Indexing Techniques for Real-Time Entity Resolution

Banda Ramadan

A thesis submitted for the degree of
Doctor of Philosophy in Computer Science
The Australian National University

March 2016

© Banda Ramadan 2016

Except where otherwise indicated, this thesis is my own original work.

Banda Ramadan
31 March 2016

To my beloved family

Acknowledgments

I grant all my gratitude to Almighty Allah, for guiding me and giving me the strength to finish this research. I also would like to give my tribute to a group of great people for their endless support that enabled me to achieve my goal.

First, I would like to give my gratitude to my main supervisor Peter Christen who offered me an invaluable assistance, motivation, support and guidance throughout the life of this research. Peter is a knowledgeable, dedicated and motivating supervisor and I have learned greatly from him. Thanks Peter for all the effort you have put throughout the years to help me produce this research. Gratitude is also due to my supervisory panel Huizhi Liang, David Hawking, Ross Gayler and Peter Strazdins who always provided me with support and valuable feedback whenever it is needed.

I wish to express my love and gratitude to my beloved parents, sisters, and sons for their support and endless love through the duration of this work. I am very lucky to have you as my family and I love you all. I cannot thank my lovely friends Dinusha Vatsalan and Minkoung Kim enough for all their encouragement and emotional support, saying thank you is never enough. I also cannot forget all my friends, who with their kindness, prayers, and support pushed me to achieve my goal, and kept me going while I am away from my family. I always thank Allah for giving me this precious gift of having you all around me.

In addition, I thank all my colleagues and the staff in the school of Engineering and Computer Science at The Australian National University for creating a pleasant atmosphere for us students throughout the years. Finally I would like to thank the Australian Research Council (ARC), Veda, and Funnelback Pty. Ltd., for funding my research under Linkage Project LP100200079.

Abstract

Entity resolution (ER), which is the process of identifying records in one or several data set(s) that refer to the same real-world entity, is an important task in improving data quality and in data integration. In general, unique entity identifiers are not available in real-world data sets. Therefore, identifying attributes such as names and addresses are required to perform the ER process using approximate matching techniques. Since many services in both the private and public sectors are moving on-line, organizations increasingly require to perform real-time ER (with sub-second response times) on query records that need to be matched with existing data sets.

Indexing is a major step in the ER process which aims to group similar records together using a blocking key criterion to reduce the search space. Most existing indexing techniques that are currently used with ER are static and can only be employed off-line with batch processing algorithms. A major aspect of achieving ER in real-time is to develop novel efficient and effective dynamic indexing techniques that allow dynamic updates and facilitate real-time matching.

In this thesis, we focus on the indexing step in the context of real-time ER. We propose three dynamic indexing techniques and a blocking key learning algorithm to be used with real-time ER. The first index (named DySimII) is a blocking-based technique that is updated whenever a new query record arrives. We reduce the size of DySimII by proposing a frequency-filtered alteration that only indexes the most frequent attribute values. The second index (named DySNI) is a tree-based dynamic indexing technique that is tailored for real-time ER. DySNI is based on the sorted neighborhood method that is commonly used in ER. We investigate several static and adaptive window approaches when retrieving candidate records. The third index (named F-DySNI) is a multi-tree technique that uses multiple distinct trees in the index data structure where each tree has a unique sorting key. The aim of F-DySNI is to reduce the effects of errors and variations at the beginning of attribute values that are used as sorting keys on matching quality. Finally, we propose an unsupervised learning algorithm that automatically generates optimal blocking keys for building indexes that are adequate for real-time ER.

We experimentally evaluate the proposed approaches using various real-world data sets with millions of records and synthetic data sets with different data characteristics. The results show that, for the growing sizes of our indexing solutions, no appreciable increase occurs in both record insertion and query times. DySNI is the fastest amongst the proposed solutions, while F-DySNI achieves better matching quality. Compared to an existing indexing baseline, our proposed techniques achieve better query times and matching quality. Moreover, our blocking key learning algorithm achieves an average query time that is around two orders of magnitude faster than an existing learning baseline while maintaining similar matching quality. Our proposed solutions are therefore shown to be suitable for real-time ER.

List of Publications

1. **Major Contributions:**

   (a) **Dynamic Sorted Neighborhood Indexing for Real-Time Entity Resolution:** Banda Ramadan, Peter Christen, Huizhi Liang and Ross Gayler. Published in the ACM Journal of Data and Information Quality (JDIQ), 2015, Volume 6, Number 4. (*Corresponds to Chapters 6 and 7 of this thesis*)

   (b) **Unsupervised Blocking Key Selection for Real-Time Entity Resolution:** Banda Ramadan and Peter Christen. In proceedings of the 19th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'15), Ho Chi Minh City, Vietnam, May 2015. In Springer Lecture Notes in Computer Science, Volume 9078, Pages 574-585. (*Corresponds to Chapter 8 of this thesis*).

   (c) **Forest-Based Dynamic Sorted Neighborhood Indexing for Real-Time Entity Resolution:** Banda Ramadan, Peter Christen. In proceedings of the 23rd ACM International Conference on Information and Knowledge Management (CIKM'14), Shanghai, China, November 2014. In ACM Digital Library, ISBN: 978-1-4503-2598-1, Pages 1787-1790. (*Corresponds to Chapter 7 of this thesis*)

   (d) **Dynamic Sorted Neighborhood Indexing for Real-Time Entity Resolution:** Banda Ramadan, Peter Christen and Huizhi Liang. In proceedings of the 25th Australian Database Conference (ADC'14), Brisbane, Australia, July 2014. In Springer Lecture Notes in Computer Science, Volume 8506, Pages 1-12. (*Corresponds to Chapter 6 of this thesis*)

   (e) **Dynamic Similarity-Aware Inverted Indexing for Real-Time Entity Resolution:** Banda Ramadan, Peter Christen, Huizhi Liang, Ross Gayler, and David Hawking. In proceedings of the International Workshop on Data Mining Applications in Industry and Government (DMApps 2013), held at the 17th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'13), Gold Coast, Australia, April 2013. In Springer Lecture Notes in Computer Science, Volume 7867, Pages 47-58. (*Corresponds to Chapter 5 of this thesis*).

2. **Minor Contribution:**

   (a) **Two Stage Similarity-aware Indexing for Large-scale Real-time Entity Resolution:** Shouheng Li, Huizhi Liang and Banda Ramadan. In Proceedings of the 11th Australasian Data Mining Conference (AusDM'13), Canberra, Australia, 2013.

Contents

Acknowledgments **vii**

Abstract **ix**

List of Publications **xi**

1 Introduction **1**
1.1 Research Problem . . . 1
1.2 Real-Time ER Problem . . . 3
1.3 Research Objectives . . . 4
1.4 Research Methodology . . . 5
1.5 Contributions of this Research . . . 6
1.6 Thesis Outline . . . 8
1.7 Notation and Abbreviation . . . 8

2 Background **11**
2.1 Entity Resolution Overview . . . 11
2.1.1 Entity Resolution Definition . . . 11
2.1.2 Benefits of Entity Resolution . . . 12
2.1.3 Applications of Entity Resolution . . . 13
2.1.4 Challenges of Entity Resolution . . . 15
2.2 The Entity Resolution Process . . . 15
2.2.1 Data Cleaning . . . 16
2.2.2 Indexing . . . 17
2.2.3 Comparison . . . 18
2.2.4 Classification . . . 20
2.2.5 Evaluation . . . 22
2.3 Real-Time Entity Resolution . . . 23
2.3.1 Real-Time Entity Resolution Overview . . . 23
2.3.2 Real-Time Entity Resolution Process . . . 24
2.4 Summary . . . 26

3 Related work **27**
3.1 Introduction . . . 27
3.2 Blocking Based Indexing . . . 28
3.2.1 Traditional Blocking . . . 28
3.2.2 Q-gram Indexing . . . 31

3.2.3 Canopy Clustering 32
3.2.4 Mapping Based Indexing 32
3.2.5 Hashing Based Indexing 33
3.3 Sorting Based Indexing 34
3.3.1 Sorted Neighborhood Method 34
3.3.2 Adaptive Sorted Neighborhood Method 35
3.3.3 Progressive Sorted Neighborhood Method 36
3.3.4 Sorted Blocks 37
3.3.5 Suffix Arrays 37
3.4 Meta-Blocking Techniques 38
3.5 Techniques for Real-Time Entity Resolution 39
3.6 Automatic Techniques for Learning Blocking Keys 41
3.7 Summary 44

4 Evaluation Framework 45
4.1 Evaluation Measures 45
4.2 Record Pair Comparison and Classification 47
4.3 Baseline Approaches 48
4.4 Implementation Environment 50
4.5 Data Sets 50
4.6 Summary 52

5 Dynamic Similarity-Aware Inverted Index for Real-Time Entity Resolution 53
5.1 Introduction 53
5.2 Terminology and Notation 54
5.3 Overview of the Approach 55
5.3.1 Index Data Structure 56
5.3.2 Building the Index 57
5.3.3 Querying the Index 59
5.4 Frequency-Based Similarity-Aware Inverted Index (DySimII-F) 60
5.4.1 Building the Frequency-Based Index 62
5.4.2 Querying the Frequency-Based Index 62
5.5 Estimating the Required Number of Comparisons 64
5.6 Experimental Evaluation 66
5.6.1 Scalability of the Proposed Solution 67
5.6.2 Effects of Having Corrupted Attributes on Quality 68
5.6.3 Required Memory Size 69
5.6.4 Effects of the DySimII-F Approach on Index Coverage 69
5.6.5 Effects of the DySimII-F Approach on Matching Quality 70
5.6.6 Effects of the DySimII-F Approach on Memory Requirements 71
5.7 Summary 71

6 Dynamic Sorted Neighborhood Index for Real-Time Entity Resolution 73
6.1 Introduction 73
6.2 Terminology and Notation 74
6.3 Overview of the Approach 76
6.3.1 Index Data Structure 76
6.3.2 Building the Index 78
6.3.3 Querying the Index 79
6.4 Generating the Window of Neighboring Nodes 80
6.4.1 Fixed Window Size (DySNI-f) 80
6.4.2 Candidate-Based Adaptive Window (DySNI-c) 81
6.4.3 Similarity-Based Adaptive Window (DySNI-s) 82
6.4.4 Duplicate-Based Adaptive Window (DySNI-d) 83
6.5 Similarity-Based Dynamic Sorted Neighborhood Index 86
6.5.1 Build and Similarity Calculation Phases 86
6.5.2 Query phase 87
6.6 Estimating the Required Number of Comparisons 89
6.7 Experimental Evaluation 92
6.7.1 Efficiency of Different Tree Data Structures 92
6.7.2 Estimation of the Number of Candidate Records 93
6.7.3 Effects of Using Different Sorting Keys on Index Size 94
6.7.4 Scalability of the Proposed Solutions 94
6.7.5 Effects of Using Different Thresholds 95
6.7.6 Effects of Having Different Number of Duplicates 97
6.7.7 Effects of Pre-calculating Similarities on Comparison Time 98
6.7.8 Required Memory Size 99
6.7.9 General Discussion 99
6.8 Summary 100

7 Forest-Based Dynamic Sorted Neighborhood Index for Real-Time Entity Resolution 103
7.1 Introduction 103
7.2 Terminology and Notation 104
7.3 Overview of the Approach 105
7.3.1 Index Data Structure 106
7.3.2 Building the Index 106
7.3.3 Querying the Index 107
7.4 Estimating the Required Number of Comparisons 109
7.5 Experimental Evaluation 109
7.5.1 Effects of Using Multiple Trees on Quality and Efficiency 110
7.5.2 Effects of Having Corrupted Attributes on Quality 112
7.5.3 Scalability of the Proposed Solution 112
7.5.4 Required Memory Size 114
7.6 Summary 115

8 Unsupervised Blocking Key Selection for Real-Time Entity Resolution 117
8.1 Introduction . . . 117
8.2 Terminology and Notation . . . 118
8.3 Overview of the Approach . . . 119
8.3.1 Generating Candidate Keys . . . 120
8.3.2 Generating Training Data Sets . . . 121
8.3.3 Learning Optimal Keys . . . 124
8.4 Experimental Evaluation . . . 126
8.4.1 Efficiency of Selected Blocking Keys . . . 128
8.4.2 Effectiveness of Selected Blocking Keys . . . 131
8.5 Summary . . . 132

9 Comparative Evaluation 137
9.1 Introduction . . . 137
9.2 Comparative Evaluation Setup . . . 138
9.3 Indexing Techniques . . . 139
9.3.1 Scalability . . . 139
9.3.2 Effectiveness and Efficiency . . . 141
9.4 Blocking Key Learning Technique . . . 149
9.5 Summary . . . 151

10 Conclusions and Future Directions 153
10.1 Conclusions . . . 153
10.2 Contribution Summary . . . 154
10.3 Future Directions . . . 156
10.4 Closing Remarks . . . 158

References 159

List of Figures

1.1 The proposed research methodology 6

2.1 Steps in the entity resolution process 16
2.2 Number of comparisons required in traditional entity resolution 18
2.3 Number of comparisons required in real-time entity resolution 24

3.1 Indexing techniques 30

4.1 Types of errors in the entity resolution process 46
4.2 The Q-gram index (QGI) that is used as a baseline in Chapter 9. 49

5.1 The framework for the proposed dynamic similarity-aware inverted indexing (DySimII) 55
5.2 The data structure for the proposed DySimII technique 56
5.3 Frequency distribution of attribute values of the NC data set 60
5.4 Illustration of the maximum and minimum number of candidate records generated using the DySimII approach 66
5.5 Average insertion, and query times required by the DySimII approach for the NC data set 67
5.6 Average query time required by the DySimII approach for the different CCA subsets 68
5.7 Recall for the different query sets using DySimII 68
5.8 The required memory to index x% of the most frequent attribute values using the DySimII-F. 71

6.1 The static sorted neighborhood method (SNM) 74
6.2 The proposed dynamic sorted neighborhood index (DySNI) 78
6.3 Candidate records generated using the proposed fixed size window approach (DySNI-f) 80
6.4 Candidate records generated using the proposed candidate-based adaptive window approach (DySNI-c) 81
6.5 Candidate records generated using the proposed similarity-based adaptive window approach (DySNI-s) 82
6.6 Candidate records generated using the proposed duplicate-based adaptive window approach (DySNI-d) 84
6.7 The proposed similarity-based dynamic sorted neighborhood indexing (SimDySNI) 86

6.8 Number of records in tree nodes generated using different SKVs for the NC data set 90
6.9 Illustration of the maximum and minimum number of candidate records generated using the DySNI approach 91
6.10 An estimation of the minimum, average, maximum number of candidate records generated by the DySNI approach 91
6.11 Average time required to generate candidate records for different types of search trees 93
6.12 A comparison between the estimated and the real number of candidate records generated by the DySNI approach 93
6.13 Average insertion, and query times required by the DySNI approach . . 95
6.14 Recall and time measures achieved by the different DySNI window approaches using OZ-1 data set 96
6.15 Recall and time measures achieved by the DySNI-s and the DySNI-d window approaches using Febrl data sets 97
6.16 The average times required to compare a query record with a single candidate record for the SimDySNI approach 98

7.1 The proposed forest-based sorted neighborhood index (F-DySNI) 105
7.2 Recall and query times achieved by the F-DySNI using different numbers of trees 111
7.3 Recall for the different OZ data sets with different corruption ratio . . . 112
7.4 Average insertion, and query times required by the F-DySNI approach . 113
7.5 Average query time required for the different CCA subsets 114

8.1 Framework of the proposed automatic blocking key selection approach 120
8.2 Generating training data sets for the proposed automatic blocking key selection approach. 122
8.3 Generating blocking key vectors for the proposed automatic blocking key selection approach 123
8.4 The indexing technique used in the experimental evaluation of the proposed automatic blocking key selection approach 129
8.5 Query time measures for the proposed automatic blocking key selection approach 130
8.6 Recall and MRR measures for the proposed automatic blocking key selection approach compared to the baseline 131
8.7 An example on how to calculate the overall score for blocking keys . . . 133

9.1 Average insertion, and query times required for all compared techniques 140
9.2 Recall values for all compared indexing techniques using the OZ-x data sets 142
9.3 MRR values for all compared indexing techniques using the OZ-x data sets 143
9.4 Query times for all compared indexing techniques using the OZ-x data sets 145

9.5 Summary of the average recall, MRR, query time values for the different techniques using the OZ-x data sets 145
9.6 Recall and MRR values for all compared approaches using the Febrl data sets 147
9.7 Average query times for all compared approaches on the Febrl data sets 148
9.8 Summary of the average recall, MRR, query time values for the different techniques using the Febrl data sets 148
9.9 Comparative results between manual and automatic key selection . . . 150

List of Tables

1.1 A summary of general notations and abbreviations used in this thesis . 9

4.1 Summary of the data sets used in the experimental evaluation 51

5.1 Summary of main notations used in Chapter 5 54
5.2 Top 10 most frequent first names and surnames in the NC data set . . . 61
5.3 Summary of data sets used to evaluate the DySimII 67
5.4 Required memory for the DySimII 69
5.5 The coverage of the DySimII-F 70
5.6 Recall for the DySimII-F 71

6.1 Summary of main notations used in Chapter 6 75
6.2 Worst case complexities achieved using different search trees 77
6.3 Summary of data sets used to evaluate the DySNI 92
6.4 Number of nodes generated using various sorting keys with the DySNI 94
6.5 Average query time for indexed queries using the DySNI 99
6.6 Required memory using various sorting keys for the DySNI 100

7.1 Summary of main notations used in Chapter 7 104
7.2 Summary of data sets used to evaluate the F-DySNI 110
7.3 Required memory for the F-DySNI 114

8.1 Summary of the main notations used in Chapter 8 118
8.2 Summary of data sets used to evaluate our learning approach 128
8.3 Keys selected by the proposed and the FDJ approaches 128
8.4 Number of candidate records generated using the blocking keys selected by the proposed and the baseline approaches 130
8.5 List of candidate keys and their coverage, average block size, variance, and overall score for the OZ-1 data set 135

9.1 Summary of the abbreviations used in Chapter 9 138
9.2 Summary of data sets used to conduct the comparative evaluation . . . 138

Chapter 1

Introduction

In this chapter we provide an introduction to the work presented in this thesis. We describe the research problem in Sections 1.1 and 1.2, the research objectives in Section 1.3, the methodology used to address the research problem in Section 1.4, and the contributions of this research in Section 1.5. We then provide an outline of this thesis in Section 1.6.

