, we study the Parallel Linkage algorithm which can exploit any distance measures (*match* functions) of data linkage algorithms and any merging methods (*merge* functions) , i.e., the proposed parallel linkage algorithms are linkage-function-independent. We address to implement the parallel structure of the optimal sequential linkage algorithm when *merge* step is applied to clean a data set. The well-known *match* function - Jaccard- is used, and $\cup\{\text{tokens}\}$ is used to merge citation records. Therefore, the output of Parallel Linkage is one clean set from one dirty set or many input sets.

Fourth, we propose the Hashed Linkage algorithm based on locality sensitive hashing (LSH). Hashed Linkage performs to clean large-scale record sets based on fast and scalable search response to a query record. Thus, the *merge* step, $\cup\{tokens\}$ with citation records, is also involved in Hashed Linkage. The Hashed Linkage algorithm exploits the LSH technique in a different way from conventional

LSH idea by using the iterative nature of a record linkage that proceeds *match* and *merge* steps at each iteration. By reducing the size of data sets at each iteration by *merge* process, with the same number of keys per records, we outperform other conventional LSH based linkage algorithms. Furthermore, our hashing algorithm will minimize the missing data that should be returned, but maximize the efficiency on both processing/query time and space uses. Similar to Parallel Linkage, Hashed Linkage also exploits any *match* and *merge* functions in its structure, and outputs one clean set.

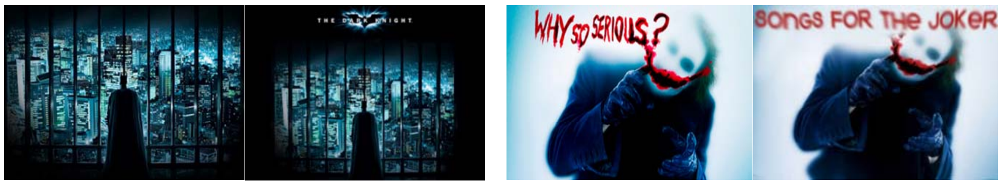
In followings, we reviews the motivations for each linkage algorithm developed for specific or general purpose in record linkage areas.

1.1 Motivation

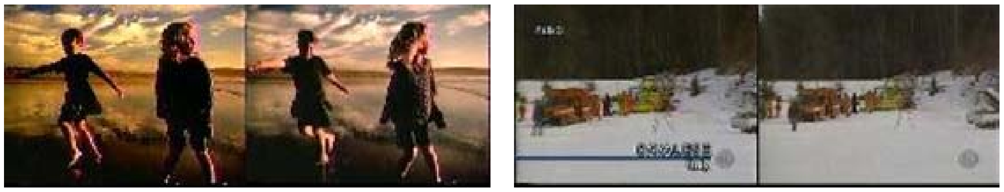
1.1.1 Image Linkage

In many applications of Multimedia Information Retrieval (MIR), the task to finding near-duplicate images becomes increasingly important - e.g.. detecting illegally copied images on the web. Image Linkage is the effort to solve *Near-Duplicate* problem by bridging two seemingly different areas, i.e., Multimedia and Biology.

Near-Duplicate problem can be classified into two folds – *Near-Duplicate Keyframes* (NDK) and *Near-Duplicate Image Detection* (NDID) problems. Due to the common characteristics of NDK and NDID, i.e., near-duplicate, it is not surprising to find the existing works [53, 101] that consider NDK and NDID as the same problem. However, they should be taken care of differently since each of them has different ways of duplication. Figures 1.1 (a) and (b) are the examples of NDID and NDK problems, respectively. Although all pairs of images seem to be very similar, they are slightly different from each other. In the examples of NDID (Figure 1.1 (a)), one image is copied from another and edited by adding a logo or changing the size. On the other hand, a pair of images in the examples of NDK (Figure 1.1 (b)) are captured from the same video but with different time or camera angle. Therefore, NDID has a different strategy and data sets for evaluation from NDK. In this thesis, we focus on NDID problem rather than NDK in terms of descriptors and matching methodology.



(a) Examples of NDID



(b) Examples of NDK

Figure 1.1. Near-Duplicate problems.

Despite many solutions to the NDID problem (to be surveyed in Section 2.1), by and large, contemporary solutions have focused on how to identify near-duplicate images accurately and fast by designing new algorithms, data structures, or models in a particular application or context. However, it is hard to apply newly-developed solutions to new data sets of different scenarios that requires using additional tools to visualize or analyze the results further. One way to approach the problem is to develop a suite of NDID algorithms and tools for "generic" usage so that the developed solutions can be used in a variety of situations by many users [63, 47, 45]. Another way is to extend an existing generic solution to solve the NDID problem so that one can leverage on the development of the generic solution and its user base [16].

In this thesis, we take the latter approach and apply one of such popular and generic solutions drawn from Biology, called BLAST (*B*asic *L*ocal *A*lignment *S*earch *T*ool) [1] to solve the NDID problem. The BLAST, developed in 1990, is one of the most popular (and best cited) algorithms for aligning biological sequence information such as nucleotides of DNA sequences and amino acid sequences of proteins. In its essence, given a query sequence s_q, BLAST finds top-k most similar sequences above a threshold from a database of sequences D using approximation heuristics based on Smith-Waterman algorithm [83].

Since Image Linkage also provides the solution to transfer images to gene sequences, we can access directly to many BLAST implementations and tools (to be detailed in Chapter 3).

1.1.2 Video Linkage

Due to the advancement of video recording/editing hardware/software, larger storage spaces, increasing network bandwidth, and the surge of Web 2.0 in recent years, the popularities of social media, video blogs, and user-created content videos have grown exponentially through online sharing sites such as YouTube[1] and Yahoo Video[2]. One of the challenges of such sites is, however, to identify and remove video clips that violate copyrights by illegally copying and editing scenes from other videos. This is because people often upload music videos, scenes of movies, commercials, or TV shows to such sites without proper authorization, often knowing its illegal nature. For instance, in 2007, Viacom[3], the owner of MTV, asked the removal of their copyrighted contents from YouTube that have been viewed more than 1.5 billion times. In 2008, Mediaset SpA, Italy's largest private broadcaster, filed a lawsuit for at least EUR500 million against YouTube for the unlawful use of the Italian company's audio and video files.

Therefore, it only gets more important for companies to be able to detect and remove such illegal contents in their collections to avoid serious legal and financial responsibilities. However, manual operations to detect and remove such pirated videos cannot keep up with demand because of the sheer number of video clips created every day. The situation will get only exacerbated as time passes. Naturally, therefore, automatic methods to detect illegally copied video clips *fast* and *accurately* in a large collection are greatly desirable. However, in general, detecting copied videos is a much harder problem than detecting similar videos, as illustrated in the following motivating example.

Example 1: Figure 1.2 shows selected key frames of an original video scene, (a), from the movie "Forrest Gump", three other scenes, (b)-(d), that are possibly copied and altered from (a), and one similar but non-copied scene, (e). We obtained all video scenes except (a) from YouTube. It is observed that the copied videos, (b)-(d), are edited by some basic operations such as integrating several

[1]http://www.youtube.com/
[2]http://video.yahoo.com/
[3]http://www.viacom.com/

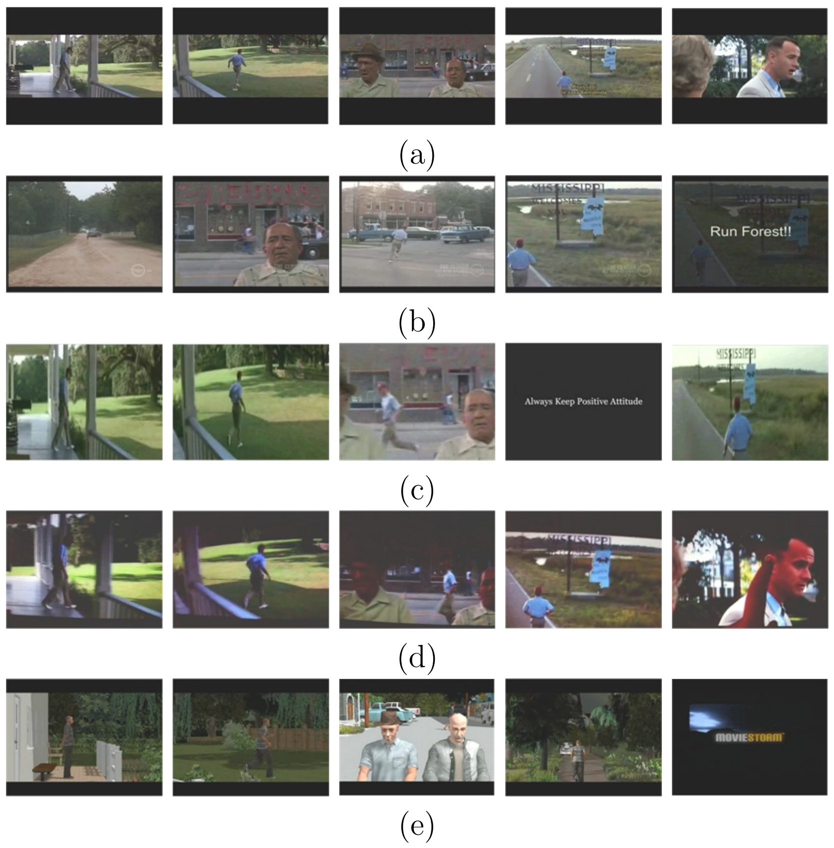


Figure 1.2. Key frames of an original video scene (a) from the movie "Forrest Gump", three copied/altered ones (b)-(d), and a similar one (e).

shots, changing format, and inserting title, transitions, and/or credit. For example, Figure 1.2(b) is edited by adding a credit scene, changing resolution, and cropping the original scene, while Figure 1.2(c) by adding a transition scene and changing the size of video. In particular, Figure 1.2(d) is (supposedly) illegally captured by a camcorder, which causes a lot of noises and unexpected modification like in the last key frame. However, on the other hand, Figure 1.2(e) is a similar video of Figure 1.2(a), but not copied one since it is an animation parodying "Forrest Gump". Therefore, a good system for copied video detection should conclude that only Figure 1.2(b)-(d) are copies of Figure 1.2(a) by utilizing editing methods. □

As shown in the example, it is hard to identify copied videos from similar videos. In order to remedy these problems, our copied video detection system acquires the record linkage algorithm which computes the similarity between two entities in a database. However, a video has different characteristics from a relational record, such as citations, usually used in record linkage problems. The utilization of video features and a video structure improves the accuracy as well as the processing time.

1.1.3 Parallel Linkage

As the growth of volumes in information, the size of database is enlarged in recent years. For example, the number of citations in citeseer is accumulated over 20 millions. Due to the number of records in a database and poor quality of records, it is nearly impossible to find all approximate same entities in a given time. In record linkage problems, many efforts to enhance the processing time to find the same entities from a large repository have been proposed such as blocking [90]. In addition, the poor quality data is very common in databases due to a variety of reasons, including transcription errors, data entry mistakes, lack of standards for recording database fields, etc. To remedy such problems, considerable recent works have focused on record linkage problem, defined by identification of all matching records between two collections of records. Such problems are, in general, known as the *de-duplication* problem or the *entity resolution* problem. The linkage problem frequently occurs in data applications (e.g., digital libraries, search engines, customer relationship management) and gets exacerbated when data are integrated from heterogeneous sources. For instance, a customer address table in a data warehouse may contain multiple address records that are all from the same residence, and thus need to be consolidated. For another example, imagine integrating two digital libraries, DBLP and CiteSeer. Since citations in two systems tend to have different formats, identifying all matching pairs is not straightforward.

Although the linkage problem has been studied extensively in various disciplines, the contemporary approaches have focused on how to identify "`matching`" records "`faster`" using a "`better`" distance function. Despite much advancement in solving the RL problem, however, the issue of efficiently handling large-scale RL problem has been inadequately studied. At its core, by and large, RL algorithms have a quadratic running time. For instance, a naive nested-loop style algorithm takes $O(|A||B|)$ running time to link two data collections, A and B, via all pair-wise comparisons as follows (assuming a distance function, *dist*(), and a preset threshold, θ):

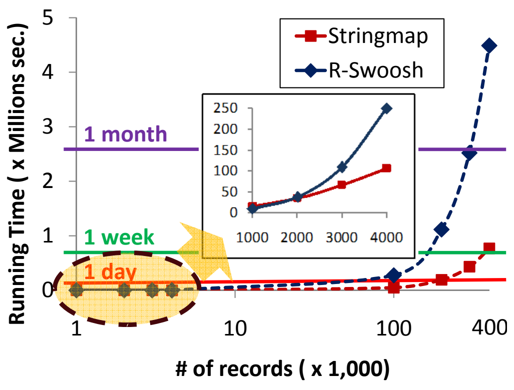
```
for each record $a_i (\in A)$
    for each record $b_j (\in B)$
        if $dist(a_i, b_j) < \theta$, then $a_i \approx b_j$
```

Due to its quadratic nature, when both sets A and B have a large number of records, the naive approach becomes prohibitively expensive. To remedy this problem, people have proposed many improvements – e.g., faster linkage approaches such as blocking [90] or computationally efficient distance functions such as upper-bound matching [71].

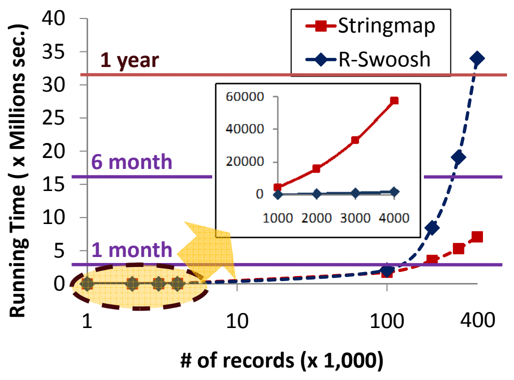
The inadequate performance issue of modern RL solutions for large-scale data collections gets much exacerbated when the RL problem becomes "iterative" in nature. In the conventional *match-only* RL model, when two records are determined to be matched, the pair is returned and no further action is needed. However, in the more general *match-merge* RL model [68], once two records r_a and r_b are matched and merged to a new record r_c, r_c needs to be re-compared to the rest of records *again* since it may incur new record matching. This makes the RL process "iterative" until convergence emerges. In such an iterative RL model with large-scale data collections, as demonstrated in Figure 1.3, conventional RL solutions become simply too slow. Taking two popular RL solutions, StringMap [48] and R-Swoosh [10], for instance, Figure 1.3 shows the running times (from the beginning of data load to the finish of the linkage) of both algorithms for self-cleaning data collections with short records (e.g., people names) and long records (e.g., citations). The number of records varies from 1,000 to 400,000 records[4]. Note that both algorithms are *not* suitable to handle the given RL task, showing quadratic increase of running times for small data (see Inset of Figure 1.3) or for large data collections (or for both).

Toward this scalability problem, however, we take a different approach from contemporary approaches, and study how to make a linkage algorithm "parallel." Therefore, we do *not* deliberate issues like generation of an efficient distance

[4]Running times for 1,000 – 4,000 records were obtained by actually running publicly available codes of both algorithms while the remaining values were fitted by the quadratic `polyfit` function in Matlab since neither codes finished within reasonable time.



(a) short record (e.g., names)



(b) long record (e.g., citations)

Figure 1.3. Running times of two RL solutions, StringMap and R-Swoosh, for two data sets (in self-clean case). X-axis is on Logarithmic scale.

function, *dist*() or how to add blocking/indexing to linkage structures (as used in blocked nested-loop or indexed join) in Parallel Linkage schemes.

1.1.4 Hashed Linkage

Another approach to enhance the processing time in general RL solutions having quadratic natures of computations is to use hashing/blocking techniques. However, a simple hashing/blocking algorithm is not accomplished the desirable efficiency - i.e., it still performs many non-necessary expensive comparisons.

For instance, advanced modern two-step RL algorithms avoid all pair-wise comparisons by employing sophisticated "blocking" stage so that comparisons are made against only a small number of candidate records within a cluster, thus achieving $O(|A| + |B| + \bar{c}(\bar{c} - 1)|C|)$ running time[5], where $\bar{c}$ is the average # of records in clusters and C is a set of clusters ("blocks") created in the blocking stage. Since # of clusters is usually much smaller than the size of data collection is and on average each cluster tends to have only a handful of records in it, often, $|A||B| \gg \bar{c}(\bar{c}-1)|C|$ holds. Therefore, in general, blocking based RL solutions run much faster than naive one does.

However, in dealing with large-scale data collections, this assumption no longer holds. RL solutions that did not carefully consider large-scale scenarios in their design tend to generate a large number of clusters and a large number of candidate

[5]Note that the hashing time per record is also considered as one unit of running time.

records in each cluster. In such a case, the cost of the term, $\bar{c}(\bar{c}-1)|C|$, alone becomes prohibitively expensive as shown in Figure 1.3.

Few researches were on construction hash structures to handle large-scale sets efficiently in record linkage problems. Recently, local sensitive hashing (LSH) and its variants have been introduced as indexing techniques for approximate similar search. LSH has been successful in similarity indexing in a high-dimensional vector space. The main requirements of LSH are to find the family of a locality sensitive functions. In such aspects, LSH indexing is properly applied to a collection of numerical data such as images and audio shown in [6, 5, 64] or strings with a substitution-based distance measure described in [3], but there are few suggestions in record linkage arena containing non-numerical information such as citations. In this thesis, toward this challenge, for the more general *match-merge* RL model, we present novel hashed record linkage algorithms that run much faster with comparable accuracy. Like Parallel Linkage, any *match* and *merge* functions can be used in Hashed Linkage frameworks. Throughout this thesis, we use same *match*, Jaccard similarity, and *merge*, $\cup\{tokens\}$, functions in both Hashed Linkage and Parallel Linkage.

1.2 Thesis organization

The rest of this thesis is organized as follows.

In Chapter 2, we review related works of linkage algorithms for specific usages to our research arena.

In Chapter 3, we introduce a new generic Image Linkage solution, BASIL, to solve near-duplicate image detection problem by bridging Multimedia and Biology. Since BLAST measures global similarity as well as local similarity, the similarity between images can be measured easily and directly by many well-developed implementations of BLAST, when the gene-like information is extracted properly from an image. Image Linkage solution includes how to extract and generate a gene string from various heterogenous image features, and how to exploit the extracted genes in BLAST system.

In Chapter 4, we study Video Linkage problems to find copied videos from video database. Due to the purpose of copied video detection (CVD) problem, Video Linkage returns a set of copied videos for an original query video. By maximal utilization of the video structure, we develop the hierarchical structure of video matching algorithm, which exploits a group-based record linkage solution. In addition, we shows how to construct algorithmic Video Linkage structures such as a pipe-lined filtering step.

In Chapter 5, we develop Parallel Linkage frameworks that parallelizes optimal sequential linkage frameworks. On both parallel and sequential frameworks, we show how to exploit input data characteristics such as dirtiness and cleanness to avoid unnecessary comparisons. Specially, in the parallel structure, we show how to partition tasks and reduce overhead. In addition, under various input data characteristics, we analyze the performance of all proposed sequential and parallel frameworks.

In Chapter 6, to enhance the processing time of general RL solution having *match* and *merge* steps, i.e., *iterative* nature of RL solution, like in Parallel Linkage, Hashed Linkage algorithms for non-numerical citation information are introduced. By implementing an iterative LSH-based hashing technique, called HARRA, the processing time and memory usage are improved drastically comparing to existing RL solutions. We show the fast and scalable performance of iterative Hashed Linkage solutions for any types or any sizes of input data sets.

Finally, we conclude this thesis by summarizing our methods and contributions, and outline future works in Chapter 7.

Chapter 2

Related Works

The general linkage problem has been known as various names – record linkage (e.g., [30, 13]), identity uncertainty (e.g., [73]), merge-purge (e.g., [44]), citation matching (e.g., [72]), object matching (e.g., [17]), entity resolution (e.g., [77]), and approximate string join (e.g., [39]) etc. Readers are referred to excellent survey papers (e.g., [90]) for the latest development of the linkage problem. Unlike the traditional methods exploiting textual similarity, Constraint-Based Entity Matching (CME) [81] examines "semantic constraints" in an unsupervised way. They use two popular data mining techniques, Expectation-Maximization (EM) and relaxation labeling for exploiting the constraints. [11] presents a generic framework, *Swoosh* algorithms, for the entity resolution problem. The recent work by [27] proposes an iterative de-duplication solution for complex personal information management. Their work reports good performance for its unique framework where different de-duplication results mutually reinforce each other (e.g., the resolved co-author names are used in resolving venue names).

Another recent trend in linkage problem is to exploit additional information beyond simple string comparison. For instance, [51] presents a relationship-based data cleaning (RelDC) which exploits context information for entity resolution, or [23] proposes a generic semantic distance metric between two terms using the page counts from the Web.

The rest of this chapter describes all related works focused on specific aspects with corresponding relevant linkage problems.

2.1 Image Linkage Problem

In particular, the task of determining if two images are near-duplicate or not, i.e., the NDID problem, becomes increasingly important in many applications. In general, such research on the NDID problem falls in two groups: *global feature* and *local feature* based approaches. The global feature based approach utilizes the similarity of two images using feature vectors extracted from entire images. For the similarity measure, most of CBIR methods can be used, such as color [34, 67], texture [82, 45], and shape [100] methods. However, due to the nature of CBIR system, they are very sensitive to small changes such as illumination or geometric distortion. The local feature based approach focuses on partial areas of image, i.e., keypoints, that can represent the characteristics of the entire image [53, 103]. The example of keypoint includes local dependency using region [19], distinctive interest point [53], or part-based [102]. To detect near-duplicated images, these approaches measure the similarity between two images by matching the keypoints [53, 34] and clustering the feature vectors [33, 67].

Table 2.1 shows the summary of a few representative solutions to the three variations of ND problem – NDID, NDK, and CBIR. The third and fourth columns describe the main descriptors and the methodology of matching used in the corresponding paper, respectively. In addition, data sets and evaluation metrics are indicated at the fifth and sixth columns of the table, respectively. We focused our survey only on the recent works, published after 2004. To limit the coverage of related literature, we do *not* survey recent proposals to detect near-duplicate documents (instead of images) such as [98, 86] and near-duplicate videos [79, 91]. With respect to images and other multimedia mediums, one of the most fundamental operations in MIR is to match two similar images [28, 35, 16, 43, 57]. There have been numerous researches on the matching problem with various names and applications – image retrieval [16], linking image [28], image searching [43, 87, 29], and image classification [35].

To our best knowledge, none of aforementioned works attempted to solve the NDID problem in MIR using the BLAST technology [1] in Biology. Furthermore, there have been very few attempts to use BLAST for applications outside of gene sequence alignment. First, [7] borrows the idea of multiple sequence alignment from

Problem	Paper	Descriptor	Matching	Data sets	Metric
NDID	CIVR07 [34]	g/l feature	L_2/point matching	create own with editing	PR
	WWW08 [67]	g feature	clustering	SapmArchive	accuracy
	MIR07 [33]	l feature	clustering	create own with editing	PR
	MM04 [53]	l feature	point matching	create own with editing	PR
NDK	MM04 [101]	key points	likelihood ratio	TRECVID 2003	PR
	CIVR07 [104]	key points	point matching	TRECVID 2006	PR
	MM07 [92]	g/key points	Euclidean/point matching	create from videos	PR
	ITM07 [103]	key points	point matching	TRECVID 2003	P(k)
CBIR	CIKM08 [87]	key points	k-NN search	Yorck (art images)	PR
	EDBT09- [29]	g feature	k-NN search	CoPhIR	Hit ratio
	MM08 [26]	key points	L_2/cosine similarity	Caltech-101	ROC

g(Global), l(Local)

Table 2.1. Survey of representative solutions to the Near-Duplicate problems.

BLAST in building a mapping dictionary, that is a lexicon of elementary semantic expressions and corresponding natural language realizations. Second, [59][1] uses BLAST to detect gene and protein names from journal articles by viewing an entire article as a query sequence and a set of known gene and protein terms as a database of sequences to match. Finally, [46] employs BLAST in the citation field segmentation problem – i.e., split a unsegmented citation string into known set of fields such as author, title, venue, and year. In the same sprit of these works but for the first time, we propose to use BLAST to solve the NDID problem. Since BLAST algorithm is based on substring matching and gapped extension, it can address the problems in both global and local feature based methods for NDID.

Even though NDK has different culture, it has been considered as the same problem as the NDID problem in literature. As shown in Table 2.1, almost all the approaches in the NDK problem [101, 32, 104, 103] are based on key points, i.e., point of interests or local descriptors, and point-wise matching, such as Locality Sensitive Hashing (LSH). Only a few of them consider both global and local features [92]. Due to the nature of the NDK problem, i.e., captured from a video, TRECVID data sets are mostly used for their evaluation.

Image Linkage solutions focus on solving NDID problem using BLAST to measure the similarity between images followed by extracting gene sequences representing image features.

[1]To our best knowledge, [59] is the first article, mentioning the idea of transforming English alphabet to DNA sequences of `A`, `C`, `G`, and `T`.

2.2 Video Linkage Problem

Generally, the video matching techniques are developed differently for two different problems, i.e., CVD(e.g., [99, 61, 85, 96, 94]) and CBVR (e.g., [31, 42, 25, 15, 88]) problems. While CBVR techniques focus on finding similar videos containing similar visual information, CVD techniques are to find video copies videos by various editing methods. A survey of comparative study regarding to features and distance measures used in the CVD problem can be found in several literatures [61, 41]. The state-of-art content-based CVD algorithms have been studied in many aspects such as feature selection, descriptor, comparison methods, and comparison structure. The signature or descriptor of a video is usually extracted from key frames in many applications [58, 96] since they are known to be a good representative. Key frames can be selected dynamically [20, 74] using relationship between frames.

To obtain a video signature from selected key frames, various features are suggested to extract appropriate descriptors. Because a frame as an image mostly provides spatial characteristic and a sequence of frames as a video contains temporal characteristic, most CVD techniques exploit spatio-temporal features. Based on spatio-temporal features, Law *et al.* [60] proposed to use local descriptors, Leon *et al.* [62] proposed video signature based on video tomography, and Wu *et al.* [93] proposed invariant pattern of visual information as a video descriptor. Xu *et al.* [96] obtained a video signature from compressed DCT domain.

In various CVD applications, to compute the similarity between videos using selected persistent features, diverse video matching structures were proposed . Depending on distortion types in video copies, each CVD algorithm suggested different features and different similarity measures. Tan *et al.* [84] utilized visual similarity and temporal consistency on a key frame sequence. In addition, they considered three levels to detect partially copied videos by considering a video signature as a set of sequences and, in turn, each sequence as a set of key frames. In [56], a spatio-temporal sequence matching is proposed to handle a wide range of modifications in videos, while Joly *et al.* [49] [50] proposed statistical similarity search using a local fingerprint based on an approximate search paradigm. A graph matching solution was proposed by solving a shortest-path problem in [20]. In their method, CVD has been thought as a partial matching problem that utilized Markov

	Used frames	Descriptor	*Match* function	Metric
MM09 [99]	uniformly selected	position correlation	dynamic programming	P, R
ICME09 [96]	fictional frames(DCT)	ordinal signature	sliding windows	P, R
ICME09 [76]	uniformly selected	visual word(SIFT)	sliding windows	P, R
MIR08 [9]	all frames	sensor fingerprint	correlation	FPR&FRR
VIE08 [93]	all frames	position correlation	histogram distance	FPR&FRR
ICME09 [94]	all frames	visual character string	self similarity	P, R
MIR08 [89]	uniformly selected	Hessian-based STIP	LSH & RANSAC	P, R
CSVT08 [20]	dynamically selected	ordinal signature	graph matching	P, R
MM08 [74]	dynamically selected	Glocal signature	indexing	P, R, S

OVP(Open Video Project), FPR(False Positive Rate), FRR(False Reject Rate), P(Precision), R(Recall), S(Speed)

Table 2.2. Comparison of a few recent CVD algorithms.

models for the probabilistic nature of the problem using key frames and candidate segments. The ordinal features that were computed by partitioned frames were utilized in [55]. In their method, the spatial matching of ordinal signatures is combined by the temporal matching of temporal video signatures from frame-partitioned temporal trails. For streaming videos, Yan *et al.* [97] introduced a video sequence similarity measure which was the composite of frame fingerprints extracted for individual frames. The key frames of incoming video are partially decoded to extract frame features. Kim *et al.* [58] computed the similarity of key frame pairs and then compared the time gap between matched key frames. Table 2.2 shows the summary and comparison of a few representative solutions to CVD problem.

Two most relevant record linkage techniques are *group record linkage* and *blocking*. Group record linkage [71] is a novel record linkage for matching data objects that have a group of elements in them. we extend the technique further to accommodate temporal information in videos. Blocking technique was first proposed by [54], and has been studied extensively [66] where initial rough but fast clustering is followed by more exhaustive record matching step. We apply the blocking idea in the pipelined Video Linkage algorithm. In addition, by analyzing the video structure, Video Linkage is developed as the hierarchical CVD solution with spatio-temporal features using group-based linkage algorithm in this thesis.

2.3 Parallel Linkage Problem

Parallel database join has been well studied (e.g., [80]). However, as mentioned in Section 1.1.3, parallel linkage has distinct characteristics, making the application of parallel join solutions non-trivial. In recent years, parallel linkage has studied in P-Febrl [22], D-Swoosh [12], and P-Swoosh [52]. P-Febrl is the parallelization model by Python module *Pypar* but no detailed algorithms are shown. Both D-Swoosh and P-Swoosh, parallel versions of Swoosh [10], are implemented by Java emulator, and runs in dual core processors. In parallel structures, D-Swoosh uses the *task graph model* while P-Swoosh uses the *master-slave model*. Our algorithm, implemented in distributed MATLAB, runs in real parallel environment (while P-Swoosh runs only in simulated environment). Our parallel solutions use the task graph model to keep simple control of load-balancing. All of works can adapt any ER algorithm as a *match* function.

2.4 Hashed Linkage Problem

More recently, the group of works on the *set-similarity join* (SSJoin) [78, 18] are relevant to this thesis. Optimization techniques developed in the literature (e.g., size filtering [4], prefix filtering [18], order filtering [8], or suffix filtering [95]) can be applied to the RL problem when the threshold model is used for measuring similarities. In a sense, all these optimization techniques aim at reducing the size of clusters via more sophisticated blocking techniques. However, none of these works considered the iterative RL with match-merge model.

On the other hand, the Locality-Sensitive Hashing (LSH) scheme [37, 2] was proposed to be an indexing method in approximate nearest neighbor (ANN) search problem. However, it still has limitations such as: how to find a family of locality-sensitive functions, how to handle excessive space issues due to many hash tables, and how to select right number of functions or tables? Recently, distance-based hashing (DBH) [5] is proposed to address the issue of finding a family of hash functions in LSH. Similarly, multi-probe LSH [64] is introduced to overcome the space issues. In many varieties of LSH-based algorithms, data sets are mostly specified to contain numerical features (e.g., image, audio, or sensor data). For

	Data	Model	Blocking	Metric
WWWJ06 [48]	54,000/133,101 names 20,000 DBLP	match only	R-tree based	running time, accuracy
VLDBJ09[10]	5,000 products 14,574 hotels	match-merge	N/A	running time, accuracy
ICDE08 [5]	15,853 UNIPEN 70,000 MNIST 80,640 hand images	match only	distance based (VP-tree like)	accuracy, efficiency
VLDB07 [64]	1,312,581 images 2,663,040 words	match only	LSH based	recall, query time, memory usage
ICDM06 [14]	Cora (1,295) DBGen (50,000)	match only	trained blocking	accuracy
AAAI06 [69]	864 restaurants 5,948 cars	match only	trained blocking	accuracy

Table 2.3. Comparison of a few recent RL algorithms.

string or sequence comparisons, substitution-based measures are proposed in [3]. In this thesis, we extend the LSH technique and propose the Iterative LSH (I-LSH) technique (and a suite of Hashed Linkage algorithms) that addresses the hash table size problem and deals with the intricate interplay between match() and merge() tasks in the RL problem. Empirically, we show that our proposal is able to address the efficiency issues well in terms of space and running time while maintaining high accuracy.

Table 2.3 shows the comparison among a few recent RL algorithms. Among these, in this thesis, we compare Hashed Linkage against *unsupervised* RL solutions such as [48, 10, 64] since they tend to be faster than supervised ones and their implementations are readily available.

Chapter 3

Image Linkage

In this chapter, toward Near-duplicate image detection (NDID) problem, we introduce Image Linkage solutions taking one of popular tools in Biology, called BLAST (Basic Local Alignment Search Tool) [1]. Extracting a proper gene sequence from an image makes possible to transform an image matching problem to biological gene sequence matching problem.

Our decision to use BLAST for the NDID problem is based on the observations that: (1) Both NDID and gene sequence alignment problems can be variants of approximate pattern matching. By capturing various content-based as well as semantic-based features of images in high dimensions and converting them into one-dimensional sequences of gene alphabets, both problems can be solved as the approximate pattern matching problem; (2) The alignment results from BLAST provide a robust similarity measure of *S-score* as well as a sound reliability measure of *E-value* with a statistical guarantee. Furthermore, BLAST is known for its fast running time and ability to handle a large amount of sequence data; and (3) BLAST has a wealth of advanced algorithms (e.g., nucleotide-nucleotide, protein-protein, protein-nucleotide, and position-specific version), implementations (e.g., NCBI BLAST, FPGA-based BioBoost BLAST, and open source versions), and tools (e.g., KoriBLAST for visualization and Parallel BLAST for parallel processing [75]) to leverage on. Therefore, if one can successfully transform the NDID problem to the gene sequence alignment problem, one can have an immediate access to a vast number of tools to use.

Our contributions are as follows: (1) To our best knowledge, this is the first attempt to solve the NDID problem using BLAST from Biology. We propose a generic framework, termed as BLASTed Image Linkage (BASIL), toward the NDID problem; (2) We propose the *Composite Conversion (CC)* table, a flexible way to generate gene sequences from a combination of multiple image features. Through the CC table, BASIL can use any set of image features (e.g., contents based or semantic annotations), suited for a particular data set on hand. This enables BASIL to be independent from a choice of image features; (3) From the Biolagy-inspired *Scoring Matrix* in BLAST, we develop and evaluate a set of novel scoring matrices that are particularly suited for the NDID problem and MIR domain; and (4) We present a comprehensive experimentations using a variety of real data sets for the NDID problem.

3.1 Problem Definition

Determining if two images are *similar* or not is a frequently studied task in the Contents-Based Image Retrieval (CBIR) problem. In particular, the task of detecting *near-duplicate* images becomes increasingly important in many applications of Multimedia Information Retrieval (MIR) – e.g., detecting illegally copied images on the Web [34] or detecting near-duplicate keyframe retrieval from videos [103]. We refer to such a problem as the *Near-Duplicate* (*ND*) problem, formally defined as follows:

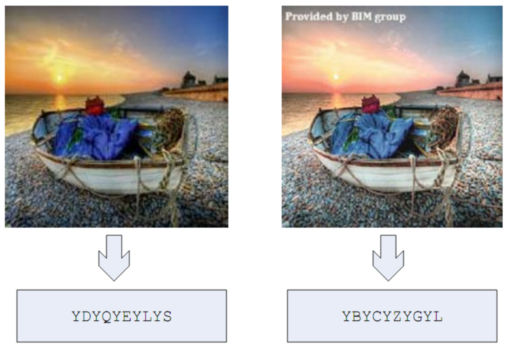
Near-Duplicate Problem. Given a set of query images I_q and a collection of source images I_s, for each query image i_q ($\in I_q$), find all images, I_r ($\subseteq I_s$) that are "near-duplicate" to i_q.

Note that we view the ND problem as a specialized problem of a general CBIR that aims to find similar images. That is, near-duplicate images are often generated by deliberate editing methods (e.g., changing colors/contrasts or cropping images), involuntary distortion (e.g., changing format/size) and variations of capturing conditions (e.g., different angle/time). Therefore, by definition, near-duplicate images are similar images, but *not* vice versa.

Depending on the types of duplicate images, ND problem can be classified into two folds: (1) Near-Duplicate Keyframes (NDK) [53, 101, 103], and (2) Near-Duplicate Image Detection (NDID) problems. Generally, NDK is defined as a pair of keyframes captured from a video, where the two keyframes are "near-duplicate" each other. On the other hand, NDID is a problem of detecting "near-duplicate" images for a query image from a source database.

