open proceedings

Summarization Algorithms for Record Linkage

Dimitrios Karapiperis
Hellenic Open University
Patras, Greece
dkarapiperis@eap.gr

Aris Gkoulalas-Divanis
IBM Watson Health
Cambridge, MA, USA
gkoulala@us.ibm.com

Vassilios S. Verykios
Hellenic Open University
Patras, Greece
verykios@eap.gr

ABSTRACT

Record linkage has received significant attention in recent years due to the plethora of data sources that have to be integrated to facilitate data analyses. In several cases, such an integration involves disparate data sources containing huge volumes of records and must be performed in near real-time in order to support critical applications. In this paper, we propose the first summarization algorithms for speeding up *online* record linkage tasks. Our first method, called `SkipBloom`, summarizes efficiently the participating data sets, using their blocking keys, to allow for very fast comparisons among them. The second method, called `BlockSketch`, summarizes a block to achieve a constant number of comparisons for a submitted query record, during the matching phase. Additionally, we extend `BlockSketch` to adapt its functionality to streaming data, where the objective is to use a constant amount of main memory to handle potentially unbounded data sets. Through extensive experimental evaluation, using three real-world data sets, we demonstrate the superiority of our methods against two state-of-the-art algorithms for online record linkage.

1 INTRODUCTION

Massive amounts of data, stored in disparate data sources, have to be integrated and matched to support data analyses that can be highly beneficial to businesses, governments, and academia. Record linkage, also known as *entity resolution* or *data matching*, is the process of identifying records that *match*, i.e., refer to the same real-world entity. The lack of common unique identifiers for records that belong to different data sources, as well as the existence of variations, errors, misspellings, and typos in various data fields, constitute record linkage a challenging process. Traditionally, record linkage consists of two main steps: *blocking* and *matching*. In the blocking step, records that potentially match are grouped into the same block. Subsequently, in the matching step, records that have been blocked together are examined to identify those that match. Matching is implemented using either a *distance function*, which compares the respective field values of a record pair against specified distance thresholds, or a *rule-based approach*, e.g., "if the surnames and the zip codes match, then classify the record pair as matching".

Several blocking approaches have been developed with the aim to scale the record linkage process to Big data sets without sacrificing accuracy [1, 6, 14, 32]. These methods perform the linkage process offline and provide the result set only when the entire linkage process has been completed. Given the size of modern data sets and the costly operations that have to be performed for record linkage, offline methods can take a significant amount of time to produce the matchings. There are many cases though, where *the linkage process has to return a fast response in order to allow for emergency actions to be triggered.* Let us assume, for example, a central crime detection system that collects data from several sources, such as crime and immigration records, central citizens' repositories, and airline transactions. Query data about a suspect could be submitted to this system in order to be matched with any possible similar records found therein. The results of this process have to be reported as fast as possible or, at least, within an acceptably low time period, in order to trigger police enforcement actions.

As another example, consider the recent series of bank and insurance company failures, which triggered a financial crisis of unprecedented severity. In order for these institutions to recover and return to normal business operation, they had to engage in merger talks. One of the driving forces of such mergers is the appreciation of the extent to which the customer databases of the constituent institutions are shared, so that the benefits of the merger can be proactively assessed in a timely manner. A very fast estimation of the extent of the overlap of the customer databases is thus a decisive factor in the merger process. To achieve this, the data custodians could use *summaries of their databases* in order to quickly estimate the overlap of their customers, instead of engaging in a tedious record linkage task. Although our motivation comes from the summarization of the blocking structure of a database, we believe that database summarization is an area of great interest with applications beyond record linkage.

To support real-world applications where record linkage has to be performed in near real-time, several online record linkage approaches have been proposed in the literature [5, 10, 24, 31]. These approaches require the availability of large amounts of main memory, which is necessary in order to store their corresponding data structures. For instance, [5] utilizes large inverted indexes, while [10, 24, 31] sort the records to form blocks by leveraging large matrices or huge graphs. Despite several efforts to utilize small amounts of memory, e.g., [24], the results in terms of performance clearly indicate the *inability of these algorithms to handle an increasingly large volume (or a continuous stream) of records in a real-time fashion.* Given that main memory is always *bounded* and the number of records may in several real-world applications be *unbounded*, the performance of these data structures quickly degrades significantly. Furthermore, in order to deal with this plethora of records and detect the matching pairs, the proposed methods usually resort to conducting *an excessive number of distance computations.* This strategy, however, is not efficient, since it incurs significant delays to the record linkage process.

In this paper, we introduce three methods for efficiently managing large volumes of records in the context of online record linkage. Our first method, called `SkipBloom`, performs a summarization (synopsis) of the blocking structure of a data set using a small footprint of main memory, whose size is logarithmic in the number of distinct processed blocking keys. This synopsis can be easily transferred to another site (or used remotely) to estimate the common number of blocking keys. Such a preliminary estimation may bring to surface important insights, which can

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

be further analyzed by the data custodians. The outcome of such analyses may encourage (or discourage) the data custodians to conduct a full-scale record linkage task.

Our second method, called BlockSketch, tackles the problem of blocks that are overwhelmed with records, which should be compared against a query record to detect matching pairs. BlockSketch instead of implementing the naïve linear approach, compares the query record with *a constant number of records in the target block*, which entails a bounded matching time. In order to achieve this optimization, BlockSketch compiles, for each block, a number of sub-blocks, which reflect the distances of the underlying blocked records from the blocking key. The algorithm places a query record to the sub-block whose records exhibit the smallest distances from the query record.

Our third method, called **S**BlockSketch, operates on data streams, where the entire data set is not known a-priori but, instead, there is an unbounded stream of incoming data records. **S**BlockSketch maintains a constant number of blocks in main memory at the cost of a time overhead during their replacement with blocks that reside in secondary storage. In this scheme, we propose a selection algorithm to effectively select the blocks that should be replaced, by taking into account their selectivity (by the incoming records) and age.

To the best of our knowledge, SkipBloom is the first algorithm for creating an appropriate synopsis of a blocking structure, while BlockSketch and **S**BlockSketch are the first methods for sufficiently summarizing a block for the needs of the matching phase of a record linkage task.

The rest of this paper is structured as follows: Section 2 presents the related work, while Section 3 outlines the building blocks utilized by our algorithms and provides the formal problem definition. Sections 4, 5, and 6 present our proposed algorithms from both a practical and a theoretical point of view. The results of our experimental evaluation, including a detailed comparison with baseline methods, are reported in Section 7, while Section 8 concludes this work.

2 RELATED WORK

A significant body of research work has been conducted in record linkage during the last four decades. This work has been nicely summarized in a number of survey articles [4, 9, 30]. However, only a very limited amount of work has targeted the area of near real-time record linkage, such as [5, 7, 8, 12, 15].

In [5], Christen et al. present an approach that involves a preprocessing phase, where the authors compute the similarities between commonly blocked values, using a set of inverted indexes. The authors use the double metaphone [3] method to encode the string values, which are then inserted into the inverted indexes. This scheme is extended in [27], where a heuristic method is presented to index the most frequent values of data fields. This method, though, requires a-priori knowledge of the values in certain fields and is not well-suited for settings where highly accurate results are needed. Ramadan and Christen in [26] utilize a tree structure where a sorting order is maintained according to a chosen field(s). A query record scans not only the node that is inserted, but also its neighboring nodes where similar records may also reside. Nevertheless, the distance computations that should be performed may degrade considerably the performance of this method in online settings.

Dey et al. in [7] develop a matching tree to speed up the decision about the matching status for a pair of records, so that it can be made without the need to compare all field values. However, the performance of this method depends heavily on the training of the matching tree, which requires a large number of record pairs. Moreover, the authors do not draw any attention to the acute problem of reducing the record pairs comparisons. Ioannou et al. in [8] resolve queries under data uncertainty, using a probabilistic database. The effectiveness of their method heavily depends on the potential of the underlying blocking mechanism, which is used implicitly, to produce blocks of high-quality. In [12], Altwaijry et al. propose a set of semantics to avoid resolving certain record pairs. Their scheme, however, focuses on how to resolve generic selection queries (e.g., range queries), rather than on minimizing the query time.

