HARRA: Fast Iterative Hashed Record Linkage for Large-Scale Data Collections

Hung-sik Kim
The Pennsylvania State University
University Park, PA 16802, USA
hungsik@psu.edu

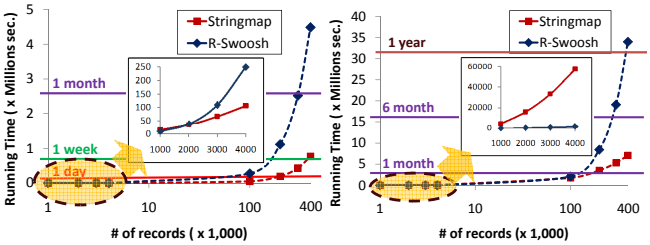
Dongwon Lee*
The Pennsylvania State University
University Park, PA 16802, USA
dongwon@psu.edu

ABSTRACT

We study the performance issue of the "iterative" *record linkage (RL)* problem, 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, by exploiting inherent characteristics within/across data sets, we develop a suite of I-LSH-based RL algorithms, named as HARRA (HAshed RecoRd linkAge). The superiority of HARRA in speed over competing RL solutions is thoroughly validated using various real data sets. While maintaining equivalent or comparable accuracy levels, for instance, HARRA runs: (1) 4.5 and 10.5 times faster than StringMap and R-Swoosh in iteratively linking 4,000 × 4,000 short records (i.e., one of the small test cases), and (2) 5.6 and 3.4 times faster than basic LSH and Multi-Probe LSH algorithms in iteratively linking 400,000 × 400,000 long records (i.e., the largest test case).

1. INTRODUCTION

The quality of the data residing in databases gets degraded due to a multitude of reasons: i.e., transcription errors (e.g., lexicographical errors, character transpositions), lack of standards for recording database fields (e.g., person names, addresses), or various errors introduced by poor database design (e.g., update anomalies, missing key constraints). To be able to query and integrate such data in the presence of data quality errors, a central task is to identify whether two entities are approximately the *same*. When each entity is represented as a relational record, this problem is often referred to as the **Record Linkage (RL)** problem: *Given two*[1] *collections of compatible records,* $A=\{a_1, ..., a_m\}$ *and* $B=\{b_1, ..., b_n\}$*, RL does: (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* $\forall c_i, c_j \in C$*,* $c_i \not\approx c_j$*.* Such a problem is also known as the *de-duplication* or *entity resolution* problem and has been extensively studied in recent years (to be surveyed in Section 2.2).



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

Figure 1: Running times of two RL solutions, StringMap and R-Swoosh, for two data sets (in self-clean case). X-axis is on Logarithmic scale.

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. A more 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, 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 records in each cluster. In such a case, the cost of the term, $\bar{c}(\bar{c}-1)|C|$, alone becomes prohibitively expensive. Take two popular RL solutions, StringMap [19] and R-Swoosh [6], for instance. Figure 1 shows the running times (from the beginning of data load to the finish of the linkage) of both algorithms for self-cleaning data collections

*Partially supported by NSF DUE-0817376 and DUE-0937891 awards.

[1]Cleaning single collection A is handled as a self-clean case of $A \times A$.

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.

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[2]. 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) or for large data collections (or for both).

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 [25], 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, conventional RL solutions become simply too slow.

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 paper are: (1) We formally introduce the RL problem with separate match and merge steps, and exploit them to have better RL framework for three data collection scenarios (i.e., clean-clean, clean-dirty, and dirty-dirty); (2) 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; (3) Using the I-LSH proposal, depending on three scenarios, we propose a suite of RL solutions, termed as HARRA (HAshed RecoRd linkAge), that exploits data collection characteristics; and (4) The superiority of HARRA in speed over competing RL solutions is thoroughly validated using various real data sets under diverse characteristics. For instance, while maintaining equivalent or comparable accuracy levels, HARRA runs: 4.5 and 10.5 times faster than StringMap and R-Swoosh algorithms in linking 4,000 × 4,000 short records (i.e., one of the small test cases), and 5.6 and 3.4 times faster than basic LSH and Multi-Probe LSH algorithms in linking 400,000 × 400,000 long records (i.e., the largest test case),

2. BACKGROUND

2.1 Preliminaries

Consider two records, r and s, with $|r|$ columns each, where $r[i]$ ($1 \leq i \leq |r|$) refers to the i-th column of the record r. Further, let us assume that corresponding columns of r and s have compatible domain types: $dom(r[i]) \sim dom(s[i])$.

