Noise-Tolerant Approximate Blocking for Dynamic Real-Time Entity Resolution*

Huizhi Liang[1], Yanzhe Wang[1], Peter Christen[1], and Ross Gayler[2]

[1] Research School of Computer Science, The Australian National University, Canberra ACT 0200, Australia
{huizhi.liang,peter.christen}@anu.edu.au, colin788@163.com
[2] Veda, Melbourne VIC 3000, Australia
ross.gayler@veda.com.au

Abstract. Entity resolution is the process of identifying records in one or multiple data sources that represent the same real-world entity. This process needs to deal with noisy data that contain for example wrong pronunciation or spelling errors. Many real world applications require rapid responses for entity queries on dynamic datasets. This brings challenges to existing approaches which are mainly aimed at the batch matching of records in static data. Locality sensitive hashing (LSH) is an approximate blocking approach that hashes objects within a certain distance into the same block with high probability. How to make approximate blocking approaches scalable to large datasets and effective for entity resolution in real-time remains an open question. Targeting this problem, we propose a noise-tolerant approximate blocking approach to index records based on their distance ranges using LSH and sorting trees within large sized hash blocks. Experiments conducted on both synthetic and real-world datasets show the effectiveness of the proposed approach.

Keywords: Entity Resolution, Real-time, Locality Sensitive Hashing, Indexing.

1 Introduction

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 for example be a person (e.g. customer, patient, or student), a consumer product, a business, or any other object that exists in the real world. Entity resolution is widely used in various applications such as identity crime detection (e.g. credit card fraud detection) and estimation of census population statistics [1].

Currently, most available entity resolution techniques conduct the resolution process in offline or batch mode with static databases. However, in real-world scenarios, many applications require real-time responses. This requires entity

* This research was funded by the Australian Research Council (ARC), Veda Advantage, and Funnelback Pty. Ltd., under Linkage Project LP100200079. Note the first two authors contributed equally.

V.S. Tseng et al. (Eds.): PAKDD 2014, Part II, LNAI 8444, pp. 449–460, 2014.
© Springer International Publishing Switzerland 2014

resolution on query records that need to be matched within sub-seconds with databases that contain (a possibly large number of) known entities [1]. For example, online entity resolution based on personal identifying details can help a bank to identify fraudulent credit card applications [2], while law enforcement officers need to identify suspect individuals within seconds when they conduct an identity check [1]. Moreover, real-world databases are often dynamic. The requirement of dealing with large-scale dynamic data with quick responses brings challenges to current entity resolution techniques. Only limited research has so far focused on using entity resolution at query time [3,4] or in real-time [5,6].

Typically, pair-wise comparisons between records are used to identify the records that belong to the same entity. The number of record comparisons increases dramatically as the size of a database grows. Indexing techniques such as blocking or canopy formation can help to significantly decrease the number of comparisons [1]. Often phonetic encoding functions, such as Soundex or Double Metaphone, are used to overcome differences in attribute values.

Locality sensitive hashing (LSH) [7] is an approximate blocking approach that uses l length k hash functions to map records within a certain distance range into the same block with a given probability. This approach [8] can filter out records with low similarities, thus decreasing the number of comparisons. However, the tuning of the required parameters k and l is not easy [9]. This is especially true for large-scale dynamic datasets. For some query records, one may need to investigate records with low similarities, while for other query records one only needs to investigate those records with high similarities with the query record. Moreover, entity resolution needs to deal with noise such as pronunciation or spelling errors. Although some LSH approaches such as multi-probe [10] are to decrease the number of hash functions needed, the question of how to make blocking approaches become more noise-tolerant and scalable remains open.

In this paper, we propose a noise-tolerant approximate blocking approach to conduct real-time entity resolution. To deal with noise, an n-gram based approach [1] is employed where attribute values are converted into sets of n-grams (i.e, substring sets of length n). Then, LSH is used to group records into blocks with various distance ranges based on the Jaccard similarity of their n-grams. To be scalable, for blocks that are large (i.e., contains more than a certain number of records), we propose to build dynamic sorting trees inside these blocks and return a small set of nearest neighbor records for a given query record.

