Dynamic Similarity-Aware Inverted Indexing for Real-Time Entity Resolution*

Banda Ramadan[1], Peter Christen[1], Ross W. Gayler[2], David Hawking[3] and Huizhi Liang[1]

[1] Research School of Computer Science, The Australian National University, Canberra ACT 0200, Australia;
{banda.ramadan,peter.christen,huizhi.liang}@anu.edu.au
[2] Veda Advantage, Melbourne VIC 3000, Australia; ross.gayler@veda.com.au
[3] Funnelback Pty. Ltd., Dickson ACT 2601, Australia; david.hawking@acm.org

Abstract. Entity resolution is the process of identifying groups of records in a single or multiple data sources that represent the same real-world entity. It is an important tool in data de-duplication, in linking records across databases, and in matching query records against a database of existing entities. Most available entity resolution techniques complete the resolution process offline and on static databases. However, real-world databases are often dynamic, and increasingly organizations need to resolve entities in real-time. Thus, there is a need for new techniques that facilitate working with dynamic databases in real-time. In this paper, we propose a dynamic similarity-aware inverted indexing technique (DySimII) that meets these requirements. We also propose a frequency-filtered indexing technique where only the most frequent attribute values are indexed. We experimentally evaluate our techniques on a large real-world voter database. The results show that when the index size grows no appreciable increase is found in the average record insertion time (around 0.1 msec) and in the average query time (less than 0.1 sec). We also find that applying the frequency-filtered approach reduces the index size with only a slight drop in recall.

Keywords: Dynamic indexing, real-time query, record linkage, data matching, duplicate detection, frequency-filtered indexing

1 Introduction

Massive amounts of data are being collected by most business and government organizations. Given that many of these organizations rely on information in their day-to-day operations, the quality of the collected data has a direct impact on the quality of the produced outcomes [7, 13]. Various data validation and cleaning practices are employed to improve the collected data. One important practice in data cleaning and data integration is the identification of all records that refer to the same real-world entity [13]. This process is called *entity resolution* [7] and may be applied to a single or to multiple data sources (within a

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

single data source the process is called *de-duplication*). A real-world entity could be a person (e.g. customer, patient or student), a product, a business, or any other object that exists in the real world.

Examples of duplicates may be a patient who is represented several times in a hospital database, a product that is inserted many times in an inventory list, or a voter who is registered more than once in an election roll. These duplicates, if not removed or merged, can lead to serious consequences for organizations or individuals. A patient's information could for example be dispersed between their duplicated records, leaving medical staff unaware of the patient's overall condition and affecting diagnosis and treatment. In another example, duplicate records in an election roll could allow voting irregularities.

In many cases, organizations need to perform the entity resolution process in real-time in order to be able to complete their operations. For example, social services need to identify individuals on the spot even if their social security number is not available. Police officers also need to identify individuals within seconds when they do an identity check on a suspect, using their personal details. Real-time matching can be achieved using indexes that reside in main memory rather than using disk-based indexes, and by reducing the size of those indexes.

The databases used by most organizations and businesses are not static but are modified constantly by adding, deleting, or updating records. This makes the entity resolution process more challenging, since most entity resolution techniques currently available are only suitable for static databases. This is because most of these techniques are based on batch algorithms that resolve all records rather than resolving those relating to a single query record. There is an urgent need to develop new techniques that support real-time entity resolution for dynamic large databases.

In this paper we develop and investigate a dynamic similarity-aware inverted indexing technique [5] for real-time entity resolution on dynamic databases. Our contribution is two-fold. The first is to dynamically update the inverted index after every query record rather than leaving the index out-of-date between periodic batch updates. The second is to investigate reducing the size of the inverted index using frequency-based filtering where only the attribute values that occur most frequently in a database are inserted into the index, rather than indexing all attribute values.

2 Related Work

The entity resolution process encompasses several steps [7]: *Data Preprocessing*, which cleans and standardizes the data to be used; *Indexing*, which creates candidate records that potentially correspond to matches; *Record Pair Comparison*, which compares the candidate records using one or more similarity matching functions [10]; *Classification*, where candidate record pairs are classified into matches and non-matches; and finally, *Evaluation*, where the entity resolution process is evaluated using a variety of measures [4].