3.2 BASIL: The BLASTed Image Linkage

In order to address the NDID problem, we propose BLASTed Image Linkage (BASIL) by adapting the BLAST algorithm. We believe that BLAST fits the NDID problem well for many reasons. In general, near-duplicate images tend to have near-identical characteristics which in turn are mapped to a long gene sequence of identical alphabetical "hits."



S		Y	Q	Y	L	Y	M	Y	P	Y	L
	0	0	0	0	0	0	0	0	0	0	0
Y	0	**10**	4	10	4	10	4	10	4	10	4
Q	0	4	**19**	13	13	11	10	9	8	7	6
Y	0	10	13	**29**	23	23	21	20	19	18	17
I	0	4	19	23	**38**	32	31	30	29	28	27
Y	0	10	13	29	32	**42**	42	41	40	39	38
K	0	4	13	23	38	**57**	**57**	51	50	49	48
Y	0	10	11	23	32	51	51	**67**	61	60	59
F	0	4	10	21	31	57	57	**61**	70	64	63
Y	0	10	9	20	30	51	51	**67**	64	80	74
I	0	4	8	19	29	50	50	61	**77**	74	76

(a) Example images and sequences (b) Example of Sequence Alignment

Figure 3.1. Sequence alignment example.

Figure 3.1(a) illustrates an example of two ND images. The image on the right is modified from the one on the left via operations such as changing contrast, compression and adding logo. The protein sequences below images are generated by BASIL using Y component in YUV color domain. The similarity of the two sequences i_s and i_q can be evaluated by means of a local alignment (e.g., Smith-Waterman) algorithm. In the algorithm, the alignment is operated on two-dimensional matrix S in which each cell $S(i,j)$ keeps a score of the current matching. S is initialized with $S(i,0) = 0, 0 \leq i \leq |i_q|$ and $S(0,j) = 0, 0 \leq j \leq |i_s|$, and is built as follows:

$$S(i,j) = \mathtt{max} \begin{cases} S(i-1, j-1) + s(i_q(i), i_s(j)) \\ \mathtt{max}_{0 \le k \le i-1} \{S(k,j) - \sigma(i-k)\} \\ \mathtt{max}_{0 \le k \le j-1} \{S(i,k) - \sigma(j-k)\} \\ 0 \end{cases} , 1 \le i \le |i_q| \ \mathtt{and} \ 1 \le j \le |i_s| \,,$$

where $s(i_q(i), i_s(j))$ is the pairwise score of i-th letter of i_q and j-th letter of i_s in scoring matrix, $\sigma(k)$ is the gap penalty of a gap of length k. Figure 3.1(b) shows the result of the alignment of the two sequences. By utilizing BLAST, alignments can be done much faster than the dynamic programming algorithms. Through the process seems complicated, BLAST which is a heuristic algorithm for sequence alignment has much better efficiency . Furthermore a single BLAST query can match a sequence against the whole database of sequences, and find the similar sequences above a certain threshold, instead of pairwise matching in other *match* algorithms.

BLAST is known to find such “hits” fast and accurately. Even if some variations occur in near-duplicate images due to editing, BLAST can still overcome the distortion of image features by the ungapped extension of multiple “hits”.

3.2.1 Overview of BASIL

Figure 3.2 shows the overview of the proposed BASIL framework. First, for each image i_s ($\subseteq I_s$, source image set), we extract a set of features, $\mathcal{F}$, and transform $\mathcal{F}$ to a (either DNA or protein) gene sequence, s_s. All the generated sequences are stored in the BLAST database D. Similarly, a query image i_q is also transformed to a corresponding gene sequence s_q. Then, using the BLAST algorithm and an appropriate scoring matrix, s_q is compared to sequences in D and top-k near-duplicate sequences (and their corresponding images) are returned as an answer.

When we generate gene sequences from images, depending on *how* we translate *which* of the extracted image features, we end up with different gene representations. For the gene sequence generation, we propose various methods depending on how we “translate” the extracted feature values to DNA or protein-like alphabets by selecting different features including various image contents as well as sematic

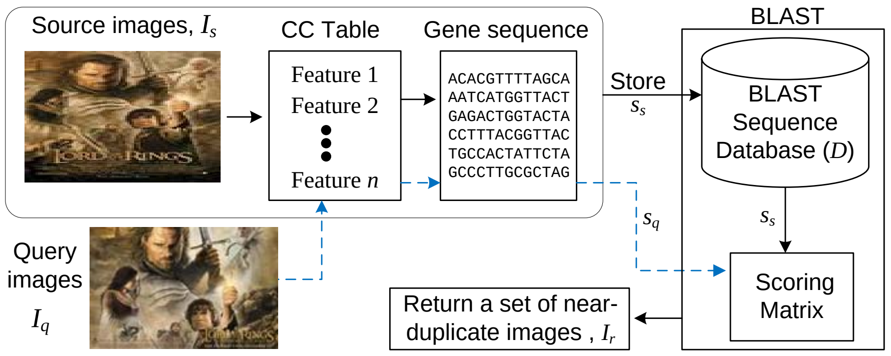


Figure 3.2. Overview of BASIL.

annotations. In particular, since it is difficult to find a set of image features that work universally well for all data sets, it is important to devise a solution orthogonal to the choice of image features. Toward this first challenge, we propose the *Composite Conversion* (CC) table that contains both pre-defined conversion rules and candidate image features so that users can select desirable features and gene sequences depending on a given data set (see Section 3.2.2). In addition, the second challenge is to devise solutions in BASIL such that the kernel of BLAST algorithm and implementation should *not* be changed to make existing tools remain useful. Instead, our proposal sits atop BLAST algorithm and manipulates query and source image sequences. For instance, the scoring matrix (that reflects the similarity between different gene alphabets) used in BLAST is originally adjusted to the Biology domain. Therefore, we propose variations of new scoring matrices that reflect the characteristics of near-duplicate image matching scenarios (see Section 3.2.3).

3.2.2 The Composite Conversion (CC) Table

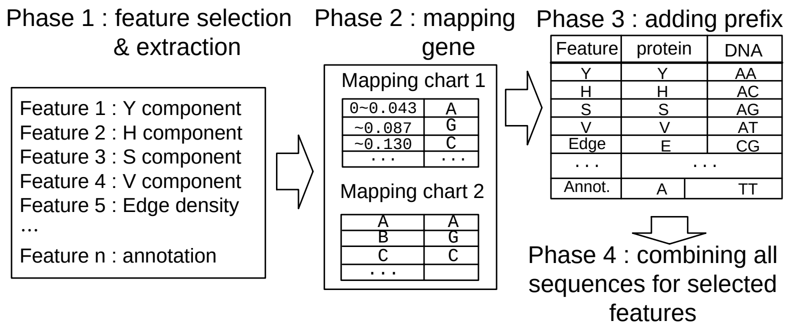
Although images are distorted by deliberated editing methods, some image features still maintain their characteristics. For example, by changing the brightness, we lose the similarity of luminance, but the image still contains similar color information. To find appropriate features for BASIL, therefore, we have tested and selected a variety of features of three groups: color-based ($\mathcal{F}_C$, Y in YC_bC_r and H in HSV), texture-based ($\mathcal{F}_T$, edge density by Law's texture energy), and semantic ($\mathcal{F}_S$, keywords and annotations) features. Each image, i, will be divided to some

n-Value	Pro.	DNA	n-Value	Pro.	DNA
$0 \sim \delta$	A	AAC	$\sim 13\delta$	L	ATT
$\sim 2\delta$	R	CCT	$\sim 14\delta$	K	ATG
$\sim 3\delta$	N	CAG	$\sim 15\delta$	M	CAC
$\sim 4\delta$	B	AAG	$\sim 16\delta$	F	ACT
$\sim 5\delta$	D	ACC	$\sim 17\delta$	P	CCC
$\sim 6\delta$	C	AAT	$\sim 18\delta$	S	CGC
$\sim 7\delta$	Q	CCG	$\sim 19\delta$	T	CGG
$\sim 8\delta$	Z	GAG	$\sim 20\delta$	W	CTG
$\sim 9\delta$	E	ACG	$\sim 21\delta$	Y	GAC
$\sim 10\delta$	G	AGC	$\sim 22\delta$	V	CTC
$\sim 11\delta$	H	AGG	$\sim 23\delta$	X	CTT
$\sim 12\delta$	I	AGT	$\sim 24\delta$		

(a) Mapping chart for $\mathcal{F}_C$ and $\mathcal{F}_T$ ($\sigma = \frac{1}{23}$)

letter	Pro.	DNA	letter	Pro.	DNA
A	A	AAC	N	N	CAG
B	B	AAG	O	Y	CAT
C	C	AAT	P	P	CCC
D	D	ACC	Q	Q	CCG
E	E	ACG	R	R	CCT
F	F	ACT	S	S	CGC
G	G	AGC	T	T	CGG
H	H	AGG	U	Z	CGT
I	I	AGT	V	V	CTC
J	X	ATC	W	W	CTG
K	K	ATG	X	X	CTT
L	L	ATT	Y	Y	GAC
M	M	CAC	Z	Z	GAG

(b) Mapping chart for $\mathcal{F}_S$



(c) The Composite Conversion Table

Figure 3.3. The CC table with two mapping charts.

blocks, say 16×16 macro blocks, and both color- and texture-based features are computed within a macro block while semantic feature is computed from associated keywords or annotations of i. Then, the feature set, $\mathcal{F}$, is the union of $\mathcal{F}_C$, $\mathcal{F}_T$, and $\mathcal{F}_S$.

In order to generate the gene sequences from $\mathcal{F}$, we consider two types of sequences used in BLAST: (1) a protein sequence is made of 23 alphabets (i.e., A, B, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, X, Y, and Z), while (2) a DNA sequence is made of four gene alphabets (i.e., A, C, G, and T). BASIL can take both protein and DNA sequences.

The *Composite Conversion (CC)* table, as illustrated in Figure 3.3(c), contains various image features and two mapping charts. A mapping chart in Figure 3.3(a) is used for mapping numeric values obtained from image contents, while another in Figure 3.3(b) is for literal words obtained from descriptive annotations. For $\mathcal{F}_C$ and $\mathcal{F}_T$, we use the normalized values to use the same mapping chart in Figure 3.3(a).

Since we have 23 gene alphabets for protein, for the best transformation of feature values, we place the normalized real values to 23 bins, as shown in Figure 3.3(a). For DNA gene sequences, since 4 gene letters are not enough to express 23 bins, 3-bit combination of 4 letters is used for each bin. For $\mathcal{F}_{\mathcal{S}}$, similarly, each literal alphabet is mapped to gene alphabet(s) by pre-defined rules, as shown in Figure 3.3(b). For protein sequences with 23 protein letters, we add 3 more artificial letters (X, Y, and Z) to have 1-to-1 mapping to 26 literal alphabets. For DNA sequences, we use 3-bit combination letters with `A`, `G`, `C`, `T`. Figure 3.3(c) shows the four phases of the CC table to generate the final gene sequences:

- **Phase 1 (Feature selection & extraction)** Among all available image features, a set of features are selected (by users) and normalized. The selection of features depends on the availability of features as well as the characteristic of the given image sets. In addition, the size of a macro block that determines the length of gene sequences is fixed.

- **Phase 2 (Mapping to gene letters)** According to the mapping tables in Figure 3.3(a), the normalized feature values from Phase 1 are mapped to appropriate gene letters. If semantic features are used in Phase 1, for instance, they are also mapped to gene letters using Figure 3.3(b). At this phase, one can decide whether to use DNA or protein genes as the final representation.

- **Phase 3 (Adding prefix)** Because of the limitation of gene alphabets, the same gene letters can be used in different features. For ensuring stronger connection within the same features, therefore, each letter from phase 2 is combined with corresponding letters representing a specific feature, as shown in Figure 3.3(c). This phase can be skipped if only one image feature is selected in phase 1.

- **Phase 4 (Combining all features)** All gene sequences from different features are combined. The final output sequence of the CC table thus captures all features of an image holistically. This phase is also skipped if only one image feature is selected in phase 1.

Since an individual feature in a CC table is very independent, the features in a CC table can be obtained by separating homogeneous components of an image such as color components. With the same reason, the features in a CC table can be acquired very heterogeneously. For example, all of image color components, texture information, meta data (such as resolution, format, and date), and annotations can be included as features in a CC table. The final gene sequence of an image captures all selected homogeneous and heterogeneous components, and is passed through BLAST to compare all features at once.

3.2.3 The Scoring Matrix

As discussed earlier, BLAST algorithm is known as the most powerful tool for searching/comparing gene sequences in the field of bioinformatics, since domain specific information are considered using scoring matrix. When two sequences are compared in BLAST, a similarity score is computed to quantify the quality of the pair-wise alignments. For this task, BLAST uses a scoring matrix that includes all possible pair-wise scores of letters in 2-dimensional matrix. For the scoring matrix, Percent Accepted Mutation (PAM), and BLOcks SUbstitution Matrix (BLOSUM) are popular. The PAM is built on theoretical analysis while the BLOSUM is on more empirical results.

Since both matrices are originally created for biological data in mind, they are not suitable for BASIL with image data. In order to take the full advantage of using BLAST algorithm in BASIL, therefore, we propose to use the following scoring matrices: (1) **Uniform matrix**, shown in Figure 3.4(a), assigns uniform score for each identity and substitution. For example, "1" is assigned for all identities (i.e., diagonal), and "-1" is assigned for the others of the matrix. The uniform matrix provides a uniform weight for all pair-wise alignments. We use the uniform matrix as the baseline; and (2) **Gaussian distributed matrix**: The uniform matrix cannot capture the diverse characteristics of features used in BASIL. For example, red and orange colors are more similar than red and blue in terms of hue (H) color domain. However, such similarity cannot be represented in the uniform matrix. To address the problem, we propose a gaussian distributed matrix. In general, the gaussian distributed matrix is good for numeric features, such as $\mathcal{F}_C$ and $\mathcal{F}_T$, since

$$
\begin{array}{c|cccccc}
 & \mathbf{A} & \mathbf{R} & \mathbf{N} & \mathbf{B} & \mathbf{D} & \cdots \\
\mathbf{A} & 1 & -1 & -1 & -1 & -1 & \cdots \\
\mathbf{R} & -1 & 1 & -1 & -1 & -1 & \cdots \\
\mathbf{N} & -1 & -1 & 1 & -1 & -1 & \cdots \\
\mathbf{B} & -1 & -1 & -1 & 1 & -1 & \cdots \\
\mathbf{D} & -1 & -1 & -1 & -1 & 1 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}
$$

(a) Uniform

$$
\begin{array}{c|cccccc}
 & \mathbf{A} & \mathbf{R} & \mathbf{N} & \mathbf{B} & \mathbf{D} & \cdots \\
\mathbf{A} & 10 & 8 & 3 & -2 & -6 & \cdots \\
\mathbf{R} & 8 & 10 & 8 & 3 & -2 & \cdots \\
\mathbf{N} & 3 & 8 & 10 & 8 & 3 & \cdots \\
\mathbf{B} & -2 & 3 & 8 & 10 & 8 & \cdots \\
\mathbf{D} & -6 & -2 & 3 & 8 & 10 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}
$$

(b) Gaussian

Figure 3.4. Characterized scoring matrices for BASIL.

it considers the relationship between two letters that are mapped from numeric feature values. As shown in Figure 3.4(b), gaussian distributed values are assigned with the identity (i.e., diagonal) as an average. Note that as long as the values follow the Gaussian distribution, one can have a different gaussian distributed matrix than shown in Figure 3.4(b).

There are several important advantages to employ characterized scoring matrices into BASIL: (1) The semantics of image features can be represented using the matrix; (2) The different weights can be applied for image matching using identities' values in the matrix; (3) Positive credits and negative penalties can be adjusted for exact/fuzzy matched and unmatched letters, respectively; (4) The more sophisticated scoring matrix than Uniform or Gaussian (e.g., Probabilistic, Linguistic, or Trained matrices) can be easily added to the CC table. We will leave this as future work.

3.3 Experimental Validation

3.3.1 Set-Up

Gene sequence generation was done on Intel Core 2 Duo (1.8GHz, 2GB RAM, Windows XP Home), and gene sequence matching was done by WU-BLAST 2.0[1] on IBM Z60t (Intel Pentium-M 1.6GHz, 1.5GB RAM, Ubuntu 7.10). The CC table in BASIL is implemented in Matlab 7.0. For the generated gene sequences, WU-BLAST 2.0 creates BLAST DB for all source images, and find near-duplicate

[1]http://www.advbiocomp.com/blast/obsolete/

Real world data set				Modified data set	
Dark Knight (DK)		The Lord of The Ring (LR)		Flickr (FK)	
Category	# of images	Category	# of images	Category	# of images
Back	19	Poster with annotations	20	Each original image (240 original images)	1 original image + 12 edited images
Batman	17				
Face	27				
Fire	42				
Joker	9				
Wsos	41	Others with annotations	200		
Others	1108				
Sub-total	1263	Sub-total	220	Sub-total	3120

Total number of images : 4603

Table 3.1. Image data sets.

images by measuring the similarity between query sequences and source sequences in the database.

Data Sets. As test sets, two real-world data sets and one edited data set are used in our experiments: Dark Knight (DK), The Lord of The Rings (LR)[2], Flickr (FK)[3] and its variations by image editing tools.

Both DK and LR are one of the most popular films, and thus their near-duplicated images such as posters and still images from a video clip are easily found on the web. We collect near-duplicate images from Google image site[4], and classify them manually. For the intensive and realistic test, we also collect irrelevant images to make the large size of a whole image data set. Specially, for DK image set, 6 different keywords are used to obtain relevant images, then 6 image categories exist in our image image sets from Dark Knight movie. The 7th category is a collection of irrelevant images. For the LR data set, one image category (LR poster) is selected with additional semantic annotations such as the title, file name, and descriptions of images. The other category consists of irrelevant images with their annotations. Thus, we can have test the heterogeneous image features in BASIL. The details are in the first column of "Real world data set" in Table 3.1.

For the FK data set, the original images are collected by using different key-

[2]collected from http://iamges.google.com/
[3]http://www.flickr.com/
[4]http://images.google.com

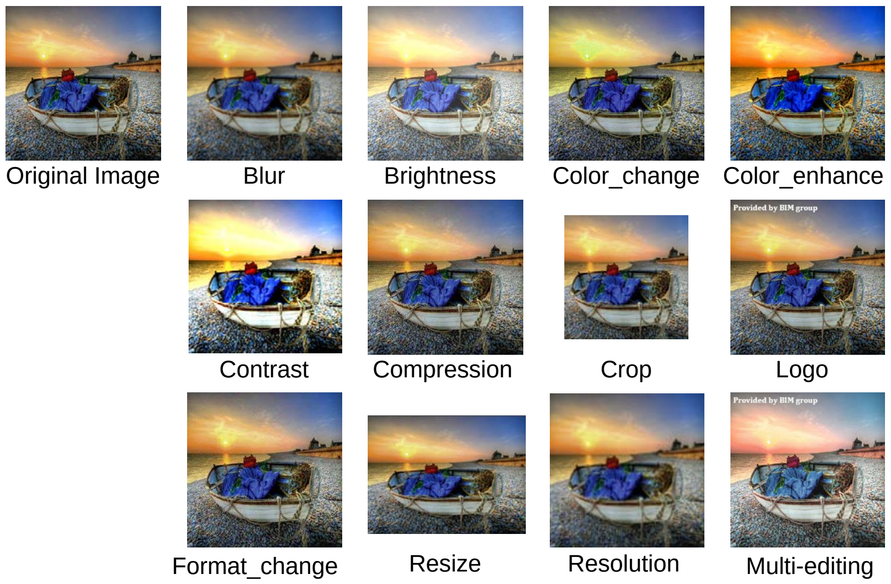


Figure 3.5. Examples of edited images.

words and random selection from Flicker site. For one original image, 12 near-duplicate images are generated by the typical editing methods, i.e., *blur, changing brightness, changing format, changing color, color enhancement, changing contrast, compression, crop, adding logo, changing resolution, changing size, and multi-editing (e.g. crop + compression + logo)*. Note that one original image represents one image category. In each category, the edited images are placed as near-duplicate images. Since we have 240 original images, total number of categories is also 240, and total number of images in FK image data sets is 3120. Figure 3.5 shows the examples of the edited images. These editing methods are also popular functions in image editing softwares such as GIMP or Photoshop. The details of FK data set are shown in the second column of Table 3.1.

The CC table translates all images to gene sequences that are stored in BLAST DB. Since the LR data set contains semantic annotations, in the phase 1 of the CC table, we also select a semantic feature for LR. At the end, we have tested BASIL with 4,603 images. In order to validate the effectiveness of BASIL system, we provide *two* different genres from three data sets. One is a real-world image set excluding/including semantic annotations, i.e., DK and LR, respectively. The other is an image set containing original images and their variations, i.e., FK. Thus, the image sets of DK and LR are combined to generate one BLAST DB.

Note that image contents and semantic annotations are heterogeneous, but a CC table makes it possible to combine them together to generate a gene sequence. Therefore, all images containing different features can be combined and compared in BASIL system. The FK image set is also possible to be combined, but we put the data set aside to show the robustness of BASIL system against various editing methods.

Evaluation Metrics. As the evaluation metrics, we mainly use the average precision and recall. Suppose that T denotes a set of true near-duplicate images according to the ground truth and R denotes a set of images found by an algorithm. Then, we have: precision=$\frac{|R \cap T|}{|R|}$ and recall=$\frac{|R \cap T|}{|T|}$. In addition, the given setting of the NDID problem can be evaluated in two models – *threshold* model (i.e., an algorithm retrieves all images whose similarity to the query image is above a pre-set threshold) and *top-k* model (i.e., an algorithm retrieves top-k near-duplicate images sorted in descending order). In the following, to be able to measure the accuracy of algorithms more carefully, we use the top-k model, by varying k. With a small k, recall is likely to be low while precision is expected to be high. Finally, in the presentation, to show a complete behavior of both precision and recall using a small space, we use the PR (precision-recall) graph where X-axis and Y-axis show the changes of recall and precision, respectively.

In DK and LR image sets, for each category except others, 9-15 query images are randomly chosen. We compute average precision and recall of returned *top-k* sequences (corresponding images) from BLAST, achieving 95% confidence levels with 6.5-9.9 confidence intervals. With FK image set, for each category (total 240 categories), 10 query images are randomly chosen to achieve 99% confidence levels with 2.9 confidence interval.

3.3.2 Comparison within BASIL

3.3.2.1 Comparison between DNA vs. Protein

First, we compare the accuracy of BASIL with both DNA and protein gene sequences. Using the DK image set, Figure 3.6 shows the accuracy comparison respect to the effect of DNA and protein gene sequences. Whether BASIL uses DNA (thus 3-bits of 4 alphabets) or protein (23 alphabets), shown in Mapping

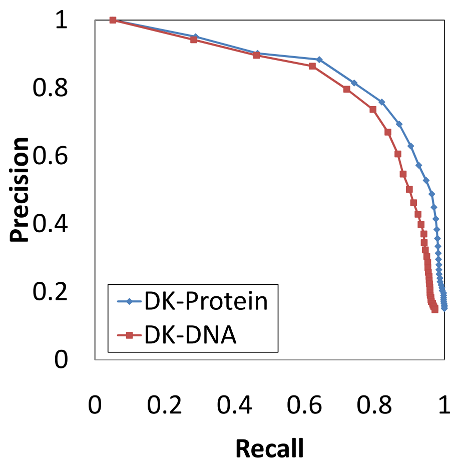


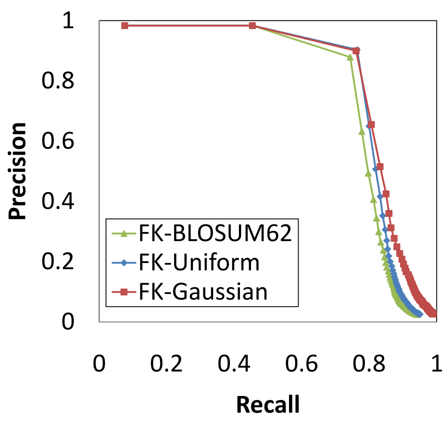
Figure 3.6. Comparison between DNA and protein gene sequence representations.

charts of Figure 3.3, to capture image features does not seem to affect the overall accuracy significantly. However, overall, Figure 3.6 shows that protein gene sequences appear to yield better accuracy than DNA sequences do, because protein provides more distinct alphabet letters than protein. Therefore, the gap between numerical values are less ambiguous at transforming those to gene letters. Even though the strict difference exists between protein letters, the small difference between numerical values can be compensated by a particular scoring matrix such as Gaussian matrix, used in this comparison. For brevity, from here forward, all experiments are done using protein gene sequences.

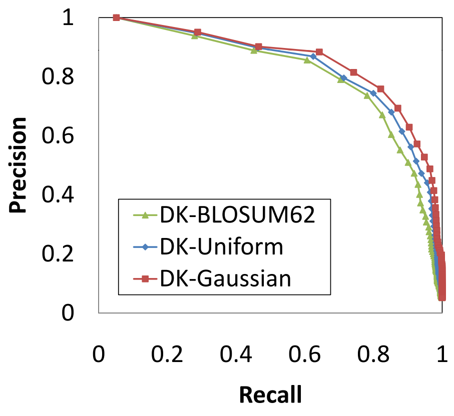
3.3.2.2 Comparison among scoring matrices

As mentioned early in Section 3.2.3, a scoring matrix is a two-dimensional matrix that specifies the score between all possible pair-wise amino acids. In order to pursue high accuracy for Biology data, by default, WU-BLAST uses BLOSUM62 to measure the similarity score between gene sequences. However, the gene sequences from BASIL are born from image sets which may have different characteristics from Biology data. Thus, we design two scoring matrices to be suitable for image features in our scenario: *Uniform* matrix (for exact matching between gene letters) and *Gaussian* matrix (for similar matching).

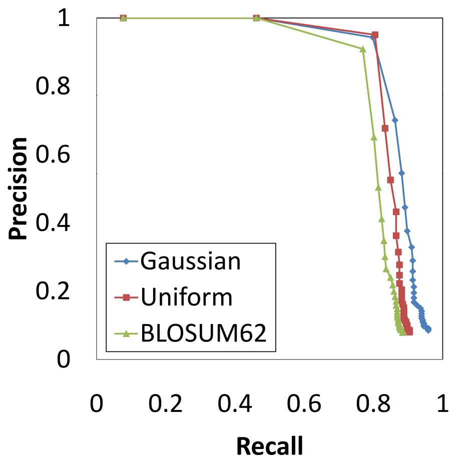
Here, we compare the accuracies by using three different matrix, on both FK and DK. Figures 3.7(a)–(b) show the average precision-recall graph of using differ-



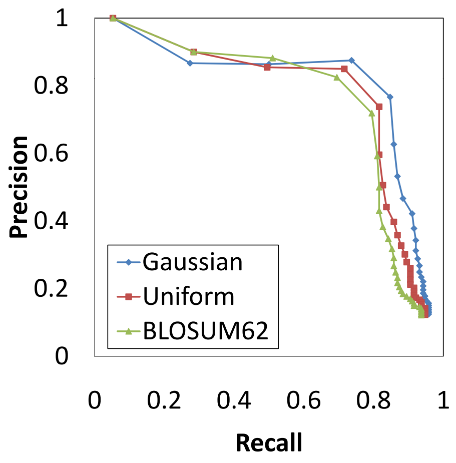
(a) Edited set (FK)



(b) Real set (DK)



(c) Mountain (FK)



(d) Back (DK)

Figure 3.7. Comparison among scoring matrices.

ent scoring matrices for FK and DF data sets, respectively. As seen in the figures, on both image sets, the Gaussian scoring matrix performs the best, followed by the uniform and BLOSUM62 matrices. In order to see the results more precisely, Figures 3.7(c)–(d) show the performance results of two representative categories of the FK data set, i.e., `Mountain`, and DK data set, i.e., `Back`, respectively. It is clear that the Gaussian scoring matrix provides the best results among all scoring matrices as shown all results. Due to the nature of Gaussian distribution, BASIL can capture the similarity between adjacent numerical values of image features with a Gaussian scoring matrix, even though the feature values are translated to different gene letters by a mapping chart. Hereafter, all experiments are done using Gaussian scoring matrix.

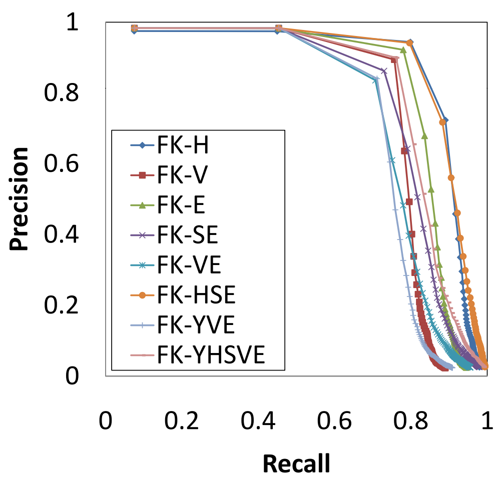
3.3.2.3 Comparison among image features

We next evaluate the overall performance of different image features for BASIL discussed in Section 3.2.2, i.e., color-based ($\mathcal{F}_C$), texture-based ($\mathcal{F}_T$), and semantic ($\mathcal{F}_S$) features. As mentioned earlier, the feature values are first extracted from images to generate gene sequences. In other words, the performance of BASIL depends on the quality of $\mathcal{F}$. In this experiment, for features ($\mathcal{F}_C$) and ($\mathcal{F}_T$), an image is divided into 16 macro blocks for feature extraction and 23 protein letters are chosen to map feature values to gene sequences by a matching chart in Figure 3.3 (a). For feature ($\mathcal{F}_S$), the collected meta data such as title and description of an image are translated to 23 protein gene sequences by a matching chart in Figure 3.3 (b).

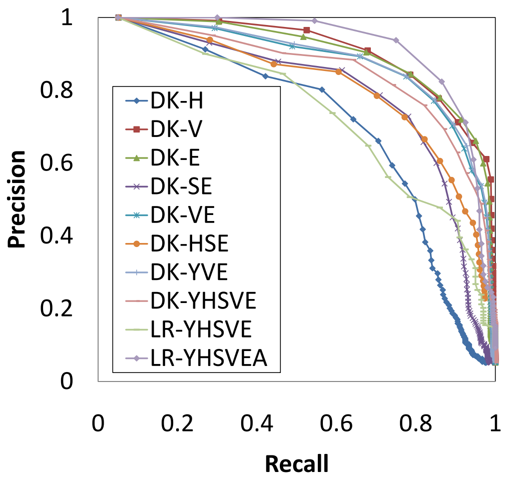
We used 6 popular features in the CC table: Y component from YC_bC_r color domain, H, S, V components from HSV color domain, Law's edge energy component, and semantic feature. Y, H, S, V, E (energy), and A (semantic annotation) stand for each component, respectively. For the evaluation of the effect of selected features, among these 6 features, one can choose any combination of them. In Figures 3.8(a) and (b), we evaluate the performance of various combination of features including 1 feature (i.e., H, V, and E), 2 features (i.e., SE and VE), 3 features (i.e., HSE and YVE), 5 features (i.e., $YHSVE$), and all 6 image features in the CC table. Note that feature A is only available in the LR image set.

The performance of BASIL is not only influenced by the selected features but also the characteristic of image sets. A particular feature selection can show positive effect in one image set, but have poor result for another set. For example. if the colors of an image are mostly black and white, then near-duplicate images are usually affected by Y feature instead of H feature or others. For colorful images, H feature should be more sensitive in near-duplicate images than Y feature under conscious or unconscious image distortion. Similarly, E feature is more influenced within very textured images. As a result, the unsusceptible features to distortion provide higher return accuracy.

For the FK set, all of H, V, and E features have a high precision until when recall becomes 0.5. However, afterward, H feature becomes the best, while both V and E show a relatively low precision. In the real-world data set (DK and LR), in Figure 3.8(b), both V and E give the best result overall in terms of both precision



(a) Features on FK



(b) Features on DK & LR

Figure 3.8. Comparison among image features.

and recall, while H yields the worst accuracy. This symptom can be explained as follows. In the real-world data set (DF and LR), people often copy and modify images with color change/enhancement functions before images are uploaded to the Web. The color feature is more sensitive to H than the others. On the other hand, the FK data set is generated by 12 editing methods. However, only 2 of them are related to the color in FK data set. Therefore, the results show that V and E for DK and LR are better features than H.

When multiple features are selected, one can usually gain the average performances of different features. For instance, in the real-world image set (DK and LR), the accuracy with multiple features is always between those with an individual feature. However, note that the combination of features from image contents usually outperforms the average of the accuracies from individual feature selection. This is because the accuracy of BASIL system follows the top-k model. That is, even though the similarity between genes are averaged from multiple features, the similarity ranking from BLAST can be changed when features are combined. Another benefit of using multiple features combined is the improved robustness of BASIL for unknown image sets. In this chapter, note that all sets are set to be unknown since we do not analyze the characteristic of data sets by sample or whole images in data sets. As a result, by combining all six features, $YHSVEA$, in LR image set, we achieve the highest accuracy from BASIL system shown in Figure 3.8

(b).

3.3.2.4 Comparison among editing methods

Here, using the FK image set, we compare the impact of different editing methods on the accuracy of BASIL. For this evaluation, we select 5 features ($YHSVE$) in the CC table. Each category in FK consists of one original image with 12 edited images. Since there exist 240 original images, we have 3,120 images total in all 240 categories. Therefore, when an original image is queried, ideally, all 12 edited images must be returned at high ranking. The ranking of returned images are out of 3,120 images in FK data set, i.e., 3,120 gene sequences in BLAST DB. Hence, the best and worst rankings are 1 and 3,120, respectively. Since BLAST DB also contains original images, note that the best ranking of edited images always starts from 2. For example, in an ideal case (i.e., for an original image as a query, BASIL returns all 12 edited images with an image by itself), the returned images should be ranked between 2 and 13.

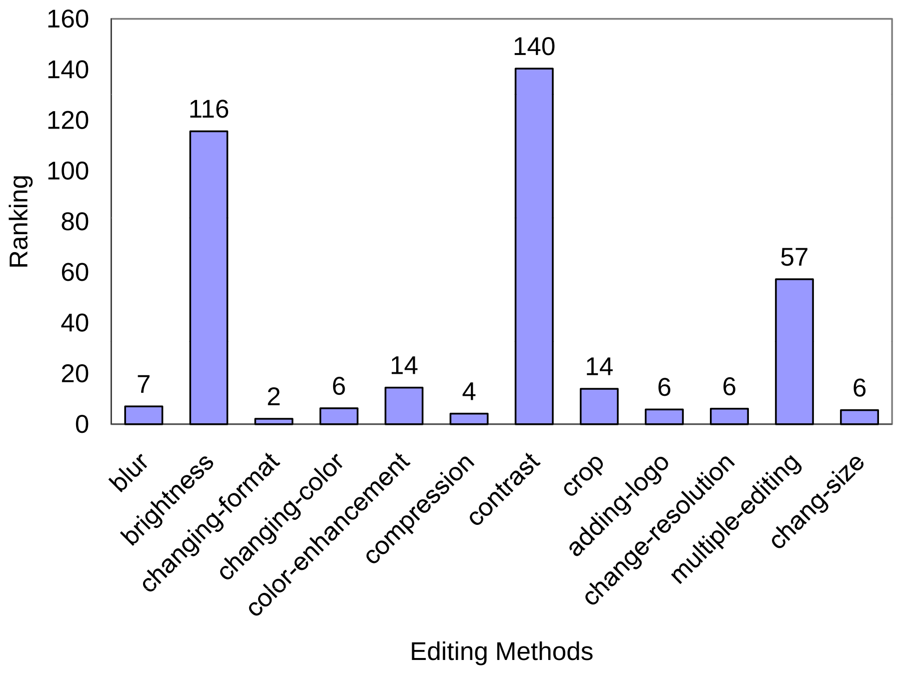


Figure 3.9. Comparison among editing methods.