2 Related Work

Indexing techniques can help to scale-up the entity resolution process [1]. Commonly used indexing approaches include standard blocking based on inverted indexing and phonetic encoding, n-gram indexing, suffix array based indexing, sorted neighborhood, multi-dimensional mapping, and canopy clustering. Some recent work has proposed automatic blocking mechanisms [11]. Only a small number of approaches have addressed real-time entity resolution. Christen et al. [5] and Ramadan et al. [6] proposed a similarity-aware indexing approach for

real-time entity resolution. However, this approach fails to work well for large datasets, as the number of similarity comparisons for new attribute values increases significantly when the size of each encoding block grows.

Approximate blocking techniques such as LSH and tree based indexing [9] are widely used in nearest neighbour similarity search in applications such as recommender systems [12] and entity resolution [8]. In LSH techniques, the collision probability of a LSH family is used as a proxy for a given distance or similarity measure function. Popularly used LSH families include the minHash family for Jaccard distance, the random hyperplane family for Cosine Distance, and the p-stable distribution family for Euclidean Distance [13].

Recently, Gan et al. [14] proposed to use a hash function base with n basic length 1 signatures rather than using fixed l length k signatures or a forest [9] to represent a data point. The data points that are frequently colliding with the query record across all the signatures are selected as the approximate similarity search results. However, as k=1 usually leads to large sized blocks, this approach needs to scan all the data points in the blocks to get the frequently colliding records each time. This makes it difficulty to retrieve results quickly for large-scale datasets. Some approaches such as multi-probe [10] have been used to decrease the number of hash functions needed. However, how to explore LSH blocks in a fast way and decrease the retrieval time to facilitate real-time approximate similarity search for large scale datasets still needs to be explored.

3 Problem Definition

To describe the proposed approach, we first define some key concepts.

- **Record Set:** $R = \{r_1, r_2, \ldots, r_{|R|}\}$ contains all existing records in a dataset. Each record corresponds to an entity, such as a person. Let U denote the universe of all possible records, $R \subseteq U$.
- **Element:** An element is an n-gram of an attribute value. Elements may overlap but do not cross attribute boundaries. A record contains $1 \ldots m$ elements, denoted as $r_i = \{v_{i1}, v_{ij}, \ldots, v_{im}\}$.
- **Query Record:** A query record $q_i \in U$ is a record that has the same attribute schema as the records in R. After query processing and matching, q_i will be inserted into R as a new record $r_{|R|+1}$.
- **Query Stream:** $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$ is a set of query records.
- **Entity Set:** $E = \{e_1, e_2, \ldots, e_{|E|}\}$ contains all unique entities in U.

For a given record $r_i \in R$, the decision process of linking r_i with the correspondent entity $e_j \in E$ is denoted as $r_i \rightarrow e_j$, and $e_j = l(r_i)$, where $l(r_i)$ denotes the function of finding the entity of record r_i. The problem of real-time entity resolution is defined as: for each query record q_i in a query stream Q, let e_j be the entity of q_i, $e_j = l(q_i)$, find all the records in R that belong to the same entity as the query record in sub-second time. Let L_{q_i} denote the records in R that belong to e_j, $L_{q_i} = \{r_k \mid r_k \rightarrow l(q_i), r_k \in R\}, L_{q_i} \subseteq R, q_i \in Q$.

Record ID	Entity ID	First Name	Family Name	City	Zip Code
r_1	e_1	Tony	Hua	Sydney	4329
r_2	e_2	Emily	Hu	Perth	1433
r_3	e_3	Yong	Wan	Perth	4320
r_4	e_1	Tonny	Hue	Syndey	4456

(a) An example record dataset

Record ID	2-grams (n=2)
r_1	to, on, ny, hu, ua, sy, yd, dn, ne, ey, 43, 32, 29
r_2	em, mi, il, ly, hu, pe, er, rt, th, 14, 43, 33
r_3	yo, on, ng, wa, an, pe, er, rt, th, 43, 22, 20
r_4	to, on, nn, ny, hu, ue, sy, yn, nd, de, ey, 43, 35, 56

(b) The elements(2-grams) for the example records

Fig. 1. An example dataset and the elements(2-grams) of each record

[**Example 1**] Figure 1(a) shows example records r_1, r_2, r_3, and r_4. They belong to three entities e_1, e_2, and e_3. Assume r_4 is a query record, the entity resolution process for r_4 is to find $L_{r_4} = \{r_1\}$ based on the four attribute values.