1.1 Research Problem

Massive amounts of data are 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 [10, 57, 82]. One important practice in improving data quality is the task of identifying all records that refer to the same real-world entity [33, 57, 82, 113]. This process is commonly known as *entity resolution*, *data matching*, or *record linkage*, among other names [33, 82]. It can be applied to a single or to multiple data sources. When applied to a single data source the process is called *de-duplication* or *duplicate detection* [57, 113]. For the rest of this thesis we will use the term entity resolution (ER).

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 data set representing a single entity include a patient who is represented several times in a hospital data set, a product that is inserted many times into an inventory list, or a voter who is registered more than once in an election 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, businesses could end up sending the same house hold several copies of advertisement mail which leads to increased costs and annoyance to customers receiving multiple copies, while duplicate records in an election roll can lead to voting irregularities.

ER is related to general similarity search approaches, which involve finding similar entities from unstructured data sets (such as emails, news articles, or scientific publications) based on a collection of relevant features that are represented as points

in high-dimensional attribute spaces [66]. However, such approaches are less suited for structured data sets that contain well defined attributes with short values. If unique entity identifiers (such as passport or social security numbers) are available for structured data sets, matching records (detecting duplicates) can be achieved using SQL join statements. However, such identifiers are commonly not available, and therefore ER approaches need to be employed [57, 113].

In general, the ER process (described below and in Chapter 2) is challenging because unique entity identifiers are not available in the data sets that are to be matched. In this case identifying attribute values (like first names, surnames, or addresses) need to be used to perform the matching process. Such attribute values are often of low quality, as they can contain errors, or they might change over time [33]. Therefore, approximate similarity functions (like the edit distance, Jaccard, Jaro, and Jaro-Winkler string similarity functions) are generally required to perform the ER process [33, 57, 113].

ER aims at classifying pairs or groups of records into matches (records that are assumed to correspond to the same entity) and non-matches (records that are assumed to correspond to different entities). The ER process encompasses several steps [33]: *data preprocessing*, which cleans and standardizes the data to be used; *indexing*, which reduces the number of candidate records to be compared in detail; *record comparison*, which compares candidate records in detail using a set of similarity functions; *classification*, where pairs or groups of compared records are classified into matches and non-matches ; and finally, *evaluation*, where the efficiency and effectiveness of the ER process are evaluated. This process is further described in Section 2.2.

Since many services in both the private and public sectors are moving on-line, organizations increasingly require real-time ER (with sub-second response times) on query records that need to be matched with existing data sets [51]. Example applications that could benefit from real-time ER include government social services which need to identify individuals on the spot even if their social security number is not available, police officers who need to identify suspect individuals within seconds when they conduct an identity check using the suspect's personal details, or applications for consumer credit where an individual's credit history needs to be retrieved and evaluated before a new loan can be approved.

Data sets used by most organizations are often not static, but rather dynamic, as queries generally result in a record being modified, added, or even removed from the data set. However, most current ER techniques are based on batch algorithms that are only suitable for static data sets. Such algorithms compare and resolve all records in one or more data set(s) rather than resolving those relating to a single query record, and thus they are not suitable for real-time ER. Therefore, there is a need to develop new techniques that support ER for large dynamic data sets that can resolve streams of query records in real-time.

Indexing is a major step in the ER process, aimed at reducing the search space by bringing similar records closer to each other using a blocking/sorting key criterion (see Chapter 2). Most existing indexing techniques that are currently used with ER are static and can only be used off-line. A major aspect of achieving ER in real-time

is to develop novel efficient and effective dynamic indexing techniques that allow dynamic updates and that facilitate real-time matching by generating a small number of high-quality candidate records that are to be compared with a query record. Such indexing techniques must consider both scalability and matching quality aspects [36].

- **Efficiency and Scalability:** This aspect ensures the ability of an ER technique to match and resolve query records in sub-second times and to adapt with the growing size of large data sets. This is challenging, since the potential number of comparisons needed to match a query record with a data set is equal to the size of that data set. These comparisons require computationally expensive similarity functions [31, 82] which makes it challenging to employ ER technique with larger data sets. Thus, indexing techniques need to address the fact that real-world data sets are getting constantly larger every day.

- **Effectiveness (or matching quality):** Real-world data sets can contain errors and variations, can change over time, or they can be incomplete [10, 36]. The noise in the data makes the ER process more challenging. Indexing techniques, which aim at bringing similar records closer to each other to reduce the search space, are affected by such noise within the data. If an indexing technique is not designed to consider noise within the data it can end up brining dissimilar records closer to each other which will drop the quality of the results produced in the ER process. Therefore, indexing techniques need to consider the fact that real-world data sets are noisy, and they should address this issue to improve the quality of their record matching capabilities.

In this study we focus on the indexing step of the ER process (see Chapter 2), and develop novel dynamic indexing techniques that consider the efficiency and the effectiveness aspects to allow matching of a query record with records in an existing data set in real-time. Section 1.3 describes the objectives of our research in more details, while the following section describes the formal definition of the real-time ER problem.

1.2 Real-Time ER Problem

We assume that data set $\mathbf{R} = \{r_1, r_2, \ldots, r_{|\mathbf{R}|}\}$ contains records of known entities. Each $r_i \in \mathbf{R}$ has a unique record identifier $r_i.id$ and an entity identifier $r_i.eid$. Note that an entity in $\mathbf{R}$ can be represented by one or several records. To indicate that record(s) in $\mathbf{R}$ correspond to the same entity, they are given the same entity identifier. On the other hand, to distinguish all records in $\mathbf{R}$ that represent a single entity, each record is given a unique record identifier. A record r_i in $\mathbf{R}$ is described by a set of attributes, denoted as $\mathbf{A} = \{a_1, a_2, \ldots, a_{|\mathbf{A}|}\}$. All records in $\mathbf{R}$ are assumed to have the same attribute structure. We also assume that a stream of query records $\mathbf{Q} = \{q_1, q_2, \ldots, q_{|\mathbf{Q}|}\}$ is to be matched with $\mathbf{R}$. Each $q_j \in \mathbf{Q}$ is given a unique identifier $q_j.id \neq r_i.id$, $\forall r_i \in \mathbf{R}$; and has the same attribute structure $\mathbf{A}$ as the records

in **R**. In addition, we assume that q_j is given a new entity identifier if it has no matching record(s) in **R** (i.e. no records that correspond to the same entity as q_j). On the other hand, if q_j has matching record(s) in **R**, it is given the same entity identifier as the matching record(s). We also assume that q_j is to be added to **R** after it has been resolved. The problem of real-time ER is defined as:

[Definition] 1.1 Real-time ER: For each query record q_j in a query stream **Q**, find the records in **R** that belong to the same entity as q_j, denoted as the set $\mathbf{M}_{q_j}$, in sub-second time, where $\mathbf{M}_{q_j} = \{r_i \mid r_i.eid = q_j.eid,\ r_i \in \mathbf{R}\},\ \mathbf{M}_{q_j} \subseteq \mathbf{R},\ q_j \in \mathbf{Q}$.

All steps in the traditional static ER process (described in Chapter 2) need to be modified to allow real-time ER. Our focus is to address the indexing step of the ER process.

1.3 Research Objectives

As described in the previous section, one of the main challenges of real-time ER is the growing size of the collected data by many organizations. Performing a query matching with such large data sets is challenging in real-time since the query potentially has to be compared with all records in the data set. Indexing techniques reduce the number of comparisons required in this matching process. The way indexing techniques work is important and affects both the quality of the matching results and the efficiency of the matching process.

Most indexing approaches proposed for ER in the literature fall into two main categories: blocking-based approaches and sorting-based approaches (described in Chapter 3). The majority of these techniques are not designed to work with real-time ER [35, 33]. Therefore, the primary aim of this research concentrates on indexing techniques that can be used for real-time ER. The following is a list of issues that this study aims to address.

1. **Blocking-based indexing techniques for real-time ER:** Blocking-based indexing techniques are used widely in ER to reduce the comparison space required to do the matching. Blocking-based techniques are based on inserting records into blocks according to a *blocking key* criterion and only comparing records that are in the same block. However, most of these techniques in their current form assume static data sets and do not support query matching with dynamic data sets in real-time. There is a need for blocking-based indexing techniques that can be used with real-time ER. We address this issue in Chapter 5.

2. **Sorting-based indexing techniques for real-time ER:** Sorting-based indexing techniques are also widely used in the indexing step of the ER process. They are based on sorting records in a data set according to a *sorting criterion* and comparing only records within a sliding window [79]. Similar to blocking-based techniques, most existing sorting-based indexing techniques assume static data

sets and do not support query matching with dynamic data sets in real-time. Thus, new sorting-based indexing techniques need to be proposed to allow their use with real-time ER. We address this issue in Chapter 6.

3. **Improve matching quality of indexing techniques:** Producing high quality matching results is as important for real-time ER as is the efficiency of the proposed indexing techniques. The quality of matching results is affected by how an index technique works and what sorting/blocking keys are selected. For example, sorting-based indexing techniques are sensitive to errors that occur at the beginning of attribute values which affect the quality of the produced results. Thus, it is important to consider how indexing techniques work and aim to improve the quality of the produced results while considering efficiency when proposing new indexing techniques. We address this issue in Chapter 7.

4. **Automatic selection of blocking/sorting keys:** Indexing aims at bringing similar records closer to each other using a blocking/sorting key criterion (see Chapter 2). Selecting blocking/sorting keys is crucial for the effectiveness and efficiency of the real-time ER process. Traditional indexing techniques require domain knowledge for optimal key selection. However, to make the ER process less dependent on human domain knowledge, automatic selection of optimal blocking keys is required. We address this issue in Chapter 8.

1.4 Research Methodology

An experimental methodology is followed in conducting our research by applying the following steps (shown in Figure 1.1):

1. **Initial exploration:** In this step a general understanding of the ER process was achieved by reading broadly in the literature which helped in recognizing the research problem.

2. **Problem definition:** In this step a definition of the research problem was developed based on the preliminary study that was conducted in the initial exploration step.

3. **Literature review:** In this step an extensive exploration and examination of the literature was conducted to better understand the area of the identified problem and to review current approaches. This step helped in refining the problem, identifying the aim of the research, and planning for possible solutions.

4. **Solution design:** Based on the problem definition and based on the review of the literature, possible solutions were proposed. New algorithms were designed to address the identified research problem.

5. **Conceptual analysis:** A theoretical analysis of the designed algorithms was conducted during this step with regard to complexity, and estimation of the number of comparisons required between a query record and data set's records.

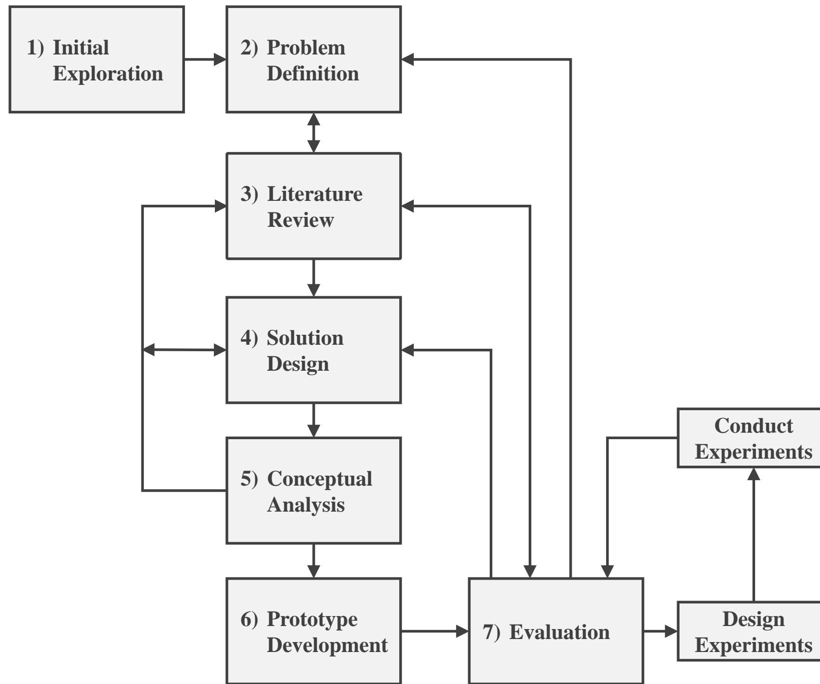


Figure 1.1: The proposed research methodology

6. **Prototype development:** Prototypes of the proposed solutions were developed to be used in the experimental evaluation.

7. **Evaluation:** The developed prototypes were experimentally evaluated using real-world, and synthetic data sets to validate proposed algorithms.

1.5 Contributions of this Research

This thesis is concerned with real-time ER. It focuses on the indexing step of the ER process (described in Chapter 2) and proposes several dynamic indexing techniques that can be used for real-time ER. The thesis encompasses six main contributions:

1. **A dynamic similarity-aware inverted indexing technique for real-time ER:** As discussed in Section 1.1, most existing ER techniques are static and work off-line using batch algorithms that resolve all records in a data set. However, real-time ER requires to resolve only records that are related to a single query record in sub-second time. Thus, we propose a blocking-based dynamic indexing technique that is updated whenever a new query record arrives; this results in a dynamic up-to-date index that supports dynamic data and works well in real-time. We also reduce the size of the index by proposing a frequency-filtered alteration that removes uncommon attribute values and only indexes the most

frequent ones, leading to only a slight drop in the quality of the matched query records. This contribution is published in a workshop paper [131] and a revised version is described in Chapter 5.

2. **A dynamic sorted neighborhood indexing technique for real-time ER:** The sorted neighborhood method (described in Chapter 3), which sorts a data set and compares records within a sliding window, has been successfully used for ER of large static data sets. However, because it is based on static sorted arrays and is designed for batch ER, it is not suitable for real-time ER on dynamic data sets which are updated constantly. We propose a tree-based technique which facilitates dynamic indexing based on the sorted neighborhood method that can be used for real-time ER. We investigate several static and adaptive window approaches when retrieving candidate records. We also improve query matching times by pre-calculating the similarities of attribute values between neighboring tree nodes, resulting in a reduction in matching times. The combined proposed techniques lead to an effective dynamic indexing that works well with real-time ER. This contribution is published in a conference paper [130] and a revised version is described in Chapter 6.