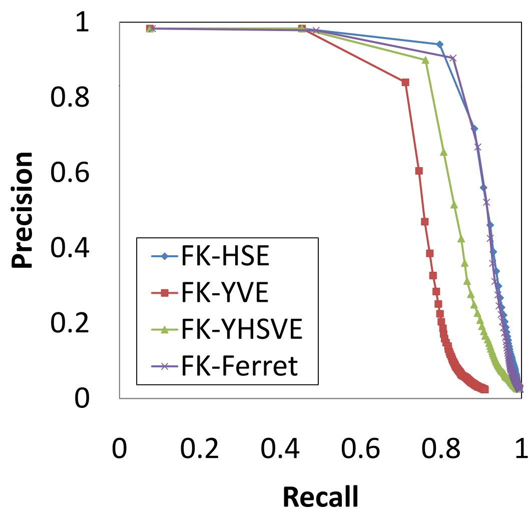
Figure 3.9 shows the average rank of 12 expected images in the returned list from BASIL with 12 different editing methods. BASIL system reveals that the average ranking of the expected duplicate images is about 2 for the best case and about 140 for the worst case. Among various editing methods, the accuracy is relatively low (i.e., the ranking of expected images are low) when images are distorted by “contrast change” and “brightness change”. Such editing methods are not considered fully in BASIL because our feature set, $\mathcal{F}$, focuses on real-world data set where the most frequent editing methods are related to the change of color, format or size. Therefore, the accuracy of BASIL with the editing methods related to color, format or size, are much better than that of “contrast change” and “brightness change”. For example, the best accuracy is shown by the editing method of “format change”. In other words, with “format change”, the edited images are ranked around 2 or 3 for all input queries, when we expect 13 or less for the ideal ranking of returned images from BASIL. Overall, BASIL is robust on various editing methods that are typically used by image editing tools.

3.3.3 Comparison against Other Methods

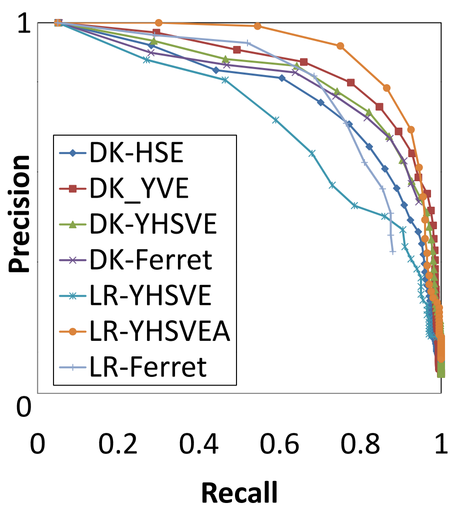
Due to the difficulty in obtaining the implementations of other NDID methods (summarized in Table 2.1), instead, we compare the performance of BASIL against two publicly available non-NDID solutions – Ferret for CBIR and ND_PE for NDK.

Comparison with Ferret. Here we first evaluate BASIL against one of the state-of-the-art CBIR alternatives, Ferret, from the CASS project at Princeton[5]. Ferret is a toolkit for content-based similarity search for various data types including digital image. The result using the FK image set is shown in Figure 3.10(a), where Ferret and HSE exhibit the best results while the balanced $YHSVE$ is behind them after the recall of 0.5. With the DK set, BASIL achieves the best accuracy using the YVE feature selection as shown in 3.10(b). Overall, both BASIL with $YSHVE$ features and Ferret show similar accuracy. One of the benefits of the CC table in BASIL is that it enables to combine any heterogenous features to the final gene sequences. For instance, heterogeneous features such as semantic or content-based one can be uniformly represented in gene sequences. As a result, Figure 3.10(b) shows that the line of LR-YHSVEA (6 features including a *semantic information*) significantly outperforms Ferret.

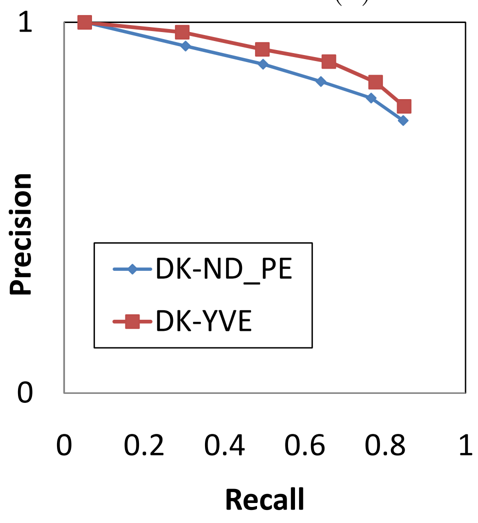
[5]http://www.cs.princeton.edu/cass/



(a) vs. Ferret on FK



(b) vs. Ferret on DK & LR



(c) vs. ND_PE on DK

Figure 3.10. Comparison BASIL to Ferret and ND_PE.

Comparison with ND_PE. The ND_PE is a near-duplicate keyframe (NDK) detection toolkit based on local features of images, developed by Video Retrieval Group (VIREO) from City University of Hong Kong[6]. In ND_PE, a set of local interest points of images are extracted and represented in PCA-SIFT descriptor. The similarity of two images is then determined on the degree of matches between two sets of keypoints such as a bipartite graph matching. We compare the accuracy of ND_PE and BASIL with YVE features on DK data set in Figure 3.10(c). In this

[6]http://vireo.cs.cityu.edu.hk/research/NDK/ndk.html

test, 9–10 images in each category are selected to measure the similarity against all images in the data set. The top-30 returned images per query are used to generate the average PR graph[7]. Figure 3.10(c) shows that overall the accuracy of BASIL outperforms that of ND_PE for the real near-duplicate data set, DK. Note that ND_PE was originally designed to solve the NDK problem, not the NDID problem. Since both the NDK and NDID problems are slightly different, therefore, direct comparison between the results of BASIL and ND_PE should be interpreted with much care.

3.4 Summary

In this chapter, we studied a novel Image Linkage solution, named as BLASTed Image Linkage (BASIL), to solve the near-duplicate image detection (NDID) problem by bridging two seemingly unrelated fields – *Multimedia* and *Biology*. In BASIL, we use the popular gene sequence alignment algorithm in Biology, BLAST, to determine the similarity between two images. To be able to handle flexible transformation from diverse image features to gene sequences, we also proposed the Composite Conversion (CC) table that hosts different images features and prefixed transformation rules. The validity of BASIL is positively measured using two real image sets on various aspects.

[7]The implementation of ND_PE crashed for a few pairs of images in testing. In preparing the PR graph of Figure 3.10(c), such images were ignored.

Chapter 4

Video Linkage

In this chapter, toward the **CVD** (Copied Video Detection) problem, we present a novel solution, termed as Video Linkage, that is based on the record linkage techniques in Databases – i.e., to determine if two entities represented as relational records are approximately the same or not. Informally, given a video v_q and a collection of videos V, we aim at detecting all videos from V that are copies of v_q. For the detection, the Video Linkage technique is based on the following observations. First, a video can be represented as a "group" of *shots* and in turn a shot as a "group" of image *frames*. Furthermore, inherently, there is a temporal ordering along shots and frames of a video. Second, two videos are deemed to be similar if two groups of shots are similar, and the notion of "groups" can be well captured by graphs. Therefore, we can measure the similarity between two videos by means of graph-based similarity measures such as bipartite matching. Note that the similarity between two shots are also obtained by the graph-based similarity measure between two groups of key frames. Thus, in the proposed hierarchical structure, at the lower level we measure the similarity between two groups of key frames, i.e. shot similarity, and at the upper level we measure the similarity between two groups of shots, i.e. video similarity. This will be elaborated in detail in section 4.2. Third, if a video v_a is illegally copied from a video v_b, then v_a and v_b must be somehow similar (having similar but altered shots and frames). Therefore, we can prune dissimilar videos out using simpler and faster similarity measures for fast detection of copied videos.

Our contributions are as follows: (1) We propose a method to transform a video

Notation	Description
V	a set of videos
v / s / f	a video / a shot / a frame
$sim_v(v_1, v_2)$	video-to-video similarity
$sim_s(s_1, s_2)$	shot-to-shot similarity
$sim_f(f_1, f_2)$	frame-to-frame similarity
θ / δ / ρ	threshold for sim_v / sim_s / sim_f
$\mathcal{F}$ / $\mathcal{S}$	feature set / signature for a frame

Table 4.1. Summary of notations in Video Linkage.

to a group of shots and in turn a shot to a group of frames to enable hierarchical group based matching idea. A set of shots is first identified from a video, and in turn a small number of key frames are extracted from each shot; (2) We propose five efficient group based shot similarity measures: (i) two exact measures, SL and $NCSL$, based on the maximum weight bipartite matching and maximum weight non-crossing bipartite matching, (ii) one greedy measure, gSL, and (iii) two approximate measures, aSL and $aNCSL$. Further, we show the partial order relationship among five measures and their utility; (3) We propose two shot linkage methods for the **CVD** problem using (i) *standalone* and (ii) *pipelined* frameworks. Further, we propose hierarchical Video Linkage structures inheriting shot linkage algorithms for the **CVD** problem; and (4) Our proposed algorithms outperform methods presented in CIVR 2007 competition in terms of speed and accuracy using MUSCLE-VCD-2007 benchmark data set. For instance, the proposed algorithm runs 3.67 times faster on average and achieves the recall of 1.0 as opposed to the recall of 0.86 of the competition. In addition, ours achieve the precision and recall of 0.94 and 0.93 respectively, using videos downloaded from YouTube with popular editing methods.

4.1 Problem Definition

Throughout this chapter, we use notations in Table 4.1. The copied video detection problem can be modeled as either *selection* problem (i.e., select top-k copied videos) or *threshold* problem (i.e., find all copied videos above a threshold). To make the presentation simple, hereafter, we use the threshold version of the problem. Formally, the copied video detection problem is defined as follows:

Copied Video Detection (CVD). Given a set of query videos V_q and a collection of source videos V_s, for each query video v_q ($\in V_q$), detect all copied videos, V_c ($\subseteq V_s$) that contain either duplicated or altered video shots from v_q.

Generally, CVD systems are different from CBVR (Content-Based Video Retrieval) Systems. The result of CVD is a set of copied videos that are illegally edited videos from copyright protected videos, while that of CBVR is a set of similar videos containing similar contents. Therefore, a CVD system should utilize both the editing methods as well as similar contents. For the editing, in this chapter, we consider six popular methods such as *cut and paste*, *cropping*, *adding logo/text*, *resizing*, *changing video format/resolution*, and *adding title, transition, or(and) credit*, observed in sites like YouTube and Yahoo Videos. For the existing benchmark data set of MUSCLE-VCD 2007, harsh distortion methods are applied in the copied videos such as *color change*, *comcording*, *flip*, *zooming*, *shift*, *contrast change*, *blur*, *adding noise*, *vertical deformation*, *inserting caption*, *changing phases*, *letter box*, *zooming*, and *changing gamma*.

A naive solution to the **CVD** problem performs the quadratic pair-wise computation between two video sets, V_q and V_s causing $O(|V_q||V_s|)$ complexity. In turn, the similarity between two video clips requires the quadratic pair-wise comparisons between two groups of sets. Therefore, one of the objectives of our proposal is to find the computationally efficient solutions for the **CVD** problem.

Note that other relevant issues such as feature selection or multimedia indexing are not fully considered in this chapter. Instead, we will select the most basic features with properly weighting The proposed hierarchical Video Linkage algorithms are not affected by the choice of indexing or feature selection, since the proposed structures can be independently applied without indexing steps using any selected features.

4.2 Hierarchical Video Linkage

A video contains a hierarchical structure with temporal orders. In other words, a video is a group of shots, and a shot is a group of frames, and both shots and frames are in time sequence. Therefore, we need three different levels of similarity

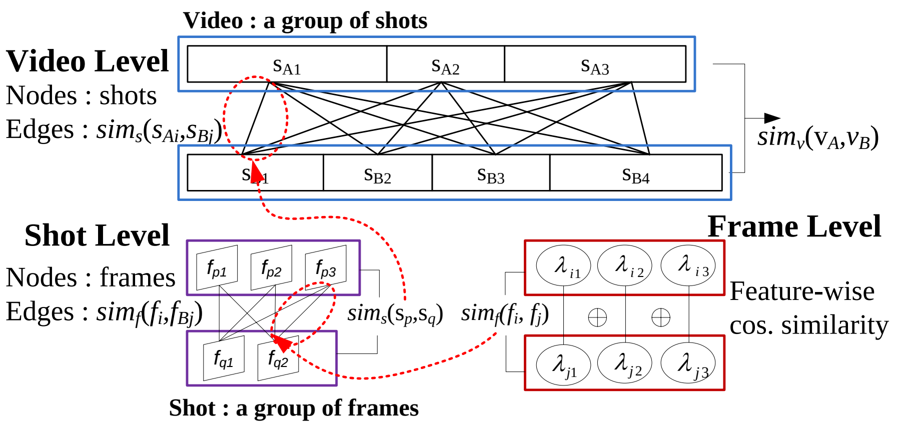


Figure 4.1. The structure of hierarchical Video Linkage.

measures in the proposed hierarchical Video Linkage. The frame level is to compute $sim_f(f_i, f_j)$ by the compound difference of feature vectors between frames. The shot level and video level are to compute $sim_s(s_p, s_q)$ and $sim_v(v_a, v_b)$, respectively. At each level, the efficient methods are acquired to reduce dense computations. The overall structure of the hierarchical Video Linkage is shown in Figure 4.1.

4.2.1 Video Pre-Processing

In this section, we briefly discuss pre-processing for videos: (1) extracting a set of feature, $\mathcal{F}$, (2) detecting shot boundaries, and (3) selecting key frames.

First, we extract a set of feature ($\mathcal{F}$) consisting of three characterized features from each frame, i.e., (1) HSV color histogram (λ_H) as a global feature representative, (2) YCbCr color layout (λ_Y) as a local feature representative, and (3) motion vector histogram (λ_M) as a temporal feature representative. The frame-to-frame similarity measure, $sim_f()$ for $\mathcal{F}$, is then defined as follows:

$$sim_f(f_1, f_2) = \sum_{\lambda \in \mathcal{F}} w_\lambda \cdot sim_\lambda(f_1, f_2) \tag{4.1}$$

where w_λ is a weight of a feature λ such that $\sum_{\lambda \in \mathcal{F}} w_\lambda = 1$, and $sim_\lambda()$ is a similarity between two frames with respect to the feature λ.

Second, to detect shot boundaries [70], we use $sim_f()$ with $\mathcal{F} = \{\lambda_H\}$ in Equation (4.1). In other words, we compute the similarity of two consecutive frames, f_i and f_{i+1}, and if $sim_f(f_i, f_{i+1})$ is more than a certain threshold, then f_i

and f_{i+1} are considered as a shot boundary. Otherwise, both f_i, f_{i+1} belong to the same shot.

Third, to select the key frames per shot, we borrow a conventional technique from [36], where the frame similarity values obtained in detecting shot boundaries are re-used to construct a similarity curve that shows how the contents change over an entire shot. The high curvature indicates a significant change around the frames while the flat indicates no change. Those frames in high curvatures of a similarity curve are, thus, selected as key frames. These key frames represent a shot and is used to extract $\mathcal{F}$.

4.2.2 Frame Level Similarity

By obtaining key frames per shot, we can reduce the number of computations while keeping the characteristic of a shot. For the initial step, the edge values between two frames is obtained by Equation (4.1). Individual $sim_\lambda(f_i, f_j)$ is computed by cosine distance between feature values, and w_λ can be selected equally to meet $\sum_{\lambda \in \mathcal{F}} w_\lambda = 1$.

4.2.3 Shot Level Similarity

By completion of obtaining all edge values by computing $sim_\lambda(f_i, f_j)$ between key frames, we are ready to discuss how to measure the similarity between two shots utilizing the "group" information and shot characteristics. In essence, we significantly extend the group-based record linkage techniques in [71] to exploit the temporal order among frames, and propose five shot linkage measures.

4.2.3.1 Exact shot linkage measure

We first propose two exact shot linkage measures, i.e., shot linkage (SL, see Def. 4) and non-crossing shot linkage (NCSL, see Def. 6).

Definition 1 (Shot as Group) *A shot s is captured as a group of key frames: g=$\{f_1, \ldots, f_m\}$.* $\Box$

Given two groups g_1 and g_2, one of the simplest and most intuitive similarity measures is the *Jaccard* similarity, defined as $\frac{|g_1 \cap g_2|}{|g_1 \cup g_2|}$. By generalizing the Jaccard similarity to be able to handle approximate matching between two frames, we can use the bipartite graph idea.

Definition 2 (Weighted Bipartite Graph for Videos) *Given two groups of image frames,* $g_1 = \{f_{11}, f_{12}, \ldots, f_{1m_1}\}$ *and* $g_2 = \{f_{21}, f_{22}, \ldots, f_{2m_2}\}$*, a weighted bipartite graph is a bipartite graph* $G = \{N, E, \Omega\}$*, where* $N = g_1 \cup g_2$*,* $E = g_1 \times g_2$*, and* $\Omega = \{\omega(i,j) | \omega(i,j) = sim_f(f_{1i}, f_{2j})\}$ □

Definition 3 (Maximum Weight Bipartite Matching) *A* matching *is a set of pairwise non-adjacent edges in* E*. A* maximum weight bipartite matching M *is a matching* M *such that* $\sum_{(f_{1i}, f_{1j}) \in M}(\omega(i,j))$ *is the maximum.* □

Conceptually, the numerator and denominator of the Jaccard similarity are equivalent to the sum of weights in M and the number of nodes in N, offset by the number of edges in M, respectively. Based on this observations, now, we propose our group based shot similarity measure as follows:

Definition 4 (Shot Linkage) *For the bipartite group* $G = \{N, E, \Omega\}$ *over two groups of image frames,* $g_1 = \{f_{11}, f_{12}, \ldots, f_{1m_1}\}$ *and* $g_2 = \{f_{21}, f_{22}, \ldots, f_{2m_2}\}$*, the shot linkage measure,* $SL_{\omega,\rho}$*, is the normalized weight of the the maximum weight bipartite matching* M_1*:*

$$SL_{\omega,\rho}(g_1, g_2) = \frac{\Sigma_{(f_{1i}, f_{2j}) \in M_1}(\omega(i,j))}{m_1 + m_2 - |M_1|}$$

such that $\omega(i,j) \geq \rho$*, where* ρ *is a user-set minimum threshold for edge similarity.*□

Note that the denominator of $SL_{\omega,\rho}$ adds up the number of edges in the matching, M_1, and the number of "unmatched" frames in each of g_1 (i.e., $m_1 - |M_1|$) and g_2 (i.e., $m_2 - |M_1|$). When the numerator is large, it captures the intuition that two videos have "many" similar frames. Similarly, when the denominator is small, it captures the intuition that a large fraction of frames in two groups are similar. Note also that we do *not* consider all pair-wise edges between two groups. Rather, we prune away those edges whose ω is substantially low (i.e., $\omega(i,j) < \rho$). Not only this early pruning helps improve the accuracy of shot linkage technique, it speeds up computation significantly since all subsequent algorithms work faster on a "sparse" bipartite graph.

In addition to $SL_{\omega,\rho}$, a shot in a copied video tends to have inherent temporal order among frames although they are altered by many editing methods (e.g., *adding logo and subtitle* and *changing contrast and brightness*). Although the visual effects and characteristics might have changed, temporal order among frames is still intact. Although it is possible to change temporal order among copied frames, we believe such cases are rare. Therefore, we extend $SL_{\omega,\rho}$ to take advantage of the order among elements of groups.

Definition 5 (Non-Crossing Bipartite Matching)
Consider an "ordered" bipartite graph G=$\{N, E, \Omega\}$ over groups g_1 and g_2, where nodes in each group are numbered in increasing order from top to bottom. Two edges between nodes, $e_1 = (i, j)$ and $e_2 = (p, q)$, are said "crossing" iff ($i \leq p$ and $j \geq q$) or ($i \geq p$ and $j \leq q$). Then, a non-crossing matching *is a subset of edges M_2 ($\in E$) such that no two edges of M_2 cross, including crossing at nodes. A* maximum weight non-crossing bipartite matching *is a non-crossing matching such that $\sum_{(f_{1i},f_{1j})\in M_2}(\omega(i,j))$ is the maximum.* □

When applied to the problem of matching two shots, s_1 and s_2, a non-crossing bipartite matching captures the intuition that once a frame f_{1i} ($\in s_1$) and a frame f_{2j} ($\in s_2$) match each other, no crossing matching can occur (i.e., the sequential order among frames must be followed). By capitalizing on this intuition, then we define our second group based shot linkage measure as follows:

Definition 6 (Non-Crossing Shot Linkage) *For the "ordered" bipartite graph G=$\{N, E, \Omega\}$ over two groups g_1=$\{f_{11}, f_{12}, \ldots, f_{1m_1}\}$ and g_2=$\{f_{21}, f_{12}, \ldots, f_{2m_2}\}$, the non-crossing shot linkage measure, $NCSL_{\omega,\rho}$, is the the normalized weight of the maximum weight "non-crossing" bipartite matching M_2:*

$$NCSL_{\omega,\rho}(g_1, g_2) \quad = \quad \frac{\Sigma_{(f_{1i},f_{2j})\in M_2}(\omega(i,j))}{m_1 + m_2 - |M_2|}$$

such that $\omega(i,j) \geq \rho$, where ρ is given. □

Both $SL_{\omega,\rho}$ and $NCSL_{\omega,\rho}$ are guaranteed to be between 0 and 1. Furthermore, from the definitions, the following follows.

Lemma 1. *For two groups g_1 and g_2:*

$$NCSL_{\omega,\rho}(g_1, g_2) \leq SL_{\omega,\rho}(g_1, g_2)$$

where ρ is given. ■

4.2.3.2 Boosting shot linkage measure

Both $SL_{\omega,\rho}$ and $NCSL_{\omega,\rho}$ capture the intuitions of two matching shots very well. However, both measures are computationally costly because of the requirement that "no node in the bipartite graph can have more than one edge incident on it." The known algorithms to find maximum weight bipartite matching and maximum weight non-crossing bipartite matching have time complexities of $O(N^2E)$ [40] (e.g., Hungarian or Bellman-Ford) and $O(N^2)$ [65], respectively. In search of faster shot linkage measures, therefore, we relax this requirement of the bipartite matching using the greedy strategy.

Definition 7 (Greedy Shot Linkage) *Consider the bipartite graph G=\{N, E, Ω\} over two groups g_1=\{f_{11}, f_{12}, ..., f_{1m_1}\} and g_2=\{f_{21}, f_{12}, ..., f_{2m_2}\}.*

- *For each frame $f_i \in g_1$, find a frame $f_j \in g_2$ with the highest ω ($\geq \rho$) and denote the set of all such frame pairs as $F1$.*
- *Symmetrically, for each frame $f_j \in g_2$, find a frame $f_i \in g_1$ with the highest ω ($\geq \rho$) and denote the set of all such frame pairs as $F2$.*

Then, a greedy shot linkage measure, $gSL_{\omega,\rho}$, is:

$$gSL_{\omega,\rho}(g_1, g_2) = \frac{\Sigma_{(f_{1i}, f_{2j}) \in F1 \cup F2}(\omega(i,j))}{m_1 + m_2 - |F1 \cup F2|}$$

such that $\omega(i,j) \geq \rho$, where ρ is given. □

Note that neither $F1$ nor $F2$ may be a matching. In $F1$, the same frame in g_2 may be the target of more than one frame in g_1 (thus violating the definition of "matching"). Similarly, in $F2$, the same frame in g_1 may be the target of more than one frame in g_2.

Lemma 2. *The greedy shot linkage measure, $gSL_{\omega,\rho}(g_1, g_2)$, can be computed in $O(N + E \log E)$ time on the bipartite graph G={N, E, Ω} over two groups g_1 and g_2.* ■

The usefulness of the greedy group-based shot linkage measure, $gSL_{\omega,\rho}$, lies on the fact that its similarity value is always an over-estimation of true similarity value of $SL_{\omega,\rho}$. Therefore, the value of $gSL_{\omega,\rho}$ is not bounded between 0 and 1.

Lemma 3. *For two groups g_1 and g_2:*

$$SL_{\omega,\rho}(g_1, g_2) \leq gSL_{\omega,\rho}(g_1, g_2)$$

where ρ is given. ■

Next, we propose two heuristics based approximations of the bipartite matching.

Definition 8 (Approximate Shot Linkage) *Consider a bipartite graph G={ N, E, Ω} over two groups g_1={f_{11}, f_{12}, …, f_{1m_1}} and g_2={f_{21}, f_{12}, …, f_{2m_2}}. For two empty sets, $R1$ and $R2$,*

- *Sort all edges ($\in E$) by ω in decreasing order.*
- *For each edge e_{ij}=(f_{1i},f_{2j}) in order, if neither f_{1i} nor f_{2j} is visited, add e_{ij} into $R1$ and mark f_{1i},f_{2j} as "visited" (initially, all nodes are set as "unvisited").*
- *For each edge e_{ij}=(f_{1i},f_{2j}) in order, if e_{ij} does not "cross" any edges from $R2$, add e_{ij} into $R2$.*

Then, two approximate shot linkage measures are:

$$aSL_{\omega,\rho}(g_1, g_2) = \frac{\Sigma_{(f_{1i},f_{2j})\in R1}(\omega(i,j))}{m_1 + m_2 - |R1|}$$

$$aNCSL_{\omega,\rho}(g_1, g_2) = \frac{\Sigma_{(f_{1i},f_{2j})\in R2}(\omega(i,j))}{m_1 + m_2 - |R2|}$$

such that $\omega(i, j) \geq \rho$, where ρ is given. □

Measure	Time Complexity
$SL_{\omega,\rho}(g_1, g_2)$	$O(N^2E)$
$NCSL_{\omega,\rho}(g_1, g_2)$	$O(N^2)$
$gSL_{\omega,\rho}(g_1, g_2)$	$O(N + E\log E)$
$aSL_{\omega,\rho}(g_1, g_2)$	$O(E\log E)$
$aNCSL_{\omega,\rho}(g_1, g_2)$	$O(E^2)$

Table 4.2. Time complexities of five shot linkage measures.

Note that unlike $F1$ and $F2$, both $R1$ and $R2$ are still a matching since no nodes participate more than once. However, they may *not* be a maximum matching. Therefore, the following holds.

Lemma 4. *For two groups g_1 and g_2:*

$$aNCSL_{\omega,\rho}(g_1, g_2) \leq aSL_{\omega,\rho}(g_1, g_2) \leq SL_{\omega,\rho}(g_1, g_2)$$

$$aNCSL_{\omega,\rho}(g_1, g_2) \leq NCSL_{\omega,\rho}(g_1, g_2)$$

where ρ is given. ■

Since the bipartite graph that we deal with is often very sparse (i.e., $N \gg E$) due to the early pruning from the constraint of $\omega(i, j) \geq \rho$, these two approximate shot linkage measures can be computed faster than their corresponding exact shot linkage measures.

Lemma 5. *On the bipartite graph G={N, E, Ω} over two groups g_1 and g_2, both $aSL_{\omega,\rho}(g_1, g_2)$ and $aNCSL_{\omega,\rho}(g_1, g_2)$ can be computed in $O(E\log E)$ and $O(E^2)$ times, respectively.* ■

The time complexities of five Video Linkage measures are summarized in Table 4.2.

Example 1. Consider a bipartite graph $G=\{N,\ E,\ \Omega\}$, where $\Omega=\ \{\omega(f_{11}, f_{22}) = 0.8,\ \omega(f_{12}, f_{21}) = 0.6,\ \omega(f_{12}, f_{23}) = 0.3,\ \omega(f_{13}, f_{21}) = 0.9,\ \omega(f_{13}, f_{22}) = 0.2,\ \omega(f_{13}, f_{23}) = 0.5\}$. Then, five shot linkage measures are computed as follows:

- Since $M_1 = \{(f_{11}, f_{22}), (f_{12}, f_{23}), (f_{13}, f_{21})\}$, $SL = \frac{0.8+0.9+0.3}{3+3-3} = 0.67$.
- Since $M_2 = \{(f_{11}, f_{22}), (f_{13}, f_{23})\}$, $NCSL = \frac{0.8+0.4}{3+3-2} = 0.3$.

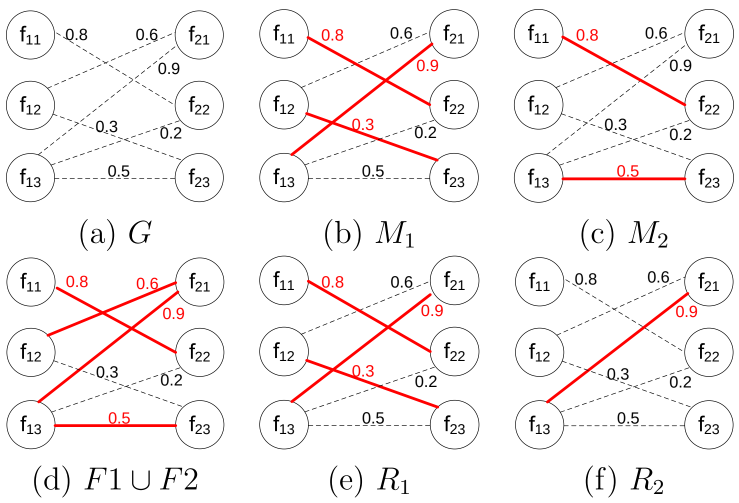


Figure 4.2. An illustration of Example 1.

- Since $F1$=$\{(f_{11}, f_{22}),\ (f_{12},\ f_{21}),\ (f_{13},\ f_{21})\}$, $F2 = \{(f_{13},\ f_{21}),\ (f_{12},\ f_{21}),\ (f_{13},\ f_{23})\}$, and $F1 \cup F2 = \{(f_{11},\ f_{22}),\ (f_{12},\ f_{21}),\ (f_{13},\ f_{21}),\ (f_{13},\ f_{23})\}$, gSL=$\frac{0.8+0.6+0.9+0.4}{3+3-4} = 1.35$.
- Since $R1 = \{(f_{13}, f_{21}), (f_{11}, f_{22}), (f_{12}, f_{23})\}$, $aSL = \frac{0.9+0.8+0.3}{3+3-3} = 0.67$.
- Since $R2 = \{(f_{13}, f_{21})\}$, $aNCSL = \frac{0.9}{3+3-1} = 0.18$.

Figure 4.2 illustrates the example. □

Lemma 6. *There is no bounding between $aSL_{\omega,\rho}(g_1, g_2)$ and $NCSL_{\omega,\rho}(g_1, g_2)$.* ■

Finally, combining Lemmas 1, 3, 4, and 6, we get the partial order among five shot linkage measures.

Theorem 1. *The following partial order exists among five shot linkage measures:*

$$aNCSL \leq NCSL, aSL \leq SL \leq gSL$$

That is, SL and $NCSL$ are bounded by $aNCSL$ (lower bound) and gSL (upper bound). ■

The advantage of these partial order is that both $gSL_{\omega,\rho}(g_1, g_2)$ and $aNCSL_{\omega,\rho}$ (g_1,g_2) can be computed faster than both $SL_{\omega,\rho}(g_1, g_2)$ and $NCSL_{\omega,\rho}$ (g_1, g_2), respectively. Therefore, quickly computing both $gSL_{\omega,\rho}(g_1, g_2)$ and $aNCSL_{\omega,\rho}$ (g_1,

g_2) can help us efficiently address our video linkage problem. For instance, imagine that we want to check if two shots s_1 and s_2 have a similarity above the pruning threshold of the shot level, δ, or not. Then, since $gSL_{\omega,\rho}$ is an upper bound of $SL_{\omega,\rho}$, if $gSL_{\omega,\rho}(g_1, g_2) < \delta$, then it must be the case that $SL_{\omega,\rho}(g_1, g_2) < \delta$. Hence, the edge between g_1 and g_2 is guaranteed to be pruned away. Reversely, using $aNCSL_{\omega,\rho}(g_1, g_2)$ as the lower bound of $SL_{\omega,\rho}$, if $aNCSL_{\omega,\rho}(g_1, g_2) \geq \delta$, then $SL_{\omega,\rho}(g_1, g_2) \geq \delta$ is true.

4.2.3.3 Shot linkage structures

Based on the findings in Section 4.2.3.1 and 4.2.3.2, we propose two different shot linkage techniques – *Standalone* and *Pipelined* shot linkages. Given two shots s_1 and s_2, we acquire $sim_s(s_1, s_2)$ as an edge value in the video level in Figure 4.1 by

- *Standalone* shot linkage algorithm that computes one of five shot linkage measures $\{SL, NCSL, gSL, aSL, aNCSL\}$.
- *Pipelined* shot linkage algorithm that exploits gSL as an upper bound. We first check $gSL_{\omega,\rho}(s_1, s_2) < \delta$ to determine if further computation is required or not. If $gSL \leq \delta$, the edge between s_1 and s_2 is pruned away foe the video level comparison. If $gSL > \delta$, then we need further computation of SL or $NCSL$ to obtain exact $sim_s(s_1, s_2)$.

4.2.4 Hierarchical Video Linkage (Video Level Similarity)

At the completion to obtain all $sim_s(s_i, s_j)$ between two videos, hierarchical Video Linkage exploits the graph structure used in shot linkage measures to the video level. Conceptually, the graph structure in the video level is identical to the graph structure in the shot level, by changing nodes from key frames to shots and edges from sim_f to sim_s. Temporal order between shots also exists like the temporal order between frames. Hence, all shot linkage measures in a shot level can be re-used for Video Linkage measure in a video level by replacing nodes and edges in a graph. However, there are two main differences between a shot and a video: (1) A video, specially copied video, can be compounded by multiple original videos. Thus a copied video may contain multiple video segments from multiple videos, while a compound shot is rare because shot boundaries are mostly detected between different segments. (2) With our observation of videos in YouTube, a copied video is usually shorter than an original video, since a copied video is usually small extraction of an original video. Thus, to measure $sim_v(v_a, v_b)$, even though we keep the same group based formulas used in a shot level, all shots are *not* selected as nodes in a video level.

4.2.4.1 Dynamic selection of shots as nodes

In a shot level comparison, all key frames are used as nodes in the graph structure. However, by assuming that we may use corresponding shots in a copied video to the shots in an original video, some shots should be discarded in the graph structure in a video level comparison. Intuitively, the sim_s between a copied shot and an original shot is relatively high. Thus, high-valued similarity between shots will be selected as edges in a video level. In turns, corresponding shots of high-valued edges (in a video level) will be selected as nodes to compute the similarity, $sim_v(v_a, v_b)$, using the proposed group-based matching techniques. It looks similar to the pruning step in a shot level. However, in a video level, when a shot is not selected as a node, then they will be completely removed from the graph structure, such that discarded nodes cannot contribute any equations in Video Linkage measures . There are four scenarios to limit to the number of high-valued edges and corresponding nodes to improve both accuracy and speed, as follows.

1. If a copied video is only the small part of an original video, the part of copied shots from an original video will only respond as high-valued edges. In other words, if all shots in an original video are included in a graph as nodes, the remnant shots in an original video will downgrade the similarity value significantly.

2. If a copied video consists of multi-segments from multiple original videos, when a copied video is compared with one of original videos, the shots in a segment from the original video will only respond as high-valued edges. The shots in other segments will downgrade the similarity, if all shots in a copied video are included in the graph.

3. If a user adds artifact shots such as transitions, the selection of high-valued edges will remove nodes representing artifact shots which cause setback of similarity.

4. Large number of edges and nodes increases running time exponentially, since exact similarity measures take more than or equal to $O(N^2)$. Therefore, certain number of N will be enough to detect copied videos, and it is empirically proven in Section 4.3.

Even though unnecessary shots are removed, the graph structure in a video level is same as that in a shot level. Therefore, by replacing key frames and sim_f to shots and sim_s, respectively, all 5 shot linkage measures can be reconsidered for all 5 Video Linkage measures. Obviously, small changes in the formulas are required as follows.

- The elements in g_1 and g_2 are replaced from frames to shots, such that $g_1 = \{s_{11}, ..., s_{1n_1}\}$ and $g_2 = \{s_{21}, ..., s_{2n_2}\}$
- A frame threshold, ρ, is changed by a shot threshold, δ, that decides if the edge between shots is pruned or not. (Note that corresponding nodes are still included in the graph.)
- A video threshold, θ, is applied to decide *if a video is copied or not.*
- An edbe value, ω, is represented by sim_s replacing sim_f.

