Similarity-Aware Indexing for Real-Time Entity Resolution

Peter Christen
School of Computer Science
Australian National University
Canberra ACT 0200, Australia
peter.christen@anu.edu.au

Ross Gayler
Scoring Solutions
Veda Advantage
Melbourne VIC 3000, Australia
ross.gayler@vedaadvantage.com

David Hawking
Funnelback Pty Ltd
Dickson ACT 2601, Australia
david.hawking@acm.org

ABSTRACT

Entity resolution, also known as data matching or record linkage, is the task of identifying records from several databases that refer to the same entities. Traditionally, entity resolution has been applied on static databases, for example to find records that relate to the same patient in different health databases. Most research in entity resolution has concentrated on either improving the matching quality, making entity resolution scalable to very large databases, or reducing the manual efforts required throughout the resolution process. Increasingly, however, many organisations are faced with the challenge of having large databases that contain entities, and a stream of query records that have to be matched with these databases in real-time, such that the best matching records are retrieved. Example applications include online law enforcement and national security databases, public health surveillance and emergency response systems, financial verification systems, and online retail stores.

In this paper, a novel inverted index based approach for real-time entity resolution is presented. At build time, similarities between attribute values are computed and stored to support the fast matching of records at query time. The presented approach differs from other recently developed approaches to approximate querying, in that it allows any similarity comparison function, and any 'blocking' function, both possibly domain specific, to be incorporated.

Experimental results on a large real-world database indicate that the total size of all data structures of this novel index approach grows sub-linearly with the size of the database, and that it allows matching of query records in sub-second time, more than two orders of magnitude faster than a traditional entity resolution index approach.

Categories and Subject Descriptors

H.3.3 [**Information Systems**]: Information Storage and Retrieval—*Information Search and Retrieval*; H.3.1 [**Information Systems**]: Information Storage and Retrieval—*Content Analysis and Indexing.*

Accepted as poster at CIKM'09, November 2–6, 2009, Hong Kong.

General Terms

Algorithms, Experimentation, Performance.

Keywords

Data matching, record linkage, scalability, similarity query, approximate string matching, inverted indexing.

1. INTRODUCTION

Increasingly, many applications that deal with data management and analysis require that data from different sources is matched and aggregated before it can be used for further processing. The aim of data matching is to identify and match all records that refer to the same real world entities. These entities can, for example, be customers, patients, tax payers, travellers, students, businesses, consumer products, or bibliographic citations. While statisticians and health researchers commonly name the task of matching records as data or record linkage, computer scientists and the database and business oriented IT communities speak of entity resolution, data or field matching, data cleansing, data integration, duplicate detection, data scrubbing, list washing, object identification, or merge/purge processing.

Traditionally, techniques for matching records that correspond to the same entities have been applied in the health sector and within the census [14, 21, 28]. Increasingly, however, entity resolution is now being used within and between many organisations in both the public and private sectors in a large variety of application domains. Examples include finding duplicates in business mailing lists, bibliographic databases (digital libraries) and online stores; crime and fraud detection within finance and insurance companies as well as government agencies; compilation of longitudinal data for social research; or the assembly of terrorism watch lists for improved national security.

Because real-world data rarely contains unique entity identifiers across all the databases to be matched, most entity resolution approaches compare records using the information available in the databases that partially identify entities, such as their names, address details, or dates of birth. For each of the partially identifying attributes compared between two records, a similarity is calculated. These similarities are then used collectively to classify each compared record pair as a match, non-match, or possible match [14, 16, 28]. The matching process is often challenged because real world data is *dirty*, i.e. contains missing or out-of-date attribute values, variations and errors, values that are swapped between attributes, or data that is coded differently [26].

Traditional entity resolution approaches assume that two or more static databases are to be matched in batch mode, in order to produce a new matched data set. Increasingly, however, entity resolution is required in an online, real-time environment, where query records have to be matched with one or several large databases, and the most similar records are to be retrieved. One example application of current interest is health surveillance and emergency response systems, where the aim is to find all records that relate to a certain individual, for example a patient showing symptoms of an infectious disease, from a variety of databases. In order to find other individuals that might have been in contact with that patient, a search needs to be conducted in an airline database, to find the details of other people who have travelled with the patient; the database of the patient's employer, to find potentially infected co-workers; the school database of the patient's children, and so on. In many cases, the search for matching records will rely upon personal details, like the name, address and date of birth of the patient, and thus be subject to errors and variations, as well as out-of-date information. Accurate and real-time approximate matching techniques are required for such situations.

The application domain of specific interest to the authors is consumer financial services. Entity resolution is increasingly important in this domain as such services are being delivered remotely. Once a consumer has established an account with a financial institution, she or he is normally required to use an unambiguous identity token, like an account number. However, the initial establishment of a consumer's identity is difficult. The normal approach taken is entity resolution of identifying information, as provided by the consumer, against one or more databases of related identifying information. The information provided is often subject to variability and error, requiring an approximate matching process. As this process will be driven by automated systems that require sub-second responses, automated and accurate matching, scalability, and real-time entity resolution are major technical challenges for such systems.

This paper presents work that is aimed towards the development of such systems. The basic idea of achieving real-time entity resolution is to combine similarity calculations used for approximate matching with inverted index techniques that are commonly used in the field of information retrieval, for example for large-scale Web search engines [3, 29, 32]. In the past decade, with the popularity and commercial success of such search engines, a large amount of research and development on optimisation techniques has been conducted in this field [3, 32]. Some of these optimisation techniques are used in the work presented in this paper to facilitate real-time entity resolution of large databases.

The contributions of this paper are a novel index approach suitable for real-time entity resolution. This approach significantly improves the matching speed over a similar approach recently presented by two of the authors [9]. Compared to this earlier approach, which was between two and one hundred times faster than a traditional index approach for entity resolution, the novel technique presented here is consistently over two orders of magnitude faster than the traditional index approach. An important aspect of the novel approach presented here is that it allows any similarity comparison function, and any encoding function for 'blocking' [2], both possibly domain specific, to be incorporated. Most other approximate matching approaches developed in recent times are limited to specific similarity functions (such as edit distance, or Jaccard or cosine similarity), and therefore may not be suitable for entity resolution in applications that require specific encoding and comparison functions.

