RECORD LINKAGE IN THE INDUSTRY:
APPLICATIONS OF AN IMPROVED SUFFIX
ARRAY BLOCKING METHOD
TECHNICAL REPORT 633

TIMOTHY DE VRIES, HUI KE AND SANJAY CHAWLA
SCHOOL OF INFORMATION TECHNOLOGIES
THE UNIVERSITY OF SYDNEY

FEBRUARY 2009
Record Linkage in the Industry: Applications of an Improved Suffix Array Blocking Method

Timothy De Vries  
School of Information Technologies  
University of Sydney, Australia  
timothy.devries@gmail.com

Hui Ke  
School of Information Technologies  
University of Sydney, Australia  
hui.ke.it@gmail.com

Sanjay Chawla  
School of Information Technologies  
University of Sydney, Australia  
chawla@it.usyd.edu.au

Abstract—Record linkage is an important data mining task that has seen many uses in the industry, from matching records in separate datasets to removing duplicates in the same dataset, in the absence of a uniquely identifying key. However, for any feasible implementation, an indexing or blocking technique must be used to reduce the computational complexity of this task, which is quadratic in the number of records to be linked when unoptimised. We propose an improvement on the Suffix Array blocking method by grouping together similar suffixes. We carry out an in-depth analysis of our method and show results from experiments using real and synthetic data. We compare our results with the blocking technique used in a record linkage technique that exists in the industry, which is also the source of our real-world dataset.

I. INTRODUCTION

Record linkage is an essential data mining technique that is increasing in importance. 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 datasets. A second common use is to use the same linking rules to find matches among records in the same dataset, for the purposes of duplicate removal[17]. 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 is 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, and often provide a wealth of information that is not readily available with the standard business practice of keeping multiple separate databases for archival purposes.

The benefit that record linkage can give is often underestimated. As the availability and quantity of data grows over time, so do the number of databases that are created and discarded. 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[8], business analytics, fraud detection[15], demographic tracking, government administration[16], and even national security.

Record linkage systems typically consist of three main parts. A model is used to determine whether two records are a match or a non-match. Similarity functions are usually defined to specify the degree to which two records are similar, which the model will then use to determine the matching result. A third critical aspect is the indexing or blocking component, which aims to make record linkage feasible in practice by greatly reducing the number of comparisons that need to be carried out, at the cost of a usually small loss in accuracy.

Record linkage is by nature a very computationally intensive task. For matches to be found across two datasets of size $N$ and $M$, $N$ times $M$ pairwise comparisons need to be made, and for duplicates to be removed from a database of size $N$, $N^2$ comparisons are required. For this reason, the time required to carry out unoptimised record linkage quickly scales upwards and becomes infeasable in practice with large datasets. 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 usually 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.

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, 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. 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.

The selection of an appropriate blocking technique is highly dependent on the data format of the records to be linked. However, there do exist some general techniques that can be easily applied to any data format. These approaches typically combine selected key fields of the data into a descriptive
string of characters. This information can then be processed to determine which of many ‘blocks’ the record can be indexed into. One of these approaches where we focus our improvement technique, is called Suffix Array blocking. We focus on the three techniques of traditional blocking, standard suffix array blocking, and our proposed improvement to the suffix array method, to allow for extensive results and analysis sections.

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 Smith, 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 $b_2$ = ‘JohnSmith’. The suffix array technique will generate suffixes ‘mith’, ‘Smith’, ‘nSmith’, ‘hnSmith’, etc. for $b_1$, and ‘nith’, ‘Snith’, ‘nSnith’, ‘hnSnith’, etc. for $b_2$, when the minimum suffix length parameter is set to 4. None of these suffixes match, and therefore standard suffix array will cause a legitimate comparison between $r_1$ and $r_2$ to be missed, a comparison that would have resulted in a match if it had been carried out. Our proposed improvement takes effect when the suffixes are added to the indexing structure, and sorted into alphabetical order. The indexing structure holds a sorted list of suffixes to enable fast querying for matches against any given input suffix, and this structure 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, will be adjacent to each other in the ordered list for this example. If we compare adjacent suffixes, we can easily see that many adjacent suffixes in this example share similarities with each other. We can use this knowledge to carry out a form of clustering or grouping of the blocking result. We group two records into the same block if some of their suffixes are adjacent in the ordered suffix list and contain a high degree of similarity. The grouping result for this example will lead to $r_1$ and $r_2$ being added to the same block, which is the desired outcome and and improvement over the standard suffix array technique. We cover the improvement in more detail in sections III and IV.

A. Contributions

Our main contribution is an improvement to the suffix array blocking technique, 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 the improved suffix array technique 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 technique. A high efficiency becomes more important as the dataset used increases in size, a critical point for data in the order of hundreds of millions of rows, as is the case for the large-scale record linkage implementation in the industry that we are directly comparing our proposed technique to.