Definition 1 (Record Matching) *When two records, r and s, refer to the same real-world entity, both are said* matching, *and written as $r \approx s$ (otherwise $r \not\approx s$).* □

Note that how one determines if two records refer to the same real-world entity or not is *not* the concern of this paper. Assuming the existence of such oracle, we instead focus on how to find matching records more effectively and faster. In practice, however, the matching of two records can be often determined by distance or similarity functions (e.g., Jaccard).

[2]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.

Symbol	Meaning
A, B, a_i, b_j	two input collections and records in A, B
m and n	size of A and B, i.e., $m = \|A\|$, $n = \|B\|$
c_{ij} or $c_{i,j}$	a merged record from a_i and b_j
θ	threshold for $a_i \oplus b_j$
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}
binary(a_i)	returns a binary vector, v_i, of a record a_i
H (or H_A, H_B)	a hash table (or a hash table created from A, B)
$AList$	a bucket of a hash table

Table 1: Summary of notations.

When two records, r and s, are matching (i.e., $r \approx s$), four relationships 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 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 $r \equiv s$).

In this paper, we focus on the *match-merge* RL model, introduced in [25]. Since the match-merge RL model is more general than traditional *match-only* RL model, an RL solution that solves the match-merge RL model (e.g., Swoosh, HARRA) can also solve the match-only RL model without any change. Therefore, 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 paper. We simply refer to a function that merges r and s to get a new record w as **merge(r,s)**. We use the following two terms to describe the quality of data collections.

Definition 2 (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 $\exists r, s \in A$, $r \approx s$.* □

Unlike the similarity between numerical values can be easily computed by L-norm distance, 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)**. Table 1 summarizes the notations.

2.2 Related Work

The general record linkage (RL) problem has been also known as various names: merge-purge (e.g., [18]), entity resolution (e.g., [30, 27]), object matching (e.g., [9]), identity uncertainty (e.g., [29]), citation matching (e.g., [28]), and approximate string join (e.g., [17]) etc.

Since the focus of our paper is orthogonal to many of these works, in this section, we survey a few recent representative works only. [14] studies the name matching for information integration using string-based and token-based methods, while [28] conducts an in-depth study on the RL problem in digital library context. ALIAS system in [30] proposes a framework to detect duplicate entities, but its focus is on learning. In recent years, there have been many attempts to exploit additional information beyond string comparison for RL. For instance, [20] presents a relationship-based data cleaning which exploits context information for RL, while [7] exploits additional relational information in data (e.g., coreference). Similarly, [12] proposes a generic semantic distance measure using the page counts from the Web. In addition, [22] exploits the given data characteristics (clean vs. dirty) in improving "parallel" RL performance. In this paper, similar idea is significantly extended to accommodate the benefits of iterative LSH.

Methods to speed up the performance of RL vary greatly. For instance, the traditional "blocking" technique was first proposed in [21] and further improved in [24]. Unlike the traditional methods exploiting textual similarity, constraint-based entity matching [32] examines semantic constraints in an unsupervised way. [6] presents a generic framework, *Swoosh* algorithms, for the entity resolution problem. The recent work by [15] 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. More recently, the group of works on the *set-similarity join* (SSJoin) [31, 10] are relevant to our proposal. Optimization techniques developed in the literature (e.g., size filtering [3], prefix filtering [10], order filtering [5], or suffix filtering [33]) can be applied to the RL problem (and thus to our techniques) when the threshold model is used for measuring similarities (as opposed to the top-k model). 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. We leave the investigation of incorporating SSJoin optimization techniques to our HARRA as future work.

On the other hand, the Locality-Sensitive Hashing (LSH) scheme [16, 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 hash tables, and how to select right number of functions or tables? Recently, distance-based hashing (DBH) [4] is proposed to address the issue of finding a family of hash functions in LSH. Similarly, multi-probe LSH [23] 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 string or sequence comparisons, substitution-based measures are proposed in [1]. In this paper, we extend the LSH technique and propose the Iterative LSH (I-LSH) technique (and a suite of HARRA 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.