There is also another body of related literature that deals with *progressive* record linkage (e.g., [10, 24, 31]). These techniques report a large number of matching pairs *early* during the linkage process and are quite useful in the event of an early termination of the linkage process, or when there is limited time available for the generation of the complete result set.

The solutions proposed by Whang et al. in [31] and Papenbrock et al. in [24] are empirical and rely heavily on lexicographically sorting the input records to formulate clusters of similar records. Although the sorting technique is quite effective in finding similar values in certain cases, it cannot guarantee identification of matching record pairs. Consider, for example, the similar strings *'Jones'* and *'Kones'*, where the first letter has been mistyped; using [24, 31], the corresponding records would definitely reside in different clusters (assuming a large number of records). Consequently, the corresponding pair of records would never be considered as matching.

More recently, Firmani et al. [10] introduced two progressive strategies that provide formal guarantees of maximizing recall, focusing though only on minimizing the number of queries to an oracle (which is an entity that replies correctly about the linkage status of a pair) and not on minimizing the running time. Both strategies implicitly assume an underlying blocking mechanism that has been applied on the data sets, and heavily rely on the effectiveness of that blocking mechanism. Their most serious shortcoming is the excessive amount of time-consuming similarity computations, which need to be performed between the formulated pairs in the blocks, *without achieving any increase in recall*. For example, in a data set of 3 million records (including the query set), more than 1.3 *billion* similarity computations should be performed without reporting any results!

There is also another body of work, termed as *meta-blocking* [22, 23], which investigates how to restructure the generated blocks with the aim of discarding redundant comparisons. Meta-blocking techniques, however, conduct a cumbersome transformation of a blocking structure into a graph, which renders these techniques not applicable to online settings.

In Section 7, we elaborate further on the approaches of Christen et al. and Firmani et al., which are the state-of-the-art methods with which we compare our proposed techniques.

3 BACKGROUND AND PROBLEM STATEMENT

In this section, we first introduce the necessary background and terminology for the understanding of our proposed schemes, and subsequently derive the problem statement.

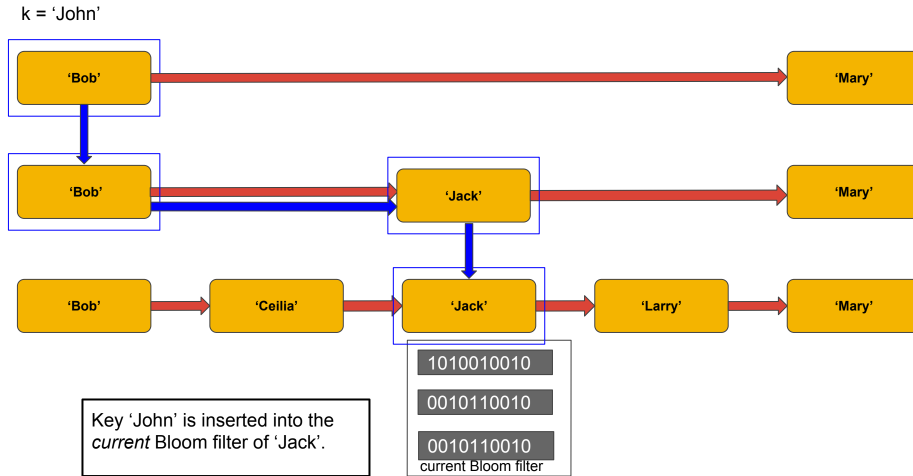


Figure 1: `SkipBloom` inserts and locates a key in logarithmic time using a small amount of main memory. The blue rectangles and arrows indicate the route to locate the nearest key to k.

3.1 Skip List

A skip list [25] is a probabilistic data structure that is designed to provide fast access to an ordered set of items. It is actually a sequence of lists, or *levels*, where the first list, termed as the *base level*, contains all the items inserted so far in sorted order. Each successive list is a copy of the previous with some elements skipped, until the empty list is reached. Its randomization lies in the number of levels an item will join, determined by tossing a fair coin[1]. Each item of each list is linked to the same item in the previous list, as well as to the next item at the same level. The query operation for an item starts at the top-level, by horizontally scanning the items therein until it encounters either the target item or a larger item. In the case of a larger item, the same process is repeated at the lower level until the base level is reached. The running time to insert an item, as well as to report the existence of an item, is $O(\log(n))$, where n is the number of inserted items.

3.2 Bloom Filter

A Bloom filter [2] is a probabilistic data structure for representing a large number of items using a small number of bits, which are initialized to 0, to efficiently support membership queries. Each item is hashed by a set of universal hash functions that map it to certain positions, chosen randomly and uniformly, in the Bloom filter. Accordingly, these positions are set from 0 to 1. Upon querying for an item, the same process is followed, where:

- one can definitely infer that this item has not appeared, if all retrieved positions are set to 0.
- one can conjecture that this item has appeared with certain probability, if all retrieved positions are set to 1. The probabilistic nature of the reply is due to the fact that these positions may have been set to 1 by other items and not the query item.

3.3 Problem Formulation

Consider two data custodians who own data sets A and B, respectively. For each record r of A (or B), the data custodians use a function $k = block(r)$ that generates the blocking key k of r. This key is used to locate a *target block* in the blocking structure to either insert r into the target block (blocking), or iterate all the records already found therein and compare them with r (matching). We use D_A and D_B to denote the set of blocking keys of each of these data sets. Moreover, we refer to the fraction $\mathcal{D} = \frac{|D_A \cap D_B|}{|D_B|}$, as the *overlap coefficient* between A and B.

In this work we introduce three algorithms, namely `SkipBloom`, `BlockSketch`, and `SBlockSketch`, for addressing the following problems[2]:

Problem Statement 1. *Calculate the overlap coefficient for A and B, by accurately summarizing D_A and D_B using* ***sublinear memory requirements*** *and* ***sublinear running time*** *in the number of inserted blocking keys.*

Problem Statement 2. *For each query record of A (or, equivalently, B), find the set of its matching records from B (or, equivalently, A) in* ***constant running time****.*

Problem Statement 3. *For each query record of A (or, equivalently, B), find the set of its matching records from B (or, equivalently, A) in constant time, using also a* ***constant amount of main memory****.*

4 THE OPERATION OF SKIPBLOOM

`SkipBloom` is an efficient blocking data structure that reports membership queries of blocking keys (derived from a large data set) to the blocking structure, using only a small footprint of main memory. It implements the following generic operations:

- `query(k)`: Reports the membership *(true* or *false)* of key k to `SkipBloom`.
- `insert(k)`: Inserts key k into `SkipBloom`.

The operation of `SkipBloom` is based on a skip list that implements a mechanism to locate efficiently a blocking key, as well

[1]As long as *tails* come up, we add the item to each successive list. We terminate this process when we encounter *heads*.

[2]`SkipBloom` aims to address Problem 1, `BlockSketch` targets Problem 2, while `SBlockSketch` tackles Problem 3.

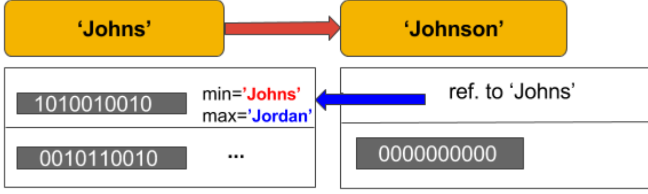


Figure 2: SkipBloom inserts a reference from the list of Bloom filters of *'Johnson'* to the first Bloom filter of *'Johns'*, in order to maintain the consistency of the blocking mechanism.

as on a series of Bloom filters, which are used as fast memory-bounded buffers.

SkipBloom maintains, in expectation, $\sqrt{n}$ blocks in main memory, stored in the base level of the skip list. Each such block, which is represented by its key[3], includes a list of Bloom filters in order to store keys that have been driven by the mechanism of the skip list to this block. This actually means that the keys stored in the Bloom filters of a block are greater than the value of the corresponding key.

