School of Computer Science

College of Engineering and Computer Science

COMP6470 - Special Topic in Computer Science

A Two Stage Similarity-aware Indexing for Real-time Large Scale Entity Resolution

---

Author:

Shouheng Li

Supervisor:

Dr Huizhi(Elly) Liang

June 3, 2013

Abstract

Entity resolution, or data integration, is the process that identifies and matches data records that refer to the same real world entity. In many cases, entity resolution needs to be performed in sub-seconds. The challenge is especially enormous when entity resolution is processed on large scale datasets.

Some techniques have been proposed to solve the problem. Indexing is a technique that enables real-time entity resolution by only comparing record pairs that have the same encoding value and thus largely reduce the number of comparisons. Similarity-aware Indexing (SAI) improves the performance of traditional indexing by pre-calculating similarities of attribute values. Another technique known as Locality Sensitive Hashing (LSH) provides a similarity based filtering. Minhash is an implementation of LSH.

In this report, a two-stage approach which combines LSH with SAI is proposed. At the first stage, LSH is adopted to approximate filter data records and preserve the potential matches. At the second stage, SAI is used to compare potential matches to obtain a precise and accurate result. This approach is evaluated experimentally on two large scale datasets taken from real-world databases. Experimental results show that the two-stage approach can processes queries 10 times faster than the original similarity-aware indexing. This approach can also perform well at scenarios where precise query results are needed.

Keywords: Entity Resolution, Real-time, Blocking, Locality Sensitive Hashing, Scalability, Dynamic Data.

Contents

Chapter 1

Introduction

With the utilisation of databases and information systems, businesses, governments and organisations are able to collect tons of information without much difficulty. Based on the collected information, data mining techniques can be adopted to analyse the data and discover useful knowledge. However, the raw data might be *dirty*, containing data that are incomplete, inconsistent and noisy. So the raw data is often required to be *pre-processed* or *cleaned* before further use. One of the important steps in data pre-processing is called entity resolution, or data integration, which is the process that identifies and matches data records that refer to the same real world entity. Entity resolution is often used when there are data from different data sources. For example, if we have a dataset from a hospital's patient records, and another dataset from the facebook profiles. These two datasets are very different, containing different attributes, different formats, etc., but they refer to the same group of people. In order to obtain a comprehensive knowledge of these people, we wish to link the datasets by using entity resolution techniques to identify records that refer to the same people.

Traditional entity resolution for large datasets is exhausting and time consuming as it requires multiple traversals of datasets. However, in many cases, entity resolution needs to be performed in sub-seconds, which brings up a real-time challenge. The challenge is especially enormous when entity resolution is processed on large scale datasets. An Example is a online credit card verification system, which decides if a credit card inquery is valid by matching it with millions of records in a credit card database.

Some techniques have been proposed to solve the problem. Indexing is a technique that enables real-time entity resolution by only comparing record pairs that have the same encoding value and thus largely reduce the number of comparisons. Similarity-aware Indexing (SAI) improves the performance of traditional indexing by pre-calculating similarities of attribute values. Nevertheless, sometimes when encoding values are frequent, a large number of records will have the same encoding value and the comparisons will still be computationally expensive. Besides indexing, another technique called Locality Sensitive Hashing (LSH) provides a similarity based filtering that "hashes" similar items together. Minhash is an implementation of LSH that simulates Jaccard similarity via "hashing". Although both LSH and indexing are proved to be effective in real-time entity resolution, the possibility of using LSH to improve the performance of indexing had never been explored.

The work presented in this report focuses on the real-time challenge for entity resolution. In this report, a two-stage approach which combines LSH with SAI is proposed. At the first stage, LSH is adopted to approximate filter data records and preserve the potential matches. At the second stage, SAI is used to compare potential matches to obtain a precise and accurate result. The proposed two-stage approach is evaluated using two real-world large datasets. Experiment shows that the two-stage approach significantly improves the matching speed without obvious loss of accuracy.

Chapter 2

Related Work

Traditional way of finding similar items in a large scale dataset is exhausting as comparison has to be conducted on every record pairs. For instance, if dataset A has 1 million reocrds and dataset B has 2 million records, linking the two data sets requires comparisons for 200 billion record pairs! Normally, instead of every record pairs, we only interest in those pairs that are potential matches. If we can filter the databases by eliminating those pairs that are obviously different, the number of comparisons can be largely reduced. Locality Sensitive Hashing and Indexing are two approaches that provide the "filtering".

2.1 Locality Sensitive Hashing (LSH)

Locality sensitive hashing (LSH), or *near-neighbor search*, is a probability based dimension reduction scheme. The basic theory behind LSH can be simply expressed as neighbourhood items in a high dimension space are very likely to stay close after being projected on a lower dimension space [12]. An example is shown in Figure 2.1. Two close points (circles) in the sphere stays close after they are projected on the printed page[12]. Although for some orientations two far-distanced points (squares) will be projected close to each other, for most orientations, they will be far apart and the close points in the sphere will stay close.

LSH is valuable for finding similar items because its dimension reduction largely save computational time and power. LSH is often implemented using "hashing" to simulate projection. "Hashing" is performed by multiple hash functions. Each

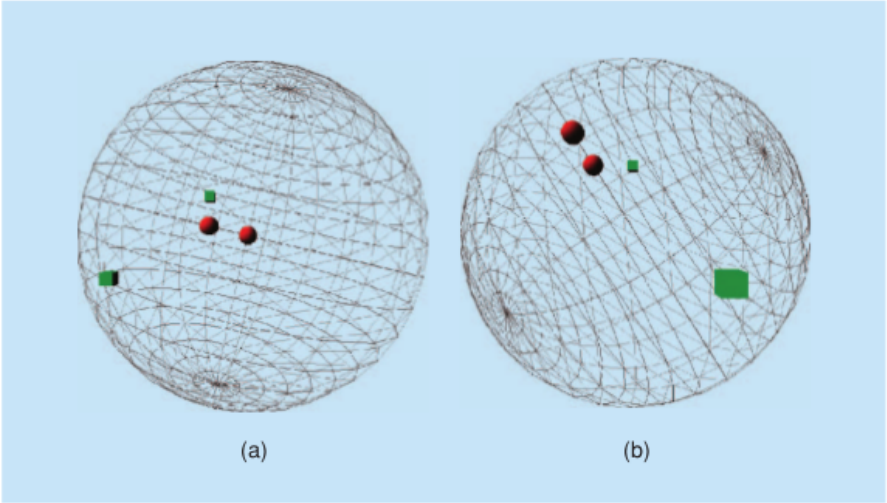


Figure 2.1: Projections of two close (circles) and two distant (squares) points onto the printed page, taken from Slaney and Casey's lecture notes[12]