3. **A forest-based sorted neighborhood indexing technique for real-time ER:** As will be described in Chapter 3, sorting-based indexing techniques are sensitive to changes and errors that occur at the beginning of attribute values that are used as sorting keys. To overcome this problem we propose a forest-based indexing technique that uses multiple distinct trees (as described in Chapter 7) in the index data structure where each tree has a unique sorting key. We investigate using single attribute values and concatenation of multiple attribute values as sorting keys to examine which sorting keys are more suitable for real-time ER. This technique showed a noticeable improvement in matching quality for both single and concatenated sorting keys when using multiple trees with only a small increase in query time. Our results show that multiple trees built using concatenated attributes as sorting keys are more suitable for real-time ER than single attribute values. This contribution is published in a conference poster paper [128] and a revised version is described in Chapter 7.

4. **A conceptual analysis of the three proposed indexing techniques:** All proposed indexing techniques are theoretically analyzed with regard to estimating the expected number of record comparisons required by each technique. The conceptual analysis for each technique can be found before the experimental evaluation section in its corresponding chapter.

5. **An unsupervised blocking key selection technique for real-time ER:** As described in Chapter 2, indexing is a crucial step in the ER process. It aims at reducing the search space by bringing similar records closer to each other using one or more blocking/sorting key(s). Selecting these keys is crucial for the effectiveness and efficiency of the real-time ER process. Traditional indexing techniques require domain knowledge for optimal key selection. However, to

make the ER process less dependent on human domain knowledge, automatic selection of optimal blocking keys is required. We propose an unsupervised learning technique that automatically selects optimal blocking keys for building indexes that can be used in real-time ER. We specifically learn multiple keys to be used with the multi-tree sorted neighborhood indexing as described in Chapter 7. This contribution is published in a conference paper [129] and a revised version is described in Chapter 8.

6. **A comprehensive evaluation for the proposed techniques:** We conduct an experimental evaluation of our proposed techniques using multiple large real-world data sets and multiple synthetic data sets in terms of quality and efficiency. We provide a comparative evaluation between our indexing approaches and an existing indexing technique that is commonly used with ER [11, 34, 107]. Our unsupervised learning technique is also compared with a recently proposed blocking key learning algorithm [89].

1.6 Thesis Outline

This thesis is structured as the following: In Chapter 2 we provide the background about ER. Then, in Chapter 3, we present a review of the literature of existing indexing techniques that are used in the ER process. We describe the evaluation framework that we use to evaluate the proposed approaches in Chapter 4. In Chapter 5 we propose a dynamic indexing technique that is based on standard blocking to be used with real-time ER. Next, in Chapter 6 we propose a second indexing technique that is based on the sorted neighborhood method to be used with real-time ER. We propose a third indexing technique in Chapter 7 that improves the matching quality of the real-time ER process by using multiple trees and keys in the index data structure, and in Chapter 8 we propose an unsupervised learning technique that learns blocking/sorting keys used for building indexes that are suitable for real-time ER. In Chapter 9 we provide a comparative evaluation between the proposed algorithms and several existing indexing techniques. Finally, Chapter 10 concludes the study, summarizes the achieved results, and discusses the follow-on future work of this research work.

1.7 Notation and Abbreviation

The following is a list of general notations and abbreviations used throughout this thesis. Additional notations specific to individual chapters are introduced in the relevant chapters.

Table 1.1: A summary of general notations and abbreviations used in this thesis

$\mathbf{R}$	A data set of know entities
$\mathbf{A}$	A set of attributes $\{a_1, a_2, \ldots, a_{\vert\mathbf{A}\vert}\}$ for each $r_i \in \mathbf{R}$
$\mathbf{Q}$	A stream of query records
$\mathbf{C}$	A list of candidate records for a query q_j
$\mathbf{D}$	An inverted index or disk-based data set table
$\mathbf{M}_{q_j}$	A set of all records in $\mathbf{R}$ that belong to the same entity of a query q_j
r_i	A record in $\mathbf{R}$
q_j	A query in $\mathbf{Q}$
n	The size of data set $\mathbf{R}$
$sim(.,.)$	A function used to calculate the similarity between two values ($0 \leq sim(.,.) \leq 1$)
w	Window size used to generate candidate records
SK	A sorting key
SKV	A sorting key value
BK	A blocking key
BKV	A blocking key value
DySimII	A dynamic indexing technique that is based on the similarity aware inverted index (proposed on Chapter 5)
DySimII-f	A frequency-based DySimII that index only most frequent attribute values (proposed in Chapter 5)
DySNI	A dynamic indexing technique that is based on the sorted neighborhood method (proposed in Chapter 6)
DySNI-f	A DySNI that uses a fixed window approach to generate candidate records (proposed in Chapter 6)
DySNI-s	A DySNI that uses an adaptive window approach to generate candidate records, it is based on using similarities between key values of nodes (proposed in Chapter 6)
DySNI-d	A DySNI that uses an adaptive window approach to generate candidate records, it is based on using number of duplicates within nodes (proposed in Chapter 6)
DySNI-c	A DySNI that uses an adaptive window approach to generate candidate records, it is based on using a controlled number of candidate records (proposed in Chapter 6)
F-DySNI	A forest-based dynamic sorted neighborhood index that is based on using multiple distinct trees in the index data structure (proposed in Chapter 7)
QGI	A q-gram based index [11, 34, 107] that is used in ER (described in Chapter 4).
FDJ	A a Fisher disjunctive algorithm that learns blocking keys (proposed by Kejriwal [89])

Chapter 2

Background

In this chapter we present the basic background knowledge needed to understand the research problem addressed in this thesis. We provide an overview of entity resolution in Section 2.1, describe the different steps in the entity resolution process in Section 2.2, and define real-time entity resolution problem in Section 2.3.

2.1 Entity Resolution Overview

Entity resolution (ER) is the process of identifying real world objects based on a group of features [145]. Humans used ER techniques long time ago, even before computers were invented. For example, hunters used animal footprints to identify which animals are living in a certain region, doctors used bone shapes of some body parts (like jaws and elbows) to determine the race of a human, and clerks used information on records (in their hard copy format) to perform a manual cleaning to remove duplicated records. However, since the development of computers, automatic ER techniques started to rise and flourish.

In 1959, Howard Borden Newcombe [115] established the foundation for probabilistic ER, which was then mathematically formalized for the first time by Fellegi and Sunter [62] in 1969. In the middle of the twentieth century, people started using computerized ER techniques in various fields [145]. Statistical and health organizations were among the first to use ER techniques extensively to enhance their operations [33, 165]. Since then, a substantial amount of research has been conducted in the area of ER, and various ER techniques have been developed [57, 82, 33, 166]. In this section we provide a brief overview of ER, its definition, benefits, applications, and challenges.

2.1.1 Entity Resolution Definition

In the last decade, researchers from different fields have studied the problem of ER. Statisticians and researchers in the health domain refer to the problem as *record* or *data linkage* [1, 83, 112]. In the database domain it is referred to as *merge/purge* [79, 78, 80], *data de-duplication* [14, 76, 144], and *instance identification* [38, 108, 155]. In the computer science domain it is referred to as *data matching* [33, 97, 140], *record*

matching [28, 61, 151], *reference reconciliation* [49, 136, 137], *entity resolution* [22, 101, 142], and *duplicate detection* [81, 113, 120] or *de-duplication* [34, 93, 138]. We will use the term entity resolution (ER) for the rest of this thesis.

Various scholars have defined the ER problem as follows. Newcombe et al. [115] defined ER as "bringing together two or more separately recorded pieces of information concerning a particular individual or family", while Fellegi and Sunter [62] described ER as "the problem of recognizing those records in two files which represent identical persons, objects, or events". ER was also defined by Winkler [167] as "the methodology of bringing together corresponding records from two or more files or finding duplicates within files", and by Elmagramid et al. [57] as "to identify records in the same or different data sets that refer to the same real-world entity, even if the records are not identical". Usually, when the terms duplicate detection or de-duplication are used, this refers to finding matches (duplicates) in a single data set [33, 113].

We refer to ER as the problem of identifying different records in multiple data sets that represent the same real-world entity. An entity could be any object that exists in the real-world like a person, a product, a business or an organization. Having duplicate records within data sets of businesses and organizations can affect the quality of their provided services and outcomes [10, 57, 82]. Therefore, businesses and organizations in various areas are increasingly using ER techniques to merge duplicates and clean data sets to improve the quality of the data sets they are using. More benefits of using ER are described next.

2.1.2 Benefits of Entity Resolution

Given that the process of collecting data is growing enormously within most organizations, it is important for such organizations to manage and analyze their data to accomplish their daily business operations effectively. Various data analysis, management, and mining tools are currently used to assist businesses and organizations to better understand, perform, and improve their business operations [71]. Providing such tools with dirty data (that contain errors and duplicates) could generate incorrect and misleading output [82]. ER techniques can be used to remove duplicate records from data sets to improve the achieved benefits of such tools.

Moreover, a data warehouse, which is a decision support system, is important for various businesses and organizations [71, 123]. It can provide organizations with strategic information and historical data which enhance data analysis, reporting and decision making [123]. A data warehouse generally requires integration of multiple data sources into one clean and consistent information system [123]. The quality of the source data has to be improved before loaded and integrated into a data warehouse which can be achieved by cleaning and standardizing the integrated data via ER techniques [127].

2.1.3 Applications of Entity Resolution

Matching records that correspond to the same entities from different data sets has been intensively used in various areas including health, national censuses, crime and fraud prevention, and national elections. The following is a list of the main application areas that use ER to improve their operations:

- **Health services:** Health services is one of the main areas that had an early start in matching data between multiple resources [33]. In health services, personal and medical information is collected each time a person comes into contact with any health services (such as a doctor, a clinic, or an emergency department of a hospital). If such information is matched between various health services, it can be analyzed to improve the health system [148].

  Matching records from different health services provides researchers with better quality data, patients with improved services, and policy makers with more reliable evidence to support their decision making [124]. Moreover, matching health data improves the ability to detect adverse health trends, identify disease outbreaks, detect health service problems (such as procedural or managerial issues), and improve clinical practice [125].

  Australia is one of the world leaders in health data matching. It established the Health Services Research Linked Database program in Western Australia in 1995 [47]. Until 2003, this program has supplied data for 258 projects, which produced 708 research outputs, including 172 journal articles. Moreover, the matched data from the Western Australian data matching program was directly responsible for various changes to policies and clinical practices [25]. In 2008 a study was conducted to evaluate the research output based on using matched health data from the Western Australian data matching program. It concluded that matching data between different services in the heath domain can make a substantial and quantifiable contribution to population health and policy development [24]. Similar health data matching programs were founded in different counties as well, including the UK, Canada, US and New Zealand [148].

- **National statistical agencies:** National statistical agencies (NSAs) are responsible for collecting and publishing data related to various areas such as population, economy, health, education, culture or politics [33]. Statistical data generated by NSAs are provided to governments, organizations or communities to improve decision making, evaluation, and assessment procedures [33]. NSAs have a long history of matching records when conducting statistical surveys and developing data collections [163].

  Matching records allows the reuse of existing data. For example, in 2006 the Australian Bureau of Statistics[1] required to produce an estimation about the number of interstate migrations. However, there were no direct data sources that could be used to measure interstate migrations and it was expensive to

[1] can be found at http://www.abs.gov.au/

conduct a survey for that purpose [147]. Therefore, the Australian Bureau of Statistics used a number of indirect administrative data sources to estimate the number of interstate migrations, including electoral roll registration, family allowance payments, and Medicare registration data from Medicare Australia [147]. Matching existing data reduces costs, improves data quality, and reduces the burden of conducting surveys [163].

The US Census Bureau was one of the first to adopt data matching techniques, and it also had a key role in ER research [126, 161, 164]. Currently, most bureaus of census, including the Australian Bureau of Statistics, apply data matching techniques to improve the quality for their collected data.

- **Fraud detection:** Fraud is defined in the Oxford Dictionary as "criminal deception; the use of false representations to gain an unjust advantage" [116]. With the rapid and enormous growth of modern technology, and with the fast communication means currently available, fraud is increasing dramatically leading to the loss of billions of dollars worldwide [21]. Many fraud detection problems involve very large data sets. For example, in the period of November 2011 to November 2012, around 1.9 billion credit card transactions were carried out in Australia [132]. This large number of transactions shows the importance of fraud detection. Assume that only 0.1% of these transactions were fraudulent, this means that 1.9 million transactions will be affected by fraudsters only in that year.

  Currently, various information systems, statistics applications, and data mining techniques are used for fraud detection in numerous businesses and organizations [30, 121]. ER techniques can be used to improve the quality of the data used to enhance the performance of such data mining and statistical tools. Moreover, ER can be used to match known fraudsters to other individuals in the data. This should improve fraud detection since fraudsters rarely work in isolation from each other [21].

- **Other applications:** ER has been applied in various other application areas. Search engines [69] use ER techniques to identify documents that cover similar topics. Many comparison shopping sites also take advantage of ER to identify similar products to be able to provide price comparisons successfully [17]. Digital libraries, on the other hand, identify similar articles to improve searching facilities, and to de-duplicate their data sets [170]. De-duplication can also be used in businesses to remove duplicate records from their customer mailing lists, which will reduce the cost of sending advertisement mails to customers [113]. Moreover, business partners can benefit from matching records across their data sets to achieve successful business activities.

2.1.4 Challenges of Entity Resolution

The aim of ER techniques is to identify all records in multiple data sources that refer to the same real-world entity. In order to conduct the matching process between data sets, ER has to compare all records within the data sets to find matching (duplicate) records. Applying ER techniques involves two main challenges:

- **Efficiency and scalability to large data sets:** In order to match records from multiple data sources all record pairs from the data sets have to be compared in detail to identify matching records. The complexity of such comparisons is quadratic in the size of the compared data sets and therefore not feasible. Moreover, the comparison process has to be conducted using approximate comparison functions to address the data quality challenge. Such comparison functions are often computationally expensive and some have a computational complexity that is quadratic in the length of the compared attribute values [33].

- **Quality of the matching results:** Most data sets do not have reliable unique identifiers to distinguish different entities. To overcome this issue, ER techniques use identifying attributes, like names and addresses, to identify different records that represent the same real-world entity. However, such attributes often contain variations and typographical errors [10] which makes distinguishing different entities across data sets more challenging, and in turn affects the quality and accuracy of the matching process.

ER techniques should aim to address the above challenges and provide effective and efficient algorithms that produce high quality matching results efficiently, and they should also scale to large data sets.

2.2 The Entity Resolution Process

The ER process encompasses five major steps as illustrated in Figure 2.1. Using clean data sets is vital to improve the quality of the matching process. Therefore, the first step in the ER process is to clean and standardize the data sets that are to be matched. The second step, indexing, addresses the efficiency and scalability of the ER process and aims at reducing the number of comparisons required to do the matching by grouping similar records together and only comparing those that are likely to be similar. Records that are grouped together in the indexing step move to the next step, the comparison step, where a list of candidate record pairs is generated from records within each group. These candidate pairs of records are compared using various comparison functions [31, 82] and classified in the classification step into 'matches', 'potential matches', or 'non-matches'. The quality of the classified record pairs is then assessed in the evaluation step. The various tasks in the ER process are described in more details in the following sections.

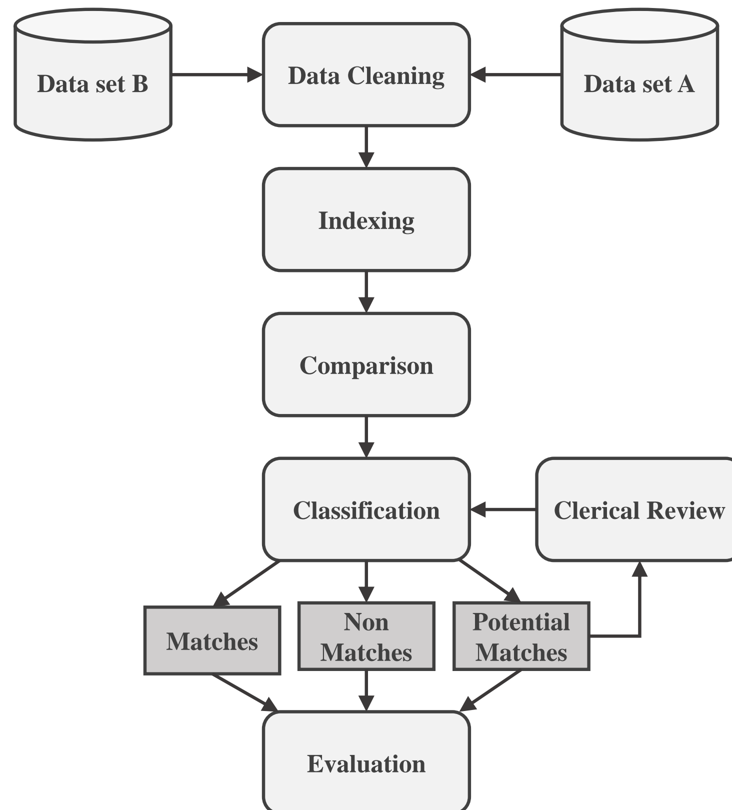


Figure 2.1: The steps of the ER process (taken from [33]).