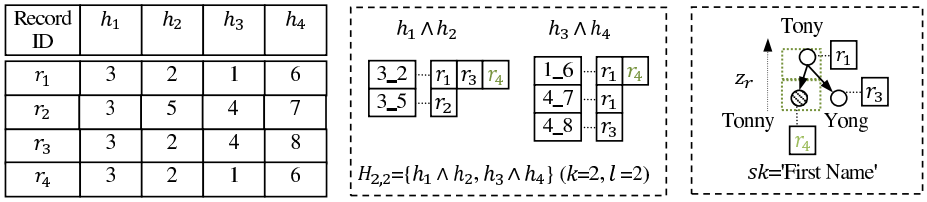
4 Proposed Approximate Blocking Approach

A blocking schema [15] is an approach to map a set of records into a set of blocks. Blocking schemes generate signatures for records. A blocking scheme can be a LSH function, a canopy clustering function, or a phonetic encoding function [1]. Those records with the same signature will be allocated together in the same block. To make a LSH blocking scheme scalable for large-scale dynamic datasets, we propose to build a dynamic sorting tree to sort the records of large-sized LSH blocks and return a window of nearest neighbors for a query record. Through controlling the window size, we can select nearest neighbors with various approximate similarity ranges for the purpose of being noise-tolerant. Thus, the proposed approach includes two parts: LSH and dynamic sorting tree, which will be discussed in Section 4.1 and 4.2. Then, the discussion of conducting entity resolution based on the proposed blocking approach will be in Section 4.3.

4.1 Locality Sensitive Hashing

LSH can help to find approximate results for a query record for high dimensional data. Let h denote a basic approximate blocking scheme for a given distance measure D, $Pr(i)$ denote the probability of an event i, and 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 records $r_x, r_y \in R$, the following conditions hold:

1. if $D(r_x, r_y) \leq d_1$ then $Pr(h(r_x) = h(r_y)) \geq p_1$
2. if $D(r_x, r_y) > d_2$ then $Pr(h(r_x) = h(r_y)) \leq p_2$



(a) The length 1 minHash values of hash functions h_1, h_2, h_3, h_4.

(b) An example LSH blocking family and generated blocks

(c) Dynamic sorting tree for block 3_2

Fig. 2. Example hash values, LSH family, and generated blocks and a dynamic sorting tree for the example dataset in Figure 1, r_4 is a query record

Minwise hashing (minHash) is a popular LSH approach that estimates the Jaccard similarity [7]. Let $J(r_x, r_y)$ denote the Jaccard similarity for any two records r_x and r_y. This hashing method applies a random permutation Π on the elements of any two records r_x and r_y and utilizes

$$Pr(min(\Pi(r_x)) = min(\Pi(r_y))) = J(r_x, r_y) = \frac{|r_x \bigcap r_y|}{|r_x \bigcup r_y|} \quad (1)$$

to estimate the Jaccard similarity of r_x and r_y, where $min(\Pi(r_x))$ denotes the minimum value of the random permutation of the elements 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 elements of the two records to that of the union of the elements of the two records. A minHash function can generate a basic signature for a given record. The basic signature is called a length 1 signature and the hash function is called a length 1 hash function.

In order to allocate records that have higher similarities with each other into the same block, k length 1 hash functions can be combined to form a length k ($k > 1$) compound blocking scheme to get the intersection records of the basic length 1 blocking schemes. Let h_c denote a compound LSH blocking scheme that is the *AND*-construction (conjunction) of k basic LSH blocking schemes h_i, $h_c = \wedge_{i=1}^{k} h_i$. Let p_c denote the collision probability of a length k compound blocking scheme. It can be calculated based on the product of the collision probabilities of its basic length 1 blocking schemes, denoted as $p_c = p^k$. To increase the collision probability, each record is hashed l times to conduct *OR*-construction (disjunction) and form l hash tables (i.e., l length k signatures), $n = k \cdot l$. Let $H_{k,l}$ denote a LSH family that has l length k hashing blocking schemes, the collision probability of $H_{k,l}$ can be estimated with $p_{k,l} = 1 - (1 - p^k)^l$.

