Robust Record Linkage Blocking using Suffix Arrays

Timothy de Vries
School of I.T.
University of Sydney
NSW, Australia
timothy.devries@gmail.com

Hui Ke
School of I.T.
University of Sydney
NSW, Australia
hui.ke.it@gmail.com

Sanjay Chawla
School of I.T.
University of Sydney
NSW, Australia
chawla@it.usyd.edu.au

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

ABSTRACT

Record linkage is an important data integration task that has many practical uses for matching, merging and duplicate removal in large and diverse databases. However, a quadratic scalability for the brute force approach necessitates the design of appropriate indexing or blocking techniques. We design and evaluate an efficient and highly scalable blocking approach based on suffix arrays. Our suffix grouping technique exploits the ordering used by the index to merge similar blocks at marginal extra cost, resulting in a much higher accuracy while retaining the high scalability of the base suffix array method. Efficiently grouping similar suffixes is carried out with the use of a sliding window technique. We carry out an in-depth analysis of our method and show results from experiments using real and synthetic data, which highlights the importance of using efficient indexing and blocking in real world applications where data sets contain millions of records.

Categories and Subject Descriptors

H.3 [**Information Storage And Retrieval**]: Content Analysis and Indexing—*Indexing methods*

General Terms

Algorithms, Performance

Keywords

Record Linkage, Blocking, Suffix Arrays

1. INTRODUCTION

Record linkage is an essential data integration technique that is increasing in importance as more and more data is collected and stored. This technique can be applied to any situation where two or more sets of data need to be linked together and there is an absence of a uniquely identifying key across these data sets. Alternatively, the same linking approaches can be used to find matches among records in the same data set for the purposes of duplicate removal [18]. In both of these cases, the main problem that must be overcome is the presence of noise and small differences between records in the data that are to be matched together. Record linkage is therefore an approximate matching technique, aiming to provide the best possible match given the available information. These linkage tasks are common and crucial early steps in most large data mining projects. Their main use is to provide a wealth of information that is not readily available under the standard practice of keeping multiple separate databases for archival or analysis purposes.

As the availability and quantity of data grows over time, so do the number of databases that are created and ultimately discarded as legacy systems. These are all useful sources of information if effort is undertaken to link them together. More specifically, record linkage can drastically increase the information available for purposes such as large medical health systems [9], business analytics, fraud detection [17], demographic tracking, government administration, and even national security.

Record linkage systems typically consist of three main parts. Blocking is first used to select candidate records to match against the selected record. Record comparison is then carried out, usually using specifically tailored similarity functions. A classification model is then used to determine whether the two records are a match or a non-match. The matching process is by nature a very computationally intensive task. For matches or duplicates to be found across two data sets of size n, up to n^2 comparisons may be required, making a brute force nesting approach infeasible in practice with large data sets. It is therefore very important to consider techniques to reduce the number of pairwise comparisons that must be made, by making some assumptions about which factors of the data will almost always lead to matches or non-matches. This indexing process is referred to as *blocking* in the context of record linkage. Blocking techniques can be tailored to favour either the accuracy or efficiency of the linkage task, in a tradeoff manner. The main aim is to make record linkage feasible on large data sets by greatly reducing the number of record pair comparisons that need to be carried out, at the cost of a usually small loss in accuracy. The design of a robust blocking technique based on suffix arrays is the focus of this paper.

CIKM'09, November 2–6, 2009, Hong Kong, China.

A large variety of blocking methods have been created, with each one providing at least a niche benefit for specific data types. One of the first methods to be proposed uses a basic exact-match index on a few key fields (chosen record attributes), and for any one record that requires matching, selects *candidate* records to match against based on exact matches in one of these key fields. This approach is known as 'traditional' blocking [3]. This basic approach nevertheless can boast a high accuracy, at the expense of a relatively low matching efficiency. However, the high accuracy and simple implementation make this the method of choice in many industrial settings. The authors of this article are involved with the production of a large-scale data mining system that utilises record linkage in the industry. Traditional blocking was chosen for this application, and labeled data is available due to the model training process that has been carried out. However, we are able to use this real labeled data to test the performance of different blocking techniques in a direct experimental comparison.

An important reason for choosing a non-traditional blocking technique is due to the ubiquitous presence of errors in the data. The use of standard exact matches will miss a correct record match even with the slightest difference in field values. To compensate for this, many indexed fields need to be used, and the set of candidate record pairs rapidly increases in size, adversely affecting the time required to carry out the matching. The most important goal of record linkage is to find these similar but non-identical matches without adversely reducing performance by such a large amount, and therefore specialised techniques designed to index data of this kind are necessary. One such highly-efficient existing blocking method is called Suffix Array blocking [1], which is our basis for an improved blocking method that retains the high efficiency of Suffix Array blocking while also increasing accuracy to an acceptable level by avoiding the misclassification of records into incorrect blocks.

We give a small example to demonstrate our proposed improvement to the Suffix Array blocking method. Given two records r_1 = 'John Smith, 10 Plum Road' and r_2 = 'John Snith, 10 Plom Rd.' to match against each other, we select given name and surname as the key blocking fields. Concatenating the values of these fields together results in the strings b_1 = 'JohnSmith' and b_2 = 'JohnSnith'. These strings are called Blocking Key Values (BKVs) in the context of record linkage. When the *minimum suffix length* parameter is set to 4 (to be discussed in Section 2.2), Suffix Array blocking will generate suffixes 'mith', 'Smith', 'nSmith', 'hnSmith', etc. for b_1, and similarly for b_2. The suffixes are added to the indexing structure and sorted, as shown in Table 1.

While highly efficient, standard Suffix Array blocking is not able to match records that exhibit qualities such as in this example, where none of the suffixes from r_1 and r_2 match. Our proposed improvement takes effect when the suffixes are added to the indexing structure. This structure holds an alphabetically sorted list of suffixes to enable fast querying for matches against any given input suffix, and is used to find candidate records for matching against any new record. The suffixes 'nSmith' and 'nSnith' as well as 'hnSmith' and 'hnSnith', among others, are adjacent to each other in the ordered list for this example. By comparing adjacent suffixes and grouping together those that exhibit a high degree of similarity, we can carry out a form of clustering or *grouping* of the blocking result. By grouping similar suffixes from r_1 and r_2 we can ensure that these records are added to the same block, which is the desired outcome and and improvement over standard Suffix Array blocking. We cover the improvement in more detail in Sections 3 and 4.