2.2.1 Data Cleaning

Data quality was defined by Tayi and Ballou [146] as "fitness for use" which suggests that data of a certain quality that is appropriate for one use could potentially not be appropriate for another use. Many data quality dimensions have been identified in previous research [10, 139, 154]. However, the data quality dimensions that have the most influence on the ER process are: *accuracy*, *consistency*, and *completeness* [33, 113]. *Accuracy* is the extent to which data are correct, reliable, and free of error [113]; *consistency* is the extent to which several data are consistent and can be combined without problems [82]; and *completeness* is the extent to which the data are complete and have no missing values [33].

Most collected data in the real-world are dirty and can include errors, missing values, or inconsistent values [10, 82]. Moreover, different data sources can have different structures and formats [33, 127]. Earlier research has confirmed that data quality is vital for the ER process and significantly affects its outcome [33, 82] and that cleaning data makes it comparable and more usable [57]. Therefore, it is important to clean data before it is used for ER. Several approaches can be applied to clean data sets, the following are the main approaches that are commonly used to clean and standardize the data in the first step of the ER process [57]:

- **Standardization:** As discussed earlier, real-world data sets usually do not have unique identifiers, which requires the use of identifying attributes like names and addresses to perform the ER process. Before attribute values are used in the matching process these attributes have to be converted into a standard form. This standardization process is performed before carrying out the ER process to improve the probability of finding matching records. Standardization usually consists of removing unwanted characters, correcting spelling errors, expanding abbreviations, and ensuring that the same coding systems and measuring units are used across data sets [33, 127, 163].

- **Parsing:** Parsing aims at identifying and isolating individual components in the standardized attribute values from the previous step into well-defined and consistent components. One example of parsing would be to split personal names into title, given name, and family name. Another example is to split phone number into country code, area code, and phone number. Paring should improve the process of data matching because it allows comparing individual components rather than complex and long attribute values [33, 57, 163].

- **Data transformation:** In this step parsed and standardized attribute values are converted into their proper and correct form. Common conversions in this step include decoding encoded attribute values to their original form, range checking, and dependency checking. Range checking, examines attribute values and ensures that they fall within the expected range. Dependency checking, makes sure that attribute values are consistent and have no conflict with others. An example of dependency checking is to validate the name of a city and its postcode if they are consistent, or if the street name and type are consistent. If values are not consistent, this usually would be a data entry error and needs to be corrected by changing one attribute value to be consistent. Data conversions can either be automatically completed or it could be flagged to indicate that the record contains some inconsistent values [33, 57, 127].

2.2.2 Indexing

After conducting cleaning and standardization in the previous step, the data sets are ready to be matched. Assume that we need to perform the matching process for data sets $\mathbf{R}_A$ and $\mathbf{R}_B$. Each record from $\mathbf{R}_A$ potentially has to be compared in detail with all records from $\mathbf{R}_B$. The number of record pair comparisons required to complete the matching process is equal to the product of the number of records in $\mathbf{R}_A$ and $\mathbf{R}_B$, as seen in Figure 2.2. The total number of record pair comparisons therefore increases quadratically with the size of the matched data sets. Such comparisons are usually accomplished using approximate comparison functions [31, 82] which are often expensive. This can lead to a performance bottleneck [36] that makes matching large data sets infeasible.

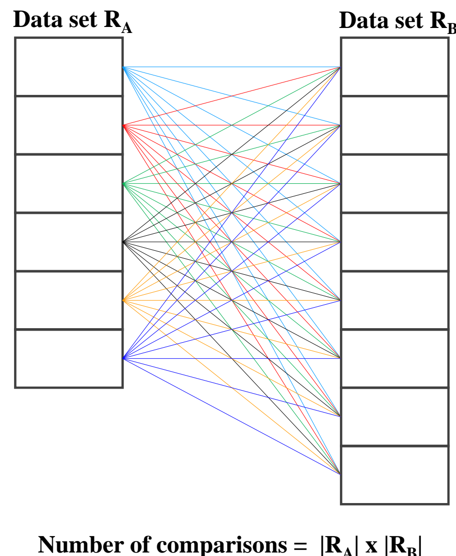


Figure 2.2: The number of comparisons required to match two data sets in the traditional ER process.

One way to overcome this problem is to reduce the number of compared record pairs using indexing techniques [34]. The idea behind indexing is to bring similar records closer to each other in order to compare only those that are more likely to be a match, and avoid comparing records that are more likely to be a non-match. There are two general forms of indexing techniques used with ER: blocking-based [34] and sorting-based [79] techniques.

In blocking-based techniques, records are inserted into blocks according to a blocking key criterion and subsequently only records that are in the same block are compared. In the sorting-based techniques, all records are sorted using a sorting key criterion, and then a window slides over the sorted records, comparing only those records that are within the sliding window at any one time. Both blocking and sorting keys are usually based on one or a concatenation of attribute values (e.g. the concatenation of first name and surname values). More details about indexing will be provided in Chapter 3. The output of the indexing step is a group of similar records that are brought together in order to be compared in detail in the comparison step, which is described next.

2.2.3 Comparison

After bringing similar records closer to each other (in the form of groups) in the previous step, record pairs that are in the same group require more detailed comparison to decide whether the records refer to the same real-world entity or not. This comparison step is generally based on comparing several record attributes (i.e. identifying attributes such as first name, last name, address and date of birth, are usually compared). Two types of comparisons can be applied:

- **Exact comparison:** An exact comparison between two attributes gives a result of 0 or 1, based on the values of the compared attributes. If the values of the compared attributes are exactly the same, the similarity of those attributes will be 1. On the other hand if the values of the compared attributes are not exactly the same this will give 0. This type of comparison is generally used when the data is clean and contain no typos and errors (which is not the case in most real-world data sets).

- **Approximate comparison:** An approximate comparison between attribute values can be achieved by using either similarity measures (that measure how much two values are similar to each other) or distance measures (that measure how close two values are to each other). The normalized similarity between two attribute values $s1$ and $s2$, denoted as $sim(s1, s2)$, can have a value between 0 and 1. The higher the similarity between the compared values the more likely these values are a match. Having a similarity of $sim(s1, s2) = 1$ indicates that $s1$ and $s2$ are exactly the same, while $sim(s1, s2) = 0$ indicates that the two values are totally different. Moreover, the normalized distance between two values, denoted as $dis(s1, s2)$, has a value between 0 and 1. The larger the distance between two values, the less likely they are a match. This means that a value of $dis(s1, s2) = 1$ indicates that the compared values are different from each other, while $dis(s1, s2) = 0$ indicates that the compared values are exactly the same. There is an association between the distance and the similarity represented by $sim = 1 - dis$. For a similarity measure to be associated with the distance measure it has to fulfill the following properties of distance functions [33]:

$dis(i, i) = 0.0$	The distance from an object to itself is zero
$dis(i, j) \geq 0.0$	The distance between two objects is non-negative
$dis(i, j) = d(j, i)$	The distance between two objects is symmetric
$dis(i, j) \leq d(i, k) + d(k, j)$	Fulfill the triangular inequality

  The triangular inequality states that the direct distance between two objects is never greater than the combined distance when going through a third object [35].

Since real-world data is dirty and often contain typographical errors and variations, using exact comparison measures to perform the ER process likely causes the missing of true matches. Therefore, approximate comparison functions should be used to overcome the problem of finding matching records that have errors or variations in their attribute values [31]. Various approximate comparison functions are used with ER for the different data types, some are described below [33, 82, 113].

Edit distance is a comparison function that is also known as Levenshtein edit distance [114]. It is defined as the minimum number of edit operations (i.e. character insertion, deletion and replacements) that are required to convert one string into another [114, 113]. This comparison function considers the compared values as a whole and are not divided into tokens. It is widely used in practice, however, it has a quadratic complexity and is not suitable when a whole part of the compared val-

ues differs, for example when comparing 'John Smith' and 'Peter John Smith' [113]. Various extensions and modifications for the original edit distance function were developed to improve its efficiency and to allow different weights of different types of edits. Extensions on the basic edit distance function can be found in [39, 70, 172]

Q-gram based function [11] compares two strings by first splitting the compared strings into sub-strings of length q, which are called *q-grams*. For example, the generated q-grams for the string 'dawoud', where $q = 2$, are: ['da', 'aw' ,'wo', 'ou', 'ud']. When comparing two attribute values, the generated q-gram sets for the two values are compared with each other to find the number of common q-grams. The number of common q-grams can be converted into a similarity using a coefficient method like the Jaccard coefficient, Dice coefficient, and the Overlap coefficient [31].

Jaro [87] introduced an approximate comparison function that considers the lengths of the two strings, the number of common characters, and the types of errors in those strings (i.e. insertions, omissions, or transpositions). Transposition means two adjacent characters are swapped in the two strings, for example, in the strings 'Sydney' and 'Sydeny' the 'ne' from the first string is swapped in the second string to 'en'. Common characters between two strings are those that are equal and have positions within the two strings that do not differ by more than half of the length of the shorter string [113].

Winkler [162] improved Jaro's comparison function, where he gives more importance to agreement on the initial characters of the compared strings than to the agreement on later characters. His idea was based on the results in [122] which conclude that the most reliable characters of a string are those at the beginning. Various enhancements of the Jaro-Winkler function were proposed. A comparison between variations is found in [171]. More approximate comparison functions are described in [33, 113].

To summarize, all records that are grouped together within the same block (generated in the indexing step) are compared in the comparison step. This comparison process includes comparing several attribute values for each record pair in the same block. Each attribute will be given a numerical similarity value (using one of the approximate comparison functions). The result will be a vector of similarity values for each record pair in a block. These comparison vectors are the input for the classification step described next, where a decision will be made on whether a compared record pair is a match or a non-match.

2.2.4 Classification

The comparison vectors (generated in the comparison step) are used in the classification step to classify record pairs into three classes: matches, non-matches, or potential matches. A potential match indicates that it is not clear (based on the given attribute values) if a record pair is a match or not. In this case, a manual classification is required to finalize the classification of the record pair. Several classification techniques are available and can be used in the ER process. These techniques can be categorized into the following four main categories [33, 82]:

- **Threshold-based classification:** In this classification technique, the similarity values (i.e. comparison vector) for the attributes of a record pair are summed into an overall similarity for the pair [33]. This overall similarity is then used for classifying the pair into one of the three matching classes based on an upper (u) and a lower (l) threshold. Record pairs with an overall similarity greater than u are classified as matches, record pairs with an overall similarity less than l are classified as non-matches, while record pairs with an overall similarity in between the two thresholds are classified as potential matches.

  A major drawback of this technique is that it does not consider how informative each attribute is, and how each individual attribute can affect the classification process. For example, the 'first name' attribute might be more informative to the matching process than the 'suburb' attribute, but this technique treats both attributes the same in regarding to classifying the record pair, and the importance of each individual attribute is lost in the summation of all attributes similarities. This drawback can be handled by giving weights to each individual attribute before summing the similarities.

- **Probabilistic classification:** Since real-world data is dirty and contain typographical errors, missing values or variations it will affect the quality of the matching process. To resolve this issue, the probabilistic approach considers assigning different weights for different attributes when they are used to classify records into a match or non-match [33]. The likelihood of two records to be classified as a match, non-match, or a potential match is based on the accumulative weight of agreement or disagreement among attribute values of the compared records. Assigning such weights generally depends on the characteristics of the attribute values, their frequencies, and their approximate similarities. For example, an attribute with a value that has high frequency in a data set (like the first name 'peter' for instance), is generally less discriminative than other attribute values which are not frequent (like the surname 'schwarzenegger' for instance) and therefore, agreement weights should be adjusted according to the frequencies of attribute values, and lower weights should be given for more frequent attribute values [33].

- **Rule-based classification:** In this technique, the classification decision is based on a set of rules (constraints). These rules are related to the similarities assigned to each attribute value in the comparison vectors of the compared records (that are generated in the comparison step). Individual rules can be built using a single or multiple attribute values. The set of decision rules can be manually built, based on domain knowledge, which is very time consuming. Alternatively machine learning techniques [71, 111] can be used to learn the optimal set of decision rules from a training data set. Examples on rule-based classification techniques for ER include [101, 157].

- **Supervised and unsupervised classification:** Data mining and machine learning are well researched areas. Many classification and clustering techniques have been developed in the last decade [71]. These techniques can be used in the classification step of the ER process. Classification techniques are grouped into two major categories: supervised and unsupervised techniques. Supervised classification uses a training data set that contains record pairs and their true classification status. This training data is used to train the classification model to be able to classify the remaining unlabeled record pairs. The main drawback with supervised classification is that generating training data sets is difficult and costly. Unsupervised techniques, like clustering, do not require training data sets. In clustering, similar record pairs that refer to the same entity are grouped together in one cluster. [107]. Various supervised and unsupervised classification techniques are detailed in [71, 168].

- **Collective classification:** The previously described techniques perform the classification process based only on the similarities of the compared records independently from all other records in the data set. However, collective classification techniques [15, 16, 81] also take the characteristics of other records in the data set into consideration. These techniques are used for data sets that have some relational information between its records, for example people who share the same workplace, or articles that share the same co-authors. The idea of these techniques is to use the relational information between the records when building the decision model. Entity resolution that uses this technique is known as *collective entity resolution* [15, 16]. Although classification that considers relational information gives better matching quality it is computationally expensive and is currently not suitable for large data sets [33].

The output of the classification step is record pairs that are classified as matches, non-matches, or potential matches. The records that are classified as potential matches still need a clerical review to be given a final classification result. The quality of the classified record pairs needs to be evaluated to observe how well the classification model performed. The following section describes the evaluation of the ER process in more details.

2.2.5 Evaluation

After conducting the ER process, it is important to know how successful the matching process was. Some of the main issues to consider when evaluating an ER technique include the quality of the matching results and the efficiency and scalability of the ER technique. Quality relates to how many of the record pairs that were classified as matches relate to a real-world entity [36]. Different measures are generally used for evaluating the quality of the matched results in ER, including precision, recall, and F-measure [36, 113]. Efficiency and scalability relates to how fast a technique is and

how well it can adapt with being used with large data sets [36]. To evaluate the scalability of an ER technique run time, the number of comparisons, and reduction ratio are commonly used [36, 113]. More details bout evaluation measures are provided in Chapter 4.

2.3 Real-Time Entity Resolution

The traditional ER process (described in Section 2.2) is generally based on batch processing and runs off-line to match multiple static data sets. However, most businesses and organizations are moving on-line where they have to provide their services to their customers instantly. In addition, the data sets used by organizations are dynamic and are modified constantly. Organizations increasingly have to deal with a stream of query records that include information about entities in their dynamic data sets. These query records have to be matched with existing data sets and resolved on-line using ER techniques. This process is referred to as real-time ER [33, 57, 113].

2.3.1 Real-Time Entity Resolution Overview

Similarity search approaches [50, 66, 74] aim at identifying similar entities within unstructured data (like social media, web pages, and email). Such approaches are not suitable for identifying entities in structured data sets [33]. Moreover, traditional ER techniques (as described in Section 2.2) are designed to match multiple data sets with each other (which means that for matching two data sets, each record in the first data set has potentially to be compared with all records in the second data set). Such traditional ER techniques are not suitable for real-time ER, therefore, new techniques that are tailored for real-time ER are required. Real-time ER can be defined as a query-based data matching process where a stream of query records has to be resolved against records in one or more existing data set(s) in sub-second time [33, 57, 113] (as illustrated in Figure 2.3). A formal definition of the real-time ER problem is described in Section 1.2.

Real-time ER is required in many applications. One example application that illustrates real-time ER on dynamic data sets is credit bureaus. A credit bureau is responsible for maintaining a large data set that contains credit history for individuals and businesses. If a person applies at a bank for a loan, the bank sends the customer's details to a credit bureau and requests a credit check. The credit bureau has to match the customer's details to their data set and if found, it will send the customer's credit history back to the bank. The bank's decision of approving the loan depends on the credit report received from the credit bureau. This process is done in real-time, where the bank receives the response instantly.

