Robust Record Linkage Blocking Using Suffix Arrays and Bloom Filters

TIMOTHY DE VRIES, HUI KE, and SANJAY CHAWLA, University of Sydney
PETER CHRISTEN, The Australian National University

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, quadratic scalability for the brute force approach of comparing all possible pairs of records necessitates the design of appropriate indexing or blocking techniques. The aim of these techniques is to cheaply remove candidate record pairs that are unlikely to match. 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 highlight the importance of using efficient indexing and blocking in real-world applications where datasets contain millions of records. We extend our disk-based methods with the capability to utilise main memory based storage to construct Bloom filters, which we have found to cause significant speedup by reducing the number of costly database queries by up to 70% in real data. We give practical implementation details and show how Bloom filters can be easily applied to Suffix Array based indexing.

Categories and Subject Descriptors: H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Record linkage, blocking, suffix arrays

ACM Reference Format:
de Vries, T., Ke, H., Chawla, S., and Christen, P. 2011. Robust record linkage blocking using suffix arrays and Bloom filters. ACM Trans. Knowl. Discov. Data 5, 2, Article 9 (February 2011), 27 pages.
DOI = 10.1145/1921632.1921635 http://doi.acm.org/10.1145/1921632.1921635

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 where there is an absence of a uniquely identifying key across these datasets. Matching records in two separate data sources can be carried out when there exists enough similarity in the structure and domain of the data in the two sources. In this case, comparisons must be made between records in the first source and records in the

T. de Vries and S. Chawla gratefully acknowledge the financial support of the Capital Markets CRC.
Authors' addresses: T. de Vries, H. Ke, and S. Chawla, School of Information Technologies, University of Sydney, Sydney NSW 2006, Australia; email: timothy.devries@gmail.com, hui.ke.it@gmail.com, chawla@it.usyd.edu.au; P. Christen, College of Engineering and Computer Science, Australian National University (ANU), Canberra ACT 0200, Australia; email: peter.christen@anu.edu.au.

DOI 10.1145/1921632.1921635 http://doi.acm.org/10.1145/1921632.1921635

second source, using a custom similarity function and a classifier. Alternatively, the same linking approach can be used to find matches among records in the same dataset for the purposes of duplicate removal [Winkler 2006]. In both of these cases, the main problem that must be overcome is the presence of errors and small differences between records in the data that are to be matched. Record linkage is an approximate matching technique that aims 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, or even from legacy systems. The additional information acquired using record linkage can be used in many ways in the data mining process, such as creating richer data models with additional features. Public data sources may also be used in the linkage process. Record linkage has been successfully used in diverse areas such as medical health systems [Gill et al. 1993], business analytics, fraud detection [Newcombe and Kennedy 1962], demographic tracking, government administration, and national security.

However, the matching process is a very computationally intensive task. For matches or duplicates to be found across two datasets of size n, potentially up to n^2 comparisons may be required, making a brute force nesting approach infeasible 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 likely lead to matches or non-matches. This indexing process is referred to as *blocking* in the context of record linkage, a naming convention from the early indexing practice of separating records into distinct blocks. Blocking techniques can be tailored to favor either the accuracy or efficiency of the linkage task, in a tradeoff manner. Blocking techniques that discard a large number of potential comparisons and generate a small candidate set will increase record linkage efficiency, but are more likely to miss potential comparisons, causing a loss in accuracy, and vice versa. The main aim is to make record linkage feasible on large datasets by greatly reducing the number of record pair comparisons that need to be carried out, usually at the cost of a very small loss in accuracy. The design of a robust blocking technique based on suffix arrays is the focus of this article.

Practical record linkage systems therefore typically consist of three main steps. In the first step, an indexing (blocking) technique is applied on the databases to be matched with the aim to find candidate record pairs that are likely correspond to matches, while discarding a large number of pairwise comparisons that are very unlikely to correspond to a match. This step is the topic of this article and will be discussed in more detail in Section 2. In the second step, the records of each candidate pair are then compared with each other in more detail, using specifically tailored domain-specific similarity functions. In the third step, a classification model is then used to determine whether the two records in the record pair are a match or a nonmatch.

A large variety of blocking methods have been developed, 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 commonly known as traditional blocking [Baxter et al. 2003]. It is simple to implement and thus is the method of choice in many industrial settings where the number of records to match is not prohibitively large. It can achieve high accuracy at the expense of a having a very low matching efficiency, because a large number of candidate record pairs are usually generated. The authors of this article are involved with the production of a large-scale

Table I. 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

Suffix	Record Number
hnSmith	1
hnSnith	2
JohnSmith	1
JohnSnith	2
mith	1
nith	2
nSmith	1
nSnith	2
ohnSmith	1
ohnSnith	2
Smith	1
Snith	2

data mining system that utilizes record linkage in the industry. Traditional blocking was initially chosen for this application, but an increase in the number of records of data to be matched called for a more efficient implementation. We were able to use the existing set of labeled model training data for training the classifier at the record linkage level to test the performance of different blocking techniques in a direct experimental comparison. The accuracy of each blocking technique is measured by counting the number of pairwise record comparisons from the labeled data that exists in the candidate set produced by the blocking technique. Performance is measured by the size of the candidate set and actual running time.

Besides improving efficiency, a second important reason for choosing a nontraditional blocking technique is 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. Traditional blocking can compensate for this by using many separate indexed fields, but this causes the set of candidate record pairs to greatly increase in size, adversely affecting the efficiency, or time required to carry out the matching. The most important goal of indexing or blocking is to find these similar but nonidentical matches without adversely reducing performance by 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 [Aizawa and Oyama 2005]. It is the basis for our improved blocking method that retains the high efficiency of Suffix Array blocking while also increasing accuracy to an acceptable level by avoiding the miss-classification 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 (ignoring address). 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 I.

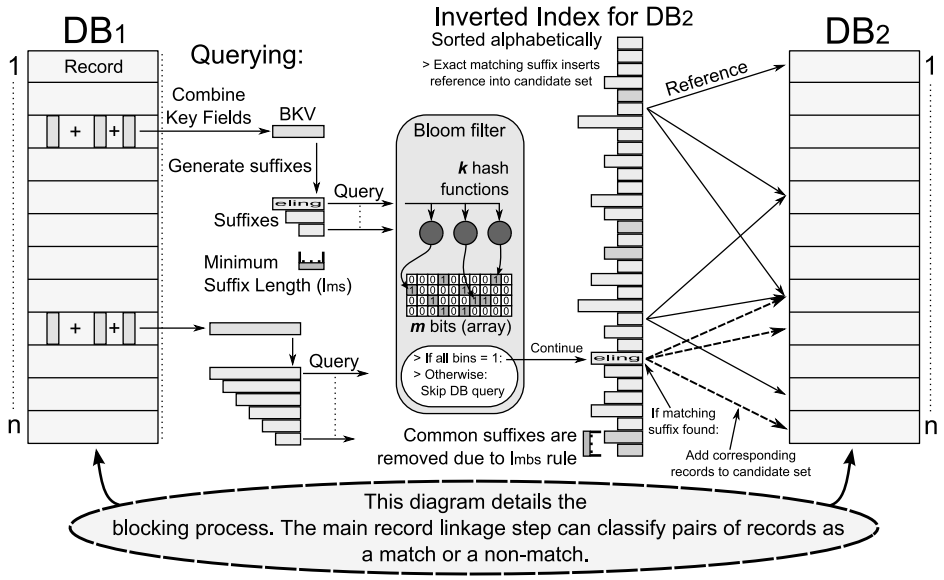

Fig. 1. The details for the overall blocking approach used by Suffix Array and Improved Suffix Array blocking, with the addition of a Bloom filter.