Suffix	Record #	Jaro similarity
hnSmith	1	0.809
hnSnith	2	0.756
JohnSmith	1	0.851
JohnSnith	2	0
mith	1	0.833
nith	2	0
nSmith	1	0.777
nSnith	2	0.722
ohnSmith	1	0.833
ohnSnith	2	0
Smith	1	0.733
Snith	2	-

Table 1: Suffixes generated from two blocking key values (BKVs), where record 1 corresponds to the BKV of 'JohnSmith' and record 2 to the BKV of 'JohnSnith'. Minimum suffix length is 4. Jaro similarity refers to the Jaro [13] measure as a similarity function between the suffix string in one row and the suffix in the following one.

1.1 Contributions

Our main contribution is a substantial improvement to the Suffix Array blocking technique [1], together with an in-depth analysis and experimental results showing the effectiveness of the technique on real and synthetic data. We compare the improved technique against the base technique of Suffix Array blocking, as well as the well-known traditional blocking method. We show that Improved Suffix Array blocking is able to attain a level of accuracy similar to the highly accurate traditional blocking technique, as well as being able to acquire this result by making many fewer full comparisons between data records, comparable to the highly efficient Suffix Array blocking method. More importantly, with careful parameter selection in our experiments, we found that Improved Suffix Array blocking can attain a 20% increase in accuracy over standard Suffix Array blocking, with less than a 5% loss in efficiency. The accuracy of Improved Suffix Array blocking also remains within 10% of traditional blocking, while enjoying an efficiency increase of more than 95% over this standard technique. A high efficiency becomes more important as the data set used increases in size, a critical point for data in the order of hundreds of millions of records. Scenarios of this type are common in the industry, including the domain of the large-scale record linkage project that we are comparing our results against.

Section 2 introduces the relevant work related to record linkage. In Section 3 we describe our proposed improvements to standard Suffix Array blocking, and carry out a more in-depth analysis in Section 4. Section 5 details our experimental method and parameters, the results are discussed in Section 6, and we summarise our conclusions in Section 7.

2. BACKGROUND

The term 'record linkage' was first used by Dunn [6] and Marshall [15], and Fellegi and Sunter [8] proposed a theory based on statistical classification.

Recent advances in record linkage were undertaken by Aizawa [1], and processes for record linkage projects and methods were improved by Christen [4]. Indexing techniques, or *blocking* methods as they are known in the context of record linkage, were quickly recognised as a key component for efficiency purposes. Blocking algorithms typically contain extra functionality over standard indexing, to solve specific record linkage issues. Blocking is especially important due to the inherently high n^2 scalability of unoptimised record linkage. Blocking solutions strive to reduce the number of candidate records for comparison as much as possible, while still retaining an accurate result by ensuring that candidate records that would match the query record are not left out of the candidate set due to the blocking rules.

A variety of blocking methods are currently used in record linkage procedures, with the most well-known ones including traditional blocking, sorted neighbourhood [11], Q-gram based blocking [3], Canopy Clustering [16], string map based blocking [14] and Suffix Array blocking [1].

All blocking methods define a set of *key fields* from the data to be matched, that are used to determine which block (or blocks) each record is to be placed into. Many of these approaches require a single string to be used as the key on which to find the correct block. Therefore, the values of the key fields are typically concatenated together into one long string. This string is called the Blocking Key Value (BKV) [10]. The selection of key fields to include in the BKV as well as the ordering of these fields is important to consider. A suitable BKV should be the attribute or combination of attributes which are as identifying as possible, uniformly distributed, and having a low error probability.

Hernandez and Stolfo [11] proposed the sorted neighbourhood blocking method. This approach begins by sorting the input data, then moves a sliding window of size w over the data file, comparing records against each other if they exist in the window at the same time. The time complexity of the sorted neighbourhood method is $O(n \log n + wn)$ where n is the number of records in each of the two data sets being linked.

The Q-gram based blocking method is achieved by transforming the blocking key values into lists of q-grams and creating all combinations of sub-lists. Christen [3] proposed that q-gram based blocking can achieve better blocking quality results than both standard blocking and the sorted neighbourhood approach. However, the number of sub-lists generated depends upon the value of the parameter q, the length of the sub-strings used. The time complexity of Q-gram based blocking is $O(n \log n + \frac{n^2}{b})$ where n is the number of records in each of the two data sets, and b is the number of generated blocks [3].

McCallum et al. [16] first proposed Canopy Clustering, a solution for clustering large, high dimensional data sets. This method can be applied to blocking by creating blocks containing records which can be found within the same canopy cluster. Experimental results are shown for linking bibliographic citations from the reference sections of research papers [16]. The time complexity of Canopy Clustering is $O(\frac{nkf^2}{c})$, where n is the number of records, c is the number of canopies, f is the average number of canopies a record belongs to, and k is the number of clusters searched for using k-means (based on the original data) [3].

String map based blocking is based on mapping the blocking key values (assumed to be strings) to objects in a multidimensional Euclidean space. Jin et al. [14] modified the FastMap algorithm to create StringMap, which has a linear complexity in the number of strings to be mapped.

Christen [4] compared and evaluated these blocking techniques, and modified two of them to make them more robust with regards to parameter settings, an important consideration for any algorithm that is to be considered for real-world applications. The experimental results showed that there are large differences in the number of true matched candidate record pairs generated by the different techniques, when tested using the same data sets. It was also discovered that many of these algorithms were unstable with the selection of the parameter values. We limit the introduction of new parameters with our proposed improvement in order to avoid this potential problem.

Record linkage is a large research area containing many important aspects [2][12]. Our paper focuses exclusively on the sub-area of finding effective blocking techniques.

2.1 Traditional Blocking

Traditional blocking is well-known and is often used in practical applications. This approach works by only comparing against records that have the same blocking key value, for example, only comparing records that have the same postcode [13]. The blocking keys are usually chosen to be very general in order to produce a high quality result, while also producing a reasonable reduction in the amount of data required to compare against for each record to be matched. Usually more than one key field is chosen to build the blocking key value. In the industrial record linkage application used for comparison, the selected key fields are given name, surname, and date of birth. When carrying our matches with the record 'John Smith, 01/01/1960, 10 Plum Road', traditional blocking will select all candidate records that exactly match with 'John' on given name, plus all candidate records that exactly match with 'Smith' on surname, plus all records that exactly match with '01/01/1960' on date of birth. A common modification to add a degree of 'fuzziness' to the matching is done by applying phonetic encoding (such as Soundex) to the key fields.