Other examples include government social services which need to identify individuals on the spot even if their social security number is not available, police officers who need to identify suspect individuals within seconds when they conduct an identity check using the suspect's personal details, and on-line product comparison sites

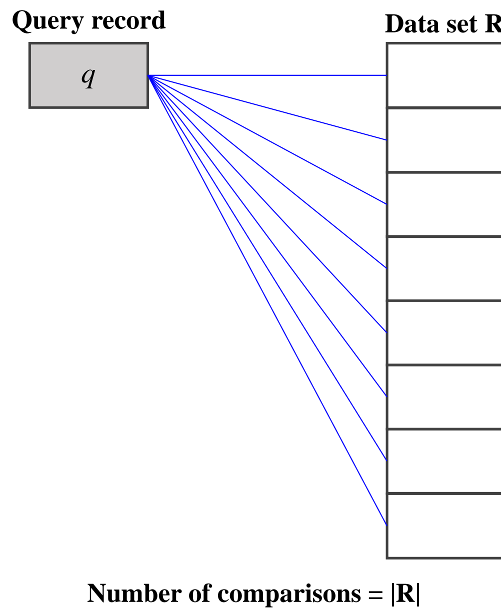


Figure 2.3: The number of comparisons required for query-based matching in the real-time ER process

where the site has to eliminate duplicate results in real-time from the result list based on the queries entered by customers.

The real-time ER process is challenging. On top of the challenges of traditional ER discussed in Section 2.1.4, real-time ER approaches have to be very efficient since query records have to be matched within sub-second time. In addition, ER approaches have to take into consideration the fact that real-world data are dynamic and that they grow and change constantly. The steps in the real-time ER process are the same as in traditional ER, however, each step has to consider the fact that the query matching process has to be conducted very efficiently (within a sub-second time). The following section describes how the different steps of the ER process can be conducted in real-time.

2.3.2 Real-Time Entity Resolution Process

For real-time ER, the various steps in the ER process (described in Section 2.2) have to consider the fact that a query record has to be resolved in real-time. The following is a description of how each step in the ER process can be carried out to facilitate query-based matching with dynamic data sets in real-time.

1. **Cleaning and standardization:** Queries are usually generated using data entry forms that consumers fill and submit to an organization. Controlling the

quality of the received queries is important for completing the matching process effectively. A query with low quality data affects the matching results and leads to low quality output. Therefore, it is important for an organization to control the quality of received queries as much as possible. This involves two main procedures.

First is controlling the data entry process before submitting the query. This can be achieved by keeping the typed-in values to a minimum to reduce possible errors and variations in the query records. For example, this includes using select drop down menus, check boxes, and radio buttons instead of text fields whenever is possible. If a text field is used, a validation check on the entered values should be conducted before allowing the submission of the query. This could include checking for empty fields, auto spelling checks, and consistency checks between entered values.

Second is performing cleaning and standardization on the submitted query using the same model that an organization uses for cleaning its existing data set. The cleaned and standardized query is then ready to be matched with records within the existing data set. Cleaning and standardizing a query record before conducting the matching process with the existing data set is not generally computationally expensive.

2. **Indexing:** Real-time ER requires using dynamic indexing techniques that can be updated constantly and can handle resolving stream of query records in real-time. The index is built using existing data sets, which can then be used to resolve arriving query records. When a query record arrives, and after it is cleaned (in the previous step), it will be inserted into the index to be resolved. A list of ranked results are then sent to the requested user. This is different from traditional ER where multiple data sets have to be matched and an index is built only once for the matched data sets. If the data sets change, these changes will not be included in the index unless it is re-built from scratch (off-line).

   In real-time ER for dynamic data sets, indexing has to be fast enough to facilitate retrieving records from the index in sub-second time. Moreover, the indexing technique should have the ability of adding new records to the existing index to facilitate working with dynamic data sets. Most available indexing techniques work with batch processing (off-line) on static data sets, and are not suitable for real-time ER with dynamic data sets.

3. **Comparison:** The comparison step is similar in both traditional (off-line) ER, and real-time (on-line) ER. In traditional ER, a pair-wise comparison has to be conducted for all records that are within the same block, while for real-time ER, a query has to be compared with all records that are inserted into the same block as the query record. The comparison step is achieved by using approximate comparison functions (that are usually expensive). For real-time ER, we

must always consider the complexity of the chosen comparison function. Since speed is vital, comparison functions with less computational complexity should be used.

4. **Classification:** We discussed earlier that the classification models used for classifying record pairs into matches, non-matches, and potential matches are based on the characteristics of the existing records (i.e. attribute values, frequencies of attribute values, dependencies between attributes, or relationships between attributes). Simple classification models (like threshold-based, rule-based, and probabilistic classification models) that do not require additional information about the matched data to be generated are more suitable to be used in the real-time ER process. On the other hand, models that require generating new information about the matched data, such as re-training the model or re-building relational models (e.g. supervised and collective classification approaches) are less suitable for real-time ER.

   However, to use such complex classification approaches with real-time ER, any additional information about the data can be re-generated off-line on a regular basis, while the matching process can be conducted on-line in real-time. For example, if a supervised classification model was used, and the data set has changed by adding more records with new characteristics, the classification model has to be re-trained on the new data to be able to continue with the classification process effectively. This training can be done off-line, then the new trained classification model can be used in real-time with the ER process.

In summary, traditional ER techniques are designed to work with static data sets based on using batch processing algorithms. Such approaches are not suitable for conducting the ER process in real-time on dynamic data sets. Therefore, techniques that can handle query-matching in real-time using dynamic data sets are required. Indexing is the main step in the ER process that affects the efficiency of real-time ER techniques as it reduces the search space. Fast and dynamic indexing techniques are required to enable conducting the ER process in real-time.

2.4 Summary

In this chapter we provided an overview of the ER process, its definition, benefits, applications, and challenges. In addition, we described the steps of the ER process and provided a brief overview of real-time ER and its steps. Next we will provide a review of the literature on existing techniques that are commonly used in ER for indexing (which is an important step in ER that makes the matching process efficient and suitable for real-time).

Chapter 3

Related work

In this chapter we review major indexing techniques that are used with entity resolution. In Sections 3.2 and 3.3 we describe major blocking-based and sorting-based indexing techniques, respectively. In Section 3.4 we discuss meta-blocking and in Section 3.5 we describe recent real-time entity resolution techniques. In Section 3.6 we review blocking key learning approaches, and we summarize the chapter in Section 3.7.

3.1 Introduction

As mentioned in Chapter 2, indexing is one of the most important steps in real-time entity resolution (ER) as it reduces the search space which leads to reducing the number of comparisons between records required in the comparison step. The idea behind indexing is to bring similar records closer to each other to make it possible to only compare those that are likely more similar. This reduction in the comparison space gives the ER process the ability to adapt to larger data sets, and makes it possible to perform the matching process in real-time.

Indexing techniques are also employed in the area of database systems to improve the performance of search operations. Major indexing techniques in the area of database systems include techniques based on hash-tables and tree data structures [77]. The main hashing-based indexing techniques include extensible hashing [58], linear hashing [102], and partial-match hashing [135]; while the main tree-based techniques include AVL trees [2] (which are usually used with main memory indexing), B and B+ trees [40, 41] (which are mostly used with disk-based indexing), and T-trees [99] (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 [77]. Such data structures (like hash tables and trees) can be used to build indexing techniques to be used with ER.

Moreover, information retrieval systems (IR) apply indexing techniques to improve the efficiency of the retrieval process. Major indexing techniques that are used in IR systems are based on utilizing suffix arrays or trees [67, 105, 150, 156] (that use suffix substrings of text in a document to refer to that text in the index [105]) , signature files [59, 60, 91, 98] (which represent documents using bit-based fingerprints

that are generated using different hashing techniques such as Bloom filters [20]), and inverted files [6, 9, 174] (a collection of lists, one per term, recording the identifiers of the documents containing that term [174]). More details about indexing techniques that are used in IR can be found in [9, 106, 174].

In addition, similarity search approaches apply indexing and filtering techniques which aim at reducing the search space to improve the efficiency of similarity search solutions. Various filtering techniques can be employed such as prefix filtering approaches [12, 29] (where records are filtered based on common prefix substrings), signature filtering approaches [12] (where records are filtered based on fingerprints generated using hashing methods), suffix filtering approaches [169] (where records are filtered based on common suffix substrings), and positional filtering approaches [96, 169] (that exploit the ordering of substrings in the compared records). Moreover, combined filtering approaches is used such as the PPJoin [96, 169] algorithm that combines both suffix and positional filtering to improve the efficiency of the search process.

The indexing step in the ER process include two tasks: building the index, and retrieving records from the index (generating candidate records) [34]. As for the building task, the index is created by reading the records from the data sets to be matched, then the records are stored in a data structure in a way that brings similar records closer together (by grouping or sorting the records using indexes).

On the other hand, the aim of the retrieval step is to retrieve records that are likely similar to each other (or similar to a query record in the case of real-time ER), and generate the candidate record pairs to be compared in the comparison step of the ER process. Various indexing techniques have been developed in previous years (major techniques are surveyed in [34, 57, 96]). An indexing technique can fall into one of two main categories (shown in Figure 3.1): blocking-based or sorting-based techniques. The following sections provide a detailed description of some of the main current indexing techniques that are used in ER.

3.2 Blocking Based Indexing

In blocking-based indexing, the basic idea is to group records into blocks, where every block contain records that are likely to be similar. Records that are in the same block are compared in the comparison step to be classified (in the classification step) as matches and non-matches. Major indexing techniques that fall into this category are described next:

3.2.1 Traditional Blocking

The traditional blocking was introduced in [62, 87]. In this technique records are grouped together based on the value of one or combination of record attributes. These attribute values are used to segregate records into blocks where each block contains only similar records. For example, if a ‘Postcode’ attribute is used as the blocking key, each generated block will contain only records that have the same

Postcode. The attributes that are used in the blocking process are called Blocking Keys (BK). The aim of segregating records between blocks is to avoid comparing all records in a data set, rather compare only records that are in the same block (which are most likely to be similar). This will reduce the comparison space.

Data in the real world usually contain errors and variations. To make sure that similar records fall into the same block even if they contain some errors or variations, attribute values can be converted into phonetic codes [33] (using encoding functions) before going through the blocking process. The aim of using such encoding functions is to make sure that similar records with typographical errors and variations will be inserted into the same blocks. Several encoding functions, like Soundex, Phonex, and Double- Metaphone, can be used before blocking, such functions are described in [33, 82].

The number of records that are inserted into each block depends on the frequency distribution of the attribute values that are used as BKs [34]. For example, if the family name attribute is selected as a BK, more frequent names (e.g. 'Smith') will have larger block sizes, while a less frequent name (e.g. 'Herzog') will be in smaller blocks. Large block sizes affect the efficiency and scalability of the ER process.

To avoid generating large block sizes Gu and Baxter [68] proposed an adaptive blocking technique that aims at filtering large block sizes. The authors proposed two filtering approaches; the first approach is based on the length of a filtering variable that is used to remove record pairs that are unlikely to be a match from the list of candidate records. If the difference between the length of the filtering variables of record pairs is greater than a specific value these pairs are removed from the list of candidate records. The second filtering approach is based on the count of common bi-grams (sub-strings of length 2) between the filtering variables of record pairs. If the number of common bi-grams in the filtering variables of record pairs is smaller than a specific threshold, the corresponding record pairs are removed from the list of candidate records. This approach managed to reduce the number of candidate record pairs with the cost of a slight decrease in matching quality.

More recently, Fisher et al. [64] have addressed the issue of large block sizes by proposing two iterative blocking approaches that control the size of the generated blocks. The idea is to split large blocks and merge small blocks until all generated blocks are within a specified size range. The first approach merges and splits blocks based on the decreasing similarity of the generated blocks, while the second approach merges and splits blocks based on the increased size of the generated blocks. This approach also managed to control (using a penalty function) the trade-off between the size and quality of the generated blocks.

Another issue to consider with traditional blocking is the quality of the attribute values that are used as BKs. If the values of attributes selected as BKs have large numbers of missing values or errors and variations, this can lead to inserting records into the wrong blocks which affects the quality of the matching process. To overcome this issue, multi-pass blocking [87] can be applied where multiple passes of the blocking process are performed using different BKs to improve the quality of the matching process. Iterative blocking [160] can also be applied where blocks are pro-

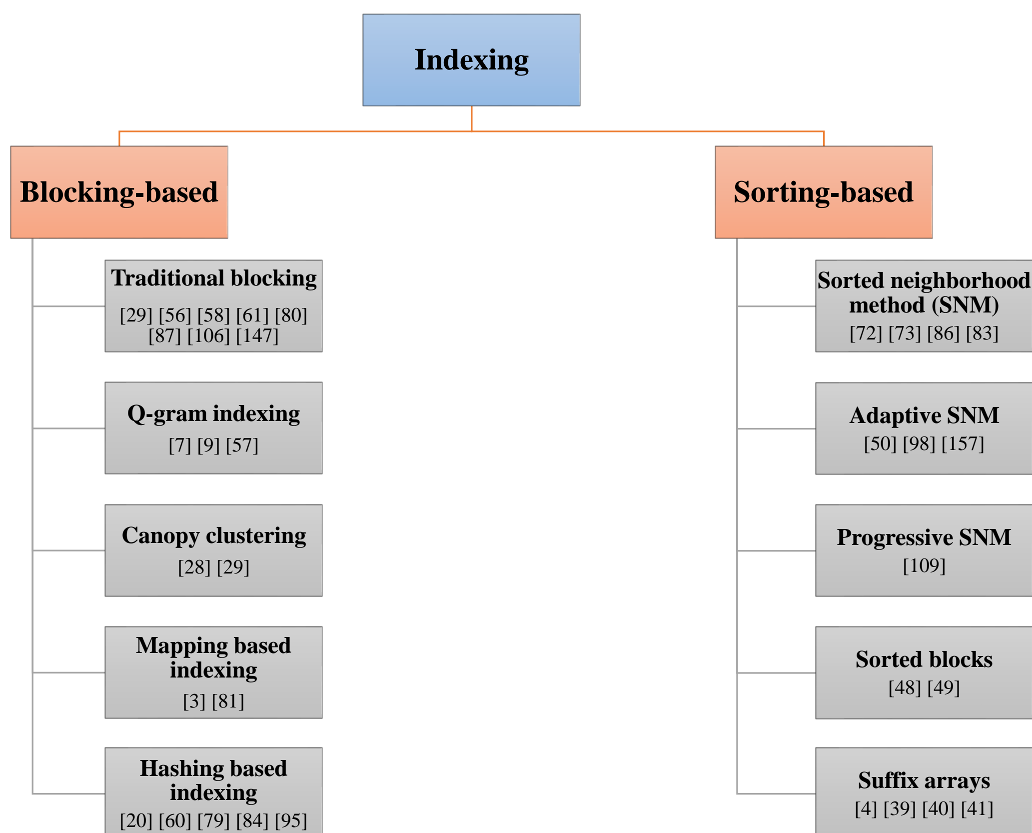


Figure 3.1: Main categories of existing indexing techniques that are used with traditional ER.

cessed iteratively using multiple BKs. In this approach, matched (resolved) records in blocks are distributed to other blocks and a record can be matched against multiple blocks which improves matching quality compared to disjoint methods (where records are only inserted into one block).

Although multi-pass and iterative blocking techniques improves the effectiveness of the ER process, they often affect the efficiency of the matching process because of the increased number of comparison and the redundancy introduced in the generated list of record pairs. The redundancy problem was addressed in [117] and [95]. In [117] the authors identified and discard blocks with redundant comparisons, and merged overlapping blocks which resulted in a block of fewer comparisons. Their solution managed to discard redundant comparisons at the cost of quadratic space complexity. Unlike [117], which works with non-distributed environments, the work presented in [95] addresses the redundancy problem in a parallel environments for MapReduce [46] (a data processing tool in a parallel environment) based approaches. Their technique managed to efficiently identify redundant pairs which are not compared at run-time.