The operation of SkipBloom is illustrated in Figure 1. In this figure, a skip list is shown that contains five keys in the base level. Upon receiving a query record, which is first filtered by a blocking function to generate its key (e.g., k = *'John'*), SkipBloom locates the block *'Jack'* very fast, using the logarithmic runtime property of the underlying skip list. According to the operation of skip lists, this block is alphabetically the nearest key to k from the left. The next step is a simple insertion of k into a Bloom filter of *'Jack'*. Each block has an active Bloom filter, termed as *current*, and a number of inactive Bloom filters, which are used only during the query process, as we will shortly explain.

To answer a query on whether a certain key k exists or not, SkipBloom follows almost the same process as described above. Assume, for example, that SkipBloom receives the query k = *'Jonathan'*. First, the skip list will be scanned to eventually locate *'Larry'*. Subsequently, each Bloom filter of this block will be iteratively queried until k is found, or the Bloom filters of *'Larry'* are exhausted.

In what follows, we provide details that will justify certain design choices, such as the reason for maintaining a series of small (in length) Bloom filters in each block, instead of having a larger one. In order to populate the skip list with keys, we apply a simple Bernoulli random sampling algorithm that chooses each key with probability equal to $n^{-1/2}$. This sampling process ensures the uniform reflection of the distribution of keys from the data set to the skip list. This is an appealing feature, since SkipBloom easily tackles distribution anomalies, such as skews of certain ranges of keys, by choosing these keys and inserting them into the skip list to effectively reduce the bottleneck of certain keys and maintain uniformity (in expectation). Any uniform sampling method can be applied; we refer the interested readers to a comprehensive survey in [13].

If a large number of similar keys are generated, then the sampling routine will choose randomly similar keys to create the corresponding blocks. For example, consider the case of blocking a large number of surnames from the US census data. Then, possible blocks might be *'Johns'*, *'Johnson'*, and *'Johnston'*, which will be created in this particular chronological order. Consequently, there will be keys other than *'Johns'*, e.g., *'Jordan'* or *'Jolly'*, that will be inserted into the Bloom filters of *'Johns'*. These Bloom filters should be now transferred to (or referenced by) *'Johnson'*, and then to (by) *'Johnston'*. For this reason, we keep the number of keys that can be inserted into each Bloom filter small; this number will be accurately specified later. Moreover, we annotate each Bloom filter with its smallest and its greatest key, in terms of alphabetical order. By doing so, upon inserting *'Johnson'*, SkipBloom scans iteratively the Bloom filters of *'Johns'* to locate Bloom filters that *might contain* *'Johnson'*, or any greater values. If such Bloom filters exist, a simple reference is established between the block of *'Johnson'* and the corresponding Bloom filters. Figure 2 illustrates the reference of a block to a Bloom filter that belongs to the previous block.

Eventually, a record is stored into a key/value database system, maintaining its original blocking key, regardless of the block that was used in SkipBloom.

[3]Henceforth, *key* and *blocking key* will be used interchangeably.

Algorithm 1 The query operation of SkipBloom.

Input: Skip list *SL*, query key k
Output: *true* if k is found, *false* otherwise
1: *Key p* ← *SL.query*(k)
2: **while** (p.*hasBloomfilters*()) **do**
3: *bf* ← p.*nextBloomfilter*()
4: **if** ($k \geq bf.min$ **AND** $k \leq bf.max$) **then**
5: **if** (*bf.member*(k) == *true*) **then**
6: **return** *true*
7: **end if**
8: **end if**
9: **end while**
10: **return** *false*

4.1 Algorithms

Algorithm 1 illustrates the query operation of SkipBloom. First, the skip list *SL* is queried to locate the nearest key p to the query k (line 1). Then, the Bloom filters that are both maintained and referenced by p[4] (line 2) are scanned iteratively to find k using the min and max values of each Bloom filter (line 4). If k is found, then the algorithm terminates (line 6). In case of composite keys, we perform a conjunction using the individual keys.

Algorithm 2 outlines the insertion of a key in SkipBloom. For each key k derived from each record, we determine with probability $\frac{1}{\sqrt{n}}$ whether k will be inserted into the skip list or not (line 1). In more detail, we generate a random value in $(0, 1)$ and then pick k if this value is less than $\frac{1}{\sqrt{n}}$. Since the generation of a random value is an expensive operation, we exploit the fact that the number of keys skipped between successive inclusions follow a geometric distribution [13]; accordingly, each time we pick a key, we generate the position of the next key, in the stream of records, that will be picked.

If a key k will be inserted into the skip list as a base level key, then a block is created after the nearest key to k (line 2). Then, SkipBloom has to locate each Bloom filter of p that may contain keys that should be now transferred to the newly created block of k (lines 4–8). In order to easily locate these Bloom filters, we annotate each Bloom filter used with the min and max keys it contains (line 5). The inclusion of a Bloom filter with a valid range of keys is achieved through a reference from p to k.

If a key will not be stored in the skip list, then the nearest key p to k is located in order to insert k in the current Bloom filter of

[4]SkipBloom locates these Bloom filters performing a recursive process.

Algorithm 2 The insert operation of `SkipBloom`.

```
Input: Skip list SL, key k
 1: if (nextSample() == true) then
 2:     Key p ← SL.insert(k)          ▷ Key p is the nearest (previous) key to k
 3:     List bfList ← k.createList()  ▷ The list bfList that will
                                        host the Bloom filters of k is created
 4:     for each bf in p do
 5:         if (k ≥ bf.min AND k ≤ bf.max) then
 6:             bfList.add(bf)        ▷ A reference is added
                                        to each Bloom filter found in p
                                        that might contain keys that belong to k
 7:         end if
 8:     end for
 9: else
10:     Key p ← SL.query(q)
11:     bf ← p.getCurrentBloomFilter()
12:     bf.insert(k)
13:     if (k ≤ bf.min) then
14:         bf.min ← k
15:     end if
16:     if (k ≥ bf.max) then
17:         bf.max ← k
18:     end if
19: end if
```

p (lines 10–12). Algorithm 2 eventually updates the `min` and `max` annotations of the current Bloom filter of p (lines 13–18).

4.2 Accuracy and Complexity Analysis

As we expect $\sqrt{n}$ blocks in the base level of the skip list, where the sampling process ensures a uniform distribution of the corresponding blocking keys, the expected number c of keys in each block is:

$$E[c] = \frac{n}{\sqrt{n}} = \sqrt{n}. \quad (1)$$

By setting $u = \sqrt{n}/m$ to be the maximum number of keys that will be stored in each Bloom filter, where m is a constant value (e.g., $m = 10$), the number of Bloom filters in each block will be (in expectation) equal to m. Furthermore, the number m_{bt} of Bloom filters contained in block b at time t, specifies the upper and lower bound of the number n_{bt} of the distinct keys inserted, which is:

$$(m_{bt} - 1)\frac{\sqrt{n}}{m_{bt}} \leq n_{bt} \leq m_{bt}\frac{\sqrt{n}}{m_{bt}}. \quad (2)$$

The accuracy of `SkipBloom` to report the existence of a key depends on the false positive probability parameter fp of the Bloom filters. First, consider the event where a query key does not exist in any Bloom filter of the resulting block. The probability of reporting correctly this event, using one such Bloom filter, is $1 - fp$. Hence, the same probability by using collectively all the m Bloom filters is:

$$(1 - fp)^m, \quad (3)$$

since the content of a Bloom filter is independent from that of another Bloom filter.

In the case that a query key does exist in any[5] Bloom filter of the resulting block, the probability of reporting this event is 1. Therefore, we bound from below the error probability of `SkipBloom` by $1 - (1 - fp)^m$.

Computational complexity: Based on Algorithm 1, the running time of querying `SkipBloom` is $O(\log(\sqrt{n}) + m + m\sqrt{n})$, where the first term denotes the time of scanning the skip list to locate the appropriate block, the second term denotes the time of scanning the Bloom filters found therein, and the last term is the time of scanning the Bloom filters referenced directly or indirectly by the chosen block.

[5]Since, we expect to have duplicate keys, it is quite natural that the same key may be stored into multiple Bloom filters of a block.

Algorithm 2 suggests that the running time of an insertion of a key into `SkipBloom` is $O(\log(\sqrt{n}) + m)$, where the two terms are the time of inserting a key into the skip list and the time of scanning the Bloom filters of the nearest key, respectively.