The remainder of this paper is structured as follows. Next, in Section 2, an overview of related research is provided. The proposed novel index approach for real-time entity resolution is then presented in Section 3, and experimentally evaluated in Section 4 using a large real-world database. The results of these experiments are discussed in Section 5, and the paper is concluded with an outlook to future work in Section 6.

2. RELATED WORK

Research into entity resolution is being conducted in various domains, including data mining, machine learning, information retrieval, artificial intelligence, digital libraries, information systems, statistics, and the database community. Several recent overview articles are available [13, 28]. Entity resolution techniques can broadly be classified into learning approaches [5, 8, 10, 11, 12], or database and graph-based methods [20, 27, 31]. So far, most research in this area has focused on the quality of the matching process, i.e. the accuracy of classifying the compared pairs or groups of records into matches and non-matches. The challenges of scalability to very large databases and real-time matching have so far only received limited attention.

For the traditional matching of (large) static databases, indexing is important, because potentially every record from one database needs to be compared with all records from the other database, resulting in a process that is of quadratic complexity in the sizes of the databases to be matched. Indexing techniques, also known as 'blocking', are therefore commonly applied to reduce the number of comparisons to be conducted. In the standard blocking approach [2], which will be presented in detail in Section 3.1 below, the databases are split into blocks according to some criteria, and only records within the corresponding block are compared with each other. A blocking criterion, also called a *blocking key*, might be based on a single record attribute (that should contain values of high quality), or based on the concatenation of values from several attributes. In order to overcome the problem of variations and errors in real-world data, one aim is to group similar sounding values into the same block. This can be accomplished by using phonetic encoding functions, such as *Soundex*, *NYSIIS* or *Double-Metaphone* [7]. These functions, which are often language or domain specific, are applied when generating the blocking keys. Examples of such phonetic encodings are shown in Figure 1.

The standard blocking approach has two major drawbacks. First, the size of the generated blocks depends upon the frequency distribution of the attribute values used in the blocking key. For example, using surname values in a blocking key will likely generate a very large block containing the common surname 'Smith', resulting in a very large number of comparisons that need to be conducted for this block. Second, if a value in an attribute used as a blocking key contains errors or variations that result in a different encoding, then the corresponding record will be inserted into a different block, and potentially true matches will be missed. This problem is normally overcome, at increased computational costs, by having two or more different blocking keys based on different record attributes.

Various alternative blocking approaches have been developed in recent times, aimed at improving the scalability of the matching process and increasing matching accuracy. With the sorted neighbourhood approach [18], the databases are sorted according to the values in the blocking key, and a fixed-size window is moved over the databases. All records within the current window will then be compared with each other. An approach related to this is to insert the blocking key values and their suffixes into a *suffix array* based inverted index [1], and to then generate blocks from all records that have the same suffix value. With this approach, each record will be inserted into several blocks, depending upon the length of its suffix values. Another approach is to allow for 'fuzzy' blocking by converting blocking key values into q-gram lists and, using sub-lists of these q-gram lists, to insert each record into several blocks according to a Jaccard-based similarity threshold [2]. While this approach can improve the accuracy of the resulting matching, its computational complexity (a large number of q-gram sub-lists need to be generated) makes it unsuitable for large databases. Another idea for indexing is to apply clustering by using a computationally efficient similarity measure to generate high-dimensional overlapping clusters (called 'canopies'), and to then extract blocks of records from these clusters [11]. Each record will be inserted into several clusters and thus several blocks, resulting in higher matching accuracy but at higher computational costs. Another recent approach is to map blocking key values into a high-dimensional Euclidean space such that the distances between all pairs of strings are preserved [19]. The records in a block then correspond to all objects in this space that are similar to each other.

Conducting entity resolution not on static databases but at query time has so far received very limited attention, with only two recent publications presenting approaches specific to such situations. The authors have earlier shown that using an inverted index approach can significantly speed-up the query matching process [9]. A second approach is based on unsupervised relational clustering, which assumes that the data to be matched contains relational information that explicitly links different types of entities [5]. The idea of this approach is to utilise the relational links between records to improve the entity resolution process. At query time, matching is conducted in an iterative fashion on a database that contains unresolved entities. While this approach can achieve much better matching accuracy compared to traditional entity resolution approaches (that only consider attribute similarities), it has much higher computational costs. Matching times of around 30 seconds for one query record on a database containing around $800,000$ records have been reported [5]. This approach is therefore impractical for real-time entity resolution on very large databases.

A large body of work has been conducted in the database community on similarity queries and their scalable and efficient implementations [4, 6, 15, 17, 23, 24, 25, 30]. Many of the presented approaches optimise indexing and filtering techniques for specific types of similarity measures, such as edit distance, or q-gram or cosine based similarities. They also mainly deal with the situation of either finding similar tuples between a set of query records and a database table, or two large tables. One real-time similarity join approach based on a modified trie hash-join has been presented very recently [22]. It calculates q-gram similarities, and then applies several filtering steps to achieve fast query times.

Thus far, scalability to very large databases has not been addressed by most recent research in the area of entity resolution, and most publications in this area have presented experimental results based on only small to medium sized data sets containing up to one million records [5, 20, 27, 31]. Most of the recently developed advanced entity resolution techniques have a computational complexity that makes them impractical for matching very large databases that contain many million records. Additionally, most approaches published so far, with the exception of two very recent techniques [5, 9], are assuming the situation of matching static databases in batch mode.

3. INDEXING FOR REAL-TIME ENTITY RESOLUTION

Indexing, as presented in the previous section, is required for real-time entity resolution systems to speed-up the matching process by reducing the number of candidate records that need to be matched with a query record.

The objective of real-time entity resolution is to match a stream of query records as quickly as possible to one or several (large) databases that contain records about existing entities, and potentially to a range of external data sources that contain additional information that can be used to verify the matched entities. The response time for matching a single query record has to be as short as possible, ideally sub-second. The matching approach must facilitate approximate matching and efficiently scale-up to very large databases that contain many millions of records. In addition, the matching should generate a match score that indicates the likelihood that a matched record in the database refers to the same entity as the query record.

Real-time entity resolution has much in common with the functionality of large-scale Web search engines. However, the databases upon which entity resolution is commonly applied do not contain Web or text documents that include a large number of terms and thus provide a rich variety of features. Rather, these databases are made of structured records with well defined attributes that often only contain short strings or numbers, such as the personal details of people (for example name, address, or date of birth values).