.

hash function generates a hash value, and multiple hash values are combined to form a "signature". The items with the same signature are said as "hashed" into a same bucket. The items of a same bucket is considered as potential matches. Comparing with the whole dataset, a bucket contains much less items, therefore much less iterations are needed to compared the candidate items.

Since LSH is a probability based projection, there are probabilities that true matches are not hashed into same buckets. However, LSH allows the probabilities to be tuned by adjusting the number of hash functions and the number of hash values in a signature. The probability of not finding true matches can be tuned as small as possible at the cost of losing precision and increasing computional time.

As mentioned before, LSH is actually a similarity simulation scheme. In this report, we focus on an implementation of LSH called *Minhash*, which simulates Jaccard similarity.

2.1.1 Minhash

Minhash, invented by Andrei Broder (1997)[2], is an efficient estimation of two sets' Jaccard similarity. The Jaccard similarity of two sets S_1 and S_2 is the ratio of the size of $S_1 \cap S_2$ to the size of $S_1 \cup S_2$,

$$J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \tag{3.1}$$

An example modified from Rajaraman and Ullman's book [1] can be used to explain Minhash. Assume there are four sets S_1, S_2, S_3, S_4 and they are all subsets of an universal set $\{1, 2, 3, 4, 5\}$. Since $S_1 = \{1, 4\}$, $S_2 = \{3\}$, $S_3 = \{2, 4, 5\}$ and $S_4 = \{1, 3, 4\}$, the four sets can be represented using a matrix:

Element	S_1	S_2	S_3	S_4	$x + 1 \bmod 5$	$3x + 1 \bmod 5$
1	1	0	0	1	1	1
2	0	0	1	0	2	4
3	0	1	0	1	3	2
4	1	0	1	1	4	0
5	0	0	1	0	0	3

Table 2.1: A matrix representing four sets (modified from the matrix representation on Rajaraman and Ullman's book, Chapter 3 [1])

The matrix represents sets in this way: if a set at column i contains the element at row j, the element on the matrix's ith column and jth row is set to 1. In another word, a set is represented as a column in the matrix representation.

A Minhash value of a set is defined as the number of the first row where a 1 appears in the column. In this example, the four sets have the following Minhash values: $h(S_1) = 1$, $h(S_2) = 3$, $h(S_3) = 2$ and $h(S_4) = 1$. If the rows of Table 2.1 is permutated, there becomes another matrix representation of the four sets as shown in Table 2.2

Element	S_1	S_2	S_3	S_4
4	1	0	1	1
5	0	0	1	0
1	1	0	0	1
2	0	0	1	0
3	0	1	0	1

Table 2.2: A permutation of the rows of 2.1 (modified from the matrix representation on Rajaraman and Ullman's book, Chapter 3[1])

Each set has another Minhash values according to Table 2.2: $h(S_1) = 4$, $h(S_2) = 3$, $h(S_3) = 4$ and $h(S_4) = 4$. Thus, if the rows are permutated for multiple times, a set will have multiple Minhash values.

So far, the way that a Minhash value is obtained seems arbitrary, but in fact it is not. Actually, the probability of the Minhash values of two sets being the same equals to the Jaccard similarity of the two sets [1].

Taking the sets S_1 and S_2 in Table 2.1 and Table 2.2 as an example. For each row of the matrix representation of sets, there are 3 possible situations: two 1s in the row (denote by X); a 1 and a 0 in the row (denote by Y); and two 0s in the rows (denote by Z). Situation z is not important because it do not affect either the Jaccard similarity or the probability of two hash values being the same. We assume there are x rows of situation X and y rows of situation Y. So the chance that $h(S_1)$ and $h(S_2)$ being the same is $x/(x+y)$. Now consider the Jaccard similarity of S_1 and S_2. The intersection of S_1 and S_2 has the size of x, the union of S_1 and S_2 has the size of $x+y$. So the Jaccard similarity is also $x/(x+y)$ [1].

In practice, the above permutation is simulated using a group of hash functions. Assume there are 5 rows in a matrix representation, for each row, a hash function calculates a hash value in a range of $[1, 5]$. Each hash value represents a permutated row number, in which way the permutations are simulated (examples in Figure 2.1). Also, instead of just using Minhash values, Minhash *signatures* are often generated by combining multiple Minhash values. The process is also known as *banding*.

To sum up, LSH is a powerful method in terms of finding approximate matches in real-time. However, in some situations, a more precise result is required instead of approximate matches. For these situations, pair-wise comparisons are still needed.

2.2 Indexing

Indexing techniques are developed to reduce the amount of redundant comparisons by only comparing potential record pairs that have common blocking key values (BKVs).

In traditional indexing, a BKV is the encoding of an attribute value. There are many encoding functions to choose from depending on the types of attributes. Phonetic encoding functions are the most commonly used for personal name en-

coding. Phonetic encoding functions encode strings based on their phonetic pronunciation. Thus, same encoding values are given to strings which are similar in pronunciation.

Attribute values with same encoding are grouped in a same block. Each attribute value of a record has a BKV, therefore a record have multiple BKVs. If two records have at least one BKV in common, they are considered as potential matches and are further compared using comparison functions. Common used comparison functions include edit distance comparison, Q-gram based comparison, Jaro and Winkler comparison, etc.[4]. Records have no BKVs in common will not be compared. Hence, number of redundant comparisons are largely reduced[3].