When the changes of elements in a graph are complete, without loss of generality, we can have five Video Linkage measures, i.e., $VL_{\omega,\delta}(g_1, g_2)$, $NCVL_{\omega,\delta}(g_1, g_2)$, $gVL_{\omega,\delta}(g_1, g_2)$, $aVL_{\omega,\delta}(g_1, g_2)$, $aNCVL_{\omega,\delta}(g_1, g_2)$ from $SL_{\omega,\rho}(g_1, g_2)$, $NCSL_{\omega,\rho}(g_1, g_2)$, $gSL_{\omega,\rho}(g_1, g_2)$, $aSL_{\omega,\rho}(g_1, g_2)$, $aNCSL_{\omega,\rho}(g_1, g_2)$, respectively. The inequality among Video Linkage measures also inherits the inequality of shot linkage measures. Then,

$$aNCVL \leq NCVL, aVL \leq VL \leq gVL.$$

4.2.4.2 Hierarchical Video Linkage structures

Like shot linkage frameworks, we also propose two algorithmic Video Linkage structures - *standalone* and *pipelined* Video Linkage, in the video level comparison. However, unlike shot linkage frameworks, there are two main differences in Video Linkage frameworks. First, in order to obtain sim_v, one computation is only required between two videos because one video is one group of shots and many groups of frames. Second, the goal of video level comparison is to decide if a video is copied or not, while obtaining sim_s is the golf of the shot level comparison. Thus, exact similarity may not be computed if not necessary. Therefore, the video level threshold, θ, is used as the final decision criteria for detecting copied videos, while the shot level threshold, δ, is used for pruning edges in a video level. Thus, in *pipelined* structure, $aNCVL$ or aVL can also be used for faster filtering in the video level. For example, when $sim_v < aNCVL$, then we conclude $sim_v < NCVL$ by inequality property. Then, a video is not detected as a copy without exact similarity. Two algorithmic Video Linkage frameworks are as follows;

- *Standalone* Video Linkage algorithm inherits shot linkage measures such that sim_v is computed by one of five Video Linkage measures $\{VL$, $NCVL$, gVL, aVL, $aNCVL\}$.

- *Pipelined* Video Linkage algorithm that exploits the upper bound, gVL, as well as the lower bound, $aNCVL$. Unlikely the goal of a shot level aiming to compute sim_s between shots, the video level should decide if two videos are copied or not. That is, to determine if $sim_v(v_a, v_b) \geq \theta$, we first check the fast (but approximate) greedy measure $gVL_{\omega,\rho}(v_a, v_b) < \theta$. If so, we conclude $sim_v(v_a, v_b) \ngeq \theta$. Else, we next check another fast (but approximate) measure $aNCVL_{\omega,\rho}(v_a, v_b) \geq \theta$. If so, we conclude $sim_v(v_a, v_b) \geq \theta$. Else, finally, we resort back to standalone Video Linkage and check $sim_v(v_1, v_2) \geq \theta$ using one of slow but exact video linkage measures $\{VL$, $NCVL\}$ for $sim_v()$ function.

The details of two Video Linkage algorithms are illustrated in Algorithms 1 and 2, respectively. The *Hierarchical* Video Linkage *Framework* is accomplished by selecting one of shot linkage measures and one of Video Linkage measures. For example, one can select $gVL|VL$ for Video Linkage to decide if a video is copied or not, and

Algorithm 1: Standalone Video Linkage.

```
Input  : A query video $v_q$ and a source video set $V_s$
Output: A copied video set $V_c$ ($\subseteq V_s$)
linkage ← {$VL$, $NCVL$, $gVL$, $aVL$, $aNCVL$};
$V_c \leftarrow \emptyset$;
foreach $v_s$ ($\in V_s$) do
    if linkage$(v_q, v_s) \geq \theta$ then $V_c \leftarrow V_c \cup v_s$;
return $V_c$;
```

Algorithm 2: Pipelined Video Linkage.

```
Input  : A query video $v_q$ and a source video set $V_s$
Output: A copied video set $V_c$ ($\subseteq V_s$)
slow-linkage ← {$VL$, $NCVL$};
$V_c \leftarrow \emptyset$;
foreach $v_s$ ($\in V_s$) do
    if $gVL(v_q, v_s) < \theta$ then continue;
    if $aNCVL(v_q, v_s) \geq \theta$ then $V_c \leftarrow V_c \cup v_s$; continue;
    if slow-linkage$(v_q, v_s) \geq \theta$ then $V_c \leftarrow V_c \cup v_s$;
return $V_c$;
```

select $gSL|NCSL$ to measure the similarity between shots in a shot level. For sim_f in a frame level, the weighted combination of cosine distances are chosen after features of key frames are extracted.

4.3 Experimental Validation

To validate our proposals, we have performed extensive experiments with a public benchmark data set of MUSCLE-VCD-2007[1] and real videos downloaded from YouTube.

4.3.1 Set-Up

All proposed algorithms are implemented in Java and JMF 2.1e, and executed in a desktop with Intel Celeron 3.20GHz, 2GB RAM. To extract the features from **.flv* video format used in YouTube videos, Java wrapper for ffmpeg, i.e., fobs4jmf, is applied for our implementation.

Data Sets. First, MUSCLE-VCD-2007 data set contains 101 source videos in V_s and two different query sets. In the first query set, $ST1$, containing 15 copied videos, each copied video is distorted from a corresponding original video by

[1]http://www-roc.inria.fr/imedia/civr-bench/data.html

YouTube Data Set											
Category	AV	CO	ET	MU	HS	PB	NP	PA	TE	GA	Total
$\|V_s\|$	261	284	239	230	247	440	289	394	332	216	2,932
$\|V_q\|$	10	10	10	10	9	10	10	9	10	10	98
$\|V_c\|$	90	90	90	90	81	90	90	81	90	90	882

MUSCLE-VCD-2007 Data Set
$\|V_s\| = 101$, $\|ST1\| = 15$, $\|ST2\| = 3$, $\|V_q\| = 18$, $\|V_c\| = 31$

Table 4.3. Description of data set.

multiple editing methods (e.g., color change, blur, re-encoding, crop, camcording, subtitles, analogic noise, change in YUV, camcording with an angle, horizontal flip, zoom, and resize) applied to a whole video. Thus, the length of a copied video is the same as that of an original video. The second query set, $ST2$, contains 3 copied videos. Each copied video is composed of several video segments from different original videos. In turn, each video segment is distorted by multiple aforementioned editing methods. Note that the length of a video segment is relatively short, i.e., ranging from 23 seconds to 2 minutes, but the length of an original video varies from 5 minutes to 100 minutes.

Second, for YouTube data set, a source video data set, V_s, is made by downloading 2,050 video clips from YouTube throughout all categories. Among the 15 categories in YouTube, 10 categories are selected: Autos & Vehicles (AV), Comedy (CO), Entertainment (ET), Howto & Style (HS), Music (MU), News & Politics (NP), People & Blogs (PB), Pets & Animals (PA), Travel & Events (TE), and Gaming (GA). In order to synthesize copied videos, we select 98 videos as original videos (V_q) from 10 categories. For each original video, 10 copied videos are generated by 'cut in a video (CV)' (3), 'cut and paste in a video (CP)' (2), 'cut and paste from different videos (CD)' (1), 'change size/resolution (CR)'(2), 'adding title and credit (TC)'(1), and 'adding logo/title (LT)' (1), where(#) indicates the number of copies. Therefore, 882 copied videos (V_c) are created in total. All copied videos are added to a source video set in the corresponding category. As a result, we have 2,932 source videos for the data set.

Table 4.3 summarizes the statistics of two data sets.

Schemes and Evaluation Metrics. Two standpoints are considered as performance metrics: (1) **accuracy** in terms of precision($\frac{N_{TruePositive}}{N_{AllPositive}}$), recall($\frac{N_{TruePositive}}{N_{AllTrue}}$), and F-measure ($\frac{2\times(Precision\times Recall)}{Precision+Recall}$), and (2) **performance** in terms of *wall-clock running time.*

In the following, we first show that our proposal outperforms competing methods using MUSCLE-VCD-2007 data set. Then, we compare among our proposal in detail using YouTube data set.

4.3.2 Video Linkage on MUSCLE-VCD-2007 data set

4.3.2.1 Parameters and decision rule

Two query sets, $ST1$ and $ST2$ of MUSCLE-VCD-2007, have different characteristic. Thus, we have different parameter setup for them.

In $ST1$, a copied video is edited by multiple distortion methods from one original video with whole length. It means a copied video contains the similar number of shots of an original video that usually have long length. Thus, the number of high-valued edges in a video level is set to 30 to exploit adequate number of shots as nodes, i.e., $N = 30$.

In $ST2$ query set having 3 copied videos, more complex and compound editing methods (e.g., *caption, color change, phase change, crop, sharpness, letter box, contrast change, logo, shift, zoom, vertical deformation, blur, horizontal flip, gamma change, brightness change*, and *vertical deformation*) are applied to all video segments in a copied video. In addition, one copied video is composed of several short video segments (from 24 to 123 seconds) from different original videos (from 316 to 6315 seconds). It means that the limited number of shots exists in copied videos. Therefore, in order to improve the running time, as long as one shot in a copied video is detected as a copied shot from an original video, then the video is considered as a copied video. Thus, in the video level comparison, only the highest valued edge is used in the decision. Note that the number of high-valued edges, N, can vary proportionally to the length of a video, assuming that longer video contains more shots.

In addition, the video level threshold, θ, and the shot level threshold, δ, are set to 0.93 and 0.9, respectively, to achieve the best performance overall. The frame level threshold, ρ, is set to 0.5 to use more features from the frame. In order to obtain proper output described in the CVIR 2007 competition, we apply the combination of top-1 and threshold (0.93) based decision rule for $ST1$, and precision/recall graph based on varying thresholds (0.75$\sim$0.98) for $ST2$.

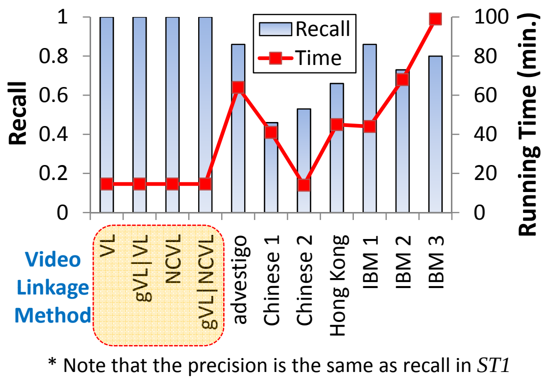
4.3.2.2 Accuracy and performance on $ST1$

Using the same evaluation metric in CIVR 2007 competition with MUSCLE video data set, the proposed Video Linkage methods acquire 15 correct answers from 15 query videos, i.e., both precision and recall are 1. It means that our proposed algorithms are robust to any types of distortion presented in copied videos. Note that the best recall in the CIVR 2007 competition is only 0.86. Figure 4.3(a) depicts the comparison between Video Linkage and methods presented in CIVR 2007 competition. The bar graph shows recall values referencing the left Y-axis, while the line graph shows the actual running time (minutes) referencing the right Y-axis. Note that the precision value is the same as recall value in $ST1$, since only one answer can be returned for each query video. Therefore, the higher bar means higher accuracy, while the lower point in a line graph means faster processing time. The processing time should include the pre-processing time (feature extraction and descriptor generation) for query videos only, as stated in the instruction of the competition. As a result, the total running time of $gVL|NCVL$ is about 14 minutes 35 seconds including the pre-processing, data loading time, and comparison time, which outperforms the existing methods in the competition by 3.67 times on average. Note that it took 44 minutes with 0.86 recall in the competition[2].

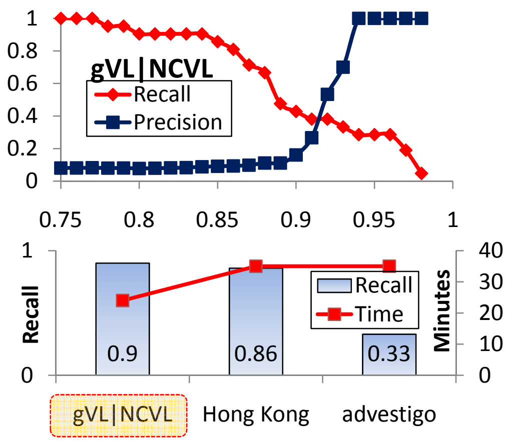
4.3.2.3 Accuracy and performance on $ST2$

In order to verify the effectiveness of the proposed CVD algorithms, $gVL|NCVL$ is selected. Note that the other algorithms such as VL, $NCVL$, and $gVL|VL$ provide similar behaviors, too. The characteristic comparison among our proposed algorithms will be fully described with YouTube videos in 4.3.3. Figure 4.3(b) shows the trace of precision and recall along different threshold levels, as well as the comparison with the existing methods in the CIVR 2007 competition with respect

[2]http://www-rocq.inria.fr/imedia/civr-bench/Results.html



(a) Recall on $ST1$



(b) Precision & Recall on $ST2$

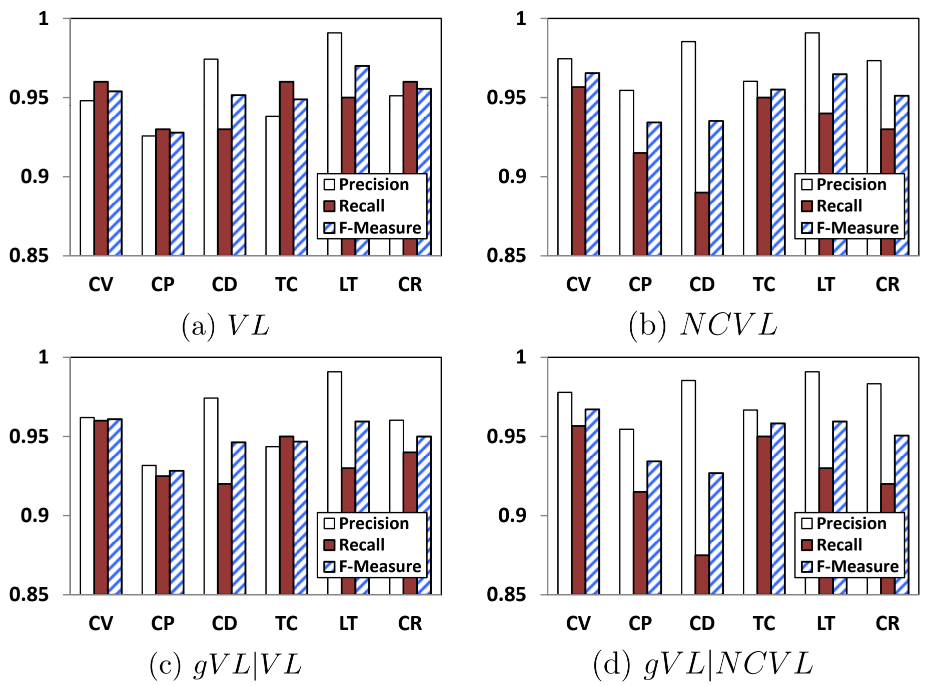
Figure 4.3. Accuracy and performance of selected Video Linkage algorithms on MUSCLE-VCD-2007 video sets.

to *recall* and *running time*. Out proposed algorithm achieves the recall of 0.9 by setting the threshold 0.84, while the the best recall among competing methods is only 0.86. In addition, the processing time of our proposed algorithm for ST2 is about 24 minutes, while all competing methods take more than 30 minutes. As shown in Figure 4.3(b), the precision is very small to achieve the proper recall, because many videos in the benchmark set contain similar features to the short period of video segments in one copied videos. This affects the sensitiveness of the threshold level. As a result, most original videos are detected as a copy at threshold value of 0.86, and the recall value is significantly increased.

4.3.3 Video Linkage on YouTube data set

4.3.3.1 Parameters

With respect to the characteristic of a copied video in YouTube data set, the copied video contains only the part of an original video, or compound video segments from multiple videos. For example, while the length of an original video is about 20 minutes, the length of a copied video is only about 2 minutes. Here, we also exploit the highest-valued edge in a video level graph, assuming only the small number of shots are copied from an original video. The other parameters such as thresholds are same as those used in MUSCLE-VCE 2007 data set. We only use



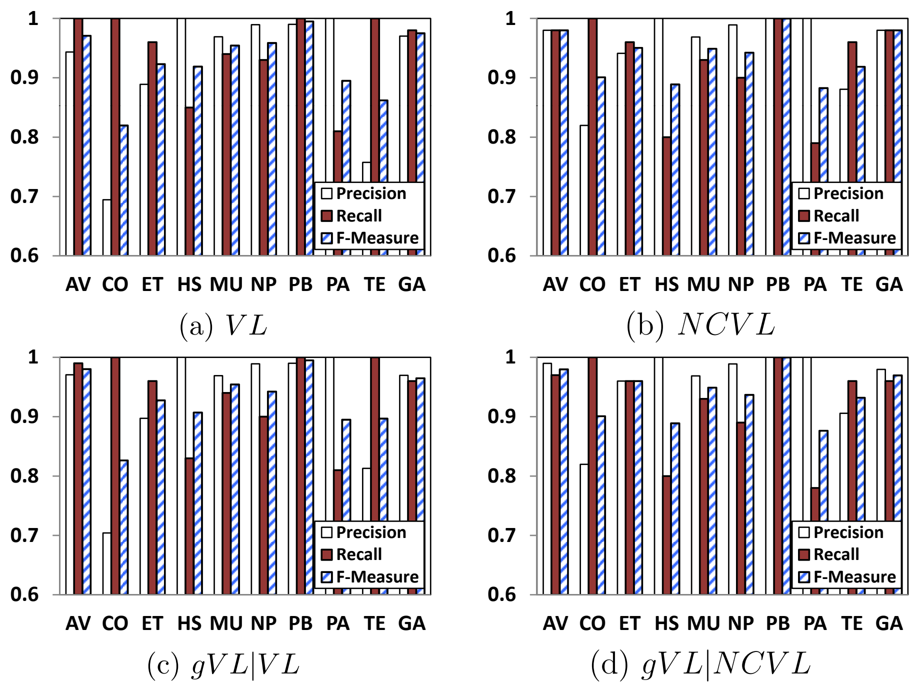
(a) VL (b) $NCVL$ (c) $gVL|VL$ (d) $gVL|NCVL$

Figure 4.4. Performance of each scheme over 6 editing methods.

threshold based decision rule for YouTube videos.

4.3.3.2 Robustness on various editing

As stated earlier, a copied video v_c in sites such as YouTube and Yahoo Video is typically created by altering the original video v_q using several basic editing methods. Therefore, a good measure for CVD should be robust against such editing methods applied to generate copied video in YouTube data set. Figure 4.4 shows the performance of each scheme for 6 editing methods mentioned in section 4.3.1. Figure 4.4(a)-(d) are the results of VL, $NCVL$, $gVL|VL$, and $gVL|NCVL$, respectively. As shown in the figure, our proposed Video Linkage measures are robust over all editing methods, i.e., 0.93 in recall, 0.94 in precision and 0.935 in F-measure on average. Specifically, the high precision and recalls of Video Linkage measures indicate that our proposals go a good job in detecting copied videos accurately from data sets.



(a) VL (b) $NCVL$

(c) $gVL|VL$ (d) $gVL|NCVL$

Figure 4.5. Performance of each scheme over 10 genres.

4.3.3.3 Robustness on various genres

Since the characteristics of videos are various dependent on specific genres such as music videos, lectures, TV shows, and news, it is crucial for CVD to work with various genres of videos. In order to verify the robustness of Video Linkage measures against video contents, we test 4 measures on 10 genres of data sets mentioned in Table 4.3. Each genres has its unique characteristics of video contents. For example, a video in People & Blogs (PB) genre usually has very static images with relatively long shots, while that of Entertainment (ET) contains a lot of fast movements. Figure 4.5 shows the performance results of each scheme for 10 genres, and the overall average performance on all genres. Figure 4.5(a)-(d) are the results of VL, $NCVL$, $gVL|VL$, and $gVL|NCVL$, respectively. Video Linkage measures provide very consistent results over various genres in terms of recalls. However, the precisions and F-measure are relatively lower than recalls in some genres, such as Comedy (CO), Entertainment (ET), and Travel & Events (TE). This is caused by a lot of motion changes in a short time period. In order to increase the performance in terms of precision and F-measure in the specific genre, we can adjust the weight value of each feature (w_λ) to the contents so that the feature from motion vectors can contribute more on similarity measure. To keep the purpose of general solution of CVD, we use the same w_λ through this experiment.

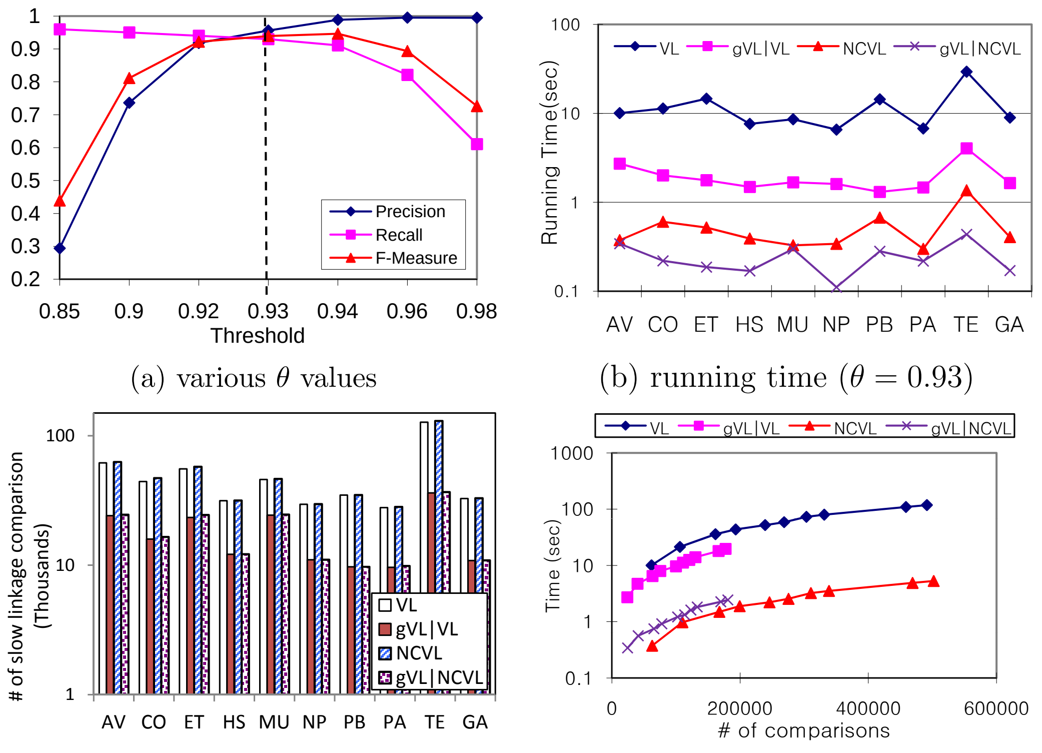
4.3.3.4 Accuracy on various thresholds

we use the threshold version of the CVD problem for YouTube data set, since we pretend NOT to know the number of copies in a source video set, V_s. Therefore, we need an appropriate threshold value θ for all Video Linkage frameworks. However, predicting an optimal θ value is a challenging problem itself. To decide the threshold, we investigate the average recalls of all schemes over various values of θ, i.e., 0.85 to 0.98. Figure 4.6(a) shows the results of average recalls for $NCVL$. As observed in the figure, $NCVL$ has hight recall values with $\theta < 0.93$. Therefore, we used 0.93 as our default threshold value.

4.3.3.5 Performance of selected schemes

Next, we evaluate the performance of our Video Linkage methods with respect to the running time and # of comparison of slow shot linkage measures. Figure 4.6(b) first shows the running time of four methods, VL, $gVL|VL$, $NCVL$, and $gVL|NCVL$, over video sets in all genres. As expected, VL is the slowest method, regardless of genres, due to its high computational cost. Furthermore, the application of gVL as the "filter" in $gVL|VL$ speeds up the performance significantly (on average 5 times faster than the processing time without filter). In fact, $NCVL$ is faster than both VL and $gVL|VL$ because of different time complexities shown in Table 4.2, and $gVL|NCVL$ is the fastest among all 4 measures.

Figure 4.6(c) shows the performance of Video Linkage measures with respect to the number of slow shot linkage computations. The number of computations of slow shot linkage measures during the copied video detection is the dominant component for the performance of overall CVD procedure. Thus, we count the number of computation for slow shot linkage measures to evaluate the performance. We can observe that $gVL|VL$ requires the smallest number of computations during the CVD. Since it filters out a lot of non-similar videos, one can reduce the computation time. Note that the running time of $gVL|VL$ is about 5 times faster than VL, since most long shots has been filtered out by gVL. Figure 4.6(d) shows the



(a) various θ values (b) running time ($\theta = 0.93$)

(c) # of slow comparison (d) scalability

Figure 4.6. Various performance results.

scalability of Video Linkage measures over the number of shot comparisons. As expected, the total computational time of all Video Linkage measures are increasing monotonic over the number of shot comparisons. Specifically, $gVL|NCVL$ is the best algorithm in terms of scalability. That indicates the proposed scheme can be applied into a real life system to detect copied videos easily.

4.4 Summary

In this chapter, we have presented the novel idea of the hierarchical structure of group based CVD solutions, Video Linkage. In order to implement hierarchical group based matching idea, we introduce a method to transform a video to a group of shots, and a shot to a group of key frames in three level structures. In the frame level, once the selected features are extracted from key frames, the similarity between frames is obtained by weighted vector distance. In the shot level, the similarity between shots are measured by proposed five shot linkage measures by considering a shot as a group of key frames. Like shot linkage measures, once a video is captured as a group of selected shots, we propose five Video Linkage measures. Using a benchmark data set of MUSCLE-VCD-2007 and videos downloaded from YouTube, our proposed hierarchical structure of Video Linkage solutions are validated with respect to robustness and performance. In addition, a sparse graph by pruning low-valued edges and pipe-lined Video Linkage frameworks enhance the performance further.

Chapter 5

Parallel Linkage

In this chapter, we focus on the parallelization of sequential data linkage algorithms to pursue the high efficiency and scalability. The data cleaning processing in one dirty data set or merging processing of two different data sets requires quadratic number of comparisons. For example, in order to clean one dirty set, D, where $|D|$=L, we need ${}_{L}C_{2} = \frac{L(L-1)}{2}$ comparisons. For another example, when we compare two different data set, A and B with $|A| = m$ and $|B| = n$, $m \times n$ comparisons are required to find all identical records, even without merging sets. Toward this problem, instead of suggesting an efficient *match* algorithm, we propose the parallel frameworks of data linkage which can adapt any record *match* function. We also investigate another important aspect of record linkage problem – how to merge records that are matched.

Our contributions in this work are as follows: (1) We formally introduce the linkage problem with separate *match* and *merge* steps, and exploit them to have better sequential linkage framework for three scenarios; (2) We extend sequential linkage algorithms to parallel ones under three scenarios such that redundant computation and overhead among multiple processors are minimized; (3) Our proposals are evaluated using citation data sets with a variety of characteristics. Our parallel algorithms achieve 6.55–7.49 times faster in *speedup* compared to sequential ones with 8 processors, and 11.15–18.56% improvement in *efficiency* compared to an existing parallel solution, P-Swoosh.

5.1 Problem Definition

Problem Overview. To take these points into consideration, the linkage problem that we consider in this proposal can be defined as follows:

> **Linkage Problem**: Given two collections of compatible records, A=$\{a_1$, ..., $a_m\}$ and B=$\{b_1$, ..., $b_n\}$, do: (1) identify and merge all matching (i.e., $\approx$) record pairs (a_i,a_j), (b_i,b_j), or (a_i,b_j), and (2) create a merged collection C=$\{c_1$, ..., $c_k\}$ of A and B such that $\forall c_i, c_j \in C$, $c_i \not\approx c_j$.

Note that neither A nor B itself is assumed to be *clean* (to be defined in Definition 10) – i.e., there may be two matching records in it.

Definition 9 (Record Matching) *When two records, r and s, are deemed to refer to the same real-world entity, both are said* matching, *and written as $r \approx s$ (otherwise $r \not\approx s$).* □

Note that the linkage algorithm in this chapter is on the parallelization of the linkage problem, but not on measuring the similarity between two records - i.e. any distance measures for record matching can be used in this framework. In practice, however, the matching of two records can be often determined by distance or similarity functions. For instance, one may use the cosine angle of token sets of r and s (i.e., cosine similarity) to determine the match of r and s. Or, one may use the ratio of intersected vs. unioned q-gram tokens of two records (i.e., jaccard similarity) for the same purpose.

When two records, r and s, are matching (i.e., $r \approx s$), four relationships, as illustrated in Figure 5.1, can occur: (1) $r \sqsupseteq s$: all information of s appears in r, (2) $r \sqsubseteq s$: all information of r appears in s, (3) $r \equiv s$: information of r and s is identical (i.e., $r \sqsupseteq s \wedge r \sqsubseteq s$), and (4) $r \oplus s$: neither (1) nor (2), but the overlap of information of r and s is beyond a threshold θ. Note that to be a flexible framework we do *not* tie the definitions of the four relationships to a particular notion of *containment* or *overlap*. Instead, we assume that the containment or overlap of two records can be further specified by users or applications. Let us assume the existence of two such functions: (1) **contain(r,s)** returns True if r contains s, and False otherwise, and (2) **match(r,s)** returns True (i.e., one of the four inter-record relationships) or False for non-matching. We assume that match(r,s) is

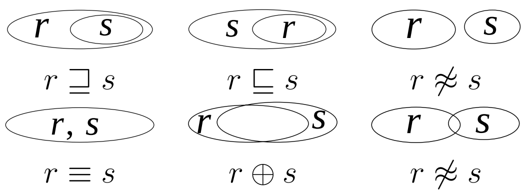


Figure 5.1. Inter-record relationships.

implemented using contain(r,s) function internally (e.g., if both contain(r,s) and contain(s,r) return True, then match(r,s) returns $r \equiv s$).

Example 2. For a table with five columns, consider the following records: r_1: ("a",–,–,–,–), r_2:(–,"b", –, –, –), r_3:("a","b","c", –, –), r_4:(–,"b","c","d",–), and r_5:(–, –, –,"d","e"). Further, let us assume two function: (1) the containment of two records is determined by the containment of token sets of two records, and (2) the overlap of two records is measured by the average jaccard similarity of two corresponding columns of two records with $\theta = 0.3$. Then, $r_1 \sqsubseteq r_3$ holds since {a} ⊆ {a, b, c}, $r_2 \sqsubseteq r_3$ holds since {b} ⊆ {a, b, c}, and $r_2 \sqsubseteq r_4$ since {b} ⊆ {b, c, d}. In addition, jaccard(r_3,r_4)= $\frac{0+1+1+0+0}{5} = 0.4 > \theta$ and jaccard(r_4,r_5)=$\frac{0+0+0+1+0}{5} = 0.2 < \theta$. Therefore, both $r_3 \oplus r_4$ and $r_4 \not\approx r_5$ hold. □

When two records r and s are matching (i.e., $r \sqsubseteq s$, $r \sqsupseteq s$, $r \equiv s$, or $r \oplus s$), one can merge them to get a record with more (or better) information. Again, how exactly the merge is implemented is not the concern of this chapter. We simply refer to a function that merges r and s to get a new record w as **merge(r,s)**.

Example 3. For instance, like [68], if one uses the set union operator, ∪, as the merge function for Example 2, then merge(r_3,r_4) would generate a new record r_{34}: ("a","b","c", "d", –), while merge(r_1, r_3) would generate r_{13}=r_3:("a", "b", "c", –, –) since $r_1 \sqsubseteq r_3$. □

Definition 10 (Clean vs. Dirty) *When a collection A has no matching records in it, it is called* clean, *and* dirty *otherwise. That is, (1) A is clean iff* $\forall r, s \in A$, $r \not\approx s$, *and (2) A is dirty iff* $\exists r, s \in A$, $r \approx s$. □

Table 5.1 summarizes the notations.

Symbol	Meaning
A, B	two input collections
m and n	size of A and B, i.e., $m = \|A\|$, $n = \|B\|$
r or a_i	a record in A
s or b_j	a record in B
c_{ij} or $c_{i,j}$	a merged record from a_i and b_j
θ	threshold for $r \oplus s$
contain(a_i,b_j)	returns True if a_i contains b_j, or False
match(a_i,b_j)	returns $\sqsupseteq$, $\sqsubseteq$, $\equiv$, $\oplus$, or $\not\approx$
merge(a_i,b_j)	returns c_{ij}

Table 5.1. Summary of notations in Parallel Linkage.

5.2 Solution Overview

In order to address the problem, we investigate three scenarios – when both collections are clean, when only one is clean, and when both are dirty. Furthermore, we show that the intricate interplay between matching and merging steps can exploit the characteristics of each scenario to achieve good parallelization. The intuition of our algorithms is that if: (1) a_i is deemed to be a duplicate of b_j, and (2) an input collection B is a set, not a bag (i.e., clean), then one does not need to check if a_i is a duplicate of b_{j+1}, ..., b_n in the algorithm. Depending on the relationship between a_i and b_j (i.e., a_i contains b_j, b_j contains a_i, a_i is identical to b_j, or a_i is overlapping with b_j), this intuition can be exploited differently.

The basic flow of our proposed solutions is as follows:

- (Sections 5.3) In order to minimize the number of comparisons, i.e. to optimize the sequential linkage structure, we suggest three different sequential frames depending on the types of input data sets as shown in Table 5.2: (1) *clean vs. clean*: This scenario is relevant when two already-clean data sources are integrated, (2) *dirty vs. clean*: Consider a search engine that has a clean data set A, but its crawler fetches new dirty data set B every day. In this case, not only B may have matching records in it, there can be new matching pairs between A and B, (3) *dirty vs. dirty*: When one has two dirty sets, one can clean each dirty set independently and apply clean vs. clean scenario. However, as we will present, one may be able to improve the linkage by matching two dirty sets directly. The scenario of cleaning single dirty set A (i.e., self cleaning) will be shown to be covered by dirty-clean or

$A \setminus B$	Clean	Dirty
Clean	Sections 5.3.1 and 5.4.1	Sections 5.3.2 and 5.4.2
Dirty	Sections 5.3.2 and 5.4.2	Sections 5.3.3 and 5.4.3

Table 5.2. Taxonomy.

dirty-dirty cases easily. Finally, we propose 6 different sequential algorithms. However, the performances on different algorithms depend on the dirtyness of input data sets.

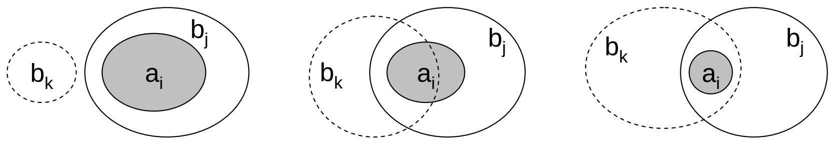
- (Section 5.4) As the presentation of types of input data sets, three scenarios are also applied to the Parallel Linkage structure. Parallel Linkage is to perform *match* and *merge* processes concurrently in multiple processors. To attain the most efficient system, the tasks are needed to be partitioned and distributed to the processors evenly. Furthermore, the messages between processors should be minimized. In reality, because of the nature of merging steps, the actual size of tasks cannot be pre-measured. However, the characteristic of data sets will predict the size of tasks (the number of comparisons). Therefore, in our solution, the data partition model is used to predict the number of comparisons, so that semi-even distribution of tasks to processors can be achieved.

5.3 Sequential Linkage

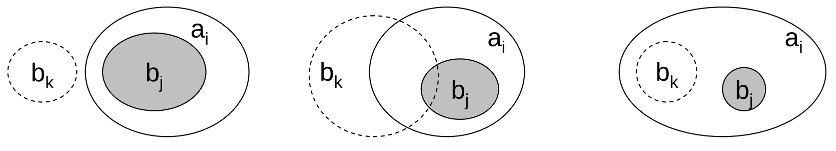
5.3.1 Clean vs. Clean

Recall that unlike database join, in the linkage problem, if two records a_i and b_j match, then a merged record c_{ij} (= merge(a_i,b_j)) is created and re-feeded into A and B. However, depending on the type of matching, one can do further saving. Suppose we use the naive algorithm mentioned in section 1.1.3 for its simplicity. Consider four records: a_i, a_l in A ($i < l \leq m$), b_j, b_k in B ($j < k \leq n$), and two sets E (to hold an instance of two identical records) and C (to hold merged record c_{ij} of a_i and b_j). Then, if:

- $a_i \not\approx b_j$: Proceed to the next match(a_i,b_{j+1}).



(a) $a_i \sqsubseteq b_j$: (left) $a_i \not\approx b_k$ (middle) $a_i \oplus b_k$ (right) $a_i \sqsubseteq b_k$



(b) $a_i \sqsupseteq b_j$: (left) $a_i \not\approx b_k$ (middle) $a_i \oplus b_k$ (right) $a_i \sqsupseteq b_k$

Figure 5.2. Six possible relationships for a_i, b_j, and b_k when $b_j \not\approx b_k$ (i.e., B is clean).