In this section, the traditional standard blocking approach to indexing for entity resolution is presented first to illustrate the basic ideas of using inverted indexing for entity resolution. Based on this approach, a similarity-aware inverted index approach that is suitable for real-time entity resolution is then discussed in detail in Section 3.2. Both index approaches are illustrated in Figures 2 and 3.

Both index approaches presented here are based on a standard inverted index [32], where the keys of the index are (possibly encoded) attribute values, and the corresponding lists contain the record identifiers of all records that have this (encoded) value. Two types of function are required by both index approaches. First, (phonetic) encoding functions are needed that group similar attribute values together. For string attributes, such as personal names or street and suburb names, phonetic encodings like *Soundex*, *NYSIIS* or *Double-Metaphone* are commonly used [7]. Figure 1, for example, shows the *Soundex* encodings of eight surname values. As can be seen, this encoding function groups the values 'smith' and 'smyth' into one block, and 'millar', 'miller' and 'myler' into another. The second type of functions required

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

Figure 1: Example records with surname values and their *Soundex* encodings, used to illustrate the two index approaches in Figures 2 and 3.

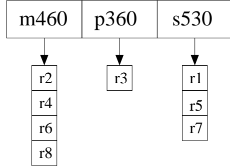

Figure 2: Standard blocking index resulting from the example records given in Figure 1. The blocking keys correspond to *Soundex* encodings.

are similarity comparisons that calculate a normalised similarity between two attribute values, such that 1 corresponds to an exact similarity and 0 to total dissimilarity [7]. Note that for different attribute types (strings, dates, numbers, etc.) different such comparison functions can be used. Additionally, domain specific comparison functions are often applied to improve the matching quality. One example of such a function would be a date of birth comparison, where a mismatch in the month or day of birth is less severe than a mismatched year of birth.

The real-time entity resolution process as discussed in this paper consists of two phases. First, in the *build* phase, an index is generated using a static database that contains a possibly large number of cleaned records that are assumed to refer to resolved entities, i.e. one single record per real-world entity only. Once built, the index is *queried* in the second phase with a stream of query records. These records can either refer to an entity stored in the index, or to a new and unknown entity. It is assumed, however, that the query records can contain variations and typographical errors, or wrong, out-of-date or missing values. Missing values can be handled by replacing them with a special character (that is outside of the character set used for an attribute) in both database and query records. For each query record, the matching process returns a ranked list of potential matches and their similarities with the query record. A match is successful if one of the top ranked records refers to the same entity as the query record.

In the following two sections, the two index approaches are described in detail, and in Section 3.3 two optimisations for reducing the query matching time are discussed. Experiments on these two index approaches are then presented in Section 4. In Algorithms 1 to 4, the attributes of a record $\mathbf{r}$ are denoted by $\mathbf{r}.i$, with $\mathbf{r}.0$ assumed to be an identifier attribute that allows unique identification of each record. A list with key k in an inverted index $\mathbf{X}$ is denoted with $\mathbf{X}[k]$. An empty list is denoted by $()$ and an empty index by $\{\}$.

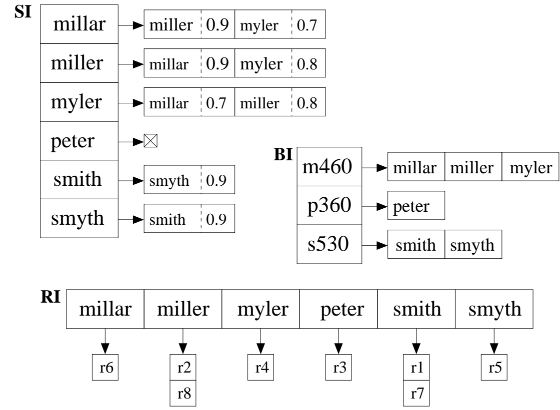

Figure 3: Similarity-aware index resulting from the example records from Figure 1. The similarity index is shown in the top left, the block index in the middle right, and the record identifier index at the bottom.

3.1 Standard Blocking

The basic idea of standard blocking is that each record in a database is inserted into a block according to the value of its blocking key [2], as illustrated in Figure 2. Encoding functions [7] are used to group similar attribute values into the same block. Each block corresponds to an inverted index list, with the key being the (encoded) blocking key value, while the values in the corresponding list are the record identifiers of all records in this block.

Standard blocking is used in this paper because it is a baseline approach upon which many other recently developed index approaches for entity resolution can be built. For example, canopy clustering [11], suffix-array blocking [1], and the sorted neighbourhood [18] approach, as discussed in Section 2, can all be implemented as extensions to a basic inverted index. Therefore, standard blocking can be seen as the basic approach for traditional batch-oriented entity resolution of static databases. Other index approaches based on it have higher computational requirements.

The build phase of standard blocking is shown in Algorithm 1. The input to this algorithm is a data set $\mathbf{D}$ containing n attributes that will be used in the entity resolution process, and n corresponding encoding functions, $\mathbf{E}_i$. A basic inverted index $\mathbf{I}$, as illustrated in Figure 2, is generated in the build phase, where record identifier values are inserted into inverted index lists according to their corresponding encoded attribute values. This encoding of attribute values might be computationally expensive. Therefore, in order to prevent repeated calculation of encodings c of attribute values, once an encoding has been computed, it is stored in the encodings cache $\mathbf{C}$ (line 9) so it can be retrieved quickly for subsequent occurrences of the same attribute value $\mathbf{r}.i$ (line 6). The algorithm returns the inverted index data structure $\mathbf{I}$ containing record identifiers, $\mathbf{r}.0$, in the inverted index lists, and the cache $\mathbf{C}$ containing the computed encodings. No similarity calculations are performed during the build phase of this index approach. Note that for simplicity the index and cache are shared between attributes. Depending upon the characteristics of the data to be processed and matched, however, it might be favourable to have separate index and cache data structures per attribute.