	Data	Model	Blocking	Metric
[19]	54,000/133,101 names 20,000 DBLP	match only	R-tree based	running time, accuracy
[6]	5,000 products 14,574 hotels	match-merge	N/A	running time, accuracy
[4]	15,853 UNIPEN 70,000 MNIST 80,640 hand images	match only	distance based (VP-tree like)	accuracy, efficiency
[23]	1,312,581 images 2,663,040 words	match only	LSH based	recall, query time, memory usage
[8]	Cora (1,295) DBGen (50,000)	match only	trained blocking	accuracy
[26]	864 restaurants 5,948 cars	match only	trained blocking	accuracy

Table 2: Comparison of a few recent RL algorithms.

Table 2 shows the comparison among a few recent RL algorithms. Among these, in this paper, we compare HARRA against *unsupervised* RL solutions such as [19, 6, 23] since they tend to be faster than supervised ones and their implementations are readily available.

3. ITERATIVE LSH

3.1 LSH with MinHash

The basic idea of the *Locality-Sensitive Hashing (LSH)* technique was introduced in [16]. 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,

Definition 3 (LSH function family) *A function family $H = \{h : R \rightarrow U\}$ 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 2):

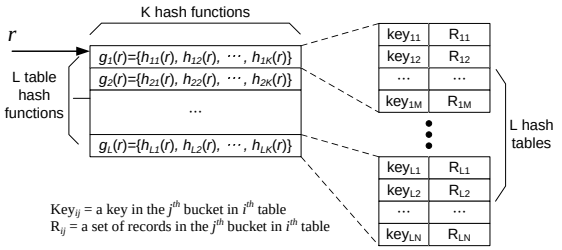


Figure 2: A basic LSH structure.

- **K**: # of hash functions from function family $G=\{g : S \rightarrow U^K\}$ 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 2. 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 [16].

There are several methods to construct an LSH family such as bit sampling or random projection. In particular, MinHash [13] 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 1. 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. □

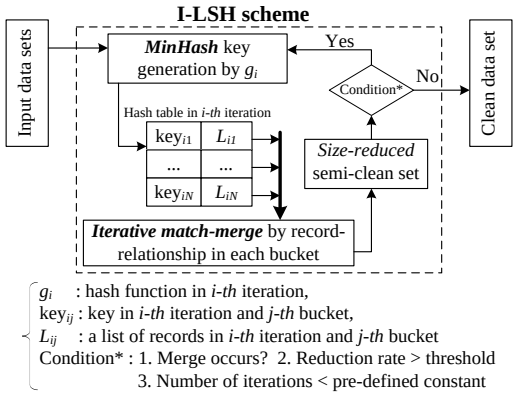


Figure 3: The general structure of I-LSH.

3.2 Iterative LSH to Clean Single Data Set

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 *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 3 illustrates the basic flow of I-LSH while Algorithm 1, **h-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 1) 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, 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 2 of Algorithm 1.

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 1: h-$\mathbf{D}_{self}$.

```
Input  : A non-empty dirty list A
Output : A non-empty clean list C
Make H as an empty hash table;
Flag ← true; /* Flag determines iterations */
while Flag = true do
    Flag ← false; j ← 1;
    while j ≤ |A| do
        /* hash a_j to a bucket, AList */
        key ← g_i(a_j); AList = H.get(key);
        if AList≠ ∅ then
            k ← 1;
            while k ≤ |AList| do
                switch match(a_k, a_j) do
                    case a_k ≡ a_j or a_k ⊒ a_j
                        Flag ← true; remove a_j from A;
                        go to line 1;
                    case a_k ⊑ a_j
                        Flag ← true;
                        remove a_k from AList; k ← k + 1;
                    case a_k ⊕ a_j
                        Flag ← true;
                        remove a_k from AList & a_j from A;
                        a_j ← merge(a_k, a_j); k ← 1;
                    case a_k ⋡ a_j
                        k ← k + 1;
        AList.add(a_j); H.put(key,AList);
1       j ← j + 1;
    A ← H.getAll(keys); /* put all records back to A */
    i ← i + 1 /* re-hash in next iteration using g_{i+1} */
2   if termination condition is met then Flag ← false;
C ← A; return C;
```

Algorithm 1. 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[3] in a hash table (i.e., *AList* in Algorithm 1). With a proper choice of hash functions, hash collisions should occur rarely. Therefore, in general, $\hat{c}$ is relatively small. Furthermore, in Algorithm 1, 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 1. *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 10 of Section 5). 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 2. *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. 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 HARRA algorithms.

