Dynamic sorted neighborhood indexing for real-time entity resolution

Article

Accepted Version

Dynamic Sorted Neighborhood Indexing for Real-Time Entity Resolution

BANDA RAMADAN, PETER CHRISTEN, and HUIZHI LIANG, Australian National University ROSS W. GAYLER, Veda

Abstract: Real-time Entity Resolution (ER) is the process of matching query records in subsecond time with records in a database that represent the same real-world entity. Indexing techniques are generally used to efficiently extract a set of candidate records from the database that are similar to a query record, and that are to be compared with the query record in more detail. The sorted neighborhood indexing method, which sorts a database and compares records within a sliding window, has been successfully used for ER of large static databases. However, because it is based on static sorted arrays and is designed for batch ER that resolves all records in a database rather than resolving those relating to a single query record, this technique is not suitable for real-time ER on dynamic databases that are constantly updated. We propose a tree-based technique that facilitates dynamic indexing based on the sorted neighborhood method, which can be used for real-time ER, and investigate both static and adaptive window approaches. We propose an approach to reduce query matching times by precalculating the similarities between attribute values stored in neighboring tree nodes. We also propose a multitree solution where different sorting keys are used to reduce the effects of errors and variations in attribute values on matching quality by building several distinct index trees. We experimentally evaluate our proposed techniques on large real datasets, as well as on synthetic data with different data quality characteristics. Our results show that as the index grows, no appreciable increase occurs in both record insertion and query times, and that using multiple trees gives noticeable improvements on matching quality with only a small increase in query time. Compared to earlier indexing techniques for real-time ER, our approach achieves significantly reduced indexing and query matching times while maintaining high matching accuracy.

CCS Concepts: **Information systems** → **Nearest-neighbor search**; Data cleaning;

Additional Key Words and Phrases: Braided tree, data matching, dynamic indexing, real-time query, record linkage

INTRODUCTION

Massive amounts of data are nowadays being collected by most business and government organizations. Given that many of these organizations rely on information in their day-to-day operations, the quality of the collected data has a direct impact on the quality of the produced outcomes. Various data cleaning practices are employed

to improve the collected data. One important practice in data cleaning is the task of identifying all records that refer to the same real-world entity [Naumann and Herschel 2010]. This process is commonly called *Entity Resolution* (ER) [Christen 2012a]. A real-world entity can be a person, a product, a business, or any other object that exists in the real world. Examples of multiple records in a database representing a single entity include a patient who is represented several times in a hospital database, a product that is inserted many times into an inventory list, or a voter who is registered more than once in an electoral roll. These duplicates, if not removed or merged, can lead to serious consequences for organizations or individuals. A patient's information could, for example, be dispersed between duplicated records leaving medical staff unaware of the patient's overall condition, which can potentially affect diagnosis and treatment, while duplicate records in an electoral roll can lead to voting irregularities.

ER is challenging because databases usually do not contain unique entity identifiers. In this case, partially identifying attribute values (such as names and addresses) need to be used for the matching process. However, such attribute values are often of low quality, as they can be incomplete, contain errors, or change over time [Christen 2012a]. Therefore, approximate matching techniques are generally required.

Since many services in both the private and public sectors are moving online, organizations increasingly require real-time ER (with subsecond response times) on query records that need to be matched with existing databases [Dong and Srivastava 2013]. These databases are often not static, but rather dynamic as queries potentially result in a record being modified, added, or even removed (depending upon the application).

Our work is based on a real-world application: the real-time matching of personal information from consumer credit applications. In this application, personal details of a customer are provided, for example, by a bank (where the customer applies for credit) and sent to a credit bureau. The bureau's ER system needs to return (in real time) a ranked list of matched candidate records from a large credit history database that stores the personal details and credit status of previous customers. In such an application, the previous details (like earlier addresses and names) of credit holders need to be kept, and records are not cleaned and merged because this could lead to loss of information (e.g., if previous address details of a customer are removed in the merging step). Rather, a query record is assigned the same *entity identifier* as an existing, older database record if it is classified as a match with that database record, and added to the database. If it is not matched to any existing database record it is assigned a new entity identifier and is assumed to be a new customer. In the context of identity fraud, which is an increasing problem in the consumer credit domain [Phua et al. 2012], it is crucial that one can match a query record to a database record of a customer with an older address, for example, as such characteristics might help identify a potential fraudster who claims to have a certain name and live at that address. In this application, the ranked list of matched database records (according to their similarities with a query record) is returned together with information if some of these records have previously been matched to a known customer (e.g., they are given the same entity identifier).

While real-time ER is becoming more important, most current ER techniques are based on batch algorithms that are only suitable for static databases. Such algorithms compare and resolve all records in one or more database(s) rather than resolving those relating to a single query record. Therefore, there is a need to develop new techniques that support ER for large dynamic databases that can resolve (streams of) query records in real time. A major aspect of achieving this goal is to develop novel indexing techniques that allow dynamic updates and facilitate real-time matching by generating a small number of high-quality candidate records that are to be compared with a query record.

Contributions: In Ramadan et al. [2013] we proposed a tree-based dynamic sorted neighborhood index that facilitates matching a stream of query records against a large database in real time with fixed and adaptive window approaches.We presented a multitree indexing technique in Ramadan and Christen [2014] that improves the matching quality of Ramadan et al. [2013]. In this article, we extend our work to include a more detailed and formalized description of the previously proposed techniques. We justify and evaluate the choice of the selected tree data structure, and propose a new adaptive window approach for retrieving candidate records. We provide a comprehensive theoretical analysis to estimate the required number of candidate records generated. Furthermore, we conduct a more extensive experimental evaluation using one large real-world and two synthetic datasets. We also provide a comparative evaluation of our proposed indexing technique with an existing q-gram technique that has previously been used for ER [Baxter et al. 2003; Christen 2012b; McCallum et al. 2000].

Outline: In the following section we provide an overview of relevant work related to real-time and dynamic ER, and in Section 3 we provide the required background including details about the sorted neighborhood method and the data structures that we use. In Sections 4, 5, and 6 we detail the different variations of our approach. Next, in Section 7 we provide an analysis of the proposed approach in relation to estimating the number of generated candidate records. We experimentally evaluate our approach on different datasets in Section 8, and conclude our article in Section 9 with a discussion of future research directions.

2. RELATED WORK

ER is related to general similarity search approaches, which involve finding similar entities from unstructured databases (such as e-mails, news articles, or scientific publications) based on a collection of relevant features that are represented as points in high-dimensional attribute spaces [Gionis et al. 1999]. However, such approaches are less suited for structured databases that contain well-defined attributes with short values, such as personal names, addresses, or dates of birth. Records in structured databases can be matched using Structured Query Language (SQL) join statements if unique entity identifiers, such as passport or social security numbers, are available. However, such identifiers are commonly not available, and therefore ER approaches need to be employed [Elmagarmid et al. 2007; Naumann and Herschel 2010].

The ER process encompasses several steps [Christen 2012a]: *data preprocessing*, which cleans and standardizes the data to be used; *indexing*, which reduces the number of candidate record pairs to be compared in detail; *record comparison*, which compares candidate record pairs in detail using a set of similarity functions; *classification*, where pairs or groups of compared records are classified into matches (records that are assumed to correspond to the same entity) and nonmatches (records that are assumed to correspond to different entities); and finally, *evaluation*, where the ER process is evaluated with regard to matching accuracy, efficiency, and completeness using various measures.

Many ER models were proposed in previous years (see Christen [2012a], Elmagarmid et al. [2007], and Winkler [2006] for existing ER models). However, recently, incremental ER techniques (that resolve entities based on previously resolved results) were proposed [Gruenheid et al. 2014; Whang and Garcia-Molina 2014]. Whang and Garcia-Molina [2014] consider the ER update problem where both the data and the rules used to resolve the data evolve and change over time. They investigated when and how previous ER results can be used with evolving data and rules to save redundant work and avoid performing the ER process from scratch. They proposed several incremental ER approaches and investigated how to use *materialized views* to improve the efficiency of

these approaches. Gruenheid et al. [2014] proposed another incremental ER approach that updates matching results when data updates (i.e., inserts, deletes, and modifications) occur. The technique uses an incremental graph clustering approach and allows new evidence from the updated data to fix previous matching errors. These techniques are aimed at dynamic data but not real-time ER.

To be able to conduct the ER process in real time we must consider completing each of its steps within the minimum time possible. Indexing is one of the most important steps in real-time ER as it reduces the search space, which leads to reducing the number of comparisons required in the comparison step. Moreover, when comparing records, we should aim to use efficient comparison functions and limit the required calculations to perform this step (refer to Section 5 for a suggestion on how to reduce such calculations). In the classification step, most classification models can be used regardless of which indexing techniques we use. However, for real-time ER, classification techniques that depend on the summation of comparison vectors to reach a classification decision can be more suitable as they are computationally efficient compared to more complex supervised and unsupervised techniques. In this article, we mainly focus on the indexing step. The following provides a review of relevant work on indexing, as well as the area of real time and ER.

Indexing techniques are employed in the area of database systems to improve the performance of search operations. Major indexing techniques that are used include techniques based on hash tables and tree data structures Hector Garcia-Molina and Ullman [2009]. The main hashing-based indexing techniques are extensible hashing [Fagin et al. 1979], linear hashing [Litwin 1980], and partial-match hashing [Rivest 1976]. The main tree-based indexing techniques include AVL trees [Adelson-Velskii and Landis 1962] (which are usually used with main memory indexing), B and B+ trees [Comer 1979; Cormen et al. 2009] (which are mostly used with disk-based indexing), and T trees [Lehman and Carey 1986] (which evolved from AVL trees and B trees and are commonly used with main memory indexing). More details about indexing techniques that are used with database systems can be found in Hector Garcia-Molina and Ullman [2009].

As for indexing techniques that are used in ER, standard blocking and the Sorted Neighborhood Method (SNM) are commonly used. Standard blocking [Christen 2012b] is based on inserting records into blocks according to *blocking key* criterion and only comparing records that are in the same block. The SNM [Hernandez and Stolfo 1995] arranges all records in the database(s) to be matched into a sorted array using a *sorting key* criterion. Then a window is moved over the sorted records, comparing only those records that are within the sliding window at any one time (explained in more detail in Section 3.1). Both blocking and sorting keys are usually based on one or a concatenation of attribute values.

Various other indexing techniques have been developed for ER, including q-gram indexing [Baxter et al. 2003], suffix array indexing [Aizawa and Oyama 2005], canopy clustering [McCallum et al. 2000], mapping-based indexing [Jin et al. 2003], and hashing-based indexing [Kim and Lee 2010]. However, all these techniques are aimed at offline batch processing of databases and are limited to indexing of static data. This means that once an index is created it is difficult to modify if new records need to be added, or when the values in existing records are changing.

Only limited research has so far concentrated on real-time ER, or on ER for dynamic databases. A first approach for real-time ER is based on a collective classification technique [Bhattacharya and Getoor 2007]. The idea behind this approach is to not use all records in a database to resolve a query, but rather to extract only those records that are related to a query, and to build a collective clustering structure using these records only. Although this approach achieves high matching quality, the authors stated that

the average time needed to resolve one query record was 31.28sec for a database that contained 831,991 records. Thus, this approach is not suitable for real-time ER, and it also is not scalable to large databases since it is computationally expensive.

A real-time ER approach that works on static databases was proposed by Dey et al. [2011]. This approach is based on using a matching tree to limit the amount of communication required for matching records between disparate databases, such that a match decision can be made without needing to compare all attribute values between records. This approach was shown to reduce the communication overhead, without affecting the matching quality. Ioannou et al. [2010] proposed an approach for RDF type data that provides ER in real time and also works on dynamic databases. Their method is based on using links between the entities in a probabilistic database to resolve entities. The approach uses existing ER techniques to find possible matches of a query. The approach stores the possible matches alongside the entities with a probability weight in a dynamic index. This stored information is then used at query time to perform ER in real time. The approach is reported to have an average time of 70msec for a query record on a database of 51,222 records. This reported query time is nearly constant and does not increase as the database gets larger.