Algorithm 1: ***Standard blocking – Build***

Input:
- Data set: $\mathbf{D}$
- Number of attributes of $\mathbf{D}$ used: n
- Encoding functions: $\mathbf{E}_i, i = 1 \dots n$

Output:
- Standard blocking index: $\mathbf{I}$
- Encodings cache: $\mathbf{C}$

```
1:   Initialise I = {}
2:   Initialise C = ∅
3:   for r ∈ D:
4:       for i = 1...n:
5:           if r.i ∈ C then:
6:               c = C[r.i]
7:           else:
8:               c = E_i(r.i)
9:               C[r.i] = c
10:          Append r.0 to I[c]
```

Algorithm 2: ***Standard blocking – Query***

Input:
- Query record: $\mathbf{q}$
- Data set: $\mathbf{D}$
- Number of attributes of $\mathbf{D}$ used: n
- Standard blocking index: $\mathbf{I}$
- Encodings cache: $\mathbf{C}$
- Encoding functions: $\mathbf{E}_i, i = 1 \dots n$
- Similarity comparison functions: $\mathbf{S}_i, i = 1 \dots n$

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

```
1:   Initialise M = ()
2:   Initialise candidate record identifier list b = ()
3:   for i = 1...n:
4:       if q.i ∈ C then:
5:           c = C[q.i]
6:       else:
7:           c = E_i(q.i)
8:       b = b ∪ I[c]
9:   for r.0 ∈ b:
10:      Retrieve r from D using identifier r.0
11:      s = 0
12:      for i = 1...n:
13:          s = s + S_i(r.i, q.i)
14:      Append (r.0, s) to M
15:  Sort M according to similarities (largest first)
```

Algorithm 2 describes the query phase. As input, it requires a query record $\mathbf{q}$, the inverted index $\mathbf{I}$, the data set $\mathbf{D}$, the encodings cache $\mathbf{C}$, the number of attributes to be used for entity resolution n, the encoding functions $\mathbf{E}_i$, and the comparison functions used to calculate the similarities between attribute values, $\mathbf{S}_i$. The query phase consists of two steps. First, in lines 1 to 8, the encoded attribute values (possibly available in the encodings cache $\mathbf{C}$) of the query record $\mathbf{q}$ are used to retrieve the record identifier lists from the corresponding blocks in the inverted index $\mathbf{I}$. The union of these lists, $\mathbf{b}$, contains all identifiers of the candidate records that will be compared with the query record in the second step of the algorithm (lines 9 to 14). The required n attribute values for each candidate record $\mathbf{r}$ need to be retrieved from data set $\mathbf{D}$ (line 10). An efficient index on $\mathbf{D}$ is therefore required that allows fast access to a random record $\mathbf{r}$ using its identifier $\mathbf{r}.0$. For each candidate record that is compared with the query record, a similarity, s, is calculated over all compared attributes (line 13) and inserted into the list of matches $\mathbf{M}$ (line 14). For simplicity a simple summing of s is assumed, however, in reality other aggregation functions, like weighted sums, can be applied. Finally, in line 15, the list of matches $\mathbf{M}$ is sorted such that the largest similarity values are at the beginning.

3.2 Similarity-Aware Index

This index is based on the idea of pre-calculating the similarities between all unique attribute value combinations within each block once during the build phase, so that the similarities do not need to be re-calculated for every query record, thereby significantly reducing the matching time required in the query phase.

As illustrated in Figure 3, this approach contains three inverted index data structures. The record identifier index, $\mathbf{RI}$, is similar to the inverted index $\mathbf{I}$ used in standard blocking, but the keys of this index are the actual attribute values and not their encodings. The block index, $\mathbf{BI}$, is the data structure that represents the blocks by having encoded attribute values as keys and the actual attribute values that have the same encoding in the corresponding inverted index lists. Each list in this index therefore contains all attribute values that are in the same block. The similarity index, $\mathbf{SI}$, stores the similarities of pairs of attribute values that are in the same block. Specifically, for each attribute value, it contains a list of other attribute values (in the same block) and the similarities between these two values.

Algorithm 3 describes how a similarity-aware index is built. The algorithm requires the same input as the build algorithm for standard blocking. Additionally, the similarity comparison functions, $\mathbf{S}_i$, are also required, because similarity scores between attribute values are calculated during the build phase rather than the query phase. For each record $\mathbf{r}$ in data set $\mathbf{D}$, its identifier $\mathbf{r}.0$ is added to the inverted index list in $\mathbf{RI}$ that corresponds to attribute value $\mathbf{r}.i$ (line 6). It is important to note that all the following steps (lines 8 to 19) only need to be done if the attribute value $\mathbf{r}.i$ has not been processed before (line 7). This will significantly reduce the computational effort if attribute values appear in a data set several times, which is the case for attributes that have a Zipf-like or exponential distribution of values, as is common for example for attributes that contain names [9].

For a new attribute value $\mathbf{r}.i$ that has so far not been indexed, the first step (lines 8 to 11) is to calculate its encoding c and to retrieve all other values in its block. The new value is then added into the inverted index list $\mathbf{b}$ of this block, and the updated list is stored back into the block index $\mathbf{BI}$. The similarities between the new attribute value $\mathbf{r}.i$ and all attribute values v already in this block are calculated next (line 13), and inserted into both the new value's similarity list $\mathbf{si}$ (line 15) and the other value's list $\mathbf{oi}$ (line 17). Finally, the similarity list $\mathbf{si}$ of the new value $\mathbf{r}.i$ is added to the similarity index $\mathbf{SI}$ in line 19.

The query phase using the similarity-aware index is described in Algorithm 4. During the query process an accumulator $\mathbf{M}$, a data structure that contains record identi-

Algorithm 3: ***Similarity-Aware Index – Build***

Input:
- Data set: $\mathbf{D}$
- Number of attributes of $\mathbf{D}$ used: n
- Encoding functions: $\mathbf{E}_i, i = 1 \ldots n$
- Similarity comparison functions: $\mathbf{S}_i, i = 1 \ldots n$

Output:
- Record identifier index: $\mathbf{RI}$
- Similarity index: $\mathbf{SI}$
- Block index: $\mathbf{BI}$

```
1:  Initialise RI = {}
2:  Initialise SI = {}
3:  Initialise BI = {}
4:  for r ∈ D:
5:      for i = 1 ... n:
6:          Append r.0 to RI[r.i]
7:          if r.i ∉ SI:
8:              c = E_i(r.i)
9:              b = BI[c]
10:             Append r.i to b
11:             BI[c] = b
12:             Initialise inverted index list si = ()
13:             for v ∈ b:
14:                 s = S_i(r.i, v)
15:                 Append (v, s) to si
16:                 oi = SI[v]
17:                 Append (r.i, s) to oi
18:                 SI[v] = oi
19:             SI[r.i] = si
```