For traditional blocking to be used with dynamic data sets, the data structures used to build the index must have the ability to be updated (add, delete, or edit val-

ues). This allows the blocks to dynamically grow when the data sets grow. To achieve real-time ER using traditional blocking, the used data structures must facilitate fast retrieval for records. In addition, block size must be small to make sure that the number of generated candidate record pairs is small and record pair comparisons can be handled in real-time. In Chapter 5 we propose a dynamic blocking-based indexing technique that is updated whenever a new query record arrives to facilitate query matching in real-time.

3.2.2 Q-gram Indexing

The idea behind the q-gram blocking technique, proposed in [11], is to convert the value of the blocking key (BK) into a list of q-grams (sub-strings of length q). Based on these generated lists of q-grams, each attribute is inserted into more than one block to reduce the effect of errors that might occur in attribute values. The results in [11] showed that the q-gram index outperformed the standard blocking and the sorted neighborhood methods with regard to matching quality.

Following [11], Ferro et al. [63] proposed a disk-based ER approach using q-grams. The authors presented an indexing technique that is based on a sorted list of q-grams (*q-gram array*). For each record in the data set, all distinct q-grams are generated and inserted into the sorted q-gram array (which is located in secondary memory). Then a bitmap table is generated for every q-gram in main memory. These bitmaps are used to perform the comparison and clustering steps. The results in [63] showed that the time required to build the index is linear with the size of the data set and the number of attributes. The results also showed that their approach achieved slightly improved matching quality compared to the q-gram indexing approach from [11] while improving the time required to build and query the index. The approach also outperformed traditional blocking [62, 87] with regard to matching quality but with the cost of efficiency reduction. A major drawback of this approach is the memory size required to store all generated bit maps.

An identity matching system that is based on using q-gram indexing was proposed in [8]. The authors aimed at improving the efficiency of the q-gram based indexing technique [11] by pruning q-grams with high frequencies and those that contain only one record. To perform the identity matching, their algorithm adds all records from the data set into a pool of records P, then selects a random record r_c from P. Next a list of q-grams Q is generated for the blocking key value of r_c. A pruning parameter l_d is then calculated for Q: $l_d = min(|Q| + (max(|Q|) + min(|Q|)) * t_d$, where t_d is a uniform term-centric threshold [8]. If $|Q| < l_d$, then Q will be included in the matching process. The parameter l_d ensures that only a small number of q-gram blocks are processed to improve matching efficiency. The results showed that their approach outperformed another automatic identity matching system from [153] (which uses the sorted neighborhood method [79] as an indexing technique) with regards to effectiveness and efficiency.

Although q-gram based indexing achieves better quality blocking (i.e. more true matches) than traditional blocking, it is computationally expensive [11]. This is be-

cause the number of generated candidate record pairs will be larger than those generated using the traditional blocking technique. This makes using q-gram based indexing techniques for real-time ER challenging.

3.2.3 Canopy Clustering

Canopy Clustering [107] is a technique that aims at speeding up the process of blocking records for large databases. This technique first inserts the data into overlapping subsets called *canopies*, then these canopies are used to create the blocks that contain the candidate records. This technique first builds an inverted index for the BK values of all records. Before BK values are inserted into the inverted index they must be converted into a list of tokens (tokens can be q-grams or words). The generated tokens are then used as keys in the inverted index; then all records from the database are inserted into the inverted index based on the tokens of attribute values.

After adding all records from the database to the inverted index, overlapping clusters (canopies) are generated by adding all records from the inverted index into one set S. From S a record r_{center} will be randomly selected and considered as the center of a new canopy C. Then, the similarity between r_{center} and all other records in S are calculated using Jaccard or TF-IDF/cosine similarity [33]. All records that have a similarity with r_{center} that is above a low threshold t_{low} will be added into the canopy C. Next, r_{center} will be removed from S, along with all records within C that have a similarity above a high threshold t_{high}, with $t_{high} > t_{low}$. This process will continue until there are no more records left in S, where all records will be divided between the clusters. Canopy clustering is shown to be scalable to large databases [107].

Christen [32] has modified this approach by replacing the two global thresholds with two neighboring based parameters: n_{loose} and n_{tight} where $n_{tight} \leq n_{loose}$. The first parameter represents the number of record identifiers that are inserted into each cluster while the second parameter represents the number of record identifiers that are removed from the pool of candidate records in each step of the algorithm. This modification resulted in generating blocks of similar sizes with a maximum size that is known (which is equal to n_{loose}). This approach has the drawback of missing true matches if these two parameters were not set properly. However, the results in [32] showed that the modified canopy clustering approach achieved better matching quality compared to the original approach as proposed in [107].

3.2.4 Mapping Based Indexing

StringMap [88] is one of the major mapping-based indexing techniques. The idea in this approach is to convert blocking key values into objects that are mapped into a multidimensional Euclidean space. Then, similar objects are grouped into the same cluster by applying a multidimensional similarity join in the Euclidean space. This approach was modified in [3] by using two levels of mapping. The first level of mapping is similar to [88] where blocking key values are mapped into a multidimensional Euclidean space, followed by a second level of mapping into a second lower-

dimensional metric space using edit distance. The authors applied a KD-tree and a nearest neighbor-similarity approach to insure efficient matching. This approach improved the speed of runtime by 30% to 60% compared to StringMap [88].

3.2.5 Hashing Based Indexing

Using hashing based indexing for similarity search was introduced in [66, 84] to solve the problem of high-dimensionality (records with a high number of features that could reach thousands) and to speed up similarity search in the approximate nearest neighbor search problem [7].

Generally, Locality Sensitive Hashing (LSH) is a popular hashing approach that is used to address indexing objects with high dimensionality [66, 84]. The basic idea is to use LSH functions to hash records, where the values of attributes (features) are converted into a set of binary numbers (bit-pattern). These patterns are then used to group records into buckets (blocks) based on their hashed values. With LSH, records that hash into the same bit-pattern are more likely to be similar. This means that similar records have a high probability to fall into the same block [66, 84].

In context of ER, several hashing approaches have been proposed to improve the efficiency of the matching process [86, 92, 104]. Kim and Lee [92] have proposed an iterative locality sensitive hashing technique (I-LSH) that dynamically merges the generated hash tables for quick and accurate blocking. The approach takes a data set as input, then the I-LSH process iterates through the data set where it generates LSH-based hash tables for the records from the data set and the I-LSH function continues to reduce the size of these hash tables and the generated set of records in each iteration until the generated tables achieve a reduction rate that is less than a given threshold. The results in [92] showed that their approach achieved better efficiency results while maintaining similar matching quality compared with several other approaches (basic LSH [66], StringMap [100], and R-Swoosh [13]).

Ioannou and Papapetrou [86] proposed an ER approach that works with RDF representations of sources on the Semantic Web, taking into consideration the semantics and structure in the resource descriptions. Their approach employs an index that is based on the LSH approach of [66] to avoid the need of large number of pairwise similarity computations that is usually required to find matching entities. Their results show that for an RDF data set with 2,711,217 entity, an average query time of less than 100 ms was achieved.

Malhotra et al. [104] proposed a parallel implementation of the ER process using MapReduce [46]. The approach combines the iterative match-merge (IMM) approach proposed in [13] with the standard LSH hashing from [23] for the purpose of improving the efficiency of the ER process. The approach managed to avoid unnecessary comparisons to reduce the required query time. For a hashing-based indexing technique to be suitable for query matching on dynamic data sets, it should be able to add hashed query records into the corresponding bucket in each hash table, retrieve candidate records and conduct the matching process within sub-second time.

3.3 Sorting Based Indexing

Sorting-based indexing techniques aim at bringing similar records closer to each other by sorting all records in a data set based on a criterion which is usually the value of one or more attributes. For instance, sorting a list of customers alphabetically based on the value of their first name will bring customers with first names that start with the same letter closer to each other. The aim is to then compare only records that are most similar. Major sorting-based indexing techniques are described below.

3.3.1 Sorted Neighborhood Method

The idea behind the sorted neighborhood method (SNM), which was introduced in [79, 80], is to sort the records in the compared data sources based on a sorting value. This sorting value is one or several of the record attributes. Then a fixed size window ($w > 1$) slides over the sorted records and only the records inside the window at any one time are compared. This will reduce the number of compared record pairs significantly.

This approach first merges the records from the compared data sets and loads all records into a data structure (the original approach [79, 80] uses a static array). The records in the array are sorted alphabetically based on a sorting key (i.e. an attribute value). One thing to consider when selecting a sorting key is that the sorting will be sensitive to the beginning of the sorting key values. This means if the sorting key value contained an error at the beginning it will not be placed close to similar records. To avoid this problem a multi-pass SNM approach can be used with different sorting key [79] to independently run the SNM several times. For example, in the first run a Surname attribute can be used, while in a second run a Suburb attribute can be used as sorting key. Each of the two runs will generate a set of candidate records that are merged. Then the transitive closure [33] is applied to those record pairs. The results will contain the union of all unique record pairs discovered by all independent runs, plus all record pairs that are found by applying the transitive closure. This approach improves the matching quality with the cost of an increase in matching time.

Both single and multi-pass SNM can be modified to run in a parallel environment to improve the efficiency of the SNM approaches. Kolb et al. [94] proposed an implementation of both single and multi-pass SNM using the MapReduce programing model [46]. The authors investigated the challenges and solutions of using MapReduce to perform the ER process in parallel using the SNM. Their results showed that single and multi-pass SNM can be parallelized with MapReduce to improve the efficiency of the ER process. The complexity of single and multi-pass SNM is discussed in details in [90].

Unlike blocking-based techniques, the number of candidate record pairs generated by the SNM is not affected by the frequency distribution of attribute values that are used as sorting keys, and only depends on the size of the sliding window and number of records in a database'. It is important to choose the optimal window size

that finds as many true matches as possible. If the window size is too small, true matches might be missed; while if it is too large unnecessary comparisons between record pairs that are not similar might occur. This problem was addressed by different approaches [54, 109, 170] that use an adaptive window size instead of a fixed one. These approaches are described in the following section.

The SNM in its current form is only suitable for indexing static data sets and only works with ER techniques that match multiple data sets or a data set with itself. In addition, it dose not work with real-time ER where streams of query records are matched with an existing data set. However, in Chapter 6 we propose a dynamic sorted neighborhood index (based on the traditional SNM) which can be used with query-based matching for dynamic data sets to facilitate real-time ER, and in Chapter 7 we propose a dynamic multi-tree index (based on the traditional multi-pass SNM) which is tailored for real-time ER.

3.3.2 Adaptive Sorted Neighborhood Method

Because selecting an optimal window size that can achieve the best matching results with a minimum number of comparisons is challenging, adaptive SNM techniques can be used to avoid the need for manual tuning of the size of the window that is used to generate the set of candidate records. Several adaptive SNM approaches have been proposed [54, 109, 170].

Yan et al. [170] proposed an approach that adaptively adjust the boundaries of the window used to generate the candidate records based on the similarities between the values in the sorted list. This approach expands the window based on the distance d between the first and last records in the initial window size. A distance that is less than a specified threshold ($d < \theta$), means that the records within the window are close to each other and there is room for expanding the window size to include more records. A distance that is $d > \theta$ means that the records within the window are spares and the size of the window can be reduced to avoid unnecessary comparisons. The results presented in [170] showed that their adaptive window approach achieves better matching quality compared to the original fixed size window approach in [79].

Draisbach et al. [54] proposed another adaptive SNM that aims at addressing the disadvantage of having a fixed size window. In this approach the expansion and reduction of the window size is based on the number of matches that are found within the window. The more matches are found, the larger the window size becomes. However, if no matches for a record are found in the window, this approach assumes that there are no more matches (based on the fact that all records are sorted and similar records are closer to each other), which means there is no need to increase the size of the sliding window.

This approach has two modifications, the first modification starts with a fixed size window w. Then the window size increases by one record as long as the ratio between the number of detected duplicates d and the number of comparisons c within the window w is equal or grater than a threshold $0 < \theta \leq 1$. The second

modification increases the window size by adding for each detected duplicate the next $w-1$ neighboring records of that duplicate even if the condition $\frac{d}{c} \geq \theta$ does not apply. The results presented in [54] showed that using the adaptive SNM is more efficient than the original SNM from [79].

To improve the efficiency of the adaptive SNM techniques, they can be modified to run in a parallel environment. Mestre et al. [109] proposed a parallelization of the adaptive approach from [54] (the second approach described above) using MapReduce [46]. They also compared their adaptive MapReduce-based SNM approach with a state-of-the-art MapReduce-based SNM [94] that uses a fixed window size, their results showed that their adaptive MapReduce-based approach outperforms [94] with regard to the overall execution time.

Similar to the original SNM, these adaptive approaches (described above) work only with static data sets, where the index remains unchanged and new records cannot be added. Although these approaches provide an adaptive and dynamic window size that can change at run time, they only work with batch-processing ER (where full data sets are matched with each other) but not with query-based matching of dynamic data sets. Therefore, these approaches are not suitable for real-time ER. In Chapter 6 we propose several adaptive window size approaches that are tailored for query-based matching and that are shown to be suitable for real-time ER.

3.3.3 Progressive Sorted Neighborhood Method

Recently, Papenbrock et al. [120] proposed a progressive sorted neighborhood method (PSNM) that aims at increasing the efficiency of the ER process and maximizing the gain from the ER process by reporting most of the matching records earlier than traditional techniques [79, 80]. In the original SNM [79], the algorithm sorts the records of a data set based on a sorting key, then a fixed size window slides over the whole sorted records comparing only records within the window at any one time.

In the PSNM, after sorting the records within a data set, the sorted records are divided into equal size partitions, the size of the partitions is calculated based on the available main memory. Then, based on the intuition that records which are close to each other, in the sorted records, are more likely to be matches than records that are far a part from each other, the PSNM aims to compare the most promising records (that are close to each other) in each partition first then the records that are far from each other. This is achieved by first using a window of a small size (size two) for each partition and the records within the window are compared. Then the window size iteratively increases until it reaches a maximum size to find the less promising records (that are far from each other) from each partition. The PSNM approach can also dynamically change the execution order of the comparisons of the records based on intermediate results.

The results presented in [120] showed that the PSNM can double the efficiency over time of the original SNM [79]. The results also showed that the majority of the matching records were reported early in the ER process.

3.3.4 Sorted Blocks

Draisbach and Naumann in [52, 53] proposed a technique that presents a generalization of the traditional blocking and traditional SNM techniques where both techniques can be adapted in a way that allows having the same number of comparisons and the same pair comparisons (i.e. the same record pairs are compared).

In the traditional SNM the window starts at the beginning of the sorted list and slides by one position each time, comparing all records within the window and resulting in having overlapping sorted blocks. While in the sorted blocks technique, the window slides forward by w positions resulting in non-overlapping sorted blocks. This approach also allows controlling the overlap between the sorted blocks. The results in [52, 53] showed that the sorted blocks technique outperforms the adaptive SNM proposed in [170] as it needs fewer number of pair comparisons to find the same number of duplicates. The results also show that the traditional SNM performs better than traditional blocking especially for small blocks.

The sorted block approach, as described in [52, 53], is designed for batch processing ER algorithms where full and static data set(s) are matched to find duplicate record and is not suitable for query-based matching that is needed for real-time ER.

3.3.5 Suffix Arrays

This approach is based on using a sorted suffix array [4] of all the subsequences of tokens that appear in a string. These tokens are used as blocking keys allowing records to be inserted into different blocks. The suffix array approach aims at reducing the effect of errors and variations at the beginning of a BK value.

In this technique, the BK value is converted into a set of possible suffixes. For example, the string 'yusuf' will generate the following suffixes: 'yusuf', 'usuf', 'suf', 'uf',and 'f'. The generated suffix strings will become keys in the suffix array index, and records are inserted into the blocks of keys that are included in their attribute values. This means that each record can be inserted into one or more blocks based on its generated suffixes.

If the generated suffix is short (e.g. 'f' from the above example), too many records will be inserted in this block. To overcome this problem, two thresholds are used: the first threshold is the minimum length of the generated suffixes l_{min}, and the second threshold is the maximum block size b_{max}. Using the l_{min} threshold will remove the large blocks that are generated for the shorter suffixes. Moreover, using the b_{max} threshold aims at pruning larger block. Either by deleting all blocks that contain a number of records that is larger than b_{max}, or by reducing the size of large blocks by deleting records that have the longest original BK value.