This paper is concerned with the indexing step, which is a vital step in entity resolution as it reduces the number of records that need to be compared in detail to resolve the matching of a given query record. Different indexing techniques are summarized in [6]. Standard Blocking [11] segregates records into blocks according to a certain criteria, and subsequently compares only records that are in the same block. Usually this criteria, called blocking key, is based on one or more attributes [7]. This approach has the disadvantage of assigning records into the wrong block in case of errors in attribute values (i.e. dirty data). To avoid this from occurring, Iterative Blocking [17] can be applied where multiple blocking keys are used and each record is inserted into more than one block.

The sorted neighbourhood method (SNM) [12] sorts all records in the database using a key that is based on attribute values. Then a fixed size widow is used to slide over the sorted records comparing only records within the window. The main drawback of this method is that using a fixed size window potentially leads to missed true matches if the window size is too small, and unnecessary pair comparisons if the window size is too large. To avoid this an Adaptive SNM [9, 19] can be used where the window size can grow or shrink adaptively.

Another approach is q-gram based indexing. The idea behind this technique is to convert the values of the blocking key into a list of q-grams(a q-gram is a sequence of q continuous characters generated from a string). Based on these generated lists of q-grams, each attribute is inserted into more than one block to reduce the effect of errors that might occur in attribute values. Although this approach achieves better quality blocking than traditional blocking and the SNM [2], it is computationally expensive and therefore is not suitable for large databases or real-time entity resolution.

Canopy Clustering [15] on the other hand is a technique that aims at speeding up the process of blocking records for large databases. This technique uses a computationally cheap clustering approach to create high dimensional overlapping clusters called *canopies*, from which block of candidate record pairs can then be generated. Another idea of indexing is to use a sorted suffix array [1] of all the subsequences of tokens that appear in a string. These tokens are used as blocking keys allowing records to be inserted in several blocks.

Existing entity resolution techniques focus on improving the accuracy and the efficiency of the entity resolution process. However, most of these techniques are aimed at offline processing of static databases. Not much research has concentrated on real-time entity resolution (where a stream of query records arrive that need to be resolved against the records in the database in real-time) or on entity resolution for dynamic databases. The first query-time entity resolution approach is based on a collective classification approach [3]. The idea behind this approach is to only use a subset of records in a database for resolving queries, by extracting records related to a query and then resolving this query using only these records. Although this approach can improve matching quality, experiments showed an average time of 31.28 sec was needed on a database with 831,991 records. Thus, this approach is not suitable for real-time entity resolution, nor is it scalable to large databases because it is computationally expensive.

The similarity-aware inverted indexing technique [5] that is described in Section 3 proposes a real-time entity resolution solution that is very fast. Although this approach has an average query time of 0.1 seconds for a query record on a database that contains nearly 7 million records, it only works on static databases. Another real-time entity resolution approach that works on static databases is proposed by Dey et al. [8]. This approach is based on using a matching tree to limit the amount of communication required for matching records between disparate databases held at different locations, where a matching decision can be made without the need of comparing all attribute values between records. This approach is shown to reduce the communication overhead, without affecting the matching quality.

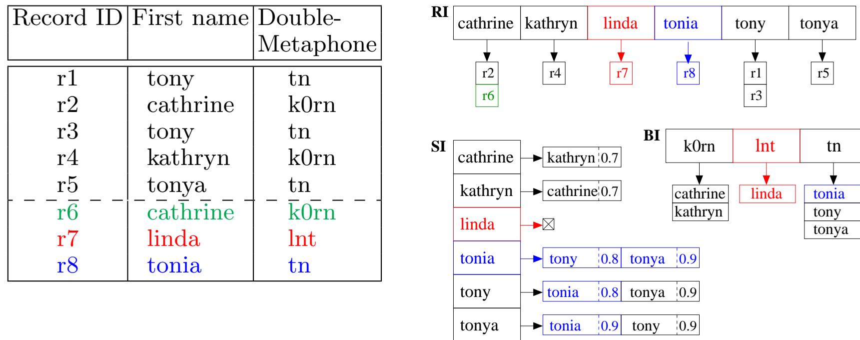
Ioannou et al. [14] on the other hand propose an approach that provides entity resolution in real-time, and works on dynamic databases. Their method is based on using links between the entities in a database combined with a probabilistic database for resolving entities. The approach uses existing entity resolution techniques to find possible matches of a query, and instead of using these possible matches to make an offline resolution decision, it stores the possible matches alongside with a probability weight in a dynamic index data structure. This stored information is then used at query time to perform entity resolution in real-time. The approach is reported to have an average time of 70 msec for a query record on a database of 51,222 records. This query time is almost constant and does not increase when the database get larger.