4. 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, based on the I-LSH idea. In total, we propose six variations of HARRA. Each HARRA 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.

4.1 Clean vs. Clean

Recall that unlike database join, in the match-merge RL model, if two records a_k and b_j match, then a merged record c_{kj} (= *merge*(a_k,b_j)) may be created and re-feeded into A and B. In particular, when $a_k \equiv b_j$ holds, since both A and B are known to be "clean" already, there cannot be any further records that match either a_k in A or b_j in B. Therefore, one does *not* need $n - j + m - k$ times of *match*() computations between a_k and $\{b_{j+1}, ..., b_n\}$ and those between b_j and $\{a_{k+1}, ..., a_m\}$. Similarly, if $a_k \sqsubseteq b_j$ holds, since b_j contains all information of a_k, b_j itself will be used as a merged record, and one can save $n-j$ times of *match*() computations between a_k and $\{b_{j+1}, ..., b_n\}$. Symmetrically, if $a_k \sqsupseteq b_j$ holds, one can save $m - k$ times of *match*() computations between b_j and $\{a_{k+1}, ..., a_m\}$. Finally, when $a_k \oplus b_j$ holds, c_{kj} is used as the new merged record and one can save $n - j + m - k$ times of *match*() computations between a_k and $\{b_{j+1}, ..., b_n\}$ and those between b_j and $\{a_{k+1}, ..., a_m\}$.

Figure 4 illustrates the structure of **h-CC**(A, B), while Algorithm 2 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 2. 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.

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

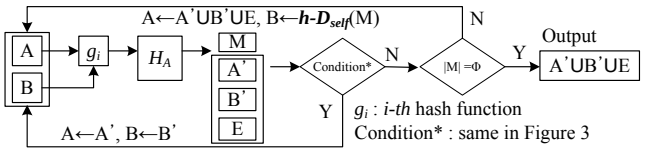


Figure 4: The structure of h-CC.

Algorithm 2: 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 1;
        case $a_k \sqsupseteq b_j$
          $Flag \leftarrow$ **true**; remove b_j from B; go to line 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 1;
        case $a_k \not\approx b_j$
          $k \leftarrow k+1$;
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;

Example 2. 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. □

Algorithm 3: 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 1;
        ...
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;

4.2 Clean vs. Dirty

The detailed procedure for clean-dirty case, **h-CD**, is shown in Algorithm 3 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 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 3, we denote this variation as **h-CD**$_{self}$. Algebraically, the following holds: **h-CD**$_{self}(A, B) \equiv$

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

h-CC$(A, \textbf{h-D}_{self}(B))$.

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 $|B|$ 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.

4.3 Dirty vs. Dirty

Since neither collection A or B is clean, more comparisons are needed for dirty-dirty case. By using HARRA 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**$(\textbf{h-D}_{self}(A), B)$
- **h-DD3**$(A, B) \equiv$ **h-CC**$(\textbf{h-D}_{self}(A), \textbf{h-D}_{self}(B))$

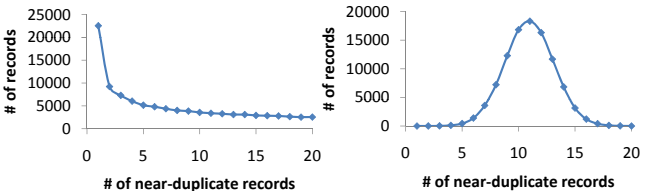
The different behaviors of variations will be evaluated experimentally.

Lemma 3. *All* HARRA *algorithms have polynomial upper bounds in time/space complexities, as shown in Table 3.* ■

5. EXPERIMENTAL VALIDATION

Under various settings (e.g., different data distributions, varying sizes of data sets, and dirtiness), we evaluated six HARRA 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 HARRA robust over various settings? (2) Does HARRA achieve high accuracy and good scalability? 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[4] (with dual 2.4-2.6GHz AMD Opteron Processors and 8GB RAM) was used. Note that HARRA are also run on LION-XO for comparison purpose.

