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

Shouheng Li[1] **Huizhi Liang**[2] **Banda Ramadan**[3]

Research School of Computer Science, College of Engineering Computer Science
Australian National University,
Canberra ACT 0200,
[1]Email: sohey33@gmail.com

[2]Email: huizhi.liang@anu.edu.au

[3]Email: banda.ramadan@anu.edu.au

Abstract

Entity resolution is the process of identifying records in one or multiple data sources that represent the same real-world entity. How to find all the records that belong to the same entity as the query record in real-time brings challenges to existing entity resolution approaches. The challenge is especially true for large-scale dataset. In this paper, we propose to use a two-stage similarity-aware indexing approach for large-scale real-time entity resolution. In the first stage, we use locality sensitive hashing to filter out records with low similarities for the purpose of decreasing the number of comparisons. Then, in the second stage, we pre-calculate the comparison similarities of the attribute values to further decrease the query time. The experiments conducted on a large-scale dataset with over 2 million records shows the effectiveness of the proposed approach.

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

1 Introduction

With the utilisation of databases and information systems, businesses, governments and organisations are able to collect massive information without much difficulty. 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 can help to reduce the noise in data and improve data quality (Elmagarmid et al. 2007). Currently, most available entity resolution techniques conduct the resolution process in offline or batch mode, while the dataset is usually static (Christen 2012). However, in real world scenarios, many applications require rapid real-time responses. For example, online entity resolution based on personal identity information can help a bank to identify fraudulent credit card applications (Christen 2012). The requirement of dealing with large-scale dynamic data and providing rapid responses brings challenges to current entity resolution techniques.

Typically, pair-wise comparison is used to find the records that belong to the same entity. However, the number of comparisons increases dramatically when the size of the dataset grows quickly. Blocking or canopy formation can help to significantly decrease the number of comparisons (Christen 2012). Blocking divides the data into blocks and only compares the query record with all other records within the same block. For example, Soundex and Double-Metaphone are commonly used blocking approaches (Christen 2012). Moreover, another technique called Locality Sensitive Hashing (LSH) can find approximate similarity records quickly via hashing. It provides a similarity based filtering that "hashes" similar data records together.

Recently, Christen et al. (2009) proposed an Similarity-aware Indexing technique that improves the performance of traditional indexing by pre-calculating similarities of attribute values. Nevertheless, this approach needs to compare every record that has one or more encoding values that are the same with the query record, even though the compared record has a low overall similarity with the query record. If we can filter out those low similarity records first, then we can reduce the number of comparisons to speed up the query time. In this paper, we propose a two-stage similarity-aware indexing approach. At the first stage, locality sensitive hashing is adopted to approximately filter out those data records that have low similarity with the query record and allocate the potential matches in the same block. At the second stage, similarity-aware indexing is used to compare potential matches to obtain a precise and accurate result.

The rest of the paper is organized as below. Firstly, the related work will be briefly reviewed in Section 2. Then, the proposed approaches will be discussed in Section 3. In this section, the two-stage similarity-aware indexing approach is presented. In Sections 4 and 5, the design of the experiments, experimental results, and discussions will be presented. Finally, the conclusions of this work will be given in Section 6.

2 Related Work

The purpose of entity resolution is to find records in one or several databases that belong to the same real-world entity. Such an entity can be a person (e.g. cus-

tomer, patient or student), a product, a business, or any other object that exists in the real world. Entity resolution aims to link different databases together (in which case it is known as Record Linkage or Data Matching) and can also identify duplicate records in one database (known as De-duplication) (Christen 2012). Entity resolution is widely used in various applications such as identity crime detection, estimation of census population statistics, and retrospective construction of samples of persons for health research (Elmagarmid et al. 2007, Christen 2012). Currently, most available entity resolution techniques conduct the resolution process in offline or batch mode. Only limited research into using entity resolution at query time (Lange & Naumann 2012) or in real-time (Christen et al. 2009, Ramadan et al. 2013) has been conducted.

Indexing techniques can help to scale-up the entity resolution process. Commonly used indexing approaches include standard blocking based on inverted indexing and phonetic encoding, q-gram indexing, suffix array based indexing, sorted neighborhood, multi-dimensional mapping, and canopy clustering (Christen 2012). Typically, these existing approaches index one or more attribute values manually selected based on expert domain knowledge. Some recent research has proposed automatic blocking mechanisms (Das Sarma et al. 2012) or learning algorithms to find the best blocking schemes for record linkage (Michelson & Knoblock 2006).

Christen et al. (2009) proposed a similarity-aware indexing approach for real-time entity resolution. However, this approach fails to work well for large-scale databases, as the number of similarity comparisons for new attribute values increases significantly as the sizes of blocks increases with the growing number of records. Recently, Ramadan et al. (2013) extended this approach to facilitate a dynamic approach to index whereby the index is extended as a database grows. They showed that the insertion of new records into the index, as well as querying the index for real-time entity resolution grows sub-linearly as the size of the database index grows. However, this approach still requires a large number of pair-wise comparisons as it compares every record that has one or more encoding values that are the same with the query record, while the comparisons of records has a low overall similarity with the query record are not necessary.

Locality Sensitive Hashing (LSH) can help to return approximate similar records of a query quickly. Such approaches are widely used in Nearest Neighbor and Similarity Search in applications such as image search (Dong et al. 2008), recommender systems (Li et al. 2011), and entity resolution (Kim & Lee 2010). More recently, the work of Gan et al. (2012) proposed to use a hash function base with n basic length-1 signatures rather than using fixed l length-k signatures or a forest to represent each record. The records that are frequently colliding with a query record across all the signatures are selected as the approximate similarity search results. However, this approach needs to scan all the data records in the blocks to get the frequently colliding records each time, which results in the difficulty of returning results quickly when the sizes of the blocks are big for large-scale datasets. This approach can be used in real-time scenario, if we use the dynamic collision counting method for length-k ($k > 1$) blocks.