One major weakness of traditional blocking is that errors in all of the blocking key values will result in records being inserted into the wrong block. Secondly, the size of each block is generally quite large, causing many unnecessary comparisons to be carried out. Finally, another drawback is that the sizes of the blocks generated depend upon the frequency distributions of each individual field used in the the blocking key value [4]. When fields are combined into a BKV such as for regular Suffix Array blocking, a drastic reduction in block size is typically encountered. The time complexity of traditional blocking is $O(dn \log n)$ where n is the number of records in each of the two data sets that are being matched and d is the number of key fields chosen [7].

2.2 Suffix Array Blocking

Akiko Aizawa and Keizo Oyama [1] proposed the Suffix Array blocking technique as a fast and efficient blocking method for large scale record linkage. We utilise the Suf-

fix Array Blocking plus Key Blocking approach from this paper, with an additional adjustment to handle string characters as the individual tokens. Analysis of this technique against many recent alternatives [4] found that the efficiency gain is very high for this method, but the accuracy can suffer with standard data sets and when the blocking key value is chosen by concatenating several key fields, as is the standard for comparison.

The main idea of Suffix Array blocking is to insert blocking key values and their variable length suffixes into a suffix array based inverted index [4]. For example, when the *minimum suffix length* parameter (l_{ms}) is 4, a BKV of 'JohnSmith' will be decomposed into the suffix array containing the suffixes 'mith', 'Smith', 'nSmith', 'hnSmith', 'ohnSmith' and 'JohnSmith'. These suffixes are then inserted into the indexing structure and sorted in alphabetical order. An example inverted index containing suffixes generated from the BKVs of 'JohnSmith' and 'JohnSnith' is shown in Table 1. The purpose of the indexing structure is to find a set of references to original records that contain a certain suffix, when queried with that suffix.

After generating BKVs, generating suffix arrays from these BKVs, and inserting the suffixes from these suffix arrays into the indexing structure, one further optimisation step is carried out. An additional parameter is introduced for this purpose, *maximum block size* (l_{mbs}). The problem that can be introduced with low values of l_{ms} is that some words may all feature a common suffix (e.g. 'ing' in the English language). This occurrence can result in the block for common suffixes to be extremely large, and this has a significant adverse effect of the efficiency of standard Suffix Array blocking. Therefore, a blanket rule is introduced to remove any particular block entirely if it contains references to more than l_{mbs} records. The technique retains accuracy by allowing the correct blocking of records that share short but rare suffixes, while excluding matching short suffixes that are common. Since each input BKV is decomposed into multiple suffixes, the removal of one of many redundant 'sub-blocks' does not adversely affect the recall of the result.

Former studies [1] have found that Suffix Array blocking is efficient primarily due to the small but highly relevant set of candidate record pairs that are produced. Another reason is the low complexity of the Suffix Array algorithm compared to some traditional blocking method implementations [7]. A further advantage of Suffix Array blocking over traditional blocking is that it is not prone to blocking key value errors that appear near the beginning of the BKV. If errors occur, usually not all of the suffixes of these BKVs will change, only some of the longer ones. One record will be inserted into several blocks, adding a form of redundancy to try to ensure that true matched record pairs are grouped into the same block.

Suffix Array blocking is able to solve the problem of fields with a large frequency in the database, and avoid excessive processing times for these records [4]. One example of this occurs when matching against the record with high frequency values of 'John' and 'Smith'. When traditional blocking is used, the candidate set will consist of all records that have a first name of 'John', as well as all records that have a surname of 'Smith'. This can be an extremely large set when large real world databases are used, with the intrinsic problem due to common occurrence of these records. There do exist a few solutions that help to improve the excessive time taken for records of this type, however. The process of combining more than one field for use as the blocking key causes the candidate set for 'John Smith' to be greatly reduced from the traditional blocking method approach, as the number of records highly similar to 'John Smith' is always much less than the number of records with 'John' as first name plus the number of records with 'Smith' as last name in most normal data sets. Improved Suffix Array blocking inherits these benefits. In practical terms, this functionality is important in near-realtime systems where a user may query for records that match a specific input. In situations like these, it can be disadvantageous for a query consisting of common terms to take an excessively longer time than normal, as would be the case with traditional blocking.

2.3 Blocking Measurement

Accuracy measurement for blocking tasks is usually carried out with the use of the pairs completeness measure (PC) [4]. This measure is the ratio of the number of true matches that the blocking algorithm correctly includes in the candidate set to be matched, and the total number of true matches that exist in the dataset and that would all be found when no blocking is used. If true matches are denoted by N_m, blocking denoted matches by S_m, and blocking denoted non-matches by S_u, then pairs completeness is given as:

$$PC = \frac{S_m}{N_m}$$

Pairs completeness measures the *recall* of the blocking technique. In [4], the pairs quality measure (PQ) is proposed as a way to measure the 'reduction ratio' or efficiency of the blocking technique. Pairs quality is a measure of *precision*, measuring the proportion of chosen candidates for matching that actually correspond to true matches. It is given as:

$$PQ = \frac{S_m}{S_m + S_u}$$

3. AN IMPROVEMENT FOR SUFFIX ARRAY BLOCKING

Suffix Array blocking is designed for efficiency, with the ability to select a very concise set of candidates for matching. However, this comes at the expense of the accuracy, or pairs completeness, of the result. The main weakness of this technique is due to the creation of the array of suffixes. Under standard Suffix Array blocking, the chosen key fields are concatenated into the BKV string. An array of suffixes is then generated from the BKV by taking suffixes of increasing length. Since every suffix created from the BKV includes the last character of this string, a difference at the last position of a BKV when compared to an original BKV will cause standard Suffix Array blocking to place the differing record into a separate block than the original, causing a valid comparison to be left out of the candidate set for the matching step. An extension of this problem occurs when the minimum suffix length parameter l_{ms} is too large. An example of this can be seen in Table 1 when minimum suffix length is 4. Careful selection of this parameter's value is therefore essential.