Memory complexity: The memory requirements of `SkipBloom` are $O(2\sqrt{n} + \sqrt{n}m) = O(\sqrt{n}(2 + m)$, because the skip list contains $O(2\sqrt{n})$ keys and each key in the base level of the skip list consists of $O(m)$ Bloom filters.

4.3 Using SkipBloom as a Synopsis of the Universe of Blocking Keys

`SkipBloom` can be used as a *synopsis*, termed also as *summarization*, of the universe of the blocking keys of a database, in order to facilitate an accurate pre-blocking process. During the execution of this process, the data custodians will resolve very fast the common blocks, which will be of great assistance in estimating the running time, in terms of the number of comparisons that will be needed (by exchanging the number of records in each common block). In turn, the data custodians will determine whether they will perform the linkage process or not, by considering several factors based on these preliminary results. For instance, if the number of common blocks is very small, then (a) the chances of identifying similar, or matching, record pairs are rather slim, and (b) the record linkage process itself may not be cost-effective.

Let us now consider the following scenario. Data custodian A generates a `SkipBloom` from database A, which is submitted to data custodian B. Subsequently, data custodian B iterates her blocking keys and queries the `SkipBloom`, which reports positive or negative answers for the existence of the query keys. This entails $O(n(\log(\sqrt{n}) + m + m\sqrt{n})) = O(n(\log(\sqrt{n}) + \sqrt{n}))$ running time,[6] since each key of B is queried against the `SkipBloom` of A.

To further accelerate this process, data custodian B also generates a `SkipBloom`, to compile a uniform sample of keys and to use this `SkipBloom` to report membership queries. The keys found in the base level of the skip list are now queried against the `SkipBloom` of A, as illustrated in Figure 3.

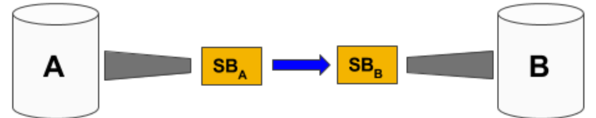


Figure 3: The blocking keys of the databases are packed into their corresponding synopses, each of which is implemented as a `SkipBloom` (symbolized by *SB*). These synopses are used to draw inferences about the source databases.

Since, the keys of B constitute a randomly and uniformly chosen sample, they can be used as input to a Monte Carlo simulation [21], which will estimate the proportion (or the number) of identical blocking keys between the data sets of the two data custodians. Using only the synopses, the data custodians will acquire a clear picture about the overlapping keys with certain approximation guarantees. Monte Carlo simulation requires $(\epsilon^2\vartheta)^{-1}$ (ignoring a small constant factor) keys from B in order to exhibit relative error ϵ with high probability. Since the proportion of identical keys is unknown, we bound it from below with a reasonable value, e.g., $\vartheta = 0.05$, to approach the number $\sqrt{n}$ of the sampled keys that

[6]We assume that the number of distinct blocking keys is n in both A and B.

'John', 'Jon'

'John', 'Jon'	'John', 'Jonkers'
'John', 'Jones', 1970	'John','Jonker', 1975
'John', 'Jonas', 1985	'John', 'Jonkar', 1970

Figure 4: Illustration of a block with $\lambda = 2$ sub-blocks, whose key is <*'John', 'Jon'*>. `BlockSketch` inserts records into the sub-blocks based on the distance of the key values of these records from the chosen representative(s). The sub-block for which one of its representatives exhibits the smallest distance from the key values of a record, is chosen as the target sub-block.

are contained in the `SkipBloom` of B. Even for a relatively small $n = 10^8$, the Monte Carlo simulation will provide its guarantees, since $\sqrt{n}$ is greater than the required number of sampled keys for $\epsilon \geq 0.05$. The fraction of the overlapping keys found in the sample is used as an estimate for the overlap coefficient of the keys between the two databases. By comparing the synopses, we eventually achieve the much faster $O(\sqrt{n}(\log(\sqrt{n}) + \sqrt{n}))$ running time, compared to using only the synopsis of data custodian A.

5 THE OPERATION OF BLOCKSKETCH

The existence of blocks that contain a large number of records makes the *matching phase* (i.e., the comparison of query records against every record found in a target block) prohibitively expensive in highly demanding environments. The situation becomes even more challenging in environments where the matching record pairs have to be reported in near real-time.

To address this shortcoming, in this work we opt for a different strategy: we compare the query record with *a constant number of records of the target block*, which entails a bounded matching time. This optimization requires maintaining λ sub-blocks $(S_1, S_2, \ldots, S_\lambda)$ in each block, whose aim is to represent sufficiently the records inserted so far. In our proposed representation, a number of records play the key role of *representatives* for each sub-block. This allows to formulate groups of records inside each block that are more likely to match. We term our proposed algorithm as `BlockSketch`, because a small number of records comprise a sketch that represents sufficiently the records of an entire block. The concept of *sufficient representation* boils down to choosing representatives that exhibit certain distances from the corresponding blocking key. We note that `BlockSketch` can operate either autonomously or in conjunction with `SkipBloom`, where the latter will be used as a fast bounded memory to report whether a certain blocking key has appeared or not.

The fact that certain records are inserted into a block, using a blocking function, implies that all these records share some degree of similarity. Therefore, it is reasonable to assume that the distance between a key and a record[7] will be upper bounded by $\lambda\theta$. Hence, `BlockSketch` formulates λ sub-blocks, each of which represents records with distances $\leq \theta, \leq 2\theta, \ldots, \leq \lambda\theta$ from the key, where θ is the distance threshold of the keys of a pair of matching records. Upon receiving a key, `BlockSketch` aims to insert this record into the sub-block of the target block, where it is more likely to formulate matching record pairs. For this reason, each key is compared against all representatives found in a block, in order to locate the sub-block whose representative exhibits the smallest distance from the newly arrived key.

As an example, assume that we use edit distance as the similarity metric, $\theta = 2$ and $\lambda = 3$, and a blocking key is used that consists of the first three letters and the whole value of the *surname* and *given name* attributes, respectively. As Figure 4 shows, record <*'John'*, *'Jones'*, 1970>, whose key values exhibit a total distance of $2 \leq \theta$ from <*'John', 'Jon'*>, is inserted into the 1-st sub-block, because of the representative <*'John'*, *'Jon'*>. Similarly, <*'John'*, *'Jonker'*, 1975>, whose distance is $3 \leq 2\theta$ from <*'John'*, *'Jon'*>, is inserted into the 2-nd sub-block, due to the comparison with the representative <*'John'*, *'Jonkers'*>.

It is important to note that for threshold θ any metric that is used in record linkage processes can be supported, whether satisfying the triangle inequality or not. For example, a very commonly used metric in record linkage is the Jaro-Winkler similarity function [3], which takes on values in $[0, 1]$. Hence, one by setting the similarity threshold to θ', and then by choosing $\theta = 1 - \theta'$, produces very reasonable sub-blocks.

The probability for a record to fall into a certain sub-block that holds its matching record, depends on the representatives of the target sub-block, as well as on the left and right neighboring sub-blocks. For instance, assume two neighboring sub-blocks with representatives *'Jacks'* and *'Jackson'*, respectively. The keys of these representatives comprise the values of the *'surname'* attribute. Key *'Jackson'* arrives, whose record is inserted into the identical sub-block of *'Jackson'*. At a later time, *'Jacksn'* arrives, that suffers from a typo, whose record is inserted into the sub-block of *'Jacks'*. We have thus missed the formulation of one matching record pair. `BlockSketch` tackles this deficiency by *using more than one representatives for each sub-block*[8], so as to give more chances for grouping together matching record pairs. By doing so, if record a has been inserted into a sub-block, `BlockSketch` compares the key of its matching record b with more similar representatives to record a. To keep the number of representatives of a sub-block constant, whenever a key is chosen for inclusion in a sub-block, the algorithm tosses a coin to determine if this newly inserted key would be a representative as well. If it is chosen, a randomly picked old representative is evicted from the set of representatives.

As a last step, the query record is inserted into that sub-block which is maintained by a key/value database. The pairs formulated in this sub-block constitute the final result set.