Although both LSH and indexing approaches (e.g., Similarity-aware Indexing(Christen et al. 2009, Ramadan et al. 2013)) can be effectively used in real-time entity resolution, how to combine them together to facilitate large-scale real-time entity resolution still needs to be explored.

Record ID	Entity ID	First name	Last name	Suburb	Zipcode
r_1	e_1	halle	bryant	turner	2612
r_2	e_2	kristine	jones	city	2601
r_3	e_3	hailey	pitt	turner	2612
r_4	e_4	christy	greg	belconnen	2617
r_5	e_2	christine	jones	city	2601

Table 1: Example records, r_5 is the query record

Record ID	h_1	h_2	h_3	h_4
r_1	1	2	3	4
r_2	5	6	7	8
r_3	9	10	3	11
r_4	12	13	14	15
r_5	5	16	7	8

Table 2: Length-1 minHash signatures, generated using four minHash functions h_1, h_2, h_3 and h_4

3 A Two-stage Similarity-aware Indexing Approach

We propose a two-stage similarity-aware indexing approach called LSI. LSI includes three indexes: a locality sensitive hashing index named LI, a similarity index named SI, and a standard blocking index named BI. At the first stage, we use the locality sensitive hashing index LI to filter out records with low similarities with the query records. Records with high similarities are preserved and stored in same blocks in the LI. Then at the second stage, records in the same block in the LI are considered as candidate matches and are compared pair-wisely. Candidate matches' attribute values are grouped together using encoding techniques and stored in the blocking index BI. A attribute value is compared with other attribute values which have the same encoding value, their similarities are pre-calculated and stored in the similarity index SI. The proposed approach performs well in real-time entity resolution scenarios for two reasons: firstly, comparisons on record pairs with low-similarities are avoided; secondly, most similarities are pre-calculated and can be retrieved from the SI therefore time of comparisons are saved. The proposed indexing approach LSI will be discussed in details in Section 3.1 and 3.2. Then in Section 3.3, we will discuss how the proposed indexing approach LSI can be applied in real-time entity resolution.

[Example 1] Table 1 shows an example of five records r_1, r_2 , r_3, r_4, and r_5 with four attributes: First Name, Last Name, Suburb, and Zipcode. They belong to four different entities e_1, e_2, e_3 and e_4. Suppose r_5 is a query record, the entity resolution process for r_5 is to find r_2 based on the four attribute values.

3.1 First stage: Locality Sensitive Hashing

A locality sensitive hashing family can help to find approximate results. Let h denote a hash function for a given distance measure approach D, $Pr(x)$ denote the probability of an event i, p_1 and p_2 are two probability values, $p_1 > p_2$, $0 \leq p_1, p_2 \leq 1$. h is called (d_1, d_2, p_1, p_2)-sensitive for D, if for any two records r_x and r_y, the following conditions hold:

1. if $D(r_x, r_y) \leq d_1$ then $Pr(h(r_x) = h(r_y)) \geq p_1$

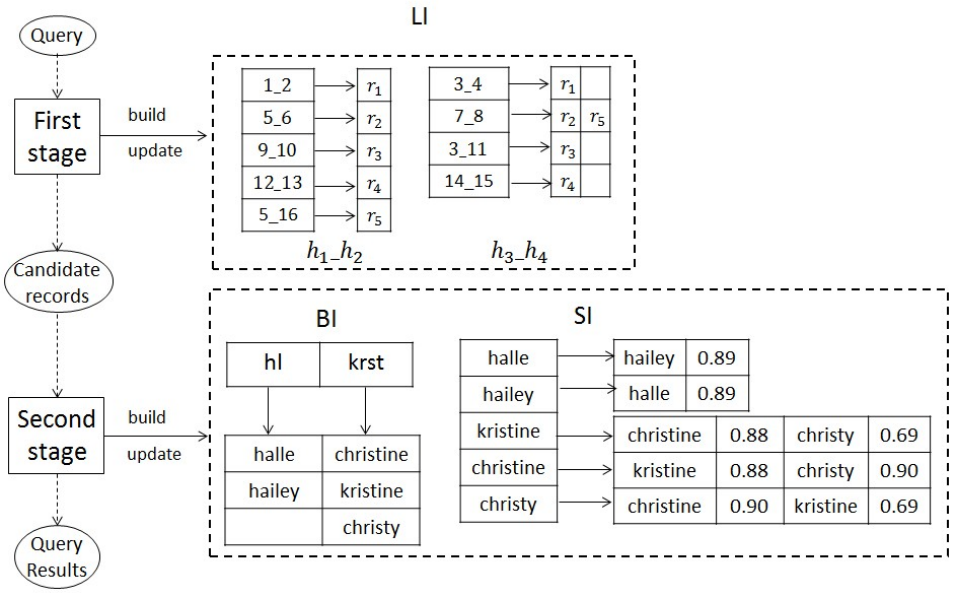

Figure 1: An example LSI index for the example records in Table 1

Record ID	First name	Encoding
r_1	halle	hl
r_2	kristine	krst
r_3	hailey	hl
r_4	christy	krst
r_5	christine	krst