Traditional indexing consists of two phases: building and querying. At the building phase, all records in the building dataset are read and their corresponding BKVs are calculated. Attribute values and their BKVs are inserted into a Blocking Index(BI). A inverted index is also used to store the mapping between BKVs and their corresponding records' identifiers. At the querying phase, a query's BKVs are calculated and used to to locate the blocks that have the same BKVs. Records in the blocks are considered as potential matches and are compared one by one with the query. At last, the most similar records are returned as results.

2.2.1 Similarity-aware Inverted Indexing

Based on the traditional indexing, Christen (2008) proposed a improved indexing approach named Similarity-aware Inverted Indexing (SAI)[5] as a solution for real-time entity resolution. The basic idea of SAI is to pre-calculate similarities between attribute values at the building phase and store them in a similarity index (SI). By doing this, if a query's attribute values have appeared previously during building, similarities between the query's attribute values and other records' attribute values can be retrieved from the SI. Because real-world records often have frequent attribute values (e.g. names), SAI saves a lot of computational time.

Figure 2.2 and 2.3 show examples taken from Christen's paper[9]. Figure 2.2 shows some example of *Soundex* encodings. *Soundex* is a phonetic encoding function that generates 4-bit BKVs. Figure 2.3 illustrates the indexes structure of SAI, which has three indexes: Similarity Index (SI), Block index (BI) and Record

Record ID	Surname	Soundex encoding
r1	smith	s530
r2	miller	m460
r3	peter	p360
r4	myler	m460
r5	smyth	s530
r6	millar	m460
r7	smith	s530
r8	miller	m460

Figure 2.2: Examples of *Soudex* encoding, taken from Christen's paper[9]

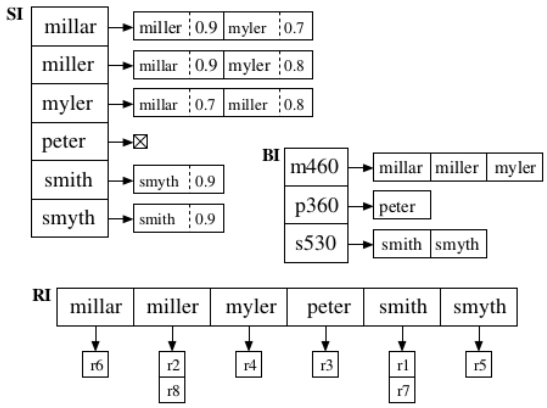


Figure 2.3: Similarity aware index structure, taken from Christen's paper[9]

identifier Index (RI). RI is like the inverted index used in traditional indexing but using the actual attribute values as keys. Attribute values are stored with their BKVs in blocks of the BI. Pre-calculated similarities in the building phase are inserted into SI.

The process of SAI can be explained using record r1 in 2.2 as an example. R1 has an attribute value *smith* (a common name in English language). The *Soundex* encoding for *smith* is s530. Assume r1 is processed during building phase, r1 is firstly inserted into RI using the attribute value *smith* as the key. The attribute value *smith* is then inserted into BI using its s530 as the key. Because there is another attribute values *smyth* in the block, the similarity between *smith* and *smyth* is then calculated and inserted into the SI as shown in Figure 2.3.

At the querying phase, when a record with attribute value *smith* is queried, its *Soundex* encoding s530 is firstly calculated. The attribute values *smith* and *smyth*are then retrieved because they are in the block s530. Next, the query value is compared with *smith* and *smyth* one by one. As the similarities are pre-calculated, they can be looked up in the SI. Finally, the attribute value with higher similarity(*smith* in this case) is identified and its record identifiers(r1 and r7) in RI are returned as candidate matches.

SAI is capable to handle queries on large datasets of 6 million records with an average query time of less than 0.1 second [9]. However, it is only suitable for static datasets because the its building phase separates with querying phase. In

order to address the problem, Ramadan [11] proposed a improved approach called DySimII that enables dynamic insertion. DySimII treats every query as a single record. If an attribute value of the query is not previously indexed, DySimII will insert the attribute value into the indexes. Therefore, if these attribute values reoccur in future queries, DySimII will be able to return results faster.

2.3 Research gap

As presented in the previous sections, LSH is a powerful real-time filter for approximate results. However, if a precise result is required, pair-wise comparisons need to be performed within LSH buckets. Another technique, SAI provides a choice to save computational time by pre-calculating similarities and limiting pair-wise comparisons to records with same BKVs. Nevertheless, if some BKVs frequently appear in a dataset, there will be very large blocks which will still result in excessive pair-wise comparisons. To solve the problems, we consider to combine LSH with SAI. In short, we firstly use LSH to filter the data records and eliminate the record pairs that are obviously different, and then we perform pair-wise comparisons on the filtered data records based on the idea of SAI.

As mentioned in 2.2.1, Ramadan [11] proposed an improved *dynamic similarity-aware inverted indexing approach* (DySimII) approach for real-time entity resolution in dynamic environment. Considering most real-world applications of entity resolution are dynamic, in the pair-wise comparison stage, we adopt Ramadan's DySimII instead of static SAI.

Chapter 3

Proposed Approach

3.1 A Two-stage Similarity-aware Indexing Approach

We name our proposed approach as the *two-stage similairty-aware indexing* because it involves two stage: the LSH based "hashing" stage for rough filtering and the SAI based indexing stage for pair-wise comparisons.

3.1.1 First stage: Locality Sensitive Hashing

At this stage, LSH is implemented using *Minhashing* (described in Section 2.1), which filters records based on simulation of Jaccard similarity.

Assuming two sets S_1 and S_2, Minhashing uses hash functions to generate a group of hash values for each set. It can be proved that the probability that S_1's minimum hash value equals to S_2's minimum hash value, $Pr(h_{min}(S_1) = h_{min}(S_2))$ is the same to the two sets' Jaccard similarity $J(S_1, S_2)$. So $Pr(h_{min}(S_1) = h_{min}(S_2))$ can be used as a estimator of $J(S_1, S_2)$. The estimator is unbiased and its variance can be reduced by taking more hashes.