3.1 Improving Suffix Array Using Grouping

The Suffix Array blocking method is suitable for a wide range of applications, but has one large limitation. If two BKV substrings are identical apart from an error positioned less than l_{ms} characters away from the end of the BKV string, standard Suffix Array blocking will fail to group these records into the same block. The 'JohnSmith' and 'JohnSnith' example shown in Table 1 contains this property when the minimum suffix length parameter l_{ms} is 4. However, it is clear from this example that many of these adjacent suffixes share a high degree of similarity.

We propose an approach towards solving this problem, by carrying out a *grouping* operation on similar suffixes in the ordered suffix index list. Many methods can be used for grouping or clustering these suffixes. However, the time complexity of the indexing method is important to consider in order to avoid an overall scalability decrease for the record linkage problem. In particular, we have to avoid a large number of comparisons between the BKV suffixes. In the worst case, we can expect nk BKV suffixes when matching among n records where the average BKV length is k (larger values of l_{ms} will reduce this). A full comparison among all of these records will therefore result in a time complexity of $O((kn)^2)$ for the suffix grouping operation. In a way, the problem we now face is very similar to the original goal of reducing the number of comparisons we have to carry out among the n original records, by instead needing to find a way to reduce the number of comparisons we have to make for the task of linking together kn suffixes.

However, we can utilise the nature of suffix generation along with the necessary step of building the indexing structure for linkage to greatly optimise this process. In the example in Table 1 we want to avoid comparisons among suffixes that were generated from the same BKV. However, we would like to carry out comparisons between similar suffixes that were generated from different BKVs. Each suffix in the suffix array generated from a single BKV will be similar to the other suffixes from that BKV, with the differences occurring near the start of the suffix. As it turns out, the suffixes are required to be ordered before they can be used in the indexing structure which is used to select candidates for matching, and indeed, the ordering is usually carried out by the data structure employed. This requirement therefore automatically disperses suffixes that were generated from the same BKV throughout the list. This behaviour can be seen in Table 1, where the record number of each row alternates between 1 and 2, the identifiers for the BKVs of 'JohnSmith' and 'JohnSnith' respectively.

It can also be seen from this example that the indexing structure has a tendency to place similar suffixes from different BKVs next to each other. This is useful from an efficiency point of view when attempting to group together similar BKV suffixes. A simple method for grouping that does not cause adverse scalability reductions can be implemented by only checking directly neighbouring records when carrying out the grouping. When a close match is found, the blocks can be merged together. There exists an option to use a larger *sliding window* [11] when processing the suffixes, to compare suffixes that may match closely but be separated by one or two alphabetically similar suffixes. For efficiency, we group only neighbouring suffixes in our experiments, effectively using a window size of 1. Larger values may be selected to increase accuracy with diminishing returns at the cost of efficiency. The grouping technique therefore exploits the alphabetical ordering required by the indexing structure. This technique cannot be easily applied to the highly similar prefix array blocking method, unless the prefix strings or the ordering comparison function are inverted. Our approach is detailed in Algorithm 1. The standard Suffix Array algorithm is equivalent to this one, minus the grouping step.

We carried out experiments using the Jaro string comparison function [13] as well as the Longest Common Subsequence (LCS) operator as the similarity metric used to decide whether to merge two BKVs together. We found that if a specific comparison function is used in the full comparison of two records, this function may be a good choice for the grouping operation as well. The Jaro string comparison function was found to be more well-suited to our problem, and we show only the experiments that were run using this similarity measure.

Algorithm 1 Improved Suffix Array Blocking

Input:
1. R_p and R_q, the sets of records to find matches between.
2. The suffix comparison function similarity threshold j_t.
3. The mimimum suffix length l_{ms} and the maximum block size l_{mbs}.

Let ***II*** be the inverted index structure used.
Let C_i be the resulting set of candidates to be used when matching with a record r_{pi}

// Index construction:
For record $r_{qi} \in R_q$:
  Construct BKV b_{qi} by concatenating key fields
  Generate suffixes a_{qi} from b_{qi}, where
  $a_{qi} = \{s_{q1}, s_{q2}, \ldots, s_{qy}\}, |a_{qi}| = y = |b_{qi}| - l_{ms} + 1$
  and $s_{qj} = b_{qi}.\text{substring}(|b_{qi}| - l_{ms} - j + 1, |b_{qi}|)$
  For suffix $s_{qij} \in a_{qi}$:
    Insert s_{qij} and a reference to r_{qi} into ***II***

// Large Block Removal
For every unique suffix s_f in ***II***:
  If the number of record references paired with $s_f > l_{mbs}$:
    Remove all suffix-reference pairs where the suffix is s_f

// Suffix grouping (Improved Suffix Array only)
For each unique suffix s_f in ***II*** (sorted alphabetically):
  Compare s_f to the previous suffix s_g using
  the chosen comparison function (e.g. Jaro)
  If $Jaro(s_f, s_g) > j_t$: (highly similar)
    Group together the suffix-reference pairs
    corresponding to s_f and s_g using
    set join on the two sets of references

// Querying to gather candidate sets for matching:
For record $r_{pi} \in R_p$:
  Construct BKV b_{pi} by concatenating key fields
  Generate suffixes a_{pi} from b_{pi}, where
  $a_{pi} = \{s_{p1}, s_{p2}, \ldots, s_{py}\}, |a_{pi}| = y = |b_{pi}| - l_{ms} + 1$
  and $s_{pj} = b_{pi}.\text{substring}(|b_{pi}| - l_{ms} - j + 1, |b_{pi}|)$
  For suffix $s_{pj} \in a_{pi}$:
    Query ***II*** for a list of record references that match s_{pj}
    Add these references to the set C_i (no duplicates)

If, however, LCS is desired as a similarity function, it can be incorporated without the requirement of an extra parameter. If s_1 and s_2 are the two input suffixes, l_1 and l_2 are the lengths of these suffixes, l_{lcs} is the length of any result of the LCS operation, and l_{ms} is the minimum suffix length parameter, then we can define the grouping result as:

$$Grouping(s_1, s_2) = \begin{cases} 1 & \text{if} \quad max(l_1, l_2) - l_{lcs} < l_{ms} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Allowing up to a difference in length of l_{ms} between the longest BKV string and the LCS result has the effect of allowing groupings between records that would be erroneously omitted due to errors in the last l_{ms} characters of the BKV under standard Suffix Array blocking, while avoiding spurious grouping results that decrease pairs quality unnecessarily and which would likely be removed in the record linkage full comparison step due to low similarity.