Table 3: Double-Metaphone Encoding of the First name attribute values in Table 1

2. if $D(r_x, r_y) > d_2$ then $Pr(h(r_x) = h(r_y)) \leq p_2$

Minwise hashing (minHash) is a popularly used locality sensitive hashing approach that estimates the Jaccard similarity. Let $J(r_x, r_y)$ denote the Jaccard similarity for any two records r_x and r_y. The minHash method applies a random permutation π on the attribute values (i.e., elements) of any two records (i.e., sets) r_x and r_y and utilizes

$$p = Pr(min(\pi(r_x)) = min(\pi(r_y))) = J(r_x, r_y) = \frac{r_x \cap r_y}{r_x \cup r_y} \quad (1)$$

to estimate the Jaccard similarity (Gionis et al. 1999) of r_x and r_y (Gionis et al. 1999), where $min(\pi(r_x))$ denotes the minimum value of the random permutation of the attribute values of record r_x. p denotes the hash collision probability $Pr(min(\pi(r_x)) = min(\pi(r_y)))$. It represents the ratio of the size of the intersection of the attribute values of the two records to that of the union of the attribute values of the two records. To avoid unnecessary comparisons with low similarity records and secure the response time for query records in real-time, we use locality sensitive hashing to filter out those low similarity records at the first stage.

Each record is assigned with a set of minHash signatures via a family of minHash functions. For example, Table 2 shows the minHash signatures of the example records in Table 1. In the example of Table 2, four hash functions h_1, h_2, h_3, h_4 are used. They generate length-1 signatures for each record. However, having common length-1 minHash values does not necessarily mean two records are similar. In situations where some attribute values are frequent, dissimilar records usually have common length-1 minHash values too, which means most records will be assigned with same length-1 minHash values and minHashing does not really narrow down the range of candidate records.

[Example 2] (Length-1 minHash signatures). Table 2 shows the length-1 minHash values of the example records given in Table 1. Records that have common attribute values are likely to have same length-1 minHash values, e.g., r_1 and r_3 have the same minHash value 3. However, although r_1 and r_3 have same length-1 minHash signatures, they actually belong to different entities.

Therefore, in order to filter out the noisy matches, a technique called *banding* is introduced to enable a rigid filtering. Banding tunes the strictness of minHash filtering by combining multiple minHash functions to form a *band*. The minHash values generated by minHash functions in a band are combined to form a minHash signature. 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 denoted as k. Together with the And-construction, Or-constructions are conducted to increase the collision probability. More Or-constructions will introduce more hash tables denoted as l.

At the first stage, for each record, k hash functions are used to generates a length-k signature through And-construction. To increase the collision probability, each record is hashed l times to conduct Or-construction and form l hash tables (i.e., l length-k signatures), $n = k \times l$. Each record is indexed by l length-k minHash signatures.

After we get the minHash signature, we store the records' identifiers and their minHash signatures into an index named Locality Sensitive Hashing Index (i.e., LI). For a query record, the records with the same minHash signatures, also known as candidate records, can be quickly found by looking up the LI index.

[Example 3] (LI: Locality Sensitive Hashing Index). Left figure in Figure 1 shows the locality sensitive hashing index of the example records in Table 1 using the parameters $k = 2$ and $l = 2$. r_2 and r_5 are put in the same block in the LI with minHash signature 7_8.

After we get the candidate records in each locality sensitive hashing block, we need to compare the similarity of each candidate record with the query record. The pre-calculation of the similarity of two attribute values can help to avoid large number of comparisons in real-time. This will be discussed in Section 3.2.

3.2 Second stage: Similarity-aware indexing

At the second stage, we adopt the idea of Similarity-aware Inverted Indexing (Christen et al. 2009, Ramadan et al. 2013) to conduct pair-wise comparisons.

As mentioned in Section 3.1, an input query's candidate matches can be obtained by looking up the query's minHash signatures in the LI. After that, the query record needs to be compared with each of the candidate records in order to get a precise list of true matches. The pair-wise comparisons are done by comparing the two records' attribute values accordingly with approximate string comparison approaches such as the Winkler function (Ramadan et al. 2013). For large datasets, there are often thousands of candidate records to compare, thus the pair-wise comparisons are computationally expensive if they have to be done in real-time. However, in real-world situations, attribute values may appear frequently, such as the names and zipcodes of populous suburbs, some popular personal names, etc. For this reason, in the Similarity-aware Inverted Indexing, the similarities of attribute values are pre-calculated, so that the similarities between attribute values can be retrieved from the Similarity Index (SI) rather than calculated online and thus the comparison time can be saved.

The Similarity-aware Inverted Indexing works in the following manner. A record is firstly processed using encoding techniques, each attribute of the record generates a encoding blocking key value. Attribute values are then stored in an index call Blocking Index (BI) under the corresponding encoding blocking key values. The BI is introduced for the purpose of reducing the number of comparisons between attribute values as an attribute value is only compared with other attribute values that have the same encoding blocking key value (same block in BI).

[Example 4] (BI: Blocking Index). The attribute "First name" in Figure 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 (shown in Figure 1). So *halle* and *hailey* are put in the block of key *sm0*; *christine*, *kristine* and *christy* are put in the block of key *krst*.

The comparisons are conducted via calculating each pair's similarity using comparison functions. The calculated similarities are then stored in an index call Similarity Index (SI) for future retrieval purpose.

[Example 5] (SI: Similarity Index). Each attribute values in the BI is compared with others in the same block using the Winkler comparison function. So *halle* is compared with *hailey*, *zach* is compared with *zack*, *christine*, *kristine* and *christy* are compared with each other. The calculated similarities are then stored in the SI. For instance, the similarities among of *kristine* and *christine* and *christy* are stored as shown in Figure 1.

3.3 Real-time Entity Resolution

The LSI includes three indexes: LSH Index (LI), Block Index (BI) and Similarity Index (SI). This section describes how the proposed approach is applied to real-time entity resolution. Similar to other indexing techniques (Christen 2012), the proposed indexing approach has two phases: building phase and querying phase.