Minhashing can also be used to process data records. An example is given in Table 3.1 where similar record pairs are marked with same colour. Similar record pairs have higher Jaccard similarities than dissimilar record pairs. For example, considering each attribute value as an element, the Jaccard similarity of r1 and r3 is 3/5 because they have 3 elements in their intersection set and 5 elements

in their union set. Thus, r1 and r3 have a probability of 0.6 of having the same minimum hash values.

Record ID	First name	Last name	Suburb	Postcode
r1	smith	bryant	turner	2612
r2	kristine	jones	city	2601
r3	smyth	bryant	turner	2612
r4	zach	williams	kingston	2604
r5	christine	jones	city	2601
r6	christy	greg	belconnen	2617
r7	zack	williams	kingston	2604
r8	robert	brown	city	2601

Table 3.1: Example records

In the first stage of the two-stage approach, records are processed by Minhashing and each of them is assigned with a group of Minhash values. In the example of Table 3.2 where four hash functions are used, each record has four Minhash values and all similar record pairs in Table 3.2 has some common Minhash values. This is because record pairs with high Jaccard similarity are likely to have same Minhash values.

Record ID	Minhash value 1	Minhash value 2	Minhash value 3	Minhash value 4
r1	993044549	442796032	5223429	468164869
r2	455172161	1055368516	52526405	333606145
r3	320704576	595939653	5223429	468164869
r4	51768384	133030212	99830085	199047425
r5	455172161	286173441	52526405	333606145
r6	77824325	439316548	170786113	168862981
r7	51768384	133030212	123482433	199047425
r8	455172161	1205302533	76178753	333606145

Table 3.2: Hash values

However, having common Minhash values does not necessarily mean two records

are similar. In situations where some attribute values are frequent, dissimilar records will have common Minhash values too. In Table 3.2, r8 has the same Minhash values with r2 and r5 because they have the same attribute values of suburb and postcode, but they are obviously dissimilar. Therefore, a technique called *banding* is introduced to enable a rigid filtering.

Record ID	Signature 1	Signature 2
r1	993044549442796032	5223429468164869
r2	4551721611055368516	52526405333606145
r3	320704576595939653	5223429468164869
r4	51768384133030212	99830085199047425
r5	455172161286173441	52526405333606145
r6	77824325439316548	170786113168862981
r7	51768384133030212	123482433199047425
r8	4551721611205302533	76178753333606145

Table 3.3: Signatures

Banding tunes the strictness of Minhash filtering by combining multiple Minhash values to form a *band*. The Minhash values in a band are combined to form a Minhash signature. In another word, banding enhances the filtering strictness by applying a logic "AND" on Minhash values. The number of Minhash values in a band is known as bits. For instance, banding the Minhash values in Table 3.2 with two bits in a band produces Minhash signatures as shown in Table 3.3. We can see by applying banding, the common signatures of r2 and r5 are preserved, while the noisy record r8 is eliminated because it does not have common Minhash signatures with r2 and r5 any more.

The final step of the first stage is to store the records' identifiers and their Minhash signatures into an index named LSH index (LI) (Figure 3.1). By doing this, When a record is queried, the query's Minhash signatures can be found by processing it with Minhashing. Therefore, the records have the same Minhash signatures can be quickly found by looking up the LSH index.

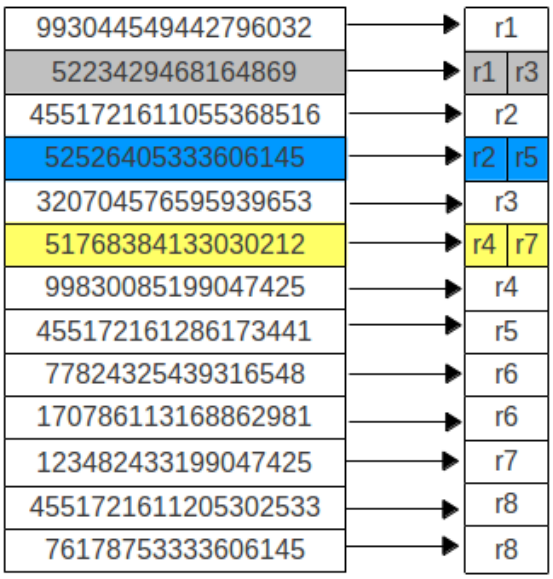


Figure 3.1: LSH index

3.1.2 Second stage: Similarity-aware indexing

At the second stage, we adopt the Dynamic Similarity-aware Inverted Indexing (DySimII)[11] described in Chapter 2 to conduct pair-wise comparisons.

As mentioned in Section 3.1.1, for an input query, we can obtain its candidate records by looking up its Minhash signature in LI. In order to obtain a precise result, we need to compare the query record with each of the candidate records. The pair-wise comparisons are done by comparing the two records' attribute values accordingly. Because there are millions of data records to compare, the pair-wise comparisons will be computationally expensive if they are done online. However, in real-world situations, same attribute values may appear frequently, examples including city names, postcodes and common personal names. DySimII[11] pre-calculates the similarities of attribute values in a dynamic environment, in which way the similarities of attribute values can be retrieved from the Similarity Index (SI) and therefore saves a lot of computational time.

At this stage, an input record is firstly processed using encoding techniques. Each attribute of the record generates a encoding value, or so called BKV. An attribute value is stored in an index call Blocking Index (BI) under its BKV (Figure 3.2). The use of BI further reduces number of comparisons because an attribute value is only compared with other attribute values that have the same

Record ID	First name	Double-Metaphone
r1	smith	sm0
r2	kristine	krst
r3	smyth	sm0
r4	zach	sk
r5	christine	krst
r6	christy	krst
r7	zack	sk
r8	robert	rprt

Table 3.4: Figure x, the

BKV (they are put in the same block in BI).