3.2 Complexity

Pairs completeness and pairs quality are not the only measurements of interest for comparing different blocking techniques. These measurements do not take into account the computational complexity underlying the algorithm used. We analyse the computational complexity changes introduced due to the grouping technique in this section.

Standard Suffix Array blocking will generate gn suffixes on average, if k is the average BKV length, $g = k - l_{ms} + 1$ and n is the number of records to match with one another. An indexing structure is used to allow for $O(gn \log gn)$ construction and $O(\log gn)$ query time for a single record. In the worst case of $l_{ms} = 1$ and every suffix being grouped together, the indexing structure will contain k suffix keys, each referencing n data set items, causing query time for a single record to equal $O(kn \log kn)$. However, with a normal data set, the indexing structure is usually able to separate records into distinct blocks and allow for an $O(b \log nk)$ query time, where b is a value that depends on the data set used, with data containing more potential linkages having a higher b.

The addition of the grouping operation has an effect on both the construction of the indexing structure as well as the query operation. For index construction, the list of suffixes of length kn must be traversed once. Grouping results can be stored by modifying the inverted index on the fly. While the time taken to construct the indexing structure may be slightly longer in practice due to grouping, it does not affect the intrinsic computational complexity that is required.

The time complexity of the querying stage is usually more important than index construction, and the grouping result has an effect on this stage as well. For any set of suffixes generated from the query BKV, the goal is to extract the set of record identifiers to be used to select the candidate record set for matching. Each suffix query of the indexing structure takes an expected $O(\log gn)$ time. Grouping adds extra record identifier results to this step, but the computational complexity is not modified if the window size is fixed (in our experiments it is fixed to 1). Under our proposed technique, the number of additional grouping results is limited to the chosen window size. Therefore, the time taken may be slowed by this small constant factor at this stage due to grouping, but again, the time complexity with regards to n or k is unchanged.

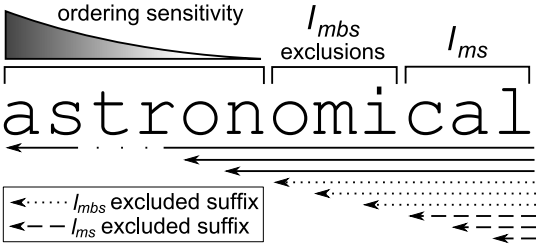

Figure 1: An example showing suffix exclusion due to l_{ms} and l_{mbs}. Ordering sensitivity is also shown.

4. ANALYSIS

We carry out a more thorough analysis into the time complexity of the proposed Improved Suffix Array blocking method, as well as adding insight into why it is effective and where it may fall short. We describe the approach in detail in Algorithm 1, which includes definitions for each component used in this section.

Standard Suffix Array will miss a correct blocking if there is a mistake within the last l_{ms} characters of the duplicate BKV. It will also miss a correct blocking if the mistake occurs within a suffix that is common enough to be excluded due to the maximum block size condition. This condition acts as a way to dynamically extend the minimum suffix length based on the rarity of the suffixes towards the end of each specific word. If one suffix is excluded due to the maximum block size condition, all smaller suffixes from the same word are excluded as well. These two suffix exclusion rules combine to exclude a continuous set of suffixes from one position up to the end of the BKV string. This behaviour is shown with an example in Figure 1.

We can build a model to estimate the probability of various types of errors that can occur between a true BKV and a 'dirty' duplicate, such as character replacement, insertion, deletion, or swapping. Given the definitions above, we can simplify our model by assuming that the true BKV b_p and the dirty duplicate b_q have the same length. We can then assume that the probability for a difference between b_p and b_q to occur at any character position to be c. The 'l_{mbs} exclusions' area in Figure 1 acts as a way to dynamically extend l_{ms} based on the rarity of the suffixes towards the end of the BKV, and changes in size in different BKVs. However, we can simplify our model by assuming that the average length of the longest suffix excluded due to the maximum block size condition is l_{se} over all record BKVs in the data set, where $l_{se} \geq l_{ms}$ and l_{se} can be visualised as the combined length of the 'l_{mbs} exclusions' and 'l_{ms}' regions in Figure 1. We then have the probability for standard Suffix Array blocking to miss a potential match between two records as:

$$P_{\text{suffix-miss}}(b_p, b_q) = 1 - (1 - c_i)^{l_{se}} \tag{2}$$

Improved Suffix Array blocking is able to solve this problem under two conditions. The main requirement is that at least one suffix from each of the two BKVs being compared must be adjacent to each other in the ordered suffix list used in the inverted index structure. With a sliding window approach instead of a strict adjacency rule, we can extend the rule to allow 'closeness' given by half of the window length instead of strict adjacency. The probability for Improved Suffix Array blocking to miss a grouping is difficult to quan-

tify, as it depends on the type of data used. However, the act of generating multiple suffixes from a single BKV results in a significant amount of redundancy that we can exploit. It turns out that in the vast majority of cases, at least one suffix from each BKV end up close together, allowing the grouping of the two BKVs to occur. This redundancy is a key to the robustness of the grouping approach, as the process is able to handle multiple errors in the BKV strings, and multiple missed suffix groupings, as long as only one suffix from each BKV matches up. Examples of BKVs producing suffixes that are dispersed throughout the alphabetically ordered inverted index are shown in Tables 1 and 2. From these examples it is clear that highly related suffixes from each BKV end up close to each other in the inverted index list.

We define *complete grouping separation* as the effect that can occur when grouping fails. A simple example occurs when selecting the two BKVs 'handbag' and 'handtag', the latter of which is a misspelling of the first. If there exists other BKVs that are ordered alphabetically between these two BKVs, such as the string 'handlebars', there exists the potential to separate some of the suffix strings ordered indexing structure. However, each BKV will have many suffixes, all of which are compared to their alphabetically sorted neighbors for grouping. For *complete grouping separation* to occur, separation must occur for every single suffix and the neighbor we would like to group it with. The example given above exhibits these characteristics, with the suffix 'andlebars' separating 'andbag' and 'andtag', and the suffix 'ndbar' separates 'ndbag' and 'ndbar', etc, as can be seen in Table 2. Nevertheless, this type of scenario is very unlikely to occur in practice, as the BKV which carries out the separation requires three characteristics. Firstly, the beginning of the separating BKV must be identical to the two BKVs that should have been grouped together. This type of behaviour is usually quite rare, typically occurring when words are constructed from multiple parts ('hand' and 'bag'), and especially uncommon when records consist mostly of names (in the case of identity matching). Secondly, the differences in the two BKVs that should be grouped must occur within the last l_{ms} characters, otherwise they will be grouped due to sharing a common suffix. Thirdly, the end of the separating BKV must be significantly different from both of the two BKVs that should have been grouped together. It can be seen from Table 2 that even in this specifically constructed example, the similarity is quite high, and grouping could still occur if the similarity threshold for grouping is reduced.