Another dynamic entity resolution approach is proposed by Whang and Garcia-Molina [18] that allows matching rules to evolve over time when new records become available. This approach aims at using materialized entity resolution results (which are a set of records that are classified as matches) to save redundant work, and does not require running the entity resolution process from scratch. The authors report that this rule evolution approach can be faster than the naive approach by up to several orders of magnitude [18].

3 Dynamic Similarity-Aware Inverted Indexing

The similarity-aware inverted index proposed by Christen et al. [5] aims at providing real-time entity resolution for a stream of query records. The main idea behind this approach is to pre-calculate similarities between attribute values that are in the same block (using traditional blocking [11]). These pre-calculated similarities are stored in main memory to be used later in the query resolution process. Avoiding similarity calculations at query time significantly reduces the time needed for matching a query record. This approach was shown to be two orders of magnitude faster than traditional approaches [5], which makes it suitable for real-time entity resolution.

To use this approach, three indexes are needed as shown in Fig. 1. The first index, called the Block Index (BI), is an inverted index that stores unique attribute values and their associated blocking key values (generated using traditional encoding techniques such as Soundex, Phonex, or Double-Metaphone [7]).



Record ID	First name	Double-Metaphone
r1	tony	tn
r2	cathrine	k0rn
r3	tony	tn
r4	kathryn	k0rn
r5	tonya	tn
r6	cathrine	k0rn
r7	linda	lnt
r8	tonia	tn

Fig. 1. The Dynamic Similarity-aware Inverted Index created from the example records in the table on the left. The example records contain first name values and their Double Metaphone encodings used as blocking key values. r6, r7 and r8 in the table illustrate the two cases of inserting an attribute value to the index as described in Sect. 3.1. RI is the record identifier index, BI is the block index and SI is the similarity index.

Keys of this index are the blocking key values, while each key points to a list of all attribute values that have this blocking key value. The second index, called the Similarity Index (SI), stores the pre-calculated similarities between attribute values that are in the same block (similarities are calculated using comparison functions such as Winkler or Jaccard [7]). Keys for the second index are unique attribute values, while each key points to a list of pre-calculated similarities between this value and all other values that are in the same block (similarity values range between 0 and 1, where 1 means exact match).

Finally, the Record Index (RI) stores all unique attribute values and their associated record identifiers. Keys of this index are the unique attribute values, while each key points to a list of all record identifiers that have the same attribute value.

To overcome the limitation of the original similarity-aware inverted indexing technique that works only with static databases, we modify the original technique to be used with dynamic databases. The main difference between the two techniques is that in the original technique indexes are static, and once they are created no more records and attribute values are added to them. On the other hand, the proposed dynamic indexing technique (DySimII) is more flexible and values are added whenever a new query is being processed. The following section describes in detail the proposed technique.

3.1 Indexing Dynamic Databases

Building the Indexes. We start with three empty indexes. First, unique attribute values for records are added to the three indexes as described in [5]. Adding attribute values to the inverted indexes is based on two cases: the first case occurs when an attribute value is new and it does not exist in the inverted indexes (r7 and r8 in Fig. 1). The second case occurs when an attribute value has been indexed previously and it exists in the inverted indexes (r6 in Fig. 1).

Algorithm 1: ***DySimII - Overall***

Input:
- Database: $\mathbf{D}$
- Number of attributes of $\mathbf{D}$ used: n
- Encoding functions: $\mathbf{E}_a, a = 1 \ldots n$
- Similarity functions: $\mathbf{S}_a, a = 1 \ldots n$
- Stream of query records: $\mathbf{Q}$

Output:
- Record index: $\mathbf{RI}$
- Similarity index: $\mathbf{SI}$
- Block index: $\mathbf{BI}$
- Ranked list of matches: $\mathbf{M}$