After being blocked, an attribute value is then compared with other attribute values in the same block. The comparisons is conducted via calculating each pair's similarity using comparison functions. The calculated similarities are then stored in an index call Similarity Index (SI) together with the corresponding attribute values for further use (Figure 3.2).

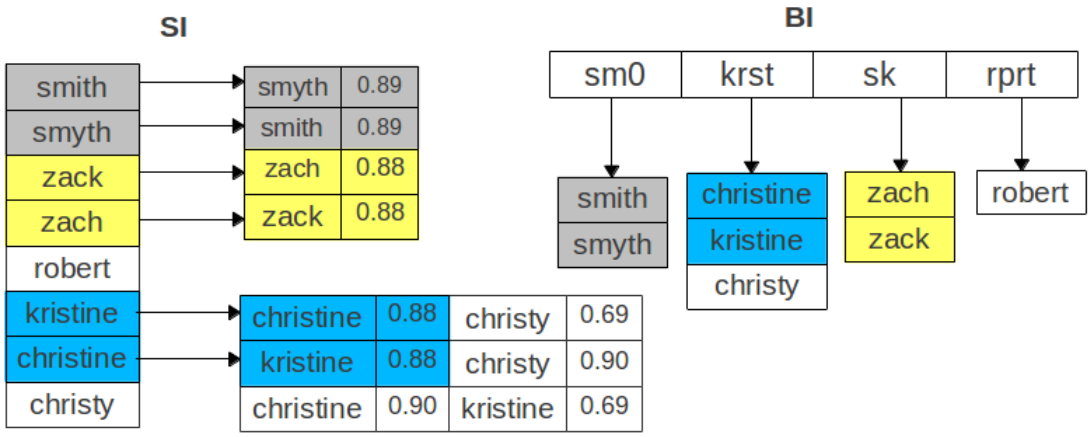


Figure 3.2: Block index and similarity index

The attribute "First name" in Figure 3.1 is used to illustrate the process. Attribute values that have the same Double-Metaphone encoding are put in the same block in the BI. So *smith* and *smyth* are put in the block of key *sm0*; *christine*, *kristine* and *christy* are put in the block of key *krst*; *zach* and *zack* are put in the

block of key *sk*. Each attribute values is compared with others in the same block using Twinkler comparison function. So *smith* is compared with *smyth*, *zach* is compared with *zack*, *christine*, *kristine* and *christy* are compared with each other. The calculated similarities are then stored in the SI. After all the attribute values are processed, we get the SI and BI as shown in Figure 3.2.

3.2 Real-time Entity Resolution

The two-stage approach introduced three indexes: LSH Index (LI), Block Index (BI) and Similarity Index (SI). This section describe how the three indexes are connected and used while being applied to real-time entity resolution. As mentioned in Chapter 2, indexing techniques often have into two phases: building phase and querying phase. The two phases of the two-stage approach are described separately in the following sections

3.2.1 Building phase

Algorithm 1 briefly describes the building process. In the beginning, all the three indexes are empty. Records are then inserted into indexes one by one. While inserting a record to the indexes, its Minhash signatures are firstly calculated. Each signature points to a unique "bucket" in the LI. The record's identifier is then inserted into every bucket that its signatures point to. After that, a BKV is calculated for each attribute value of the inserted record. The attribute values are then added into the inverted index BI using the BKVs as keys. At last, each attribute value of the inserted record is compared with other attribute values in the same "block" with it. The similarities of each comparison pairs are then stored in the third index, SI.

Algorithm 1: ***Build phase***

input : input dataset **D**; number of attributes n; Minhash function **H**; encoding functions $\mathbf{E}_i$ and similarity functions $\mathbf{S}_i$ for $i = 1...n$;
output: Indexes **SI**, **BI** and **LI**

```
Initialise SI = {}
Initialise BI = {}
Initialise LI = {}
for r ∈ D do
    Sig = H(r)
    Insert(r, n, E_i, S_i, Sig, SI, BI, LI)
```

The most important part of the building phase is insertion. The procedure of insertion is shown in Algorithm 3.3a. The insertion function takes in a record **r** whose first attribute **r**.0 is the identifier. The insertion starts by inserting **r**.0 into the buckets in the LI where the Minhash signatures point to. After that, every attribute values for **r** is checked and if a attribute value **r**.i is not indexed previously, it will be added into SI and BI. Assume **r**.i is not previously indexed, the inserting processing will firstly compute its encoding value c, and add **r**.i into BI using the c as BKV. The block list **b** of the BKV c is then retrieved. Each attribute value in **b** is denoted as v. The similarity (denoted as s)of each $(\mathbf{r}.i, v)$ pair is then calculated using comparison function **S**. In the SI, each attribute value has a list that stores its similarity with other attribute values. So we initialise a similarity list (si) **r**.i and add similarity pairs into it in the form of tuples (v, s). For each of the other attribute values v, we retrieve its similarity oi and add $(\mathbf{r}.i, s)$ to oi. Finally, the updated indexes SI, BI and LI are returned as result.

3.2.2 Querying Phase

The querying phase is briefly described in Algorithm 3.3b using pseudo-code. For the purpose of dynamic indexing, in the querying phase, every query is treated as a record. If the query record is not detected in the indexes, it is viewed as a new record and will be inserted into indexes. Hence, the insertion function used in the building phase is also used during querying. We denote a query record by **q**. Just like the building phase, t**q** is firstly processed using Minhashing to generate its Minhash signatures (denoted as **Sig**). Records "hashed" into the same Minhash signatures are obtained from LI, and are considered as candidate records. Next, the

```
input : Record r number of attributes
        n; Minhash signatures Sig;
        encoding functions E_i and
        similarity functions S_i for
        i = 1...n; indexes SI, BI and LI.
output: Updated indexes SI, BI and LI
for sig ∈ Sig do
    bk = LI[sig]
    Append r.0 to bk
    LI[sig] = bk
for i = 1...n do
    if r.i ∉ SI then
        c = E_i(r.i)
        b = BI[c]
        Append b.i to b
        BI[c] = b
        Initialize si = []
        for v ∈ b do
            s = S_i(r_i, v)
            Append (v, s) to si
            oi = SI[v]
            Append (r.i, s) to oi
            SI[v] = oi
        SI[r.i] = si
```