5.1 Algorithm

Algorithm 3 outlines the basic operation of `BlockSketch`. For a query record q, the algorithm first retrieves an object S that contains the corresponding sub-blocks, either from a key/value database or from a cache structure in main memory (line 2). `BlockSketch` then iterates over the representatives of each sub-block and performs the distance computations between the key values of q and these representatives,[9] whose results are stored in array u (line 5). The representative that exhibits the smallest

[7] The distance either between a pair of records, or between a blocking key and a record, is determined by the distances of the certain field values, part of which usually make up the blocking key.

[8] The exact number of representatives will be specified later.

[9] A representative, being essentially a blocking key, has only key values.

distance from the key values of q specifies the sub-block (line 12) into which q is finally inserted (line 17). For ease of presentation, we omit from Algorithm 3 the details regarding the random choice and eviction of a representative from a sub-block.

Algorithm 3 The core operation of `BlockSketch`.

```
Input: Query record q
 1: k ← block(q)                 ▷ Function block(·) generates the
                                   blocking key, which will be used to
                                   look up the corresponding sub-blocks.
 2: SubBlocks S ← retrieve(k)    ▷ S, which is retrieved
                                   from secondary storage or
                                   from a cache structure,
                                   contains the sub-blocks of block k.
 3: for i = 1 to λ do
 4:     for j = 1 to ρ do
 5:         u[i][j] ← d(k, S[i][j])   ▷ S[i][j] denotes the j-th
                                   representative of the i-th sub-block.
 6:     end for
 7: end for
 8: min ← u[1][1]
 9: for i = 1 to λ do
10:     for j = 1 to ρ do
11:         if (min > u[i][j]) then
12:             min ← i          ▷ Find the i-th sub-block whose at least
                                   one of its representatives exhibits
                                   the smallest distance from k.
13:         end if
14:     end for
15: end for
16: represent(k, min)            ▷ Determine with a coin toss
                                   if k would be a representative for
                                   the chosen sub-block.
17: insert(q, k, min)            ▷ Store q in a key/value database by
                                   setting the key as the concatenation of k and min.
```

5.2 Accuracy and Complexity Analysis

The probability of a record to fall into the correct sub-block is $1/\lambda$, since it completely relies on the distance from the corresponding representative. Hence, the inverse probability of a record not falling into the correct sub-block, and therefore not formulating a record pair, is $\leq 1 - 1/\lambda$. In order to amplify the probability of formulating a matching record pair, we give more chances for grouping together the two constituent records, by comparing each key with a number ρ of representatives, chosen randomly and uniformly from the underlying stream. We rigorously specify the required number of representatives that each sub-block should maintain, as the following lemma suggests.

Lemma 5.1. *If a pair of records, which constitute a matching pair, has been brought in a certain block, then by maintaining* $\rho = \lambda \ln(\frac{1}{\delta})$ *representatives in each sub-block, this matching pair is detected with probability at least* $1 - \delta$.

Proof. The probability of not detecting a matching pair that exists in a certain block is $(1 - \frac{1}{\lambda})^{\rho}$. We bound this probability above by δ and solve for ρ in the following:

$$(1 - \frac{1}{\lambda})^{\rho} < \delta \approxeq -\frac{\rho}{\lambda} < \ln(\delta) \Longleftrightarrow \rho > \lambda \ln(\frac{1}{\delta}), \quad (4)$$

since $\ln(1 - \frac{1}{\lambda}) \leq -\frac{1}{\lambda}$. □

We subsequently apply the ceiling function on the value of ρ ($\lceil \cdot \rceil$), in order to select the smallest integer following ρ for the sake of optimality. ■

Computational complexity: The running time of `BlockSketch` is $O(\log n + \lambda\rho)$, which consists of the time to retrieve a block from the database (which is logarithmic[10]), and the execution of the subsequent $\lambda \times \rho$ distance computations (ρ representatives for each of the λ sub-blocks).

Memory complexity: The storage requirements of `BlockSketch` are $O(\lambda n)$, where n is the number of blocking keys.

6 THE OPERATION OF SBLOCKSKETCH

Let us now suppose that the number of records, which are initiated from multiple sources, e.g., from different hospitals, is unbounded (or endless). This literally turns the record linkage scenario of a large number of records, into the record linkage of a stream of records. Therefore, `BlockSketch` will grow in both directions; it will not grow only in terms of sub-blocks, but also its number of blocks might unexpectedly grow considerably. Since our main memory is bounded, `BlockSketch` adapts its operation to record linkage tasks that involve streams of records.

In this version of `BlockSketch`, called **S**`BlockSketch`, we bound the number of blocks, that are maintained in main memory, by an integer value μ which depends on the available main memory. Since the number of blocks is bounded, **S**`BlockSketch` applies an eviction strategy, so as to insert a newly arrived blocking key from the stream, when there is not an empty slot to accommodate the corresponding block. We annotate each *live*[11] block with (a) the number of incoming records that generated its key, i.e., the number ξ of times this block has been chosen as the target block, and with (b) its age α, in terms of the number of times that this block has survived eviction, since its admission into main memory. We derive the eviction status of each block as follows:

$$es = e^{(w\xi - \alpha)}, \quad (5)$$

where factor w adjusts the weight of successes ξ of a block to its *es*. The intuition behind this scheme is that we promote (a) newer blocks against older ones, and (b) blocks that exhibit higher eligibility. The status of old blocks, that are additionally not chosen by the incoming records, will exponentially decay, which will result in their eviction from the main memory. **S**`BlockSketch` is materialized by a hash table, which holds the live blocks, and the corresponding sub-blocks, and a priority queue, that is used to indicate which of these live blocks should be evicted in case of a newly arrived block (key).

Figure 5 illustrates the components of **S**`BlockSketch`, namely the hash table T and the priority queue pq. T exists in main memory and contains a specified number μ of rows, each of which holds a block, as a function of the available main memory. Each row of T contains the sub-blocks of the corresponding block. The priority queue pq stores the eviction status of each live block in ascending order, so as to return the key of the block that holds the minimum eviction status. In the example shown in Figure 5, we observe that the block with key k_4 has survived $\alpha = 4$ evictions and has not been chosen as target block since its admission into T. These two events lead inevitably to its eviction, despite the existence of block k_2, which has $\alpha = 10$ survivals, but it additionally exhibits $\xi = 6$ successes.

[10] For instance, `LevelDB` (see https://github.com/google/leveldb) uses an in-memory highly efficient multi-level data structure, which enables logarithmic disk seeks in the number of stored blocking keys.

[11] A *live* block is a block that is stored in main memory.

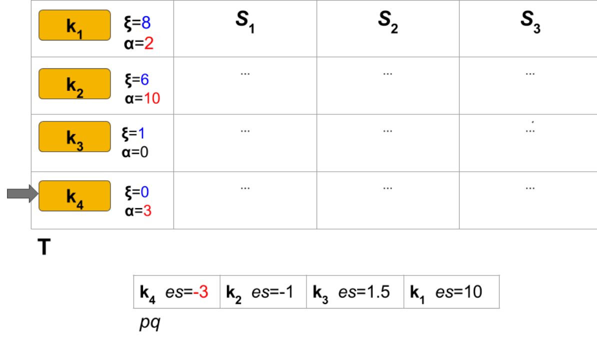


Figure 5: In this example, SBlockSketch uses a hash table T with $\mu = 4$ blocks, $\lambda = 3$ sub-blocks, and the weight of successes set to $w = 1.5$. On the arrival of an incoming new key, the block with key k_4 is evicted because of its low eviction status. The priority queue *pq* stores the eviction status (on a logarithmic scale) of each *live* block.

6.1 Algorithm

Algorithm 4 illustrates the operation of SBlockSketch, using a stream of data records. Upon receiving a record from the stream, the algorithm first derives its key, and then queries T (line 2). Only if this query is fruitless, SBlockSketch resorts to the structures of secondary storage (line 4). If the block that corresponds to the incoming record exists neither in T nor in secondary storage, then SBlockSketch initiates the eviction of the block from T that exhibits the minimum eviction status, as indicated by *pq* (line 7). Eventually, SBlockSketch computes the eviction status of each live block and rebuilds *pq*.

Algorithm 4 The eviction algorithm of SBlockSketch using a stream of records.