[**Example 2**] (LSH blocking scheme and blocks) Figure 2 (a) shows the example length 1 minHash signatures of the example records in Figure 1. Figure 2 (b) shows an example LSH blocking scheme $H_{2,2}$ and the generated LSH blocks. $H_{2,2} = \{h_1 \wedge h_2, h_3 \wedge h_4\}$, $k = 2$ and $l = 2$. The records are allocated into different blocks based on the given blocking scheme, shown in Figure 2 (b). For example, the length 2 blocking scheme $h_1 \wedge h_2$ generated block signatures 3_2 and 3_5. Records r_1, r_3 and r_4 are allocated in block 3_2 while r_2 is in block 3_5.

4.2 Dynamic Sorting Tree

Based on a given LSH Family $H_{k,l}$, we can allocate records with certain similarity into the same LSH block and filter out those records that have lower similarities. Using a large value of k will result in smaller sized blocks and decrease the number of pair-wise comparisons, but it may result in low recall because a large value for k may filter out some true matched records that have low similarities. A small value for k usually can get higher recall values but may result in large-sized blocks. As scanning the records in large-sized blocks to conduct pair-wise comparisons is usually time-consuming, how to quickly identify a small set of nearest neighbors in a large sized LSH block is very important.

If we sort the records in a LSH block and only return a small set of similar records for each query record, then the exploration of the whole large-sized block can be avoided. $B+$ trees are commonly used to sort the data points for dynamic data in one dimension [9]. We build a $B+$ tree to sort the records inside each large sized LSH block. As forming sub-blocks for those small sized blocks is not necessary, we set up a threshold for building sorting trees in a block. Let γ ($\gamma > 1$)denote this threshold, if the size of a block B_i is greater than γ, then we build a $B+$ tree for B_i, otherwise, no sorting tree will be formed. To build a sorting tree, we firstly discuss how to select a sorting key.

Selecting a Sorting Key Adaptively. Typically, the sorting key can be assigned by a domain expert [16,17]. For example, for the example dataset, we can select 'First Name' as the sorting key. As the LSH blocks are formed by the random permutation of elements, the common elements of the records in block B_i may be different from those in another block B_j. Using a predefined fixed sorting key may result in a whole block being returned as query results, which will fail to return a sub-set of records in a block. On the other hand, we can select the sorting key adaptively for each LSH block B_i individually.

One or several attributes can be selected as sorting key. For a block B_i, a good sorting key should divide the records into small sub-blocks. Thus, we can select those attributes that have a greater number of unique values in γ records. Moreover, if the attribute value occurrences are uniformly distributed, we can get sub-blocks with the same or similar sizes. Thus, for an attribute a_j, we calculate a sorting key selection weight w_j, which consists of the linear combination of two components: attribute cardinality weight and distribution weight:

$$w_j = \alpha \cdot \frac{n_j}{\gamma} + (1 - \alpha) \cdot \sigma_j \tag{2}$$

Where $0 \leq \alpha \leq 1$, and n_j is the number of unique values of attribute a_j in block B_i. $\frac{n_j}{\gamma}$ measures the attribute cardinality weight, $0 \leq \frac{n_j}{\gamma} \leq 1$. σ_j measures the distribution of occurrences of each unique value of a_j, calculated as the standard deviation of the occurrences of the value of a_j. Let o_{jc} denote the occurrence of attribute value v_{jc} in γ records, m_a denote the maximum occurrence, $m_a = \max_{b\in[1,K]}(o_{jb})$,where K is the number of unique attribute values of attribute j in γ records, m_i denotes the minimum occurrence, $m_i =$

$\min_{b\in[1,K]}$. To get a normalized weight between 0 and 1, we set $n_{jc} = \frac{o_{jc}-m_i}{m_a-m_i}$, thus, $\sigma_j = \frac{1}{K}\sqrt{\Sigma_{c=1}^{K}(n_{jc}-\mu)^2)}$, where μ is the average value of n_{jc} for a_j, $\mu = \frac{\Sigma_{c=1}^{K} o_{jc}}{K}$.