Algorithm 4: ***Similarity-Aware Index – Query***

Input:
- Query record: $\mathbf{q}$
- Number of attributes of $\mathbf{D}$ used: n
- Record identifier index: $\mathbf{RI}$
- Similarity index: $\mathbf{SI}$
- Block index: $\mathbf{BI}$
- Encoding functions: $\mathbf{E}_i, i = 1 \ldots n$
- Similarity comparison functions: $\mathbf{S}_i, i = 1 \ldots n$

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

```
1:  Initialise M = ()
2:  for i = 1 ... n:
3:      if q.i ∈ RI:                          // Case 1
4:          ri = RI[q.i]
5:          for r.0 ∈ ri:
6:              M[r.0] = M[r.0] + 1.0
7:          si = SI[r.i]
8:          for (r.i, s) ∈ si:
9:              ri = RI[r.i]
10:             for r.0 ∈ ri:
11:                 M[r.0] = M[r.0] + s
12:     else:                                 // Case 2
13:         c = E_i(q.i)
14:         b = BI[c]
15          for v ∈ b:
16:             s = S_i(q.i, v)
17:             ri = RI[v]
18:             for r.0 ∈ ri:
19:                 M[r.0] = M[r.0] + s
20: Sort M according to similarities (largest first)
```

fiers and their (partial) similarities with the query record, is generated [29, 32]. Two possible cases can occur for each attribute of the query record $\mathbf{q}$. The first case occurs when an attribute value is available in the index, and its similarities with other attribute values have been calculated in the build phase. In this case, in lines 4 to 6, the identifiers $\mathbf{r}.0$ 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 $\mathbf{r}.0$ will be added to the accumulator if it doesn't exist. Next, all other attribute values in the same block and their similarities with the query attribute value are retrieved from the similarity index $\mathbf{SI}$ (line 7). For each of these values, their record identifiers are retrieved from the record identifier index $\mathbf{RI}$, and their similarities are added into the accumulator in line 11.

The second case occurs when an attribute value in the query record $\mathbf{q}$ is not available in the index, and thus the similarities between this value and other attribute values need to be calculated (lines 13 to 19). This is similar to the query phase of the standard blocking index. First, in lines 13 and 14, the encoding for this unknown attribute value is calculated, and then all records in its corresponding block are retrieved from the block index $\mathbf{BI}$. In lines 16 and 17, the similarities between the attribute value from the query record and each of the other records in the block are calculated, and the record identifiers of all corresponding records are retrieved from the record identifier index $\mathbf{RI}$. The accumulator $\mathbf{M}$ is then updated in line 19 for each of these records. Finally, in line 20, the accumulator is sorted such that the largest similarities are at the beginning.

The overall efficiency of the similarity-aware index depends upon how many attribute values of the query record are already stored in the index (in which case no similarity calculations need to be performed) compared to how many are new. With increased size of the data set $\mathbf{D}$, and especially as $\mathbf{D}$ is covering larger portions of a population, one would assume that a larger portion of values would be available in the index, thereby improving the efficiency of this index approach. In Section 4 this assumption will be evaluated experimentally.

3.3 Optimisations

A variety of optimisation approaches have been developed for inverted index techniques [3, 29, 32]. These approaches apply compression to reduce the amount of memory required by the index data structure, sorting of the inverted index lists, and filtering of candidate records that are guaranteed not to be in the top ranked matches.

Currently, two such optimisation techniques are implemented in the two index approaches presented in this paper. The first is a *minimum similarity threshold*, t_{min} (with $0 < t_{min} < 1$). Within the query phase of standard blocking, this threshold is used together with the *overall minimum threshold* (discussed below) to reduce the number of matches to be stored in the ranked match list $\mathbf{M}$. Similarities, as calculated in line 13 of Algorithm 2, are not added to the overall similarity s of two records if they are below t_{min}. Within the similarity-aware index, the minimum threshold t_{min} is used in the build phase to only store similarities be-

	Given name	Surname	Suburb name	Postcode
Number of unique values	78,386	404,642	13,109	2632
Number of values with count 1	7116	193,437	931	18
Six most frequent values (and their counts)	John (149,817) Peter (116,985) David (101,859) Robert (89,564) Michael (89,222) Margaret (69,165)	Smith (65,243) Jones (32,234) Williams (31,647) Brown (31,024) Wilson (26,940) Taylor (26,044)	Toowoomba (29,127) Frankston (18,856) Croydon (15,556) Port Macquarie (15,499) Reservoir (14,784) Glen Waverley (14,756)	4350 (35,129) 4670 (24,701) 4740 (23,981) 2250 (23,454) 2170 (22,726) 4870 (21,639)

Table 1: Characteristics of the data set used for experiments.

tween attribute values that are above t_{min}. Specifically, lines 15 to 18 in Algorithm 3 are only executed if the similarity s, as calculated in line 14, is larger than t_{min}. Not storing lower similarities will reduce the memory requirements of the similarity-aware index, and also speed-up the matching time during the query phase, because the inverted index lists in the similarity index **SI** will be shorter.

The second optimisation is an *overall minimum threshold*, T_{min}, with $0 < T_{min} < n$, and n being the number of record attributes that are used in the entity resolution process. Within the standard blocking query phase, this threshold can be used in line 14 of Algorithm 2 to only append record identifiers to $\mathbf{M}$ that have a summed similarity $s \geq T_{min}$. This will reduce the size of the list of matches $\mathbf{M}$ and thus reduce the time needed to sort $\mathbf{M}$.