(a) Algorithm: ***Insert***

```
input : Dataset D query record q
        number of attributes n; Minhash
        function H; encoding functions
        E_i and similarity functions S_i for
        i = 1...n; indexes SI, BI and LI.
output: Ranked match list M
Initialise M = []
Sig = H(q)
for sig ∈ Sig do
    bk = LI[sig]
    for r.0 ∈ bk do
        if r.0 ∉ M then
            Retrieve r from D using r.0
            Initialise s = 0
            for i = 1...n do
                if q.i ∉ SI then
                    Insert(q, n, E_i, S_i,
                    Sig, SI, BI, LI)
                sl = SI[q.i]
                s = s + sl[r.i]
            Append (r.0, s) to M
        Sort M according to similarity
```

(b) Algorithm: ***Query phase***

Figure 3.3: two algorithms

query **q** is compared to every candidate records. The comparison is done through comparing the attribute values of **q** and **r**. Since the similarities of previously appeared attribute values are pre-calculated and stored in SI, they can be simply retrieved from SI. The similarity of two records are calculated by summing the similarities of the two records' attribute values. After similarities of all candidate pairs are obtained, the candidate pairs can be ranked according to their similarity values.

Up to here, there are two ways to brief the result. Firstly, a similarity threshold can be set that only the pairs that have similarities higher than the threshold are returned as query result. Secondly, a result size k can be set that the pairs with the top k similarities are returned as result.

However, there are cases that the attribute values of **q** is not previously indexed and cannot be found in SI. For such situation, **q** is treated as a whole new record and the insertion function is called to insert it into indexes just like the insertion

in building phase. After the query record is inserted, the query record is queried again and this time a query result will be returned.

Chapter 4

Experimental Evaluation

4.1 Experiment Setup

4.1.1 Dataset

The two-stage approach is evaluated using two large scale datasets: the North Carolina Voter Registration Dataset, and the Australian Telephone Directory Dataset.

- **Dataset 1**. The North Carolina Voter Registration Dataset, which is downloaded between October 2011 and December 2012 from a real-world voter registration database of North -Carolina, USA. It has records of 2,567,642 individuals. 263,974 of them have two record, 15,093 of them have three records and 662 of them have four records. Each record has four attributes: first name, last name, city, and postcode. Through the dataset examination three main reasons are identified as the causes of individuals have multiple records: nicknames and typos in the first name attribute, surname changes due to martial status, and address change due to moving addresses.

- **Dataset 2**. The Australian Telephone Directory Dataset, which is a modified dataset based on an Australian telephone directory from 2002 (Australia On Disc). It contains four attributes: first name, last name, suburb and postcode. It has two subsets with 25% overlapped records, each subset has at most one attribute value modification. The dataset is modified to evaluate entity resolution techniques' robustness to noisy datasets which contains errors and variations in attribute values.

4.1.2 Comparison techniques

- **LSH**. This is the typical Locality Sensitive Hashing approach implemented using Minhashing. Candidate matches are retrieved based on their frequencies of appearance. It contains a LSH index which stores Minhash signatures and identifiers of records that have the signatures. When a query is input, LSH calculates its Minhash signatures and retrieves identifiers of records that have the same signatures. The record identifiers' appearance are counted and those that appears the most are returned as results.

- **DySimII**. This is the Dynamic Similarity-aware Indexing approach for real-time entity resolution discussed in Chapter 2. It pre-calculated the similarity of each record value pairs of the same encoding block to decrease the number comparisons in real time query. It also enables dynamic indexing by allowing insertion during querying phase.

4.1.3 Evaluation metrics

To evaluate the performance of a record resolution approach, we introduce four categories[4] to classify record pairs: *a*) True positives (TP). These are the correctly matched pairs in the query results. *b*) False positives (FP). These are the incorrectly matched pairs (they not true matches) in the query results. *c*) True negative (TN). These are the non-matches pairs that are correctly classified as non-matches. *d*) False negative (FN). These are the true matches pairs that are incorrectly classified as non-matches.

- **Processing time**. It is measured for both the building phase and querying phase. Average query time is the most concerned as it represents the real-time query performance.

- **Memory usage**. It is measured for both the building phase and querying phase.

- **Recall**. It measures the ability of the entity resolution technique to find true matches. It is calculated as $\frac{TP}{TP+NP}$.

- **Precision**. It measures the quality query results. It is calculated as $\frac{TP}{TP+FP}$.

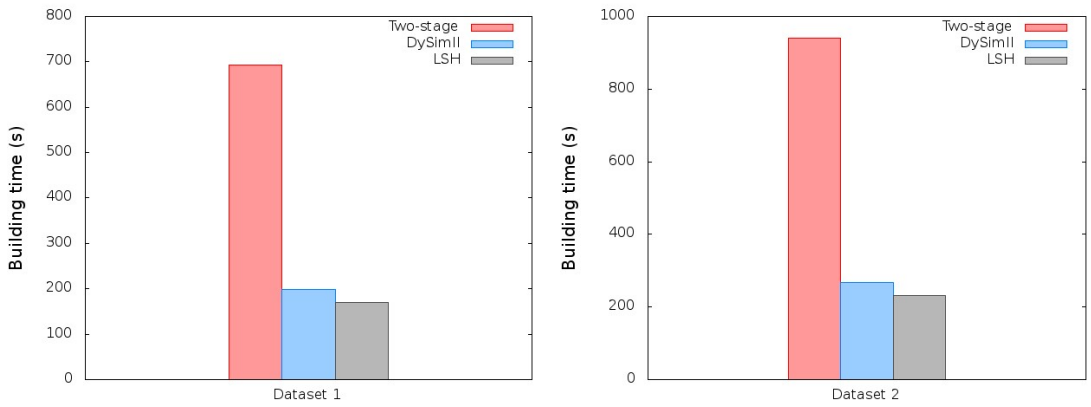
4.1.4 Parameter setting

To simulate an intensive query environment, 50% data records of each dataset is used as query records and the other 50% are used for building the indexes. Number of hash functions and number of bits in each band are crucial parameters as they tune the probability of similar records being "hashed" into same buckets. We tested the parameters on many combinations of hash functions and bits. However, because each test on the whole datasets runs about 3 hours and there are in total more than 500 potential combinations, we firstly tested the parameters on small datasets and picked parameters that generate good results to do a second test on the whole dataset. At last, we choose to use 60 hash functions with 4 bits in each band for both LSH and the two-stage approach. The top 10 candidate matches are used to calculate recall and precision.