3.3.1 Building phase

In the building phase, every record is treated as a new record and is used to build the indexes. In the beginning, all the three indexes are initialised to be empty. While a record is processed, it is firstly inserted into the LI according to the record's minHash signatures. Each minHash signature corresponds to a unique "bucket" in the LI, empty "buckets" are initialised for new minHash signatures. The record's identifier is then inserted into every bucket that the minHash signatures correspond to. Afterwards, the record is used to build the BI. Encoding blocking key values are generated based on attribute values of the record. Similar to the LI, each encoding blocking key value corresponds to a "block" in the BI. Since phonetic encoding is used, attribute values in a same "block" are similar in pronunciations. The attribute values of the inserted record are then added into the "blocks" which the encoding blocking key values correspond to in the BI. Finally, each attribute value of the inserted record is compared with other attribute values in the same "block" in the BI. The calculated similarities are then stored in the third index, SI. The building process is briefly described in Algorithm 1. A record $\mathbf{r}$'s identifier $\mathbf{r}.0$ is firstly inserted into the LI. Then the *insertion* subroutine is called to insert $\mathbf{r}$ into the BI and SI.

In the building phase, every record is treated as a new record and is used to build the indexes. In the beginning, all the three indexes are initialized to be empty. While a record is processed, it is inserted into the locality Sensitive Hashing Index (i.e., LI) based on the record's minHash signatures at the first stage. Each minHash signature corresponds to a unique block in the LI, empty blocks are initialised for new minHash signatures. The record's identifier is then inserted into every bucket that this record's minHash signatures correspond to.

Then, at the second stage, for each attribute value of a record in every Locality Sensitive Hashing block, we build the standard encoding block (i.e., BI) based on their encoding blocking key value. Afterwards, the record is used to build the BI. encoding blocking key values are generated based on attribute values of the record. Similar to the LI, each encoding blocking key value corresponds to a block in the BI. If phonetic encoding is used, then the attribute values in the same block are similar in pronunciations. The attribute values of the inserted record are then added into the blocks which the encoding blocking key values correspond to in the BI. After we build the BI, each attribute value of the inserted record is compared with other attribute values in the same block in the BI. The calculated similarities are then stored in the Similarity Index SI. The building process is briefly described in Algorithm 1. A record $\mathbf{r}$'s identifier $\mathbf{r}.0$ is firstly

Algorithm 1: ***Building phase***

input : Input dataset **D**; number of attributes n; minHash function **H**; encoding functions **E**; similarity functions **S**; **r** is a record in **D**, **r**.0 is the record identifier, **r**.i is **r**'s attribute value, $i = 1, ..., N$
output: Indexes **SI**, **BI** and **LI**

```
1  Initialise SI = {}
2  Initialise BI = {}
3  Initialise LI = {}
4  for r ∈ D do
       // First stage, insert record ID into the LI.
5      Sig = H(r)
6      for sig ∈ Sig do
7          if sig ∉ LI then
8              Initialise bk = []
9              Append r.0 to bk
10             LI[sig] = bk
11         else
12             Append r.0 to LI[sig]
       // Second stage, insert attribute values into the BI
          and SI.
13     Insert(r, n, E, S, SI, BI, LI)
```

Algorithm 2: ***Insertion***

input : Record **r** number of attributes n; encoding functions **E**; similarity functions **S**; indexes **SI**, **BI** and **LI**.
output: Updated indexes **SI**, **BI** and **LI**

```
// Second stage, insert attribute values into the BI and
   SI.
1  for i = 1...n do
2      if r.i ∉ SI then
3          c = E_i(r.i)
4          b = BI[c]
5          Append r.i to b
6          BI[c] = b
7          Initialise si = []
8          for v ∈ b do
9              s = S_i(r.i, v)
10             Append (v, s) to si
11             oi = SI[v]
12             Append (r.i, s) to oi
13             SI[v] = oi
14         SI[r.i] = si
```

inserted into the LI. Then the *insertion* subroutine is called to insert **r** into the BI and SI.

The most important part of the building phase is the insertion subroutine, it is shown in Algorithm 2.

The insertion subroutine takes a record **r**, attribute values of **r** are checked to see if they have been indexed. If an attribute value **r**.i has not been indexed previously, e.g. not in the SI, the attribute value will be inserted into both SI and BI. For instance, if **r**.i is not previously indexed, the inserting process will firstly compute its encoding value c, and add **r**.i into BI using c as encoding blocking key value. The block list **b** which contains other attribute values with the same encoding blocking key value c will then be retrieved. The similarities (denoted as s) between **r**.i and other attribute values v in **b** will be calculated using the comparison function **S**. In the SI, each attribute value has a list that stores its similarity with other attribute values. So a new similarity list si will be initialised and similarities will be stored in it in the form of tuples (v, s) where v is other attribute value and s is the similarities between **r**.i and v. Next, for each of the attribute values v in **b**, its similarity list oi will be retrieved and the similarity will be added to it in the form of tuple (**r**.i, s). Finally, the updated indexes SI, BI and LI are returned.