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



(a) Power-Law (PL) (b) Gaussian (GA)

Figure 5: Example records 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 HARRA

Table 4: Symbols used in experiments.

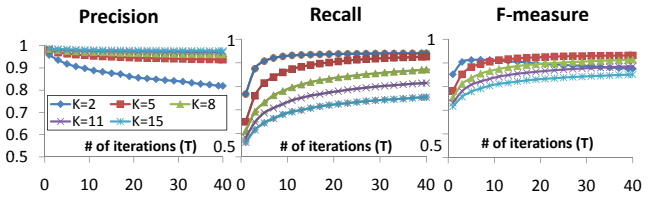
5.1 Set-Up

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 5(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.

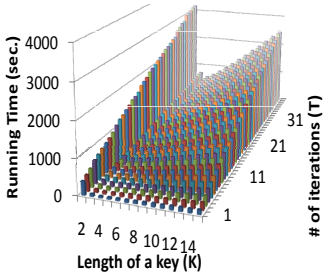
In addition to different distributions, we also used two matching rates: (1) Internal Matching Rate of A: **IMR**(A) = $\frac{\text{\# of dirty records in } A}{\text{\# of all records in } A}$, and (2) Cross Matching Rate of A against B: **CMR**(A,B) = $\frac{\text{\# of records in } A \text{ that matches a record in } B}{\text{\# of all records in } A}$. For instance, if IMR=0.5, then half of records in a collection are dirty and need to be merged. By varying both IMR and CMR, we control the "dirtiness" within a data set, and between two data sets. 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 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.

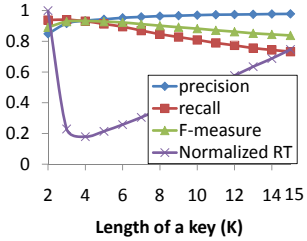
Symbols used in experiments are summarized in Table 4.



(a) Precision, Recall, and F-measure



(b) RT comparison



(c) Summary ($T = 25$)

Figure 6: Impact of K and T using h-D$_{self}$ to one dirty set generated by PL distribution.

5.2 Choice of Parameters: K, L, and T

Several factors may impact the performance of HARRA algorithms – some of them (e.g., K, L, and T) are control parameters that HARRA 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 HARRA implementations (and their rationale) in this section. On the other hand, the robustness of HARRA with respect to varying distribution patterns and dirtiness of data sets is validated through Sections 5.5–5.7.

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(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(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(b), clearly, RT is proportional to T. That is, if HARRA 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(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(c) shows the summary of precision, recall and F-measure of varying K with $T = 25$. The *normalized RT* of Figure 6(c) is computed by dividing all RT's by the maximum RT at $K = 2$. Based on Figure 6(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 [4, 23] . For this reason, in subsequent experiments, we used $K = 5$. Note that in Section 5.4 where we compare HARRA 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 5.4 at all.

L: The behavior of conventional LSH schemes (e.g., [16, 23]) is controlled by both K and L parameters. While HARRA 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 HARRA, one key per record is generated *per iteration* (see Figure 3). In order to have a fair comparison between HARRA and conventional LSH schemes in Section 5.4, we need to have the *same* number of keys per record. Therefore, in HARRA, we have to have at least L times of iterations to have L keys per record. As a conclusion, in HARRA, 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 8(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 HARRA, 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 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, HARRA usually performs *less* number of iterations T with PL distribution. In all of subsequent experiments, we used $\sigma_i = 0.01$.

5.3 Comparison Against Existing RL Solutions

First, we chose two well-known RL algorithms[5], StringMap[6] [19] and R-Swoosh[7] [6], and compared them against one of HARRA algorithms, **h-D**$_{self}$, for the case of cleaning one dirty set. The first three columns of Table 5 show the differences among StringMap, R-Swoosh, and HARRA in detail. Since StringMap supports only the match-only[8] RL model,

[5]We also considered Febrl [11] 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.

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

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

[8][19] actually supports merge() operation. However, the

	StringMap	R-Swoosh	HARRA	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 5: Comparison between HARRA and four other RL solutions.