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 improvement over standard Suffix Array blocking. We cover this improvement in more detail in Sections 3 and 4.

1.1 Contributions

We summarise the results of the improved Suffix Array blocking technique as found in our previous paper [de Vries et al. 2009]. We extend these results by focusing on the large-scale problems encountered as datasets increase 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 to. We take advantage of an implementation optimisation to provide results showing the scalability of Improved Suffix Array when matching datasets with millions of records, a problem requiring a number of comparisons potentially in the order of 10^{12}.

We also demonstrate that the use of a Bloom filter in main memory as a filtering step can lead to significant savings in disk read operations for real world data, reducing the overall time taken (See Figure 1). We found that we could skip from 40% to 70% of all disk queries in our real world identity dataset of up to 3 million records. We found that Bloom filters always improve the overall time for the matching process even

when there is no network travel time required, and disk access speed is fast. We show that the use of Bloom filters provides much greater benefits when costly operations such as these are necessary, as would be the case for large scale distributed record linkage or the use of slow legacy system hardware. We describe a practical and robust parameter-free Bloom filter implementation that avoids the issues of selecting parameters and defining multiple hash functions. To the best of our knowledge, the use of the Bloom filter for distributed record linkage has not been covered before in the literature.

2. BACKGROUND

The problem of record linkage began receiving attention with the early work of Dunn [1946] and Marshall [1947]. A common classification model used was the one based on statistical methods by Fellegi and Sunter [1969]. In the past decade the problem of matching records has been investigated by researchers in the fields of machine learning, data mining, and the database community. For a recent review see Elmagarmid et al. [2007].

The issue of scalability to match large databases was quickly recognized as a key component for efficiency purposes, and a variety of different indexing techniques have been developed under the name of *blocking* in the context of record linkage. Recently developed advanced blocking techniques typically contain extra functionality over standard blocking in order to solve specific record linkage issues. Blocking is especially important due to the n^2 scalability of naïve unoptimized record linkage. Blocking solutions strive to reduce the number of candidate record pairs for comparison as much as possible, while still retaining an accurate result by ensuring that candidate pairs that likely correspond to a true match are not being removed in the blocking process.

A variety of blocking methods are currently used in record linkage procedures, with the most well-known ones including traditional blocking, sorted neighborhood [Hernandez and Stolfo 1998], q-gram based blocking [Baxter et al. 2003], canopy clustering [Mccallum et al. 2000], string map based blocking [Jin et al. 2003], and suffix array blocking [Aizawa and Oyama 2005]. Additional detail and analysis is given in de Vries et al. [2009].

Common to all blocking techniques is that they require the definition of a set of *key fields* to be selected from the data, as a subset of the total set of fields. These key fields are used to determine into which block (or blocks) each record is to be placed. 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, or alternatively the values of key fields are encoded using phonetic encoding techniques such as Soundex or NYSIIS [Christen 2006] in order to improve the blocking of values that contain errors and variations. The resulting string is called the Blocking Key Value (BKV) [Gu et al. 2003]. 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 (encoded) attribute or combination of attributes that are as identifying as possible, uniformly distributed, and having high predictive power. A suitable example attribute in many countries could be zip- or postcode, because these commonly have a roughly uniform distribution and are of high quality as they are recorded automatically.

2.1 Traditional Blocking

Traditional blocking is well known and often used in practical applications. This approach works by only comparing records with each other that have an exactly

matching key field, for example only comparing records that have the same postcode [Jaro 1989]. 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 BKV in order to increase accuracy, or several BKV are used in several iterations of a record linkage process to overcome errors that might occur in one key field. In the industrial record linkage application used for comparison, the selected key fields are given name, surname, and date of birth. For example, when carrying our matches with the record "John Smith, 01/01/1960, 10 Plum Road," for a blocking key based on the key field "given name" traditional blocking will select all candidate records that exactly match with "John" on given name, while for a key field "surname" all candidate records that exactly match with "Smith" on surname will be selected. A common modification to add a degree of "fuzziness" to the matching is done by applying phonetic encoding (such as Soundex) to some of the key fields [Christen 2006].

One major weakness of traditional blocking is that errors in any of the blocking key values will result in records being inserted into the wrong block. A second drawback is that the size of each block is almost always very large, causing many unnecessary comparisons to be carried out and a very low overall efficiency. Finally, the sizes of the blocks generated depend upon the frequency distributions of each individual field used in the blocking key value [Christen 2007]. 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 typically $O(n^2/b)$, where n is the number of records in each of the two datasets that are being matched and b is the number of blocks generated by the blocking process [Elfeky et al. 2002].

2.2 Suffix Array Blocking

The Suffix Array technique proposed by Aizawa and Oyama [2005] is a fast and efficient blocking method for large scale record linkage. We utilise this technique in combination with a Key Blocking approach with an additional adjustment to handle string characters as the individual tokens. Analysis of this technique against several other recent alternatives [Christen 2007] found that the efficiency gain is 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 [Christen 2007]. For example, when the *minimum suffix length* parameter (l_{ms}) is 4, a BKV of "JohnSmith" will generate a suffix array containing the values "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 I. 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 and their corresponding suffixes, and inserting them into the indexing structure, one further optimisation step will be carried out by introducing the additional parameter *maximum block size* (l_{mbs}). This parameter will lead to large blocks (containing more than l_{mbs} references to records) to be deleted. 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, causing a significant drop in efficiency for standard Suffix Array blocking. Therefore, any particular block is entirely

removed 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 subblocks not adversely affect the recall of the result. The steps undertaken by standard Suffix Array can be seen in more detail in Algorithm 1, when excluding the suffix grouping step.

Former studies [Aizawa and Oyama 2005] 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 [Elfeky et al. 2002]. 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 in this location, typically only some of the longer suffixes of the BKV will retain the error, with many smaller suffixes remaining error free, causing the corresponding record to be inserted into the correct block. The redundancy created by using an array of suffixes in this manner is one of the strengths of Suffix Array blocking.

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 [Christen 2007]. One example of this occurs when matching of a record that contains high frequency values such as "John" or "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 datasets. Improved Suffix Array blocking inherits these benefits. In practical terms, this functionality is important in (near) real-time 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) [Christen 2007]. 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 nonmatches 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 Christen [2007], 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 I when minimum suffix length is 4. Careful selection of this parameter's value is therefore essential.