1: Initialise $\mathbf{RI} = \{\}$
2: Initialise $\mathbf{SI} = \{\}$
3: Initialise $\mathbf{BI} = \{\}$
4: **for** $\mathbf{r} \in \mathbf{D}$**:**
5:     **for** $a = 1 \ldots n$**:**
6:     $Insert(\mathbf{r}.a, \mathbf{r}.id, \mathbf{E}_a, \mathbf{S}_a)$
7: **for** $\mathbf{q} \in \mathbf{Q}$**:**
8:     $Query(\mathbf{q}.a, \mathbf{q}.id, \mathbf{E}_a, \mathbf{S}_a)$

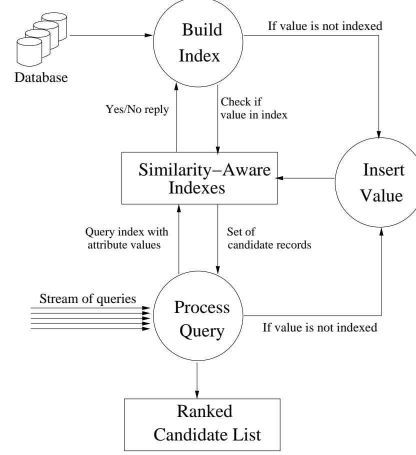


Fig. 2. Algorithm 1 illustrates the overall process of the DySimII technique. The right side illustrates the framework of the DySimII technique, as described in Sect. 3.1

1. In the first case, an attribute value is first inserted into the RI with its associated record identifier. Then its encoded value is calculated to decide into which block it should be added. If the block of that encoded value exists then the attribute value will be inserted into that block in the BI. Otherwise, a new block for the encoded value is created and this attribute value is inserted. If this value is added into an existing block, the similarities between this new attribute value and all other values within this block are calculated. These similarities are then stored in the similarity index (SI).
2. In the second case, where a value has been indexed previously, the only action needed is to add the identifier of this query record that holds this attribute value into the corresponding record list in the RI.

The process of loading and indexing attribute values will continue until we reach the last record in the database. As a result we will have three inverted indexes (BI, SI, and RI). In the original similarity-aware indexing technique, building the indexes will stop at this point. If new values arrive, these values cannot be added to the indexes. However, this issue is handled in DySimII allowing more values to be added to the indexes. If a new record arrives, its attribute values are loaded, encoded values are calculated, and the steps described above will take place adding any new value to the previously built indexes. The overall DySimII framework is illustrated in Fig 2.

The process of inserting a new attribute value (Algorithm 2) requires the identifier of the record of this attribute, $\mathbf{r}.id$, the encoding function, the similarity comparison function, and the inverted indexes RI, SI, and BI as input. The process starts with inserting the record identifier of the attribute value to the RI (line 1). If this attribute value does not exist in the SI (line 2), the following steps

Algorithm 2: *DySimII - Insert*

Input:
- Attribute value: $\mathbf{r}.a$
- Record identifier: $\mathbf{r}.id$
- Encoding functions: $\mathbf{E}_a, a = 1 \dots n$
- Similarity functions: $\mathbf{S}_a, a = 1 \dots n$
- Indexes: $\mathbf{RI}, \mathbf{SI}, \mathbf{BI}$

Output:
- Updated indexes: $\mathbf{RI}, \mathbf{SI}, \mathbf{BI}$

1: Append $\mathbf{r}.id$ to $\mathbf{RI}[\mathbf{r}.a]$
2: **if** $\mathbf{r}.a \notin \mathbf{SI}$**:**
3:     $c = \mathbf{E}_a(\mathbf{r}.a)$
4:     $\mathbf{b} = \mathbf{BI}[c]$
5:     Append $\mathbf{r}.a$ to $\mathbf{b}$
6:     $\mathbf{BI}[c] = \mathbf{b}$
7:     Initialise inverted index list $\mathbf{si} = ()$
8:     **for** $v \in \mathbf{b}$**:**
9:         $s = \mathbf{S}_a(\mathbf{r}.a, v)$
10:         Append (v, s) to $\mathbf{si}$
11:         $\mathbf{oi} = \mathbf{SI}[v]$
12:         Append $(\mathbf{r}.a, s)$ to $\mathbf{oi}$
13:         $\mathbf{SI}[v] = \mathbf{oi}$
14:     $\mathbf{SI}[\mathbf{r}.a] = \mathbf{si}$