Algorithm 3: ***Querying phase***

input : Dataset **D** query record **q** number of attributes n; minHash function **H**; encoding functions $\mathbf{E}_i$ and similarity functions $\mathbf{S}_i$ for $i = 1...n$; indexes **SI**, **BI** and **LI**.
output: Ranked match list **M**

```
1  Initialise M = []
   // First stage
2  Sig = H(q)
3  for sig ∈ Sig do
4      if sig ∉ LI then
5          Initialise bk = []
6          Append q.0 to bk
7          LI[sig] = bk
8      else
9          bk = LI[sig]
           // Second stage, pair-wise comparisons
10         for r.0 ∈ bk do
11             if r.0 ∉ M then
12                 Retrieve r from D using r.0
13                 Initialise s = 0
14                 for i = 1...n do
15                     if q.i ∉ SI then
16                         Insert(q, n, E, S, SI, BI, LI)
17                     sl = SI[q.i]
18                     s = s + sl[r.i]
19                 Append (r.0, s) to M
20         Append q.0 to bk
21         LI[sig] = bk
22     Sort M according to similarity
```

3.3.2 Querying Phase

At querying phase, the main aim is to return the most similar records that match with a query. As mentioned before, in many scenarios, queries are required to be processed in real-time. In the LSI, real-time querying is implemented using the three indexes built in the building phase. Also, similar to the extended similarity-aware inverted indexing proposed by Ramadan et al. (2013), querying in the LSI is performed in a dynamic manner, which means every single query is regarded as a new record and is used to enrich the three indexes. The main idea of the LSI querying is to use minHash to filter out low similarity records. Thus, rather than using all the records, only the records that share the same minHash signatures with the query are considered as candidate matches for pair-wise comparisons. Because the minHash filtered out most non-matches, an enormous number of unnecessary comparisons are avoided. For the necessary comparisons, since previously appeared attribute values are all indexed and their pair-wise similarities are stored in the SI, we can get their similarity values directly rather than on-line calculation (Christen et al. 2009). Because retrieving similarities from the SI is computationally cheaper than on-line comparisons, the querying is much faster.

The querying phase is briefly described in Algorithm 3 where a query record is denoted by **q**. **q** is firstly processed using minHash to obtain its minHash signatures (denoted as **Sig**). If a minHash signature corresponds to a empty block in the LI, the query record's identifier **q**.0 will be added to the empty block. If the block is not empty, records that have been hashed into the same block will be retrieved from the LI and considered as candidate matches. Next, the query **q** will be compared to every single candidate record. The comparison is done through comparing each attribute value of **q** and the candidate record **r** accordingly. Since the similarities are pre-calculated and stored in the SI, they can be retrieved directly. However, there

are cases that the attribute values of **q** are not previously indexed and cannot be found in the SI. For such cases, **q** is treated as a whole new record and the insertion function used in the building phase is called to insert it into indexes. As the querying is performed in a dynamic environment where queries are also considered as new records, **q**'s identifier is inserted into the LI for future querying.

[**Example 6**] Suppose r_5 in Table 1 is a query record, and the querying is performed based on the three indexes shown in Figure 1. In the work of Christen et al. (2009) and Ramadan et al. (2013), r_5 is compared with both r_2 and r_4 because they share the same Double-Metaphone encoding "hrst". In the LSI, r_5 is no longer compared with the noisy record r_4 because they are in different blocks in the LI (Figure 1). Thus, the number of comparisons can be decreased.

The overall similarity between two records is the sum of the similarities of all the attribute values of two records. The candidate result records are ranked based on their overall similarity values. We can set a similarity threshold to return those candidate results that have similarities higher than the threshold as the query results. Alternatively, we can select top N highly ranked records as query results.

4 Experiment

4.1 Dataset

To evaluate the approach, we conducted experiments on North Carolina Voter Registration Dataset. This dataset is a large real-world voter registration database from North Carolina (NC) in the USA (*North Carolina State Board of Elections: NC voter registration database* Last accessed 11 December 2012). We downloaded this database every two months since October 2011 until December 2012. This data set contains the names, addresses, and ages of more than 2.4 million voters. The attributes used in our experiments are: first name, last name, city, and zip code. The entity identification is the unique voter registration number. This data set contains 2,567,642 records. There are 263,974 individuals (identified by their voter registration numbers) with two records, 15,093 with three records, and 662 with four records. Examination of the record sets for individuals with multiple records shows that many of the changes in the first name attribute contain nicknames and small typographical mistakes. The changes in last name and address attributes are mostly real changes that occur when people get married or move address.

4.2 Evaluation Approaches

To evaluate the effectiveness of the proposed approximate blocking approach, we employ the commonly used Recall, Memory Cost and Query Time to measure the effectiveness and efficiency of the whole real-time top-N entity resolution approach. We divided each dataset into training (i.e., building) and test (i.e., query) set. Each test dataset contains 50% of the whole dataset. For each test query record, the entity resolution approach will generate a list of ordered result records. The top N records (with the highest rank scores) will be selected as the query results. If a record in the results list has the same entity identification as the test query record, then this record is counted as a hit (i.e., an estimated true match). The recall value is calculated as the ratio of the total number of candidate recrods of all the test queries to the total number of true matches in the test query set. We compared the performance produced by the following approaches:

- ***LSI***. This is the proposed two-stage similarity-aware indexing approach. It contains two stages, at the first stage, we use locality sensitive hashing to filter out records with low similarities for the purpose of decreasing the number of comparisons. Then, at the second stage, we pre-calculating the comparison similarities of the attribute values to further decrease the query time.
- ***LSH***. This is the Locality Sensitive Hashing approach. It generates l length-k signatures for each data record. MinHash is used to generate the signatures of each record. This is an improved locality sensitive hashing approach that uses dynamic collision counting (Gan et al. 2012) in real-time scenario. Work (Gan et al. 2012) uses a base of length-1 basic hash functions to represent each data record. The similar data records are ranked based on the dynamic collision counting number with the query record. As blocks with length-1 signatures usually have very large blocks for large-scale datasets, LSH uses length-k signatures ($k > 1$) rather than length-1 ones.
- ***SAI***.This is the Similarity-Aware Indexing approach for real-time entity resolution discussed in (Ramadan et al. 2013). It pre-calculated the similarity of each record value pairs of the same encoding block to decrease the number comparisons in real-time querying.