The attribute a_j that has the largest w_j will be selected as the sorting key. Let sk denote a selected sorting key for B_i, then the $B+$ tree is built by sorting the records in B_i on key sk. For text records, we can sort them lexically by the alphabetic order of the sorting key. One advantage of sorting by alphabetic order is that such an ordering is a global unique order for all the attribute values of the sorting key in B_i. The distance of any two attribute values of the sorting key can be measured by the distance of their alphabetic order. Thus, for a query record, a set of nearest neighbors can be obtained through measuring their alphabetic order distance. Each unique sorting key value denoted as v_{sk} is one node of the $B+$ tree. Each node is an inverted index that points to the records that have the same v_{sk}, denoted as $dt_i = (v_{sk}, I(v_{sk}))$, where v_{sk} is an attribute value of sk, and $I(v_{sk})$ is the set of records that have the same v_{sk}. Let $|B_i|$ denote the number of records in a block B_i, the time complexity of searching, insertion and deletion of $B+$ tree is $O(\log|B_i|)$, which is quicker than scanning all the records in a block, $O(\log|B_i|) < O(|B_i|)$, for large blocks.

Selecting Nearest Neighbors. Every record in a block B_i can be selected as a candidate record for query record q_i. However, for those blocks that have a large number of records, we can select a set of nearest neighbors as candidate records to reduce the query time. For a given query record q_i, we firstly insert it into the $B+$ tree of a LSH block based on the alphabetic order of the block's sorting key sk. Then the nearest neighbor nodes of the $B+$ tree will be selected as the candidate records for this query record. Let $v_{i,sk}$ denote the sorting key value of query record q_i, we choose z_r nodes that are greater than $v_{i,sk}$ and z_l nodes that are smaller than $v_{i,sk}$ to form the nearest neighbor nodes of $v_{i,sk}$, $0 \le z_l + z_r \le |B_i|$. How to set the z_l and z_r value is important.

If we set z_l and z_r=0, then only those records with the exact same value as the query record will be selected. If z_l or $z_r = |B_i|$, then all the records of B_i will be selected. As the distance of two nodes in a sorting $B+$ tree reflects the distance of two records, we set a distance threshold θ of two nodes. Let $v_{i,sk}$ denote the attribute value of sorting key sk for query record q_i. For a node $v_{j,sk}$ of a given $B+$ tree, let $D(v_{i,sk}, v_{j,sk})$ denote the distance of $v_{i,sk}$ and $v_{j,sk}$ for a given distance measure D (e.g., edit distance [1]). If the distance of two nodes is less than θ, then we increase the window size to include node $v_{j,sk}$ inside the window. Thus, we firstly set z_l and $z_r = 0$. The window size expansion is along both directions [16] (i.e., greater than or smaller than the query record node i) of the $B+$ tree. For each direction, we expand the window size (i.e., z_l or z_r) and include neighbor node j, if $D(v_{i,sk}, v_{j,sk}) < \theta$, with $0 \le \theta \le 1$.

To further decrease the number of candidate records and select a smaller set of nearest neighbors, we count the collision number of each record of the neighboring nodes inside the window of the sorting tree in all l LSH blocks to rank the records that are attached to the selected neighboring nodes. This is because the

co-occurrence of a record r_x that appears together with q_i in the LSH blocks reflects the similarity of the two records [14]. The higher the co-occurrence is, the more similar the two records are. We set a threshold φ to select those records that appear at least φ times with the query record q_i together in LSH blocks. Let g_{ix} denote the co-occurrence of record r_x and query record q_i. Let $N_{v_{i,sk}}$ denote the nearest neighbor record set of query record q_i in block B_i. For each record r_x of the neighbor node j inside the window of the sorting tree DST_i(i.e., $r_x \in I(v_{j,sk})$), we add r_x to $N_{v_{i,sk}}$ if $g_{ix} > \varphi$, $0 \leq \varphi \leq l$.

[**Example 3**] (Dynamic sorting tree) Figure 2(c) shows the dynamic sorting tree DST_{3_2} for block 3_2. DST_{3_2} is sorted by attribute 'First Name'. Through setting window size parameters, we can get a set of nearest neighbor records for a query record. For example, if we set $z_r = 0$, no records will be selected as candidate records, and only the records with node value 'Tonny' will be selected. Assume $\theta = 0.6$, $\varphi = 1$, the Jaccard similarity of the bi-grams of 'Tony' and 'Tonny' is 0.75 and that of the bi-grams of 'Tonny' and 'Yong' is 0.167. Thus, we can set $z_l = 0$ and $z_r = 1$ for DST_{3_2}. Record r_1 appears twice together with query record r_4 in blocks 3_2 and 1_6, thus r_1 is selected.