Algorithm 3: *DySimII - Query*

Input:
- Query record: $\mathbf{q}$
- Number of attributes of $\mathbf{D}$ used: n
- Encoding functions: $\mathbf{E}_a, a = 1 \dots n$
- Similarity functions: $\mathbf{S}_a, a = 1 \dots n$
- Indexes: $\mathbf{RI}, \mathbf{SI}, \mathbf{BI}$

Output:
- Ranked list of matches: $\mathbf{M}$

1: Initialise $\mathbf{M} = ()$
2: **for** $a = 1 \dots n$**:**
3:     **if** $\mathbf{q}.a \notin \mathbf{RI}$**:**
4:         $Insert(\mathbf{q}.a, \mathbf{q}.id, \mathbf{E}_a, \mathbf{S}_a)$
5:     $\mathbf{ri} = \mathbf{RI}[\mathbf{q}.a]$
6:     **for** $\mathbf{r}.id \in \mathbf{ri}$**:**
7:         $\mathbf{M}[\mathbf{r}.id] = \mathbf{M}[\mathbf{r}.id] + 1.0$
8:     $\mathbf{si} = \mathbf{SI}[\mathbf{r}.a]$
9:     **for** $(\mathbf{r}.a, s) \in \mathbf{si}$**:**
10:         $\mathbf{ri} = \mathbf{RI}[\mathbf{r}.a]$
11:         **for** $\mathbf{r}.id \in \mathbf{ri}$**:**
12:             $\mathbf{M}[\mathbf{r}.id] = \mathbf{M}[\mathbf{r}.id] + s$
13:     Sort $\mathbf{M}$ according to similarities

Fig. 3. Algorithm 2 illustrates the process of inserting an attribute value to the index structures. Algorithm 3 illustrates the process of querying an attribute value from the index structures.

are conducted. First, the encoding value c is calculated and all other values in its block are retrieved from the BI. The new value is then added into the inverted index list $\mathbf{b}$ of this block, and the updated list is stored back into BI (lines 3-6). Next the similarities between the new attribute value $\mathbf{r}.a$ and all attribute values v that already exist in this block are calculated (line 8), and inserted into both the new value's similarity list $\mathbf{si}$ (line 10) and the other value's list $\mathbf{oi}$ (line 12). Finally, the similarity list $\mathbf{si}$ of the new value $\mathbf{r}.a$ is added to SI in line 14.

Querying in Real-Time. Since similarities between attribute values that are located in the same block are pre-calculated and stored in the SI, the process of resolving queries is faster than with traditional entity resolution techniques [5]. However, in the original similarity-aware inverted indexing approach after resolving and matching a certain query record it is not added into the inverted index data structures. As a result, when a previously resolved attribute value arrives in another query record we would again have to do the similarity calculations to resolve that query. To avoid such repeated unnecessary computations, the DySimII approach adds any new attribute value from a query record into the indexes. Therefore, even if the same attribute value occurs in several query records the encoding and similarity computations need to be performed only the first time. Algorithm 3 illustrates how the query is handled in the DySimII.

First an accumulator $\mathbf{M}$, which is a data structure that contains record identifiers and their similarities with the query record, is initialized (line 1). Then for every attribute value in the query, the algorithm checks if this value is in the RI (lines 2 and 3). If it is not in the RI, then it will be inserted into the three

Table (a).

First Name (118,565)		Second Name (189,890)	
james	41,022	smith	28,243
michael	37,536	williams	23,247
john	34,141	johnson	22,100
robert	33,241	jones	19,807
william	31,918	brown	16,256
david	30,126	davis	15,002
mary	22,993	moore	10,968
christopher	20,590	miller	10,435
jenifer	19,468	wilson	10,254
charles	17,438	harris	9,380

Table (b).	Frequency = 1	Frequency <= 10
First Name	67.72%	93.63
Last Name	47.14%	90.76%

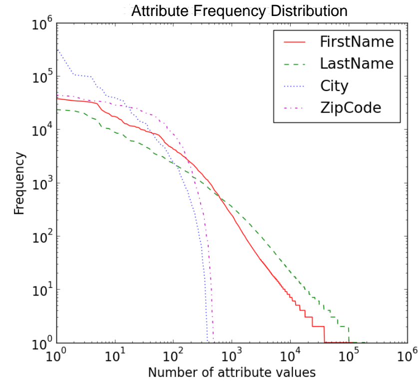


Fig. 4. Table (a) represents a list of the top 10 most frequent first/last names in the North Carolina (NC) voter database and the total number of distinct attribute values. Table (b) illustrates the percentage of uncommon first/last names. The right plot illustrates the distribution of attribute values in the NC database.