Within the query phase of the similarity-aware index, T_{min} is used to reduce the growth of the accumulator $\mathbf{M}$ in lines 6, 11 and 19 of Algorithm 4. Assume n attributes are being compared, resulting in a summed similarity $(n \times t_{min}) \leq s \leq n$ for each compared record pair, with only similarities between individual attribute values above t_{min} being stored in the index. When calculating the total similarity between a query record $\mathbf{q}$ and the records stored in the index, line 2 of Algorithm 4 loops over the n attributes used for the matching. With an overall threshold $T_{min} < n$, a *phase threshold*, p, can be calculated as $p = \lceil n - T_{min} \rceil$. As long as the loop counter $i \leq p$, all attribute similarities need to be added into $\mathbf{M}$, because potentially any new partial match can reach T_{min}. However, once loop counter $i > p$, no new record identifiers (and their similarities) need to be added to the accumulator, because the total similarity for these records cannot reach T_{min}. For example, assume there are four attributes to be used in the entity resolution process ($n = 4$) and $T_{min} = 2.5$, so $p = \lceil 4 - 2.5 \rceil = 2$. For the first two attributes ($i = 1, 2$), new record identifiers are added into the accumulator. However, for the third and fourth attributes ($i = 3, 4$), no new record identifiers will be added to the accumulator because even if such a new record has an exact match with the query record in both attributes three and four, the maximum total similarity of this record will be $s = 2.0$, which is below T_{min}. This optimisation can significantly reduce the final length of the accumulator.

For the standard blocking approach, a further optimisation in the query phase can be implemented if only the top matching record is (or records are) of interest. Rather than storing all matches (and their similarities) in the match list $\mathbf{M}$ (line 14 in Algorithm 2), and having to sort them before returning the ranked list (line 15), only the match(es) with the highest similarity need to be stored in $\mathbf{M}$, and no sorting will be required.

4. EXPERIMENTAL EVALUATION

The proposed similarity-aware index approach is experimentally evaluated and compared with the standard blocking approach. The issues of interest were the time used to build the index and query it with records of varying quality, and the accuracy of the retrieved matches. The experiments were conducted on a Linux server containing two Intel Xeon quad-core 64-bit CPUs with 2.33 GHz clock frequency, 8 Gigabytes of main memory, and two SAS drives (446 Gigabytes in total). No other users were logged onto this machine, and no other jobs were run during the experiments.

A large real-world data set containing 6,917,514 records was used for the presented experiments. It contained surnames, postcodes and suburb (town) names sourced from an Australian telephone directory from 2002 (*Australia On Disc*[1]). This data corresponds to all entries in Australian telephone books in late 2002, and thus has characteristics similar to many other real-world data collections used by Australian organisations. Additionally, a list containing about 80,000 different given names and their frequencies of occurrence, supplied to the authors by a major Australian government agency, was used to generate and add a given name attribute. For each record in the data set, a given name was randomly selected (with replacement) from the given name list according to its frequency, and appended to the record. As such, this is a typical example data set that contains a large number of unique and cleaned entities, with similar data being collected by many other private and public sector organisations in many countries.

Table 1 provides an overview of the resulting data set used in the presented experiments. As expected, all the name attributes exhibit a strongly skewed distribution of values, with a small number of very common values and a large number of very rare values. For example, 40% of all surnames only appear once in the data set, while the top five most frequent surnames account for nearly 7% of the population. Only postcodes are more uniformly distributed, which is due to the process by Australia Post to split populated regions into similar sized postcode areas.

Both index approaches were implemented in Python, with version 2.5.2 used for the experiments. For the encoding functions $\mathbf{E}_i$, used to block the test data sets, the Double-Metaphone [7] phonetic encoding was applied on the three name attributes, while for the postcode attribute the blocking was based on selecting the last three digits (i.e. all records where the postcode value has the same last three digits were inserted into the same block). For the comparison functions, $\mathbf{C}_i$, the Winkler [7] approximate string comparison was used for the three name attributes, while for postcodes the sim-

[1] `http://www.australiaondisc.com`

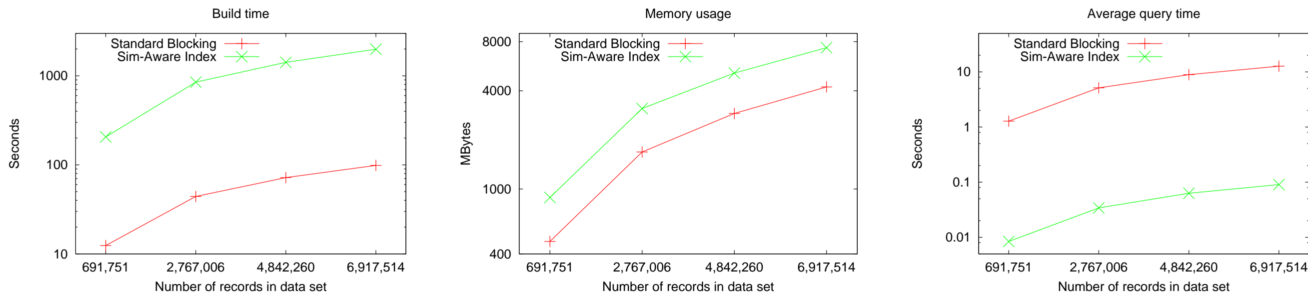

Figure 4: Summary experimental results: Build time (left); memory usage (middle); and average query time per record (right). Note that all three graphs are shown with a logarithmic y-axis scale.

ilarity was calculated by counting the number of matching digits divided by four. For example, the similarity of the two postcode values '2346' and '2356' is 0.75.

In order to evaluate the scalability of the similarity-aware index, test data sets of four different sizes were built containing 10% (691,710), 40% (2,767,006), 70% (4,842,260) and 100% of the records in the original data set. The full data set was split into ten data sets of equal size. Next, from each of these ten data sets, ten query records were randomly selected (giving one hundred base query records in total). To assess the matching quality, five query sets of one hundred records each were created by transforming the hundred base records in different ways. The first set of hundred records were made by exactly copying the base query records. In the second set one modification was inserted into one of the four attributes in each record (a different attribute in each record in a round robin fashion); in the third set two modifications were inserted; in the fourth set three modifications, and in the fifth query set all four attribute values were modified in each record. The modifications, while done manually, were based on the authors' experience with real-world name data. They mostly corresponded to common phonetic and typographical variations, for example changes such as 'Dickson' to 'Dixon', nickname substitutions like 'Robert' to 'Bob', or simple character inserts, deletes, substitutions or transpositions. For postcodes, only substitutions and transpositions of digits were applied, such as '2607' changed into '2601'.