4.3 Real-Time Entity Resolution

For a query record q_i, we can obtain the nearest neighbor records that are being allocated in the same block with q_i as the candidate records. Then, we can conduct pair-wise comparisons for all candidate records with the query record q_i. We use the Jaccard similarity of the n-grams of a candidate record and the query record, or other appropriate approximate distance/similarity measures to rank their similarity [1]. Let C_{q_i} denote the candidate record set, for each candidate record $r_j \in C_{q_i}$ and query record $q_i \in Q$, the similarity can be calculated with $sim(q_i, r_j) = J(q_i, r_j)$. The top N candidate records will be returned as the query results L_{q_i}. The algorithm is shown as Algorithm 1.

5 Experiments and Results

5.1 Data Preparation

To evaluate the proposed approach, we conducted experiments on the following two datasets.

1) **Australian Telephone Directory.** [5] (named OZ dataset). It contains first name, last name, suburb, and postcode, and is sourced from an Australian telephone directory from 2002 (Australia On Disc). This dataset was modified by introducing various typographical and other variations to simulate real 'noisy' data. To allow us to evaluate scalability, we generated three sub-sets of this dataset. The smallest dataset (named OZ-Small) has 34,596 records, the medium sized dataset (OZ-Median) has 345,996 records, and the largest dataset (OZ-Large) has 3,458,758 records. All three datasets have the same features including similarity distribution, duplicate percentages (i.e., 25%) and modification types.

Algorithm 1: Query(q_i,$H_{k,l}$, N)

Input:
- $q_i \in Q$ is a given query record
- $H_{k,l}$ is a given LSH blocking schema
- N is a given number of returned results

Output:
- L_{q_i} is the ranked list of retrieved records

1: $L_{q_i} \leftarrow \{\}$, $C_{q_i} \leftarrow \{\}$ // Initialization, C_{q_i} is the candidate records set
2: $B_{q_i} \leftarrow H_{k,l}(gram(q_i, n)$ // Conduct LSH blocking for the n-grams of q_i
3: **For** each block $B_{bid} \in B_{q_i}$:
4: If $|B_{bid}| < \gamma$:
5: $C_{q_i} \leftarrow C_{q_i} \bigcup B_{bid}$ // Get all records in B_{bid} as candidate records
6: If $|B_{bid}| = \gamma$: // Build sorting tree and select nearest neighbor records. See Section 4.2.
7: Get sorting key sk
8: Build dynamic sorting tree DST_{bid}
9: Get nearest neighbours $N_{v_{i,sk}}$
10: $C_{q_i} \leftarrow C_{q_i} \bigcup N_{v_{i,sk}}$
11: If $|B_{bid}| > \gamma$: // Select nearest neighbor records. See Section 4.2.
12: Insert q_i to DST_{bid}
13: Get nearest neighbours $N_{v_{i,sk}}$
14: $C_{q_i} \leftarrow C_{q_i} \bigcup N_{v_{i,sk}}$
15: **For** each candidate record $r_j \in C_{q_i}, r_j \neq q_i$:
16: Get $sim(q_i, r_j)$ // Conduct pair-wise similarity comparisons
17: $L_{q_i} \leftarrow max\{C_{q_i}, N\}$ // Return top N ranked results

2) **North Carolina Voter Registration Dataset.** (i.e., NC dataset). This dataset is a large real-world voter registration database from North Carolina (NC) in the USA [18]. We downloaded this database every two months since October 2011. 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.

5.2 Evaluation Approaches

In the experiments, we employ the commonly used Recall, Memory Cost and Query Time to measure the effectiveness and efficiency of real-time top N entity resolution approach [1]. We divided each dataset into a training (i.e., building) and a 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 hits of all the test queries to the total number of true matches in the test query set. We conducted comparison experiments with three other state-of-the-art methods as described below. All methods were implemented in Python, and the experiments were conducted on a server with 128 GBytes of main memory and two 6-core Intel Xeon CPUs running at 2.4 GHz.