The above techniques are all built for dynamic indexing where queries are regarded as new records and are inserted into indexes for future quering. All the techniques are implemented using Python (version 2.7.3). Experiments are ran on a server with 128 GBytes of main memory and four 6-core 64-bit Intel Xeon CPUs running at 2.4 GHz.

4.3 Parameter setting

For the encoding (blocking) functions, the Double-Metaphone technique was used for the first three attributes (first name, last name, and suburb), while the last 4 digits were used for the zip code attribute. For the string comparison functions, the Winkler function was used for the first three attributes, while for the zip code the similarity was calculated by counting the number of matching digits divided by the length of the zip code. In order to simulate an intensive query environment, 50% data records of each dataset is used for indexes building and the other 50% are used for indexes querying.

The number of hash functions and number of bits in each band are crucial parameters, they together control the Jaccard similarity threshold of the minHash filter by tuning the logic combination of "AND" and "OR" (Gan et al. 2012). Dividing the number of hash functions by the number of bits, we get the number of minHash signatures for a record, which is also the number of buckets the record will be assigned to in the LI. Normally, a big number of minHash signatures leads to larger memory usage and more pairwise comparisons but better query accuracy, while a small number of minHash signatures leads to less

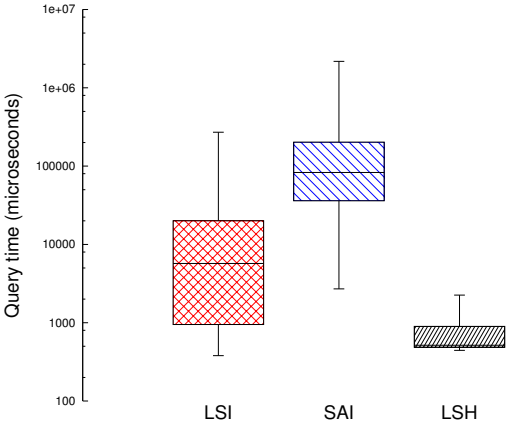

Figure 2: Summary of query time distribution, y axis is the query time in logarithmic scale $N = 100$. (boxplot with whiskers with maximum 1.5 IQR, outliers are not plotted).

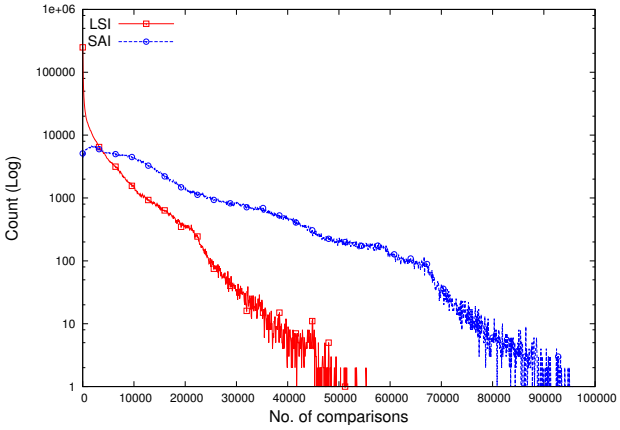

Figure 3: Distribution of the number of comparisons, $N = 100$, y axis is the count of queries in logarithmic scale and x axis is the number of comparisons. (LSH does not involve pair-wise comparisons, therefore it is not included in this figure).

query time, less memory usage and less accuracy. Different parameters can be chosen based on different scenarios. After extensive experiments, we set $k = 4$, $l = 15$ for LSH and LSI.

5 Results and Discussions

The experimental results show that the LSI's average processing time for a single query is 13.67 milliseconds, which is almost 10 times faster that the SAI (Figure 2). The improvement can be explained by decrease in the number of pair-wise comparisons as shown in Figure 3. Although the LSH is the fastest in terms of query processing, its recall is relatively low: less than 0.6 while $N = 50$. The LSI shows a good recall of around 0.7 when a small number of query results are returned. If a larger N is allowed, the recall of the SAI increases and surpasses the LSI at $N = 27$ (Figure 4). Consequently, the LSI requires more time for building indexes and more memory for storing indexes (Figure 5 and 6).

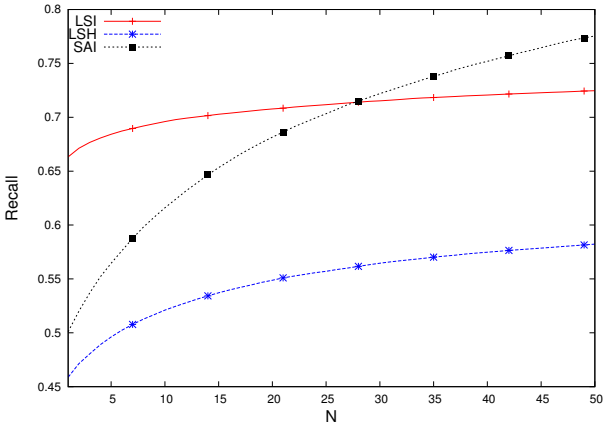

Figure 4: Top N recall results ($N = 1, ..., 50$).

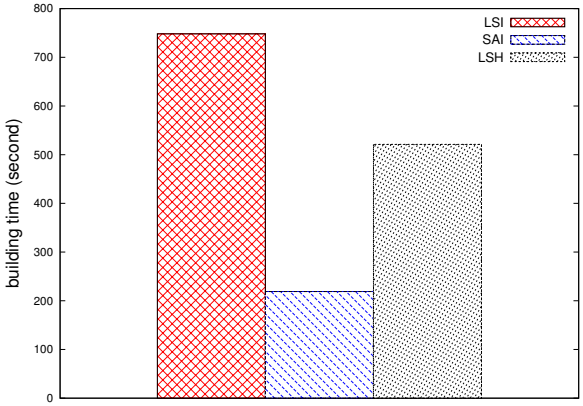

Figure 5: Building time

Discussions:

1) Query time

The distribution of query time is summarised in Figure 2 using boxplot with the range of maximum 1.5 IQR. IQR, shorted for interquartile range, equals to the difference between the upper and lower quartiles, i.e. $IQR = Q_3 - Q_1$.