Complete grouping separation, while rare, can be reduced even further by extending the *window size* of the grouping operation. In our experiments, the result is good even when the window size is limited to 1. Extending the window size will increase the number of candidate selected for matching, usually reducing the pairs quality. Additionally, the time complexity of the grouping operation is $O(nk)$ when the window size $w = 1$, but $w = 2$ will cause the grouping operation to double in cost compared to $w = 1$, so for practical applications, lower values of this parameter are necessary to allow for a rapid grouping operation. More complex window-based techniques can also be used, such as the approach described by Yan et al. [19].

The second requirement for grouping to successfully occur is simply that the two BKVs being compared must exhibit enough similarity to pass the similarity check that the grouping operation carries out. The process is robust to multiple errors in the dirty duplicate BKV, allowing grouping if only one suffix matches up to a suffix from the clean BKV. However, this suffix match must pass the string similarity check (e.g. Jaro similarity). In some examples, all suffix comparisons may contain too many errors to be grouped together. This loss of a true match is typically unavoidable, as every blocking method will find it difficult to block together two records with these characteristics, and even if the records are entered into the same block, the full record comparison carried out by the record linkage process would likely discard the two records as a non-match, or at least assign a very low similarity score.

Suffix	Record #	Jaro similarity
...		
andbag	1	0.796
andlebars	**3**	0.703
andtag	2	-
...		
dbag	1	0
dlebars	**3**	0
dtag	2	-
...		
ndbag	1	0.766
ndlebars	**3**	0.658
ndtag	2	-
...		

Table 2: An occurrence of *complete grouping separation*. Two BKVs that should be blocked together are 'handbag' and 'handtag'. However, the suffixes of a third BKV 'handlebars' will separate all suffixes of the original two BKVs, causing complete separation and therefore Improved Suffix Array will not be able to improve its result over standard suffix array, due to the optimisation that only allows grouping of adjacent suffixes. Minimum suffix length (l_{ms}) must be 3 or more for this condition to occur for this example. Jaro similarity refers to the similarity between the suffix string on one line and the suffix on the following one.

Finally, the use of the grouping technique guarantees that no loss in pairs completeness will occur compared to standard Suffix Array blocking.

LEMMA 1. *The pairs completeness (recall) of Improved Suffix Array blocking (ISAB) is always greater than or equal to that of Standard Suffix Array blocking (SSAB).*

PROOF. Let T_i be the true matching records of record r_i. Let C_i^{ssa} and C_i^{isa} be the candidate records matched by SSAB and ISAB respectively. Then $PC^{ssa}(r_i) = \frac{|C_i^{ssa} \cap T_i|}{|T_i|}$ and $PC^{isa}(r_i) = \frac{|C_i^{isa} \cap T_i|}{|T_i|}$ Now, as SSAB groups on exact matching suffixes, and ISAB adds approximate suffixes to this result, $C^{ssa} \subseteq C^{isa}$. Thus, $PC^{isa}(r_i) \geq PC^{ssa}(r_i)$ □

The reverse is not true for pairs quality. In most cases, a slight loss in pairs quality will occur when utilising grouping. However, as is evident in some of our results using synthetic data, some situations do occur where both pairs completeness and pairs quality is higher when using Improved Suffix Array over standard Suffix Array blocking (E.g. Figure 5).

5. EXPERIMENTS

Our experiments are designed to compare Improved Suffix Array blocking against standard Suffix Array blocking as well as traditional blocking, primarily using the measurements of pairs completeness and pairs quality. We run the experiments on two real data sets as well as a synthetic one. The real data sets are sourced from an insurance company where a large-scale record linkage module exists as part of a larger surveillance system. The 'identity' data set consists of personally identifying information such as names and addresses. The synthetic data set was generated using the Febrl [5] tool with standard settings. Even though the source for the real data contains millions of records, examples need to be hand labeled to produce accurate test data sets. Therefore, we were only able to obtain $n = 4135$ records for the real identity data set. For our results to be comparable, we used $n = 5000$ records for the synthetic data set. We use larger data sets in our performance comparison experiments.

Our experiments are conducted as follows:

1. We vary l_{ms} while keeping $l_{mbs} = 12$.
2. We vary l_{mbs} while keeping $l_{ms} = 6$.
3. We utilise a large scale database implementation to measure the performance of all methods on a large set of data from the real identity database, with parameter values of $l_{ms} = 6$ and $l_{mbs} = 12$. Desktop Specifications: Intel Xeon 3.6GHz, 3.25GB RAM. Database Server: Intel Xeon X5460 3.16GHz, 8GB RAM.
4. We vary the BKV composition to demonstrate that our results are consistent for different BKV compositions, using the parameter values of $l_{ms} = 6$ and $l_{mbs} = 12$.

All experiment results shown use Jaro for the grouping similarity function, and the threshold for determining Jaro similarity between two strings is set at 0.85 for all experiments.

6. RESULTS

The results from varying l_{ms} can be seen in Figures 2 and 3. The results from varying l_{mbs} are displayed in Figures 4 and 5. It is clear that for a good selection of l_{ms} we can obtain a very high pairs completeness (accuracy), while achieving a pairs quality (efficiency) very similar to the highly efficient standard Suffix Array blocking. It is also clear from the pairs quality results that a large time saving can be achieved by utilising Improved Suffix Array blocking over the traditional blocking method, while experiencing only a slight loss in accuracy or pairs completeness. The advantages for using Improved Suffix Array blocking over standard Suffix Array blocking are most notable in the experiments using synthetic data, which turned out to be more 'dirty' with a higher frequency of errors compared to the real data. This shows the robustness of Improved Suffix Array blocking as it is able to gracefully handle data with more errors and mistakes. This quality may be a key advantage in some data environments where error-sensitive techniques cannot be used.