Scalability was evaluated by building an index for each test data set (each containing 10%, 40%, 70%, or 100% of the records in the full data set) and then querying it with each of the five query sets. The time used to build each index was recorded, as well as the total amount of memory used by that index. During the query phase, the time for querying each record was measured, as well as whether the top ranked returned record was a true match (i.e. if the record identifier of the best returned match was the same as the record identifier of the query record). For the similarity-aware index, the number of case 1 and case 2 matches (as discussed in Section 3.2 and shown in Algorithm 4) was also recorded. While test runs were conducted with both optimisations turned off and on, due to space limitations of this paper only results with activated optimisations are reported. The minimum threshold t_{min} was set to 0.55 and the overall minimum threshold T_{min} to 2.0. These values were selected such that the experiment on the full database for the similarity-aware index still fitted into the 8 Gigabytes main memory available on the experimental platform.

For the experiments with the smaller test data sets (less than 100% of the full data set), each experiment was conducted ten times (with component 10% data sets selected in a round robin fashion) and all results averaged, while for the full database an experiment was only run once.

5. RESULTS AND DISCUSSION

A summary of the experimental results is shown in Figure 4. As expected, building a standard blocking index is significantly faster than building a similarity-aware index, by a factor ranging from 16 times for the smallest test data set to 20 times when building the index for the largest test data set. The main reason for this is that during the build phase of the standard blocking index no similarity calculations between attribute values are performed. The build time for both index approaches however does grow sub-linearly with the size of the data set. For standard blocking, this is because the encodings of attribute values are cached (line 6 in Algorithm 1), so the more records are loaded and inserted into the index, the more often cached encoding values can be retrieved and fewer need to be calculated. For the similarity-aware index, the calculation of similarities between attribute values and inserting them into the similarity and blocking indices **SI** and **BI** again only needs to be done the first time a new, previously unseen attribute value occurs.

Similarly, the amount of memory required by both index approaches (shown in the middle of Figure 4) grows sub-linearly with the size of the test data set, because as the data set grows fewer new attribute values, which need to be processed and stored, will occur. For the test data sets used in the experiments, the similarity-aware index required around 1.8 times as much memory on average as the standard blocking index. The rate of growth for both build time and memory requirements depends upon the distribution of attribute values in the data set to be indexed. Given that many real-world databases contain attributes that follow a Zipf-like or exponential distribution, such as names [9], a sub-linear growth can be expected in practice. A theoretical analysis of the growth factor is one avenue of future work planned by the authors.

One of the most important aspects of the novel index approach presented in this paper is its fast query matching time. As can be seen in the right graph in Figure 4, the novel approach achieves average query times below 0.1 seconds even for the index that is based on the full test data set containing nearly 7 million records. Over the different test

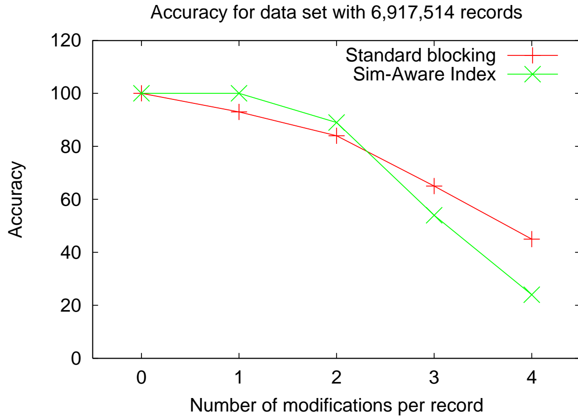

Figure 5: Query matching accuracy for the full test data set for varying number of modifications per record. Similar accuracy results were achieved for the smaller test data sets.

Given name	Surname	Suburb name
Gail (g400)	Billman (b455)	Boystown (b235)
Gayle (g400)	Pillman (p455)	Boydtown (b350)
0.827	0.905	0.942

Figure 6: An example record pair that will be missed by the similarity-aware index approach because of different encoding values, but will be compared by standard blocking. The values in brackets are the corresponding *Soundex* encodings, and the similarities (bottom row) were calculated using the *Winkler* approximate string comparison function [7].

data sets (10%, 40%, 70% and 100% of the full data set size), the query time for the similarity-aware index is between 140 and 150 times faster than standard blocking. However, for both index approaches, the query time currently increases linearly with the size of the indexed data sets. Improving upon this is a current effort by the authors.

The query matching accuracy results are shown in Figure 5 for the largest test data set with varying number of modifications per record. As can be seen for both index approaches, matching accuracy gets lower with an increased number of modifications. This is what one would expect, as with more modifications per record the likelihood that another record (with similar attribute values) becomes the best matching record is increased.

The accuracy for the similarity-aware index is higher compared to standard blocking for the query sets with one and two modifications, but then drops more rapidly for the query sets with three and four modifications. This is due to the requirement of the similarity-aware index that the values of all attributes for a record pair need to be in the same block in order to have their similarity added to the accumulator. If two attribute values are in different blocks, then the corresponding similarity, which can be high, will not be considered. For standard blocking, on the other hand, only one pair of attribute values needs to be in the same block in order that two records are being compared.

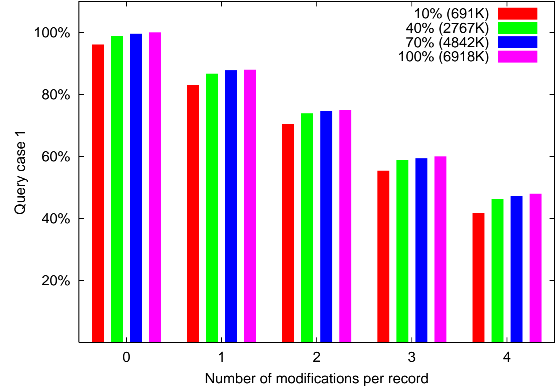

Figure 7: Proportion of case 1 (query attribute value is available in similarity-aware index) to case 1 plus case 2 (new unknown attribute value) for varying number of modifications per record.

This is illustrated in Figure 6 with two example records that have an overall similarity of 2.674 out of a maximum of 3.0. These records would be compared by standard blocking because at least one attribute (given name) contains values that are in the same block; whereas they would not be compared by the similarity-aware index, because two of the three attributes (surname and suburb name) have different blocking key values and thus the corresponding similarities would not be added into the accumulator.