Input: Query record q
1: $k \leftarrow block(q)$
2: $SubBlocks\ S \leftarrow T.get(k)$ ▷ Function *get*() retrieves an entry from hash table T.
3: **if** (S == *NULL*) **then**
4: $SubBlocks\ S \leftarrow retrieve(k)$
5: **end if**
6: **if** (S == *NULL*) **then**
7: $SubBlocks\ S \leftarrow pq.poll()$; ▷ *pq* is a priority queue that holds the eviction status of each live block in ascending order.
8: *S.evict*(); ▷ Function *evict*() transfers a certain block, which is essentially a structure of sub-blocks, from main memory into secondary storage.
9: *calculateStatus*(); ▷ Function *calculateStatus*() computes the status of each live block and inserts it into *pq*.
10: **end if**

6.2 Accuracy and Complexity Analysis

The accuracy of SBlockSketch is not affected by the use of T, since the block in question might exist either in main memory or in secondary storage. However, T, whose operations are of $O(1)$ time, affects both running time and space.

Computational complexity: The running time depends on two mutually exclusive possibilities. The first one is when a block exists in T, where the running time is $O(\lambda)$ (see Section 5.2), while the other possibility is when a block should be evicted from T. The eviction requires accessing the priority queue, which is of $O(\sqrt{\mu})$ time, and then transferring the incoming block into T. The latter step consumes, as we discussed in Section 5.2, $O(\log(n))$ time in the number n of available blocks found in the secondary storage. Finally, we have to add the time to build the priority queue, which is $O(\mu \log(\sqrt{\mu}))$. Hence, the total running time for replacing a block is $O(\sqrt{\mu} + \log(n) + \mu \log(\sqrt{\mu}))$.

Memory complexity: The space occupied in main memory is exactly $O(\mu\lambda)$, where μ corresponds to the rows and λ to the cells of T (by assuming T as a two-dimensional array).

Table 1: Technical characteristics of the data sets used. The blocking fields used, and their length (in characters) are shown in bold ($m = 5$).

	DBLP	NCVR	LAB
$\lvert Q \rvert$	300K	500K	100K
$\lvert A \rvert$	300M	500M	100M
fields	***'author'[50%]***, *'venue'*, *'year'*	***'given name'***, ***'surname'[50%]***, *'address'*, *'town'*	***'assay'[6]***, *'result'*, *'year'*
	$u = 3{,}465$	$u = 4{,}473$	$u = 2{,}000$

7 EXPERIMENTAL EVALUATION

For the experimental evaluation, we used three real-world data sets, namely (a) DBLP[12], which includes bibliographic data records, (b) NCVR[13], which comprises a registry of voters, and (c) LAB[14], which includes biological assays (e.g., albumin, hepatitis, or creatinine) and their corresponding results. The technical characteristics of these data sets are summarized in Table 1. For each record of each data set, denoted by Q, we generated 1,000 perturbed records, which were placed in a separate data set symbolized by A. We perturbed all the available fields using at most four edit, delete, insert, or transpose operations, chosen at random.

The blocking methods that were used for the needs of the evaluation were standard [4] and LSH blocking [18], which relies on the Locality-Sensitive-Hashing [11] technique. LSH blocking generates from a single record a certain number of blocking keys that are placed in multiple hash tables. This number of blocking keys is a function of several parameters [19] of LSH blocking, such as the distance threshold. The LSH technique is commonly used in the domain of record linkage [17, 18, 20, 29] because of its efficiency and accuracy guarantees. We used Hamming LSH blocking [18], in which records are embedded into the Hamming space using record-level Bloom filters [28]. LSH blocking implements redundant blocking, because a record is inserted into multiple independent blocks, which are accommodated into independent hash tables. In contrast, standard blocking inserts records that exhibit identical values, in an appropriately chosen blocking field(s), into the same block.

For performing the standard and the LSH blocking, we utilized LevelDB[15] and LSHDB [16], respectively. The length of each Bloom filter, utilized by SkipBloom, was set to 32,000 bits for storing 5,000 keys, with false positive probability set to $fp = 0.05$.

We evaluated our schemes and their competitors according to the time needed, and the memory that was consumed to perform the record linkage process, as well as the recall and precision rates

[12]http://dblp.uni-trier.de/xml
[13]http://dl.ncsbe.gov/index.html?prefix=data/
[14]https://idash-data.ucsd.edu/community/43
[15]https://github.com/google/leveldb

that were achieved. We ran each experiment 20 times and plotted the average values in the figures. The software components were developed using the Java programming language (ver. 1.8) and the experiments were conducted in a virtual machine utilizing 4 cores of a Xeon CPU and 32GB of main memory.

7.1 Baseline Methods

We compared our schemes with two state-of-the-art methods for online record linkage. The first method, termed as INV [5], uses inverted indexes as its basic blocking structure. The main idea behind this method is the pre-computation of similarities between field values that have been inserted into the same block. An inverted index is used for this purpose, which stores the blocking keys encoded by the double metaphone method[16]. A weakness of this structure regards the storage of all field values of a record into the same set of indexes. As a result, one cannot be certain for a value encountered therein, to which field this value belongs. This ambiguity affects negatively the recall rates of INV.

The second method we compared against is the Edge Ordering strategy, termed as EO, which was introduced in [10]. EO utilizes an oracle, which is aware of the ground-truth, to resolve the matching status of a record pair. A graph is constructed by assuming each record pair, which materializes an edge connecting two vertices/records, formulated in each block. The algorithm performs all similarity computations in the target block in order to assign a probability estimate to each edge (pair) based on its similarity. In turn, EO selects those edges that are expected to maximize the recall, and submits them to the oracle that returns their matching status.

Both EO and INV utilize only key/value pairs, materialized by hash tables that map a key to list of record *Id*'s. These methods do not offer any component to report efficiently the membership of a certain key, or to adequately summarize the data set. Thus, in order to be fair in our comparison with these methods, we maintained the key/*Id*'s mappings, as well as the entire records, in secondary storage.

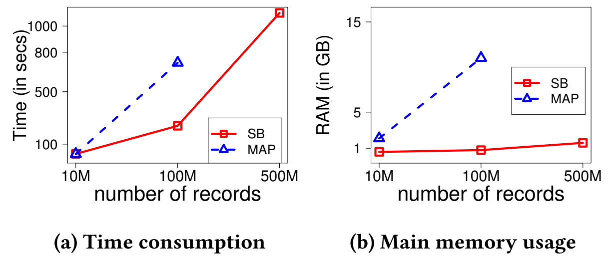
Both the baseline methods and our proposed schemes used the Jaro-Winkler [3] function as the similarity measure, where the corresponding threshold was set to $\theta = 0.75$.

7.2 Experimental Results

In our first set of experiments, we evaluated the running time and memory performance, as well as the ability of the `SkipBloom` algorithm to provide accurate estimates in the pre-processing step of record linkage.

Figure 6a shows the total time needed to build the `SkipBloom`, by scaling the number of the streaming records using the NCVR data set. It is quite obvious that the time increases by a constant factor, depending on the number of records that are processed. The consumption of main memory is illustrated in Figure 6b, where `SkipBloom` exhibits almost linear performance. Specifically, although the number of records increases by 10 and 50 times, `SkipBloom` utilizes 0.6GB, 0.8GB, and 1.4GB of main memory, respectively. In contrast, a map data structure, symbolized by MAP, e.g., a HashMap in the Java programming language, exhibits a steep linear performance. In both scenarios, MAP throws fatal errors and terminates when it reaches the processing of 500M records.

[16]Using the double metaphone encoding method, *'SMITH'* and *'SMYTH'* are both encoded as *'SM0'*.



(a) Time consumption **(b) Main memory usage**

Figure 6: Scaling the number of records to measure the time and space requirements of `SkipBloom`.

Table 2: Time (in seconds) consumed by `SkipBloom` for reporting the existence of a key.