B. Organisation

Section II introduces the relevant work related to record linkage. In section III we describe our proposed improvements to the standard suffix array technique, and carry out a more in-depth analysis in section IV. Section V details our experimental method and parameters, and section VI displays the results. We conclude the paper in section VII.

II. BACKGROUND

The term ‘record linkage’ was first used by Dunn[5] and Marshall[13]. Fellegi and Sunter[7] developed a mathematical model to provide a theory describing the construction and properties of the optimal rule for record linkage in the early 70s.

Early advances in record linkage were undertaken by Aizawa[1], and processes for record linkage projects and methods were improved by Christen[3]. A key component that was recognised as being required for large-scale record linkage was identified as an indexing solution or, as it is called in the context of record linkage, a blocking algorithm. 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 unoptimized 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. Current blocking methods

A variety of blocking methods are currently used in record linkage procedures, with the most well-known ones including traditional blocking, sorted neighbourhood, Q-gram based blocking, Canopy Clustering with TFIDF, string map based blocking and suffix array blocking.

All blocking methods define a set of key fields from the data to be matched, that are used in some way to determine which block 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). 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[9].

Hernandez and Stolfo[10] proposed the sorted neighbourhood blocking method. This approach begins by sorting the input data according to some metric, 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.
Experimental results demonstrate that this approach may work accurately in practice but with a significant cost in efficiency. Only the merge part of the merge/purge problem has been analysed in this paper. The time complexity of the sorted neighbourhood method is defined to be $O(n \log n + w n)$ where $n$ is the number of records in each of the two datasets 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[2] 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 $O(n \log n + \frac{n^2}{q})$ where $n$ is the number of records in each of the two datasets, and $b$ is the number of generated blocks [2].

McCallum et. al.[14] first proposed Canopy Clustering, a solution for clustering large, high dimensional datasets. 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[14]. The time complexity of Canopy Clustering is described to be $O(n + \frac{f^2}{c})$, where $n$ is the number of records in each of the two data sets, $c$ is the number of canopies and $f$ is the average number of canopies a record belongs to[2].

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.[12] modified the FastMap algorithm to create StringMap, which has a linear complexity in the number of strings to be mapped.

Peter Christen[3] 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 datasets. It was also discovered that many of these algorithms were unstable with the selection of the parameter values. We do not introduce any new parameters with our proposed improvement in order to avoid this potential problem.

### B. Traditional blocking

One of the techniques that we compare against in this paper, traditional blocking is one of the most well-known and is often used in practical applications. Traditional blocking works by only comparing against records that have the same blocking key value, for example, only comparing records that have the same postcode[11]. 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. The industrial application of large-scale record linkage that we are comparing our technique to selects three key fields, namely 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.

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 blocking key value [3]. The time complexity of traditional blocking is $O(d n \log n)$ where $n$ is the number of records in each of the two datasets that are being matched and $d$ is the number of key fields chosen[6].

### C. Suffix Array blocking

Akiko Aizawa and Keizo Oyama[1] proposed the Suffix Array technique as a fast and efficient blocking method for large scale record linkage. Analysis of this technique against many recent alternatives[3] found that the efficiency gain is very high for this method, but the accuracy can suffer with standard datasets 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[3]. For example, a BKV of ‘abc’ will be decomposed into the suffix array containing the suffixes ‘bc’ and ‘abc’ when the minimum suffix length parameter ($l_{ms}$) is 2. Similarly, ‘bcd’ will be decomposed into ‘cd’ and ‘bcd’.

These four suffixes are then inserted into the inverted index, or indexing structure, and sorted in alphabetical order. 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. These suffix arrays are variable length n-grams on a suffix array representation[1].

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. The problem that can be introduced with low values of Minimum Suffix Length is that some words may all feature a common suffix, such as ‘ing’. This occurrence can result in the block for common suffixes such as ‘ing’ to be extremely large, and this has a significant adverse effect of the efficiency of the suffix array blocking method. Therefore, a basic rule is introduced to remove any particular block entirely if it contains references to more than Maximum Block Size records. The technique retains accuracy
by allowing the correct blocking of records that share common rare suffixes, even if they are short, while excluding very common suffixes. Since each input BKV is decomposed into multiple suffixes, the removal of the block for some of these suffixes does not adversely affect the result.

Experiments conducted by former studies show that suffix array blocking is an efficient method for multi-source information integration[1]. Apart from the high reduction ratio compared to other blocking methods, another reason for this is the low complexity of the suffix array algorithm compared to some traditional blocking method implementations[6]. Another advantage of suffix array blocking over traditional blocking is that it is not prone to blocking key value errors. If errors occur in the BKVs, 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 at some stage.