Our performance experiment over a large selection from the real identity data set is shown in Figure 6. It is clear that the scalability of both Suffix Array techniques out-perform traditional blocking. Also of interest is the extremely low amount of extra processing required to carry out the grouping aspect of Improved Suffix Array blocking, both in index construction and querying.

Results from our experiments where we changed the *feature set* of fields used to construct the BKV are shown in Figure 7 and 8. Different BKV combinations show consistent results. Of interest here are the results achieved from the feature selections which contain 'suburb' at the end of the concatenated BKV. For the synthetic data, the suburb field contained errors that may occur typographically if the field is captured in the real dataset using free text entry. However, our real dataset utilises a list of suburbs that contains fixed entries, and the operator must select the correct suburb from this list. Therefore, there are no typographical errors in the suburb field of our real dataset. Therefore, when 'suburb' is used as the last string to be concatenated into the BKV, most of the short suffixes generated from different records are exactly the same. The suburb field therefore does not have enough *discriminating power* to be used at the most important position in the concatenation of strings to form the BKV for the real dataset.

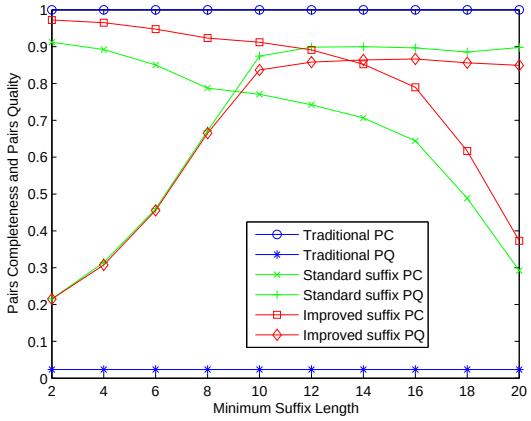

Figure 2: Pairs completeness and pairs quality obtained while varying minimum suffix length (l_{ms}) on the real identity data set.

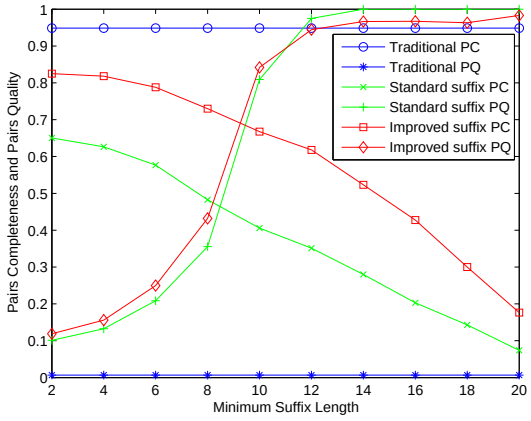

Figure 3: Pairs completeness and pairs quality obtained while varying minimum suffix length (l_{ms}) on the synthetic data set.

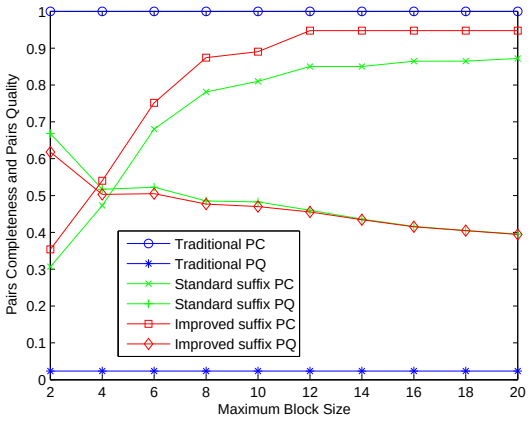

Figure 4: Pairs completeness and pairs quality obtained while varying maximum block size (l_{mbs}) on the real identity data set.

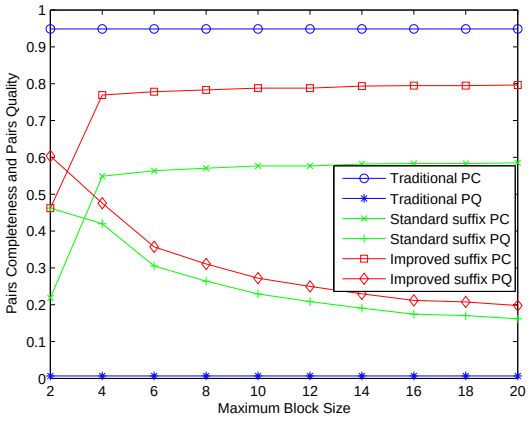

Figure 5: Pairs completeness and pairs quality obtained while varying maximum block size (l_{mbs}) on the synthetic data set.

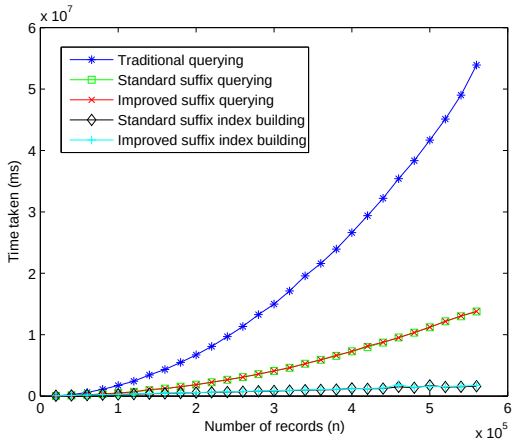

Figure 6: Overall running time on a large set of real identity data. Both Suffix Array techniques require an index construction step as shown.

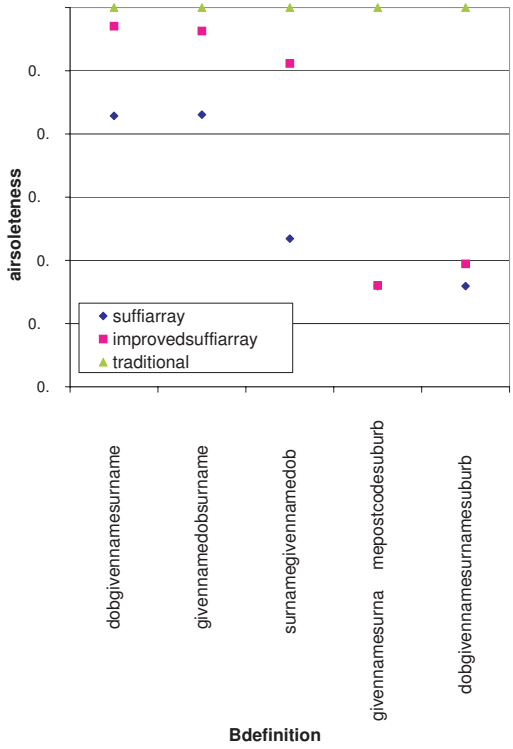

Figure 7: Pairs completeness for different BKV combinations, using the real identity data set.