Data size	1,000		2,000		3,000		4,000	
	pre.	rec.	pre.	rec.	pre.	rec.	pre.	rec.
HARRA	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: Precision & Recall comparison among h-$\mathbf{D}_{self}$, StringMap, and R-Swoosh.

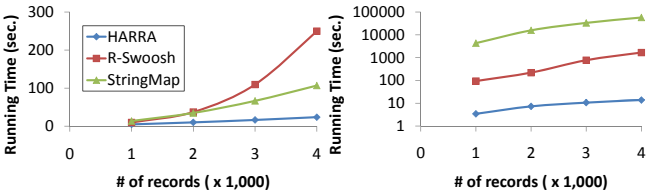
comparison was done under the match-only RL model (i.e., no merges). Since both StringMap and R-Swoosh could not finish larger data sets within a reasonable time, data sets with 1,000 – 4,000 records were mainly used for comparison. Two record types were used: short records (e.g., people names of 10-20 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.

As shown in Figure 7(a), with simple record contents such as people names, all three algorithms run well within a reasonable time. Both HARRA 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, HARRA 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, if both StringMap and R-Swoosh could have been able to run for larger data sets such as 400,000 records, the gap between HARRA and them would have widened further. For the second type of a long record set, as shown in Figure 7(b), StringMap works worse than R-Swoosh. With initial data set of 1,000 citations, RT of HARRA 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 shows. The precision values for all 3 methods are all equal as 1.0. Recall that the ground truth of this experiment in Section 5.3 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 blocking may miss true positives[9]. Since

merged record does not incur new comparison in subsequent iterations.

[9]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) short record (e.g., names) (b) long record (e.g., citations)

Figure 7: RT comparison among h-$\mathbf{D}_{self}$, StringMap, and R-Swoosh. Note that Y-axis of (b) is on Logarithmic scale.

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 HARRA 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 HARRA got {0.998, 0.996, 0.992, 0.985}.

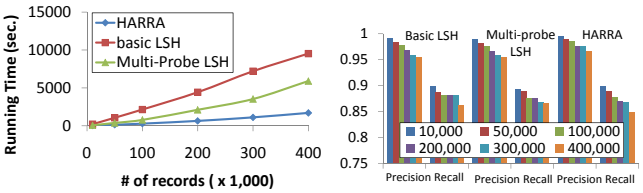
Overall we claim that HARRA *runs much faster with negligible loss of recall.* For instance, HARRA runs 4.5 and 10.5 times faster than StringMap and R-Swoosh with 4,000 short name record data set while missing only 1.5% of true matches out of 4,000 × 4,000 record pairs.

5.4 Comparison Against Existing LSH Based RL Solutions

Next, we compare **h-$\mathbf{D}_{self}$** against both basic LSH [16] and multi-probe LSH [23] 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 5 show the differences among HARRA, basic LSH, and multi-probe LSH in detail.

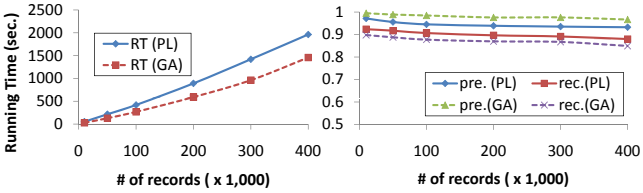
Figure 8(a) shows the comparison of RT among three approaches with varying size of input records, while Figure 8(b) shows the trend of precision and recall which are managed to be similar for the proper RT comparison. Since HARRA uses one re-usable hash table, the memory usage of HARRA is significantly lower than regular LSH techniques. In fact, the memory requirement in both basic LSH and multi-probe LSH depends on the number of hash tables used in the systems, while HARRA uses *one* dynamic reusable hash table. In addition, with respect to running time for 400,000 records (i.e., the largest test set), HARRA 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 HARRA 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 HARRA algorithms against the basic and multi-probe algorithms shows similar pattern,