To facilitate real-time ER, Christen et al. [2009] proposed a similarity-aware indexing technique where similarities between attribute values are precalculated during the building phase of the index. This approach is based on standard blocking and uses phonetic encodings to overcome errors and variations in attribute values to ensure that similar values are inserted into the same block. An average query time of 10msec per record on a large database of 7 million records was reported by the authors. However, this index was only applicable for static databases.

More recently, Ramadan et al. [2013] extended this similarity-aware index to work with dynamic data. In their proposed approach, new records can be inserted dynamically into the index allowing the index to grow. The authors stated that the average record insertion time (around 0.1msec), and the average query time (less than 10msec) were approximately constant as the index grew for a large database with 2.5 million records. While this dynamic similarity-aware indexing technique is based on the idea of standard blocking described before, in this article, we propose a novel dynamic real-time indexing approach based on the sorted neighborhood method, as we will describe next.

3. PRELIMINARIES

In this section, we summarize key aspects that are required to describe our approach. We use the following notation:

—*Dataset*: We assume that dataset $R = \{r_1, r_2, \ldots, r_{|R|}\}$ contains records of known entities. Each $r_i \in R$ has a unique record identifier $r_i.id$ and an entity identifier $r_i.eid$. All records in R are assumed to have the same attribute structure.

—*Query Stream*: We assume that a stream of query records $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$ is to be matched with R. Each $q_j \in Q$ is given a unique identifier $q_j.id \neq r_i.id, \ \forall r_i \in R$; and has the same attribute structure as records in R. It is assumed that q_j is to be added to R after it has been resolved.

—*Sorting Key*: A Sorting Key (SK) is defined as the list of attributes that are used to sort records in R alphabetically. Selecting SKs generally requires domain knowledge. SKs are usually generated by concatenating the attributes in the SK list. A Sorting Key Value (SKV) of a record in R is the value of the attributes used as a SK for that record. For example, assume that a record r has the following attribute values: $r =$ {Firstname = 'percy', Surname = 'smith', City = 'new york', Zipcode = '10007'} and assume that a concatenation of the 'Firstname' and 'Surname' attributes is used as a SK, then the value 'percysmith' would be the SKV generated for r.

RecID	Firstname	Surname	City	Zipcode
r1	percy	smith	new york	10007
r2	paul	smith	boston	02120
r3	robin	stevens	denver	80202
r4	pedro	smith	los angeles	90005
r5	abby	bond	new york	10001
r6	sally	taylor	los angeles	90002
r7	peter	smith	los angeles	90012
r8	sally	taylor	seattle	98168
r9	pedro	smith	bosten	02121
r10	peter	smith	los angelos	90002

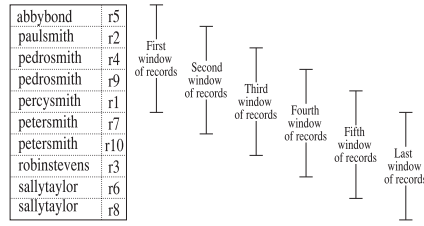


Fig. 1. The static sorted neighborhood method applied on the example table on the left with a fixed window size of $w = 5$ and sorting key values consisting of the concatenation of 'Firstname' and 'Surname' values.

The problem of real-time ER is defined as follows: for each query record q_j in a query stream Q, find all the records in R that belong to the same entity as q_j, denoted as the set M_{q_j}, in subsecond time, $M_{q_j} = \{r_i \mid r_i.eid = q_j.eid, r_i \in R\}$, $M_{q_j} \subseteq R, q_j \in Q$.

3.1. Sorted Neighborhood Indexing

The SNM is an indexing technique that was developed with the aim of reducing the number of comparisons between candidate records in the process of deduplicating large databases [Hernandez and Stolfo 1995, 1998]. The basic SNM consists of the following steps: First, if multiple databases are to be matched, they are merged and a unique identifier is assigned to each record. Then, a SKV is generated for every record in the merged database. Next, the records are sorted according to the generated SKVs. Finally, a comparison step that consists of a fixed-size window w (with $w > 1$) moves over the sorted records, and only the records within the sliding window at any time are compared with each other. Assuming the sorted database contains n records, the sorting step of the SNM has a complexity of $O(n\ log(n))$, while the comparison step is $O(w \times n)$ [Hernandez and Stolfo 1995]. Figure 1 shows an example of the basic SNM.

One of the main drawbacks of the SNM is its sensitivity to data quality of the attributes used as SKVs. Specifically, if a SKV has an error or variation at the beginning, then its record will potentially not be placed close to similar records, and therefore will likely be missed. For example, "christine" and "kristine" will not be close to each other in the sorted array if a "Firstname" attribute was used as a SK. A commonly used approach to overcome this drawback is to run the SNM several times using different SKs, followed by the calculation of the transitive closure of the identified matching records [Hernandez and Stolfo 1995, 1998].

Another major drawback of the basic SNM is the fixed setting of the window size w. If w is set too small, true matches are likely missed; on the other hand, if it is too large, unnecessary comparisons between records will be conducted. This problem has recently been addressed by two approaches that adaptively adjust the window size according to the characteristics of the SKVs or database records. One approach expands the window size if SKVs are similar with each others' according to a minimum similarity threshold [Yan et al. 2007], while an alternative approach expands a window if a certain minimum number of records are classified as matches within the current window [Draisbach et al. 2012].

The SNM in its current form (when using a fixed or an adaptive window size, or when using several runs of different sorting keys) is only suitable for indexing static databases and for batch-oriented ER. Due to the static nature of the sorted array it does not work for real-time ER applications where a stream of query records needs to be matched against a database consisting of entity records, and where these query records are commonly inserted into the database after matching. Our proposed technique, described in Section 4, provides an indexing technique that facilitates real-time ER,

and can handle dynamic databases. Next, we present the basic data structures we use in our indexing approach.

3.2. Index Data Structures

The original SNM uses a static array data structure to store the SKV of all records in the databases that are to be deduplicated or matched [Hernandez and Stolfo 1995]. However, a static array is not suitable for dynamic data because each time a new record is added to the index the existing elements in the array would need to be shifted to maintain the order, leading to a worst-case complexity of $O(n)$, where n is the number of records in a database. Real-time ER on dynamic databases requires an index data structure with efficient searching, inserting, and retrieving capabilities. Search trees are more efficient than sorted arrays [Cormen et al. 2009] and are commonly used for indexing in different application domains. In the following, we describe different search tree data structures.

—A basic binary search tree is a nonbalanced tree data structure that consists of nodes and edges to organize data in a hierarchical manner, where each node can have 0, 1, or 2 child nodes. Every node in the tree has a unique key value that is used for sorting the tree based on the following properties [Cormen et al. 2009]. Let x and y be nodes in the tree with different key values, that is, $x.key \neq y.key$ and y being a child node of x, x_L is the left subtree of x, and x_R is the right subtree of x: (1) $if\ y.key < x.key\ then\ y \in x_L$; (2) $if\ y.key > x.key\ then\ y \in x_R$. The shape and the height of a basic binary search tree depend on the sequence of the key values that are inserted into the tree. Because it is not a balanced tree, the worst-case time needed for operations such as insertion of new nodes, searching for certain key values, and retrieving the previous or next nodes, all have a complexity of $O(k)$, where k is the number of nodes in the tree.

—An AVL tree is a self height-balanced binary search tree where for each node in the tree the difference between the heights of its left and right subtrees never exceeds 1 [Rice 2007]. For a tree of size k nodes, the height of an AVL tree never exceeds $1.44log(k)$. This height is sufficient for providing $O(log(k))$ time for the operations described previously in the worst-case scenario [Sedgewick and Flajolet 2013].

—A braided AVL tree is a data structure that combines the properties of both AVL trees and double-linked lists [Rice 2007]. Each node in a braided AVL tree has a link to its predecessor and successor nodes according to an alphabetical sorting of the key values in the nodes. Because this data structure has the property of a double-linked list, accessing the next and previous nodes for a given node only requires $O(1)$ steps.

—A B tree is a balanced search tree data structure that is designed to work with secondary storage devices [Comer 1979; Cormen et al. 2009]. The difference between a B -tree and a binary tree is that nodes in the former can have more than two children. A B tree of order m reduces the depth of a binary tree of k nodes to $O(log_m k)$, where m is the number of allowed children for each node. This means that every node would have m children and $m-1$ keys. For example, a B tree of order $m = 10$ allows having 10^6 keys in six levels, while a binary tree needs 20 levels to accommodate the same number of keys. However, when searching for keys in a B tree, we have to search through the m keys in each node, which reduces the gain from having fewer levels of nodes in the tree. Also, retrieving the successor/predecessor keys in a B tree requires a complexity of $O(log(k))$ if the successor/predecessor keys were in the next or previous nodes and not in the same node of the query record. Similar to a braided AVL tree, a B tree can be extended with a double-linked list to improve retrieving the next and previous nodes, leading to a B+ tree [Comer 1979] which is a B tree that has a link between its leaf nodes. A B+ tree is to be investigated in future work.

Table I. Worst-Case Time Complexities for the Different Operations Achieved Using the Discussed Search Trees. k Is the Number of Nodes In a Tree

Operation	Binary	AVL	Braided AVL	B Tree
Search	$O(k)$	$O(log(k))$	$O(log(k))$	$O(log(k))$
Insert	$O(k)$	$O(log(k))$	$O(log(k))$	$O(log(k))$
Get successor node	$O(k)$	$O(log(k))$	$O(1)$	$O(log(k))$
Get predecessor node	$O(k)$	$O(log(k))$	$O(1)$	$O(log(k))$

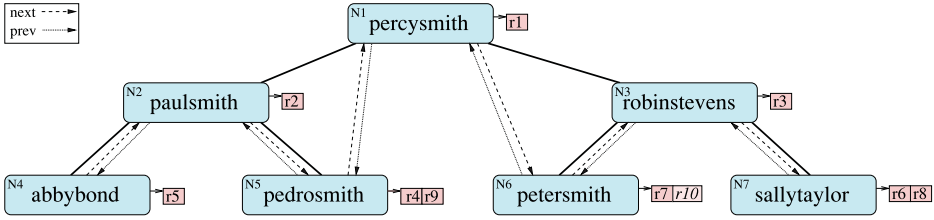


Fig. 2. DySNI for the 10 records from Figure 1 using a braided AVL tree data structure. The sorting key values are the concatenation of "Firstname" and "Surname" values.

Table I summarizes the complexities of the different operations achieved by the previously described search trees. As can be seen, the characteristics of the braided AVL tree make it highly suitable to be used with our proposed index (described in Section 4), since this data structure provides efficient retrieval times of neighboring nodes that are needed to generate the list of candidate records required to do the matching. In this article, we therefore use a braided AVL tree defined as follows:

Definition 1. BraidedTree (BRT) is a balanced binary AVL tree where each node in the tree has a link to its predecessor and successor nodes according to an alphabetical sorting of the key values in the nodes. We denote a node in the BRT as $N_i = (skv, I, prev, next)$ (with $1 \leq i \leq k$), where skv is a unique key value, I is the list that contains the record identifiers attached to that node, and $prev$ and $next$ are links to the predecessor and successor nodes, respectively. A node is denoted as N_q if a query record is inserted into the list I of that node. Figure 2 illustrates the BRT generated based on the small example dataset from Figure 1.

Because the key values in the trees are the SKVs of records, all records that have the same SKV will be appended to I in the same tree node (as shown in Figure 2). Assuming there are k different SKVs (nodes) in a tree (with $k \leq n$, and often $k \ll n$, where n is the number of records in the database to be indexed), this will reduce the number of steps when searching for SKVs from $O(log(n))$ to $O(log(k))$. In the next section, we discuss the four variations of our dynamic sorted neighborhood indexing approach in detail.