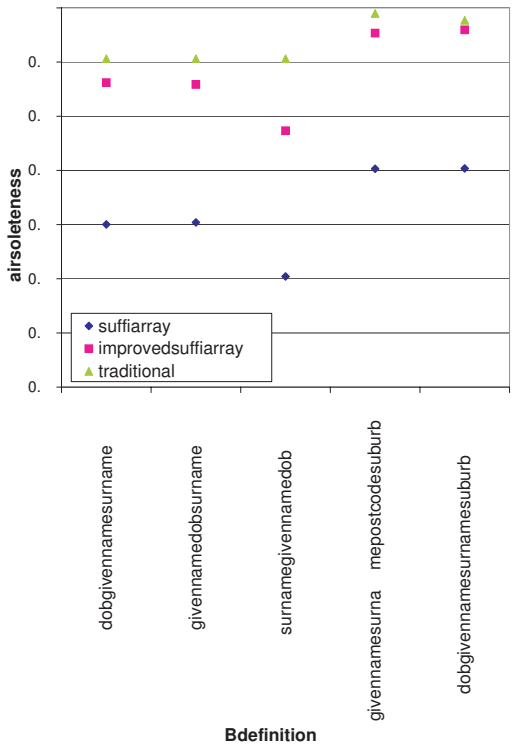

Figure 8: Pairs completeness for different BKV combinations, using the synthetic data set.

7. CONCLUSION

Suffix Array blocking is highly efficient and able to outperform traditional methods in scalability, at the cost of a significant amount of accuracy, depending on the attributes of the data used. Our improvement inherits these qualities, but significantly improves the accuracy at the cost of a very small amount of extra processing. The qualities of Improved Suffix Array blocking make it well-suited for large-scale applications of record linkage. Our experimental results show that our approach is much more scalable than the traditional approach for data sets containing millions of records. This is a common situation in many industrial applications where many large data sets exist, both current and archival, and it is beneficial to bring data together from different sources in order to increase the amount of knowledge that is available to inform and drive decisions. Scalability becomes a matter of feasibility for very large scale record linkage tasks. It is also a critical property for high-performance and real-time applications. For this approach, the average query time is also important, which may fluctuate significantly when traditional blocking is used. A given example for identity matching was that querying 'John Smith' will take much longer than some rare names, often by several orders of magnitude. Improved Suffix Array blocking is able to overcome this problem and can avoid excessive query times for records with common field values.

We have also shown that the accuracy or pairs completeness of Improved Suffix Array blocking is much higher than standard Suffix Array blocking for the data sets we used in our experiments. In fact, Improved Suffix Array is able to achieve a result highly similar to the highly accurate traditional blocking method. This shows the strength of the additional grouping process that is carried out on the sorted list of suffixes in the indexing structure, even when we limit our implementation to an efficient one that does not take into account the position of differences within the BKVs being compared, and does not compare suffixes more than one record away in the ordered suffix list. However, further improvements could be designed to utilise these additional sources of information.

8. ACKNOWLEDGEMENTS

Timothy de Vries and Sanjay Chawla acknowledge the financial support of the Capital Markets CRC.

9. REFERENCES

[1] A. Aizawa and K. Oyama. A fast linkage detection scheme for multi-source information integration. In *WIRI '05: Proceedings of the International Workshop on Challenges in Web Information Retrieval and Integration*, pages 30–39, Washington, DC, USA, 2005.

[2] C. R. Arvind Arasu and D. Suciu. Large-scale deduplication with constraints using dedupalog. In *ICDE '09: Proceedings of the 25th International Conference on Data Engineering*. IEEE Computer Society, March 2009.

[3] 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.

[4] P. Christen. Towards parameter-free blocking for scalable record linkage. Technical Report TR-CS-07-03, Department of Computer Science, The Australian National University, Canberra, 2007.

[5] P. Christen. Febrl – An open source data cleaning, deduplication and record linkage system with a graphical user interface (Demonstration Session). In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD'08)*, pages 1065–1068, Las Vegas, 2008.

[6] H. L. Dunn. Record linkage. In *American Journal of Public Health*, pages 1412–1416, 1946.

[7] M. Elfeky, V. Verykios, and A. Elmagarmid. Tailor: a record linkage toolbox. *Data Engineering, 2002. Proceedings. 18th International Conference on Data Engineering*, pages 17–28, 2002.

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

[9] L. Gill, M. Goldacre, H. Simmons, G. Bettley, and M. Griffith. Computerised linking of medical records: methodological guidelines. *J Epidemiol Community Health*, 47(4):316–319, 1993.

[10] L. Gu, R. Baxter, D. Vickers, and C. Rainsford. Record linkage: Current practice and future directions. Technical report, CSIRO Mathematical and Information Sciences, 2003.

[11] M. A. Hernandez and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[12] L. Huang, L. Wang, and X. Li. Achieving both high precision and high recall in near-duplicate detection. In *CIKM '08: Proceeding of the 17th ACM conference on Information and knowledge management*, pages 63–72, New York, NY, USA, 2008. ACM.

[13] M. A. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa. *Journal of the American Statistical Association*, 84(406):414–420, June 1989.

[14] L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. In *DASFAA '03: Proceedings of the Eighth International Conference on Database Systems for Advanced Applications*, page 137, 2003.

[15] J. T. Marshall. Canada's national vital statistics index. In *Population Studies*, pages 204–211, 1947.

[16] A. Mccallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Knowledge Discovery and Data Mining*, pages 169–178, 2000.

[17] H. B. Newcombe and J. M. Kennedy. Record linkage: making maximum use of the discriminating power of identifying information. *Commun. ACM*, 5(11):563–566, 1962.

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

[19] S. Yan, D. Lee, M.-Y. Kan, and L. C. Giles. Adaptive sorted neighborhood methods for efficient record linkage. In *JCDL '07: Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, pages 185–194, New York, NY, USA, 2007. ACM.