- $a_i \equiv b_j$: First, add a_i to E. Then, remove a_i from A and b_j from B. Since B is a clean set, by definition, there cannot be any matching records to b_j in B. Since a_i is identical to b_j, in addition, there cannot be any matching records to a_i in B, either. Therefore, we do not need to compute: match(a_i,b_k). Symmetrically, since A is also a clean set and a_i is identical to b_j, there cannot be any matching records to b_j in A, either, and thus we do not need to compute match(a_l,b_j). At the end, therefore, $m - i + n - j$ times of computation of *match* function is saved. Finally, proceed to the next match(a_{i+1},b_1).

- $a_i \sqsubseteq b_j$: First, remove a_i from A. Between a_i and b_k, two relationships, $b_k \sqsubseteq a_i$ and $a_i \equiv b_k$, cannot occur since B is a clean set (e.g., if $b_k \sqsubseteq a_i \wedge a_i \sqsubseteq b_j$, then $b_k \sqsubseteq b_j$ by transitivity, but since B is a clean set, $b_k \not\sqsubseteq b_j$, leading to a contradiction). However, the other three relationships can occur, as illustrated in Figure 5.2(a). Note that if the only possible relationship between a_i and b_k was $a_i \not\approx b_k$, then we could have skipped the computation of match(a_i,b_k). However, since there are three possibilities, we may not entirely skip match(a_i,b_k). However, note that in Figure 5.2(a), (1) if $a_i \sqsubseteq b_k$, then even if we do compute match(a_i,b_k), it does not generate any new record since merge(a_i,b_k)=b_k. Therefore, this case can be ignored, and (2) $a_i \oplus b_k$ case is rare in practice. Since $b_j \not\approx b_k$, if $a_i \sqsubseteq b_j$, then most likely $a_i \not\approx b_k$ holds, although an extreme case like Figure 5.2(a) (middle) can happen. Therefore, we can skip the entire computation of match(a_i,b_k) with the risk

Algorithm 3: s-CC-single.

```
    Input  : Two non-empty clean lists $A$ and $B$
    Output: Intermediate lists $A'$, $B'$ and a clean list $C$
    /* $E$ is a temporary list to contain equality records */
    $i \leftarrow j \leftarrow 1$, $A' \leftarrow B' \leftarrow C \leftarrow E \leftarrow \emptyset$;
3.1 while $i \leq |A|$ and $j \leq |B|$ do
        switch match($a_i$,$b_j$) do
            case $a_i \equiv b_j$
                add $a_i$ to $E$; remove $a_i$ from $A$ and $b_j$ from $B$;
                $i \leftarrow i+1$, $j \leftarrow 1$;
            case $a_i \sqsubseteq b_j$
                remove $a_i$ from $A$; $i \leftarrow i+1$, $j \leftarrow 1$;
            case $a_i \sqsupseteq b_j$  remove $b_j$ from $B$; $j \leftarrow j+1$;
            case $a_i \oplus b_j$
                remove $a_i$ from $A$ and $b_j$ from $B$;
3.2             $C \leftarrow$ s-merge($c_{ij}$,$C$); $i \leftarrow i+1$, $j \leftarrow 1$;
            case $a_i \not\approx b_j$  $j \leftarrow j+1$;
        if $j > |B|$ then $i \leftarrow i+1$, $j \leftarrow 1$;
    $A' \leftarrow A$, $B' \leftarrow B \cup E$; return ($A'$, $B'$, $C$);
```

Algorithm 4: s-CC.

```
    Input  : Two non-empty clean lists $A$ and $B$
    Output: A single merged clean list $C$
    $A' \leftarrow B' \leftarrow C \leftarrow E \leftarrow \emptyset$;
4.1 while $A \neq \emptyset$ do
        $(A', B', C) \leftarrow$ s-CC-single$(A, B)$; $A \leftarrow C$; $B \leftarrow A' \cup B'$;
    $C \leftarrow B$; return $C$;
```

of rare false negatives. In the experimentation, we empirically show that the risk is quite low[1]. Finally, proceed to the next match(a_{i+1},b_1).

- $a_i \sqsupseteq b_j$: First, remove b_j from B. between a_i and b_k, two relationships, $a_i \sqsubseteq b_k$ and $a_i \equiv b_k$, cannot occur since B is a clean set, but the other three relationships can occur, as illustrated in Figure 5.2(b). Since there are three possibilities between a_i and b_k, we may not skip the computation of match(a_i,b_k). Finally, proceed to the next match(a_i,b_{j+1}).

- $a_i \oplus b_j$: First, remove both a_i from A and b_j from B. Then, add c_{ij} (= merge(a_i,b_j)) to C. Finally, proceed to the next match(a_i,b_{j+1}).

The iterative sequential linkage algorithm for clean-clean case, referred to as **s-CC**, is shown in Algorithm 4 that terminates when no more *merge* occurs (line 4.1). The main functionality of **s-CC** using the five cases of inter-record relationships

[1]By setting the threshold for overlap, θ, as substantially high or low, we can decrease the risk of false negatives even further.

Algorithm 5: **s-DC-single**.

```
    Input  : Non-empty dirty list $A$ and clean list $B$
    Output: An intermediate list $B'$ and merged dirty list $C$
    $i \leftarrow j \leftarrow 1$, $C \leftarrow \emptyset$;
    while $i \leq |A|$ and $j \leq |B|$ do
        switch match($a_i$,$b_j$) do
            case $a_i \equiv b_j$ or $a_i \sqsubseteq b_j$
5.1             remove $a_i$ from $A$; $i \leftarrow i+1$, $j \leftarrow 1$;
5.2         case $a_i \sqsupseteq b_j$  remove $b_j$ from $B$; $j \leftarrow j+1$;
            case $a_i \oplus b_j$
                remove $a_i$ from $A$ and $b_j$ from $B$;
5.3             add $c_{ij}$ to $C$; $i \leftarrow i+1$, $j \leftarrow 1$;
            case $a_i \not\approx b_j$ $j \leftarrow j+1$;
        if $j > |B|$ or $B = \emptyset$ then
            add $a_i$ to $B$; remove $a_i$ from $A$;
            $i \leftarrow i+1$, $j \leftarrow 1$;
    $B' \leftarrow B$; return ($B'$,$C$);
```

is captured in **s-CC-single** of Algorithm 3. At line 3.1 of **s-CC-single**, both $m = |A|$ and $n = |B|$ continue to shrink as records in A or B are removed. The function s-merge(c_{ij},C) merges a record c_{ij} into a clean list C, ensuring that resulting list C be still clean by comparing c_{ij} to all records in C.

5.3.2 Dirty vs. Clean

The detailed procedure for the dirty-clean case, **s-DC**, is shown in Algorithm 6 that uses Algorithm 5 as a sub-step. When only one collection, A, is *dirty*, one can use the other clean collection, B, as the final merged clean set C to minimize space cost. Therefore, when either $\sqsubseteq$ or $\sqsupseteq$ relationship occurs, the record can be simply removed from one collection (lines 5.1 and 5.2 of **s-DC-single**). Similarly, when $\oplus$ relationship occurs (line 5.3), both original records, a_i and b_j, are removed and the new matched record c_{ij} is added to the dirty collection C. After the iteration, when a_i is not matched to any records b_j from B (line 5.3), it becomes safe to move a_i to the clean collection, B. From the next iteration, this newly-moved record a_i will be compared to the rest of records A. This step is necessary since A was dirty. When no more merge occurs in the line 5.3 of Algorithm 5 (i.e. A is empty in the line 6.1 of Algorithm 6) the algorithm terminates.

By using **s-DC**, note that one can clean a single dirty collection A. That is, by moving the first record from A to B, one can turn the problem into the sequential linkage of dirty-clean case. As a syntactic sugar, let us call this algorithm as

Algorithm 6: **s-DC**.

```
    Input  : Non-empty dirty list $A$ and clean list $B$
    Output: One merged clean list $C$
6.1 while $A \neq \emptyset$ do
        $(B', C) \leftarrow$ s-DC-single(A,B); $A \leftarrow C$; $B \leftarrow B'$;
    $C \leftarrow B$; return $C$;
```

Algorithm 7: **s-self**.

```
Input  : A non-empty dirty list $A$
Output: A non-empty clean list $C$
$B \leftarrow C \leftarrow \emptyset$; move $a_1$ from $A$ to $B$; $C \leftarrow$ s-DC($A$,$B$); return $C$;
```

s-self, shown in Algorithm 7. Then, another way to implement the sequential linkage for dirty-clean case to use **s-self** and **s-CC** – i.e., clean the dirty collection A using **s-self** first and apply the sequential linkage for clean-clean case. To distinguish from the **s-DC** of Algorithm 6, we denote this implementation as **s-DC**$_{self}$. Algebraically, the following holds: **s-DC**$_{self}(A,B) \equiv$ **s-CC**(**s-self**(A),B).

Note that algorithms **s-DC** and **s-DC**$_{self}$ behave differently depending on the level of "dirty-ness" within A or between A and B. For instance, consider three records, a_i, $a_k \in A$ and $b_j \in B$. Suppose the following relationship occurs: $a_i \oplus a_k \sqsubset b_j$. Then, using **s-DC**$_{self}$, $a_i \oplus a_k$ will be compared again with other records in A. However, using **s-DC**, a_i and a_k will be removed, saving $|A|$ number of comparisons. On the other hand, for instance, assume that $a_i \approx a_k$, $a_i \not\approx b_j$, and $a_k \not\approx b_j$. Using **s-DC**$_{self}$, there is only one comparison after $a_i \approx a_k$ is made. However, using **s-DC**, both a_i and a_k are compared to b_j before $a_i \approx a_k$ occurs, increasing the number of comparisons. In general, if the number of matches in A is significantly higher than that between A and B, then **s-DC**$_{self}$ is expected to perform better.

5.3.3 Dirty vs. Dirty

Since neither collection A or B is clean, more comparisons are needed for dirty-dirty case. By using sequential linkage algorithms for clean-clean or dirty-clean cases, we propose three variations, referred to as **s-DD1**, **s-DD2**, and **s-DD3**, as follows:

1. **s-DD1**$(A,B) \equiv$ **s-DC**(A,**s-self**(B))

2. **s-DD2**$(A,B) \equiv$ **s-CC**(**s-self**(A),**s-self**(B))
3. **s-DD3**$(A,B) \equiv$ **s-self**$(A \cup B)$

The different behaviors of variations will be evaluated experimentally.

5.4 Parallel Linkage

The high-level overview of Parallel Linkage using 2 processors is illustrated in Figure 5.3. Parallel linkage is a distributed algorithm to perform *match* and *merge* processes concurrently. To achieve this, either data or task needs to be partitioned and distributed to multiple processors. In an ideal parallel model, tasks are evenly distributed among all processors. However, in reality, such an even partition is not trivial since a task cannot be measured easily before it actually executes. In our setting, therefore, we estimate the number of record comparisons by the size of data, and use the data partition model to *simulate* the task partition model.

Given two inputs, A and B with $|A| \leq |B|$, the gist of our parallel algorithms is that each processor P_i has a replicated A and a partition of B, called B_i. At each P_i, then, an appropriate sequential linkage is done separately (e.g., **s-CC-single**(A,B_i) for clean-clean case). Once intra-processor cleanness is ensured using sequential linkage, next, inter-processor cleanness needs to be addressed. Therefore, the outputs of sequential linkage at P_i are then properly shipped and compared to the rest of data at the other processors. This process repeats until no more *merge* occurs at any processors.

To make the presentation simpler, we assume that each processor P_i has a local queue, Q_i, while there is a single global queue Q_G. With an efficient implementation using the linked list or priority queue, we assume that operations such as *enqueue*(a,Q), *enqueue*$(\{a_1, a_2\},Q)$ (= *enqueue*$(a_2,$*enqueue*$(a_1,Q)))$, and *dequeue*(Q) are efficiently supported for all queue data structures. Furthermore, we assume the following functions:

- The *partition*(Q_G) function, shown in Algorithm 8, takes a global queue Q_G (containing a "list" of list of records) as input, dequeues a list, say B, from Q_G, partitions it to k pieces of B_1, ..., B_k, and ships both the remainder of Q_G and B_i to each P_i.

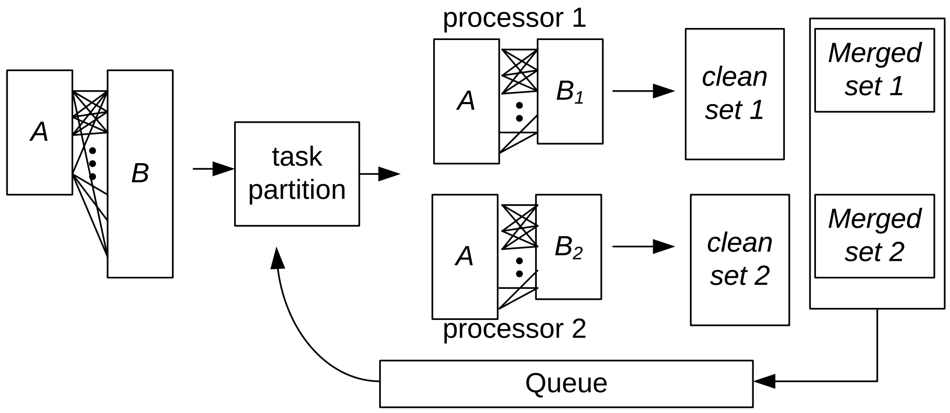


Figure 5.3. Parallel Linkage model with 2 processors.

- The *sync*($\{A_1$, ..., $A_k\}$, *op*) function synchronizes values of input sets of all processors with respect to the specified operator, *op*. For instance, with A_1=\{1,3,4\} and A_2=\{1,2,4,5\}, *sync*($\{A_1,A_2\}$, $\cap$) would synchronize two sets by applying $\cap$ to yield a set \{1,4\}. However, *sync*($\{A_1,A_2\}$,$\cup$) would yield a set \{1,2,3,4,5\}. To avoid communication cost among processors, in the implementation, one exchanges only indexes of input sets, instead of actual data sets.

- Recall that each processor P_i initially has a replicated A and partitioned B_i data sets. After a sequential linkage runs at each processor, depending on the occurrence of *match* and *merge* functions, results of each linkage may be different. In such a case, to ensure even distribution of data/task among all processors, one needs to re-partition B_i again. The *re-partition*($\{A$, B_1, ..., $B_k\}$) re-partitions B_i while considering A so that data are evenly distributed among A and all B_is. For example with two processors, if A=\{1,3,4\} and B_1=\{5,6\}, B_2=\{7,8,9\}, then *re-partition*($\{A,B_1,B_2\}$) results in A=\{1,3,4\}, B_1=\{5,6,1,3\}, and B_2=\{7,8,9,4\}. This will do load-balancing by adjusting number of partitioned records of B_i.

5.4.1 Clean vs. Clean

The Parallel Linkage for two clean inputs, referred to as **p-CC** in Algorithm 9, is the parallelization of the sequential linkage **s-CC**. Suppose there are k processors,

Algorithm 8: partition.

```
Input : A queue, $Q_G$, containing a list of list of records
Result: At each $P_i$, a sub-list $B_i$ and a local queue $Q_i$ are set
$B \leftarrow$ dequeue($Q_G$);
partition $B$ to $k$ sub-lists: $B_1$, ..., $B_k$;
ship $B_i$ and $Q_G$ to $P_i$;
foreach $P_i$ ($1 \leq i \leq k$) do $Q_i \leftarrow Q_G$
```

Algorithm 9: p-CC.

```
    Input  : Two non-empty clean lists $A$ and $B$
    Output: A single merged clean list $C$
9.1 enqueue({$B$,$A$},$Q_G$); /* initialize global queue */
9.2 partition($Q_G$); /* $Q_i$ and $B_i$ at $P_i$ are set */
    foreach $P_i$ ($1 \leq i \leq k$) do
        while $Q_i \neq \emptyset$ do
            $A_i \leftarrow$ dequeue($Q_i$);
            ($A'_i$,$B'_i$,$C_i$) $\leftarrow$ s-CC-single($A_i$,$B_i$);
9.3         $A'_i \leftarrow$ sync({$A'_1$, ..., $A'_k$}, $\cap$);
9.4         $B_i \leftarrow$ re-partition({$A'_i$,$B'_1$, ..., $B'_k$});
9.5         $Q_i \leftarrow$ sync({$C_1$, ..., $C_k$}, enqueue);
    $C \leftarrow B_1 \cup ... \cup B_k$; return $C$;
```

$P_1, ..., P_k$. Once the data set B is partitioned to B_i and shipped to each processor (lines 9.1 and 9.2), at each processor P_i, a single iteration of sequential linkage for clean-clean, **s-CC-single**, is applied to generate a clean set C_i and two intermediate sets of A'_i and B'_i. Note that the initial input data set A was replicated to all processors. However, each intermediate set of A'_i may be different since A is compared to different piece of B. Therefore, to avoid redundant comparison, we needs to synchronize all intermediate A'_i from all processors (line 9.3). Similarly, intermediate B'_i at each processor may have different values after **s-CC-single**. To increase the efficiency of Parallel Linkage, therefore, one needs to re-distribute B'_i across all processors (line 9.4). Finally, all clean sets C_i generated from **s-CC-single** are gathered and re-feeded into the local queue Q_i (line 9.5). This step is necessary since a clean set of P_1 still needs to be compared to intermediate sets in P_1 as well as in another processor P_2.

The algorithm to clean n clean input sets, termed as **p-CC-multi**, can be straightforwardly made by extending the **p-CC** that links "two" clean input sets (i.e., at line 9.1 of **p-CC**, all n input clean sets need to be enqueued to Q_G).

Example 4. Using Figure 5.4, let us illustrate how merged sets can have inter- and intra-comparisons to intermediate sets iteratively in **p-CC**. Consider two clean input sets, $A = \{a_1, a_2, a_3\}$ and $B = \{b_1, b_2, b_3, b_4, b_5\}$, and two processors, P_1 and

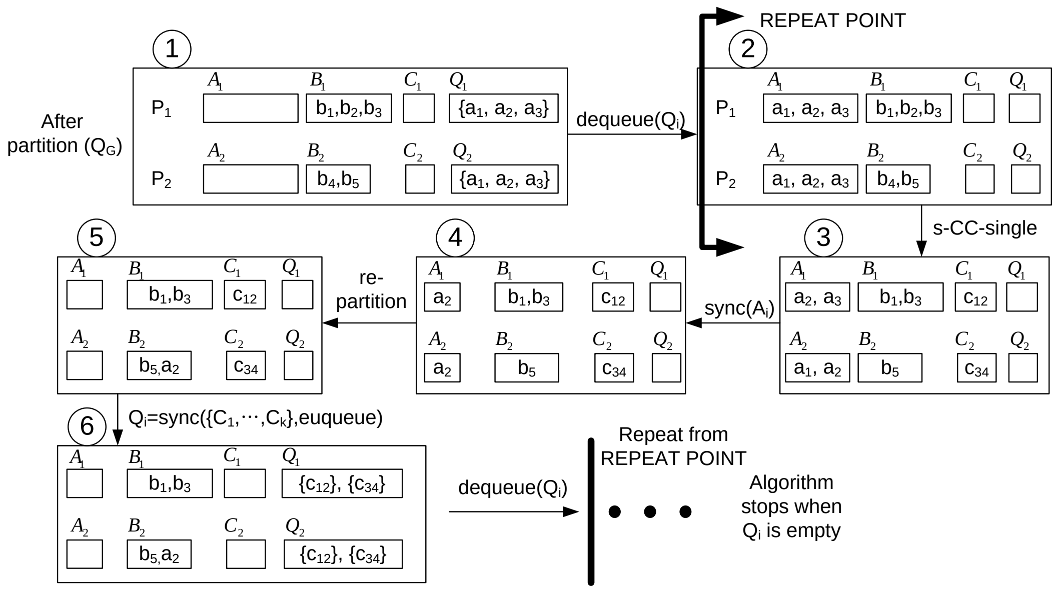


Figure 5.4. Single iteration of **p-CC**.

P_2. Furthermore, let us assume that only two *merge*'s occur: $c_{12} = a_1 \oplus b_2$ and $c_{34} = a_3 \oplus b_4$. Since $|B| > |A|$, B is partitioned to $B_1 = \{b_1, b_2, b_3\}$ in P_1 and $B_2 = \{b_4, b_5\}$ in P_2 and A is replicated to Q_1 and Q_2 (first box in Figure 5.4). Now, after dequeue(Q_i) is done (second box) and **s-CC-single** runs at each processor (third box), we have: $A'_1 = \{a_2, a_3\}$, $B'_1 = \{b_1, b_3\}$, and $C_1 = \{c_{12}\}$ at P_1, and $A'_2 = \{a_1, a_2\}$, $B'_2 = \{b_5\}$, and $C_2 = \{c_{34}\}$ at P_2 (third box). Then, sync($\{A'_1, A'_2\}, \cap$) of line 9.3 in **p-CC** yields $A'_1 = A'_2 = \{a_2\}$ (fourth box). Since B_1 and B_2 have different left-over and $|B_2| < |B_1|$, by re-partition on line 9.4 in **p-CC**, a_1 is added to B_2 to make $B_1 = \{b_1, b_3\}$ and $B_2 = \{b_5, a_2\}$ (fifth box). On line 9.5 in **p-CC**, if C_i is not empty, algorithm enqueues C_i to all local queues. Then, both Q_1 and Q_2 contain both C_1 and C_2 by synchronizing C_1 and C_2 among other processors (sixth box). This steps repeat until a queue Q_i is empty. □

5.4.2 Dirty vs. Clean

We propose two different parallel schemes, named as **p-DC**$_{self}$ and **p-DC**, similar to two sequential schemes of **s-DC**$_{self}$ and **s-DC**, respectively.

In **p-DC**$_{self}$ of Algorithm 10, first *dirty* input set A is partitioned to A_i and distributed to each processor. Then, each A_i is separately cleaned by applying

s-self at each processor. At this point, k *clean* sub-lists and one *clean* input list, B, remain. Then, all these clean sub-lists can be gathered and cleaned, including B, by **p-CC-multi**.

In **p-DC**, like **p-DC**$_{self}$, the *dirty* list A is partitioned to k sub-lists, $A_1, ..., A_k$ at each processor. However, unlike **p-DC**$_{self}$, the *clean* list B is also shipped to each processor. Then, clean-clean case of sequential linkage algorithm, **s-CC-single**(A_i, B) is executed at each processor instead of **s-DC-single**, because we cannot simply union A'_i and B'_i in each processor to avoid duplicate matching processes in the next iteration. Note that A_i is a partitioned list but B_i is common in all processors. However, by comparing dirty set directly to the clean set without the self-clean process in advance like **p-DC**$_{self}$, when records in a dirty set, A, mostly exist in a clean set, B, then the merging time is improved as the same case in a **s-DC**. The result lists from the first operation by **s-CC-single**, we have k sub-lists, $A'_1, ..., A'_k$, k sub-lists, $C_1, ..., C_k$, and a synchronized B_i. Among these lists, $A'_i, ..., A'_k$ are dirty, so that **s-DC-single** is applied to each A'_i at each processor by assuming the initial clean set is empty. As a result, we have k clean lists, A'_i and k merged dirty lists, M_i. Even though M_i is cleaned by **s-self**, it should be compared again with B_i and A'_i. Thus, M'_i by **s-self**(M_i) is put aside to perform **p-CC-multi** later. Even each A'_i is clean at each processor, the set of A is dirty initially. Thus, A'_is should be compared each other. In addition, the nature of iterative merging, the merged records generated by comparing A'_is should be compared again with all other list. Now, we use **p-CC-multi-alter** in a line 11.1 that returns two sets. Note that this algorithm is omitted because it can be simply modified by **p-CC-multi**. A return set of A' is for original records belonging to an initial set of A, and R is merged records modified by merging process. The R should be compared again with all other lists, but it is clean by itself. Once A and B were compared at an initial step, simple union should work for merging two sets. Therefore, in each processor, we have a new B_i by union of A and B_i in a line 11.2. Now, we have all clean lists, but they should be compared each other. Thus, multiple clean lists from all processors can be gathered and cleaned by **p-CC-multi** finally.

Algorithm 10: **p-DC**$_{self}$.

```
Input  : A non-empty dirty list $A$ and a clean list $B$
Output: A single merged clean list $C$
/* Dirty set $A$ is partitioned */
enqueue($\{A\}$,$Q_G$); partition($Q_G$);
foreach $P_i$ ($1 \le i \le k$) do  $A_i \leftarrow$ s-self($A_i$);
/* There are $k$ clean sub-lists of $A_i$ and one clean $B$ */
enqueue($\{B, A_1, ..., A_k\}$,$Q_G$);
$C \leftarrow$ p-CC-multi($Q_G$); return $C$;
```

Algorithm 11: **p-DC**.

```
     Input  : A non-empty dirty list $A$ and a clean list $B$
     Output: A single merged clean list $C$
     /* Dirty set $A$ is partitioned */
     enqueue($\{A, B\}$,$Q_G$); partition($Q_G$);
     /* $Q_i$ and $A_i$ at $P_i$ are set */
     foreach $P_i$ ($1 \le i \le k$) do
         $B_i \leftarrow$ dequeue($Q_i$);
         $A'_i, B'_i, C_i \leftarrow$ s-CC-single($A_i$,$B_i$);
         $B'_i \leftarrow$ sync($\{B'_1$, ..., $B'_k\}$, $\cap$);
         /* $A'_i$ and is still dirty, but $B'_i$ and $C_i$ are clean */
         $D_i \leftarrow \emptyset$;
         $(D'_i, M_i) \leftarrow$ s-DC-single($A'_i, D_i$); $A'_i \leftarrow D'_i$;
         $M'_i \leftarrow$ s-self($M_i$);
         $Q_A \leftarrow$ sync($\{A'_1$, ...,$A'_k\}$, enqueue);
11.1 $(A', R) \leftarrow$ p-CC-multi-alter($Q_A$);
     move $A'$ to each processor;
     foreach $P_i$ ($1 \le i \le k$) do
11.2     $B_i \leftarrow B_i \cup A'$;
         $Q_G \leftarrow$ sync($\{B'_i$, $M'_1$, ..., $M'_k$, $C_1$, ..., $C_k\}$, enqueue);
     enqueue($\{R\}$, $Q_G$); $C \leftarrow$ p-CC-multi($Q_G$); return $C$;
```

5.4.3 Dirty vs. Dirty

We propose two Parallel Linkage schemes to handle two dirty lists: (1) **p-DD1**, parallelization of **s-DD1**, cleans one dirty set first, then apply **p-DC**, while **p-DD2**, parallelization of **s-DD2**, attempts to clean both sets at the same time and apply **p-CC**.

In **p-DD1** of Algorithm 12, first, data set B is partitioned to k pieces and cleaned by **s-self** at each processor. When k clean sub-lists of B are created, they are merged back via a queue and cleaned by the Parallel Linkage solution **p-CC-multi**. After B is cleaned and stored in B_{clean}, then, we apply **p-DC** to get a merged clean list of C. In the **p-DD2** scheme, each input set A and B are separately partitioned and cleaned by **s-self**, generating $2k$ clean sub-lists of A and B. Sub-lists from A and Sub-lists from B are cleaned by **P-CC-multi**, respectively. Finally, cleaned A and cleaned B are merged again by **p-CC**.

Algorithm 12: **p-DD1**.

```
Input  : Two non-empty dirty lists $A$ and $B$
Output: A single merged clean list $C$
/* Clean $B$ first */
enqueue($B$,$Q_G$); partition($Q_G$); /* $B_i$ and $Q_i$ are set at $P_i$ */
foreach $P_i$ $(1 \leq i \leq k)$ do  $B_i \leftarrow$ s-self($B_i$);
$B_{clean} \leftarrow$ p-CC-multi($\{B_1, ..., B_k\}$);
$C \leftarrow$ p-DC($A, B_{clean}$); return $C$;
```

Algorithm 13: **p-DD2**.

```
Input  : Two non-empty dirty lists $A$ and $B$
Output: A single merged clean list $C$
/* Clean $A$ and $B$ independently */
partition($A$); partition($B$); /* $A_i$ and $B_i$ are set at $P_i$ */
foreach $P_i$ $(1 \leq i \leq k)$ do  $B_i \leftarrow$ s-self($B_i$); $A_i \leftarrow$ s-self($A_i$);
$B_{clean} \leftarrow$ p-CC-multi($\{B_1, ..., B_k\}$); $A_{clean} \leftarrow$ p-CC-multi($\{A_1, ..., A_k\}$);
$C \leftarrow$ p-CC($A_{clean}, B_{clean}$); return $C$;
```

5.5 Experimental Validation

In this section, under various settings, we evaluate the performance of six sequential linkage frameworks (**s-CC**, **s-DC**, **s-DC**$_{self}$, **s-DD1**, **s-DD2**, and **s-DD3**) and five Parallel Linkage frameworks (**p-CC**, **p-DC**, **s-DC**$_{self}$, **p-DD1**, and **p-DD2**).

5.5.1 Set-Up

All proposed algorithms are implemented in the Distributed MATLAB, and executed in the LION-XO PC Cluster at Penn State[2], which includes 133 nodes, each with dual 2.4–2.6GHz AMD Opteron processor and 8GB–32GB memory. Since it is a multi-user multi-tasking machine, RT is measured as the average of multiple runs (i.e., 5-10).

Data Sets. Errors are synthetically introduced to real citation data from DBLP according to two matching rates: (1) Internal Matching Rate of A: **IMR**(A) = $\frac{\text{\# of dirty records in } A}{\text{\# of all records in } A}$, and (2) Cross Matching Rate of A against B: **CMR**(A,B) = $\frac{\text{\# of records in } A \text{ that matches a record in } B}{\text{\# of all records in } A}$. By varying both IMR and CMR, we control the "dirty-ness" of data sets. When errors are introduced, four types of matching errors (e.g., $\equiv$, $\sqsubseteq$, $\sqsupseteq$, and $\oplus$) are *uniformly* distributed. For instance, to have an IMR of 0.4 for A, we synthetically generate 40% of A as matching (i.e., dirty) records, with 10% each for $\equiv$, $\sqsubseteq$, $\sqsupseteq$, and $\oplus$ types. To compare directly against

[2]http://gears.aset.psu.edu/hpc/systems/lionxo/

P-Swoosh and P-Febrl and keep the experimentation manageable, data sets of 100 – 50,000 records in size are used. Despite their relatively small sizes, consistent performance patterns emerge (to be shown) and one can easily extrapolate the performance for very large data sets. Note that distance metric between two citation records used Jaccard similarity with the threshold $\theta = 0.5$ by default.

Evaluation Metrics. Two main metrics are used as baseline: *wall-clock running time*, denoted as **RT** and *number of comparison* for *match* functions, denoted as **NC**. Then, the speedup and efficiency for parallel algorithms are defined in terms of RT and NC.

- **Speedup** shows the rate of increase for parallel system, and takes into account the overhead (e.g., time for startup, communication, synchronization for deadlock prevention, or data re-distribution) of parallel execution: $\textbf{speedup}_{RT} = \frac{RT_s}{RT_p}$, where RT_p and RT_s is the RT of the parallel and the "best" serial execution, respectively, and $\textbf{speedup}_{NC} = \frac{NC_s}{NC_p}$, where NC_p and NC_s is the NC of parallel and the "best" serial execution, respectively. Here NC_p is the accumulation of maximum NC among processors during iterations.

- **Efficiency** indicates the ability to gain proportionate increase in speedup with the addition of more processors [38]: $\textbf{efficiency}_{RT} = \frac{\textbf{speedup}_{RT}}{\text{\# of processors}}$ and $\textbf{efficiency}_{NC} = \frac{\textbf{speedup}_{NC}}{\text{\# of processors}}$.

5.5.2 Among Sequential Linkages

First, RT and NC among sequential algorithms are compared. Figure 5.5 shows both RT and NC of six sequential algorithms using IMR=0.0 and CMR=0.3 and 100 to 5,000 records. Although data is a clean-clean case (i.e., IMR=0.0), sequential algorithms for dirty-clean or dirty-dirty cases pretend *not* to know that they are clean so that comparison among all six algorithms is possible. Note that both graphs for RT and NC show consistent patterns of $O(N^2)$, where N is size of input. Since one does not need to compare records internally, **s-CC** shows the best RT and NT among sequential algorithms. For dirty-clean case, **s-DC** outperforms **s-DC**$_{self}$. Since **s-DC** compares records across A and B, due to high CMR of

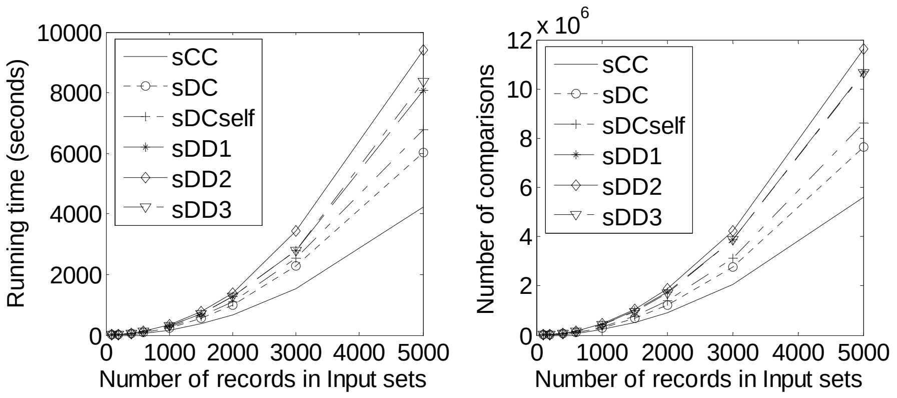


Figure 5.5. The RT and NC of six sequential algorithms (IMR=0.0 & CMR=0.3).

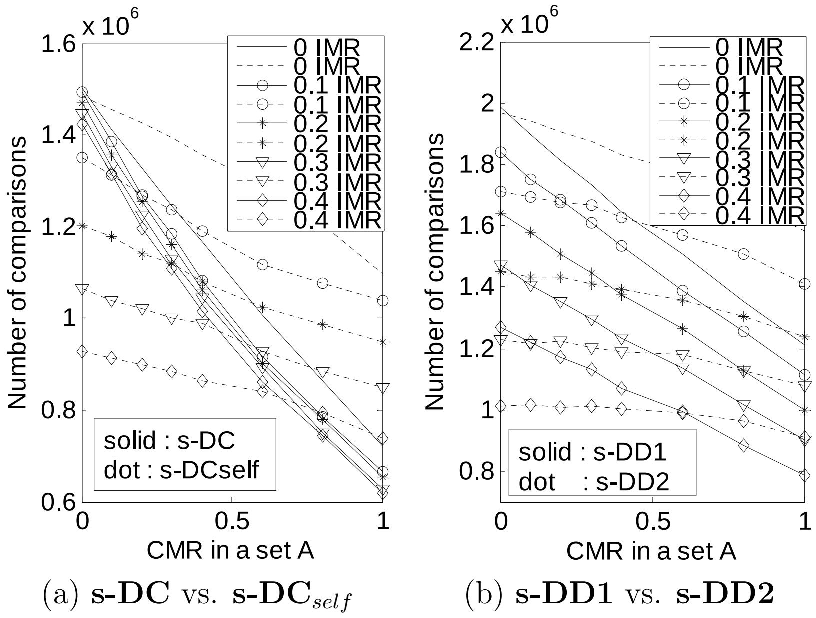


Figure 5.6. The total NC of four sequential algorithms with different IMR and CMR.

0.3, after one iteration, **s-DC** matches and merge many records, reducing NC at subsequent iterations. For dirty-dirty case, RT of **s-DD1** and **s-DD3** are similar while both outperform **s-DD2**. Because of direct comparisons between A and B, records in A will be removed before being compared with records in the same set. Therefore, RT of **s-DD1** or **s-DD3** is faster than that of **s-DD2**.