D. Blocking Measurement

Accuracy measurement for blocking tasks is usually carried out with the use of the Pairs Completeness measure (PC)[3]. This measure is simply 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 would all be found when using no blocking. If true matches are denoted by \( N_m \), blocking denoted matches by \( S_m \), true non-matches by \( N_u \) and blocking denoted non-matches by \( S_u \), then Pairs Completeness is given as:

\[
PC = \frac{S_m}{N_m}
\]

The measure of efficiency is typically carried out with the use of the Reduction Ratio measure (RR)[3]. This is simply the ratio of the number of records in the candidate set as decided by the blocking technique, and the total number of records that would be in the candidate set and matched against when no blocking technique is used. The result is then taken as the difference from 1, to cause a higher RR to equal a more desirable result. This measure is given as:

\[
RR = 1 - \frac{S_m + S_u}{N_m + N_u}
\]

III. AN IMPROVEMENT FOR SUFFIX ARRAY BLOCKING

Suffix array is an efficient blocking method with a high reduction ratio. However, this comes at the expense of the quality, or pairs completeness, of the result. The main weakness of this technique is due to the creation of the array of suffixes. Under the suffix array technique, the chosen key fields are concatenated into a string, called the blocking key value (BKV). An array of suffixes, called the suffix array, 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 to place the differing record into a separate block than the original, causing a valid comparison to be left out of the candidate set when matching.

This is due to the design of the suffix array method, which selects substrings from the BKV in such a way as to create at the most \( m^3 \) substrings if the length of the BKV is \( m \). There exists the potential to select \( m^m \) combinations of characters from the BKV, any of which may exhibit similarities with other combinations from other similar records to be matched. However, algorithms that select a large number of BKV substrings may cause the record linkage technique to scale adversely with the size of the blocking key values, \( m \). As well as an increased time requirement, the space requirement of the indexing structure used will scale adversely by the same amount. This highly significant limitation is the reason for the inherent limitations with the suffix array technique. The prefix array blocking technique has the same limitation, as it is the same as suffix array, apart from using prefixes instead of suffixes.

Another situation where the suffix array method will group suffixes into incorrect blocks can occur when the minimum suffix length parameter is too large. Consider the situation where there exist two candidate records that should be matched. Their blocking key values are identical, apart from a character replacement error 3 characters away from the end in one of the BKVs. These two BKVs may have the same suffix, but if the minimum suffix length parameter is 4 or larger, the matching suffixes of length 3 are excluded. This parameter is necessary to avoid a large number of comparisons being made between records that exhibit only a very small amount of similarity. Therefore, it is critical to carefully select the value for this parameter.

A. Improving suffix array using Longest Common Subsequence

The suffix array technique is suitable for a wide range of applications, but has one large limitation. Two BKV substrings may be long enough to surpass the minimum suffix length parameter, but contain even just one small difference due to misspellings or typographical errors, resulting in an inability to group these two BKVs into the same block. An example would be the BKVs ‘tophills’ and ‘topbills’. When the minimum suffix length parameter \( l_{ms} \) is 5, the suffix arrays will be ‘hills’, ‘phil’, ‘ophills’, and ‘tophills’ for the first BKV, and ‘bills’, ‘phil’, ‘ophill’, and ‘tophills’ for the second. Clearly, most of these suffixes have a high similarity. However, the standard suffix array blocking method will not group any of these suffixes into the same block, causing it to miss the comparison that should be carried out between these two BKVs.

We propose an approach towards solving this problem, by carrying out a grouping operation on similar suffixes. 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 \( mm \) BKV
suffixes when matching among \( n \) records where the average BKV length is \( m \) (Larger values of the minimum suffix length parameter \( l_{\text{ms}} \) will reduce this). A full comparison among all of these records will therefore result in a time complexity of \( O((mn)^2) \) 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 \( mn \) 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 given above, we want to avoid comparisons among suffixes that were generated from the same BKV. However, we would like to carry out comparisons between suffixes generated from the first and second BKVs. Each suffix in the suffix array generated from a BKV will be highly similar to the other suffixes, 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, it is usually the structure itself which carries out the ordering. This requirement therefore automatically disperses suffix arrays that were generated from the same BKV throughout the list. The ordering for the example given above is shown in Table III-A, where BKV number is 1 for the BKV ‘tophills’ and 2 for ‘topbills’.

It can be seen that the ordering required by the indexing structure has a tendency to both:

1) Disperse suffixes that were generated from the same BKV
2) Group together similar suffixes from different BKVs

This second tendency 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 nearby neighbors when carrying out the grouping. Specifically, we implemented a method that can iterate over the sorted set of BKV suffixes, comparing each current suffix with the following one. If a similarity exists, the following suffix can be merged into the same group as the current suffix. If no similarity exists, the method continues with the next suffix becoming the one currently being processed. An interesting observation that can be made is that 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.

There are many ways that grouping in this manner can be carried out on the list of sorted BKV suffixes. These methods can be condensed into two parts:

1) A similarity metric to compare BKV suffixes
2) A method for selecting which BKV suffixes to compare, and a rule for deciding whether to merge two suffixes or not

The implementation details of the BKV comparison selection and merging rule can affect the matching result slightly. However, the similarity metric will have a much more significant effect.