The experiments are run on a 64-bit server with a 24-Core Intel Xeon 2.40G CPU and 132GB RAM.

4.2 Experimental Results and Discussion

building time



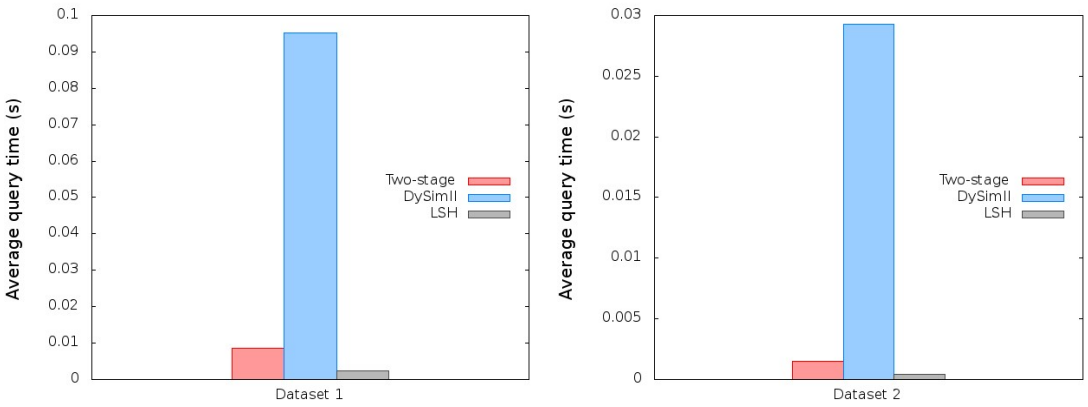
(a) Building time on Dataset 1 (b) Building time on Dataset 2

Figure 4.1: Building time

The building time comparisons are shown in Figure 4.1. As expected, the two-stage approach takes much longer time to build indexes, and the building

time increases when dataset gets larger. The two-stage approach takes around 15 minutes to build indexes for Dataset 2, which is 3 times longer than DySimII and 5 times longer than LSH. The main reason is that besides SI and BI, the two-stage approach introduced a LSH Index (LI) which needs to be built during building phase too. The experiment setting for LSH is 60 hash functions with 4 bits in a band, which means each record is "hashed" into 15 buckets in LI. This makes the LI the largest indexes and thus takes much more time to build. Considering building is done off-line, the increase in building time is acceptable.

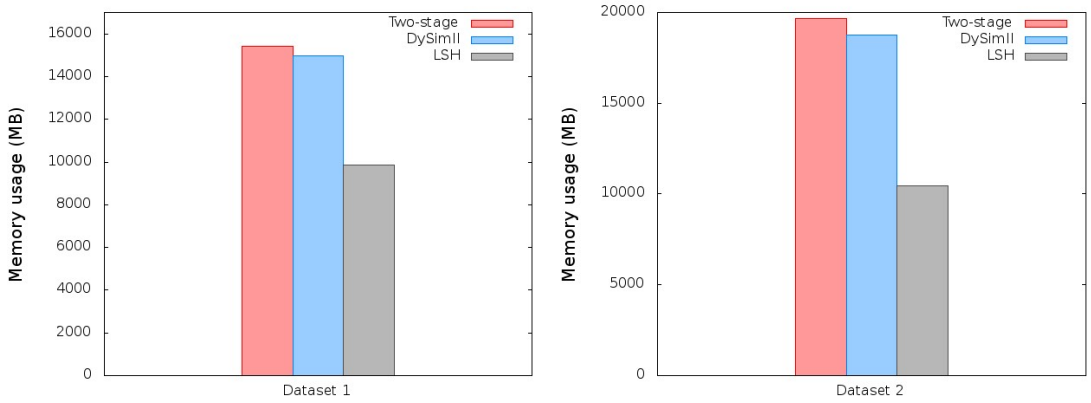
query time



(a) Average query time on Dataset 1 (b) Average query time on Dataset 2

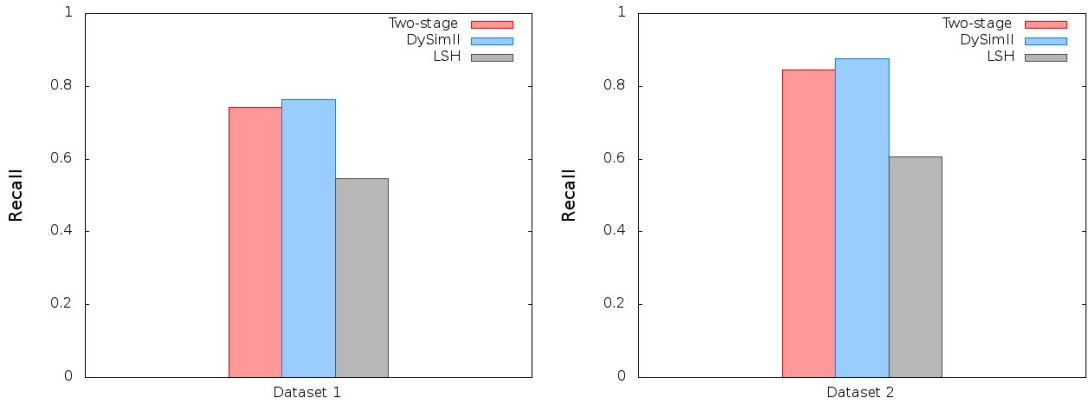
Figure 4.2: Average query time

Average query time is a crucial measurement in the evaluation as it refers to the performance of entity resolution. The results are shown in Figure 4.2. The two-stage approach performs surprisingly in this aspect: it achieves average query time below 0.01 seconds for Dataset 1 and below 0.005 seconds for Dataset 2, which are 10 times faster than DySimII. Also, as expected, the average query time of the two-stage approach is slightly longer than that of the LSH. This is because the LSH only gives a approximate result for queries and does not involve pair-wise comparisons.