3.1 Improving Suffix Array Using Grouping

We tackle the main weakness of standard Suffix Array blocking, where a single error in in a position less than l_{ms} in a BKV when comparing two BKVs will result in an incorrect blocking allocation and potentially a missed true match. See Table I for an example of an ordered suffix index list where standard Suffix Array Blocking will fail to assign two records into the correct block. However, it is easy to see that many of these adjacent suffixes are highly similar. We propose a solution that works 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 I 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 behavior

can be seen in Table I, 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* [Hernandez and Stolfo 1998] 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 [Jaro 1989] 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. The Jaro distance of two strings s_1 and s_2 is given by:

$$Jaro(s_1, s_2) = \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right),$$

where m is the number of matching characters between s_1 and s_2. Two characters from s_1 and s_2 respectively are considered *matching* if their position relative to each other is no further than $\lfloor \frac{max(|s_1|,|s_2|)}{2} \rfloor - 1$. t refers to the number of transpositions that is required to arrange the matching characters from s_1 and s_2 in the correct order.

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

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 mimimumsuffix 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)

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 dataset items, causing query time for a single record to equal $O(kn \log kn)$. However, with a normal dataset, 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 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 and the query operation. For index construction, the list of

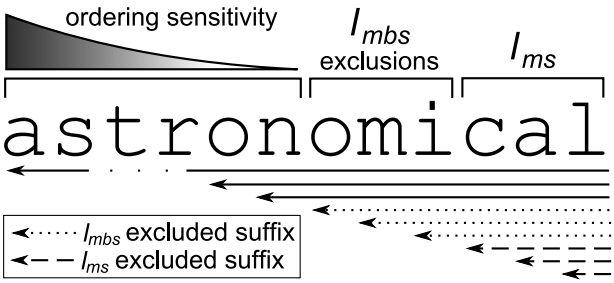

Fig. 2. An example showing suffix exclusion due to l_{ms} and l_{mbs}. Ordering sensitivity is also shown as an approximation of the likelihood for the order of the word in the list to change if one letter is changed at any given position.

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.

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 behavior is shown with an example in Figure 2.

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 2 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 dataset, where $l_{se} \geq l_{ms}$ and l_{se} can be visualized as the combined length of the l_{mbs} exclusions and l_{ms} regions in Figure 2. We then have the probability for standard Suffix Array blocking to miss a potential match between two records as:

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

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 quantify, 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 as well as 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 I and II. 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 II. 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 behavior 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 II that even in this specifically constructed example, the similarity is quite high, and grouping could still occur if the similarity threshold for grouping is sufficiently low.

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

Table II. 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 Optimization 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

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	-
...		

$w = 1$, 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. [2007].

The second requirement for successful grouping is simply that the two BKVs being compared must exhibit enough similarity to pass the similarity check that the grouping operation carries out (e.g. Jaro similarity). 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. 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 low similarity score.

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. Intuitively, if the candidate records added by the grouping step for Improved Suffix Array blocking all correspond to true matches, then the

proportion of true matches in the candidate set increases compared to false matches, increasing pairs quality. The reverse is also true if all candidate records added by the grouping step correspond to nonmatches, which would cause a loss in pairs quality. The effect of the grouping step on pairs quality can vary depending on the structure of the data used in the matching process, as is evident in some of our results using synthetic data, where pairs completeness and pairs quality are both higher when using Improved Suffix Array over standard Suffix Array blocking in some cases (See Figure 7). Generally, data that contains a higher quantity of errors will see an improved benefit in pairs quality from the grouping step of Improved Suffix Array blocking. The synthetic dataset used in our experiments was built to contain a significantly higher number of errors compared to the real dataset.

5. UTILISING MEMORY-BASED BLOOM FILTERS

The indexing structure described thus far is designed for a disk-based implementation, where suffixes and record identifiers are stored on disk in an inverted index data structure [Zobel and Moffat 2006]. However, if fast local random access memory is available, it can be utilised for a filtering step between the main code module and the database. The purpose of this additional step is to avoid having to carry out database queries that we know will not return any results. Bloom filters are a natural choice for this function [Bloom 1970], due to their extremely efficient space-saving properties. Additionally, the guarantee for no false negatives causes the accuracy of our approach to be identical to the normal approach when using Bloom filters as a filtering step for querying the database with suffixes. With the filtering step in place, a query to the database to find the records containing a suffix s will cause the Bloom filter to be queried with s instead. If the Bloom filter indicates that it does not contain an entry for s, we know we do not have to carry out the database query. However, if the Bloom filter indicates that it may contain an entry corresponding to s with some nonzero probability, we undertake the database query as normal. The use of a Bloom filter produces some false positives, but these simply cause a database query to be carried out as would normally occur when using no Bloom filter.

5.1 Bloom Filters

A Bloom filter is a space-efficient hash-based structure that may be used to test set membership where space is a concern [Bloom 1970]. Bloom filters have been used to great effect in many different areas including distributed databases, networking [Broder and Mitzenmacher 2004] and privacy preservation [Bawa et al. 2003; Schnell et al. 2009]. They have been modified to support deletion (counting Bloom filters) as well as other extra functions under the name of Bloomier filters [Chazelle et al. 2004]. Limited space can be optimized even further with the use of compressed Bloom filters [Mitzenmacher 2002].

A Bloom filter consists of an array of m bits, and a set of hash functions to map entries to locations on the array. Each position on the array starts out initialized to 0. Entries to the Bloom filter pass through one or more hash functions, whose output are spread among the m bits. The resultant bit location of a hashed entry to the Bloom filter is then set to 1. As additional elements are hashed into the bit array, the probability for false positives to arise increases. A false positive exists where querying for a given element may return 1, even though that element does not exist in the Bloom filter, as other elements have been hashed into the same k bins used by the query element. Besides m and n, the probability for a false positive to occur also depends on the number of hash functions used (k). Each hash function used causes a different position in the bit array to be set to 1, allowing for some redundancy against

the effect of false positives from other hashed elements. For a given query element, the bit array locations for all k hash function results of the hashed element value must contain a 1 for the query element to exist in the Bloom filter.

We assume that the output for each hash function used is evenly spread over the range $\{1 \ldots m\}$. Bloom filters generally perform adequately if practical hash function implementations are used, even if they do not satisfy this condition completely. After adding n distinct elements to the Bloom filter with bit array size m and using k hash functions, the probability that a specific bit in the Bloom filter is set to 1 is then defined as r [Bloom 1970], where:

$$r = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}. \tag{3}$$

After applying the effect of multiple hash functions used, the *false positive rate* p of the Bloom filter [Bloom 1970] is given as:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx (1 - e^{-kn/m})^k = (1 - r)^k. \tag{4}$$

5.2 Choosing Hash Functions

Bloom filters suffer from the requirement to specify not only the parameters for the size of the bit array (m) and the number of hash functions to use (k), but also the requirement that each of the k hash functions be independent. One solution for the problem of selecting hash functions is to generate multiple hash functions from two independent original hash functions [Kirsch and Mitzenmacher 2008]. While there will generally be some similarity among hash functions derived in this manner, this approach can work well in practice.

An alternative practical implementation is to use a random number generator along with a single hash function to produce k bit array locations.[1] The Bloom filter entry is hashed, producing a hash string. This hash string is then used as the *seed* for a random number generator. We are then able to read off k consecutive numbers from the random number generator and map these numbers to the bit array. We can repeatedly acquire the same set of bit array locations when initializing a random number generator from the same seed each time, thereby causing the accuracy of this implementation of a Bloom filter not to be affected by the random nature of the random number generator. If there is confidence in the random number generator and the single hash function used, all of the k bit locations chosen can be assumed to be evenly spread over the range $\{1 \ldots m\}$.