We can see that the LSI approach performs better than the SAI in this aspect. The median query time for the LSI is 5.71 milliseconds, which is more than 10 times faster than the 83.03 milliseconds of the SAI. Also, as expected, the query time of the LSI is 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.

The distribution of number of comparisons can be used to explain the improvement of the LSI in query time. Processing time is much faster for queries that requires little comparison times and much slower for queries that need to be compared for many times. As shown in Figure 3, a significant number of queries in the two-stage approach are distributed in the lower range of the number of comparisons. 90% of the total queries are in the range of less than 1,000 comparison times. This is because dissimilar records are filtered out based on Jaccard similarity using minHash in the first stage, the number of candidate matches are relatively small and thus less pair-wise comparisons are needed. For many queries, pair-wise comparisons are not even needed because no candidate matches are found for them in the first stage (i.e., their minHash signatures are not found in the LI). There are

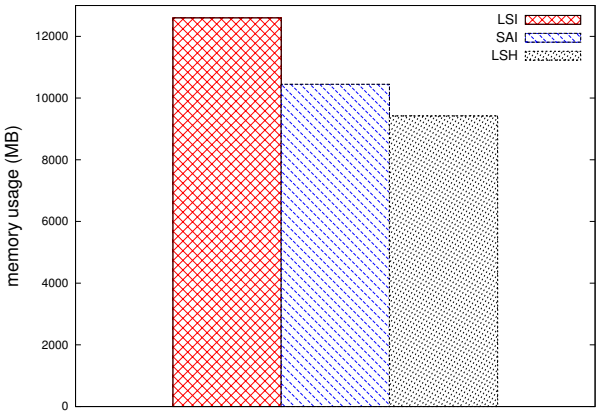

Figure 6: Memory usage, $N = 100$.

also a small proportion of queries that requires massive numbers of comparisons. This is because these queries' minHash signatures direct to large blocks in the LI, so a large number of candidate records are found for them in the first stage. As a result, the distribution of the LSI's query time shows a big range as seen in Figure 2

Comparatively, for the SAI, the number of comparisons is distributed more evenly and the maximum number of comparisons reaches almost 100,000. In the SAI, a query's comparison number totally depends on whether or not the Double-Metaphone encodings of the query's attribute values is common in the dataset. If the attribute values direct to large blocks in the BI, the query will be compared with a lot of candidate records, and vice versa. Although blocks in the BI of the SAI are of a variety of sizes, the number of Double-Metaphone encodings are limited, which makes most blocks in the BI huge. As a result, because most queries in the SAI are compared for much more time than those in the LSI, queries are processed slower in the SAI.

2) Recall

The recall distribution is given in Figure 4. As it shows, while the number of query results (i.e., N) is less than 27, the LSI's recall is higher than the SAI's. Starting from 0.66 while $N = 1$, the LSI's recall increases and slowly stabilised at 0.72. The SAI's recall grows steeply from 0.47 to 0.76 and surpasses the LSI at $N = 27$. The recall of the LSH is relatively low: less than 0.6 while $N = 50$.

In the LSH, the candidate data records are ranked based on the collision counting number with the query record. Because approximate comparison functions such as Winkler are not used in this approach, it fails to capture the spelling variations of attribute values and results in missing this type of true match records.

In the SAI, records have the same Double-Metaphone encodings with the query are all considered as candidate records and compared with the query using the Winkler function. Because attribute values with the same Double-Metaphone encoding often share common characters, they are likely to have high Winkler similarities and cannot be differentiated via similarity ranking. The top ranked records can be false matches as they may have analogical similarities. Therefore, when only a small N is allowed, false matches are likely to be included, which leads to low recalls. However, as N increases, more true matches are included than false match and the recall increases sublinearly.

The situation of the SAI does not happen to the LSI because most false matches are already eliminated by minHash based on their Jaccard similarities at the first stage, which ensures query results are of high quality while N is small. At the same time, some true matches with relatively low Jaccard similarities are also eliminated by minHash at the first stage. Consequently, the recall of the LSI grows slowly and soon settles at around 0.72 while the SAI's recall grows all the way to 0.76. So, while a small number of query results is required, the LSI outperforms the SAI in terms of recall. But if a big number of query results is acceptable, the SAI provides a better matching recall.

3) Building time and memory usage

The building times of the tree approach are shown by histograms in Figure 5. As expected, the LSI takes much longer time for building indexes. 748 seconds are used by the LSI to build indexes, which is 3 times longer than the SAI and 5 times longer than LSH. The main reason for the difference is the introduction of the LI in the LSI. The experiment setting is 60 hash functions with 4 bits in a band, which means each record is "hashed" into 15 buckets in the LI. As a result, the LI becomes the largest indexes and thus takes much longer time to build. Considering building is done off-line, the increase in building time does not have a big impact on real-time entity resolution scenarios.

While the LSI processes queries faster, it consumes more memory than other two techniques as shown in Figure 6. The LSI used more than 12,598 MB memory which is 2,158 MB more than the SAI and 3,179 MB more than the LSH. Similar to building time. the large LI plays an big part in the large memory usage of the LSI too. Additionally, for the purpose of avoiding signature collisions, minHash signatures are often large integer numbers. Storing millions of large integers in the LI also consumes a lot of memory.