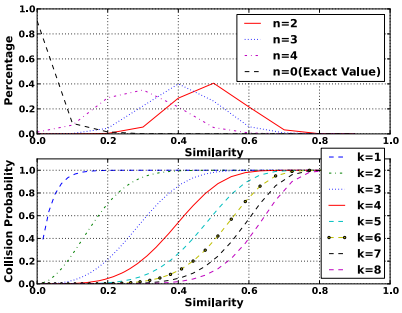


Fig. 3. The similarity distribution of various n values, and the collision probability of various k values with l=30 for OZ-Large

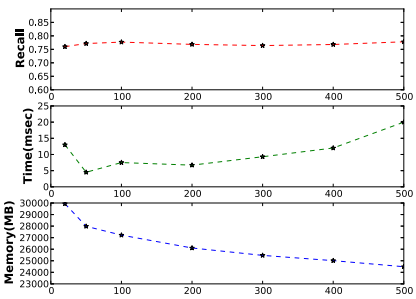


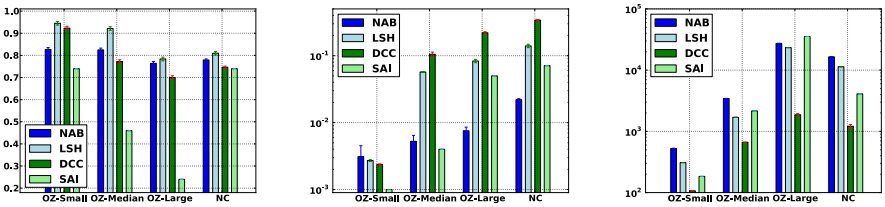
Fig. 4. Top N=10 results of NAB with various γ values for the OZ-Large, with the x-axis showing values for γ

- ***NAB***: This is the proposed noise-tolerant approximate blocking approach that includes LSH and dynamic sorting trees.
- ***DCC***: This is a locality sensitive hashing approach that uses dynamic collision counting [14] approach. It uses a base of length 1 basic hash functions.
- ***LSH***: This is the basic LSH approach. It scans the data points in each block and conducts pair-wise comparison for all the records inside the blocks.
- ***SAI***: This approach pre-calculates the similarity of attribute values to decrease the number of comparisons at query time [5,6].

5.3 Parameter Setting

We firstly discuss the parameter setting for the OZ datasets. To set the parameters k and l, we calculated the Jaccard similarity distribution of the exact values and n-grams with n=2, 3, 4 of the true matched records of the training sets. The similarity distribution is shown in Figure 3. The Jaccard similarity of 90% of the exact values of the true matched records is zero. This means that it would be very difficult to find true matched records if we use the exact value of the records. Also, the similarity range of the majority (i.e., 95%) of the 2-grams of the true matched records is between 0.3 and 0.7. Thus, using an n-gram based approach can help to find those true matched records that contain small variations or errors. n is therefore set to 2 in the experiments. Figure 3 also shows the collision probability of various k values with l=30. We set k=4 and l=30 for the NAB approach to let most true matched records have a higher collision probability. To get a similar collision probability, we set k=4 and l=30 for the LSH approach, and k=1 and l=20 for the DCC approach. The settings of the other parameters of the NAB approach are α=0.5, θ=2, φ=0.1.

Figure 4 shows the top N=10 evaluation of NAB for the OZ-Large dataset with various γ value. With various γ value, recall remains stable while average query time is increasing, with the increase of γ. This shows that building dynamic



(a) Recall (b) Time (sec) (c) Memory Usage (MB)

Fig. 5. The top N=10 Recall, Average Query Time and Memory Usage results

sorting trees inside LSH blocks can help to decrease the query time through selecting a small number of nearest neighbor records as candidate records. A small γ value (e.g., γ=20) will not necessarily decrease the average query time, as the building of trees also takes time and space. When γ is set to a large value (e.g., γ=500), the average query time increases because scanning and comparing a large number of candidate records is time consuming. We set γ=200. For the NC dataset, we set n=3, k=2, l=20 for LSH and NAB, and k=1, l=10 for DCC. The other parameters for NAB are the same as with the OZ datasets.

5.4 Comparison with Baseline Models

The performances of the compared approaches are shown in Figure 5. To eliminate the influence of random permutation, the LSH based approaches LSH, DCC and NAB were run three times. The average query time and memory usage are shown on a logarithmic scale. From Figure 5 we can see that SAI achieved good recall value for OZ-Small and NC data but low recall for OZ-Large. The LSH based approaches (LSH, DCC and NAB) had higher recall than SAI. This can be explained that these approaches can capture the common elements (i.e., n-grams) of the attribute values to deal with the noise of the data. Moreover, through controlling the k and l value, these approaches can filter out the records that have lower similarities with the query record. DCC had very low memory usage but high average query time. NAB had slightly lower recall and higher memory usage than that of LSH, but with the help of dynamic sorting trees, the average query time (e.g., 8 msec for OZ-Large) is much lower than LSH (e.g., 0.1 sec for OZ-Large). Thus, NAB can be effectively and efficiently used for large scale real-time entity resolution, especially for noisy data.