These two Bloom filter implementations are illustrated in Figure 3. We found the second approach to be easy to implement and very efficient to use in practice, as computing the result of a hash function can take a nontrivial amount of time for a large number of elements. This is true especially when good hash functions need to be selected to ensure an even spread over the m bits, as the results of these hash functions tend to take longer to compute than for faster but less balanced hash functions. For these reasons, we used the single hash function and random number generator based implementation of the Bloom filter in all of our experiments.

[1]Original implementation:
http://blog.locut.us/2008/01/12/a-decent-stand-alone-java-bloom-filter-implementation.

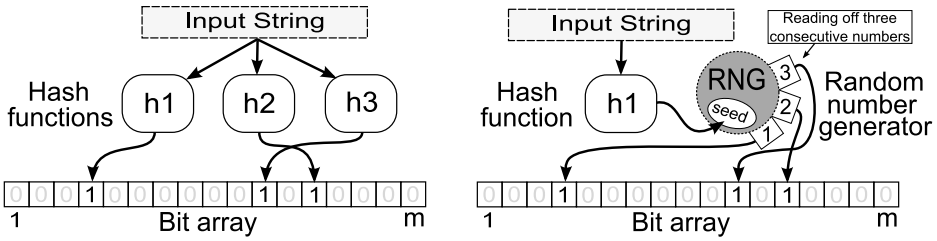

Fig. 3. Two possible Bloom filter implementations, one using k independent hash functions and one using one hash function and a random number generator (k = 3 in this example).

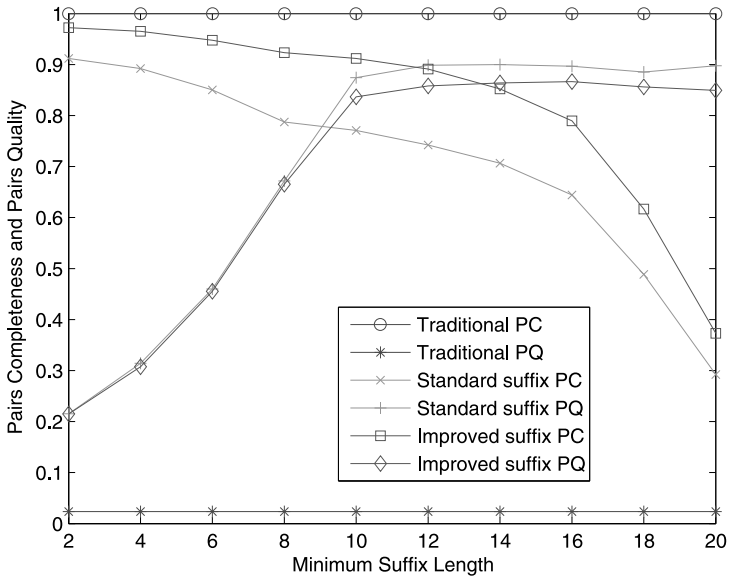

Fig. 4. Pairs completeness and pairs quality obtained while varying minimum suffix length (l_{ms}) on the real identity dataset.

5.3 Bloom Filters for Suffix Array Blocking

We can define the expected number of BKVs b to be inserted into the Bloom filter as $b = nc$ where c is the average number of suffixes generated for a record in the dataset and n is the number of records. A good estimate for the size of the bit array used in the Bloom filter given b and p from Equation (4) is then:

$$m = \frac{-b}{\log(1-p)}. \tag{5}$$

This size m may be adjusted due to space constraints. If more space is available, m can be chosen to utilize this space and reduce the number of false positives expected from the Bloom filter. Similarly, if space is limited, m can be reduced causing a graceful degradation of the effectiveness of the Bloom filter. When a value of m is chosen, a reasonable estimate for the number of hash functions k to use in the Bloom filter can be chosen by the rule:

$$k = \frac{m}{b} \ln 2. \tag{6}$$

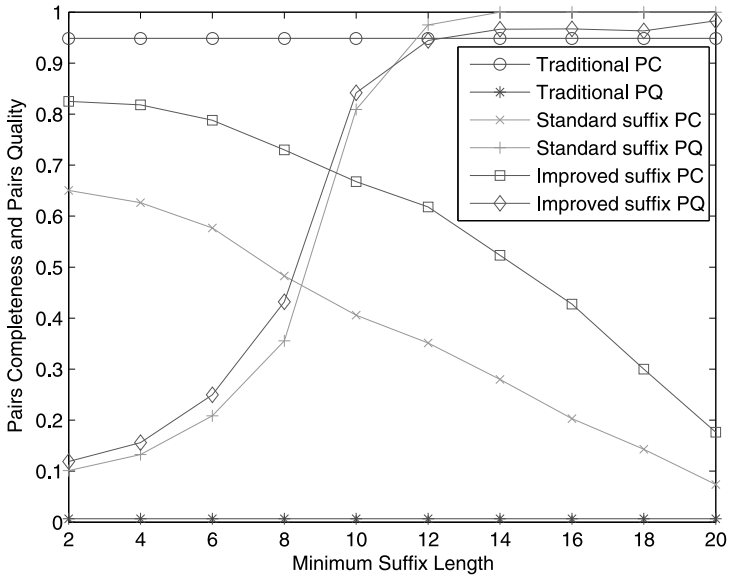

Fig. 5. Pairs completeness and pairs quality obtained while varying minimum suffix length (l_{ms}) on the synthetic dataset.

With k hash functions chosen in this way, the expected space usage of the Bloom filter bit array is approximately $\ln 2$ or 70% of the total m bits if $m \geq b$. We selected the values of m and k using these rules in our experiments. Additionally, we used only one base hash function and derived k bit locations using a random number generator with the hash string result as the seed, as described above. This overall approach removes the necessity to manually select the parameter values of m, k, and to define a large number of independent hash functions to allow for all expected values of k. This approach is therefore very easy to implement and use in a practical setting.

The expected false positive rate p of the Bloom filter when constructed with a selection of the parameters m and k for the purposes of Suffix Array blocking given Equation (4) with b expected Bloom filter elements is then:

$$p = (1 - e^{-kb/m})^k. \tag{7}$$

6. 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 datasets as well as a synthetic one. The real datasets are sourced from an insurance company where a large-scale record linkage module exists as part of a larger surveillance system. The identity dataset consists of personally identifying information such as names and addresses. The synthetic dataset was generated using the Febrl tool with standard settings [Christen 2008].

Even though the source for the real data contains millions of records, examples need to be hand labeled to produce accurate test datasets. Therefore, we were only able to obtain $n = 4135$ records with known linkage status for the real identity dataset. For our results to be comparable, we used $n = 5000$ records for the synthetic dataset. We use larger datasets in our performance comparison experiments.