We carried out experiments using the Longest Common Subsequence (LCS) operator as the similarity metric. In almost all cases, the selection of the similarity metric will give different results depending on the underlying aspects of the data being linked. For example, an edit-distance based similarity metric such as the Jaro metric will be useful in datasets where errors are mostly typographical in nature.

Longest Common Subsequence (LCS) is a well-known general technique for finding similarities between two strings. In our case, we are interested more in the length of the LCS, rather than the LCS itself. This length, when compared to the overall length of both suffix strings being compared, allows us to determine a measure of similarity between those suffix strings. In addition, the base LCS algorithm may produce more than one longest sequence, all of maximum length, in a process with a computational complexity of \( O(2^n) \) in the worst case, when comparing two strings of length \( m \). However, if we are only interested in the length of the LCS result, we can utilise a faster algorithm to compute this value with \( O(m^2) \) worst-case complexity, by only selecting the first of all available longest common subsequence results. In practical applications, we typically have \( m << n \), so the difference may be negligible. However, choosing an appropriate similarity function is an important avenue to consider for the implementation of blocking techniques such as this one.

We require an extra parameter to determine the degree of similarity required to merge two adjacent suffixes. The main aim when merging suffixes is to group together suffixes that have a few mistakes near the end of the string. For simplicity, and to avoid adding another parameter, in our experiments we created a simple rule for this task. If \( s_1 \) and \( s_2 \) are the two input suffixes, \( l_1 \) and \( l_2 \) are the lengths of these suffixes, \( l_{\text{cs}} \) is the length of any result of the LCS operation, and \( l_{\text{ms}} \) is the minimum suffix length parameter, then we can define the grouping result as:

<table>
<thead>
<tr>
<th>Suffix</th>
<th>BKV number</th>
</tr>
</thead>
<tbody>
<tr>
<td>bills</td>
<td>2</td>
</tr>
<tr>
<td>hills</td>
<td>1</td>
</tr>
<tr>
<td>opbills</td>
<td>2</td>
</tr>
<tr>
<td>ophills</td>
<td>1</td>
</tr>
<tr>
<td>pbills</td>
<td>2</td>
</tr>
<tr>
<td>phills</td>
<td>1</td>
</tr>
<tr>
<td>tphills</td>
<td>2</td>
</tr>
<tr>
<td>topbills</td>
<td>1</td>
</tr>
</tbody>
</table>

TABLE I
The suffix array ordering for the example with two BKVs: 1) ‘tophills’ and 2) ‘topbills’. Minimum suffix length is 5.
Similar suffixes. In practice, the use of the $l_{ms}$ parameter for this extra task works well, due to it’s use in the suffix array technique. Standard suffix array will not group into the same block BKVs that have mistakes that occur $l_{ms}$ or fewer positions away from the last character, even if the other characters of the BKVs are identical to each other. In order to overcome this, we have two aims:

1) Group together suffixes which exhibit high similarity, and where differences occur $l_{ms}$ or fewer positions away from the end character of the BKVs

2) Avoid grouping together suffixes that exhibit high similarity, but where there are no differences that occur $l_{ms}$ positions away from the end character of the BKV

The first point aims to overcome the main weakness of standard suffix array, and improve the pairs completeness (PC). The second point aims to avoid a drop in reduction ratio (RR).

An approach targeted at these characteristics should work well. However, we found that the result is good even when completely disregarding the positioning of the differences between $s_1$ and $s_2$, a process which also avoids the extra computation and allows for an easier analysis. Specifically, if the exact positioning of the differences is required, the full set of all possible LCS results must be gathered and analysed to find the positioning of the differences, a process that increases the time required significantly and may cause the overall process to scale by $2^n$ in certain situations. In practice, though, a better result may be obtained with the use of this or some other difference positioning information for most datasets. Additionally, the general technique we selected will satisfy the first condition listed above, at the possible expense of the second. However, it turns out that even by discriminately grouping together suffixes with high similarity at any position in the strings, the RR does not decrease significantly. This is due to the required step towards the end of the process, that carries out duplicate removal.