4. DYNAMIC SORTED NEIGHBORHOOD INDEXING

Our proposed Dynamic Sorted Neighborhood Index (DySNI) is based on a BRT as described previously. The aim of the DySNI is to dynamically index and resolve a stream of query records in real time. We assume we keep all records in the dataset R unmodified after they are created, since they can provide evidence about earlier queries on individual entities. An example application is applying for consumer credit where an individual's credit history needs to be retrieved and evaluated before a new loan can be approved. Replacing records with their cleaned and merged versions will likely result in a loss of accuracy, because details such as previous names or addresses of a customer are lost.

Algorithm 1: *DySNI – Query*

Input:
- Query record: q
- Similarity functions: $\mathbf{S}$
- Similarity threshold: θ
- Sorting key: $\mathbf{SK}$
- Database table with complete records: $\mathbf{D}$

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

1: $\mathbf{skv} = GenerateKey(\mathbf{SK}, q)$
2: $N_q = FindTreeNode(\mathbf{skv})$
3: $\mathbf{D}[q.id] = q$
4: **if** $N_q ==$ NULL **then**
5: $N_q = CreateNode(\mathbf{skv}, q.id)$
6: **else**
7: Append $q.id$ to $N_q.I$
8: $\mathbf{C} = GenerateWin(N_q, \mathrm{S}, \theta)$
9: $\mathbf{M} = CompareRecords(\mathbf{C}, \mathbf{S}, \mathbf{D}, q)$
10: Sort $\mathbf{M}$ according to similarities

Algorithm 2: DySNI
GenerateWin - Sim-based Adaptive approach

Input:
- Query node: N_q
- Similarity threshold: θ
- Similarity function: $\mathbf{S}$

Output:
- Candidate record set: $\mathbf{C}$

1: $\mathbf{C} = N_q.I$
2: $\mathbf{next\_nd} = N_q.next$
3: **while** $sim(N_q.skv, \mathbf{next\_nd}.skv) > \theta$ **do**
4: $\mathbf{C} = \mathbf{C} \cup \mathbf{next\_nd}.I$
5: $\mathbf{next\_nd} = \mathbf{next\_nd}.next$
6: $\mathbf{prev\_nd} = N_q.prev$
7: **while** $sim(N_q.skv, \mathbf{prev\_nd}.skv) > \theta$ **do**
8: $\mathbf{C} = \mathbf{C} \cup \mathbf{prev\_nd}.I$
9: $\mathbf{prev\_nd} = \mathbf{prev\_nd}.prev$

Fig. 3. Algorithm 1 describes the query phase in DySNI using the similarity-based adaptive window approach. Algorithm 2 describes the method *GenerateWin()* using a similarity-based adaptive window.

The DySNI has an initial *build phase* where a certain number (possibly none) of entity records from an existing database are inserted into the BRT. The built index is then used to generate candidate records to resolve query records during the *query phase*. The DySNI is dynamic since query records can be added into the BRT as they arrive.

Build Phase: In this phase, records are loaded from dataset R, their SKVs are generated, and they are inserted into the BRT. The SKVs become the key values *skv* used as tree nodes. If the SKV of an inserted record is new (i.e., has not been indexed earlier), a new node is created in the tree for this SKV, whereas if a SKV already exists in the tree as a node key, then the identifier of its record is added to the list I of this node. For example, node $N1$ in Figure 2 was generated when record $r1$ with SKV "percysmith" was inserted into the empty index, while for record $r8$ with SVK "sallytaylor" node $N7$ already exists in the BRT (the node was generated when record $r6$ was inserted), and so the identifier $r8$ can be directly added to the list I of $N7$.

After having indexed all records in R, the index is ready for resolving query records. The complete records in R with all attribute values are also indexed into an inverted index or disk-based database table $\mathbf{D}$, where the actual attribute values of records can be retrieved efficiently during the record comparison step, which is part of the query matching process.

Query Phase: In this phase (shown in Algorithm 1, Figure 3), a query record q is matched against the built index in real time. We assume that all query records are added to the DySNI. When a query record arrives, the first step is to generate the SKV for the record (line 1) and a new unique record identifier $q.id$ is assigned to it (in the *GenerateKey* function). This SKV and $q.id$ are then inserted into the BRT in the same way as records were inserted during the build phase (lines 3–7). The query record is also added into $\mathbf{D}$.

The window of neighboring nodes can now be generated (line 8). All record identifiers that are stored in the nodes within the window are added to the candidate record set $\mathbf{C}$. Whole records (for each record identifier within $\mathbf{C}$) are then retrieved from the inverted index or database table $\mathbf{D}$, and the attributes of q are compared with the retrieved records using similarity comparison functions [Christen 2012a] appropriate to the content of each attribute (line 9). The compared candidate records are returned in the list $\mathbf{M}$ sorted according to their overall similarities with the query record (line 10).

4.1. Generating the Window of Neighboring Nodes

To generate the window of neighboring nodes, we propose fixed and adaptive window size approaches. The aim of using an adaptive window size is to limit the number of comparisons between the query and candidate records to only those records that likely correspond to true matches. This issue was addressed for static SNM by Yan et al. [2007] where expanding the window is based on the similarities between SKVs, and by Draisbach et al. [2012] where expanding the window is based on the number of classified matches within the window. In the following, we describe one static and three adaptive window approaches.

4.1.1. Fixed Window Size (DySNI-f). The original SNM is based on using a fixed-size window w that corresponds to the number of candidate records that fall inside the window at any one time. As our DySNI approach is a tree-based index, and because all records that have the same SKV are inserted into one node, we set the window as the number of neighboring tree nodes in one direction (previous and next). With $w \geq 1$ the number of neighboring tree nodes in one direction of the query node N_q, the total number of neighboring nodes to be visited when generating the candidate records will be $2w$. A window of size $w = 0$ refers to the query node N_q only (the node where the query record was inserted).

For the index tree shown in Figure 2, assuming query record $r10$ has just been inserted into the tree in node $N6$ with key value 'petersmith,' and assuming a fixed window size $w = 1$ in each direction, the previous node $N1$ with key value 'percysmith' and the following (next) node $N3$ with key value 'robinstevens,' are included into the window. The final set of candidate records for query record $r10$ using this fixed window size approach is the set $\mathbf{C} = \{r1, r3, r7\}$. Note that $r7$ is also included as it is located in the same tree node as the query record ($N6$), and so it also needs to be compared with the query record.

As this example illustrates, a fixed-size window can lead to both unnecessary comparisons with records in nodes that are unlikely to have a high enough similarity to be matching with a given query record (like $r3$ from node $N3$), as well as missed potential true matches that are outside the window (such as the records attached to node $N5$ with key value 'pedrosmith').

4.1.2. Candidates-Based Adaptive Window (DySNI-c). This approach aims at matching a certain minimum number of candidate records that can be processed within a certain period of time. In a real-time environment this allows for a controlled number of candidate records to be returned for detailed comparisons. In practice, users can investigate different numbers of candidate records to achieve the required maximum query time. The minimum total number of candidate records to be returned, δ, is used to stop window expansion regardless of the similarities between SKVs.

The initial candidate record set $\mathbf{C}$ contains the records located in the query record's node. Then a decision on whether to expand the window on both sides or not is made based on the following criterion. If the count of records at the query record's node N_q is greater than or equal to the minimum candidate threshold $|\mathbf{C}| >= \delta$, then no expansion is needed, and only records located at the query node are included in $\mathbf{C}$. On the other hand, if $|\mathbf{C}| < \delta$, then the window expands on both sides of the initial node individually until $|\mathbf{C}| >= \delta$. The remaining number of records needed for the total candidate records to reach δ is calculated as $r = \delta - |\mathbf{C}|$. A new expansion threshold is set to $\lceil r/2 \rceil$ for each side of the query node, and the window on each side will continue expanding as long as the total number of candidate records from that side is smaller than or equal to $\lceil r/2 \rceil$.

The following example explains the approach. From Figure 2, assume we set the minimum candidate threshold $\delta = 6$. After inserting query record $r10$ into the index,

generating the window of candidate records begins from the query node $N6$. The number of records in $N6$ is $|\mathbf{C}| = 1$ ($\mathbf{C} = \{r7\}$). Because $|\mathbf{C}| <= \delta$, we calculate $r = 6 - 1 = 5$, and $\lceil 5/2 \rceil = 3$. We therefore need to add a minimum of three records from each side to the candidate list to exceed δ before stopping the expansion process. We add nodes $N1$ and $N5$ from the left side and $N3$ and $N7$ from the right side, resulting in $\mathbf{C} = \{r1, r3, r4, r6, r7, r8, r9\}$.

4.1.3. Similarity-Based Adaptive Window (DySNI-s). This approach is based on Yan et al. [2007], which uses the similarities between the SKVs in the index to adjust the boundaries of the window. In the original static approach, the window size w changes based on the similarities between SKVs, and the window slides over the static array starting from the first to the last record in the index. In our approach (shown in Algorithm 2, Figure 3), we adaptively expand a window on each side of the tree node of a query record separately in each direction based on the following steps. We initialize the window size in a direction as $w = 0$ (i.e., only include the query node N_q in the initial window). We expand the window in one direction based on the similarity between the query node's SKV and the SKV of the previous (or next) nodes using an approximate string comparison function. If this similarity is above a certain threshold θ, then we expand the window by adding the record identifiers in this node's list I and increase the window size w by 1. We repeat this process until the calculated similarity between SKVs falls below θ. This approach will only include tree nodes that are sufficiently similar to the records in the query record's tree node.

Based on Figure 2 and setting $\theta = 0.6$, after inserting query record $r10$ into the index, generating the window starts from the query node $N6$. To expand the window forwards (next), we compare the SKV of node $N6$ with the SKV "robinstevens" of its next neighbor $N3$ using the edit distance approximate string comparison function [Christen 2012a]. This gives us a low value of $sim(\text{'}petersmith\text{'},\text{'}robinstevens\text{'}) = 0.25$. Because $0.25 < \theta$ the window does not expand forward and the node $N3$ and its record identifier list will not be included into the set of candidate records $\mathbf{C}$. The same process will take place in the node's backwards (previous) direction. We get the SKV of the previous node $N1$ and compare it to the SKV of the query node, which leads to $sim(\text{'}petersmith\text{'}, \text{'}percysmith\text{'}) = 0.7$. Therefore, $N1$ and its record identifier $r1$ is added to $\mathbf{C}$. The comparison process continues in this direction until we reach a similarity that is less than θ. This occurs at node ($N4$) where $sim(\text{'}abbybond\text{'},\text{'}petersmith\text{'}) < \theta$. This means all records in the nodes $N5$ and $N2$ are included into $\mathbf{C}$. The final set of candidate records is $\mathbf{C} = \{r1, r2, r4, r7, r9\}$.

4.1.4. Duplicate-Based Adaptive Window (DySNI-d). This third adaptive approach is based on Draisbach et al. [2012]. The authors used an adaptive window size that grows or shrinks based on the number of classified matches that are found within the window. The window slides over the static array starting from the first to the last record in the index to match records in the whole database. In our approach (shown in Algorithm 3, Figure 5), we adaptively expand a window on each side of the query tree node based on the following steps.