Different variations of the original suffix array technique [4] have been proposed such as [43, 44, 45]. De Vries et al. [44, 43] have proposed a robust suffix array approach that groups the sorted generated blocks (generated from suffixes) using a sliding window approach. Then, similar blocks are merged at marginal extra cost resulting in a higher accuracy while retaining efficiency compared to the original suffix array in [4]. The robust suffix array technique was extended in [45] where a

disk-based technique that combines the suffix array approach and Bloom filters [20] is presented. The aim of the Bloom filter is to act as filtering step to save disk read operations which are required when building the index data structure. Their presented results showed that with Bloom filters the disk read operations were reduced by up-to 70%.

A drawback of the suffix array is its sensitivity to errors and variation that occurs at the end of the compared strings. This approach also generates more blocks compared to traditional blocking which makes it more complex. In its current form Suffix Array indexing works only with static databases, and once the index is built it cannot change. However, if it is modified to insert new records into the generated blocks using the original approach of creating suffixes then it can be used with dynamic databases. Suffix-based indexing is more complex than traditional blocking and generates larger number of candidate records which makes it more challenging to be used with real-time ER.

3.4 Meta-Blocking Techniques

Indexing techniques with overlapping blocks (such as suffix array, q-gram based indexing, SNM, hashing-based indexing, and iterative blocking) aim at improving the matching quality by inserting a record into multiple blocks. This process can cause redundancy in the comparison step where record pairs are compared more than once which affects the efficiency of the matching process. Meta-blocking approaches, introduced in [118], can be used on top of any blocking (indexing) technique to overcome this issue. It is considered as an intermediate step between generating the blocks and matching record pairs within the generated blocks and aims at removing the redundancy that accompanies overlapping blocks (note that meta-blocking does not replace but complements existing blocking techniques). The input for meta-blocking approaches is a set of generated blocks (that are generated using any indexing technique with overlapping blocks) and the output is a new set of blocks with fewer record pairs. The following meta-blocking techniques were recently proposed.

Papadakis et al. [118] have proposed a meta-blocking approach that aims at improving the efficiency of indexing techniques that produces overlapping blocks. This is achieved by discarding redundant comparisons for record pairs within the generated blocks. Assuming a set of blocks, **B**, every record (from the blocks in **B**) is mapped to a node in a *blocking graph* and every record pair (records that are compared in at least one block) is linked with an edge. Redundant record pairs are identified during the creation of the blocking graph. If a record pair is linked with an edge in the graph they cannot be linked (to each other) again with a new edge. Therefore, each record pair (mapped from **B**) will be linked with at most one edge even if they occur multiple times in different blocks. The generated graph aims at eliminating redundant comparisons without having an impact on the effectiveness of the blocks.

The authors improved the efficiency of the blocking graph by giving weights to the edges between linked nodes, in a *weight blocking graph*, and then removing record pairs with low weights at a small cost in recall. The results presented by the authors showed that meta-blocking can improve the efficiency of the overlapping blocks generated in the indexing step of the ER process. To improve the efficiency of meta-blocking further, Efthymiou et al. [56] have proposed a parallelized variation of the meta-blocking approach based on MapReduce [46]. A supervised meta-blocking approach was proposed in [119] to enhance the performance of the unsupervised meta-blocking from [118]. This approach replaces the edge weights with feature vectors to generate a *generalized blocking graph*. The authors have proposed a set of generic features that combine low extraction cost with high discriminatory (the ability to distinguish between records).

Meta-blocking approaches are designed to reduce the number of record pairs (number of comparisons) produced from static overlapping blocks (which are used in batch processing ER). To be able to use meta-blocking with dynamic indexes in real-time, the blocking graph has to be dynamic to facilitate the dynamic nature of the blocks generated by indexing techniques that are used with real-time ER.

3.5 Techniques for Real-Time Entity Resolution

Existing ER techniques focus mainly on improving the accuracy and efficiency of the ER process. However, the majority of these techniques are aimed at off-line entity matching (using batched algorithms) of static databases.

The first query-time ER approach was based on a collective classification approach [16]. The idea behind this approach is to only use a subset of records in a database for resolving queries, by extracting records related to a query and then resolving this query using only these records. Although this approach can improve matching quality, experiments showed an average time of 31.28 sec was needed on a database with 831,991 records. Thus, this approach is not suitable for real-time ER, nor it is scalable to large databases since it is computationally expensive.

Christen et al. [35] proposed a similarity-aware inverted indexing technique that is suitable for real-time ER. The main idea behind this technique is to pre-calculate similarities between attribute values that are in the same block. These pre-calculated similarities are stored in main memory to be used later in the query matching process. Avoiding similarity calculations at query time significantly reduces the time required for matching a query record. This technique was shown to be two orders of magnitude faster than standard blocking [62, 87], which makes it suitable for real-time ER. However, this technique is static and once the index data structures are created, new records and attribute values cannot be added to the index.

Another real-time ER approach that is also designed to work with static databases was proposed by Dey et al. [48]. It is based on using a matching tree to limit the amount of communication required for matching records between disparate databases held at different locations, where a matching decision can be made without the need

of comparing all attribute values between records. This approach was shown to reduce the communication overhead, without affecting the matching quality.

Ioannou et al. [85] on the other hand proposed an approach that provides ER in real-time for RDF dynamic databases. Their method is based on using links between the entities in a database combined with a probabilistic database for resolving entities. The approach uses existing ER techniques to find possible matches of a query, and instead of using these possible matches to make an off-line resolution decision, it stores the possible matches alongside with a probability weight in a dynamic index data structure. 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 70 ms for a query record on a database of 51,222 records. This query time is almost constant and does not increase when the database get larger.

Another dynamic ER approach is proposed by Whang and Garcia-Molina [158] that allows matching rules to evolve over time when new records become available. This approach aims at using materialized ER results (which are a set of records that are classified as matches) to save redundant work, and does not require running the ER process from scratch. The authors report that this rule evolution approach can be faster than the naive approach by up to several orders of magnitude [158].

Whang et al. [159] proposed a pay-as-you-go ER technique that can be used with real-time ER. The authors build their technique on top of the indexing step (before record comparison and classification steps). This approach propose the use of *Hints* to give information on records that are likely to refer to the same real-world entity. The aim is to order candidate record pairs (that are generated in the indexing step) by the likelihood of a match. Then, in the comparison step, the records that are more likely to be a match will be compared first. The results in [159] showed that using Hints before the comparison step improves the ER process by finding the majority of matching records within a fraction of the total runtime at the cost of an overhead in time and space. However, the authors also proposed a trade-off between time overhead and the benefit of using Hints.

Lately, Rezig et al. [133] proposed a general ER framework for on-line query-matching which is based on iterative caching. The idea of their approach is to de-duplicate and cache a set of frequently requested records that are obtained from different Web databases using sampling. These de-duplicated record pairs are used for future reference when new queries arrive. When a new query arrives, it is matched jointly with the cached record pairs and then it is added to the cache. The result presented by the authors showed that their approach provides a fast and effective ER framework that can be employed with on-line settings.

The techniques reviewed above provide general ER frameworks that can be used in real-time. Nevertheless, none of those techniques (except the indexing technique proposed in [35]) focuses on the indexing step of the ER process. However, as discussed in Sections 2.2.2 and 2.3.2, we believe that indexing is a vital step for the real-time ER process, and creating new indexing techniques that are particularly designed to work with real-time ER allows conducting the real-time ER process using

existing classification and comparison techniques that are designed originally for traditional ER. In our research we focus on creating new and novel indexing techniques that are tailored for real-time ER which is vital for making the matching process efficient and suitable for real-time.

3.6 Automatic Techniques for Learning Blocking Keys

Selecting blocking and sorting keys is a critical aspect for the indexing step as it affects the efficiency, scalability, and the quality of the ER process. An optimal blocking/sorting key aims at grouping similar records into the same block or closer to each other in the sorted data sets [33] to include as many true matches as possible while keeping the set of candidate record pairs as small as possible [33].

Real-world data sets are dirty and contain errors and variations which can affect the process of selecting blocking/sorting keys. In addition, selecting optimal keys can be domain dependent, and several potential blocking keys can be suitable over multiple record attributes [17]. This makes manual blocking key selection challenging. Most existing indexing techniques (such those discussed in the above sections) require manual blocking key selection by an expert. An alternative would be to learn blocking/sorting keys automatically.

Various automatic techniques have been proposed that allow learning optimal blocking/sorting keys based on supervised learning which requires the use of gold standard data for training. The training data sets consist of record pairs that are labeled as true matches or true non matches and can be used by the learning algorithms, as examples, to find which keys achieve high quality blocks [33].

Bilenko et al. [18] proposed an approach that deals with the learning process as an approximation problem that is based on the red-blue set cover problem [27]. The authors investigated using a Disjunctive Normal Form [5] of different length keys to be used in creating blocks. The aim was to discard blocking keys that cover many negative pairs (that are classified as true non-matches in the training data set) and to select keys that covers the most possible positive pairs (that are classified as true matches in the training data set). The authors compared their learning algorithm with the canopy clustering approach from [107] and the results showed that their learning approach achieved better matching quality compared to canopy clustering with the cost of an increase in the average matching time.

Michelson and Knoblock [110] proposed an approach for learning *blocking schemes*. They defined a blocking scheme as a pair of an attribute and a comparison method. Their aim was to learn which attributes are more suitable as blocking keys, and which methods should be used for comparing these attributes. The authors used a modified version of the Sequential Covering Algorithm which originally learns disjunctive sets of rules from labeled training data sets [111]. In their case, they have learned a conjunction of blocking schemes (a combination of {attribute, method}) that maximize the reduction ratio (the ratio between the generated candidate record pairs and all possible record pairs). Their algorithm automatically adds {attribute, method} pairs to conjunctions until the reduction ratio stops improving. The results

presented in [110] showed that their learning algorithm outperforms manual selection of blocking schemes produced by non-domain expert, and is comparable with schemes selected manually by a domain expert with regard to matching quality and efficiency.

Another supervised approach was recently proposed by Vogel and Naumann [152]. The authors use unigrams of attribute values (i.e. a combination of single characters from different attributes) as blocking keys. Both accuracy and efficiency of the generated blocks are used to learn the set of optimal blocking keys. They also improved their approach by taking the length of attribute values into consideration when generating the unigrams to be used as keys.

Das Sarma et al. [42] proposed an automatic hierarchal tree-based blocking system for ER that executes in a distributed setting (like the MapReuce framwork [46]). Their work presents an automatic disjoint blocking function that partitions records in a data set in a hierarchical fashion by successively applying blocking functions. The hierarchical tree is build using training data, a set of blocking functions, and a block-size estimates. Each leaf node in the tree corresponds to a block and the path (from the root) to a node corresponds to a blocking function. An optimal blocking function is picked greedily by counting, for all blocking functions, the number of pairs that get eliminated on choosing that blocking function. The blocking function that minimize the number of eliminated pairs is selected. The authors compared their approach with conjunctive blocking [18, 110] and the results show that the approach proposed by Das Sarma et al. [42] outperforms conjunctive blocking with regard to quality and efficiency.

All of the above automatic approaches require labeled training data. However, such labeled data is not always available and is usually expensive to generate. To overcome this problem, several unsupervised automatic blocking key selection techniques have been developed [26, 65, 89, 103, 143].

Cao et al. [26] proposed employing unlabeled data generated using a sampling technique alongside labeled data. The authors utilized a modified version of the sequential covering algorithm from [110] with the difference of having the unlabeled data to be incorporated into the learning process in order to improve optimal key selection. The authors compared their approach with the learning algorithm presented in [110] and the results showed that using unlabeled data alongside labeled data in the learning process can reduce the number of candidate matches.

Song and Heflin [143] proposed a general unsupervised automatic key selection technique for structured data. The authors used two measures to select optimal keys: discriminability (the ability to distinguish between different entities) and coverage (the number of record pairs that evaluate to the same key value). The aim was to iteratively select a set of discriminating predicates (blocking keys) with high coverage. They calculated an F1-score based on the two measures $F1 = \frac{2*discriminality*coverage}{discriminality+coverage}$. Keys with low discriminality are ignored and those with F1-score values that are above a threshold α are considered for selection (the key with the highest F1-score is selected). If there were no keys with F1-score that are above the threshold α the

algorithm combines the key that has the highest discriminability with every other key to form virtual keys instead of the old keys for the purpose of finding a suitable key. The authors compared their approach with automatic blocking techniques [17, 110] and non-automatic techniques [169, 164] using three RDF data sets. The results showed that their unsupervised blocking algorithm outperformed baselines in two of the data sets with regards effectiveness and efficiency.

Kejriwal and Miranker [89] proposed an unsupervised algorithm for learning blocking keys to be used with indexing techniques. The algorithm consists of two phases. In the first phase, the algorithm generates a weakly labeled training data set using a TF-IDF weighting scheme. In the second phase the algorithm uses the generated labeled training data sets to learn the optimal blocking keys using a Fisher discrimination criterion [55]. This Fisher score is used to select the optimal blocking key (the key with the highest Fisher score). The algorithm considers key coverage (which is defined as the number of record pairs that evaluate to the same key value) when calculating Fisher scores and selecting optimal blocking keys. The authors compared their learning algorithm to both unsupervised approaches presented in [17] and [110] and the results showed that their approach outperformed the supervised blocking key learning approach proposed by Bilenko et al. [17].

Giang [65] proposed a technique that learns the blocking keys in context of the classifier function that is used in the classification step of the ER process. The classifier is used to generate labeled data. The author consider learning the blocking keys as a Disjunctive Normal Form problem [5] and uses the Probably Approximately Correct [73] approach to learn the blocking keys. The results in [65] showed that their approach achieved comparable quality results with manual key selection by an expert and faster execution time of around one order-of-magnitude.

Unlike the approaches described above (that considers learning blocks for entities of similar types) Ma et al. [103] proposed an approach that is based on type semantics, where the authors consider types and sub-types of entities when learning the blocking keys for data from the web (for example the type product can have subtypes such as a camera, an e-reader, a phone, etc). The idea was to learn specific keys for specific sub-types of entities in the data to improve the quality of the matching process. The authors compared their approach with other blocking techniques like [85] and [143] and the results show that their approach improved the quality and efficiency of the matching process.

Most available blocking key learning approaches mainly focus on the quality of the generated blocks. However, a blocking key that can be used with real-time ER must ensure that the generated blocks are small enough and are similar in size to be able to resolve queries in real-time. In Chapter 8, we propose an automatic blocking key selection technique that can be used for real-time ER. Our approach considers the coverage, the maximum and average size, as well as the distribution of the sizes of the generated blocks when learning the blocking keys.

3.7 Summary

In this chapter we have provided a review of various existing indexing techniques that are used in the ER process. The majority of these techniques are static and are designed to work with batch processing ER where single/multiple static data source(s) are matched. Such techniques are not suitable for query-based matching where a query record is matched with dynamic data sets in real-time. Not much research has been conducted on real-time ER, and few indexing techniques are available that are specifically designed to work with real-time ER. There is a need for novel dynamic indexing techniques that are tailored for real-time ER.

Moreover, available automatic blocking key selection techniques are not specifically designed to learn blocking keys that are suitable for building indexes to be used with real-time ER. Most existing automatic blocking key selection techniques focus on the quality of the generated blocks when learning blocking keys. However, to learn blocking keys that are suitable for use with real-time ER, the size and the frequency distribution of the generated blocks have to be considered. There is a need for new novel learning techniques that are specifically designed for learning keys that are suitable for use with real-time ER.

Chapter 4

Evaluation Framework

In this chapter we provide a description for the framework that we use in evaluating the approaches that we propose in Chapters 5 to 8. We first provide a description of the used evaluation measures in Section 4.1. Then, in Section 4.2, we describe how the record pair comparison and the classification steps are achieved. In Section 4.3 we describe the baseline approaches that are compared with our proposed approaches, and we describe the implementation environment and the data sets used to conduct the experimental evaluation in Sections 4.4 and 4.5 respectively.