	10M	100M	500M
Time	0.000277	0.000315	0.000365

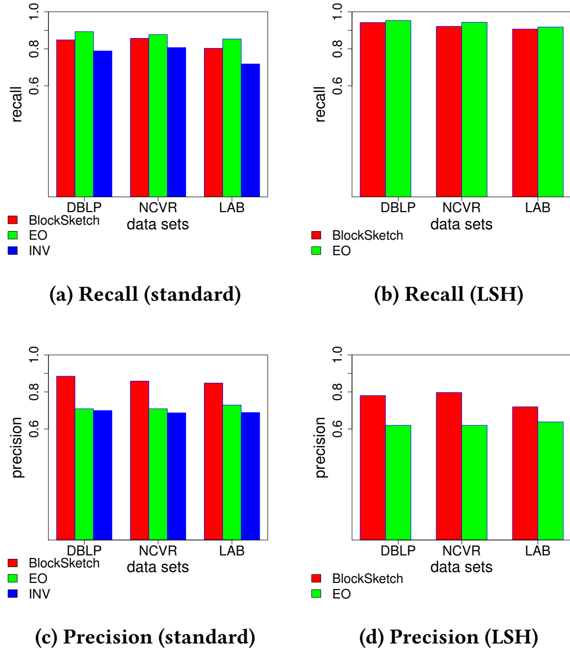
Table 3: Evaluating the accuracy of `SkipBloom` in estimating the fraction of matching pairs.

ϵ	DBLP	NCVR	LAB
.10	0.94 ± .023	0.95 ± 0.021	0.94 ± 0.022
.05	0.97 ± .022	0.98 ± 0.021	0.98 ± 0.024

Table 2 illustrates the time consumed by `SkipBloom` to report the existence of a key. We remind to the reader the probabilistic nature of `SkipBloom`, whose performance depends on the number of comparisons that will take place until the target block is located (which is $O(\log(\sqrt{n}))$). For this reason, we observe that `SkipBloom` almost consumes the same amount of time when it has to process either 100M or 500M records.

The accuracy of `SkipBloom` is evaluated by the fraction of overlapping keys it estimates using the above-mentioned data sets. Table 3 clearly shows that `SkipBloom` approximates the overlap coefficient of A and Q for each data set, where in the worst case it exhibits an error nearly equal to 0.06 (which is within its approximation guarantees specified by ϵ).

In the next set of experiments, we compared our schemes against EO and INV. Figures 7a and 7b display the recall rates achieved by all methods using standard and LSH blocking, respectively. We observe in Figure 7a that EO exhibits slightly better recall rates than `BlockSketch`, by using all data sets, although the differences lie in the small range $[0.01, 0.04]$. Also, INV falls short in formulating those matching pairs that exhibit a high degree of perturbation, which is due to the weakness of the double metaphone scheme to group together such pairs into the same blocks. The recall rates of DBLP and NCVR are also higher than LAB, which is due to the longer (in characters) blocking keys, which render them more tolerant to the perturbation errors. `BlockSketch` achieves to maintain high recall rates, although we have to stress that the underlying blocking method drives the whole linkage process. As Figure 7b suggests, LSH blocking, which leverages redundancy, scores much better rates than standard blocking. Only `BlockSketch` and EO can use LSH blocking, because they essentially run on top of the blocking mechanism.



(a) Recall (standard) (b) Recall (LSH)

(c) Precision (standard) (d) Precision (LSH)

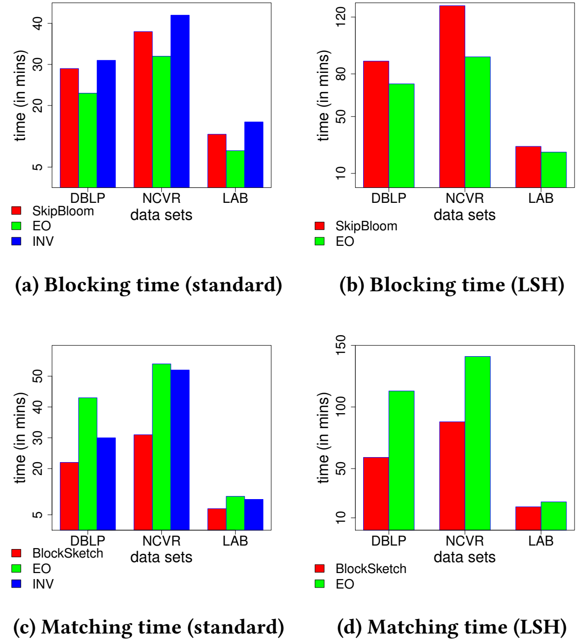
Figure 7: Measuring the recall and precision rates using standard blocking and LSH blocking.

On average, BlockSketch and EO achieve 10% and 8% higher recall rates, respectively, using LSH blocking.

Figures 7c and 7d show the precision rates using the two different blocking approaches described before. As one can observe, BlockSketch outperforms both EO and INV by a large margin, due to the effective categorization of records into the sub-blocks of each block. This minimizes significantly the required number of comparisons. Specifically, the precision rates of EO and INV fall by 18% and 21%, respectively, compared with the rates of BlockSketch. The reasons for this recession vary between the two methods. EO starts to produce meaningful recall rates after performing a large number of comparisons to derive the probability estimates for each pair. These comparisons, however, considerably reduce the precision rates. On the other hand, the double metaphone scheme of INV groups a large number of non-matching pairs into the same block, whose comparisons also result in low precision rates. The redundancy of LSH blocking accounts for the reduced precision rates of both BlockSketch and EO, as shown in Figure 7d, since both methods perform a larger number of comparisons for the pairs formulated in the blocks of each hash table. We observe though that BlockSketch retains its superiority over EO by scoring, on average, rates that are very close to 0.75. The time needed to perform the blocking step is illustrated in Figures 8a and 8b. EO and INV block each record a little faster than the combination of SkipBloom and BlockSketch, which for each insertion have to perform a constant number of comparisons with the representatives of the sub-blocks. Specifically, BlockSketch, through a single `get` operation, retrieves the representatives of a block from the database, as well as replaces them, through a single `set` operation, when needed. INV utilizes three hash tables to store the precomputed similarities, the encoded, and the original field values, which leads to certain delays.

Table 4: Average time (in seconds) for resolving a query record.

	DBLP	NCVR	LAB
standard	0.0051	0.0055	0.0045
LSH	0.0097	0.0098	0.0088



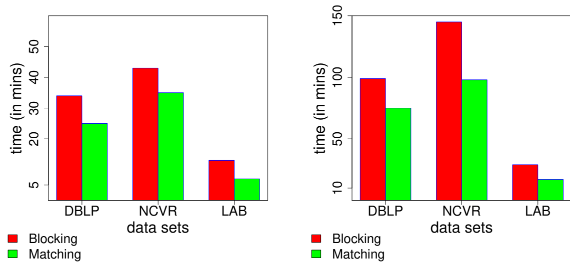
(a) Blocking time (standard) (b) Blocking time (LSH)

(c) Matching time (standard) (d) Matching time (LSH)

Figure 8: Measuring the time needed for blocking and matching for BlockSketch.

In Figures 8c and 8d, we present the time performance of BlockSketch and its competitors for resolving the query data sets, symbolized by Q (see Table 1), after having populated the blocking structures with the records of A. For each query record of Q, BlockSketch performs a constant number of comparisons in each target block, which results in superior performance. As Figure 8c suggests, BlockSketch is 2× and 1.5× faster than EO and INV, respectively, which both struggle to compare all records found in a block. Moreover, EO should build the graph to locate these record pairs that are expected to maximize the recall. Using LSH blocking, which is shown in Figure 8d, both BlockSketch and EO exhibit longer time rates, which are nearly 3× slower than before, due to the inherent redundancy of LSH. Since, a record pair might appear several times during the matching phase, for each record of Q, we utilize a map data structure[17] to discard the comparisons of duplicate record pairs. Table 4 illustrates the time for resolving a single query record of Q during the matching phase. The constant number of distance computations for a single record accounts for the stable time performance of BlockSketch regardless of the size of the corresponding data set. In contrast, EO and INV consume running times which apart from the fact that in most cases they are almost the double of those of BlockSketch, they also depend on the number of records found in each block.

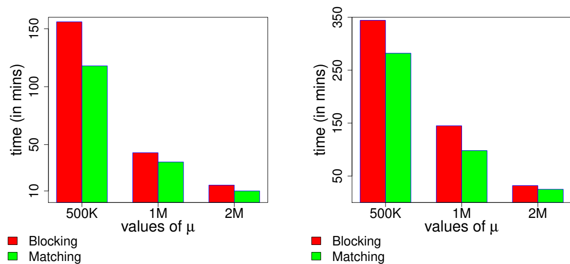
[17] The map structure is initialized for each record of Q.