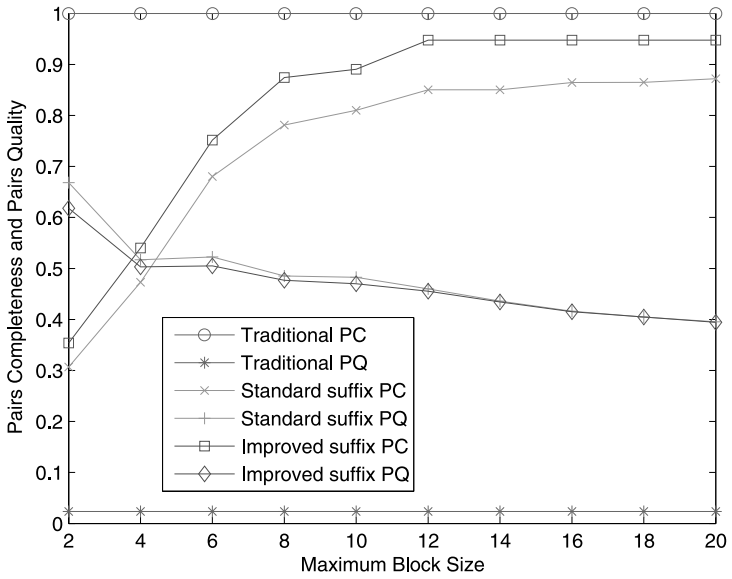

Fig. 6. Pairs completeness and pairs quality obtained while varying maximum block size (l_{mbs}) on the real identity dataset.

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 utilize 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. This value was chosen by tuning in 0.025 increments from 0.75 to 0.95 for all dataset sizes, all chosen values for l_{ms} and l_mbs and both suffix array based blocking methods.

The optimal values for the parameters of l_{ms}, l_{mbs} and the Jaro similarity threshold will vary depending on the properties of the data. l_{ms} is perhaps the most critical and can be selected by tuning over different values ranging up to a maximum of 50% of the average length of the BKV. Smaller values for l_{ms} is the safer option as high accuracy will be retained at the expense of only a very small amount of efficiency. Furthermore, the second parameter of l_{mbs} acts to remove the negative effect of common short suffix strings that may appear with the use of low l_{ms}. The parameter l_{mbs} can again be selected by tuning, as accuracy often quickly reaches a plateau with increasing values of this parameter, as can be seen in Figure 6 for a value of 12 and Figure 7 for a value of 4. The optimal value of the Jaro similarity threshold is highly data dependent and should be selected with tuning in the range of approximately 0.75 to 0.95. For cases where the data being matched is not predominately English text, a completely different comparison function with corresponding parameters may produce a better result.

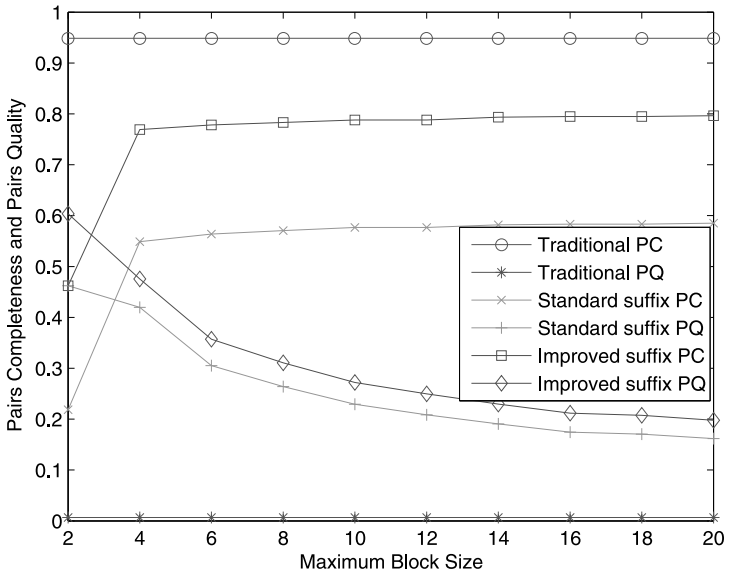

Fig. 7. Pairs completeness and pairs quality obtained while varying maximum block size (l_{mbs}) on the synthetic dataset.

6.1 Applying Bloom Filters

We carry out a second set of experiments to test the effect of using a Bloom filter as a filtering step when querying the database inverted index structure with suffixes. We select the size of the bit array m and the number of hash functions k according to the optimal rules described above. Using the database architecture described previously, we carry out experiments to measure the three main effects of using a Bloom filter for suffix array based blocking:

— The space required (in main memory)
— The time taken to initially construct the Bloom filter
— The percentage of database queries that were able to be skipped due to the use of the Bloom filter

The time required to query the Bloom filter in local memory compared to the time required for the full database lookup may vary significantly in different scenarios, causing a direct time measurement to have no significant meaning in our experiments. The measurement of the percentage of database queries that were able to be skipped was therefore used instead, as is common in the literature for Bloom filters. This measure is indicative of the total time difference that should be expected when using a Bloom filter over the standard approach, once implementation-specific characteristics are taken into account. If the cost of record access is known for a specific implementation, the cost savings from the use of a Bloom filter can be easily computed given the number of records to query with, the percentage of database queries skipped by the use of a Bloom filter, the time taken to construct the Bloom filter, and the time taken to query the Bloom filter. We found that in every experiment with the real identity dataset, the time taken to use the Bloom filter for every query and to undertake the following necessary database queries was always shorter than the time required to carry out every database query while using no Bloom filter. Our experiments were run on the database server (specifications above), and thus there was no network travel

time between the Java code and the database, and disk access times were fast due to modern disk-array storage. This ideal scenario reduces the effectiveness of the Bloom filter, which was nevertheless found to be faster than the use of no Bloom filter in every experiment.

However, the largest benefit from using a Bloom filter in this manner occurs when disk access is expensive. One example of this is for legacy systems, where data is often stored on slow disk hardware without disk arrays. Another application well suited for Bloom filters is distributed or privacy-preserving record linkage, where full-scale data copying or sharing is not feasible. A third example exists when matching records on extremely large datasets which are distributed over many different locations due to their size, causing significant network travel time. All of these examples plus any other situation where disk access is expensive will receive a large benefit from the use of a Bloom filter.

In all experiments we vary the size n of the datasets used. We select n records from one dataset for use as query records. Candidate records are then selected from the second dataset also of size n for every query record. This is the setup needed to match two databases together, both containing n records, with a potential n^2 comparisons between them before blocking is used to remove unnecessary comparisons. A similar situation occurs when carrying out duplicate removal, simply by comparing one dataset against itself. We also vary the Bloom filter false positive rate p. When m and k are chosen using the rules defined above, and the Bloom filter implementation with the random number generator is used, p remains as the one and only parameter needed to use the Bloom filter. The approach taken for our experimental setup is therefore one that greatly simplifies the process of selecting parameter values, which is a common weak point when implementing systems that require many parameter settings to be tuned for optimal usage.

Additionally, we carry out experiments to test the effect of the chosen hash function on the Bloom filter. MD5 is chosen as a well-known relatively evenly spread cryptographic hash function that takes slightly longer to compute, and a string-based hash function is chosen as a faster alternative. The string-based hash function is generated using the well-known operation of:

$$hash(s) = s_{[0]} * 31^{(n-1)} + s_{[1]} * 31^{(n-2)} + \ldots + s_{[n-1]} \tag{8}$$

where square brackets are used to describe substring references to a single character.