### B. Duplicate Removal

A necessary step for a large number of blocking techniques is duplicate removal after the set of candidates have been selected. This is particularly necessary for techniques such as suffix array, which generate an array of suffixes from a single BKV. Each of these suffixes is then taken as a key, to be used to select a list of candidate records to compare against the original record. However, due to the nature of the generation of the suffix array, many of the suffixes are the same, varying from their neighbor only by one added character. An example can be seen by searching for possible records to link against the string ‘cats’. Our existing database contains the strings ‘bats’, ‘rats’ and ‘wombats’. The array of suffixes for this existing data is given in order in Table III-B.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Record Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>bats</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>mbats</td>
<td>3</td>
</tr>
<tr>
<td>wombats</td>
<td>3</td>
</tr>
<tr>
<td>rats</td>
<td>2</td>
</tr>
<tr>
<td>fs</td>
<td>1, 2, 3</td>
</tr>
</tbody>
</table>

In this example, the three existing strings share common features that cause them to be grouped into the same block. However, this happens many times for each of the slightly different suffixes. Therefore, when querying for possible candidate records to match against a ‘cats’, the generated suffix array is ‘ts’, ‘ats’ and ‘cats’. The corresponding set of record numbers found by searching the indexing structures with these keys is then ‘1,2,3’, ‘1,2,3’, and nothing for the third key. Clearly, there exist many duplicate record numbers even for this small example, and in practice, many duplicates are found when using a suffix-array based technique and indeed most of the current blocking techniques. A duplicate removal step at this point has the effect of avoiding extra duplicate comparisons being carried out. This is advantageous because the full comparison between records can be a time consuming step in the record linkage process, and carrying out duplicate full record comparisons is usually a waste of resources. The duplicate removal step itself can be carried out easily in constant time and linear (with $n$) space with the use of a basic hash map structure.

### IV. Analysis

We carry out a more thorough analysis into the time complexity of the proposed improved suffix array technique, as well as adding insight into why the technique is effective and where it may fail short.

We first analyse the standard suffix array approach. Consider two records of information, $r_p$ and $r_q$. Selected information in these records is combined to form the BKV strings $b_p$ and $b_q$, of length $m_p$ and $m_q$ respectively. The minimum suffix length parameter is denoted by $l_{ms}$, and maximum block size by $l_{mbs}$. The suffix arrays generated from $b_p$ and $b_q$ are then $a_p$ and $a_q$ respectively, where each array contains a set of suffix strings, therefore $a_p = \{s_{p1}, s_{p2}, \ldots, s_{pk}\}$, $b_p = m_p - l_{ms} + 1$ and equivalently for $a_q$. The suffixes $s_{p1}, \ldots, s_{pk}$ and $s_{q1}, \ldots, s_{qk}$ are added to the indexing structure as keys, with the value for the $s_p$’s being the record number of $r_p$, and the $s_q$’s having the record number of $r_q$.

We can build a model to estimate the probability of various types of errors, such as character replacement, insertion, deletion, or swapping. Assuming that $r_p$ and $r_q$ are generated from
the original ground truth data, we can simplify our model by assuming that \( b_p \) and \( b_q \) have the same length, \( m_s = m_p = m_q \). We can then define a simple model to estimate the probability for a difference between \( b_p \) and \( b_q \) to occur at position \( i \) to be \( c_i \), \( i \in 1, \ldots, m \). Standard suffix array will miss a correct blocking if 1) there is a mistake within the last \( l_{ms} \) characters of the duplicate BKV, or 2) due to the maximum block size condition. For the first situation, given our character replacement model, we can define the probability for standard suffix array to miss a correct blocking as:

\[
P_{\text{suffix-miss}}(b_p, b_q) = 1 - \prod_{i=1}^{m} (1 - c_i)
\]

Where \( k = m - l_{ms} + 1 \). The second situation occurs only in the relatively rare case where \( l_{ms} \) is low, and equal to the length of a common suffix that appears in the dataset (e.g. 'ing'). This common suffix can match other records exactly, and have no errors, but since the suffix is common, the block corresponding to this suffix may be larger in size than \( l_{ms} \). Due to the maximum block size condition, the block corresponding to the common suffix will be entirely removed. If there then exists a mistake at the position of the BKV one more than \( l_{ms} \) characters away from the end of the BKV, the suffixes that survive the maximum block size condition will not match each other. In this situation, a suffix that is common enough causes an effective increase to the value of \( l_{ms} \) for that record, equal to the length of the largest common suffix that is removed by the maximum block size parameter. If the block for a suffix of length \( g \) is removed, all suffixes with length less than \( g \) will also be removed. The probability for this situation to occur depends on the dataset used.

Improved suffix array is able to solve the first, and most common, of these problems. Given the model above, a number of different situations can occur when comparing \( b_p \) and \( b_q \) with standard and improved suffix array:

1) Differences occur within the last \( l_{ms} \) characters of one string
2) Differences occur in positions of the strings not within the last \( l_{ms} \) characters. These differences are close enough to the beginning of the string to cause complete grouping separation
3) Differences occur in positions of the strings not within the last \( l_{ms} \) characters. These differences do not cause the effect of situation 2.