6 Conclusions

We discussed a noise-tolerant approximate blocking approach to facilitate real-time large scaled entity resolution. To deal with noise, we use an LSH approach to group records into blocks with various distance ranges based on the Jaccard

similarity of their n-grams. Moreover, we propose to build dynamic sorting trees inside large-sized LSH blocks. Through controlling the window size, we select a small set of nearest neighbors with various approximate similarity ranges to be noise-tolerant. Experiments conducted on both synthetic and real-world large scaled datasets demonstrates the effectiveness of the proposed approach. Our future work will focus on how to conduct adaptive real-time entity resolution.

Acknowledgements. The authors would like to thank the great help of Professor David Hawking.

References

1. Christen, P.: Data Matching. Data-Centric Systems and Appl. Springer (2012)
2. Christen, P., Gayler, R.W.: Adaptive temporal entity resolution on dynamic databases. In: Pei, J., Tseng, V.S., Cao, L., Motoda, H., Xu, G. (eds.) PAKDD 2013, Part II. LNCS, vol. 7819, pp. 558–569. Springer, Heidelberg (2013)
3. Lange, D., Naumann, F.: Cost-aware query planning for similarity search. Information Systems, 455–469 (2012)
4. Bhattacharya, I., Getoor, L., Licamele, L.: Query-time entity resolution. In: SIGKDD, pp. 529–534 (2006)
5. Christen, P., Gayler, R., Hawking, D.: Similarity-aware indexing for real-time entity resolution. In: CIKM, pp. 1565–1568 (2009)
6. Ramadan, B., Christen, P., Liang, H., Gayler, R.W., Hawking, D.: Dynamic similarity-aware inverted indexing for real-time entity resolution. In: Li, J., Cao, L., Wang, C., Tan, K.C., Liu, B., Pei, J., Tseng, V.S. (eds.) PAKDD 2013 Workshops. LNCS (LNAI), vol. 7867, pp. 47–58. Springer, Heidelberg (2013)
7. Gionis, A., Indyk, P., Motwani, R., et al.: Similarity search in high dimensions via hashing. In: VLDB, pp. 518–529 (1999)
8. Kim, H.S., Lee, D.: HARRA: Fast iterative hashed record linkage for large-scale data collections. In: EDBT, pp. 525–536 (2010)
9. Bawa, M., Condie, T., Ganesan, P.: LSH forest: Self-tuning indexes for similarity search. In: WWW, pp. 651–660 (2005)
10. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe LSH: Efficient indexing for high-dimensional similarity search. In: VLDB, pp. 950–961 (2007)
11. Das Sarma, A., Jain, A., Machanavajjhala, A., Bohannon, P.: An automatic blocking mechanism for large-scale de-duplication tasks. In: CIKM, pp. 1055–1064 (2012)
12. Li, L., Wang, D., Li, T., Knox, D., Padmanabhan, B.: Scene: A scalable two-stage personalized news recommendation system. In: SIGIR, pp. 125–134 (2011)
13. Anand, R., Ullman, J.D.: Mining of massive datasets. Cambridge University Press (2011)
14. Gan, J., Feng, J., Fang, Q., Ng, W.: Locality-sensitive hashing scheme based on dynamic collision counting. In: SIGMOD, pp. 541–552 (2012)
15. Michelson, M., Knoblock, C.A.: Learning blocking schemes for record linkage. In: AAAI, pp. 440–445 (2006)
16. Yan, S., Lee, D., Kan, M.Y., Giles, L.C.: Adaptive sorted neighborhood methods for efficient record linkage. In: DL, pp. 185–194 (2007)
17. Draisbach, U., Naumann, F., Szott, S., Wonneberg, O.: Adaptive windows for duplicate detection. In: ICDE, pp. 1073–1083 (2012)
18. Christen, P.: Preparation of a real voter data set for record linkage and duplicate detection research. Technical report, Australian National University (2013)