7. EXPERIMENTAL RESULTS

The results from varying l_{ms} can be seen in Figures 4 and 5, and the results from varying l_{mbs} are displayed in Figures 6 and 7. 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 utilizing 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 dataset is shown in Figure 8. It is clear that the scalability of both Suffix Array techniques

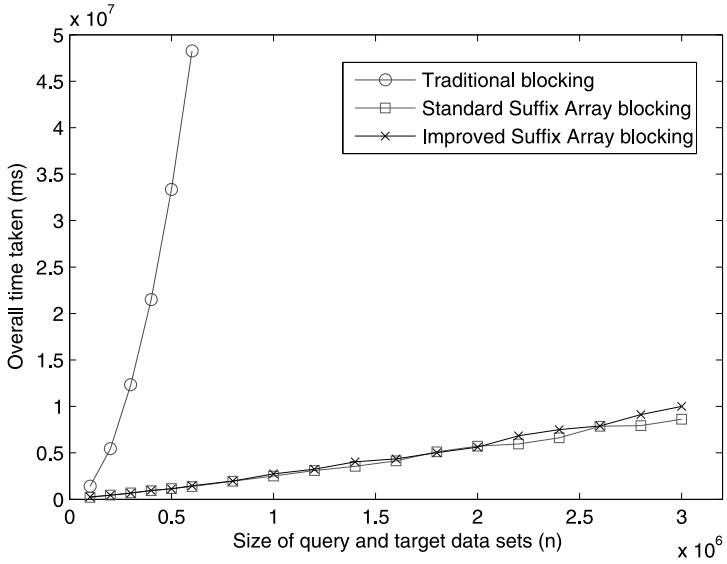

Fig. 8. Overall running time on a large set of real identity data, including index construction and querying for the Suffix Array based techniques.

outperform 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 9(a) and 9(b). 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.

7.1 Bloom Filter Results

Figure 10 shows the percentage of database queries that are able to be skipped due to the use of the Bloom filter as an intermediate step between the code and the database. The percentage of skipped queries fluctuates over n due to the differing distribution of data in the different-sized datasets (we did not carry out any preprocessing steps such as randomization). However, it is clear that when the sample size is large enough, small fluctuations in the data do not affect the percentage of skipped queries, which remains approximately constant.

It can also be seen that the percentage of skipped queries increases approximately linearly with respect to reducing the required false positive rate. When the false positive rate is reduced, however, the amount of storage required increases exponentially. A tradeoff can therefore be negotiated between the amount of storage space available

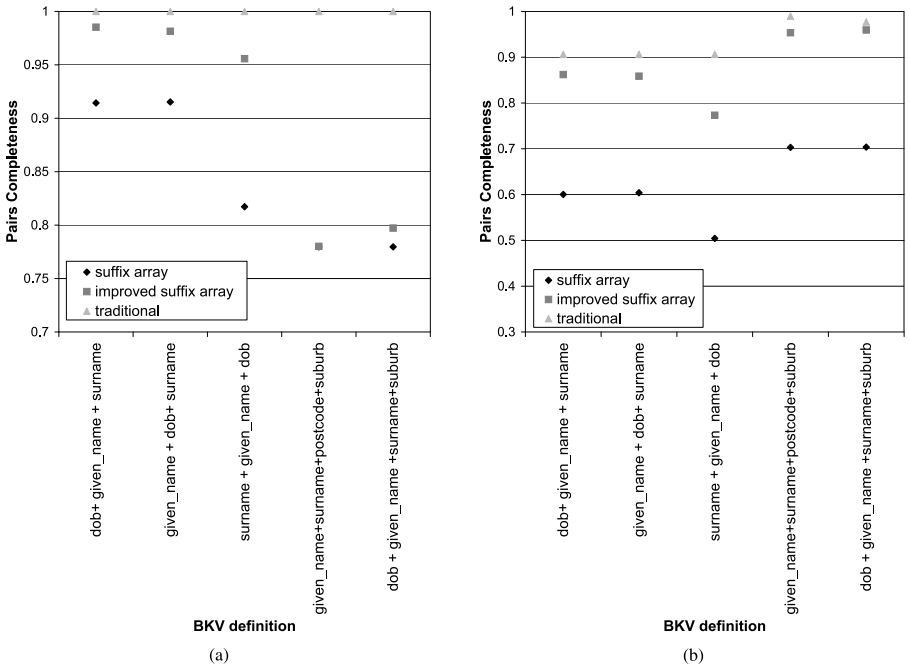

Fig. 9. Pairs completeness for different BKV combinations, using the real identity dataset (a), and the synthetic dataset (b).

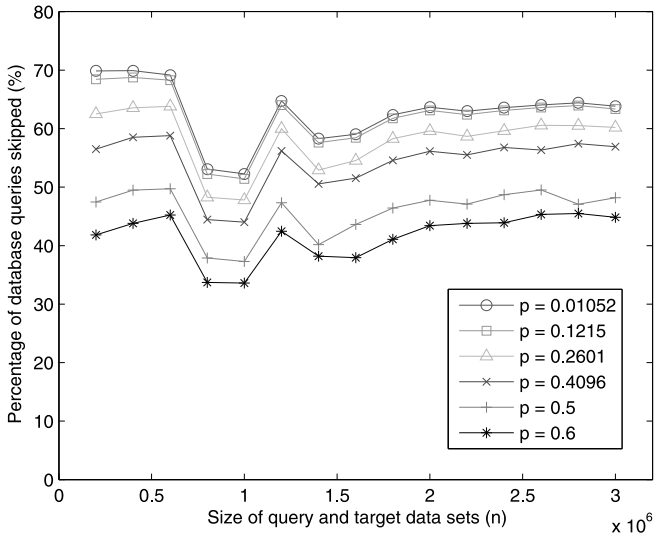

Fig. 10. Percentage of database queries skipped due to the use of a Bloom filter for varying dataset size (n) and expected Bloom filter false positive rate (p).

and the amount of skipped database queries, and consequently the speed of the database querying procedure. As the Bloom filter is very space efficient, a very low false positive rate is often feasible. These results can be used to calculate the real-world time savings from the use of the Bloom filter, if an implementation-specific disk access

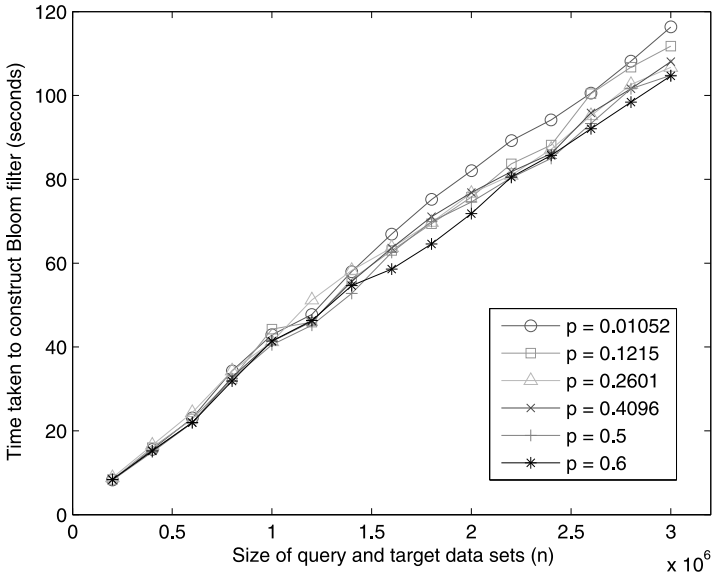

Fig. 11. Time taken to construct the Bloom filter over varying dataset size (n) and expected Bloom filter false positive rate (p).

time is known, along with query and construction times for an implementation-specific Bloom filter.