When a query record arrives, and after it is inserted into the index data structure, a window of initial fixed size $w \geq 1$ is generated. Note this is different from the initial window size $w = 0$ for the similarity-based adaptive approach described before. A window size $w \geq 1$ is required because the duplicate-based approach needs to be able to compare candidate records to get a set of matching and nonmatching record pairs. The query record is compared (using a set of attribute-specific similarity functions) to all candidate records that are in the initial window. Those records that have a similarity above a certain threshold with the query record are classified as matches, and all others as nonmatches. Assume that the number of classified matches is m out of a total of c

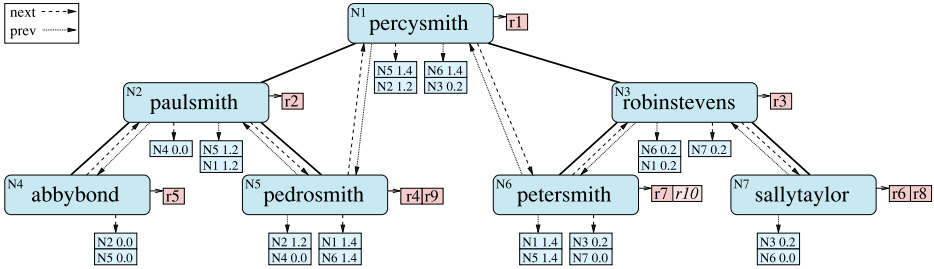


Fig. 4. SimDySNI for the 10 records from the table in Figure 1 using a braided AVL tree data structure. The sorting key values are the concatenation of 'Firstname' and 'Surname' values, and the window size for precalculation of similarities is set as $w = 2$.

candidate records compared with the query record, and assume that the expansion threshold (expansion ratio) is σ [Draisbach et al. 2012]. A window is expanded to the next tree node if the following holds:

$$\frac{m}{c} \geq \sigma. \tag{1}$$

In the same way as the similarity-based adaptive approach expands the window in each direction independently, the duplicate-based approach also calculates Equation (1) independently in the forward (next) and backward (prev) direction (as shown in Algorithm 3).

Let us use the example in Figure 2, and assume that the initial window size is $w = 2$, the expansion threshold $\sigma = 0.5$, and $r10$ is again the query record. With $w = 2$, the previous window will initially include two tree nodes ('percysmith' and 'pedrosmith') and the next window will also include two nodes ('robinstevens' and 'sallytaylor'). The query node has one candidate record $r7$ (which is not included in the expansion ratio calculation), the window into the previous direction has three candidate records $\{r1, r4, r9\}$, and the window into the next direction also has three candidate records $\{r3, r6, r8\}$. Therefore, $c = 3$ in both directions.

The decision on whether the previous window needs to be expanded or not depends on the number of matches found in the window based on Equation (1). Based on the full example records in Figure 1, assume that both $r4$ and $r7$ are matching records (and so $m = 2$). Because $2/3 > \sigma$, this means that the window will expand in the previous direction to include $N2$. The expansion process will continue until $m/c < \sigma$.

5. SIMILARITY-BASED DYNAMIC SORTED NEIGHBORHOOD INDEXING

The idea behind the Similarity-based Dynamic Sorted Neighborhood Index (SimDySNI) is to precalculate the similarities between the attribute values used to generate the SKVs, and to store these similarities in the tree. These precalculated similarities are used in the query phase to reduce the time required for the calculation of similarities between records. A similarity-based BRT is used to build the index where precalculated similarities are stored within nodes of the tree index.

Definition 2. Similarity-Based BRT (S-BRT) is a BRT tree with two extra lists attached to its nodes as illustrated in Figure 4. These lists contain the precalculated similarities for the neighboring nodes that are within the window (whether it is a fixed or an adaptive window). A node in the SimDySNI index is denoted as $N_i = (skv, I, prev, next, S_p, S_n)$, where S_p and S_n are the lists of the precalculated similarities between this node's SKV and the SKVs of all neighboring nodes within the

Algorithm 3: DySNI
Dup-based Adaptive window - GenerateWin

Input:
- Query record: q
- Query node: N_q
- Expansion threshold: σ

Output:
- Candidate record set: $\mathbf{C}$

1: $\mathbf{C} = N_q.I$
2: $\mathbf{next\_nd} = N_q.next$
3: $\mathbf{m\_next} = GetNumMatches(\mathbf{next\_nd}, q)$
4: $\mathbf{c\_next} = GetNumCandidates(\mathbf{next\_nd})$
5: **while** $\frac{m\_next}{c\_next} \geq \sigma$ **do**
6: $\quad\mathbf{C} = \mathbf{C} \cup \mathbf{next\_nd}.I$
7: $\quad\mathbf{next\_nd} = \mathbf{next\_nd}.next$
8: $\quad\mathbf{m\_next} = GetNumMatches(\mathbf{next\_nd}, q)$
9: $\quad\mathbf{c\_next} = GetNumCandidates(\mathbf{next\_nd})$
10: $\mathbf{prev\_nd} = N_q.prev$
11: $\mathbf{m\_prev} = GetNumMatches(\mathbf{prev\_nd}, q)$
12: $\mathbf{c\_prev} = GetNumCandidates(\mathbf{prev\_nd})$
13: **while** $\frac{\mathbf{m\_prev}}{\mathbf{c\_prev}} \geq \sigma$ **do**
14: $\quad\mathbf{C} = \mathbf{C} \cup \mathbf{prev\_nd}.I$
15: $\quad\mathbf{prev\_nd} = \mathbf{prev\_nd}.prev$
16: $\quad\mathbf{m\_prev} = GetNumMatches(\mathbf{prev\_nd}, q)$
17: $\quad\mathbf{c\_prev} = GetNumCandidates(\mathbf{prev\_nd})$

Algorithm 4: *SimDySNI – Query (fixed window)*

Input:
- Query record: q
- Similarity functions: $\mathbf{S}$
- Sorting key attributes: $\mathbf{SK}$
- Window size: w
- Database table with complete records: $\mathbf{D}$

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

1: $\mathbf{skv} = GenerateKey(\mathbf{SK}, q)$
2: $N_q = FindTreeNode(\mathbf{skv})$
3: $\mathbf{D}[q.id] = q$
4: **if** N_q == NULL **then** \\ *Case* 1 (new SKV)
5: $\quad N_q = CreateNode(\mathbf{skv}, q.id)$
6: $\quad\mathbf{C} = GenerateWin(N_q, w)$
7: $\quad\mathbf{M} = CompareRecords(\mathbf{C}, \mathbf{S}, \mathbf{D}, q)$
8: $\quad PreCalcNodeSimilarities(N_q, w, \mathbf{S})$
9: $\quad UpdateSimNextNodes(N_q, w, \mathbf{S})$
10: $\quad UpdateSimPreviousNodes(N_q, w, \mathbf{S})$
11: **else** \\ *Case* 2 (indexed SKV)
12: $\quad$Append $q.id$ to $N_q.I$
13: $\quad\mathbf{C} = N_q.I \cup N_q.Get\_I\_from\_nxt\_wind()$
$\qquad\cup\ N_q.Get\_I\_from\_prv\_wind()$
14: $\quad\mathbf{M} = ComparePreCalcRec(\mathbf{C}, \mathbf{S}, \mathbf{D}, q)$
15: Sort $\mathbf{M}$ according to similarities

Fig. 5. Algorithm 3 describes the function GenerateWin() using a duplicate-based adaptive window. Algorithm 4 describes the query phase for the SimDySNI.

window in the previous and following (next) directions, respectively, while the other node elements are the same as in Definition 1.

Build and Similarity Calculation Phases: The build phase is the same as the build phase of the DySNI described in Section 4. However, after the build phase a similarity calculation phase is conducted where the precalculated similarity lists S_p and S_n are added into the built DySNI tree index. Both lists are ordered according to the distance of the neighboring node from the query record's node (i.e., the first element in these lists is the closest neighboring node, and so on). The process of calculating similarities is conducted for all nodes in the tree. In this phase each node N_i in the tree is visited and the similarities between the attributes that are used to generate this node's SKV and the attribute values that are used to generate the SKV of the neighboring nodes within the window (in both the previous and next direction) are calculated using an approximate string similarity function.

The calculated similarities are stored into the lists S_p and S_n for each tree node. Figure 4 gives an example for SimDySNI when using a fixed window of size $w = 2$. The SimDySNI can be used with both fixed and similarity-based adaptive window approaches. For the adaptive window we continue calculating similarities with SKVs of neighboring nodes until the similarity threshold is reached (as described in Section 4.1.3), which means that in this case the precalculated lists can have different sizes for different nodes, while for a fixed size window approach we always have the same size lists for all nodes that is equal to w.

Query Phase: In the query phase of the SimDySNI approach we benefit from the precalculated similarities that are stored in each node to reduce the time needed to resolve a query. Querying the built index (from the build phase) is based on two cases (as shown in Algorithm 4, Figure 5): The first case occurs when the SKV of a query record q is new and it has not been indexed earlier. The second case occurs when the SKV of a query record q has been indexed previously and it already exists in the tree.

(1) *New SKV*: In this case (lines 4–10), because q is new, we create a new node N_q for this query and we resolve it using the original DySNI approach as described in

Section 4 (i.e., without benefiting from any precalculated similarities). After resolving the query, we generate the two precalculated similarity lists (i.e., S_p and S_n) for both directions for N_q by calculating the similarities for its w next and previous neighboring tree nodes (if we are using a fixed-size window) or until a similarity threshold is reached (if we are using a similarity-based adaptive window). Next, we update the similarity lists for all w previous and next tree nodes of the newly inserted tree node (lines 9 and 10). This step ensures that the precalculated similarities are up to date at any time.

(2) *Indexed SKV*: In this case (lines 11–15), because the SKV of the query record already exists in the S-BRT, there is no need to create a new node, and all required precalculated similarities are ready for use. In this case, a query can benefit from using these similarities as described in the next paragraph.

To generate candidate records, we retrieve (from the inverted index or database table **D**) all records that are stored in the tree nodes in the window. While the record comparison process in the DySNI compares all attribute values between the query and the candidate records to calculate an overall record similarity, in the SimDySNI we only need to compare attributes that are not used in the SK. To calculate the overall similarities between the query record and candidate records, we retrieve the precalculated similarities from the S_p and S_n lists, retrieve the corresponding records from **D** using the record identifier lists of these tree nodes, and then calculate the similarities of those attributes that are not used in the SK. Therefore, the more attributes are used in a SK the more similarities can be precalculated, but at the cost of a larger tree (as likely a larger number of distinct SKVs will be generated). In our experimental evaluation we investigate how different SK influence the amount of memory required to build the index, the percentage of query records that benefit from the precalculated similarities (i.e., queries with indexed SKVs), as well as the reduction in comparison time that can be achieved.

The following example describes the query phase on the small example set of records from Figure 1 and the index tree shown in Figure 4, and assuming that query record $r10$ has just been inserted into the tree in node $N6$. We assume $w = 2$. The candidate records for query record $r10$ are the records from the nodes stored in the next and previous list of the query node $N6$. These are the records from nodes $N1$ and $N5$ (previous), and $N3$ and $N7$ (next). The total set of candidate records for query $r10$ will therefore be $\{r1, r3, r4, r6, r7, r8, r9\}$. To compare query $r10$ with these candidate records, we first retrieve the actual records from the inverted index or database table **D** of record details, and for each candidate record we retrieve the precalculated similarity from the SimDySNI index. For example, the precalculated similarity between query record $r10$ and candidate record $r1$ is 1.4 as retrieved from the previous list of node $N6$. This similarity corresponds to the precalculated edit-distance similarities of the 'Firstname' and 'Surname' attributes (each is between 0 and 1). To get the total similarity between $r10$ and $r1$ we then calculate the similarities for the 'City' and 'Zipcode' on the actual attribute values of these two records. This overall similarity is then used to decide if a candidate record is a match or nonmatch.

6. MULTITREE DYNAMIC SORTED NEIGHBORHOOD INDEXING

Indexing techniques that are based on sorting records in the database using a SK have the drawback of being sensitive to errors that occur at the beginning of a SKV [Hernandez and Stolfo 1995]. To overcome this issue, we propose a multitree index based on the DySNI (M-DySNI). The index consists of multiple tree data structures where each tree is built using a different SK. In real-world data, attribute values are likely not completely independent of each other. However, we assume that SKs

are selected by domain experts to be as complementary to each other as possible (a technique for learning optimal sorting keys has recently been developed [Ramadan and Christen 2015]). Using several trees with different SKs can help improve the quality of results in cases where errors and variations occur at the beginning of attribute values. For example, 'christine' and 'kristine' will not be inserted into the same tree node if a 'Firstname' attribute was used as a SK, but they might be inserted into the same node in another tree where a different SK is used. The M-DySNI has two phases.