Although this effect may lead to standard blocking having higher accuracy on more heavily modified query records, it can also lead to lower accuracy for standard blocking compared to the similarity-aware index when query records are of relatively good quality, as can be seen in Figure 5 for the query sets with one or two modifications only. Improving the similarity-aware index and achieving equal or even better matching accuracy than standard blocking in all cases is one of the current research efforts by the authors.

Finally, Figure 7 shows the proportion of query attribute values that were available in the similarity-aware index (case 1) and thus no similarities had to be calculated at query time. As can be seen, the more modifications a query record had, the more likely the modified attribute values were not in the index and thus their similarities had to be calculated. However, even with modifications in all four query record attributes, more than 40% of all attribute values were available in the index and thus their similarities were pre-calculated. This results shows the efficiency of the similarity-aware index in speeding up query matching by pre-computing similarities between records while the index is built.

6. CONCLUSIONS AND FUTURE WORK

In this paper, a novel index approach for real-time entity resolution has been presented and evaluated experimentally on a large real-world data set. The experiments showed that this approach can match query records more than two orders of magnitude faster than a basic standard index approach that is traditionally used for entity resolution. The novel approach requires less than double the amount of memory of the standard index, but building the index can take up-to twenty times longer.

For query records that do not contain too many variations and errors, the accuracy of the novel index approach can be better than the standard blocking approach. However, when most or all attribute values in a query record contains variations and errors, then matching accuracy can drop significantly. Improving upon this drawback is one of the major avenues for additional work on this novel index approach. Other areas of future research include a theoretical analysis of the complexity and scalability of this index approach, improving the query matching time, and conducting experiments on a variety of other real-world databases.

To the best of the authors' knowledge, the similarity-aware inverted index presented in this paper is the first approach aimed at developing real-time entity resolution on large databases that combines approaches from information retrieval with traditional entity resolution techniques.

7. REFERENCES

[1] A. Aizawa and K. Oyama. A fast linkage detection scheme for multi-source information integration. In *WIRI'05*, Tokyo, 2005.

[2] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *ACM SIGKDD'03 Workshop on Data Cleaning, Record Linkage and Object Consolidation*, Washington DC, 2003.

[3] R. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW'07*, Banff, Canada, 2007.

[4] A. Behm, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search. In *IEEE ICDE'09*, pages 604–615, Shanghai, China, 2009.

[5] I. Bhattacharya and L. Getoor. Query-time entity resolution. *Journal of Artificial Intelligence Research*, 30:621–657, 2007.

[6] M. Celikik and H. Bast. Fast error-tolerant search on very large texts. In *ACM Symposium on Applied Computing*, pages 1724–1731, Honolulu, Hawaii, 2009.

[7] P. Christen. A comparison of personal name matching: Techniques and practical issues. In *Workshop on Mining Complex Data, held at IEEE ICDM'06*, Hong Kong, 2006.

[8] P. Christen. Automatic record linkage using seeded nearest neighbour and support vector machine classification. In *ACM SIGKDD'08*, pages 151–159, Las Vegas, 2008.

[9] P. Christen and R. Gayler. Towards scalable real-time entity resolution using a similarity-aware inverted index approach. In *AusDM'08, CRPIT vol. 87*, Glenelg, Australia, 2008.

[10] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IJCAI'03 Workshop on Information Integration on the Web (IIWeb)*, pages 73–78, Acapulco, 2003.

[11] W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *ACM SIGKDD'02*, pages 475–480, Edmonton, Canada, 2002.

[12] M. Elfeky, V. Verykios, and A. Elmagarmid. TAILOR: A record linkage toolbox. In *IEEE ICDE'02*, pages 17–28, San Jose, 2002.

[13] A. Elmagarmid, P. Ipeirotis, and V. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[14] I. Fellegi and A. Sunter. A theory for record linkage. *Journal of the American Statistical Society*, 64(328):1183–1210, 1969.

[15] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB'01*, pages 491–500, Roma, Italy, 2001.

[16] L. Gu and R. Baxter. Decision models for record linkage. In *Selected Papers from AusDM, Springer LNCS 3755*, pages 146–160, 2006.

[17] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *IEEE ICDE'08*, pages 267–276, Cancun, Mexico, 2008.

[18] M. A. Hernandez and S. J. Stolfo. The merge/purge problem for large databases. In *ACM SIGMOD'95*, San Jose, 1995.

[19] L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. In *DASFAA'03*, pages 137–146, Tokyo, 2003.

[20] D. Kalashnikov and S. Mehrotra. Domain-independent data cleaning via analysis of entity-relationship graph. *ACM Transactions on Database Systems*, 31(2):716–767, 2006.

[21] C. Kelman, J. Bass, and D. Holman. Research use of linked health data – A best practice protocol. *Aust NZ Journal of Public Health*, 26:251–255, 2002.

[22] M. Kumar, S. Moriah, and S. Krishnamoorthy. Performance evaluation of similarity join for real time information integration. In *Bangalore Annual Compute Conference*, Bangalore, India, 2009.

[23] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *IEEE ICDE'08*, pages 257–266, Cancun, Mexico, 2008.

[24] C. Li, B. Wang, and X. Yang. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB'07*, pages 303–314, Vienna, Austria, 2007.

[25] X. Liu, G. Li, J. Feng, and L. Zhou. Effective indices for efficient approximate string search and similarity join. In *IEEE WAIM'08*, pages 127–134, 2008.

[26] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4), 2000.

[27] M. Weis and F. Naumann. Space and time scalability of duplicate detection in graph data. Technical Report 25, Hasso-Plattner-Institut, University of Potsdam, Germany, 2007.

[28] W. E. Winkler. Overview of record linkage and current research directions. Technical Report RR2006/02, US Bureau of the Census, 2006.

[29] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and indexing documents and images*. Morgan Kaufmann, 2nd edition, 1999.

[30] C. Xiao, W. Wang, X. Lin, and J. Yu. Efficient similarity joins for near duplicate detection. In *WWW'08*, pages 131–140, Beijing, 2008.

[31] X. Yin, J. Han, and P. Yu. Linkclus: Efficient clustering via heterogeneous semantic links. In *VLDB'06*, pages 427–438, Seoul, Korea, 2006.

[32] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 2006.