6 Conclusion and Future Work

In this paper, a two-stage similarity-aware indexing approach named LSI has been presented for large-scale real-time entity resolution. LSI firstly filter out records with low similarities using locality sensitive hashing, and then pre-calculating the comparison similarities of the attribute values to further decrease the query time.

This approach is evaluated experimentally on a large-scale datasets taken from a real-world database. The experimental results demonstrated the effectiveness of the proposed approach.

Like other similarity-aware indexing techniques, the two-stage similarity-aware indexing approach requires to store pre-calculated similarities in memory, which consumes a large proportion of memory as query records are being added to the indexes continuously. Improving upon the memory consumption 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 direction for future work.

References

Anand, R. & Ullman, J. D. (2011), *Mining of massive datasets*, Cambridge University Press.

Bawa, M., Condie, T. & Ganesan, P. (2005), Lsh forest: self-tuning indexes for similarity search, *in* 'Proceedings of the 14th international conference on World Wide Web', ACM, pp. 651–660.

Baxter, R., Christen, P. & Churches, T. (2003), 'A comparison of fast blocking methods for record linkage', *ACM SIGKDD Workshop on Data Cleaning, Record Linkage and Object Consolidation, pages 2527, Washington DC* .

Broder, A. Z. (1997), On the resemblance and containment of documents, *in* 'Compression and Complexity of Sequences 1997. Proceedings', IEEE, pp. 21–29.

Christen, P. (2012), 'A survey of indexing techniques for scalable record linkage and deduplication', *Knowledge and Data Engineering, IEEE Transactions on* **24**(9), 1537–1555.

Christen, P. & Gayler, R. (2008), 'Towards scalable real-time entity resolution using a similarity-aware inverted index approach', *AusDM '08 Proceedings of the 7th Australasian Data Mining Conference* .

Christen, P., Gayler, R. & Hawking, D. (2009), Similarity-aware indexing for real-time entity resolution, *in* 'Proceedings of the 18th ACM conference on Information and knowledge management', ACM, pp. 1565–1568.

Das Sarma, A., Jain, A., Machanavajjhala, A. & Bohannon, P. (2012), An automatic blocking mechanism for large-scale de-duplication tasks, *in* 'Proceedings of the 21st ACM international conference on Information and knowledge management', ACM, pp. 1055–1064.

Dasgupta, A., Kumar, R. & Sarlós, T. (2011), Fast locality-sensitive hashing, *in* 'Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining', ACM, pp. 1073–1081.

Dong, W., Wang, Z., Josephson, W., Charikar, M. & Li, K. (2008), Modeling lsh for performance tuning, *in* 'Proceedings of the 17th ACM conference on Information and knowledge management', ACM, pp. 669–678.

Elmagarmid, A. K., Ipeirotis, P. G. & Verykios, V. S. (2007), 'Duplicate record detection: A survey', *Knowledge and Data Engineering, IEEE Transactions on* **19**(1), 1–16.

Gan, J., Feng, J., Fang, Q. & Ng, W. (2012), Locality-sensitive hashing scheme based on dynamic collision counting, *in* 'Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data', ACM, pp. 541–552.

Gionis, A., Indyk, P., Motwani, R. et al. (1999), Similarity search in high dimensions via hashing, *in* 'VLDB', Vol. 99, pp. 518–529.

Hernandez, M. A. & Stolfo, S. J. (1995), 'The merge/purge problem for large databases', *ACM SIGMOD95, San Jose* .

Ioffe, S. (2010), Improved consistent sampling, weighted minhash and l1 sketching, *in* 'Data Mining (ICDM), 2010 IEEE 10th International Conference on', IEEE, pp. 246–255.

Kim, H.-s. & Lee, D. (2010), Harra: fast iterative hashed record linkage for large-scale data collections, *in* 'Proceedings of the 13th International Conference on Extending Database Technology', ACM, pp. 525–536.

Lange, D. & Naumann, F. (2011), Efficient similarity search: arbitrary similarity measures, arbitrary composition, *in* 'Proceedings of the 20th ACM international conference on Information and knowledge management', ACM, pp. 1679–1688.

Lange, D. & Naumann, F. (2012), 'Cost-aware query planning for similarity search', *Information Systems* .

Li, L., Wang, D., Li, T., Knox, D. & Padmanabhan, B. (2011), Scene: a scalable two-stage personalized news recommendation system., *in* 'SIGIR', pp. 125–134.

Lv, Q., Josephson, W., Wang, Z., Charikar, M. & Li, K. (2007), Multi-probe lsh: efficient indexing for high-dimensional similarity search, *in* 'Proceedings of the 33rd international conference on Very large data bases', VLDB Endowment, pp. 950–961.

Michelson, M. & Knoblock, C. A. (2006), Learning blocking schemes for record linkage, *in* 'Proceedings of the National Conference on Artifiical Intelligence', Vol. 21, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, p. 440.

North Carolina State Board of Elections: NC voter registration database (Last accessed 11 December 2012).
URL: *ftp://www.app.sboe.state.nc.us/*

Ramadan, B., Christen, P., Liang, H., Gayler, R. W. & Hawking, D. (2013), Dynamic similarity-aware inverted indexing for real-time entity resolution, *in* 'Trends and Applications in Knowledge Discovery and Data Mining', Springer, pp. 47–58.

Slaney, M. & Casey, M. (2008), 'Locality-sensitive hashing for finding nearest neighbors [lecture notes]', *Signal Processing Magazine, IEEE* **25**(2), 128–131.

Sood, S. & Loguinov, D. (2011), Probabilistic near-duplicate detection using simhash, *in* 'Proceedings of the 20th ACM international conference on Information and knowledge management', ACM, pp. 1117–1126.

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