indexes as described in Algorithm 1 (line 4). In lines 5 to 7, the identifiers $\mathbf{r}.id$ of all other records that have the same attribute value are retrieved and their similarities (exactly 1, as they have the same attribute value) are added into the accumulator $\mathbf{M}$. A new element for record identifier $\mathbf{r}.id$ will be added to the accumulator if it does not exist. Next, all other attribute values in the same block and their similarities with the query attribute value are retrieved from the SI (line 8). For each of these values, their record identifiers are retrieved from the RI and their similarities are added into the accumulator in line 12. Finally, in line 13, the accumulator is sorted such that the records with largest similarities are located at the beginning, and this sorted list is returned.

3.2 Frequency-Filtered Indexing

The original similarity-aware inverted indexing technique is based on building the inverted indexes for all unique attribute values in a database. However, some attribute values may be uncommon and have low frequencies. Indexing such rare values might not be of use. For instance, the first name *John* in the voter database that we use for our experiments (see Sect. 4) is a common name and it has a frequency of 34,141 in this database, while a first name like *Juvena* is uncommon and only occurs once in the same database. This suggests that the probability of receiving a query with the first name value *John* is much higher than the probability of receiving a query with the value *Juvena*. Figure 4 illustrates the frequency distributions of the four attributes that are in this database (i.e. first name, last name, city, and zip code). It can be seen that only a small number of attribute values have a high frequency, while most have low frequencies.

Many databases that contains values such as personal details are found to have a frequency distribution that follow Zipf's law [20], which states that in

a list of words ranked according to their frequencies, the word at rank r has a relative frequency that corresponds to $1/r$. Distributions in which the relative frequency approximates $1/r^\alpha$ are considered Zipfian, even if $\alpha \neq 1$. This means that the number of uncommon values in many databases is large, and since these values are taking space in the inverted index while they are not queried very often, we suggest indexing only the most frequent attribute values.

We therefore investigate the affect of indexing only a certain percentage $x\%$ of the most frequent attribute values where $0 < x\% < 100$. This requires a list of frequent values, which can be generated for example from an online telephone book. Before we add any value into the index structures, we check if this value is in the $x\%$ of most frequent values. If this is the case, the value is added to the index structures, otherwise it will not. This process is expected to reduce the size of the index structures since we are only indexing the most frequent values.

4 Experimental Evaluation

In our experimental evaluation, we use a large real-world voter registration database from North Carolina (NC) in the US [16]. We downloaded this database every two months since October 2011 to build a compound temporal database. This database contains the names, addresses, and ages of more than 2.5 million voters, as well as their unique voter registration numbers. Each record has a time-stamp attached which corresponds to the date a voter originally registered, or when any of their details have changed. This database therefore contains realistic temporal information about a large number of people. We identified 263,974 individuals with two records, 15,093 with three, and 662 with four records in this database.

The attributes used in our experiments are: first name, last name, city, and zip-code. An exploration of the database has shown that many of the changes in the first name attribute are corrections of nicknames and small typographical mistakes, while changes in last name and address attributes are mostly genuine changes that occur when people get married or move address. Our analysis also shows that this database approximately follows a Zipfian frequency distribution for the first name and last name attributes, and that most attribute values have low frequencies, as can be seen in Fig. 4.

We implemented our approach using Python (version 2.7.3), and ran experiments on a server with 128 GBytes of main memory and two 6-core Intel Xeon CPUs running at 2.4 GHz. For the encoding (blocking) functions $\mathbf{E}_a$, the Double Metaphone [7] technique was used for the first name, last name, and city attributes, while the last 4 digits were used for the zip code. For the string comparison functions $\mathbf{S}_a$, the Winkler function was used [7] 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 our first set of experiments, we evaluate whether the proposed approach facilitates real-time querying and updating of dynamic databases. We use the first 10% of the database to build the inverted index data structures, and we

(a) (b)