Second, we compared how IMR or CMR affects the performance of sequential algorithms. We used five variations of IMR (0.0, 0.1, 0.2, 0.3, and 0.4) and eight variations of CMR (0.0, 0.1, 0.2, 0.3, 0.4, 0.6, 0.8, and 1) with 2,000 records. Both IMR and CMR are applied on A in dirty-clean case. For dirty-dirty case, both A

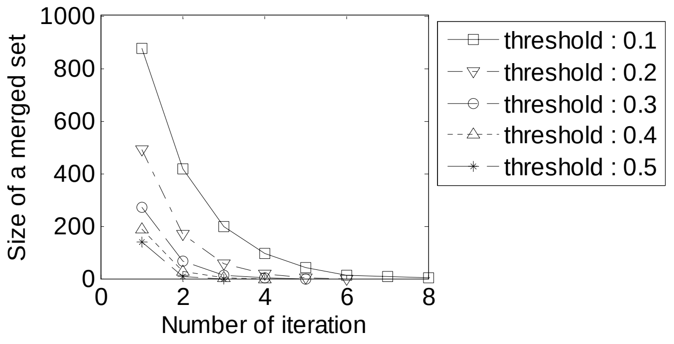


Figure 5.7. Iterations of **s-self** algorithm.

and B take the same IMR, and CMR is applied only to A. Since results of both RT and NC are similar, here, we present only results of NC. In Figure 5.6, **s-DC/s-DD1** and **s-DC**$_{self}$/**s-DD2** are shown as solid and dotted lines, respectively. For dirty-clean case, NC decreases as both IMR and CMR increase in both **s-DC** and **s-DC**$_{self}$. Therefore, the data set with IMR=0.4 and CMR=1 (i.e., dirtiest data set) gives the best performance. The effect of IMR is more significant in **s-DC**$_{self}$ while the effect of CMR is more significant in **s-DC**.

This happens because, in **s-DC**$_{self}$, all redundant data are first merged during **s-self** stage, reducing CMR when **s-CC** is applied later. On the other hand, in **s-DC**, since the dirty set A is first compared to the clean set B, if CMR is high, then more records are merged at the first iteration. In conclusion, CMR (resp. IMR) is the dominant factor in **s-DC** (resp. **s-DC**$_{self}$). This conclusion can be interpreted as: the MR at the first iteration plays a major role on NC. Because of this, with IMR=0.0, **s-DC** always performs better. NC is significantly reduced by the increase of CMR on **s-DC**, i.e., **s-DC** performs better when it gets a higher CMR. As an example, with IMR=0.1, there is a cross-over point between **s-DC** and **s-DC**$_{self}$ at CMR=0.2. In general, **s-DC** is better with a higher CMR, and **s-DC**$_{self}$ is better with a higher IMR. The patterns of dirty-dirty case is analogous to those of dirty-clean case. With a large CMR and a small IMR, in general, **s-DD1** outperforms **s-DD2**.

Finally, Figure 5.7 illustrates the iterative nature of sequential linkage algorithm, **s-self**. Matched records at each iteration are merged into a new record, and re-compared to the rest of records at subsequent iteration. Therefore, sequential linkage algorithms continue until no more new merged record occur. With the

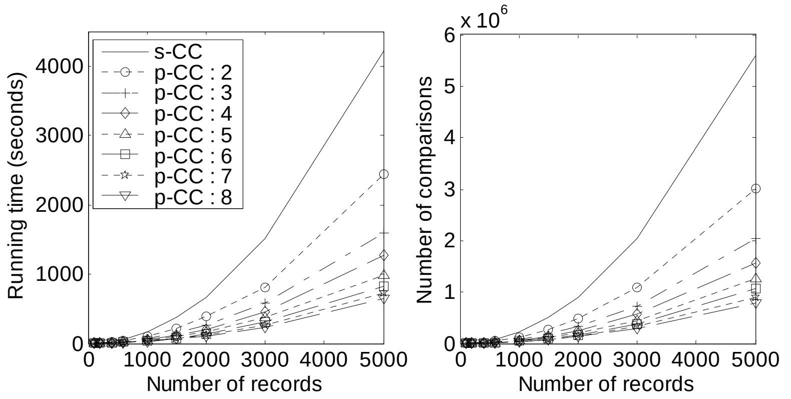


Figure 5.8. The RT and NC of **p-CC** (# in legend is # of processors).

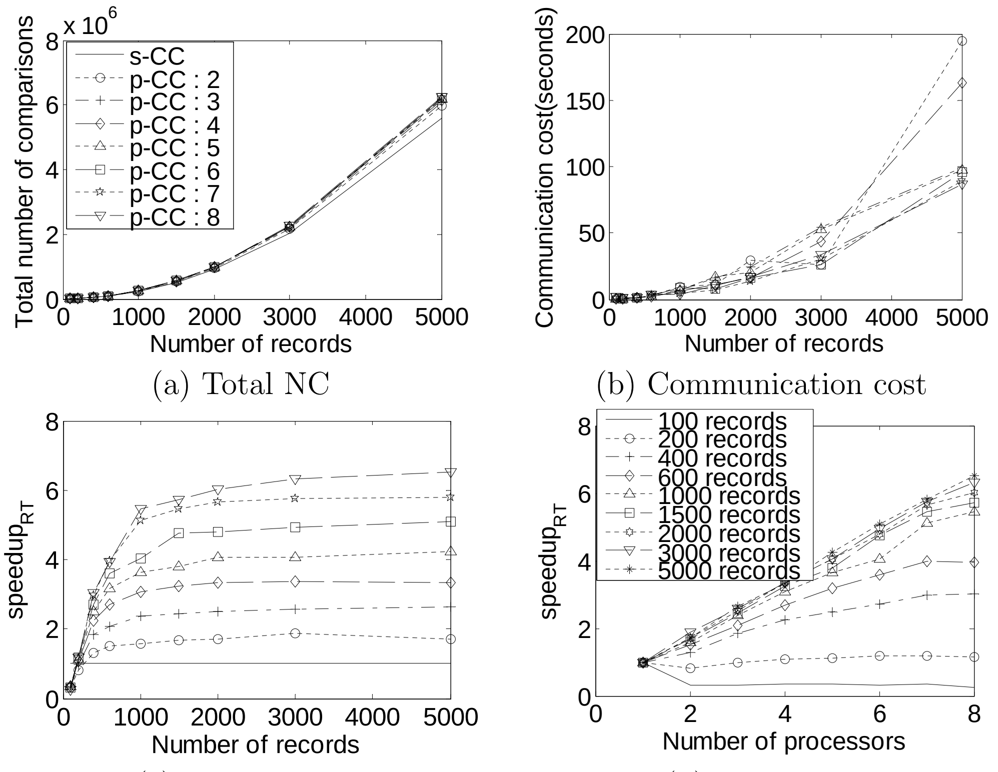
data set of 2,000 records, depending on the default threshold θ for distance metric, Figure 5.7 shows that 3-8 iterations are needed to do complete self-clean.

5.5.3 Sequential vs. Parallel Linkages

Now, using $\mathbf{speedup}_{RT}$ and $\mathbf{speedup}_{NC}$, we compare sequential and parallel solutions. Up to 32 processors and 50,000 records are used. Since relative performance of all parallel algorithms, compared to sequential ones, are similar, we show only detailed behavior of **p-CC** under various characteristics.

In Figure 5.8, intuitively, both RT and NC decrease as # of processors increases. Despite overhead cost of parallel execution, both RT and NC show the same pattern. However, speedup on RT is more affected by parallel overhead and it is shown in Figure 5.9. Figure 5.9(a) shows that although total # of NC increases as the input size increases, it gets little affected by # of processors used. That is, as more # of processors are used, it may increase involved overhead among processors (as communication cost shown in Figure 5.9(b)), but it does *not* increase # of comparison since **p-CC** has few redundant computations among processors – an ideal property for parallel algorithm. In our experimentation, communication overhead such as ones in Figure 5.9(b) typically consume 10% of total RT.

Therefore, in general, $\mathbf{speedup}_{RT}$ is less than $\mathbf{speedup}_{NC}$. Because it takes about 4-7 seconds to submit a parallel job and recollect final clean sets from processors, $\mathbf{speedup}_{RT}$ is very low when input data is less than 100 in Figure 5.9(c). With larger input data, however, RT is less affected by communication cost, and speedup with more processors is higher than that with less processors. When 8



(a) Total NC (b) Communication cost

(c) **speedup**$_{RT}$ (d) Scalability

Figure 5.9. Details of **p-CC** (legend shown in (a)).

processors are used, **speedup**$_{RT}$ is close to 6.55. Finally, Figure 5.9(d) shows how scalable **p-CC** is. When data is sufficiently large and enough # of processors is used, **p-CC** shows linear increase of **speedup**$_{RT}$ – another ideal property of parallel algorithm.

5.5.4 Among Parallel Linkages

Figure 5.10 shows the comparison of five Parallel Linkage algorithms with respect to their efficiency. Also, both RT and NC are shown after being normalized (i.e., re-scaled to 0-1 range). Overall, **efficiency**$_{NC}$ is better than **efficiency**$_{RT}$ due to various overhead negatively affecting RT of parallel algorithms. Among parallel algorithms, **p-DD2** shows the best efficiency in both RT and NC. Because of the characteristic of input data (IMR=0.0 and CMR=0.3), using clean property gives better performance on both RT and NC. Thus, even though **p-DD2** gives the best efficiency overall, its RT and NC are also the highest. Specifically, note that RT of **s-DC** is 1.11 times faster than that of **s-DC**$_{self}$ and RT of **s-DD1** is also 1.11 times faster than that of **s-DD2**. In general, in terms of RT, parallel algorithms are in order of **p-CC** (fastest) <**p-DC**<**p-DC**$_{self}$<**p-DD1**<**p-DD2** (slowest).

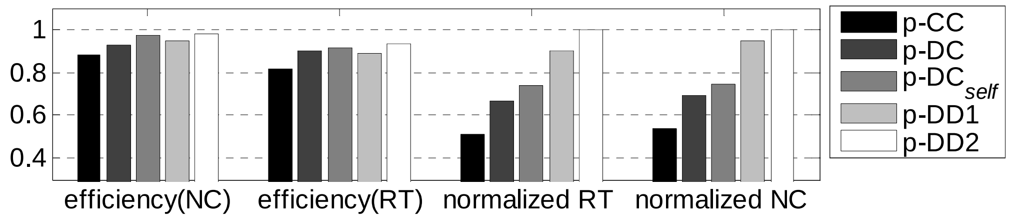


Figure 5.10. The RT and NC of five parallel algorithms (IMR=0.0, CMR=0.3, 8 processors, and 5,000 records).

We also tried larger data set with more processors– 50,000 records with 16 processors and 32 processors. Since RT follows $O(N^2)$, an approximate equation for RT of a sequential linkage, say **s-CC**, can be obtained by *polyfit()* in MATLAB as: RT$= 0.000173x^2 - 0.01035x + 3.645$, where x is the size of input data. That is, for 50,000 records, a sequential linkage algorithm such as **s-CC** would take about 120 hours to finish the job. However, when Parallel Linkage algorithms are used, RT can be reduced to about 9.17 hours with 16 processors and 4.7h with 32 processors. That is, we achieve **speedup**$_{RT}$=13.08/**efficiency**$_{RT}$=0.8175 and **speedup**$_{RT}$=25.53/**efficiency**$_{RT}$=0.7979 with 16 and 32 processors, respectively.

5.5.5 Against P-Swoosh and P-Febrl

To our best knowledge, there are two Parallel Linkage solutions comparable to our proposals: P-Swoosh [52] from Stanford SERF project and parallel Febrl (P-Febrl) [22] from ANU record linkage project. Since space complexities of all three proposals are similar, let us focus on the comparison of speedup and efficiency. It is important to emphasize that in parallel experimentation, it is not straightforward to compare RT or NC directly. This is because RT may change depending on parallel execution model, choice of data characteristics, environment of execution system. However, the "ratio" of how much parallel solutions improve upon sequential solutions is meaningful. That is, if the ratio such as speedup in environment X is higher than that in Y, regardless of algorithmic details, one can argue that the parallel solution of X be superior to that of Y[3].

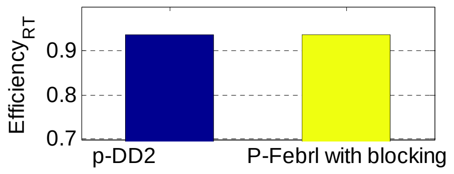
Since both P-Febrl and P-Swoosh studied only dirty-dirty case, here, we compare using **s-DD2** for sequential and **p-DD2** for parallel case. In addition, in [52], P-Swoosh reports only $\textbf{speedup}_{NC}$ while in [22], P-Febrl reports only $\textbf{speedup}_{RT}$. Therefore, we compare each against our solution separately. # of records and processors used in the experimentation are:

```
P-Febrl  : 20,000 records, 4 processors (w. blocking)
P-Swoosh : 5,000 records, 16 processors (NO blocking)
Ours     : 50,000 records, 32 processors (NO blocking)
```

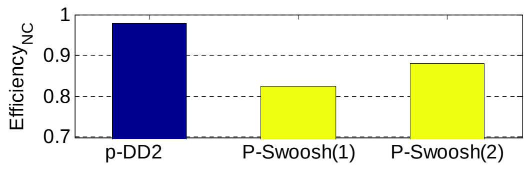
Note that P-Febrl reports only results using *blocking* in linkage while both P-Swoosh and ours use nested-loop style linkage (thus no blocking). Therefore, the speedup of P-Febrl should be much higher than those of P-Swoosh and ours. As shown in Figure 5.11(a), however, our algorithms performs only 0.16% worse than P-Febrl (0.9360 vs. 0.9375). In [22], P-Febrl reports that their use of blocking and indexing on both sequential and parallel algorithms reduces substantial communication cost so that at the end only 0.35% of overall RT is due to the communication. In our experimentation, however, communication cost consumes about 7–13% of overall RT, leaving much room for improvement if blocking was used. Therefore, despite seemingly lower efficiency of our parallel algorithms in Figure 5.11(a) than that of P-Febrl, we believe that our parallel solutions is *more* efficient compared to P-Febrl. We plan to verify this claim in future work when blocking/indexing is combined with our parallel solutions. As shown in Figure 5.11(b), against P-Swoosh, our algorithm, **P-DD2**, with $\textbf{efficiency}_{NC}$=0.9781 is 11.15% better than P-Swoosh (shown as P-Swoosh(2)) that has $\textbf{efficiency}_{NC}$ of 0.88. However, the result of P-Swoosh in [52] only considered the number of slave nodes in computing the efficiency, without including the master node. If the master node is also counted as # of processors (as it should be), then their $\textbf{efficiency}_{NC}$ drops to 0.825 (shown as P-Swoosh(1) in Figure 5.11(b)). Therefore, our proposals shows 11.15–18.56% improvement on $\textbf{efficiency}_{NC}$ against P-Swoosh[4].

[3]The only matter that significantly affects the performance of both sequential and parallel algorithms is the characteristics of data sets. To ensure fair comparison, at all possible, we tried to compare similar input cases such as clean-clean or dirty-dirty.

[4]According to [52], their best $\textbf{speedup}_{NC}$ is more than 30 times using 10 processors.



(a) **efficiency**$_{RT}$



(b) **efficiency**$_{NC}$

Figure 5.11. Comparison with other parallel schemes.

5.5.6 Observation of Parallel Linkages

With our input data set, RT and NC of sequential algorithms are ordered by **s-CC** (best) <**s-DC**<**s-DC**$_{self}$<**s-DD1**≈**s-DD3**< **s-DD2** (worst). Study on IMR and CMR shows that different algorithms can be used for better performance. Specially, for the one dirty set, **s-DD2** is always better than **s-DD1**. In Parallel Linkage, RT and NC of parallel algorithms are in the order of **p-CC** (best)<**p-DC**<**p-DC**$_{self}$ <**p-DD1**<**p-DD2** (worst). However, the order of **efficiency**$_{NC/RT}$ among parallel algorithms is **p-CC**<**p-DC**≈**p-DC**$_{self}$ ≈**p-DD1**<**p-DD2**. Efficiency of our algorithm performs comparably against P-Febrl (but ours do not use blocking and indexing while P-Febrl does) and performs better than P-Swoosh when the same comparison schemes are used on both parallel and sequential algorithms.

5.6 Summary

In this chapter, parallel version of record linkage problem is studied for the corresponding sequential algorithms in detail. For three input cases of clean-clean, dirty-clean, and dirty-dirty, we presented six sequential and five parallel solutions. Our proposed parallel algorithms are shown to exhibit consistent improvement in speedup and efficiency when compared to sequential ones, by minimizing communication costs and well designed task partitioning. In addition, compared to two other competing parallel solutions, ours show 11.15–18.56% improvement in efficiency.

However, since they used radically different sequential and parallel schemes in so doing, this improvement of 30 is not meaningful. Therefore, we compare ours against their compatible model of FIX-1.

Chapter 6

Hashed Linkage

In this chapter, we study the performance issue of a hashing structure for the "iterative" *record linkage (RL)* problem discussed in Chapter 5, where *match* and *merge* operations may occur together in iterations until convergence emerges. We first propose the *Iterative Locality-Sensitive Hashing (I-LSH)* that dynamically merges LSH-based hash tables for quick and accurate blocking. Then, each *I-LSH* structure is developed by exploiting inherent characteristics within/across data sets (i.e., clean-clean, clean-dirty, and dirty-dirty). As mentioned in Chapter 1, the iterative nature of a record linkage problem requires immense amount of matching comparisons, specially with large-scale data collection.

Toward this challenge, for the more general *match-merge* RL model, we present novel hashed record linkage algorithms that run much faster with comparable accuracy. In particular, our contributions in this chapter are: (1) We extend the MinHash based LSH technique to propose the *Iterative LSH (I-LSH)* that iteratively and dynamically merges LSH-based hash tables to provide a quick and accurate blocking; (2) Using the I-LSH proposal, depending on three scenarios, we propose a suite of RL solutions, termed as HARRA (<u>HA</u>shed <u>R</u>eco<u>R</u>d link<u>A</u>ge) or Hashed Linkage, that exploits data collection characteristics. (3) The superiority of Hashed Linkage in speed over competing RL solutions is thoroughly validated using various real data sets, while maintaining equivalent or comparable accuracy levels.

6.1 Iterative LSH

6.1.1 Vector Presentation

LSH-based hashing idea exploit high-dimensional vectors as input data. Unlike the numerical values can be directly applied to LSH idea, string format records cannot be used directly. In turn, unlike the similarity between numerical values can be easily computed by various measures such as cosine or L-norm distances, the similarity between string values are not. To use hashing for the RL process, in particular, we convert string format records into multi-dimensional binary vectors as follows. First, unique q-gram tokens (e.g., bi-gram or tri-gram) from all records are gathered. If one only considers English alphabets, the maximum dimensions, N, are $26^2 = 676$ and $26^3 = 17,576$ for bi-gram and tri-gram, respectively.

The N dimensions of a token vector, D, is expressed by $\{d_1, d_2, ..., d_N\}$ where d_i is a unique token in a data collection. Note that each record r contains non-duplicate tokens of $\{t_1, t_2, ..., t_n\}$. Then, an N-dimensional binary vector of a record is obtained by setting the value of a token dimension to 1 if the token in a record exists in $\{d_1, d_2, ..., d_N\}$, and 0, otherwise. We refer to a function that converts a string format record r to a binary vector v as **binary(r)**. We use notations in Table 5.1 throughout this chapter.

6.1.2 LSH with MinHash

The basic idea of the *Locality-Sensitive Hashing (LSH)* technique was introduced in [37]. The LSH method originally addresses the approximate nearest neighbor problem by hashing input data objects (with respect to their features) such that similar objects are put into the same buckets with high probability. Since the number of buckets is much smaller than that of universe, LSH can be also viewed as a method for probabilistic dimension reduction. In the context of the RL problem, therefore, if LSH can hash input records into buckets such that duplicate records are put into the same buckets (and non-duplicate records in different buckets), then LSH can solve the RL problem. We first briefly introduce LSH. Let R be the domain of objects, and $dist()$ be the distance measure between objects. Then,

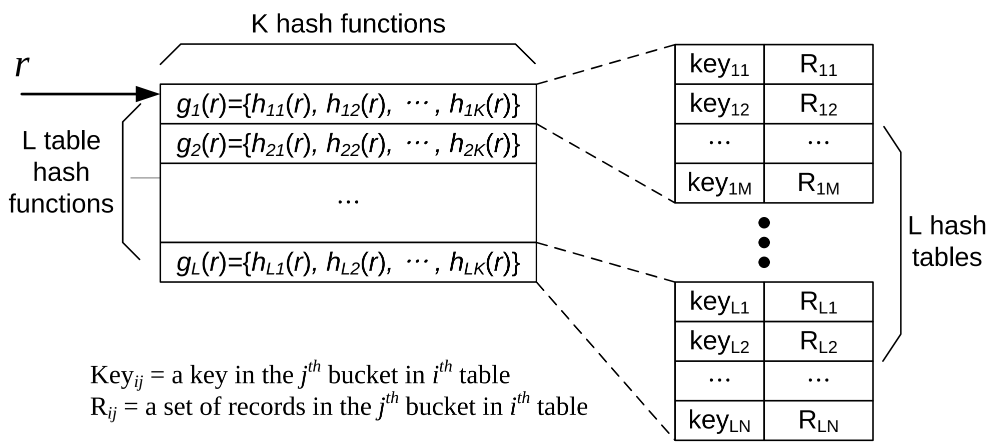

K hash functions
r
L table hash functions
g1(r)={h11(r), h12(r), ⋯, h1K(r)}
g2(r)={h21(r), h22(r), ⋯, h2K(r)}
⋯
gL(r)={hL1(r), hL2(r), ⋯, hLK(r)}
key11 R11
key12 R12
key1M R1M
keyL1 RL1
keyL2 RL2
keyLN RLN
L hash tables
Keyij = a key in the jth bucket in ith table
Rij = a set of records in the jth bucket in ith table

Figure 6.1. A basic LSH structure.

Definition 11 (LSH function family) *A function family $H = \{h : R \to U\}$ is called $(\gamma, c\gamma, p_1, p_2)$-sensitive for $dist()$ if for any $r_1, r_2 \in R$:*

- *If $dist(r_1, r_2) < \gamma$, then $Pr_H[h(r_1) = h(r_2)] \geq p_1$*
- *If $dist(r_1, r_2) > c\gamma$, then $Pr_H[h(r_1) = h(r_2)] \leq p_2$* □

We pick $c > 1$ and $p_1 > p_2$ for proper LSH. Different LSH function families can be used for different distance measures. By concatenating K number of LSH functions in H, one can generate a table hash function, $g()$. Therefore, various multi-table LSH indexing methods can be constructed by controlling two parameters, K and L, as follows (refer to Figure 6.1):

- **K**: # of hash functions from function family G=$\{g : S \to U^K\}$ such that $g(r)$=$\{h_1(r), h_2(r), ..., h_K(r)\}$
  where $h_i() \in H$ and $r \in R$.
- **L**: # of table hash functions (i.e., # of hash tables), $\{g_1, g_2, ..., g_L\}$, where $g_i(r)$=$\{h_1(r), h_2(r), ..., h_K(r)\}$.

As one increases K (i.e, the number of $h()$), one can reduce the probability of having non-matching records in the same buckets. However, it also increases the possibility of having matching records in different buckets. Therefore, in general, multiple hash tables, controlled by L, are required to achieve overall good precision and recall results. The overall structure of the LSH idea is illustrated in Figure

6.1. A record, r, returns L hash keys from $g_i(r)$ where $1 \leq i \leq L$. Each key, $g_i(r)$, returns candidate records in a bucket in i-th hash table. From L hash tables, then, the one final bucket containing all candidate records is selected for further probing. For further details of LSH, readers may refer to [37].

There are several methods to construct an LSH family such as bit sampling or random projection. In particular, MinHash [24] was shown to work well with sparse binary vectors. In our context, MinHash can be used as follows: (1) Select random re-ordering of all vector dimensions – i.e. select a random permutation of indices of D; (2) Apply this random permutation to re-order indices of a sparse binary vector. Note that one selected random permutation is used for all records to generate one digit of a key; and (3) Find the index (i.e., position) in which the first "1" occurs in a vector. This index becomes one of components in a hash key. Suppose a function **fi()** returns the first index containing "1" while **rp()** returns a selected random permutation. Then, with the record r and its binary vector representation v, the hash functions in H can be defined as: $h_i(r) = fi(rp(binary(r))) = fi(rp(v))$. We choose K random permutations to generate K hash functions, $\{h_1(), ..., h_K()\}$. By concatenating K hash functions, we finally obtain $g_i()$ as the i-th table hash function.

Example 5. Using three binary vectors as input: v_1 = [1,1,1,0,0], v_2 = [0,1,1,0,0], and v_3 = [0,0,1,1,1], let us construct a hash table $g(r) = \{h_1(r), h_2(r)\}$. The indices of vectors are (1,2,3,4,5). Suppose we select two random permutations: $rp_1()$=(2,3,5,4,1) and $rp_2()$=(5,4,2,1,3). Then, $h_1(r_1) = fi(rp_1([1,1,1,0,0])) = fi([1,1,0,0,1])$=1 and $h_2(r_1) = fi(rp_2([1,1,1,0,0])) = fi([0,0,1,1,1]) = 3$. Hence, a key of r_1 is $g(r_1) = \{h_1(r_1), h_2(r_1)\}$ ={1,3}. Likewise, $g(r_2)$ =$\{1,3\}$ and $g(r_3)$ =$\{2,1\}$. Therefore, in a hash table, two records, r_1 and r_2, are put into the same bucket, while r_3 in other bucket. Note that if we select $rp_2()$=(5,1,2,4,3), then r_1 and r_2 are placed in different buckets. To overcome this issue, the basic LSH requires multiple hash tables. $\Box$

6.1.3 Iterative LSH to Clean Single Dirty Collection

The basic LSH scheme hashes input records to different buckets in multiple hash tables. Despite its fast and effective performance, however, the basic LSH does

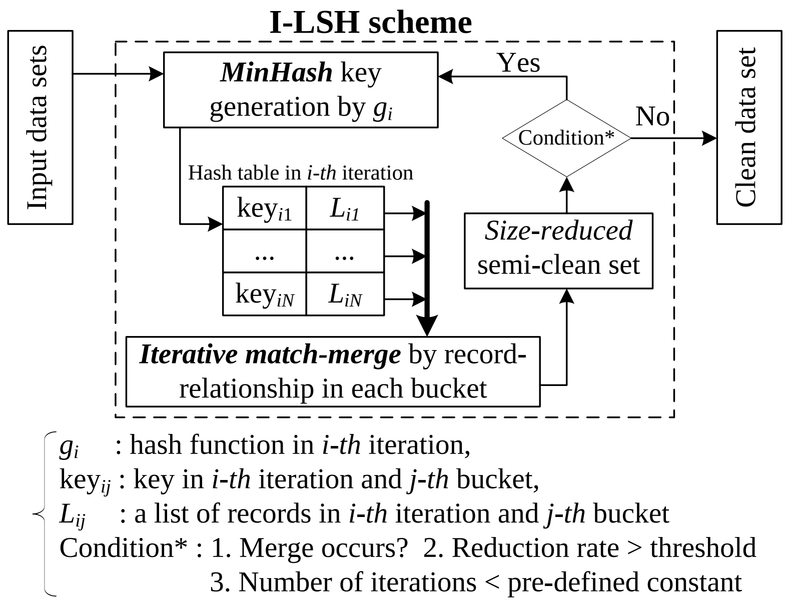

I-LSH scheme
Input data sets
MinHash key generation by g_i
Yes
No
Condition*
Clean data set
Hash table in i-th iteration
key$_{i1}$ L_{i1}
... ...
key$_{iN}$ L_{iN}
Size-reduced semi-clean set
Iterative match-merge by record-relationship in each bucket
g_i : hash function in i-th iteration,
key$_{ij}$: key in i-th iteration and j-th bucket,
L_{ij} : a list of records in i-th iteration and j-th bucket
Condition* : 1. Merge occurs? 2. Reduction rate > threshold
3. Number of iterations < pre-defined constant

Figure 6.2. The general structure of I-LSH.

not consider the “iterative” nature of *match-merge* process in the RL problem and has the following problems: (1) substantial hash table generation time using entire records at each iteration; (2) excessive memory requirement for hash tables and candidate record sets; and (3) high running time to handle many duplicate records in a candidate set. Therefore, the basic LSH scheme is not suitable for linking large-scale record collections. To address these problems, in this section, we introduce the *Iterative Locality-Sensitive Hashing (I-LSH)* where hash table generation time is greatly reduced via only single (re-usable) hash table.

Figure 6.2 illustrates the basic flow of I-LSH while Algorithm 14, $\mathbf{h\text{-}D}_{self}$, shows the detailed steps to clean single dirty data set. When a record, a_j, is hashed (by the MinHash key selection) into one of the buckets in a hash table, if the bucket (i.e., *AList* in Algorithm 14) is empty, a_j is initially placed in the bucket. If the bucket contains other records already, a_j is compared to existing records of the bucket, say a_k. If $match(a_k, a_j)$ returns $\equiv$ or $\sqsupseteq$, we remove one of the equivalent copies, say a_j, from A and continue to process a subsequent input record, a_{j+1}. For $\sqsubseteq$ relationship, on the other hand, we remove a_k from the bucket and a_j continues to be compared to a next record a_{k+1} in the bucket. For $\oplus$ relationship, a_k is removed from the bucket, then a_j is replaced by the “merged” record, created by $merge(a_k,\ a_j)$. Then, a_j is compared to the rest of records in the bucket. Once,

Algorithm 14: **h-D$_{self}$.**

```
Input  : A non-empty dirty list $A$
Output: A non-empty clean list $C$
Make $H$ as an empty hash table;
$Flag \leftarrow$ true; /* $Flag$ determines iterations */
while $Flag = true$ do
    $Flag \leftarrow$ false; $j \leftarrow 1$;
    while $j \leq |A|$ do
        /* hash $a_j$ to a bucket, AList */
        key $\leftarrow g_i(a_j)$; AList = H.get(key);
        if $AList \neq \emptyset$ then
            $k \leftarrow 1$;
            while $k \leq |AList|$ do
                switch $match(a_k, a_j)$ do
                    case $a_k \equiv a_j$ or $a_k \sqsupseteq a_j$
                        $Flag \leftarrow$ true; remove $a_j$ from $A$;
                        go to line 14.1;
                    case $a_k \sqsubseteq a_j$
                        $Flag \leftarrow$ true;
                        remove $a_k$ from AList; $k \leftarrow k + 1$;
                    case $a_k \oplus a_j$
                        $Flag \leftarrow$ true;
                        remove $a_k$ from AList & $a_j$ from $A$;
                        $a_j \leftarrow merge(a_k, a_j)$; $k \leftarrow 1$;
                    case $a_k \not\approx a_j$
                        $k \leftarrow k + 1$;
        AList.add($a_j$); H.put(key,AList);
14.1    $j \leftarrow j + 1$;
    $A \leftarrow H.getAll(keys)$; /* put all records back to $A$ */
    $i \leftarrow i + 1$ /* re-hash in next iteration using $g_{i+1}$ */
14.2 if termination condition is met then $Flag \leftarrow$ false;
$C \leftarrow A$; return $C$;
```

the system scans all records in A, then A is reset with all records in the hash table, and hashed again until the termination condition is met.

Every *match-merge* step of I-LSH reduces the size of both record set and hash table. Ideally, iteration will stop when convergence emerges (i.e., no more merge occurs). However, in practice, it is *not* plausible to re-iterate whole input only because single merge occurs in the previous iteration. Therefore, instead, I-LSH stops if the reduction rate σ_i ($= 1 - \frac{|\text{semi-cleaned set}_i|}{|\text{input set}_i|}$) at i-th iteration is less than a given threshold. This termination condition is captured as the "if" statement at line 14.2 of Algorithm 14.

Now, we analyze the time/space complexities of **h-D$_{self}$**. The running time of **h-D$_{self}$** at one iteration is bounded by the quadratic upper bound of two nested "while" loops in Algorithm 14. The worst case occurs when all records of A are hashed into the same bucket. Then, # of required comparison in **h-D$_{self}$** becomes:

$1+2+...+(m-1) = \frac{(m-1)(m)}{2}$, where $m = |A|$. That is, **h-D**$_{self}$ does not improve much upon the naive pair-wise comparison. Reversely, the best case occurs when no hash collision occurs. Then, # of required comparison at one iteration becomes simply m since a single scan of A suffices. In general, **h-D**$_{self}$ at one iteration has the running time of $O(m\hat{c})$, where $\hat{c}$ is the average # of hashed records in *dynamically-changing* buckets[1] in a hash table (i.e., *AList* in Algorithm 14). With a proper choice of hash functions, hash collisions should occur rarely. Therefore, in general, $\hat{c}$ is relatively small. Furthermore, in Algorithm 14, whenever one of the matching conditions occurs, the removal of a record either from A or *AList* occurs, limiting the growth of $\hat{c}$.

Lemma 7. ***h-D***$_{self}(A)$ *has the complexity of* $O(\sum_{i=1}^{T} m\sigma_i\hat{c}_i)$*, where* T *is the number of iterations,* $m = |A|$*,* σ_i *and* $\hat{c}_i$ *are the reduction rate and the average # of records in buckets, respectively, at i-th iteration.* ■

Note that, for a given data collection, most of similar records are merged during the first a few iterations (to be experimented in Figure 6.9 of Section 6.3). As a result, the reduction rate σ is significantly abated, i.e., only a small number of merges occur at later iterations. In addition, in **h-D**$_{self}$, the final running time is heavily influenced by the time to generate hash keys and hash tables.

As to the space complexity, since a hash table is re-used in I-LSH, regardless of the number of iterations, **h-D**$_{self}(A)$ requires only $O(P)$, where P is # of keys in a hash table. Similarly, since a set A is re-used at each iteration for a semi-cleaned set, the initial size of A is the largest needed.

Lemma 8. ***h-D***$_{self}(A)$ *has the space complexities of* $O(P)$ *for a hash table and* $O(m)$ *for a data set, respectively, where* P *is # of keys in a hash table.* ■

Unlike **h-D**$_{self}$ scheme, one can alternate *match-merge* process to make *two* output sets: one clean set A' and one dirty set M. We call this variation as **h-D**$_{alter}$. When *match*() returns $\oplus$ relationship in **h-D**$_{self}$, **h-D**$_{alter}$ scheme instead adds a merged record to a merged set M: i.e., $M \leftarrow M \cup \{merge(a_k, a_j)\}$. This avoids the direct re-feeding of the merged record to a dirty set to compare with others.

[1]Note that we use a notation $\hat{c}$, slightly different from $\bar{c}$ in Section 1.1.4 to emphasize that buckets in I-LSH are dynamically expanding or shrinking.

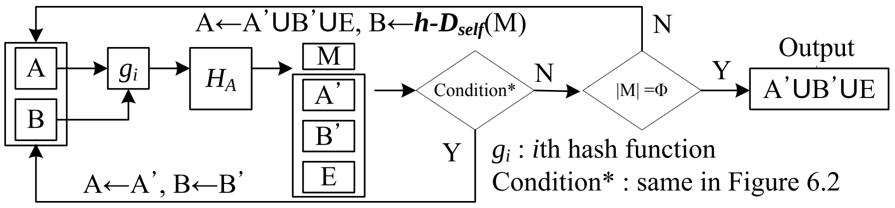

A←A'UB'UE, B←h-D_{self}(M)
N
A
B
g_i
H_A
M
A'
B'
E
Condition*
N
\|M\| =Φ
Y
Output
A'UB'UE
Y
g_i : ith hash function
Condition* : same in Figure 6.2
A←A', B←B'

Figure 6.3. The structure of **h-CC**.

In other words, in A', all records contain original information (i.e., no additional information by $\oplus$ relationship), while in M, the records have been changed by the *merge* function. We utilize this **h-D**$_{alter}$ in a suite of Hashed Linkage algorithms.

6.2 Hashed Record Linkage: HARRA

In this section, we investigate three different scenarios depending on the types of input collections, in terms of "clean" and "dirty", and present a suite of RL algorithms, HARRA (*HA*shed *R*eco*R*d link*A*ge), based on the I-LSH idea. In total, we propose six variations of Hashed Linkage. Each Hashed Linkage algorithm is denoted by the prefix "h" followed by one of three scenarios, CC for clean-clean, CD for clean-dirty, and DD for dirty-dirty cases.

6.2.1 Clean vs. Clean

As clearly mentioned in Section 5.3.1, we can exploit the relationship, such as $\not\approx, \equiv, \sqsubseteq, \sqsupseteq, \oplus$, between two records, a_i, b_j, from two clean sets. However, the iterative *match* and *merge* steps will be differently developed from **s-CC** scheme, since the iterative hashing structure is applied to the whole output records from a whole hash table or the partial records from a bucket.