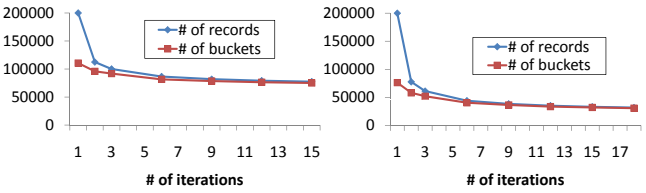
(a) RT comparison (b) Accuracy comparison

Figure 8: Comparison, h-D$_{self}$ to others with Gaussian distribution, $\sigma = 1$ and $mean = 11$



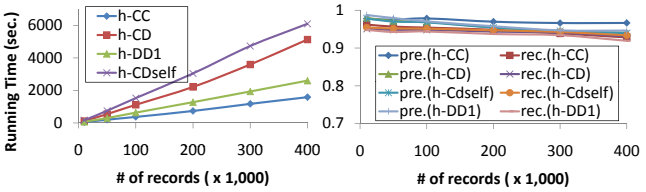
(a) Running Time (b) Precision & Recall

Figure 9: h-D$_{self}$ with Power-Law (PL) and Gaussian (GA) distributions



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

Figure 10: # of records & # of buckets in a hash table at each iteration with 200,000 records.



(a) Running time (b) Accuracy

Figure 11: All algorithms for different scenarios.

in subsequent sections, we focus on comparing various aspects among HARRA algorithms in detail.

5.5 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 9, **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.

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 used as a control factor to stop iterations. In Figure 10, 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 10 (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.

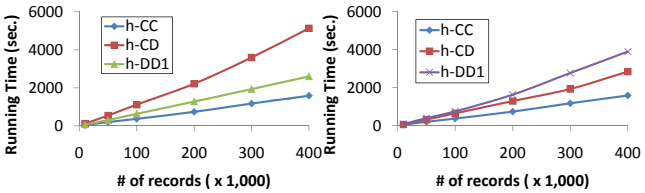
5.6 Cleaning Pairs of Data Collections

In this section, we validate all HARRA 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 11 shows the scalability and accuracy of four HARRA algorithms for clean-clean, clean-dirty, and dirty-dirty scenarios. Virtually, all algorithms show similar patterns. As to scalability, as shown in Figure 11(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 HARRA 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 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 11(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 12(b) appear "faster" than those shown in Fig-



(a) Known characteristic (b) Unknown characteristic

Figure 12: 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)

	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 7: Precision and recall with various dirtiness

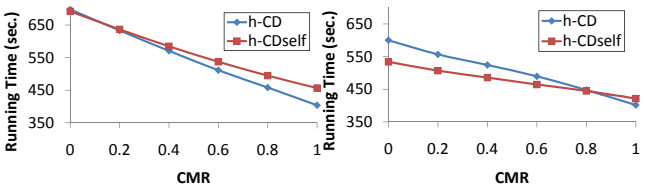
ure 12(a), contrary to the intuition. This is because two cases use different data set. For Figure 12(b) with unknown characteristics, single data set is used to exploit clean characteristic for all algorithms when it is known to user. In **h-CC**, one does not need to compare records internally in each 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 of keys are generated for one record in all algorithms by setting the number of iterations.

5.7 Robustness of HARRA

In section 5.5, 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 HARRA 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 7 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 HARRA 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



(a) $IMR = 0.0$ (b) $IMR = 0.4$

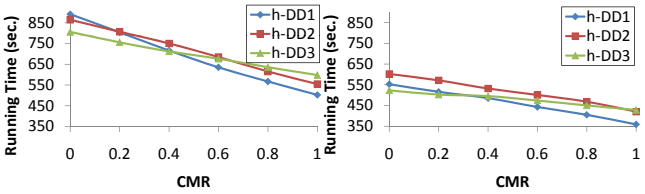
Figure 13: Algorithm comparison in clean-dirty case by varying IMR and CMR, with 100,000 records

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 13(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, 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 13(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 13(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 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 14(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-**



(a) $IMR = 0.0$ (b) $IMR = 0.4$

Figure 14: Algorithm comparison in dirty-dirty case by varying IMR and CMR, with 100,000 records

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 14(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 14(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. CONCLUSION

We proposed HARRA by investigating iterative structures of LSH algorithms to clean and merge data sets. For three input cases of clean-clean, dirty-clean, and dirty-dirty, we presented six solutions. 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), HARRA 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 HARRA. Many directions are ahead for future work. First, we plan to extend all HARRA algorithms to be parallel. Second, the current HARRA algorithms can be extended to other data problems (e.g., clustering) or domains (e.g., multimedia).