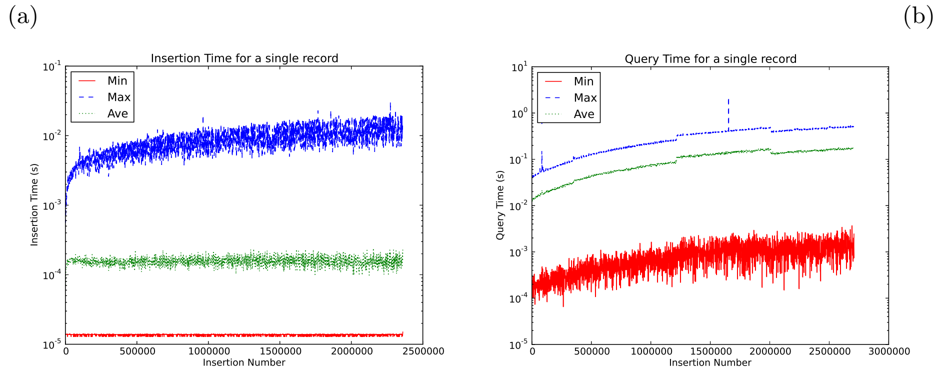


Fig. 5. Plot (a) illustrates the time needed for a single record insertion into the index data structures. Plot (b) illustrates the average time required for querying the growing index data structures.

treat the remaining records as queries. When these query records are processed, they are added to the already existing index data structures. The record insertion time, the average query time and the memory usage are measured for the growing size of the index structures.

In our second set of experiments, we evaluate the frequency-filtered indexing and investigate its effect on the original static similarity-aware inverted indexing technique [5]. The investigation includes the effect on recall (i.e. found number of true matches over total number of true matches), memory requirements, and query coverage (i.e. the ratio between the number of times an attribute value of a query is available in the index and the number of times that it is not available). In this set of experiments we index the most frequent $x\%$ of attribute values where x ranges from 10% to 100%. A value of $x = 10\%$ means that only the most frequent 10% of the attribute values from the NC database are added to the index structures. A list of all distinct values and their frequencies in the NC database has been generated earlier to be used in the decision of adding an attribute value to the index structures or not. We finally apply the frequency-filtered indexing approach to the dynamic similarity-aware indexing technique to see if it has the same effect as on the original static approach.

5 Results and Discussions

The results from running the first set of experiments are illustrated in Fig. 5. In plot (a), it is shown that the average time needed for a single record insertion into the index structures is almost constant. This implies that even if the index structures grow dynamically when new values are added to it, it does not affect the time required for inserting a single record. The plot shows that the maximum insertion time for a single record is around 10 msec, the average insertion time is

Measure	x% of the most frequent attribute values indexed									
	10	20	30	40	50	60	70	70	90	100
Coverage	0.73	0.85	0.92	0.95	0.98	0.98	0.99	0.99	0.99	1.00
Recall (%)	64	81	92	95	98	99	100	100	100	100
Memory (MB)	1,026	1,086	1,151	1,222	1,299	1,395	1,489	1,595	1,710	1,828

Fig. 6. Recall, query coverage, and memory usage when indexing only x% of the most frequent attribute values using the original similarity-aware indexing technique.

around 0.1 msec, and the minimum insertion time is around 0.01 msec. In plot (b), it can be seen that the query time is not affected by the growing size of the index structures. The plot shows that the maximum query time for a single record is less than 0.5 sec, the average query time is less than 0.1 sec, and the minimum query time is around 1 msec.

The memory required for building the index structures for the whole database was 3,641 MB. This memory required was only a tiny fraction of the amount available on the experimental machine, indicating that our implementation would be viable even for much larger databases. If the size of the problem increased substantially relative to the physical memory, a possible approach is to prune low-frequency attributes from the index structures.

A summary of the results for the second set of experiments (frequency-filtered indexes) is illustrated in Fig. 6. As seen in this table, the query coverage is very high for all values of $x\%$. For example when we index only 30% of the most frequent attribute values from the NC voter database, 92% of the arriving queries are actually found in the index. Recall drops only slightly when we do not index all attribute values. For example, when we index 50% of the most frequent attribute values, the recall drops only by 2%. The acceptability of this drop in recall depends on who is using the proposed approach. For example, a 2% drop in recall might be acceptable for a general business company, but not for a national security organization. The experiments also show that memory requirements drop by around 43% when we index only 10% of the most frequent attribute values in the database. When applying the frequency-filtered approach on the dynamic similarity-aware inverted indexing technique, similar effect occurs where the recall remains almost the same for all different values of $x\%$, and memory is dropped by almost half when we index only 10% of the most frequent attribute values in the voter database.