Figure 6.3 illustrates the structure of **h-CC**(A, B), while Algorithm 15 shows the details. Once A is hashed to a hash table H_A, B is hashed again to H_A using the same hash function. Then, one can perform $match(a_k, b_j)$ and $merge(a_k, b_j)$ only within the same bucket of H_A, by using both matching relationship and set characteristics as shown in Algorithm 15. The remainders, A' and B', are only considered to be iteratively hashed since all merged records in M will be cleaned by **h-D**$_{self}$, and the identical records in E will be unioned without further

Algorithm 15: **h-CC**.

```
Input  : Two non-empty clean lists $A$ and $B$
Output: A non-empty clean list $C$
Make $H$ as an empty hash table;$M \leftarrow \emptyset$;
$Flag \leftarrow$ true; $i \leftarrow i+1$;
while $Flag = true$ do
    $Flag \leftarrow$ false; $j \leftarrow 1$;
    while $j \leq |A|$ do
        key $\leftarrow g_i(a_j)$; AList = H.get(key);
        AList.add($a_j$); H.add(key,AList); $j \leftarrow j+1$;
    $j \leftarrow 1$;
    while $j \leq |B|$ do
        key $\leftarrow g_i(b_j)$; AList $\leftarrow$ H.get(key); $k \leftarrow 1$;
        while $k \leq |AList|$ do
            $a_k \leftarrow AList.get(k)$;
            switch $match(a_k, b_j)$ do
                case $a_k \equiv b_j$
                    $Flag \leftarrow$ true; $E.add(a_k)$;
                    remove $a_k$ from AList; remove $b_j$ from $B$;
                    go to line 15.1;
                case $a_k \sqsupseteq b_j$
                    $Flag \leftarrow$ true; remove $b_j$ from $B$; go to line 15.1;
                case $a_k \sqsubseteq b_j$
                    $Flag \leftarrow$ true; remove $a_k$ from AList; $k \leftarrow k+1$;
                case $a_k \oplus b_j$
                    $Flag \leftarrow$ true; $M \leftarrow M \cup \{merge(a_k, b_j)\}$;
                    remove $a_k$ from AList, remove $b_j$ from $B$;
                    go to line 15.1;
                case $a_k \not\approx b_j$
                    $k \leftarrow k+1$;
15.1    $k \leftarrow 1$; $j \leftarrow j+1$;
    $A \leftarrow H.getAll(keys)$; /* put all records in H to $A$ */
    $i \leftarrow i+1$;
    if $A \neq \emptyset$ or $B \neq \emptyset$ then $Flag \leftarrow$ false;
    if termination condition is met then $Flag \leftarrow$ false;
$A \leftarrow A \cup B \cup E$;
if $|M| > 0$ then  $B \leftarrow$ h-D$_{self}(M)$; $C \leftarrow$ h-CC$(A, B)$;
return $C$;
```

processing. The records in A' and B' are hashed again to meet similar records between two clean sets. In order to optimize the memory usage for hash tables through multiple iterations, we use the same spaces of A and B repeatedly. After one iteration, for instance, records in A and B are distributed to four sets: A', B', E, and M, where E contains records that have $\equiv$ relationship with other records while M contains newly created merged records. Then, at subsequent iteration, we reset A to $A' \cup B' \cup E$ and B to **h-D**$_{self}(M)$. Note that we have to clean M first via **h-D**$_{self}$ since newly created merged records in M may again match each other within M, making M as a "dirty" collection.

Example 6. Suppose the *match*() function uses Jaccard similarity with threshold

0.5 and $merge()$ uses $\cup$. Given two clean sets, $A = \{a_1 = [1,1,1,0,0], a_2 = [0,0,1,1,1], a_3 = [1,0,0,0,1], a_4 = [1,0,0,1,0]\}$ and B=$\{b_1 = [1,0,0,0,1]$, $b_2 = [1,1,0,1,0]$, $b_3 = [0,0,0,1,1]\}$, we apply a table hash function, g_i at i-th iteration, to each record. Assuming $g_1 = \{h_{11}, h_{12}\}$ where $rp_{11} = (2,3,5,4,1)$ and $rp_{12} = (5,4,2,1,3)$, and $g_2 = \{h_{21}, h_{22}\}$ where $rp_{21} = (2,1,4,3,5)$ and $rp_{22} = (1,2,5,3,4)$, then $g_1(a_1) = \{1,3\}$, $g_1(a_2) = \{2,1\}$, $g_1(a_3) = \{3,1\}$, $g_1(a_4) = \{4,2\}$, $g_1(b_1) = \{3,1\}$, $g_1(b_2) = \{1,2\}$, $g_1(b_3) = \{3,1\}$. Due to $a_3 \equiv b_1$, we get $E = \{a_3\}$, $A' = \{a_1, a_2, a_4\}$, and $B' = \{b_2, b_3\}$. Note that a_3 in E is clean toward other records in A' and B'. At the second iteration with g_2, we use only the records in A' and B'. Thus, $g_2(a_1) = \{1,1\}$, $g_2(a_2) = \{3,3\}$, $g_2(a_4) = \{2,1\}$, $g_2(b_2) = \{1,1\}$, $g_2(b_3) = \{3,3\}$, then $match(a_1, b_2) = \oplus$ and $match(a_2,$
$b_3)$=$\sqsupseteq$. Then, $A' = \{a_2, a_4\}$, $B' = \Phi$, and $M = \{c_{1,2}\}$ where $c_{i,j} = merge(a_i \oplus b_j)$. Since B' is empty, I-LSH will stop. At the second **h-CC** call, $A = A' \cup B' \cup E = \{a_2, a_3, a_4\}$ and B =**h-D**$_{self}(M)$, and $c_{1,2}$ in a new B will eventually contain a_4. As shown, I-LSH scheme is used with recursive calls to complete all necessary comparisons to handle merged records. □

6.2.2 Clean vs. Dirty

The detailed procedure for clean-dirty case, **h-CD**, is shown in Algorithm 16 that uses **h-D**$_{alter}$ as a sub-step. In this algorithm, we consider one clean collection A and one dirty collection B. Similar to **h-CC**, first, A is put to a hash table H_A without merging records. Then, records in B are hashed to H_A. Between a_k and b_j records, five relationships are considered. The only difference from **h-CC** is the case of $\equiv$. When $a_k \equiv b_j$ holds, only b_j is removed from B and a_k proceeds to the next *match-merge* step with b_{j+1}. a_k is *not* removed since a_k may still find matching records in B since B is "dirty". Once all records in B have been scanned by H_A, a set M that contains newly created merged records is added to a dirty collection B. This new set of $B \cup M$ should be re-compared with A since there may be new matching records. This iteration will stop if no more merge occurs or other termination conditions are met.

When one iteration of *match-merge* steps finishes, we have a new clean set A and a new dirty set B. Because, in previous iteration, the relationships between

Algorithm 16: **h-CD**.

```
Input  : Non-empty $clean$ list $A$ and dirty list $B$
Output: A non-empty $clean$ list $C$
...
while $Flag = true$ do
    ...
    while $j \leq |B|$ do
        key $\leftarrow g_i(b_j)$; $k \leftarrow k + 1$; AList $\leftarrow$ H.get(key);
        while $k \leq |AList|$ do
            $a_k \leftarrow AList.get(k)$;
            switch $match(a_k, b_j)$ do
                case $a_k \equiv b_j$
                    $Flag \leftarrow$ true; remove $b_j$ from $B$;
                    go to line 16.1;
                ...
16.1    $k \leftarrow 1$; $j \leftarrow j + 1$;
    $B \leftarrow B \cup M$; $M \leftarrow \emptyset$;
    $A \leftarrow H.getAll(keys)$ /* put all records in H to $A$ */;
    if $A \neq \emptyset$ or $B \neq \emptyset$ then $Flag \leftarrow$ false;
    if termination condition is met then $Flag \leftarrow$ false;
$(B', M) \leftarrow$ h-D$_{alter}(B)$; $A \leftarrow A \cup B'$;
if $M' \not\equiv \emptyset$ then $C \leftarrow$ h-CD$(A, M)$; else $C \leftarrow A$;
return $C$;
```

A and B were fully investigated, the new sets do not have to be compared again. However, since B is still dirty, B can be cleaned by **h-D**$_{self}$. When B is cleaned by **h-D**$_{self}$, however, if merged records are generated, they should be compared again with all other records in A and B. In addition, if the information of a record does not change while B is self-cleaned, it does not have to be compared again with records in A. For this reason, in order to avoid duplicate comparisons, we use **h-D**$_{alter}(B)$ (instead of **h-D**$_{self}$) to extract "un-changed" records in B' and "merged" records in M. We simply add all records in B' to A by union operation (i.e., $A \leftarrow A \cup B'$), and call **h-CD**(A, M) recursively until no merge occurs in the **h-D**$_{alter}$ step. Finally, one clean set C will be returned.

An alternative way to handle the clean-dirty case to use **h-D**$_{self}$ and **h-CC** – i.e., clean the dirty collection A using **h-D**$_{self}$ first and apply **h-CC**. To distinguish this alternative from **h-CD** of Algorithm 16, we denote this variation as **h-CD**$_{self}$. Algebraically, the following holds: **h-CD**$_{self}(A, B) \equiv$ **h-CC**$(A,$ **h-D**$_{self}(B))$.

Note that algorithms **h-CD** and **h-CD**$_{self}$ behave differently depending on the level of "dirtiness" within B or between A and B. For instance, consider three records, $a_i \in A$ and $b_j, b_k \in B$. Suppose the following relationship occurs: $a_i \sqsupseteq merge(b_j, b_k)$. Then, using **h-CD**$_{self}$, $merge(b_j, b_k)$ will be compared again with other records in B. However, using **h-CD**, b_j and b_k will be removed, saving

Scheme	Space(HT)	Space(data size)	Time
h-CC	$O(P_A + P_B)$	$O(m + n + \|A \cap B\|)$	$O(\alpha(m + n) + 2\beta N)$
h-CD	$O(P_A + P_B)$	$O(m + n + \|A \cap B\|)$	$O(\alpha(m + n) + 2\beta N)$
h-CDself	$O(P_A + P_B)$	$O(m + n + \|A \cap B\|)$	$O(\alpha(m + n) + 2\beta N)$
h-DD1	$O(P_{A \cap B})$	$O(m + n)$	$O(\alpha(m + n) + 2\beta N)$
h-DD2	$O(P_A + P_B)$	$O(m + n + \|A \cap B\|)$	$O(\alpha(m + n) + 2\beta N)$
h-DD3	$O(P_A + P_B)$	$O(m + n + \|A \cap B\|)$	$O(\alpha(m + n) + 2\beta N)$

Table 6.1. Complexities of six Hashed Linkage algorithms, where N is time for generating a hash table at 1-st iteration, $\alpha = \sum_{i=1}^{T} \sigma_i \hat{c}_i$, $\beta = \sum_{i=1}^{T} \sigma_i$, and P_X is # of keys in a hash table for a set X.

$|B|$ number of comparisons. On the other hand, for instance, assume that $a_i \not\approx b_j$, $b_j \approx b_k$, and $a_i \not\approx b_k$. Using **h-CD**$_{self}$, there is only one comparison after $b_i \approx b_k$ is made. However, using **h-CD**, both b_j and b_k are compared to a_i before $b_j \approx b_k$ occurs, increasing the number of comparisons. In general, if the number of matches in B is significantly higher than that between A and B, **h-CD**$_{self}$ is expected to perform better.

6.2.3 Dirty vs. Dirty

Since neither collection A or B is clean, more comparisons are needed for dirty-dirty case. By using Hashed Linkage algorithms for clean-clean or clean-dirty cases, we propose three variations, referred to as **h-DD1**, **h-DD2**, and **h-DD3**.

- **h-DD1**$(A, B) \equiv$ **h-D**$_{self}(A \cup B)$
- **h-DD2**$(A, B) \equiv$ **h-CD**(**h-D**$_{self}(A), B)$
- **h-DD3**$(A, B) \equiv$ **h-CC**(**h-D**$_{self}(A)$, **h-D**$_{self}(B))$

The different behaviors of variations will be evaluated experimentally.

Lemma 9. *All* Hashed Linkage *algorithms have polynomial upper bounds in time/ space complexities, as shown in Table 6.1.* ■

6.3 Experimental Validation

Under various settings (e.g., different data distributions, varying sizes of data sets, and dirtiness), we evaluated six Hashed Linkage algorithms (**h-CC**, **h-CD**,

h-CD$_{self}$, **h-DD1**, **h-DD2**, and **h-DD3**). Two main questions to study in experiments are: (1) Is Hashed Linkage robust over various settings? (2) Does Hashed Linkage achieve high accuracy and good scalability?

6.3.1 Set-Up

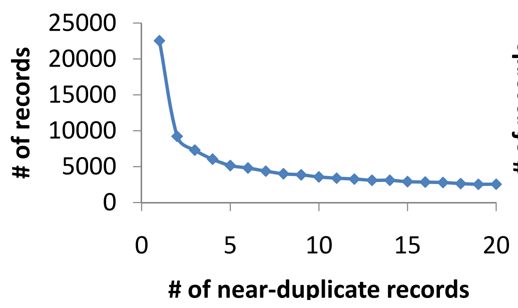
All algorithms are implemented in Java 1.6 and executed on a desktop with Intel Core 2 Quad 2.4GHz, 3.25GB RAM, and Windows XP Home. For comparison with existing RL solutions (that were optimized to run in Unix System), LION-XO PC Cluster at Penn State[2] (with dual 2.4-2.6GHz AMD Opteron Processors and 8GB RAM) was used. Note that Hashed Linkage are also run on LION-XO for comparison purpose.

Data Sets. The raw data that we used is 20 Million citations from CiteSeer whose ground truth is known to us. From this raw data, through random sampling, we generated test sets with different sizes between 10,000 and 400,000 records. As a whole, CiteSeer shows the Power-Law distribution in terms of # of matching citations. That is, while most citations have only 1-2 duplicates known, a few have more than 40 matching citations. From this raw data set, we created two types of distribution patterns – *Power-Law (PL)* and *Gaussian (GA)* distributions. Figures 6.4(a) and (b) show the corresponding distributions of # of duplicates (up to 20 duplicates) with the size of 100,000 records. For PL distribution, $f(x) = (1-x)^{\frac{1}{1-\alpha}}$ is used where $\alpha = 0.5$ and x is uniformly distributed between 0 and 1. The value of $f(x)$ is quantized to 20 bins, and the event of x is accumulated to a corresponding bin. For GA distribution, we used the mean of 11 and variance of 1. Similar to PL distribution, the Gaussian randomized values are quantized to 20 bins.

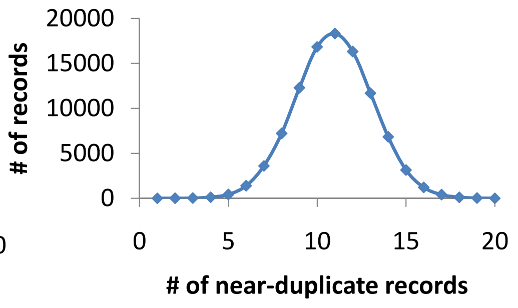
In addition to different distributions, we also used two matching rates, **IMR** and **CMR** that have been exploited in section 5.5.1. For instance, if IMR=0.5, then half of records in a collection are dirty and need to be merged. In this section, various IMR and CMR combinations are investigated to study the differences between **h-CD** and **h-CD**$_{self}$ in clean-dirty scenario, and among **h-DD1**, **h-DD2**, and **h-DD3** in dirty-dirty scenario.

Evaluation Metrics. For measuring the similarity between records, we used the

[2]http://gears.aset.psu.edu/hpc/systems/lionxo



(a) Power-Law (PL)



(b) Gaussian (GA)

Figure 6.4. Examples of record distributions in two different CiteSeer sets with 100,000 records.

Symbol	Description
HT	Hash table
RT	Running Time
PL	Power-Law distribution
GA	Gaussian distribution
K	# of indices = # of random functions = length of a key
L	# of HTs or table hash functions = # of keys per record
T	# of iterations of Hashed Linkage

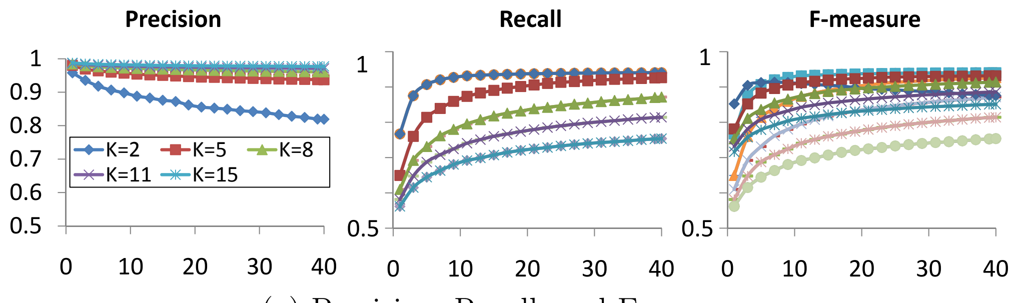
Table 6.2. Symbols used in experiments.

Jaccard similarity with the threshold, $\theta = 0.5$, by default. Two standpoints are considered as evaluation metrics: (1) **accuracy** in terms of precision $= \frac{N_{TruePositive}}{N_{AllPositive}}$, recall $= \frac{N_{TruePositive}}{N_{AllTrue}}$, and F-measure $= \frac{2 \times precision \times recall}{precision + recall}$ and (2) **scalability** in terms of *wall-clock running time* denoted by **RT** along the size of data set.

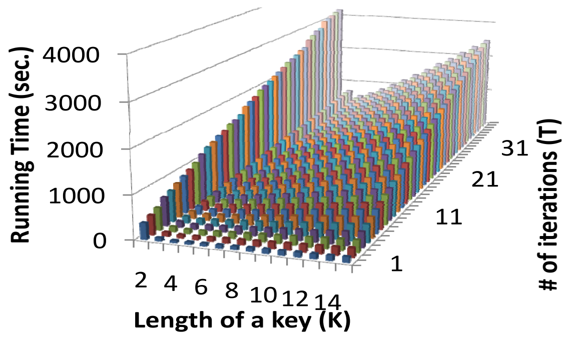
Symbols used in experiments are summarized in Table 6.2.

6.3.2 Choice of Parameters: K, L, and T

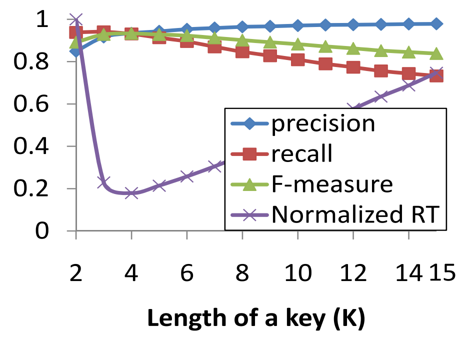
Several factors may impact the performance of Hashed Linkage algorithms – some of them (e.g., K, L, and T) are control parameters that Hashed Linkage relies on to fine tune its performance, while others (e.g., distribution pattern or dirtiness) are parameters determined by data sets. We first discuss the choices of K, L, T parameters in current Hashed Linkage implementations (and their rationale) in this section. On the other hand, the robustness of Hashed Linkage with respect to varying distribution patterns and dirtiness of data sets is validated through Sections 6.3.5–6.3.7.



(a) Precision, Recall, and F-measure



(b) RT comparison



(c) Summary ($T = 25$)

Figure 6.5. Impact of K and T using **h-D**$_{self}$ to one dirty set generated by PL distribution.

K: # of random permutation functions to generate a single hash key per record (K) plays an important role in the performance of any LSH techniques. For a closer study, we ran **h-D**$_{self}$ to a data set with 100,000 records of PL distribution. As shown in Figure 6.5(a), precision increases along K (worst at $K = 2$), but decreases along T. Asymmetrically, recall increases along T, but decreases along K. This phenomenon is expected since small K tends to increase the probability of having non-matching records located in the same bucket (mistakenly), which increases running time. On the other hand, large K may increase the chance of having matching records put into different buckets, which usually lowers recall. The F-measure in Figure 6.5(a) shows that overall we get the best precision and recall trade-off when K is around 2–5.

In terms of running time (RT) of Figure 6.5(b), clearly, RT is proportional to T. That is, if Hashed Linkage runs more iterations, its overall RT increases as well. However, the relationship between K and RT is peculiar in that when K is set very small (K=2), RT becomes longer than when $K \geq 3$. This is because with $K = 2$, each cluster (i.e., block) tends to have excessive number of irrelevant records, causing expensive pair-wise computations in subsequent stage of RL. In addition, if K is set very high ($K = 15$), RT also increases substantially since most of RT is devoted on the generation of "long" hash keys of records. As a result, in Figure 6.5(b), RT has a convex shape along K throughout varying iterations, suggesting that K should be set neither too high nor too low. Figure 6.5(c) shows the summary of precision, recall and F-measure of varying K with $T = 25$. The *normalized RT* of Figure 6.5(c) is computed by dividing all RT's by the maximum RT at $K = 2$. Based on Figure 6.5(c), therefore, we conclude that using $K = \{3, 4, 5\}$ gives the best compromised accuracy and RT overall for the given data sets. This result is also consistent with the finding of other LSH schemes in literature [5, 64] . For this reason, in subsequent experiments, we used $K = 5$. Note that in Section 6.3.4 where we compare Hashed Linkage against two other LSH based schemes, all of them use the same value: $K = 5$. Therefore, the choice of K value does *not* affect the results of Section 6.3.4 at all.

L: The behavior of conventional LSH schemes (e.g., [37, 64]) is controlled by both K and L parameters. While Hashed Linkage uses K in the same way as conventional LSH, it uses L differently. In conventional LSH, L is assumed to be # of hash tables. Then, # of keys per record is also L. However, in Hashed Linkage, one key per record is generated *per iteration* (see Figure 6.2). In order to have a fair comparison between Hashed Linkage and conventional LSH schemes in Section 6.3.4, we need to have the *same* number of keys per record. Therefore, in Hashed Linkage, we have to have at least L times of iterations to have L keys per record. As a conclusion, in Hashed Linkage, L is set dynamically, proportionate to # of iterations, T. Since all LSH based schemes in experiments have the same number of keys per record, they all show very similar accuracy, later to be shown in Figure 6.7(b).

T: # of iterations (T) has a direct impact on the running time of an iterative RL algorithm. Ideally, an RL algorithm wants to run as few times of iterations as possible while achieving the highest accuracy. In current implementations of Hashed Linkage, instead of having fixed number for T, it dynamically stops the iteration if the *reduction rate* σ_i ($= \frac{|\text{semi-cleaned set}_i|}{|\text{input set}_i|}$) at i-th iteration is less than a threshold. For instance, σ_i becomes 0.1 if 10 merged records are generated from

	StringMap	R-Swoosh	Hashed Linkage	Basic LSH	MP LSH
Input	text	XML	text	text	text
Language	C++	Java	Java	Java	Java
Model	match-only	match-merge	match-merge	match-only	match-only
Blocking	R-tree	N/A	I-LSH	LSH	MP LSH
Distance	Jaccard	Jaro	Jaccard	Jaccard	Jaccard

Table 6.3. Comparison between Hashed Linkage and four other RL solutions.

100 input records. Note that σ_i behaves differently for different data distributions (PL vs. GA). In addition, the size of the final cleaned set is different between PL and GA distributions – final cleaned set with PL distribution is usually larger than that with GA distribution. It also implies that σ_i with PL distribution tends to be smaller than that with GA distribution at each iteration. Thus, Hashed Linkage usually performs *less* number of iterations T with PL distribution. In all of subsequent experiments, we used the threshold to stop iterations is set to 0.01.

6.3.3 Comparison Against Existing RL Solutions

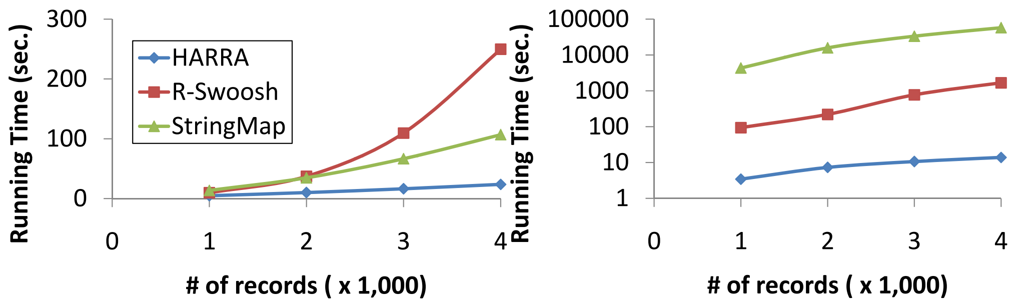
First, we chose two well-known RL algorithms[3], StringMap[4] [48] and R-Swoosh[5] [10], and compared them against one of Hashed Linkage algorithms, **h-D**$_{self}$, for the case of cleaning one dirty set. The first three columns of Table 6.3 show the differences among StringMap, R-Swoosh, and Hashed Linkage in detail. Since StringMap supports only the match-only[6] RL model, comparison was done under the match-only RL model (i.e., no merges). Since both StringMap and R-Swoosh could not finish larger data sets within a reasonable time, data sets with 1,000 – 4,000 records were mainly used for comparison. Two record types were used: short records (e.g., people names of 10-20 characters) are from StringMap package while long records (e.g., citations of 100-200 characters) are made from CiteSeer data set. Since short records from StringMap did not have a ground truth, we estimated one by running naive pair-wise comparison first.

[3]We also considered Febrl [21] for the comparison. However, since Febrl does not provide means to measure running time and its installation was problematic due to a version conflict, we omitted it.

[4]http://flamingo.ics.uci.edu/releases/2.0.1/

[5]http://infolab.stanford.edu/serf/

[6][48] actually supports merge() operation. However, the merged record does not incur new comparison in subsequent iterations.



(a) short record (e.g., names) (b) long record (e.g., citations)

Figure 6.6. RT comparison among **h-D**$_{self}$, StringMap, and R-Swoosh. (Note that Y-axis of (b) is on logarithmic scale.)

As shown in Figure 6.6(a), with simple record contents such as people names, all three algorithms run well within a reasonable time. Both Hashed Linkage and StringMap show linear increase along the size of records thanks to the blocking step, while R-Swoosh shows a quadratic increase due to nested-loop style comparisons. With an efficient blocking via I-LSH, Hashed Linkage provides the best running time among all for all size ranges. The StringMap which also employs blocking was slow (but still faster than R-Swoosh) since it spent majority of time in generating R-tree based structure as part of blocking. As we demonstrated in Figure 1.2, if both StringMap and R-Swoosh could have been able to run for larger data sets such as 400,000 records, the gap between Hashed Linkage and them would have widened further. For the second type of a long record set, as shown in Figure 6.6(b), StringMap works worse than R-Swoosh. With initial data set of 1,000 citations, RT of Hashed Linkage is only 3.428 sec., for instance, while RT of StringMap is 4,318 sec. R-Swoosh also shows its quadratic nature in running time. Thus, with larger number of records, R-Swoosh becomes impractical.

Next, in terms of precision and recall trade-off, R-Swoosh is the best, as Table 6.4 shows. The precision values for all 3 methods are all equal as 1.0. Recall that the ground truth of this experiment in Section 6.3.3 was "estimated" by running a blockbox match function since data set (i.e., short and long records) from StringMap package did not provide one. Then, for a fair comparison, all 3 methods used more or less the same blackbox match step but different blocking strategies. Therefore, all 3 have the identical precision of "1" but varying recall, since some

Data size	1,000		2,000		3,000		4,000	
	precision	recall	precision	recall	precision	recall	precision	recall
Hashed Linkage	1	0.998	1	0.996	1	0.992	1	0.985
StringMap	1	0.999	1	1	1	1	1	1
R-Swoosh	1	1	1	1	1	1	1	1

Table 6.4. Precision and recall comparison among **h-D**$_{self}$, StringMap, and R-Swoosh.

blocking may miss true positives[7]. Since R-Swoosh essentially does all pair-wise comparisons, it does not miss any matching record pairs and yields perfect recall all the time. For both Hashed Linkage and StringMap, due to the blocking stage, some false dismissals may occur. Therefore, for four data sets of {1,000, 2,000, 3,000, 4,000} records, StringMap got recalls of {0.999, 1, 1, 1} while Hashed Linkage got {0.998, 0.996, 0.992, 0.985}.

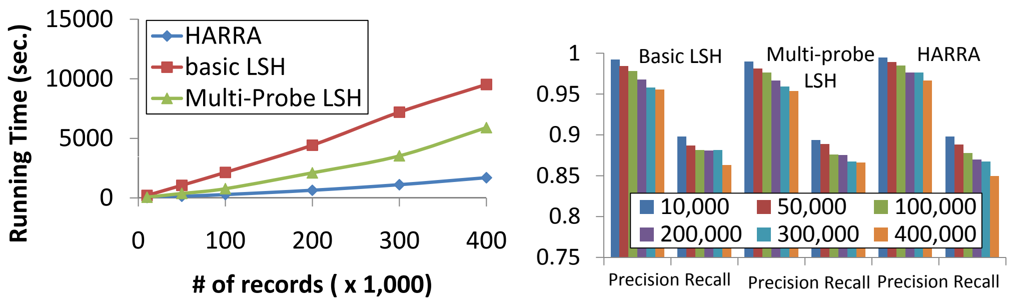
Overall we claim that Hashed Linkage *runs much faster with negligible loss of recall.* For instance, Hashed Linkage runs 4.5 and 10.5 times faster than StringMap and R-Swoosh with 4,000 short name record data set while missing only 1.5% of true matches out of 4,000 $\times$ 4,000 record pairs.

6.3.4 Comparison Against Existing LSH Based RL Solutions

Next, we compare **h-D**$_{self}$ against both basic LSH [37] and multi-probe LSH [64] algorithms for the case of cleaning one dirty set, made from CiteSeer data. Unlike StringMap and R-Swoosh, now, both basic LSH and multi-probe LSH algorithms are capable of handling large data sets such as 400,000 records in multiple iterations. Therefore, this time, the comparison was done for all ranges of data sets. The last three columns of Table 6.3 show the differences among Hashed Linkage, basic LSH, and multi-probe LSH in detail.

Figure 6.7(a) shows the comparison of RT among three approaches with varying size of input records, while Figure 6.7(b) shows the trend of precision and recall which are managed to be similar for the proper RT comparison. Since Hashed Linkage uses one re-usable hash table, the memory usage of Hashed Linkage is significantly lower than regular LSH techniques. In fact, the memory requirement

[7]However, for subsequent experiments with data sets from CiteSeer, which already provides a ground truth, we have varying precision scores depending on the choice of match functions.



(a) RT comparison (b) Accuracy comparison

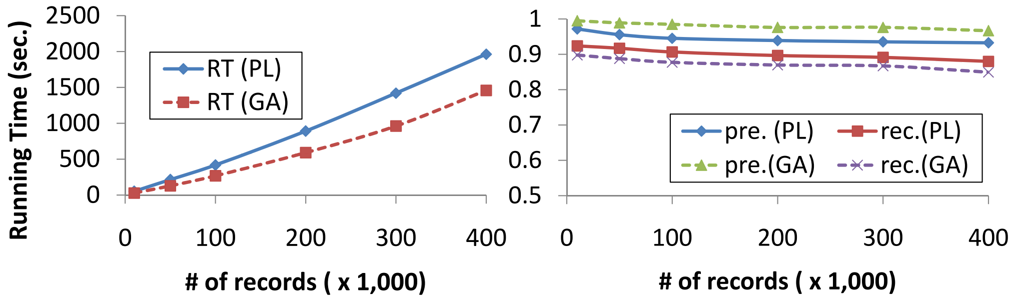
Figure 6.7. Comparison, **h-D**$_{self}$ to others with Gaussian distribution, $\sigma = 1$ and $mean = 11$.

in both basic LSH and multi-probe LSH depends on the number of hash tables used in the systems, while Hashed Linkage uses *one* dynamic reusable hash table. In addition, with respect to running time for 400,000 records (i.e., the largest test set), Hashed Linkage runs 5.6 and 3.4 times faster than basic LSH and multi-probe LSH, respectively while maintaining similar precision and recall levels. This is because the decrease of the total size of a set will affect the hash table generation time at each iteration in Hashed Linkage system, while all records are used to build hash tables in both basic LSH and multi-probe LSH. Furthermore, even though multi-probe LSH provides better results than basic LSH, it may not avoid the nature of quadratic number of comparisons, since input records should be compared with many records from multiple buckets in a hash table. Therefore, the processing time of multi-probe LSH is not improved enough for the case of cleaning sets.

Since the comparison of remaining Hashed Linkage algorithms against the basic and multi-probe algorithms shows similar pattern, in subsequent sections, we focus on comparing various aspects among Hashed Linkage algorithms in detail.

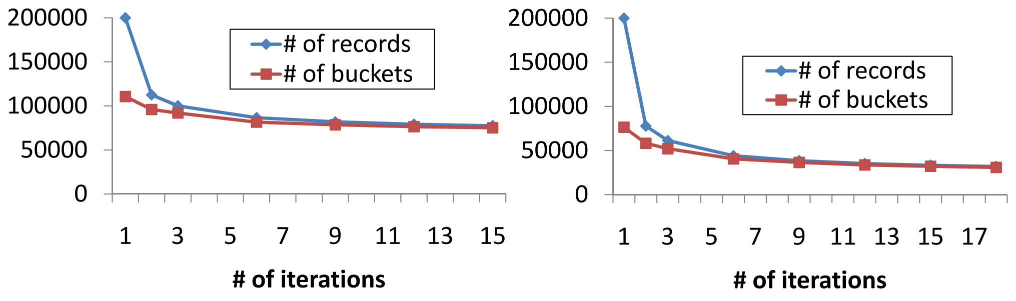
6.3.5 Cleaning Single Data Collection

Next, the **h-D**$_{self}$ algorithm is closely evaluated with two data distributions – Power-Law (PL) and Gaussian (GA). As shown in Figure 6.8, **h-D**$_{self}$ works efficiently and robustly for both data sets. As the size of a dirty data set increases, both precision and recall values decrease slightly. However, the running time to clean a dirty data set increases linearly. This suggest that **h-D**$_{self}$ is scalable to the size of a data set regardless of statistical distributions.



(a) Running Time (b) Precision & Recall

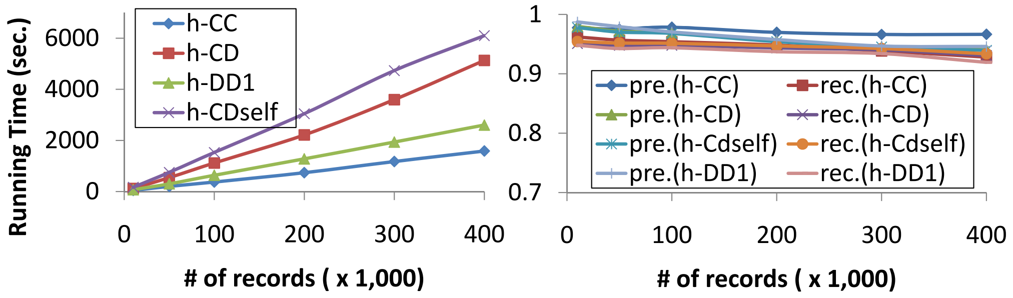
Figure 6.8. h-D$_{self}$ with Power-Law (PL) and Gaussian (GA) distributions



(a) Gaussian distributed set (b) Power-Law distributed set

Figure 6.9. # of records & # of buckets in a hash table at each iteration with 200,000 records.

The number of iterations can be pre-defined as a constant by investigating the number of records in a semi-clean set at each iteration. As mentioned earlier, the reduction rate between the size of input set and that of output set can be also used as a control factor to stop iterations. In Figure 6.9, both # of records in a semi-clean set and # of buckets (= # of keys = the size of a hash table) from records in a semi-clean set are depicted with the number of iterations. As can be seen in Figures 6.9 (a) and (b), if reduction rate is used as a control factor, the number of iterations in GA distributed set is greater than that in PL distributed set. This happens because the size of the final clean set in GA distributed set is smaller than that in PL distributed set. In other words, the dirtiness of GA distributed set is greater than that of PL distributed set in terms of the number of matched records. At each iteration, both # of records in a semi-clean set and # of buckets in a hash table decrease as expected. Note that the size of a hash table is always smaller than the number of records due to duplicate keys from different records.



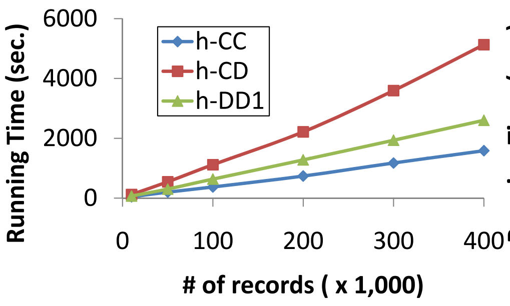
(a) Running time (b) Accuracy

Figure 6.10. All algorithms for different scenarios.