Figure 11 shows the time taken to construct the Bloom filter on the database server. Different values for the Bloom filter expected false positive rate does not significantly affect the construction time, which is linear in the number of records.

Figures 12 and 13 show the amount of main memory (RAM) required by the Bloom filter for varying dataset sizes and required false positive rates. It is clear that storage requirements increase linearly with dataset size and exponentially with more stringent false positive rate requirements. As the percentage of skipped database queries decreases exponentially with p, and the storage requirement of the Bloom filter increases exponentially with p, it is easy to select a breakpoint with a moderate false positive rate such as $p = 0.1$ to minimize the amount of storage required while still enjoying a large amount of speedup. The parameter p may also be adjusted automatically with the amount of storage available, resulting in a parameter-free Bloom filter implementation.

Figure 14 shows the time taken to build the Bloom filter, comparing Standard Suffix Array (SSA) and Improved Suffix Array (ISA), as well as the effect of using the string-based hash function over MD5. As expected, MD5 takes longer to compute than the string-based hash function, and ISA takes longer than SSA due to the extra suffixes it needs to process. It is also clear that the amount of database queries skipped is almost completely unaffected by these parameter choices. This effect is mainly due our robust Bloom filter implementation using a random number generator, which provides a good result no matter which hash function is used. Under most Bloom filter implementations, MD5 and other cryptographic functions are generally found to cause less false positives than simpler string-based hash functions, with the downside of course being that they are generally slower to use, something that can be avoided without repercussions with the robust Bloom filter.

Figure 15 compares the percentage of database queries skipped for SSA and ISA, while also comparing the use of the string-based hash function to MD5. It is clear that

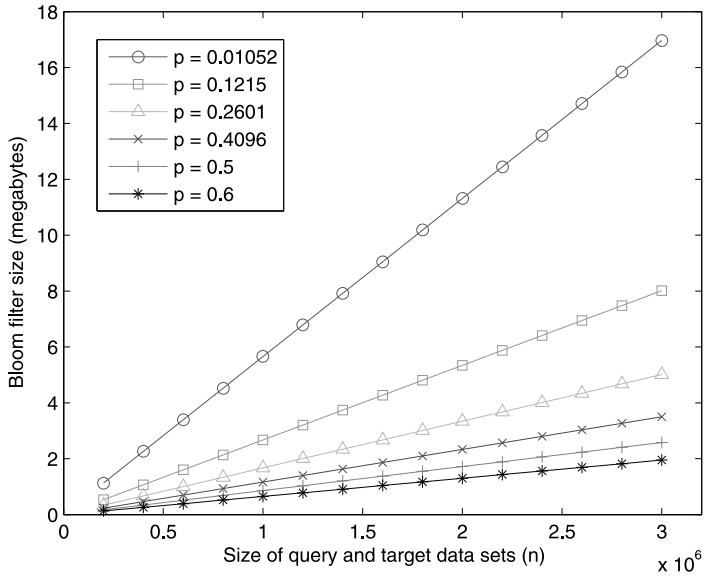

Fig. 12. Storage size required by the Bloom filter for varying dataset size (n) and expected Bloom filter false positive rate (p).

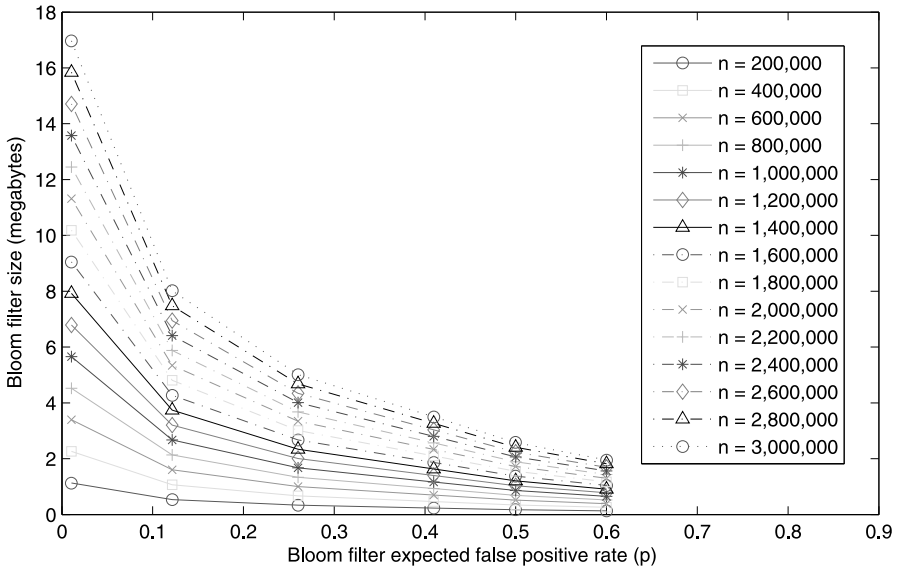

Fig. 13. Storage size required by the Bloom filter for varying dataset size (n) and expected Bloom filter false positive rate (p).

the choice of hash function or the use of the grouping step of ISA does not adversely affect the result.

8. CONCLUSION

We show that an improved Suffix Array blocking method retains high efficiency and benefits from greatly improved accuracy. The new technique is shown to significantly

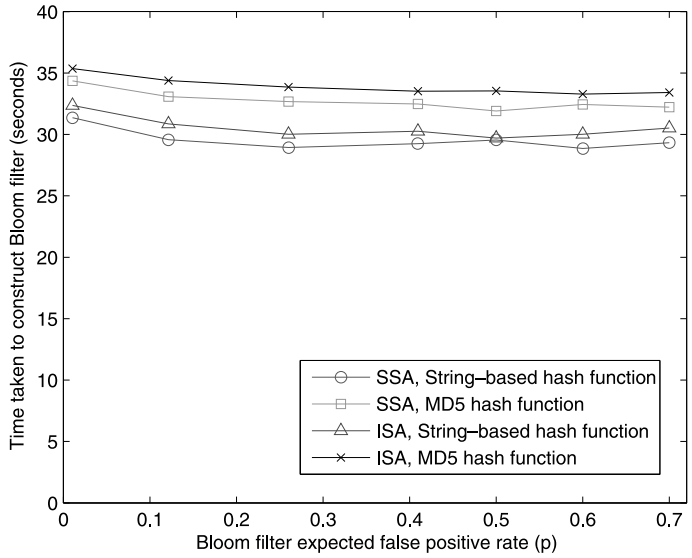

Fig. 14. Time taken to construct the Bloom filter for 600,000 records and varying Bloom filter false positive rate (p). Standard suffix array (SSA) is compared with Improved Suffix Array (ISA) for two choices of hash functions.