Build Phase: During the build phase, multiple trees are built using different SKs where a record is inserted into every tree in the index. Building one tree is similar to the build phase described in Section 4 where records are loaded from a database, and their SKVs are generated and inserted into the tree data structure. The steps for building one tree are repeated to build all trees using different SKs. For example, we can use a 'Firstname' attribute as a SK to build the first tree in the index, a 'Surname' attribute to build a second tree, a 'Postcode' attribute to build a third tree, and so on.

Query Phase: In this phase, a query record q is first inserted into the M-DySNI and then it is matched in real time against all trees that were constructed in the build phase. A new unique identifier is created for q and the different SKVs that are associated with the different trees in the index are created. q is then inserted into all trees using these SKVs and its record identifier is added in the same way records were inserted during the build phase (Section 4). The full attribute values of q are also added to **D**.

The process of retrieving candidate records from a single tree is similar to what is described in Section 4 using any of the different window approaches. However, candidate records are retrieved from every tree in the M-DySNI each adding candidate records into the overall candidate record set **C**, which becomes the union of candidates returned from the different trees. Then the query record q is compared with all unique records in **C** in detail using similarity comparison functions [Christen 2012a] in the same way as is done in the DySNI. The process of merging the returned sorted candidate records from each tree in the index has an overhead that is not present when only one tree is used.

7. ANALYSIS OF THE DYNAMIC SORTED NEIGHBORHOOD INDEX

In the ER process, the comparison step is usually the most time-consuming step because of the calculations performed when candidate records are compared. Estimating the number of comparisons beforehand gives users an insight about the expected runtime required to match a query record with a dataset of a certain size. In this section, we provide a way of estimating the number of generated candidate records using DySNI and M-DySNI.

The proposed DySNI approach groups records in the dataset with the same SKVs into one node (i.e., block). To obtain a better understanding of the number of candidate record pairs that will be generated for a certain query record, we assume two types of distributions that are common in attributes used for ER, namely, the uniform and Zipfian distributions.

As can be seen in Figure 6, the possible SKs that can be used to build the index (using the different attributes in the datasets that we are using in Section 8) either have a Zipfian-like distribution (like 'Firstname' and 'Surname'), or a distribution that is more similar to a uniform distribution (like the concatenation of 'Surname, 'Firstname,' and 'Zipcode'). Other SKs have a distribution that falls somewhere in-between these two distributions (like 'Zipcode,' and the concatenation of 'Surname' and 'Firstname'). For this reason, we will estimate the number of comparisons based on uniform and Zipfian distributions. Previous work that estimated the number of candidate record pairs for static indexing techniques also assumed these two distributions [Christen

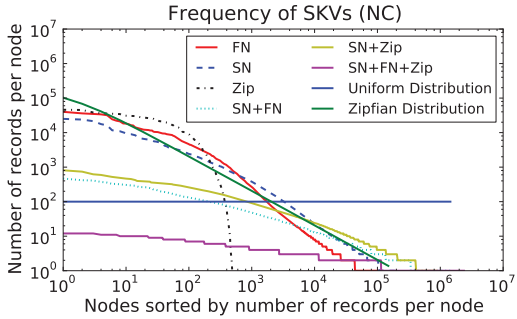


Fig. 6. Number of records in tree nodes generated using different SKVs for the NC datasets described in Section 8 (FN = Firstname, SN = Surname, Zip = Zipcode).

2012b]. These two distributions will allow us to provide lower and upper estimates of the number of candidate records that can be expected when matching real-world databases.

Assuming a uniform distribution for the frequency of attribute values will lead to a uniform distribution of the frequencies of the SKVs, which means all nodes in the index tree are of the same size. On the other hand, having a Zipfian distribution for the SKVs means that a few SKVs have a high probability of occurrence, while the majority of SKVs occur only rarely. According to the Zipfian law [Zipf 1949], for a list of values ranked according their frequencies, the frequency of any value is proportional to its rank in the ranked list of values and can be estimated as $1/r$, where r is the rank of a value.

For a uniform distribution all nodes in the tree data structure are assumed to have a uniform size of n/k where n is the number of records in the database and k is the number of SKVs in the tree. Assuming the fixed-size window approach, the number of candidate records generated in this case will be affected only by the number of nodes that are included in the generated window $2w + 1$ (for a fixed-window approach) and the number of records n in the database. Therefore, the estimated number of generated candidate records in $\mathbf{C}$ is

$$|\mathbf{C}| = \frac{n(2w+1)}{k}. \tag{2}$$

On the other hand, assuming a Zipfian frequency distribution of the SKVs will lead to a Zipfian distribution of the sizes of tree nodes in the index. In this case, the number of generated candidate records will not only be affected by the number of nodes that are included in the window and the number of records n in the database, but also by the size of the window's tree nodes (i.e., the number of records in each node). Assuming we rank the tree nodes N_i, $1 \leq i \leq k$, according to their sizes (number of records in a node's list I), the size of a node S_i is calculated as

$$S_i = \frac{1/i}{\sum_{i=1}^{k}(\frac{1}{i})} * n, \tag{3}$$

where the denominator is the Harmonic number of the partial harmonic sum [Christen 2012b]. The number of candidate record pairs S_w in a window that includes $2w+1$ nodes is then calculated as

$$S_w = \sum_{i-w}^{i+w} S_i. \tag{4}$$

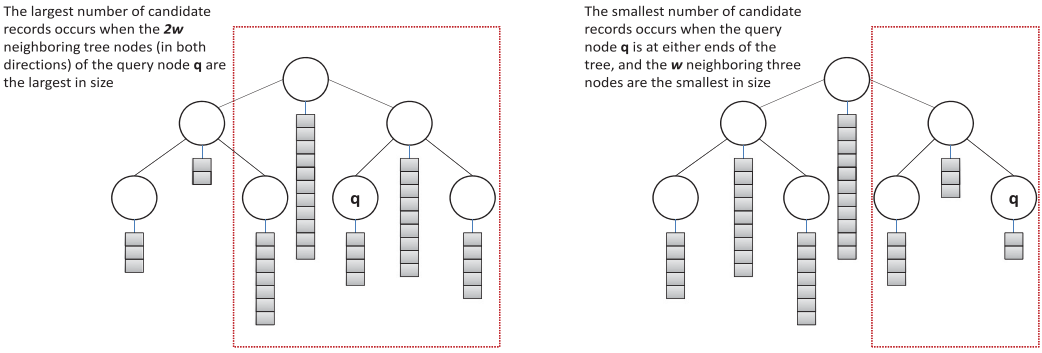


Fig. 7. Illustration of the situations where the maximum (left) and minimum (right) number of candidate records are obtained for the DySNI. The window size is set to $w = 2$.

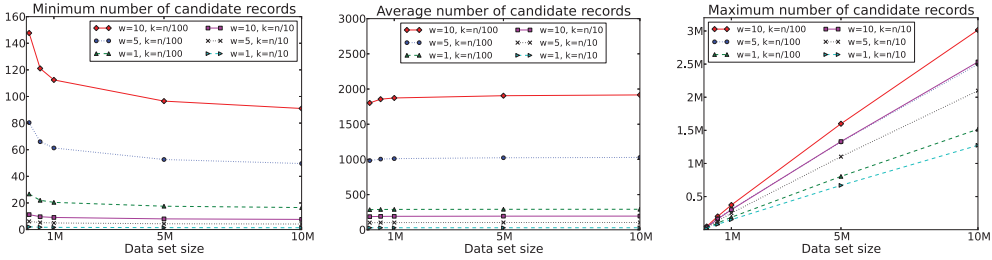


Fig. 8. The minimum, average, and maximum of number of candidate records estimated based on Equations (3) and (4), with n the number of records in a dataset and k the number of nodes in the tree index.

To estimate the minimum and maximum numbers of candidate records for a query q generated using a fixed-size window approach (assuming window size w) we calculate S_w according to the illustration given in Figure 7. The largest number occurs when the $2w$ tree nodes that are neighbors of the query tree node are the $2w$ largest tree nodes. On the other hand, the smallest number of candidate records are generated if the query node N_q is at either end of the tree, and the w neighboring tree nodes are the smallest in size.

Figure 8 illustrates estimates of the minimum, average, and maximum number of candidate records for increasing sizes of datasets based on Equations (3) and (4). From the figure, we can see that the maximum estimated number of candidate records increases linearly with the growing size of the dataset, while the minimum estimated number decreases with larger datasets, because the number of nodes k increases, which leads to smaller numbers of records in each node.

Importantly, the average number of generated candidate records for the different window sizes and number of tree nodes is constant with increasing dataset sizes. This indicates that on average the number of generated candidate records is not affected by the increasing size of the index, which confirms the experimental results in Figure 11 (discussed in Section 8.2.2) where the average query time is nearly constant with the growing size of the index.

The estimates in Equations (2), (3) and (4) also apply for M-DySNI since the difference between DySNI and M-DySNI is that the latter index has several distinct trees that are built using different SKs. This implies that the estimated maximum number of candidate records will be influenced by the number of trees in the index. The maximum

Table II. Datasets Summary

Database	Provenance	Records	Duplicates	Average number of duplicates per entity
NC	Real	7,997,234	146,331	4
OZ	Real (modified)	345,876	34,308	1
OZ-x	Real (modified)	345,876	172,938	5
Febrl-5	Synthetic	100,000	80,000	5
Febrl-10	Synthetic	100,000	90,000	10
Febrl-20	Synthetic	100,000	95,000	20

number of candidate records using multiple trees, assuming a uniform distribution, is calculated as

$$|\mathbf{C}| = \sum_{i=1}^{i=t} \frac{n(2w+1)}{k_{t_i}}, \tag{5}$$

where t is the number of trees in the M-DySNI data structure, and k_{t_i} is the number of nodes in a tree. As for the Zipfian distribution, the maximum number of candidate records using multiple trees can be calculated as

$$|\mathbf{C}| = \sum_{i=1}^{i=t} S_{w_{t_i}}, \tag{6}$$

where t is the number of trees and $S_{w_{t_i}}$ is the number of candidate records returned from a single tree (as calculated in Equation (4)). In practice, for a certain query record, duplicate candidate records will likely be returned from the different trees in the index, but only unique ones are compared in detail with the query record. Thus, the number of unique candidate records returned by all trees ranges between $S_{w_{t_i}}$ (which occurs when all the trees in the index return the same candidate records) and $\sum_{i=1}^{i=t} S_{w_{t_i}}$ (which occurs when each tree in the index returns unique candidate records). When estimating $|\mathbf{C}|$, we assume getting unique candidate records from each tree. The process of merging the lists of returned sorted candidate records (using a heap) from each tree in the index has an overhead of $O(t * S_{w_{t_l}} * log(t))$, where $S_{w_{t_l}}$ is the length of the longest list of the returned candidate record [Cormen et al. 2009].

8. EXPERIMENTAL EVALUATION

In this section, we describe the experiments conducted to evaluate our proposed approaches. We start by describing the various datasets we used and the experimental framework, followed by a discussion of the obtained results.

8.1. Experimental Setup

We implemented our DySNI, SimDySNI, and M-DySNI approaches using Python (version 2.7.3) and ran all experiments on a server with 128GBytes of main memory and two 6-core Intel Xeon CPUs running at 2.4GHz. To facilitate repeatability of our experiments, the prototype codes and the synthetic datasets are available from the authors.

8.1.1. Datasets. To evaluate different aspects of our proposed approaches we used both real as well as synthetic datasets. Table II summarizes these datasets.