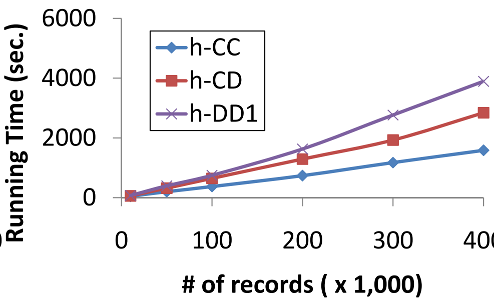
6.3.6 Cleaning Pairs of Data Collections

In this section, we validate all Hashed Linkage algorithms with test sets with different characteristics. When one knows if a set is "clean" or "dirty" beforehand, one can exploit such characteristics to reduce running time. In addition, the relationship between records can enhance the running time further. On the other hand, when one does not know if a set is "clean" or "dirty" beforehand, even though the set is clean, algorithms should use "dirty" characteristics.

1. With Known Characteristics. Figure 6.10 shows the scalability and accuracy of four Hashed Linkage algorithms for clean-clean, clean-dirty, and dirty-dirty scenarios. Virtually, all algorithms show similar patterns. As to scalability, as shown in Figure 6.10(a), all four algorithms show that running time increases linearly as input data size increases, suggesting all algorithms are scalable with respect to their input size. Note that **h-CC** is the fastest while **h-DD1** is the next. **h-CC** uses the clean-clean characteristics so that records in hash table from one set does not have to be compared. Thus, we can save time by reducing the number of comparisons by an expensive $match()$ function. In addition, the iteration will stop sooner than others when the same reduction rate is set for all algorithms. For dirty-dirty sets, because Hashed Linkage uses a semi-cleaned data set by merging records that hit in a hash table at each iteration, the size of hash table will be reduced drastically along the number of iteration, even though more iterations are requested. This results in the reduced total running time in **h-DD1**. Between **h-CD** and **h-CD**$_{self}$, we apply both algorithms to the same data sets. Because the effect of CMR is more forceful than that of IMR, the effect of merging between



(a) Known characteristic



(b) Unknown characteristic

Figure 6.11. Comparison among **h-CC**, **h-CD**, and **h-DD1** with 10,000 to 400,000 records. (Note that running time between (a) and (b) are *not* comparable.)

two sets is higher than that of self-merging. Thus, **h-CD** shows better running time than **h-CD**$_{self}$. The detailed comparison between **h-CD** and **h-CD**$_{self}$ will be more investigated in subsequent sections. As to accuracy, as shown in Figure 6.10(b), all four algorithms again achieve similar precision and recall, ranging from 0.91 to 1.

2. With Unknown Characteristics. Three different algorithms from three different scenarios are compared by putting 10,000 to 400,000 records with IMR=0.0 and CMR=0.5. Although data is a clean-clean case (i.e., IMR=0.0), algorithms for clean-dirty or dirty-dirty cases pretend *not* to know that they are clean so that comparison is possible.

The running times among **h-CC**, **h-CD**, and **h-DD1** in Figure 6.11(b) appear "faster" than those shown in Figure 6.11(a), contrary to the intuition. This is because two cases use different data set. For Figure 6.11(b) with unknown characteristics, single data set is used to exploit clean characteristic for all algorithms when it is known to user. In **h-CC**, one does not need to compare records internally in each input set. In other words, at each hash-match-merge iteration, after hash step, we do not perform *match-merge* steps within one set. In addition, we can save more time using E set that only exists in **h-CC** algorithm by using clean characteristic on both input sets. However, **h-CD** uses the clean characteristic on one side only. Thus, the algorithm requires the steps to clean the other set. Therefore, **h-CD** demands more processing time. Similarly, **h-DD1** considers that all sets are dirty, i.e., the clean property is not used at all. We may require more processing time to clean both sets. For proper comparison, the same number

	IMR	0.0						IMR	0.4					
	CMR	0.0	0.2	0.4	0.6	0.8	1.0	CMR	0.0	0.2	0.4	0.6	0.8	1.0
Precision	h-CD	.98	.98	.98	.98	.98	.98	h-CD	.98	.98	.98	.98	.98	.99
	h-CD$_{self}$	.98	.98	.98	.98	.98	.98	h-CD$_{self}$	.98	.98	.98	.98	.98	.98
	h-DD1	.96	.96	.97	.97	.97	.97	h-DD1	.97	.97	.98	.98	.98	.98
	h-DD2	.97	.97	.98	.98	.98	.98	h-DD2	.98	.98	.98	.98	.98	.98
	h-DD3	.97	.97	.97	.98	.98	.98	h-DD3	.99	.98	.96	.95	.93	.93
Recall	h-CD	1	.98	.96	.94	.92	.90	h-CD	.96	.95	.94	.93	.92	.90
	h-CD$_{self}$	1	.98	.96	.94	.92	.90	h-CD$_{self}$	.96	.95	.94	.93	.93	.90
	h-DD1	1	.99	.97	.96	.94	.92	h-DD1	.95	.94	.93	.92	.92	.90
	h-DD2	1	.98	.96	.94	.91	.89	h-DD2	.99	.96	.96	.94	.90	.88
	h-DD3	1	.98	.95	.93	.90	.88	h-DD3	.95	.90	.88	.85	.82	.80

Table 6.5. Precision and recall under various dirtiness

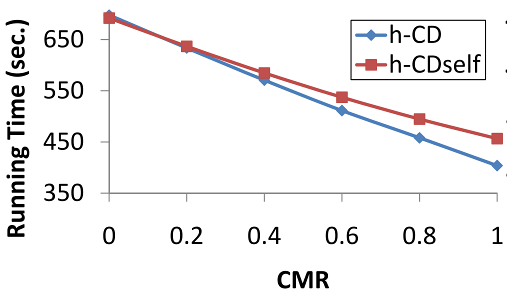
of keys are generated for one record in all algorithms by setting the number of iterations.

6.3.7 Robustness of HARRA

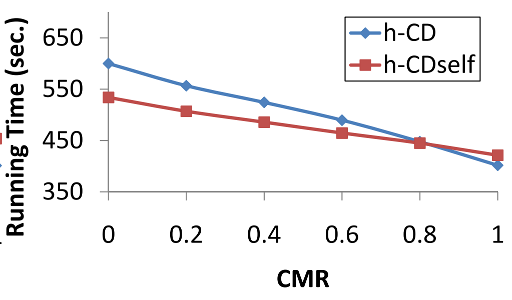
In section 6.3.5, we already showed the robustness of **h-D**$_{self}$ with different statistical distribution of data sets. In this section, we also show the robustness of Hashed Linkage against the varying dirtiness of data sets. Within the same scenario, we compare different approaches to clean data sets with various setting of reduction rate. For clean-dirty case, we investigate the difference between **h-CD** and **h-CD**$_{self}$ by comparing running time with various IMR and CMR. For dirty-dirty case, we also compare three different algorithms of **h-DD1**, **h-DD2**, and **h-DD3** with various IMR and CMR. All data sets include 100,000 records.

First, Table 6.5 shows the details of both precision and recall with different combinations of IMR and CMR values. Note that regardless of the chosen combination, accuracy of Hashed Linkage is robust – always both precision and recall are above 0.9 except a few very dirty cases. For this experiment, we use the same number of keys per record.

1. Clean-Dirty Case. Many data sets are generated by 2 variations of IMR (0.0 and 0.4) and 6 variations of CMR (0.0, 0.2, 0,4, 0.6, 0.8, and 1.0) to show the behavior between **h-CD** and **h-CD**$_{self}$ with 100,000 records. Figure 6.12(a) shows the running time when IMR is 0.0 (i.e., both set are actually clean). As shown in Figure, with CMR=0.0, **h-CD** and **h-CD**$_{self}$ show very similar running time,



(a) $IMR = 0.0$



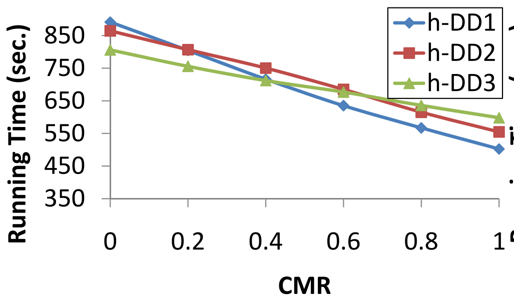
(b) $IMR = 0.4$

Figure 6.12. Algorithm comparison in clean-dirty case by varying IMR and CMR with 100,000 records.

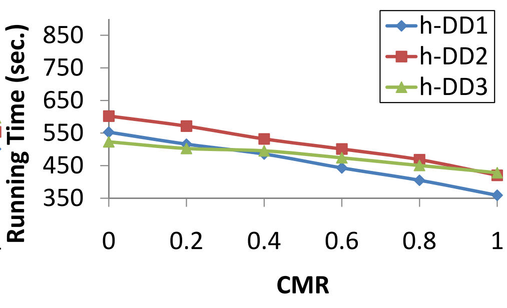
because **h-CD** compares two sets A and B directly at first, then cleans B, and **h-CD**$_{self}$ cleans B first, then compare A and B. Thus, little difference exists in terms of the number of comparisons between two schemes. By increasing CMR, **h-CD** has more merits than **h-CD**$_{self}$. By comparing A and B first in **h-CD**, $|B'|$ (the remainder of B) is reduced. Later, we can save more time to clean B'. In **h-CD**$_{self}$, because IMR is 0.0, the process to clean B does not change the total number of records in input sets at the first step. Therefore, **h-CD** always shows better running time with IMR=0.0 with various CMR.

Figure 6.12(b) shows the running time when IMR is 0.4 with 6 CMR (0.0, 0.2, 0,4, 0.6, 0.8, and 1.0). Note that both IMR and CMR are applied to a set B. Overall, in terms of running time, **h-CD**$_{self}$ is better with small CMR, but **h-CD** is with larger CMR. For example, with CMR=0.0, **h-CD**$_{self}$ runs faster than **h-CD** does. When we apply **h-D**$_{self}$ to B, the size of B reduces at the first iteration. Thus, we can save running time when B is compared with A at the next iteration. However, in **h-CD**, all records in B are compared to A at the first iteration. Thus, we will spend more time in merging two sets. With CMR=1, **h-CD** runs faster than **h-CD**$_{self}$ does, because the effect of CMR is much higher than that of IMR. Therefore, **h-CD** is preferable beyond CMR=0.8.

Between Figures 6.12(a) and (b), we compare the effect of IMR on **h-CD** and **h-CD**$_{self}$. The running time is more sensitive along CMR when we have low IMR. It means that the effect of CMR is more significant when IMR is low. The total running time with higher IMR is faster at the same CMR point, because the size of a final clean set is smaller than that with lower IMR. This fact implies that the



(a) $IMR = 0.0$



(b) $IMR = 0.4$

Figure 6.13. Algorithm comparison in dirty-dirty case by varying IMR and CMR with 100,000 records.

number of records at each iteration is reduced more with higher IMR. Similarly, with higher CMR, we have faster running time because of the same reason in the effect of higher IMR.

2. Dirty-Dirty Case. For dirty-dirty case, we compare the running time among **h-DD1**, **h-DD2**, and **h-DD3** with 2 IMR (0.0 and 0.4) and 6 CMR (0.0, 0.2, 0,4, 0.6, 0.8, and 1.0) with 100,000 records. As shown in Figure 6.13(a), with IMR=0.0, both **h-DD2** and **h-DD3** show similar patterns of running time for low CMR, while **h-DD1** has steeper slope overall (i.e., more sensitive to the change of CMR). Within the structure of **h-DD1**, A and B are compared at the same iteration as cleaning A and B. However, in **h-DD2**, comparing A and B is followed by applying **h-D**$_{self}$ to B, and in **h-DD3**, we apply **h-D**$_{self}(A)$ and **h-D**$_{self}(B)$ at first before comparing A and B. With CRM=0.6 or higher, the running time in **h-DD1** is the best by the effect of CMR, and **h-DD2** and **h-DD3** are followed in order.

In Figure 6.13(b), with IMR=0.4, **h-DD3** shows the fastest running time with low CMR. However, **h-DD1** provides the best running time with CMR=0.4 or higher, because the effect of CMR is higher than that of IMR, Between Figures 6.13(a) and (b), similar to the pattern in clean-dirty case, with IMR=0.0, the running time is more sensitive to CMR. Thus, with IMR=0.0, the slopes on all algorithms are steeper than those with IMR=0.4. However, overall running time with IMR=0.4 is much faster at the same CMR on all algorithms, since we have higher reduction rate of the number of records at each iteration with higher IMR. Similarly, with higher CMR, we will have faster running time.

6.4 Summary

In this chapter, we studied Hashed Linkage by investigating iterative structures of LSH algorithms to clean and merge data sets. In general, iterative nature of a record linkage problem occurs immense number of computations for large-scale data collections. For three input cases of clean-clean, dirty-clean, and dirty-dirty for known data characteristics, we presented six solutions through Hashed Linkage variations. Our proposed algorithms are shown to exhibit fast running time as well as scalability along data size. In addition, compared to four competing record linkage solutions (StringMap, R-Swoosh, basic and multi-probe LSH), Hashed Linkage shows 3 – 10 times of improvements in speed with equivalent or comparable accuracy. The significant saving is due to the dynamic and re-usable hash table and exploitation of data characteristics in Hashed Linkage.

Chapter 7

Conclusion

7.1 Contribution

In this thesis, we studied high performance record linkages in various aspects. In order to enhance the performance of record linkage solutions, we proposed four different applications for the specific or general usages.

First, in Image Linkage, we introduced a novel solution, named as BLASTed Image Linkage (BASIL), to find near-duplicate images by bridging two different areas, i.e., Multimedia and Biology. We exploit the biological gene sequence matching system, BLAST, that presents a famous and well-developed tool in biology, to the NDID problem. Image Linkage solution promises that NDID problem can be solved in BLAST by proper translating image features into gene sequences through the proposed CC table. The CC table combines various homogeneous or heterogenous features to generate one gene sequence per image. In addition, by the nature of the CC table, more features can be added to it without additional efforts. Lately, by adjusting scoring matrices in BLAST, the accuracy of similarity between images was improved further.

Second, in Video Linkage, we studied the hierarchical structure of linkage algorithms that exploit the video structure considered as a sequence of shots, and in turn a shot as a sequence of frames. The Video Linkage solution identified the copied videos containing various distortions. To enhance the performance of Video Linkage, the temporal order of frames/shots are exploited in similarity measures. In addition, we developed pipe-lined filtering structures to avoid unnecessary exhaustive measures using partial inequality between proposed algorithms (greedy, approximate, and exact solutions) – i.e., filter-out avoidable comparisons through faster but loose Video Linkage measures.

Third, in Parallel Linkage, we developed the parallel structure of linkage solutions, when the record linkage has *match* and *merge* behaviors. For this reason, Parallel Linkage solutions are *not* applied simply to images or videos (no *merge* steps on such data for now). Instead, we use citation information as our input set. Due to the iterative nature of *match-merge* based record linkage problems, the merged records have been re-fed to the linkage structure to generate the final clean set. The input data characteristics are also exploited to remove unnecessary comparisons. With our parallel structure, we could optimize the solution in terms of **speedup** as well as **efficiency** by minimizing overhead such as processor idling time and message passing time. In addition, our proposed Parallel Linkage structure can adopt any types of *match* and *merge* functions.

Fourth, in Hashed Linkage, we studied the iterative hashing structure that is most appropriate to the high dimensional non-numerical records such as citation information. For *match-merge* based record linkage problem, the iterative hashing structure of LSH-based idea was well developed to outperform the conventional RL as well as other LSH-based hashing techniques, in terms of the processing time and space usage, while keeping similar high accuracy. Like Parallel Linkage, Hashed Linkage algorithms exploit the characteristics of input data sets such as *clean-clean*, *clean-dirty*, and *dirty-dirty*, and can employ any *match* and *merge* functions.

7.2 Research Plan

Many directions are ahead for future work. In order to achieve high performance in record linkage solutions, we can expand current our proposed algorithms with various aspects.

First, with Image Linkage solutions, we plan to extend BASIL to apply it to different mediums such as video, audio, or time series. In addition, the structural characteristics of multi-media inputs will be studied to achieve structural alignment matching algorithms.

Second, since the accuracy of CVD solution also depends on specific video features such as key points, we can improve the accuracy of current Video Linkage by investigating more persistent features. Video indexing technique will be another direction to classify and store approximate similar videos in the same group, so that CVD processing time will be improved further.

Third, we plan to extend the current nested-loop style linkage solutions to support *blocking* as in blocked nested-loop style in the parallel structure. This enables linkage solutions to quickly filter out unnecessary records so that only small number of candidate records are further examined. Further, we can reduce the overhead occurred by massage passing time between processors when records in the same block are shipped to one or a few processors.

Fourth, we plan to extend all Hashed Linkage algorithms with Parallel Linkage structure with corresponding characterized data sets, so that the overhead in the parallel structure will be reduced drastically. The proposed LSH based hashing function are only for non-numerical high dimensional records. However, we can exploit the Hashed Linkage structure with other hashing functions for other data domains (e.g., multimedia or web documents). For instance, by implementing video indexing technique using the proposed Hashed Linkage structure, we can use Video Linkage algorithms as the *match* function in our future work. We can also expand Hashed Linkage algorithms to other data problems (e.g., clustering or classification).

Bibliography

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool. *J. Mol. Biology*, 215(3):403–410, 1990.

[2] A. Andoni and P. Indyk. "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions". In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 459–468, 2006.

[3] Alexandr Andoni and Piotr Indyk. "Efficient Algorithms for Substring Near Neighbor Problem". In *Proceedings of the Symposium on Discrete Algorithyms (SODA)*, pages 1203–1212, 2006.

[4] A. Arasu, V. Ganti, and R. Kaushik. Efficient Exact Set-Similarity Joins. 2006.

[5] Vassilis Athitsos, Michalis Potamias, Panagiotis Papapetrou, and George Kollios. "Nearest Neighbor Retrieval Using Distance-Based Hashing". In *IEEE International Conference on Data Engineering (ICDE)*, pages 327–336, 2008.

[6] S. Baluja, M. Covell, and S. Ioffe. "Permutation Grouping: Intelligent Hash Function Design for Audio & Image Retrieval". In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2008.

[7] R. Barzilay and L. Lee. Bootstrapping Lexical Choice via Multiple-Sequence Alignment. In *Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, 2002.

[8] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling Up All Pairs Similarity Search. 2007.

[9] Sevinc Bayram, Husrev Taha Sencar, and Nasir Memon. Video copy detection based on source device characteristics: a complementary approach to

content-based methods. In *ACM MIR*, pages 435–442, Vancouver, Canada, October 2008.

[10] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: A generic approach to entity resolution. *VLDB J.*, 18(1):255–276, January 2009.

[11] O. Benjelloun, H. Garcia-Molina, Q. Su, and J. Widom. "Swoosh: A Generic Approach to Entity Resolution". Technical report, Stanford University, 2005.

[12] O. Benjelloun et al. "D-Swoosh: A Family of Algorithms for Generic, Distributed Entity Resolution". Technical report, Stanford University, 2006.

[13] M. Bilenko, R. Mooney, W. Cohen, P. Ravikumar, and S. Fienberg. "Adaptive Name-Matching in Information Integration". 18(5):16–23, 2003.

[14] Mikhail Bilenko, Beena Kamath, and Raymond J. Mooney. Adaptive Blocking: Learning to Scale Up Record Linkag. In *ICDM*, December 2009.

[15] Paul Browne. Video information retrieval using objects and ostensive relevance feedback. *SIGIR Forum*, 39(1):54–54, 2005.

[16] Deng Cai, Xiaofei He, and Jiawei Han. Spectral Regression: A Unified Subspace Learning Framework for Content-Based Image Retrieval. 2007.

[17] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. "Robust and Efficient Fuzzy Match for Online Data Cleaning". 2003.

[18] S. Chaudhuri, V. Ganti, and R. Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. 2006.

[19] Xiangang Cheng, Yiqun Hu, and Liang-Tien Chia. Image Near-Duplicate Retrieval using Local Dependencies in Spatial-Scale Space. 2008.

[20] Chih-Yi Chiu, Chu-Song Chen, and Lee-Feng Chien. A framework for handling spatiotemporal variations in video copy detection. In *IEEE TCSVT*, volume 18, pages 412–417, 2008.

[21] P. Christen. Febrl: an open source data cleaning, deduplication and record linkage system with a graphical user interface. pages 1065–1068, August 2008.

[22] P. Christen, T. Churches, and M. Hegland. "A Parallel Open Source Data Linkage System". In *Springer Lecture Notes in Artificial Intelligence*, Sydney, Australia, May 2004.

[23] R. Cilibrasi and P. M. B. Vitanyi. “The Google Similarity Distance”. (3):370–383, 2007.

[24] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman, and C. Yang. Finding Interesting Sssociations Without Support Pruning. volume 13, pages 64–78, Jan/Feb 2001.

[25] Daniel DeMenthon and David Doermann. Video retrieval using spatio-temporal descriptors. In *ACM MULTIMEDIA*, pages 508–517, Berkeley, CA, USA, November 2003.

[26] Wei Dong, Zhe Wang, Moses Charikar, and Kai Li. Efficiently matching sets of features with random histograms. pages 179–188, 2008.

[27] X. Dong, A. Y. Halevy, and J. Madhavan. “Reference Reconciliation in Complex Information Spaces”. 2005.

[28] Berna Erol, Jonathan J. Hull, and Dar-Shyang Lee. Linking Multimedia Presentations with Their Symbolic Source Documents: Algorithm and Applications. pages 498–507, 2003.

[29] Fabrizio Falchi, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fausto Rabitti. Caching content-based queries for robust and efficient image retrieval. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 780–790, 2009.

[30] I. P. Fellegi and A. B. Sunter. “A Theory for Record Linkage”. *J. of the American Statistical Society*, 64:1183–1210, 1969.

[31] Shaolei Feng and R. Manmatha. A discrete direct retrieval model for image and video retrieval. In *ACM CIVR*, pages 427–436, Niagara Falls, Canada, July 2008.

[32] Jun Jie Foo and Ranjan Sinha. Pruning SIFT for scalable near-duplicate image matching. In *ADC '07: Proceedings of the eighteenth conference on Australasian database*, pages 63–71, 2007.

[33] Jun Jie Foo, Justin Zobel, and Ranjan Sinha. Clustering near-duplicate images in large collections. In *MIR '07: Proceedings of the international workshop on Workshop on multimedia information retrieval*, pages 21–30, 2007.

[34] Jun Jie Foo, Justin Zobel, Ranjan Sinha, and S. M. M. Tahaghoghi. Detection of Near-Duplicate Images for Web Search. In *Int'l Conf. on Image and Video Retrieval (CIVR)*, 2007.

[35] Yuli Gao and Jianping Fan. Semantic Image Classification with Hierarchical Feature Subset Selection. In *ACM SIGMM Int'l workshop on Multimedia information retrieval (MIR)*, pages 135–142, 2005.

[36] Ciocca Gianluigi and Schettini Raimondo. "An Innovative Algorithm for Key frame Extraction in Video Summarization". *J Real-Time Image Proc*, 1:69–88, 2006.

[37] Aristides Gionis, Piotr Indyky, and Rajeev Motwaniz. "Similarity Search in High Dimensions via Hashing". In *vldb*, pages 518–529, 1999.

[38] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *"Introduction to Parallel Computing (2nd Edition)"*. Addison Wesley, 2003.

[39] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. "Text Joins in an RDBMS for Web Data Integration". 2003.

[40] S. Guha, N. Koudas, A. Marathe, and D. Srivastava. "Merging the Results of Approximate Match Operations". pages 636–647, 2004.

[41] Arun Hampapur, Ki-Ho Hyun, and Ruud Bolle. "Comparison of Sequence Matching Techniques for Video Copy Detection". In *SPIE Storage and Retrieval for Media Databases*, San Jose, CA, Jan 2002.

[42] Alexander Haubold and Apostol Natsev. Web-based information content and its application to concept-based video retrieval. In *ACM CIVR*, pages 437–446, Niagara Falls, Canada, July 2008.

[43] Xiaofei He, Deng Cai, Ji-Rong Wen, Wei-Ying Ma, and Hong-Jiang Zhang. Clustering and Searching WWW Images using Link and Page Layout Analysis. *ACM Trans. Multimedia Comput. Commun. Appl.*, 3(2):10, 2007.

[44] M. A. Hernandez and S. J. Stolfo. "The Merge/Purge Problem for Large Databases". 1995.

[45] Peter Howarth and Stefan M. Rüger. Evaluation of Texture Features for Content-Based Image Retrieval. In *ACM Int'l Conf. on Image and Video Retrieval (CIVR)*, July 2004.

[46] I.-A. Huang, J.-M. Ho, H.-Y. Kao, and S.-H. Lin. Extracting Citation Metadata from Online Publication Lists Using BLAST. 2004.

[47] A. Jain and A. Vailaya. Image Retrieval using Color and Shape. *Pattern Recognition*, 29(8):1233–1244, 1997.

[48] L. Jin, C. Li, and S. Mehrotra. Supporting Efficient Record Linkage for Large Data Sets Using Mapping Techniques. *World Wide Web J.*, 9(4):557–584, December 2006.

[49] Alexis Joly, Olivier Buisson, and Carl Frélicot. Statistical similarity search applied to content-based video copy detection. In *IEEE ICDE*, pages 1285–1295, Toyko, Japan, April 2005.

[50] Alexis Joly, Olivier Buisson, and Carl Frelicot. Content-based copy retrieval using distortion-based probabilistic similarity search. In *IEEE TMM*, volume 9, pages 293–306, 2007.

[51] D. V. Kalashnikov, S. Mehrotra, and Z. Chen. "Exploiting Relationships for Domain-independent Data Cleaning". In *SIAM Data Mining (SDM) Conf.*, 2005.

[52] Hideki Kawai, Hector Garcia-Molina, Omar Benjelloun, David Menestrina, Euijong Whang, and Heng Gong. "P-Swoosh: Parallel Algorithm for Generic Entity Resolution". Technical report, Stanford University, 2006.

[53] Yan Ke, Rahul Sukthankar, and Larry Huston. An Efficient Parts-based Near-Duplicate and Sub-Image Retrieval System. 2004.

[54] R. P. Kelley. "Blocking Considerations for Record Linkage Under Conditions of Uncertainty". In *Proc. of Social Statistics Section*, pages 602–605, 1984.

[55] Changick Kim and B. Vasudev. Spatiotemporal sequence matching for efficient video copy detection. In *IEEE TCSVT*, volume 15, pages 127–132, 2005.

[56] Changick Kim and Bhaskaran Vasudev. "Spatiotemporal Sequence Matching for Efficient Video Copy Detection". *IEEE TCSVT*, 15(1):127–132, 2005.

[57] H. Kim, Jeongkyu Lee, Haibin Liu, and Dongwon Lee. Video Linkage: Group Based Copied Video Detection. In *ACM Int'l Conf. on Image and Video Retrieval (CIVR)*, Jul 2008.

[58] Joosub Kim and Jeho Nam. Content-based video copy detection using spatio-temporal compact feature. In *IEEE ICACT*, volume 3, pages 1667–1671, Phoenix Park, Korea, February 2009.

[59] M. Krauthammer, A. Rzhetsky, P. Morozov, and C. Friedman. Using BLAST for Identifying Gene and Protein names in Journal Articles. *Gene*, 259(1-2):245–252, 2000.

[60] Julien Law-To, Olivier Buisson, Valérie Gouet-Brunet, and Nozha Boujemaa. Robust voting algorithm based on labels of behavior for video copy detection. In *ACM Multimedia*, pages 835–844, Santa Barbara, CA, Oct 2006.

[61] Julien Law-To, Li Chen, Alexis Joly, Ivan Laptev, Olivier Buisson, Valerie Gouet-Brunet, Nozha Boujemaa, and Fred Stentiford. "Video Copy Detection: A Comparative Study". Amsterdam, The Netherlands, July 2007.

[62] Gustavo Leon, Hari Kalva, and Borko Furht. Video identification using video tomography. In *IEEE ICME*, pages 1030–1033, New York, NY, USA, June 2009.

[63] H. Lu, B. Ooi, and K. Tan. Efficient Image Retrieval by Color Contents. In *Int'l Conf. on Applications of Databases*, pages 95–108, June 1994.

[64] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. "Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search". In *VLDB*, pages 950–961, 2007.

[65] F. Malucelli, T. Ottmann, and D. Pretolani. "Efficient Labelling Algorithms for the Maximum Non Crossing Matching Problem". *Discrete Applied Mathematics*, 47(2):175–179, 1993.

[66] A. McCallum, K. Nigam, and L. H. Ungar. "Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching". Boston, MA, August 2000.

[67] Bhaskar Mehta, Saurabh Nangia, Manish Gupta, and Wolfgang Nejdl. Detecting Image Spam using Visual Features and Near Duplicate Detection. In *Int'l Conf. on World Wide Web (WWW)*, 2008.

[68] D. Menestrina, O. Benjelloun, and H. Garcia-Molina. "Generic Entity Resolution with Data Confidences". In *VLDB CleanDB Workshop*, Seoul, Korea, September 2006.

[69] M. Michelson and C. A. Knoblock. Learning Blocking Schemes for Record Linkage. In *AAAI*, 2006.

[70] Chong-Wah Ngo, Ting-Chuen Pong, and R.T. Chin. Video partitioning by temporal slice coherency. In *IEEE TCSVT*, volume 11, pages 941–953, 2001.

[71] B.-W. On, N. Koudas, D. Lee, and D. Srivastava. "Group Linkage". In *23rd Int'l Conf. on Data Engineering (ICDE)*, Istanbul, Turkey, 2007.

[72] B.-W. On, D. Lee, J. Kang, and P. Mitra. "Comparative Study of Name Disambiguation Problem using a Scalable Blocking-based Framework". June 2005.

[73] H. Pasula, B. Marthi, B. Milch, S. Russell, and I. Shpitser. “Identity Uncertainty and Citation Matching”. In *Advances in Neural Information Processing Systems*. MIT Press, 2003.

[74] Sébastien Poullot, Michel Crucianu, and Olivier Buisson. Scalable mining of large video databases using copy detection. In *ACM Multimedia*, pages 61–70, Vancouver, Canada, October 2008.

[75] H. Rangwala, E. Lantz, R. Musselman, K. Pinnow, B. Smith, and B. Wallenfelt. Massively Parallel BLAST for the Blue Gene/L. In *High Availability and Performance Computing Workshop (HAPCW)*, Santa Fe, NM, 2005.

[76] Huamin Ren, Shouxun Lin, Dongming Zhang, Sheng Tang, and Ke Gao. Visual words based spatiotemporal sequence matching in video copy detection. In *IEEE ICME*, pages 1382–1385, New York, NY, USA, June 2009.

[77] S. Sarawagi and A. Bhamidipaty. “Interactive Deduplication using Active Learning”. 2002.

[78] S. Sarawagi and A. Kirpal. Efficient Set Joins on Similarity Predicates. 2004.

[79] Shin’ichi Satoh, Masao Takimoto, and Jun Adachi. Scene duplicate detection from videos based on trajectories of feature points. In *MIR ’07: Proceedings of the international workshop on Workshop on multimedia information retrieval*, pages 237–244, 2007.

[80] D. A. Schneider and D. J. DeWitt. “A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment”. Portland, OR, May 1989.

[81] W. Shen, X. Li, and A. Doan. “Constraint-Based Entity Matching”. In *AAAI*, 2005.

[82] J.R. Smith and S.-F. Chang. Automated Binary Texture Feature Sets for Image Retrieval. In *IEEE Int. Conf. Acoust, Speech and Signal Proc.*, 1996.

[83] T. Smith and M. Waterman. Identification of Common Molecular Subsequences. *J. Mol. Biology*, 147:195–197, 1981.

[84] Hung-Khoon Tan, Chong-Wah Ngo, Richard Hong, and Tat-Seng Chua. Scalable detection of partial near-duplicate videos by visual-temporal consistency. In *ACM Multimedia*, pages 145–154, Beijing, China, October 2009.

[85] Hung-Khoon Tan, Xiao Wu, Chong-Wah Ngo, and Wan-Lei Zhao. Accelerating near-duplicate video matching by combining visual similarity and alignment distortion. In *ACM Multimedia*, pages 861–864, Vancouver, Canada, October 2008.

[86] Martin Theobald, Jonathan Siddharth, and Andreas Paepcke. SpotSigs: Robust and Efficient Near Duplicate Detection in Large Web Collections. 2008.

[87] Eduardo Valle, Matthieu Cord, and Sylvie Philipp-Foliguet. High-dimensional Descriptor Indexing for Large Multimedia Databases. In *ACM CIKM*, pages 739–748, Napa Valley, USA, October 2008.

[88] S. Velusamy, S. Bhatnagar, S.V. Basavaraja, and V. Sridhar. SPSA based feature relevance estimation for video retrieval. In *IEEE MMSP: Multimedia Signal Processing*, pages 598–603, Cairns, Queensland, Australia, October 2008.

[89] Geert Willems, Tinne Tuytelaars, and Luc Van Gool. Spatio-temporal features for robust content-based video copy detection. In *ACM MIR*, pages 283–290, Vancouver, Canada, October 2008.

[90] W. E. Winkler. "The State of Record Linkage and Current Research Problems". Technical report, US Bureau of the Census, April 1999.

[91] Xiao Wu, Alexander G. Hauptmann, and Chong-Wah Ngo. Novelty detection for cross-lingual news stories with visual duplicates and speech transcripts. pages 168–177, 2007.

[92] Xiao Wu, Alexander G. Hauptmann, and Chong-Wah Ngo. Practical elimination of near-duplicates from web video search. pages 218–227, 2007.

[93] Xiao Wu, Jintao Li, Yongdong Zhang, and Sheng Tang. Spatio-temporal visual consistency for video copy detection. In *IEEE VIE*, pages 414–419, Xian, China, July 2008.

[94] Zhipeng Wu, Qingming Huang, and Shuqiang Jiang. Robust copy detection by mining temporal self-similarities. In *IEEE ICME*, pages 554–557, June 2009.

[95] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient Similarity Joins for Near Duplicate Detection. 2008.

[96] Zhihua Xu, Hefei Ling, Fuhao Zou, Zhengding Lu, Ping Li, and Tianjiang Wang. Fast and robust video copy detection scheme using full dct coefficients. In *IEEE ICME*, pages 434–437, June 2009.

[97] Ying Yan, Beng Chin Ooi, and Aoying Zhou. Continuous content-based copy detection over streaming videos. In *IEEE ICDE*, pages 853–862, April 2008.

[98] Hui Yang and James P. Callan. Near-Duplicate Detection by Instance-Level Constrained Clustering. 2006.

[99] Mei-Chen Yeh and Kwang-Ting Cheng. A compact, effective descriptor for video copy detection. In *ACM Mutimedia*, pages 633–636, Beijing, China, October 2009.

[100] D. Zhang and G. Lu. Review of Shape Representation and Description Techniques. *Pattern Recognition*, 37(1):1–19, 2004.

[101] Dong-Qing Zhang and Shih-Fu Chang. Detecting Image Near-Duplicate by Stochastic Attributed Relational Graph Matching with Learning. pages 877–884, Oct 2004.

[102] Dong-Qing Zhang and Shih-Fu Chang. Detecting Image Near-Duplicate by Stochastic Attributed Relational Graph Matching with Learning. 2004.

[103] Wan-Lei Zhao, Chong-Wah Ngo, Hung-Khoon Tan, and Xiao Wu. Near-Duplicate Keyframe Identification with Interest Point Matching and Pattern Learning. *IEEE Trans. on Multimedia*, 9:1037–1048, August 2007.

[104] Yan-Tao Zheng, Shi-Yong Neo, Tat-Seng Chua, and Qi Tian. The Use of Temporal, Semantic and Visual Partitioning Model for Efficient Near-Duplicate Keyframe Detection in Large Scale News Corpus. In *ACM CIVR*, pages 409–416, Amsterdam, The Netherlands, July 2007.

Vita

Hung-sik Kim

Hung-sik Kim was born in Gangreung, South Korea in 1971. He received his B.S. degree from Electrical Engineering, Konkuk University, Seoul, Korea, in 1999. Then, He received M.S. degree from Electrical Engineering, the Pennsylvania State University, University Park, USA, in 2001. Then, he joined Ph.D program in the Department of Computer Science and Engineering at the Pennsylvania State University in 2004. His current research focuses on High Performance Computing, Data Cleaning, Data Clustering, and Multimedia Copy Detection.