(a) Running time (standard blocking)

(b) Running time (LSH)

Figure 9: Measuring the time needed for blocking and matching for SBlockSketch.



(a) Running time (standard)

(b) Running time (LSH)

Figure 10: Measuring the time needed for blocking and matching for SBlockSketch by varying μ using the NCVR data set.

In SBlockSketch, we initially set μ to a moderate size (μ = 1M[18]). In Figures 9a and 9b, we observe an average of 10% increase in time consumption than BlockSketch, only in NCVR and DBLP. The large number (over 60M) of distinct blocking keys that are generated in these data sets, resulted in relatively frequent evictions and disk seeks for the replacement of blocks in T. Nevertheless, the eviction status of highly selective (high ξ) but old (high α) blocks remained high during the blocking phase, which prevented their eviction from T. The running time of LAB remained almost intact due to the small number of blocking keys (about 10M) and the corresponding replacements. Since, SBlockSketch utilizes a single hash table T, LSH keys were formulated in a composite format *HashTableNo_Key* to accommodate all of them in T.

We next varied the values of μ and initiated the streaming of records of the NCVR data set. Figures 10a and 10b illustrate the time performance of SBlockSketch, where we observe that by doubling μ, we achieve significantly lower running time. For instance, by setting μ = 1M, the corresponding time value is 43 minutes, which is almost 4× faster than the previous value (156 minutes) on the y-axis. In LSH blocking, the number of incoming records increases by a constant factor, which is the number of the LSH keys that are generated for each record. Since a large number of these keys are identical, the running time increases by 156% on average, as Figure 10b suggests, compared to the use of standard blocking.

[18]We had 32GB of main memory available.

Summary: Based on our conducted experiments, it becomes apparent that our proposed schemes are suitable for processing online queries for performing record linkage by using synopses of the blocking structures maintained in the persistent storage. They significantly outperform the state-of-the-art baselines, which rely their operation on memory-resident indexes regardless of the increasing volume of the underlying data sets.

8 CONCLUSIONS

In recent years, several applications have emerged which require access to consolidated information that has to be computed and presented in near real-time, through the linkage of records residing in voluminous disparate data sources. To address this need, we proposed the first summarization algorithms that operate in the blocking and matching steps of online record linkage to boost their performance. SkipBloom compiles a synopsis of the blocking structure of a data set using a small footprint of main memory, while BlockSketch compares each query record with a constant number of records in the target block, which results in a bounded matching time. Our experimental findings indicate that SkipBloom and BlockSketch outperform the state-of-the-art algorithms, in terms of the time needed, the memory used, and the recall and precision rates that are achieved during the linkage process. SBlockSketch utilizes a constant memory footprint to perform the linkage in settings that use streaming data.

REFERENCES

[1] M. Bilenko, B. Kamath, and R. J. Mooney. 2006. Adaptive blocking: Learning to scale up record linkage. In *ICDM*. 87–96.
[2] A. Broder and M. Mitzenmacher. 2002. Network Applications of Bloom filters: A Survey. In *Internet Mathematics*. 636–646.
[3] P. Christen. 2012. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer, Data-Centric Sys. and Appl.
[4] P. Christen. 2012. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *TKDE* 12, 9 (2012), 1537 –âĂŞ 1555.
[5] P. Christen, R. Gayler, and D. Hawking. 2009. Similarity–aware indexing for real-time entity resolution. In *CIKM*. 1565 âĂŞ– 1568.
[6] W. W. Cohen and J. Richman. 2002. Learning to Match and Cluster Large High-Dimensional Data Sets for Data Integration. In *SIGKDD*. 475–480.
[7] D. Dey, V. Mookerjee, and D. Liu. 2011. Efficient techniques for online Record Linkage. *TKDE* 23, 3 (2011), 373 – 387.
[8] E.Ioannou, W. Nejdl, C. Niederee, and Y. Velegrakis. 2010. On-the-fly entity-aware query processing in the presence of linkage. *PVLDB* 3, 1 (2010), 429–438.
[9] A. Elmagarmid, P. Ipeirotis, and V. Verykios. 2007. Duplicate Record Detection: A Survey. *TKDE* 19, 1 (2007), 1–16.
[10] D. Firmani, B. Saha, and D. Srivastava. 2016. Online Entity Resolution Using an Oracle. In *PVLDB*, Vol. 9. 384 – 395.
[11] A. Gionis, P. Indyk, and R. Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB*. 518–529.
[12] H. Altwaijry and D. Kalashnikov and S. Mehrotra. 2013. Query-driven Approach to Entity Resolution. In *PVLDB*, Vol. 6. 1846–1857.
[13] P. J. Haas. 2016. Data-Stream Sampling: Basic Techniques and Results. *Data Stream Management: Processing High-Speed Data Streams* (2016), 13–44.
[14] M.A. Hernandez and S.J. Stolfo. 1995. The Merge/Purge Problem for Large Databases. In *SIGMOD*. 127–138.
[15] I. Bhattacharya and L. Getoor and L. Licamele. 2006. Query-time entity resolution. In *KDD*. 529–534.
[16] D. Karapiperis, A. Gkoulalas-Divanis, and V. Verykios. 2016. LSHDB: a parallel and distributed engine for record linkage and similarity search. In *ICDM Demo*. 1–4.
[17] D. Karapiperis, D. Vatsalan, V.S. Verykios, and P. Christen. 2016. Efficient Record Linakge Using a Compact Hamming Space. In *EDBT*. 209 – 220.
[18] D. Karapiperis and V.S. Verykios. 2015. An LSH-based Blocking Approach with a Homomorphic Matching Technique for Privacy-Preserving Record Linkage. *TKDE* 27, 4 (2015), 909–921.
[19] D. Karapiperis and V.S. Verykios. 2016. A fast and efficient Hamming LSH-based scheme for accurate linkage. *KAIS* (2016), 1–24.
[20] H. Kim and D. Lee. 2010. Fast Iterative Hashed Record Linkage for Large-Scale Data Collections. In *EDBT*. 525 – 536.
[21] R. Motwani and P. Raghavan. 1995. *Randomized Algorithms*. Cambridge Univ. Press.
[22] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. 2014. Meta-blocking: Taking Entity Resolution to the Next Level. *TKDE* 26, 8 (2014), 1946–1960.

[23] G. Papadakis, G. Papastefanatos, and G. Koutrika. 2014. Supervised meta-blocking. In *PVLDB*. 1929–1940.
[24] T. Papenbrock, A. Heise, and F. Naumann. 2015. Progressive Duplicate Detection. *TKDE* 27, 5 (2015), 1316 – 1329.
[25] W. Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *CACM* 33, 6 (1990), 668–676.
[26] B. Ramadan and P. Christen. 2014. Forest-Based Dynamic Sorted Neighborhood Indexing for Real-Time Entity Resolution. In *CIKM*. 1787 âĂŞ– 1790.
[27] B. Ramadan, P. Christen, H. Liang, R. Gayler, and D. Hawking. 2013. Dynamic Similarity-Aware Inverted Indexing for Real-Time Entity Resolution. In *PAKDD Workshops*. 47 âĂŞ– 58.
[28] R. Schnell, T. Bachteler, and J. Reiher. 2009. Privacy-preserving Record Linkage using Bloom Filters. *Central Medical Inf. and Decision Making* 9 (2009).
[29] R. Steorts, S. Ventura, M. Sadinle, and S. Fienberg. 2014. A Comparison of Blocking Methods for Record Linkage. In *PSD*. 253–268.
[30] D. Vatsalan, P. Christen, and V.S. Verykios. 2013. A Taxonomy of Privacy-Preserving Record Linkage Techniques. *Inf. Sys.* 38, 6 (2013), 946 – 969.
[31] S. E. Whang, D. Marmaros, and H. Garcia-Molina. 2013. Pay–as–you–go Entity Resolution. *TKDE* 25, 5 (2013), 1111–1124.
[32] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. 2009. Entity resolution with iterative blocking. In *SIGMOD*. 219–232.