We define complete grouping separation as the effect that can occur to bypass the optimisation step in our proposed grouping algorithm. Since we only consider for grouping suffixes that are next to each other in the ordered indexing structure, differences in the BKV near the beginning can cause the ordering to disperse similar suffixes into different areas of the indexing structure, where they are not considered for grouping. An example would be the BKVs 'handbag' and 'handtag', the latter of which is a mispelling of the first. If there exists other BKVs that are ordered alphabetically between these two BKVs, such as the string 'handlebar', 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 'andlebar' separating 'ndbag' and 'ndtag', and the suffix 'ndbar' separates 'ndbag' and 'ndtag', etc, as can be seen in Table IV. Of note, however, is that this situation only occurs when \( l_{ms} \) is 3 or larger, as otherwise the suffixes 'ag' and 'ar' may be close enough to be grouped.

However, even if complete grouping separation occurs, all is not lost. One solution to this problem is to ensure that overlapping groupings are all grouped together. Therefore, if three suffixes are next to each other in the ordered suffix list, and suffix 1 matches suffix 2, and suffix 2 matches suffix 3, then ensure that all three suffixes are grouped together. However, this solution does not work when the 'separating' suffix strings contain \( l_{ms} \) differences or more from their neighbors. An example of this is for the ordered suffixes 'andbag', 'andlebar' and 'andtag'. When \( l_{ms} \) is small enough, 'andbag' will not be similar enough to 'andlebar' to allow a grouping, and similarly for 'andlebar' and 'andtag', even though 'andbag' and 'andtag' are similar enough to group together and would have been grouped if they were adjacent in the ordered suffix list.

A second solution is to increase the range around the current suffix for selecting other suffixes for grouping. This modification allows for more leeway in the grouping of similar but alphabetically separated suffixes, at the expense of extra suffix comparisons being required. The time complexity of

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Record Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>andbag</td>
<td>1</td>
</tr>
<tr>
<td>andtag</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>dbag</td>
<td>1</td>
</tr>
<tr>
<td>dlebar</td>
<td>3</td>
</tr>
<tr>
<td>dtag</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>ndbag</td>
<td>1</td>
</tr>
<tr>
<td>ndlebar</td>
<td>3</td>
</tr>
<tr>
<td>ndtag</td>
<td>2</td>
</tr>
</tbody>
</table>

**TABLE III**

An occurrence of complete grouping separation. Two BKVs that should be blocked together are 1) 'handbag' and 2) 'handtag'. However, the suffixes of a third BKV 3) 'andlebar' will separate all suffixes of the original two BKVs, causing complete separation and causing the improved suffix array technique to 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.
the grouping operation is \(O(nm)\) when this range is set to a constant value, but a range of 2 will cause the grouping operation to double in cost compared to a range of 1, so for practical applications, lower values of this parameter are necessary to allow for a rapid grouping operation.

In our experiments, we tested a conservative approach to this problem, by only considering directly neighboring suffixes for grouping. Much like our approach towards selecting a similarity function for the suffix grouping purpose, we found that even when this simpler approach is used, the results are quite good. Again, adjustments to the optimisation procedures will affect the results from different datasets in different ways, and some will work better than others. Our recent example used names of objects, which consist of standard objects such as ‘hand’, ‘bag’, and ‘bar’ concatenated together to form new meanings. Therefore, these concatenated words exhibit a large amount of similarity with other words that have the same original word as a prefix. Record linkage is more commonly used to link together personally identifying information such as names, which does not often exhibit this behaviour. The selection of the conservative approach for our experiments allows for a more general result as well as an easier analysis and lower computational complexity.

We return to the analysis of the list of situations that can occur due to differences in various positions of the suffix strings being compared for grouping. These situations do not necessarily occur exclusively. For example, if situation 1 occurs on its own, standard suffix array will fail. If the number of differences is less than \(l_{ms}\), however, improved suffix array will correctly block these two records together, otherwise, it will also fail. If situation 2 or 3 occurs on its own, both suffix array and improved suffix array will correctly block these records together. If situation 1 occurs at the same time as situation 3, standard suffix array will still fail, but improved suffix array is able to retain the correct blocking as long as the number of differences is less than \(l_{ms}\). If situation 1 occurs at the same time as situation 2, both standard and improved suffix array will fail.

Improved suffix array therefore is able to overcome the weaknesses of standard suffix array when mistakes occur within the last \(l_{ms}\) characters of the two BKVs being compared, and when complete grouping separation does not occur. Again, this second condition is difficult to analyse as the probability for it’s occurrence depends greatly on the type of data used. However, this is usually quite rare. The probabilistic model given above gives an approximate guideline to the chance for standard suffix array to fail and improved suffix array to be able to correctly block two records together.

The effect of complete grouping separation can occur in one other situation. Even if some suffixes from \(b_p\) and \(b_q\) are adjacent, they may not be grouped because they have more than \(l_{ms}\) differences compared to each other. This may be due to an inherent low similarity between \(b_p\) and \(b_q\), in which case every known blocking technique will find it difficult to block these records together. As discussed earlier, while some suffixes for two BKVs may not be grouped together, it is rare that all suffixes for these BKVs will be either alphabetically separated or dissimilar enough for no grouping to occur, causing the complete grouping separation effect.