—*NC Dataset*[1] is a large real voter registration dataset from the U.S. state of North Carolina. We have downloaded the NC dataset every 2 months since October 2011 to build a compound temporal dataset. This dataset contains the names, addresses, and ages of around 8 million voters, as well as their voter registration numbers (the used attributes are 'Firstname,' 'Surname,' 'City,' 'Zipcode'). Each record has a time stamp attached that corresponds to the date a voter originally registered, or when any of their details have changed. This dataset therefore contains realistic temporal information about a large number of people. We identified 142,673 individuals with two records, 3,566 with three, and 92 with four records in this dataset. This dataset is used for scalability evaluation since it has a large number of records.

—*OZ-x Datasets*: We generated four datasets with various corruption ratios using the GeCo data generator and corrupter [Tran et al. 2013] for the purpose of investigating the effect of having different levels of data quality in attribute values on matching quality. The four datasets each contain 345,876 records of personal details ('Firstname,' 'Surname,' 'Suburb,' 'Postcode') selected randomly from a clean Australian telephone directory, modified by adding duplicate records that had randomly corrupted attribute values based on typing, scanning, and OCR errors, or phonetic variations. "x" refers to the number of corrupted attributes in the dataset that we used (from OZ-1 to OZ-4). Each entity is represented on average by five duplicates. These datasets are used to evaluate the effect of using different thresholds for the different proposed window approaches, and to evaluate how different levels of noise (i.e., different data quality) in a dataset affect the performance of the proposed approach.

—*Febrl Datasets*: We generated three fully synthetic datasets where we specified the average number of records per entity (person) using the Febrl data generator [Christen and Pudjijono 2009]. This allowed us to evaluate our proposed approaches with regard to how datasets with different numbers of duplicates affect the DySNI adaptive window approaches. The three datasets each contain 100,000 records consisting of name and address attributes. In the first dataset (named Febrl-5) each entity is on average represented by five records (with a maximum of eight records per entity), in the second dataset (named Febrl-10) each entity is on average represented by 10 records (with a maximum of 15 records per entity), and in the third dataset (named Febrl-20) each entity is on average represented by 20 records, with a maximum of 30 records. Records were generated by first creating an "original" record for an entity, followed by the application of various modifications to generate "duplicate" records (by applying keyboard edits, phonetic and OCR modification, and setting values to missing). These sets are used to evaluate the effect on the proposed approach of having a different number of duplicates in a dataset.

8.1.2. Experimental Baseline. Two baseline approaches are compared with DySNI. The first is the dynamic similarity-aware index (DySimII) [Ramadan et al. 2013]. This approach is a dynamic indexing technique that is based on standard blocking. The DySimII has a build and a query phase. In the build phase, the similarities between attribute values are precalculated and stored in the index to reduce the required calculation in the query phase when a query record is matched. This method can be used for real-time ER with dynamic, large databases. The second baseline approach is a q-gram-based inverted index (QGI) [Baxter et al. 2003; Christen 2012b; McCallum et al. 2000] that converts the attribute values of each record in the database into a list of q grams. Each unique q gram becomes a key in the inverted index where its value is the list of all records in the database that have this q gram in their attribute values. To

[1]Available from `ftp://alt.ncsbe.gov/data/`.

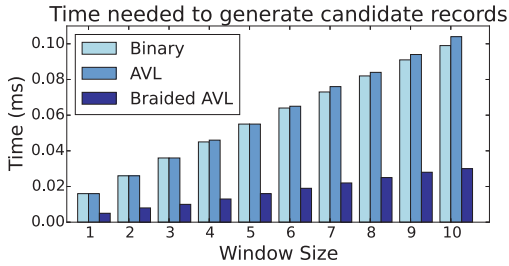


Fig. 9. The average time needed to generate candidate records for different types of trees. The plot is generated using the OZ-1 dataset.

match a query record with the q-gram inverted index, its attribute values are converted into a q-gram list, then it is compared only with records that have a certain number of common q grams that achieve a minimum similarity threshold. The approach returns a list of all records that have a Jaccard-based similarity with the query record that is greater than the minimum similarity threshold.

8.1.3. Evaluation Metrics. With real-time ER, the aim is to match a query record with all records in the dataset that represent the same entity as the query record in the least possible time (subsecond time). Thus, we focus on measuring the quality of obtained results, and the efficiency of the compared approaches. We use recall (the fraction of retrieved true matches over the total number of true matches) to measure the quality of the compared approaches, and both insertion time (time required to insert a single record into the index) and query time (time required to resolve a single query record) to measure efficiency.

Precision, which for a query record is the fraction of retrieved true matches over the number of retrieved candidate records, was not used in the experimental evaluation. This is because our approach returns the y top-matched candidate records when resolving a query record, which means that the number of retrieved candidates is always the same (y) for all resolved query records.

8.2. Experimental Results

To evaluate our approaches we conducted several sets of experiments as described next. Note that the SK used for conducting all of the following experiments is the concatenation of 'Surname+Firstname' attribute values.

8.2.1. Evaluating the Efficiency of Generating Candidate Records Using Different Tree Data Structures. We evaluated the efficiency of the three data structures described in Section 3.2 with respect to generating the candidate records by comparing the average time needed to generate a list of candidate records in the query phase using the fixed-size window approach for different window sizes ranging from $w = 1$ to $w = 10$. The results in Figure 9 show that the braided AVL tree is more than three times faster than the other two tree data structures for all evaluated window sizes. This highlights that the double-linked list in the braided AVL tree significantly reduces the time required to retrieve the candidate records for a given query record. All remaining presented results are based on using the braided AVL tree in the DySNI.

8.2.2. Evaluating the Estimation of the Number of Candidate Records. We conducted this set of experiments to evaluate the estimation function in Equation (4) from Section 7. We collected the minimum, maximum, and average number of candidate records used when running the DySNI-f approach on 30% of the NC dataset (with $n = 2,567,638$

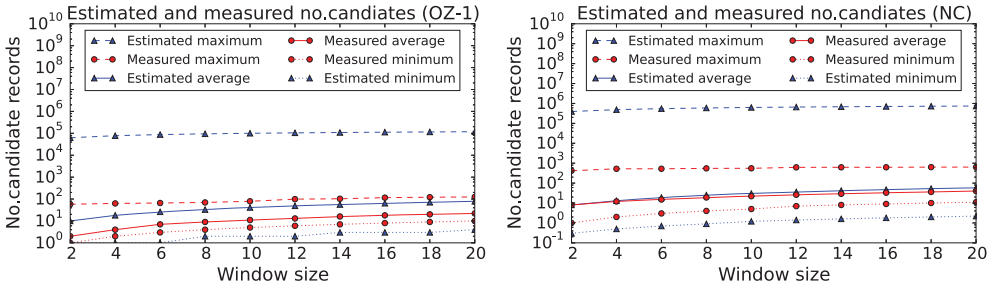


Fig. 10. The estimated number of candidate records for the OZ-1 and 30% of the NC datasets using Equation (4) from Section 7, compared with the measured number of candidate records required for running DySNI-f on both the NC and OZ-1 datasets using a concatenation of 'Surname' and 'Firstname' as SK for different window sizes.

Table III. Required Memory For Different Tree Types Using Various SKs for 30% of the NC Dataset

SK	Num nodes	New SKV (%)	Indexed SKV (%)	BRT (MB)	S-BRT (MB)
FN	120,632	5	95	74	98
SN	194,090	8	92	106	144
Zip	508	1	99	21	21
SN + FN	1,634,650	64	36	754	1,079
SN + Zip	897,215	35	65	420	598
SN + FN + Zip	2,237,069	87	13	1,130	1,614

and $k = 1,634,650$) and on the OZ-1 dataset (with $n = 345,876$ and $k = 160,058$) using different window sizes ($2 \leq w \leq 10$). Then we used the proposed estimation function to estimate the minimum, maximum, and average number of candidate records required using the same values of n and k of both the OZ-1 and NC datasets for window sizes ($2 \leq w \leq 10$) and plotted the results in Figure 10.

The results show that the measured and the estimated average number of candidate records are highly similar in both OZ-1 and NC datasets. The results also show that the measured minimum and maximum number of candidate records fall within the estimated minimum and maximum number of candidates. This indicates that Equation (4) can successfully estimate the number of candidate records for both OZ-1 and NC datasets.

8.2.3. The Effect of Using Different SKs on Index Size. We conducted an experiment on how using different SKs in building the index will affect the tree size (number of nodes k in the tree), and the number of records inserted into tree nodes for the NC dataset. Figure 6 and Table III show that using single attribute values as SKs results in trees with fewer nodes (with more records inserted into the nodes), while using several attributes to generate concatenated SKs results in larger trees with a smaller number of records being inserted into each node. This experiment confirmed that using various SKs leads to $k < n$ and often $k \ll n$, especially for large datasets.

8.2.4. Scalability Experiments. In this set of experiments we evaluated whether the proposed DySNI scales to large databases while facilitating real-time ER. We measured the average time required to insert a single record, and the average query time required to resolve a single query record across the growing size of the index structure. These experiments were conducted on 2.5 million records from the NC datasets. The proposed approach with a fixed-size window (DySNI-f), a candidate-based adaptive window (DySNI-c), a similarity-based adaptive window (DySNI-s), and a

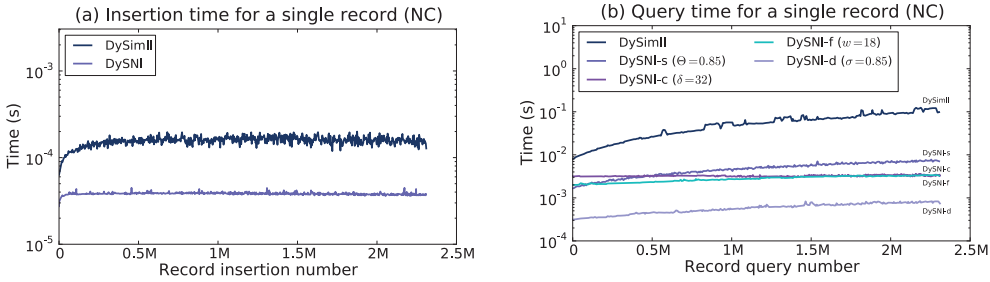


Fig. 11. Plot (a) shows the average time needed for inserting a single record into the index. Plot (b) illustrates the average time required for querying the growing index. A subset of 30% of the NC dataset is used (M = Million). All compared approaches give similar recall values.

duplicate-based adaptive window (DySNI-d) was compared with the DySimII baseline technique. Threshold selection for the various proposed approaches was based on achieving the same recall value for all compared approaches.

As can be seen from Figure 11, the DySNI approaches significantly outperform the earlier DySimII by up to one order of magnitude faster insertion time. As for the query time, the results show that the various proposed approaches have between one to two orders of magnitude faster query time than the DySimII approach while achieving similar recall values. Moreover, the results show that the average insertion times are not affected by the growing size of the index data structure, while the query time only increases slightly as the index becomes larger. As expected, larger window sizes lead to slower query times, and the DySNI-s approach is slower than the DySNI-d and DySNI-c adaptive approaches. This is due to the fact that the calculation of similarities between SKVs is an overhead of the DySNI-s approach that does not occur with the other two adaptive approaches. Additionally, the different proposed approaches generate different sets of candidate records that lead to different query times. However, the results show that the different variations of the proposed DySNI approach achieve very fast average query times that range between 0.02 and 3.0msec per query record.

8.2.5. The Effect of Using Different Thresholds on Quality and Efficiency. In this set of experiments, we investigated the effect of using different window sizes and thresholds on the quality of the obtained results and the efficiency of the approaches. We compare the various proposed approaches and the DySimII and QGI baselines. The OZ dataset (with only one duplicate per record) was used for this set of experiments.

—Fixed size window (DySNI-f): We investigated using different window sizes for this approach by measuring recall and average query times needed to resolve query records. As shown in Figure 12(a), recall values are improving with an increase in window size since more records are compared with a query record. Because the number of comparisons increases for larger window sizes, the time needed to resolve queries will also increase. This means, as one would expect, that larger window sizes can achieve better recall values but at the cost of increasing query time. Comparing DySNI-f to DySimII, when $w = 10$, to achieve similar recall value, DySNI-f is one order of magnitude faster.