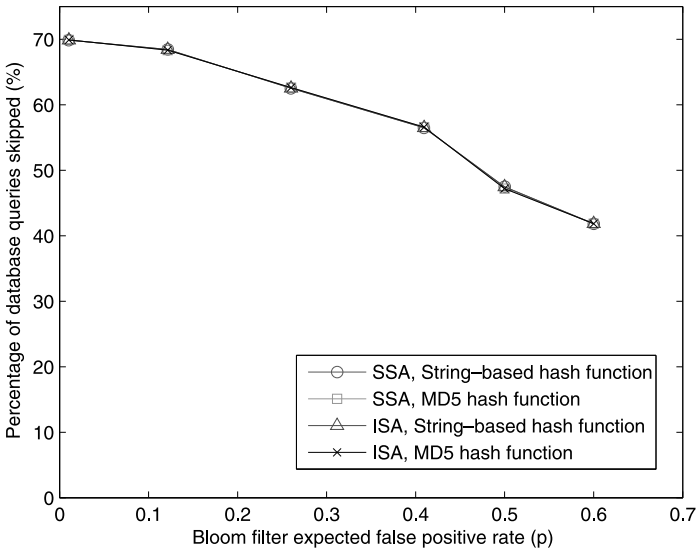

Fig. 15. Percentage of database queries skipped due to the use of a Bloom filter for 200,000 records, varying Bloom filter false positive rate (p). Standard suffix array (SSA) is compared with Improved Suffix Array (ISA) for two hash functions.

outperform traditional blocking in efficiency. These 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 datasets containing millions of records. This is a common situation in many industrial applications where many large datasets exist, both current and archival, and it is beneficial to bring data together from different sources to increase

the amount of knowledge that is available to inform and drive decisions. Scalability is a very important factor for very large scale record linkage tasks that are increasingly needed in an age with exponential data growth and storage.

The use of a Bloom filter implemented in main memory was found to greatly reduce the number of costly database queries that had to be carried out under the standard implementation. The Bloom filter was found to be a remarkably space-efficient structure for this purpose and was easily applied to the Improved Suffix Array technique. Construction time was found to be minimal, and the use of the Bloom filter always resulted in time savings even for our experimental implementation where a disk operation was relatively inexpensive due to no network travel time and standard disk access time. The benefits of using a Bloom filter are amplified greatly when network travel time is required such as for distributed large scale record linkage, or when disk access is slower such as for legacy system linkage. The Bloom filter implementation using a random number series with hash-generated seeds was found to cause no noticeable difference between the use of an efficient string-based hash function and a more balanced hash function (MD5), allowing us to take advantage of the time saved by using one efficient hash function instead of the standard k expensive hash functions. Additionally, this implementation allows for an almost parameter-free Bloom filter, which is a great benefit for applications where the fine tuning of the many parameters and different hash functions required by Bloom filters is not feasible.

REFERENCES

AIZAWA, A. AND OYAMA, K. 2005. A fast linkage detection scheme for multi-source information integration. In *Proceedings of the International Workshop on Challenges in Web Information Retrieval and Integration (WIRI'05)*. 30–39.

BAWA, M., BAYARDO, JR., R. J., AND AGRAWAL, R. 2003. Privacy-preserving indexing of documents on the network. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*. VLDB Endowment, 922–933.

BAXTER, R., CHRISTEN, P., AND CHURCHES, T. 2003. A comparison of fast blocking methods for record linkage. In *Proceedings of the Workshop on Data Cleaning, Record Linkage and Object Consolidation (ACM SIGKDD'03)*.

BLOOM, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Comm. ACM 13*, 7, 422–426.

BRODER, A. AND MITZENMACHER, M. 2004. Network applications of bloom filters: A survey. In *Internet Mathematics*. Vol. 1. 485–509.

CHAZELLE, B., KILIAN, J., RUBINFELD, R., AND TAL, A. 2004. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*. SIAM, Philadelphia, PA, 30–39.

CHRISTEN, P. 2006. A comparison of personal name matching: Techniques and practical issues. In *Proceedings of the Workshop on Mining Complex Data (MCD)*.

CHRISTEN, P. 2007. Towards parameter-free blocking for scalable record linkage. Tech. rep. TR-CS-07-03, Department of Computer Science, Australian National University, Canberra.

CHRISTEN, P. 2008. Febrl – An open source data cleaning, deduplication and record linkage system with a graphical user interface (Demonstration Session). In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD'08)*. 1065–1068.

DE VRIES, T., KE, H., CHAWLA, S., AND CHRISTEN, P. 2009. Robust record linkage blocking using suffix arrays. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM'09)*. ACM, New York, NY, 305–314.

DUNN, H. L. 1946. Record linkage. In *Am. J. Pub. Health*. 1412–1416.

ELFEKY, M., VERYKIOS, V., AND ELMAGARMID, A. 2002. Tailor: a record linkage toolbox. In *Proceedings of the 18th International Conference on Data Engineering*, 17–28.

ELMAGARMID, A., IPEIROTIS, P., AND VERYKIOS, V. 2007. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Engin. 19*, 1, 1–16.

FELLEGI, I. P. AND SUNTER, A. B. 1969. A theory for record linkage. *J. Amer. Statist. Ass. 64*, 328, 1183–1210.

GILL, L., GOLDACRE, M., SIMMONS, H., BETTLEY, G., AND GRIFFITH, M. 1993. Computerised linking of medical records: Methodological guidelines. *J Epidemi. Commun. Health 47*, 4, 316–319.

GU, L., BAXTER, R., VICKERS, D., AND RAINSFORD, C. 2003. Record linkage: Current practice and future directions. Tech. rep., CSIRO Mathematical and Information Sciences.

HERNANDEZ, M. A. AND STOLFO, S. J. 1998. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Min. Knowl. Discov. 2*, 1, 9–37.

JARO, M. A. 1989. Advances in record-linkage methodology as applied to matching the 1985 census of tampa. *J. Amer. Statist. Ass. 84*, 406, 414–420.

JIN, L., LI, C., AND MEHROTRA, S. 2003. Efficient record linkage in large datasets. In *Proceedings of the 8th International Conference on Database Systems for Advanced Applications (DASFAA'03)*. 137.

KIRSCH, A. AND MITZENMACHER, M. 2008. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algor. 33*, 2, 187–218.

MARSHALL, J. T. 1947. Canada's national vital statistics index. In *Population Stud*. 204–211.

MCCALLUM, A., NIGAM, K., AND UNGAR, L. H. 2000. Efficient clustering of high-dimensional datasets with application to reference matching. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*. 169–178.

MITZENMACHER, M. 2002. Compressed bloom filters. *IEEE/ACM Trans. Netw. 10*, 5, 604–612.

NEWCOMBE, H. B. AND KENNEDY, J. M. 1962. Record linkage: making maximum use of the discriminating power of identifying information. *Comm. ACM 5*, 11, 563–566.

SCHNELL, R., BACHTELER, T., AND REIHER, J. 2009. Privacy-preserving record linkage using bloom filters. *BMC Med. Informatics Decis. Mak. 9*, 1, 41.

WINKLER, W. E. 2006. Overview of record linkage and current research directions. Tech. rep. RR2006/02, US Bureau of the Census.

YAN, S., LEE, D., KAN, M.-Y., AND GILES, L. C. 2007. Adaptive sorted neighborhood methods for efficient record linkage. In *Proceedings of the 7th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL'07)*. ACM, New York, NY, 185–194.

ZOBEL, J. AND MOFFAT, A. 2006. Inverted files for text search engines. *ACM Comput. Surv. 38*, 2.

Received January 2010; revised May 2010; accepted July 2010