It is also useful to compare the effect of improved suffix array on the reduction ratio. During grouping, it is likely that suffixes grouped may be similar but not belong to the same original record. When this situation occurs, the reduction ratio of improved suffix array will suffer compared to standard suffix array. This can be expected as we add more ‘fuzziness’ to the matching. However, as can be seen from the results, this effect has a minor impact when our suffix grouping optimisation is in place, namely the rule to limit potential grouping to suffixes that are adjacent in the ordered suffix list. The probability for an incorrect blocking result to occur, adversely affecting the reduction ratio, is difficult to analyse as it depends on the type of data being used. Data that contains many records that are highly similar to one another, both in length and in their contents, especially near the beginning of the BKV, will cause the grouping operation to group together more suffixes that do not belong to the same original record, therefore causing more incorrect blocking results to occur and a lower reduction ratio overall result.

A. Complexity

Pairs Completeness and Reduction Ratio are not the only measurements of interest for comparing different blocking techniques. Another important measurement is the computational complexity underlying the algorithm used. The actual runtime may be affected by the exact implementation used, but the computational complexity should remain unchanged or similar assuming that a reasonable implementation was chosen. We analyse the computational complexity changes introduced due to the grouping technique in this section, along with a suitable implementation to store and use the grouping result.

The standard suffix array technique will generate \(kn\) suffixes on average, if \(m\) is the average BKV length, \(k = m - l_{ms} + 1\) and \(n\) is the number of records to match with one another. The indexing structure is then constructed, usually using a tree-based structure to allow for \(O(kn \log kn)\) construction and \(O(\log kn)\) 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 \(m\) suffix keys, each referencing \(n\) dataset items, causing query time for a single record to equal \(O(nm \log n)\). However, with a normal dataset, the indexing structure is usually able to separate records into distinct blocks and allow for an \(O(b \log nm)\) query time, where \(b\) is a value that depends on the dataset 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 in a hash-map type structure if a suitable hash function can be found, otherwise the original index structure may need to be modified. The addition of related groups will slow the
query for a particular record down by a factor of $d$, where $d$ is a parameter that again depends on the type of data used, being equal to the average size of a grouping result for any one record. With our suggested optimisation, $d$ will always be less than 1 or 2, depending on the exact implementation. Therefore, datasets containing highly similar records that are alphabetically close result in a larger value of $d$. If the original indexing structure is modified, linked-list type capabilities need to be added to allow for node sizes to be arbitrarily large, with any node being able to include all grouped node results. 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 $O(\log kn)$ time in the expected operating situation where records are able to be separated into blocks reasonably well. The result of each suffix query is the set of record identifiers to be traversed, checked to ensure no duplicates exist, and then used in the full matching. Grouping adds extra record identifiers to this step, but the computational complexity is not modified. Both standard and improved suffix array queries may result in a list of record identifiers that scale with $n$, in some extremely rare dataset types. The time complexity of a query in this case would be $O(n \log kn)$ for both standard and improved suffix array in this example. When our optimisation technique is used, the number of additional grouping results is limited to 1 or 2 per query. Therefore, the time taken may be slowed by a factor of $d$ at this stage due to grouping, but again, the time complexity with regards to $n$ or $m$ is unchanged.

V. EXPERIMENTS

Our first set of experiments is designed to compare the improved suffix array method against standard suffix array as well as the traditional blocking method, using the standard measurements of Pairs Completeness and Reduction Ratio. We run the experiments on a real dataset as well as a synthetic one. The real dataset is sourced from an insurance company where a large-scale record linkage module exists as part of a larger surveillance system. The synthetic dataset was generated using the Febrr[4] tool. In our first experiment we vary the Minimum Suffix Length ($l_{ms}$) parameter while keeping the value of the Maximum Block Size parameter fixed to the value of 20. In our second experiment, we vary the Maximum Block Size parameter while keeping the value of Minimum Suffix Length fixed at 4.

Our second set of experiments are designed to show that our results are consistent across multiple different compositions of BKVs. They also illustrate the importance of the ordering chosen when constructing the BKV by concatenating the chosen key fields from the original data. The selection of the key parameters used in construction of the BKV is varied, both in order and in composition, resulting in a small but distinct set of BKVs. We call these the BKV Feature Selection experiments. Again, we measure Pairs Completeness and Reduction ratio on both the real and synthetic datasets.

Finally, the performance of a blocking technique depends not only on the reduction ratio and pairs completeness it can achieve, but also on the intrinsic computational complexity of the algorithm itself. We carried out one experiment to prove that the addition of the grouping rule does not negatively impact the running time performance of the suffix array technique by any significant amount.