—Candidate-based adaptive window (DySNI-c): We investigated using a different minimum number of candidate records for the candidate-based adaptive window approach; the results in Figure 12(b) show that the more candidate records we have in the window the better recall values we get but with an increase in the time needed

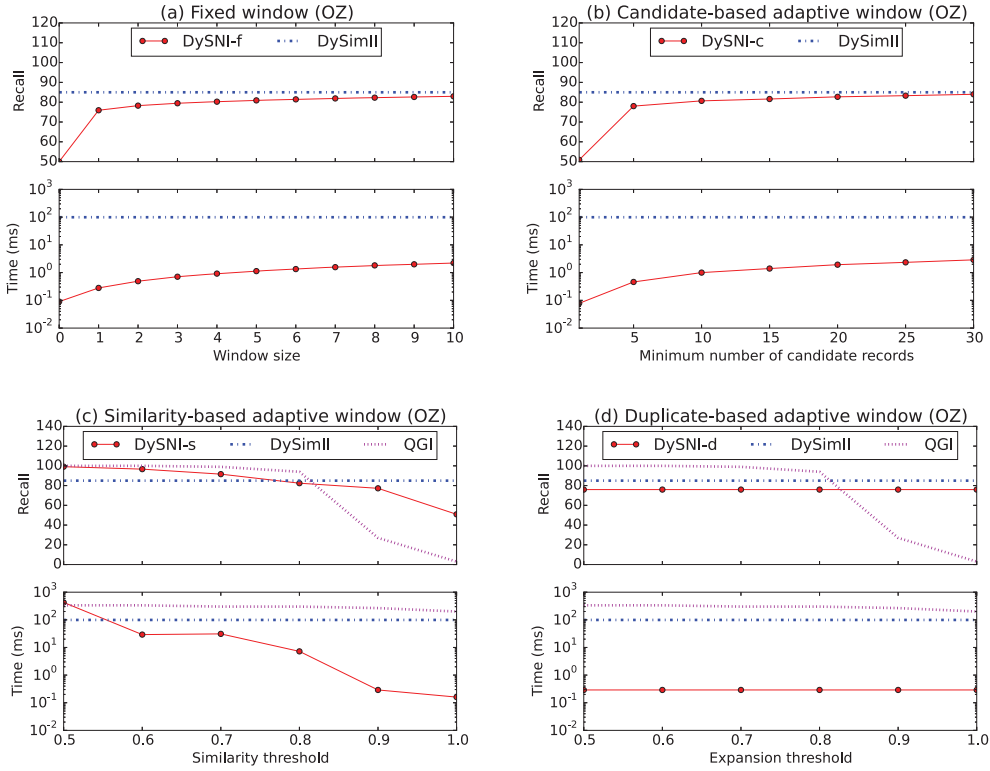


Fig. 12. Recall and time measures for the different DySNI approaches compared to the baselines DySimII and QGI. All plots are generated using the OZ dataset. Each entity can have one duplicate.

to resolve queries. DySNI-c achieves almost one magnitude faster query times than DySimII for parameter settings where they both achieve similar recall values.

—Similarity-based adaptive window (DySNI-s): We investigated using different similarity thresholds to expand the window on both sides for this adaptive window approach. Figure 12(c) shows that smaller threshold values achieve better recall values but more time is needed to resolve queries. This is because smaller similarity thresholds mean that more records are included in the window, which leads to better recall but larger query times. Compared with DySimII and GQI, when all techniques achieve the same recall values of 100%, DySNI-s achieves similar query time to QGI, but slower than DySimII. However, query time for QGI is almost constant for all thresholds where for DySNI-s query time decreases when the similarity threshold increases.

—Duplicate-based adaptive window (DySNI-d): We investigated using different expansion thresholds for the duplicate-based adaptive window approach. Figure 12(d) shows that both recall and average query times were almost constant with different expansion thresholds. This means that the expansion of the window was very limited. The reason for that is the way that DySNI-d is structured. Because each tree node contains all records with the same SKV, we can see that most of the duplicates are already found in the query record's node or its nearest neighboring nodes, which limits the expansion process for the duplicate-based approach to a very small number of neighboring tree nodes. Therefore, the duplicate-based approach

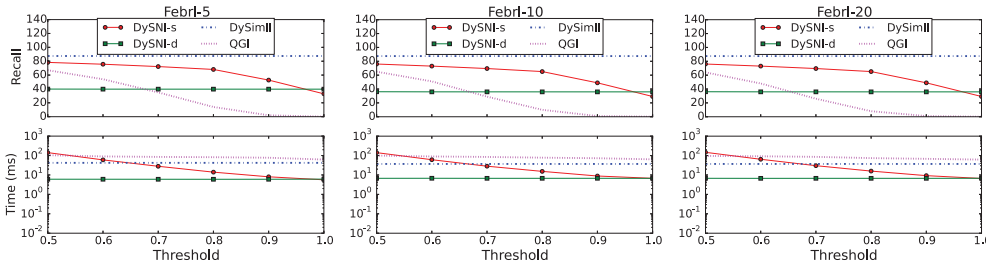


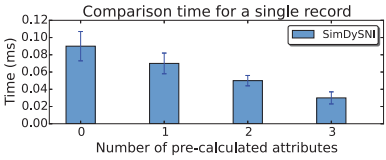
Fig. 13. Recall and time measures for similarity and duplicate adaptive approaches using Febrl datasets. Note that the threshold is different between all approaches; for DySNI-s it represents the similarity threshold between the SKV of the query's node and neighboring nodes, while for DySNI-d it represents the ratio between the found true matches and the number of candidate records within the window. For QGI it is the minimum Jaccard similarity between a query record and a candidate record to be considered as a match.

is not suitable for DySNI. Both baselines performed better than DySNI-d in regard to matching quality, but DySNI-d achieved lower query time. Note that the recall values for QGI increases when the Jaccard similarity threshold decreases. However, it achieves almost constant query time for all different thresholds.

8.2.6. The Effect of Having Different Number of Duplicates. The aim of this set of experiments was to investigate the effect of the number of duplicate records on recall, query times, and on the expansion of the adaptive window in both DySNI-s and DySNI-d approaches. These experiments are conducted using the Febrl datasets. From the results shown in Figure 13, we can see DySNI-s outperformed QGI with regards to both recall and query times. QGI did not perform well for Febrl datasets. We can also see that DySimII gives better recall values than both DySNI-s and DySNI-d and that, in general, the recall values achieved by the two adaptive approaches are less than the recall values achieved with the OZ dataset in the previous set of experiments. This is due to the fact that in the OZ dataset the maximum number of duplicates that a record can have is one, while in the Febrl datasets the average number of duplicates ranges between 5 and 20. Having a larger number of modified duplicates in the datasets increases the chance of having an error in the first character of the attribute value that is used as a SKV. Additionally, because the proposed approach is based on sorting records alphabetically according to SKVs, this increases the chance of having records located far away from other records that represent the same entity in the index data structure. This issue can be resolved by building multiple trees using different SKVs (Section 6) to increase the chance of having records that represent the same entities close to each other. The results of using multiple trees are shown in Figure 15.

In addition, the results show that having a different number of duplicates in the Febrl datasets does not affect window expansion for DySNI-d, and similar to the results from the OZ dataset from the previous set of experiments, DySNI-d still has very limited expansion in the adaptive window. Recall and query time values are also almost constant, which confirms the findings from the previous set of experiments where most of the found duplicates are located in the query record's node or its nearest neighboring nodes, which limits the expansion process for the DySNI-d approach to a very small number of neighboring tree nodes.

8.2.7. The Effect of Precalculating Similarities on Comparison Time. Here we investigate how the SimDySNI is able to improve query time. First, we measure the average time needed to compare a query record with a single candidate record for queries where a SKV has been indexed previously and already exists in the index, using the SimDySNI



Overall average query time for queries with indexed SKVs (ms)			
SK	DySNI	SimDySNI	Improvement
FN	108	60	44 %
SN	31	11	64 %
SN+FN	11	0.9	93 %
SN+FN+Post	10	0.8	92 %

Fig. 14. The plot on the left presents average times required to compare a query record with a single candidate record for the SimDySNI approach using different numbers of attributes as sorting keys. The table on the right shows the average total query time (in ms) for queries with indexed SKVs (Section 5) using various SKs. The OZ-1 dataset was used for both the table and the plot. Note that the case of having no pre-calculated attributes is the same as the original DySNI.

approach (as shown in Algorithm 4) with a different number of attributes being used as the SK. We ran the experiments using the OZ dataset with 1, 2, and 3 attributes used as SKs for different possible combinations of the four attributes: 'Firstname,' 'Surname,' 'Suburb,' and 'Postcode.' The average comparison times over these combinations are shown in the left plot in Figure 14. The results show that for queries where the node is preexisting, SimDySNI can significantly reduce the time required to compare a query record with a single candidate record. This improvement in time is almost linear with the number of attributes used in a sorting key. The results show that for a one-attribute sorting key the comparison time reduction is around 20%, for a two-attribute sorting key it is around 40%, and for a three-attributes sorting key it can be up to 70%.

Second, we measure the average total query time for queries with indexed SKVs (where the SKV of the query record already exists in the index) for both DySNI and SimDySNI approaches. The table in Figure 14 presents the average overall query time required for queries with indexed SKVs using various SKVs for of the OZ dataset. The results show that using SimDySNI for queries with indexed SKVs the overall average query time has improved by 44% to 93% for the various SKVs. Table III provides the percentages of both queries with new SKV and queries with indexed SKV when using various SKs for the NC dataset. Results show that between 13% and 99% of arriving queries can benefit from query time improvement by SimDySNI for queries with indexed SKV for the different SKs.

8.2.8. Required Memory Size. Table III shows the memory requirements of the different tree index data structures using different sorting keys. As can be seen, with concatenated SKs the number of unique SKVs increases significantly and therefore the size of the tree index structure also grows. The additional overhead of the SimDySNI compared to the total amount of memory required by the tree structure is negligible for small trees, but can be quite significant for large trees.

8.2.9. The Effect of Using Multiple Trees and Different SK Combinations on Recall and Query Time. In this set of experiments, we evaluate the effect of using multiple trees and different SK combinations (using all possible single attributes and concatenated pairs of attributes) on recall and average query time. DySNI-s with a similarity threshold between 0.5 and 1.0 was used on the OZ-1 dataset for this set of experiments. A single record in this dataset has an average of five corrupted duplicates. From Figure 15 we can see that using more trees increases recall at the cost of a slight increase in the average query time. We can also see that although SKVs generated from single-attribute values give better recall values, they require a longer time to resolve queries, which means that they are not suitable for real-time ER. On the other hand, SKVs that are generated from a concatenation of two attribute values reduce the average query time significantly but still achieve high recall values. The figure also shows

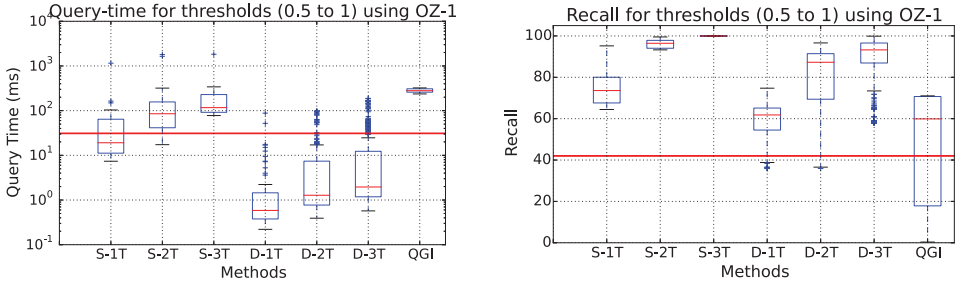


Fig. 15. Results for running M-DySNI on the OZ-1 dataset using different numbers of trees. "S" refers to using all single attributes as SK. "D" refers to using all SKs generated from the concatenation of two attributes. "xT" refers to the number of trees. DySNI-s is used with similarity thresholds from 0.5 to 1.0.