Availability. HARRA implementations and sample data sets used in this paper are available at:

`http://pike.psu.edu/download/edbt10/harra/`

7. REFERENCES

[1] A. Andoni and P. Indyk. Efficient Algorithms for Substring Near Neighbor Problem. In *Symp. on Discrete Algorithyms (SODA)*, pages 1203–1212, 2006.

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

[3] A. Arasu, V. Ganti, and R. Kaushik. Efficient Exact Set-Similarity Joins. In *VLDB*, 2006.

[4] V. Athitsos, M. Potamias, P. Papapetrou, and G. Kollios. Nearest Neighbor Retrieval Using Distance-Based Hashing. In *IEEE ICDE*, pages 327–336, 2008.

[5] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling Up All Pairs Similarity Search. In *Int'l World Wide Web Conf. (WWW)*, 2007.

[6] 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, Jan. 2009.

[7] I. Bhattacharya and L. Getoor. Collective Entity Resolution In Relational Data. *ACM Trans. on Knowledge Discovery from Data (TKDD)*, 1(1):1–36, 2007.

[8] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive Blocking: Learning to Scale Up Record Linkag. In *ICDM*, December 2009.

[9] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and Efficient Fuzzy Match for Online Data Cleaning. In *ACM SIGMOD*, 2003.

[10] S. Chaudhuri, V. Ganti, and R. Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. In *IEEE ICDE*, 2006.

[11] P. Christen. Febrl: an open source data cleaning, deduplication and record linkage system with a graphical user interface. In *ACM KDD*, pages 1065–1068, Aug. 2008.

[12] R. Cilibrasi and P. M. B. Vitanyi. The Google Similarity Distance. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, (3):370–383, 2007.

[13] 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. IEEE Trans. on Knowledge and Data Engineering (TKDE), Jan/Feb 2001.

[14] W. Cohen, P. Ravikumar, and S. Fienberg. A Comparison of String Distance Metrics for Name-matching tasks. In *IIWeb Workshop held in conjunction with IJCAI*, 2003.

[15] X. Dong, A. Y. Halevy, and J. Madhavan. Reference Reconciliation in Complex Information Spaces. In *ACM SIGMOD*, 2005.

[16] A. Gionis, P. Indyky, and R. Motwaniz. Similarity Search in High Dimensions via Hashing. In *VLDB*, pages 518–529, 1999.

[17] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text Joins in an RDBMS for Web Data Integration. In *Int'l World Wide Web Conf. (WWW)*, 2003.

[18] M. A. Hernandez and S. J. Stolfo. The Merge/Purge Problem for Large Databases. In *ACM SIGMOD*, 1995.

[19] 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, Dec. 2006.

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

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

[22] H. Kim and D. Lee. Parallel Linkage. In *ACM CIKM*, Lisbon, Portugal, 2007.

[23] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *VLDB*, pages 950–961, 2007.

[24] A. McCallum, K. Nigam, and L. H. Ungar. Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching. In *ACM KDD*, Aug. 2000.

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

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

[27] B.-W. On, N. Koudas, D. Lee, and D. Srivastava. Group Linkage. In *IEEE ICDE*, Istanbul, Turkey, Apr. 2007.

[28] B.-W. On, D. Lee, J. Kang, and P. Mitra. Comparative Study of Name Disambiguation Problem using a Scalable Blocking-based Framework. In *ACM/IEEE Joint Conf. on Digital Libraries (JCDL)*, Jun. 2005.

[29] H. Pasula, B. Marthi, B. Milch, S. Russell, and I. Shpitser. Identity Uncertainty and Citation Matching. In *NIPS*. 2003.

[30] S. Sarawagi and A. Bhamidipaty. Interactive Deduplication using Active Learning. In *ACM KDD*, 2002.

[31] S. Sarawagi and A. Kirpal. Efficient Set Joins on Similarity Predicates. In *ACM SIGMOD*, 2004.

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

[33] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient Similarity Joins for Near Duplicate Detection. In *Int'l World Wide Web Conf. (WWW)*, 2008.