VI. RESULTS

The results for our first set of experiments, comparing traditional blocking, standard suffix array, and improved suffix array while varying the parameters Minimum Suffix Length ($l_{ms}$) and Maximum Block Size, can be seen in Figures 1 and 2. It is clear that for a good selection of $l_{ms}$, we can obtain a very high pairs completeness, while achieving a reduction ratio very similar to the highly efficient standard suffix array method. While the actual numerical differences may be small for the Reduction Ratio measurements, every decimal place counts when blocking techniques are used for very large datasets. For a 10,000,000-row dataset, the difference between a reduction ratio of 0.98 and 0.999 causes approximately a 20-fold difference in time saved. It is clear from the reduction ratio results that a large time saving can be achieved by utilising the improved suffix array technique over the traditional blocking method, while experiencing only a slight loss in accuracy or pairs completeness. It is also clear that when compared to the standard suffix array method, our improved approach boasts a much higher pairs completeness, at the cost of a small amount of reduction ratio. This large effect of a comparatively small modification shows the extent of the main weakness of the standard suffix array method.

The results for our second set of experiments, where we change the feature set of fields used to construct the BKV, are shown in Figures 3 and 4. These results show that our results are consistent for many different types of feature selection when constructing the BKVs. However, of interest 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. The difference in pairs completeness between the real and synthetic datasets highlight the effects of this chosen ordering.
with improved suffix array showing the usual increase over standard suffix array for the synthetic dataset, similar to other BKV constructions where the last string to be concatenated into the BKV contains many errors and differences.

The result for our time test experiment, intended to gauge the computational complexity of the three approaches, is shown in Figure 5. It can be clearly seen that the curve corresponding to the improved suffix array method closely follows the standard suffix array approach, showing that the extra computation time required for the grouping step is minimal compared to the main processing required by the base algorithm.

One distinctly useful benefit of the suffix array technique is the ability to reduce excessive processing times for common field values. A classic example occurs when matching against the record ‘John 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 databases are used, and the intrinsic problem that is encountered when trying to distinctly block highly similar records such as this is easy to see. 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, such as in the suffix array method, causes the candidate set for ‘John Smith’ to be greatly reduced, as the number of records highly similar to ‘John Smith’ is much less than the number of ‘John’s plus the number of ‘Smith’s. Improved suffix array 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 for traditional blocking methods.

VII. CONCLUSION

Suffix array works well when \( n_i \) is small and \( n_i \) is large, both in accuracy (pairs completeness) and scalability. The reduction ratio experienced by this method will always be very high. Our improvement inherits these qualities, but improves the accuracy at the cost of a small amount of extra processing. The qualities of the improved suffix array technique make it well-suited for large-scale applications of record linkage. Our experimental results show that we can achieve a significantly higher reduction ratio than the traditional method, a technique that is often used in industrial applications of record linkage. For very large data sets, where blocking becomes more important, the use of the improved suffix array method results in a greatly reduced number of comparisons to carry out for every record that must be matched, a property that is critical for many high-performance and real-time systems. We also highlight the benefit that suffix array based techniques have over the traditional method for large datasets where some field values are much more common than others (the John Smith example), by avoiding an excessively long computation requirement for matching records of this type.

We have also shown that the accuracy or pairs completeness of the improved suffix array technique is much higher than standard suffix array for all of the surveyed datasets. 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 our additional grouping process that is carried out on the sorted list of suffixes in the indexing structure, even when we limit our implementation to a basic and 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, we described possible improvements to our approach to benefit from these additional sources of information.

REFERENCES

Fig. 1. Pairs Completeness measurements for the first set of experiments.

(a) Pairs Completeness for the Minimum Suffix Length ($l_{ms}$) experiments, using the real data set.

(b) Pairs Completeness for the Minimum Suffix Length ($l_{ms}$) experiments, using the synthetic data set.

(c) Pairs Completeness for the Maximum Block Size experiments, using the real data set.

(d) Pairs Completeness for the Maximum Block Size experiments, using the synthetic data set.

Fig. 2. Reduction Ratio measurements for the first set of experiments.

(a) Reduction Ratio for the Minimum Suffix Length ($l_{ms}$) experiments, using the real data set.

(b) Reduction Ratio for the Minimum Suffix Length ($l_{ms}$) experiments, using the synthetic data set.

(c) Reduction Ratio for the Maximum Block Size experiments, using the real data set.

(d) Reduction Ratio for the Maximum Block Size experiments, using the synthetic data set.
Fig. 3. Pairs Completeness measurements for the second set of experiments.

(a) Pairs Completeness for the BKV Feature Selection experiments, using the real data set.

(b) Pairs Completeness for the BKV Feature Selection experiments, using the synthetic data set.

Fig. 4. Reduction Ratio measurements for the second set of experiments.

(a) Reduction Ratio for the BKV Feature Selection experiments, using the real data set.

(b) Reduction Ratio for the BKV Feature Selection experiments, using the synthetic data set.
Fig. 5. The timing experiment showing query times for the three blocking techniques over datasets of differing sizes. The curve for the improved suffix array method closely follows the standard suffix array result.