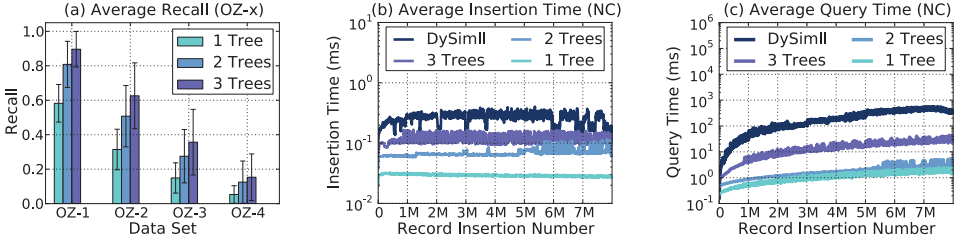


Fig. 16. Plot (a) shows the average recall values for the OZ-x datasets with different corruption rates using all possible SKs generated from the concatenation of two attributes. Plots (b) and (c) show the average insertion and query times for the full NC dataset compared with the baseline. The M-DySNI is used to generate the plots.

that when using three trees recall can be increased significantly compared to when using a single tree. Also, using multiple trees allows using more strict SKs (like the concatenation of more than one attribute) to build multiple trees with smaller node sizes while maintaining matching quality. We obtained similar results with the other OZ-x datasets where two, three, or four attribute values have been corrupted (see Figure 16(a)).

M-DySNI achieved better recall and query time results compared to both baseline approaches. Recall values for DySimII and QGI have dropped compared to results in Figure 12 because the dataset used in this set of experiments has a larger number of corrupted duplicates. We also noted from running this set of experiments that for M-DySNI using attributes that have less dependency between them (e.g., 'Firstname,' 'Postcode') as SKs gave better recall results than attributes with higher dependency (e.g., 'Suburb' and 'Postcode').

8.2.10. Scalability of the Multiple-Tree Index. In this set of experiments we evaluate the scalability of the M-DySNI on the full NC dataset using different numbers of trees. The results illustrated in Figure 16(b) show that the average insertion times using the various numbers of trees is not affected by the growing size of the index data structure, while plot (c) shows that the average query time only increases slightly as the index becomes larger. As expected, the results show that using more trees increases the average insertion and query times, but the achieved times are still very fast (around 1 ms and 15 ms insertion and query time, respectively) for three trees. The memory required for the index of one, two, and three trees for the full NC dataset was 1.8, 3.6, and 4.3 Gigabytes, respectively.

8.3. General Discussion

The experimental results described previously illustrate the effectiveness of DySNI. The fast insertion and query times achieved by the approach, and the ability to facilitate querying of large and dynamic datasets, make it effective for real-time ER. Moreover, SimDySNI improves query time by storing pre-calculated similarities between SKVs of neighboring nodes in the index, but at the cost of extra memory requirement.

The similarity-based (DySNI-s) approach showed better recall results since the expansion decision in this approach depends on the similarities between SKVs. Because these SKVs are sorted, neighboring nodes are likely to have similar SKVs. The duplicate-based (DySNI-d) approach does not work effectively because in DySNI all records with the same SKV are inserted into the same tree node, which limits window expansion and reduces the quality of the achieved results. Candidate-based adaptive windows (DySNI-c), on the other hand, can be used to control and limit the number of comparisons to achieve lower query times by choosing a low minimum number of candidates threshold.

Among the various proposed window types, DySNI-s has shown better recall values at the cost of increased query time, which makes it suitable for applications that need high-quality query matching results. As for DySNI-f and DySNI-c, they both can be used with applications that require a controlled time for resolving queries.

Moreover, our results illustrated that using the similarity-based SimDySNI reduces the average comparison time between 20% and 70% (based on the number of attributes used to generate SKV) while it increases the memory footprint between 13% and 40% for various SKs. Finally, the drawback of sensitivity to errors and variations at the beginning of SKVs was addressed by proposing a multitree indexing approach that improved the matching quality while maintaining efficiency. Our results also show that SKs that are based on a concatenation of more than one attribute value are more suitable for real-time ER since they reduce query time significantly while still achieving high matching accuracy.

All results confirm that the DySNI is well suited for use with real-time ER where a stream of query records needs to be resolved against a large and dynamic database.

9. CONCLUSIONS AND FUTURE WORK

We have presented a dynamic tree-based sorted neighborhood indexing technique that can be used for real-time ER on large databases. The technique was shown to be scalable with large databases as it has fast insertion and query times. We improved query times using a variation where we precalculate the similarities between the attribute values that are used to generate the sorting key values. We investigated several query matching approaches using both a fixed-size window and various adaptive window techniques. We showed that both the fixed-size window and the candidate-based adaptive window approaches provide more control over the time used to resolve queries. We also showed that the similarity-based adaptive window approach achieves better matching quality at the cost of requiring more time to resolve queries. Any sorted indexing technique has the drawback of being sensitive to errors at the beginning of the sorting keys. We addressed this issue by proposing an index with multiple dynamic trees where each tree uses a different sorting key. We evaluated the proposed techniques using a large real-world and two synthetic datasets. This evaluation showed that the proposed DySNI is suitable for real-time ER.

For future work we plan to extend the proposed index using a combination of a B+ tree and BRT to work with disk-based memory to allow indexing of very large datasets that do not fit into main memory. We also plan to parallelize the multiple-tree index to improve performance. Moreover, we plan to explore how DySNI can be integrated

with classification and clustering techniques [Hassanzadeh et al. 2009] to make the complete ER pipeline applicable for real-time matching.

REFERENCES

Georgy Maximovic Adelson-Velskii and Evgenii Mikhailovich Landis. 1962. An information organization algorithm. In *Doklady Akademia Nauk SSSR*, Vol. 146. 263–266.

Akiko Aizawa and Keizo Oyama. 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)*. IEEE, 30–39.

Rohan Baxter, Peter Christen, and Tim Churches. 2003. A comparison of fast blocking methods for record linkage. In *SIGKDD Workshops*. ACM, 25–27.

Indrajit Bhattacharya and Lise Getoor. 2007. Query-time entity resolution. *Journal of Artificial Intelligence Research* 30 (2007), 621–657.

Peter Christen. 2012a. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer Verlag, New York.

Peter Christen. 2012b. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Transactions on Knowledge and Data Engineering* 24, 9 (2012), 1537–1555.

Peter Christen, Ross Gayler, and David Hawking. 2009. Similarity-aware indexing for real-time entity resolution. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM)*. ACM, 1565–1568.

Peter Christen and Agus Pudjijono. 2009. Accurate synthetic generation of realistic personal information. In *Proceedings of the 13th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD'09), LNAI*, Vol. 5476. Springer, 507–514.

Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2009. *Introduction to Algorithms* (3rd ed.). McGraw-Hill Higher Education.

Debabrata Dey, Vijay S. Mookerjee, and Dengpan Liu. 2011. Efficient techniques for online record linkage. *IEEE Transactions on Knowledge and Data Engineering* 23, 3 (2011), 373–387.

Xin Luna Dong and Divesh Srivastava. 2013. Big data integration. In *2013 IEEE 29th International Conference on Data Engineering (ICDE'13)*. IEEE, 1245–1248.

Uwe Draisbach, Felix Naumann, Sascha Szott, and Oliver Wonneberg. 2012. Adaptive windows for duplicate detection. In *2012 IEEE 28th International Conference on Data Engineering (ICDE'12)*. IEEE, 1073–1083.

Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering* 19, 1 (2007), 1–16.

Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. 1979. Extendible hashing—A fast access method for dynamic files. *ACM Transactions on Database Systems (TODS)* 4, 3 (1979), 315–344.

Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99)*. Morgan Kaufmann, 518–529.

Anja Gruenheid, Xin Luna Dong, and Divesh Srivastava. 2014. Incremental record linkage. *Proceedings of the VLDB Endowment* 7, 9 (May 2014), 697–708.

Oktie Hassanzadeh, Fei Chiang, Hyun Chul Lee, and Renée J. Miller. 2009. Framework for evaluating clustering algorithms in duplicate detection. *Proceedings of the VLDB Endowment* 2, 1 (Aug. 2009), 1282–1293.

Jennifer Widom Hector Garcia-Molina and Jeffrey D. Ullman. 2009. *Database Systems: The Complete Book* (2nd ed.). Pearson Prentice Hall.

Mauricio A. Hernandez and Salvatore J. Stolfo. 1995. The merge/purge problem for large databases. In *ACM SIGMOD*. ACM, 127–138.

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

Ekaterini Ioannou, Wolfgang Nejdl, Claudia Niederée, and Yannis Velegrakis. 2010. On-the-fly entity-aware query processing in the presence of linkage. *Proceedings of the VLDB Endowment* 3, 1–2 (Sept. 2010), 429–438.

Liang Jin, Chen Li, and Sharad Mehrotra. 2003. Efficient record linkage in large data sets. In *Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA'03)*. IEEE, 137–146.

Hung-Sik Kim and Dongwon Lee. 2010. HARRA: Fast iterative hashed record linkage for large-scale data collections. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT'10)*. ACM, 525–536.

Tobin J. Lehman and Michael J. Carey. 1986. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB'86)*. Kyoto.

Witold Litwin. 1980. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB'80)*, Vol. 6. 1–3.

Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. 2000. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD'00)*. ACM, 169–178.

Felix Naumann and Melanie Herschel. 2010. An introduction to duplicate detection. *Synthesis Lectures on Data Management* 2, 1 (2010), 1–87.

Clifton Phua, Kate Smith-Miles, Vincent Lee, and Ross Gayler. 2012. Resilient identity crime detection. *IEEE Transactions on Knowledge and Data Engineering* 24, 3 (2012).

Banda Ramadan and Peter Christen. 2014. Forest-based dynamic sorted neighborhood indexing for real-time entity resolution. In *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management (CIKM'14)*. Shanghai, 1787–1790.

Banda Ramadan and Peter Christen. 2015. Unsupervised blocking key selection for real-time entity resolution. In *Proceedings of Advances in Knowledge Discovery and Data Mining, LNCS*. Springer.

Banda Ramadan, Peter Christen, Huizhi Liang, Ross Gayler, and David Hawking. 2013. Dynamic similarity-aware inverted indexing for real-time entity resolution. In *PAKDD Workshops*, LCNS, Vol. 7868. Springer, Gold Coast, Australia, 47–58.

Stephen V. Rice. 2007. Braided AVL trees for efficient event sets and ranked sets in the SIMSCRIPT III simulation programming language. In *Western Multiconference on Computer Simulation*. Woodhead Publishing, San Diego, 150–155.

Ronald L. Rivest. 1976. Partial-match retrieval algorithms. *SIAM J. Comput.* 5, 1 (1976), 19–50.

Robert Sedgewick and Philippe Flajolet. 2013. *An Introduction to the Analysis of Algorithms* (2nd ed.). Addison-Wesley.

Khoi-Nguyen Tran, Dinusha Vatsalan, and Peter Christen. 2013. GeCo: An online personal data generator and corruptor. In *Proceedings of the 22nd International Conference on Information and Knowledge Management (CIKM'13)*. 2473–2476.

Steven Euijong Whang and Hector Garcia-Molina. 2014. Incremental entity resolution on rules and data. *The International Journal on Very Large Data Bases* 23, 1 (Feb. 2014), 77–102.

William E. Winkler. 2006. *Overview of Record Linkage and Current Research Directions*. Technical Report RR2006/02. US Bureau of the Census, Washington, DC.

Su Yan, Dongwon Lee, Min-Yen Kan, and Lee C. Giles. 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, 185–194.

George Kingsley Zipf. 1949. Human behavior and the principle of least effort. Addison-Wesley Press.

Received July 2014; revised May 2015; accepted August 2015