(a) Memory usage on dataset 1 (b) Memory usage on dataset 2

Figure 4.3: Memory usage

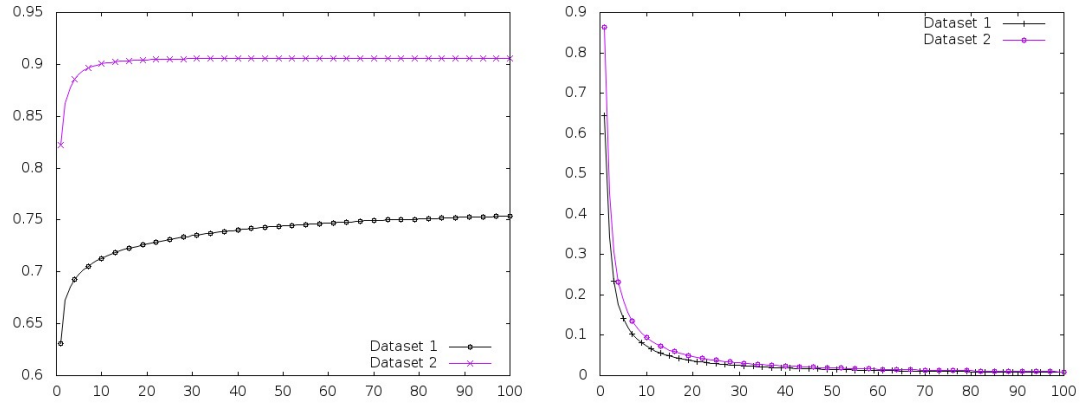


(a) Recall on dataset 1 (b) Recall on dataset 2

Figure 4.4: Recall

Memory usage

In most cases, query time and memory usage are trade-offs: faster query often leads to more memory usage. In this case, although the two-stage approach largely improves average query time, it only uses a little more memory than DySimII. The main reason is that the number of attribute values indexed in SI and BI have been reduced because only the attribute values of records that have same Minhash signatures are inserted into BI and SI.



(a) Recall of two-stage approach

(b) Precision of two-stage approach

Figure 4.5: Recall and precision change on number of returned results

Matching quality

The query matching recall results are shown in Figure 4.4. As can be seen, for both datasets, the two-stage approach's matching recall is slightly lower than that of DySimII but much higher than that of the LSH. This result is what was expected, as the two-stage approach uses Minhashing to filter out dissimilar records, true matches may also be filtered out. However, considering the improvement in query time, the loss is insignificant.

The change of matching quality according to the number of returned results is shown in Figure 4.5. It can be seen from the figure that the two-stage approach achieves high recall at early stage when few results are returned. That is to say, the two-stage approach is able to provide good matching quality for applications where high prevision is preferred.

Chapter 5

Conclusion and Future Work

In this report, a two stage similarity-aware indexing approach has been presented as a solution for real-time large scale entity resolution. This approach is evaluated experimentally on two large scale datasets taken from real-world databases. According to the experimental results, the two-stage approach can process a query within 0.01 seconds on a large dataset of more than two million records. Without obviously increase of memory usage, the two-stage approach processes queries 10 times faster than the original similarity-aware indexing. This approach can also perform well at scenarios where precise query results are needed. However, building the indexes may take 3 times longer.

Nevertheless, similarity-aware indexing requires to store pre-calculated similarities in memory, which consumes a large proportion of memory as time goes on. Improving upon the memory consumption drawback by adopting other indexing techniques such as sorted neighbourhood indexing is one of the future research directions of this approach. Additionally, exploring the possibility of applying this approach to other application areas such as real-time recommender system is another track for future work.

Bibliography

[1] Rajaraman Anand and Jeffrey David Ullman. *Mining of massive datasets.* Cambridge University Press, 2011.

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

[3] Peter Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. Z, NO. Y, ZZZZ 2011*, 2011.

[4] Peter Christen. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer, 2012.

[5] Peter Christen and Ross Gayler. Towards scalable real-time entity resolution using a similarity-aware inverted index approach. *AusDM '08 Proceedings of the 7th Australasian Data Mining Conference*, 2008.

[6] Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. Fast locality-sensitive hashing. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1073–1081. ACM, 2011.

[7] M. A. Hernandez and S. J. Stolfo. The merge/purge problem for large databases. *ACM SIGMOD95, San Jose*, 1995.

[8] Lei Li, Dingding Wang, Tao Li, Daniel Knox, and Balaji Padmanabhan. Scene: a scalable two-stage personalized news recommendation system. In *ACM Conference on Information Retrieval (SIGIR)*, 2011.

[9] Ross Gayler Peter Christen and David Hawking. Similarity-aware indexing for real-time entity resolution. *ANU Computer Science Technical Report Series*, 2009.

[10] P. Christen R. Baxter and T. Churches. A comparison of fast blocking methods for record linkage. *ACM SIGKDD Workshop on Data Cleaning, Record Linkage and Object Consolidation, pages 2527, Washington DC*, 2003.

[11] Banda Ramadan, Huizhi Liang Peter Christen, and David Ross Gayler, Hawking. Dynamic similarity-aware inverted indexing for real-time entity resolution.

[12] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *Signal Processing Magazine, IEEE*, 25(2):128–131, 2008.

[13] S. Yan, D. Lee, M. Y. Kan, and L. C. Giles. Adaptive sorted neigh-borhood methods for efficient record linkage. *ACM/IEEE-CS joint conference on Digital Libraries*, 2007.