6 Conclusions

In this paper, a dynamic similarity-aware inverted indexing technique for real-time entity resolution on dynamic databases is presented, and a frequency-filtered indexing is investigated where only the most frequent attribute values are indexed. The two approaches are evaluated using a large real-world database. The experiments show that the growing size of the dynamic index structures proposed in our approach does not affect the time required to insert new records

into the index structures, and it does not affect the time required to resolve query records. Moreover, it was shown that the frequency-filtered approach reduces the size of the index structures with only a slight drop in recall. Our future work aims at conducting a theoretical analysis of scalability to much larger databases and investigating the use of this approach in a parallel environment.

References

1. Aizawa, A., Oyama, K.: A Fast Linkage Detection Scheme for Multi-Source Information Integration. In: WIRI, pp.30-39. Tokyo (2005).
2. Baxter, R., Christen, P., Churches, T.: A Comparison of fast blocking methods for record linkage. In: ACM SIGKDD Workshop on Data Cleaning, Record Linkage and Object Consolidation, Washington DC (2003)
3. Bhattacharya, I., Getoor, L.: Collective Entity Resolution. Journal of Artificial Intelligence Research 30, 621-657 (2007)
4. Christen, P., Goiser, K.: Quality and Complexity Measures for Data Linkage and Deduplication, In: Guillet, F., Hamilton, H.J. (eds.) Studies in Computational Intelligence (SCI), vol. 43, Springer (2007)
5. Christen, P., Gayler, R., Hawking, D.: Similarity-Aware Indexing for Real-Time Entity Resolution. In: ACM CIKM , pp. 1565-1568. Hong Kong (2009)
6. Christen, P.: A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. IEEE Transactions on Knowledge and Data Engineering (2012)
7. Christen, P.: Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution and Duplicate Detection. Springer, Canberra (2012)
8. Dey, D., Mookerjee, V., Liu, D.: Efficient Techniques for Online Record linkage. IEEE Transactions on Knowledge and Data Engineering 23(3), 373-387 (2010)
9. Draisbach, U., Naumann, F., Szott, S., Wonneberg, O.: Adaptive Windows for Duplicate Detection. In: International Conference on Data Engineering. IEEE (2012)
10. Elmagarmid, A.K. and Ipeirotis, P.G. and Verykios, V.S.: Duplicate Record Detection: A Survey. Knowledge and Data Engineering. 19(1), 1-16 (2007)
11. Fellegi, I.P. and Sunter, A.B.: A Theory for Record Linkage. Journal of the American Statistical Association, 64, 1183-1210 (1969)
12. Hernandez, M.A., Stolfo, S.J.: The Merge/Purge Problem for Large Databases. In: ACM SIGMOD, pp. 127-138. San Jose (1995)
13. Herzog, T.N., Scheuren, F.J., Winkler, W.E.: Data Quality and Record Linkage Techniques, Springer, New York (2007)
14. Ioannou, E., Nejdl, W., Niederee, C., Velegrakis, Y.: On-the-fly entity-aware query processing in the presence of linkage. Proceeding of the VLDB Endowment 3 (2010)
15. McCallum, A., Nigam, K., Ungar, L.H.: Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching. In: ACM SIGKDD, pp. 169-178. Boston (2000).
16. North Carolina State Board of Elections, NC voter registration database, `ftp://www.app.sboe.state.nc.us/`
17. Whang, S.E., Menestrina, D., Koutrika, G., Theobald, M., Garcia-Molina, H.: Entity Resolution with Iterative Blocking. Technical report, Stanford University (2008)
18. Whang, S.E., Garcia-Molina, H.: Entity Resolution with Evolving Rules. Proceeding of the VLDB Endowment 3 (1-2), 1326-1337 (2010)
19. Yan, S., Lee, D., Kan, M.Y., Giles, L.C.: Adaptive Sorted Neighborhood Methods for Efficient Record Linkage. In: ACM/IEEE-CS Joint Conference on Digital Libraries, pp. 185-19 (2007)
20. Zipf, G.K.: Human Behavior and the Principle of Least Effort. Addison-Wesley (1949)