4.1 Evaluation Measures

It is important to evaluate any developed indexing techniques to examine if they are suitable for use with real-time entity resolution (ER). We conduct our experimental evaluation with regard to the challenges of real-time ER as (described in Chapter 2): the quality (effectiveness), the scalability, and the efficiency of the ER process. The measures that we use are described in the following:

1. **Quality evaluation:** When evaluating the matching quality of an ER approach, two types of errors can be expected [36, 113]. The first error occurs when the ER approach falsely classifies candidate record pairs as matches while they are not actual true matches. These record pairs are called false positives (FP). The second error occurs when the ER approach falsely classifies candidate record pairs as non-matches while they are actual true matches. Theses record pairs are called false negatives (FN). Moreover, record pairs that are classified as matches and that are actual true matches are called true positives (TP), and record pairs that are classified as non-matches and that are actual non-matches are called true negatives (TN). For traditional static ER approaches, based on these error types described (summarized in Figure 4.1), the quality of the classified record pairs is commonly evaluated using recall and precision [113]:

    - **Recall:** The proportion of the actual true matches that have been classified correctly [36], calculated as:

$$Recall = \frac{|TP|}{|TP| + |FN|} \quad (4.1)$$

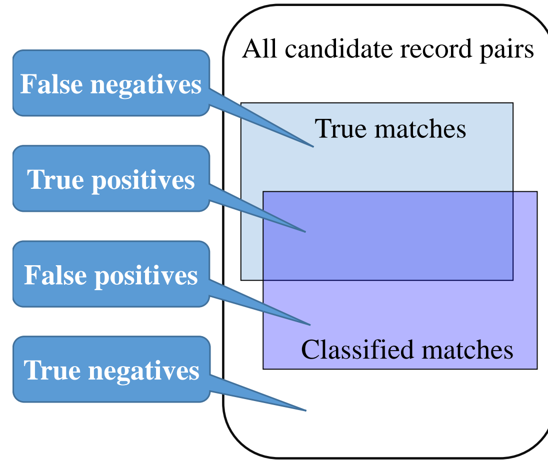


Figure 4.1: Types of errors in the ER process (taken from [113]).

- **Precision:** The proportion of the number of record pairs classified as matches which are actual true matches [36], calculated as:

$$Precision = \frac{|TP|}{|TP| + |FP|} \tag{4.2}$$

However, for real-time ER, when matching query records the ER approach generally returns the m top-matched records. These are selected from the set of candidate records $\mathbf{C}$ that are generated in the indexing step and ranked in a descending order based on the overall similarity with the query record as described in Chapter 2. This means that the returned ranked list of matching records, denoted as $\mathbf{M}$, will always have a size of $|\mathbf{M}| \leq m$ for all query records (where m is the number of top-matched records that are returned from matching a query record with the indexed data set). Thus, the precision value for a query record can be calculated as $|TP|/m$. Because m is constant, this measure will only reflect the number of the true positives that are found by the evaluated approach.

Moreover, the average number of true duplicates for a query is often small in relation to the number of records, including in the data sets that we use in our evaluation experiments (see Table 4.1). Therefore, using the precision measure for evaluating the quality is not useful in the case of real-time ER. Instead, other measures like the mean reciprocal rank (which is commonly used to evaluate information retrieval systems [75]) can be used to evaluate the overall performance of the matching process:

- **Mean reciprocal rank (MRR):** An overall relevant performance measure for the matching process. It calculates the average of the reciprocal of the

rank (rk_i) of the first true matching record in the returned result set $\mathbf{M}$ for each query record q_j in a query stream $\mathbf{Q}$ as:

$$MRR = \frac{1}{|\mathbf{Q}|} \sum_{i=1}^{|\mathbf{Q}|} \frac{1}{rk_i} \tag{4.3}$$

For example, given the following query records:

Query Record	Classified Matches	First True Match	Rank	Reciprocal Rank
peter	**peter**, pedro, pete, polo	peter	1	1/1
smith	smart, mith, **smith**, smint	smith	3	1/3
robert	rob, **robert**, roby, roben	robert	2	1/2

we calculate the MRR as (1 + 1/3 + 1/2)/3 = 0.61.

We will be using both recall and MRR to evaluate the quality of the matching process of our proposed approaches.

2. **Efficiency and scalability evaluation:** For evaluating the efficiency and the scalability of our proposed approaches we use two timing measures:

   - **Average insertion time:** The average time required to insert a query record into the index data structure.
   - **Average query time:** The average time required to match a query record with records in an existing index data structure.

   To evaluate scalability we investigate the effect of the growing size of the indexes (which are first build using a small number of records, then grow by adding millions of record) on both average insertion and query times to examine whether or not the proposed indexes scale with large data sets.

4.2 Record Pair Comparison and Classification

As described in Chapter 2, after performing the indexing step, a record pair comparison step is required to compare a query record $q_j \in \mathbf{Q}$ (where $\mathbf{Q}$ is the stream of query records to be matched) with all records in the set of candidate records $\mathbf{C}$ that is generated in the indexing step. Then a classification step is required to classify all records in $\mathbf{C}$ into matches and non-matches:

- **Record pair comparison:** We conduct the record pair comparison step by comparing attribute values of the compared record pairs. The Jaro-Winkler similarity function is commonly used in ER techniques to compare string values. It is specifically designed for comparing names (like persons or cities) [33]. We use this function to compare all attributes with string values. The Levenshtein edit

distance [114] calculates the minimum number of edit operations (i.e. character insertion, deletion and replacements) that are required to convert one string into another [114, 113] and is suitable for comparing short attribute values. We use this function to compare all numerical values in our data sets such as the 'Postcode' and 'Year' attributes.

- **Classification:** To classify candidate records in **C** into matches and non-matches, an overall similarity between the query record q_j and each $r_i \in \mathbf{C}$ is produced by calculating the sum of the similarities between all compared attribute values. The overall similarity is then normalized into a value between 0 and 1. A record $r_i \in \mathbf{C}$ is classified as a match with q_j if the overall similarity between q_j and r_i, denoted as $sim(q_j, r_i)$, is above a threshold t, where $0 \leq t \leq 1$. Records that are classified as matches with q_j are added to the set of returned matching results **M**. Selecting a high value of t can lead to missing true matches while selecting a low value of t can lead to falsely identifying true non-matching records as matches. However, since the returned matching results in **M** are ranked descending based on their similarities with the query record, and only the m top-matches are returned, selecting a value of t that is not too low should help to identify true matching records with less number of falsely identified matches. In all our experiments we use a threshold value of $t = 0.75$ (unless specified otherwise).

The pair comparison and classification details described above are used in our ER process for all proposed and baseline approaches. Note that the comparison and the classification steps are outside our research scope.

4.3 Baseline Approaches

As described in Chapter 1, in this thesis we focus on the indexing step of the ER process. In the next four chapters we propose four different approaches that are categorized into indexing and learning:

1. **Indexing:** In Chapters 5, 6 and 7 we propose three dynamic indexing techniques that works with real-time ER. As discussed in the Chapter 3, most of the available indexing techniques that are currently used to perform the ER process are static and solely designed for batched matching algorithms which work offline. This was a problem for selecting a proper baseline indexing technique that supports query-based matching which is required for real-time ER. Therefore, we use a q-gram indexing (QGI) technique which we generated by modifying the static indexing techniques from [11, 107] to produce a dynamic index that can be use with real-time ER, as described below, and we used this technique to compare with our proposed dynamic indexes.

   The **Q**-**G**ram **I**ndexing (**QGI**) technique is a q-gram based inverted index [11, 34, 107] that converts the attribute values of each record in a data set into a list of q-grams (sub-strings of length q). Each unique q-gram becomes a key

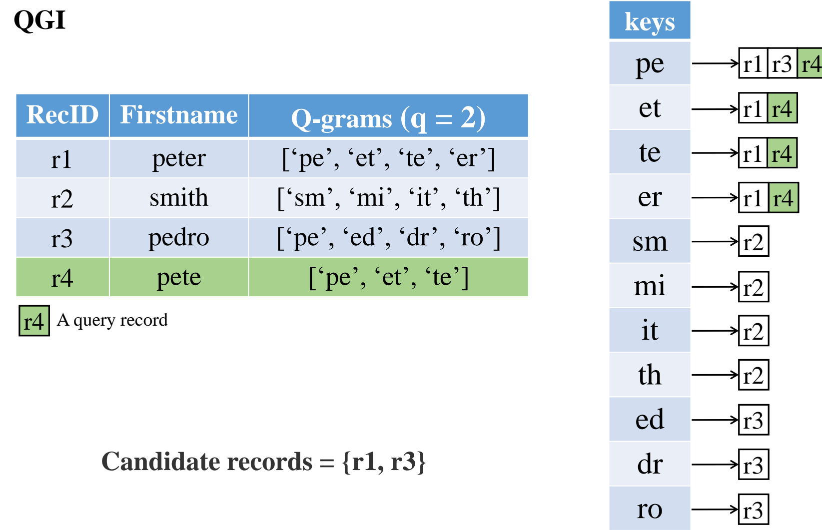


Figure 4.2: The Q-gram index (QGI) that is used as a baseline in Chapter 9.

in the inverted index where its corresponding value is the list of all records in the data set that have this q-gram in their attribute values. To 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 [33] with the query record that is greater than the minimum similarity threshold.

Figure 4.2 illustrate an example of the QGI approach. Assume that $r4$ is a query record and it is required to be matched with the existing index from the figure using an overall similarity threshold of $t = 0.75$. First, the 'Firstname' attribute value of $r4$ ('pete') will be converted into the following q-grams ['pe','et', 'te']. Then, the record identifier $r4$ of the query record is inserted into the inverted index by adding it to the value of all keys that are included in its q-gram list (i.e. 'pe', 'et', and 'te'). The list of candidate records ($\mathbf{C}$), that will be compared in detail with the query record, is generated by taking all the records that have at least one common q-gram with the query record. From the example in Figure 4.2 only $r1$, and $r3$ share common q-grams with the query record $r4$. These records are then compared in detail with the query record $r4$ by using the Jaccard-based similarity measure:

$$sim_{jaccard}(q_j, ri) = \frac{|Qgram(q_j) \cap Qgram(r_i)|}{|Qgram(q_j) \cup Qgram(r_i)|} \tag{4.4}$$

The jaccard similarity between the query record $r4$ and the candidate records in $\mathbf{C} = \{r1, r3\}$ are calculated as follows: $sim_{jaccard}(r4, r1) = 3/4 = 0.75$, and $sim_{jaccard}(r4, r3) = 1/6 = 0.16$. A record is considered a match if $sim_{jaccard}(q_j, ri) \geq t$. This means that only $r1$ is identified as a match to $r4$, when $t = 0.75$.

2. **Learning:** In Chapter 8 we propose an unsupervised learning algorithm that automatically selects optimal blocking keys for building indexes that can be used in real-time ER. We compare our proposed blocking key learning algorithm with the following baseline:

   The **F**isher **D**is**J**unctive (**FDJ**) technique is an unsupervised algorithm for learning blocking keys to be used with indexing techniques [89]. We selected the recently proposed **FDJ** approach as our baseline since it was shown in [89] to outperform two of the major supervised blocking key learning approaches proposed by Bilenko et al. [17] and Michelson and Knoblock [110]. This baseline algorithm consists of two phases. In the first phase, the algorithm generates a weakly labeled training data set using a TF-IDF weighting scheme to calculate the similarity between record pairs $(r_x, r_y) \in R$ as follows. A lower and upper thresholds $0 < l < u < 1$ are used to generate the training data sets. Record pairs (r_x, r_y) that have a TF-IDF similarity value $sim(r_x, r_y)$ below l are labeled as negative matches, and all pairs that have a TF-IDF value above u are labeled as positive matches (how TF-IDF values are calculated is described in more details in Chapter 8).

   In the second phase the **FDJ** algorithm uses the generated labeled training data sets to learn the optimal blocking keys using a Fisher discrimination criterion [55]. This Fisher score is used to rank the candidate blocking keys, then selects the optimal blocking key (the key with the highest Fisher score). The **FDJ** algorithm only considers key coverage (which is defined as the number of record pairs that evaluate to the same key value) when calculating Fisher scores and selecting optimal blocking keys. More details about generating the training data sets and the calculation of key coverage can be found in Chapter 8).

4.4 Implementation Environment

We implemented all proposed and baseline approaches using Python (version 2.7.3). The evaluation experiments were conducted using a server with 128 GB of main memory and two 6-core Intel Xeon CPUs that run at 2.4 GHz speed (only a single core was used). To facilitate repeatability of our experiments, the prototype codes and the synthetic data sets are available from the author.

4.5 Data Sets

To evaluate different aspects of our proposed approaches we used both real as well as synthetic data sets. Table 4.1 summarizes these data sets.

- **NC data set**[1]**:** is a large real voter registration data set from the US state of North Carolina (NC) that contains the names, addresses, and ages of around 8

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

Table 4.1: Data sets summary. 'No. Records' is the number of the records in a data set. 'No. Duplicates' is the number of records that are represented more than once in a data set. 'No. Entities' is the number of real-world entities that exist in the data set. 'Ave Duplicates' represents the average number of duplicates per entity in a data set. Note that each entity in a data set can be represented by one or more records.

Data sets	Provenance	No. Records	No. Duplicates	No. Entities	Ave Duplicates
NC	Real	7,997,234	150,089	7,847,145	3.0
CCA-1	Real	689,928	*CNF	CNF	CNF
CCA-3	Real	2,064,823	CNF	CNF	CNF
CCA-10	Real	6,900,163	CNF	CNF	CNF
CCA-30	Real	20,708,303	CNF	CNF	CNF
Cora	Real	1,295	1,183	112	23.3
DBLP/ACM	Real	4,910	2,224	2,686	1.5
OZ-(1,2,3,4)	Real (modified)	345,876	172,938	172,938	◇3.5
Febrl-5	Synthetic	100,000	80,000	20,000	5.0
Febrl-10	Synthetic	100,000	90,000	10,000	10.0
Febrl-20	Synthetic	100,000	95,000	5,000	20.0

* CNF: Confidential ◇ Unless specified otherwise

million voters, as well as their voter registration numbers (the used attributes are 'Firstname', 'Surname', 'City', and 'Zipcode'). Each record has a time-stamp attached which corresponds to the date a voter originally registered, or when any of their details have changed. This data set 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 data set. This data set is used for scalability evaluation.

- **CCA data set**: is a confidential commercial data set which contains names and addresses of tens of millions of individuals, as well as a log file of query records against this data set. To evaluate the scalability of our proposed approaches, we generated four subsets of different sizes by randomly selecting records from the full CCA data set. The first subset (CCA-1) contains 689,928 data set records and 50,190 query records, the second sebset (CCA-3) contains 2,064,823 data set records and 151,343 query records, the third subset (CCA-10) contains 6,900,163 data set records and 504,226 query records, and the last subset (CCA-30) contains 20,708,303 records and 1,513,233 query records. The number of records in the larger subsets relative to CCA-1 is 3 times, 10 times, and 30 times, respectively.

- **OZ-x data sets**: We generated four data sets with various corruption ratios using the GeCo data generator and corrupter [149], for the purpose of investigating the effect of having different levels of data quality in attribute values on

matching quality. The four data sets each contains 345,876 records of personal details ('Firstname', 'Surname', 'Suburb', and 'Postcode') selected randomly from a clean Australian telephone directory, and 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 data set that we used ranging from OZ-1 to OZ-4. For example, in OZ-1 the added duplicates have been corrupted by modifying only one attribute while for OZ-4 added duplicates have been corrupted by modifying all four attributes in a record. Each entity is represented on average by 3.5 duplicates. These data sets are used to evaluate the effect of how different levels of noise (i.e. different data quality) in a data set affect the performance of the proposed approaches.

- **Febrl data sets:** We generated three fully synthetic data sets where we specified the average number of records per entity (person) using the Febrl data generator [37]. The three data sets each contains 100,000 records consisting of name and address attributes. In the first data set (named Febrl-5) each entity is on average represented by 5 records (with a maximum of 8 records per entity), in the second data set (named Febrl-10) each entity is on average represented by 10 records (with a maximum of 15 records per entity), and in the third data set (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 such as keyboard edits, phonetic and OCR modifications, and setting values to missing. These data sets are used to evaluate the effect of having different number of duplicates in a data set on the proposed approaches.

- **Cora[2] and DBLP/ACM [96]**: Are both real-world bibliographic gold standard data sets that are commonly used in ER research. Cora has 1,295 records and 112 entities (authors), while DBLP/ACM has 2,616/2,294 records and 2,686 entities (authors). For both data sets, we used the following attributes to conduct the ER process: 'authors', 'title', 'venue' and 'year'. These data sets are used to evaluate the proposed blocking/sorting key selection algorithm as they are commonly used in this area.

4.6 Summary

In this chapter we described the framework that is used in evaluating the proposed approaches. We provided details on the evaluation measures, baselines, record pair comparison, implementation environment, and data sets we will use. The next four chapters describe the proposed approaches. We will empirically evaluate these approaches using the evaluation framework presented in this chapter.

[2] Available from: `http://secondstring.sourceforge.net`

Chapter 5

Dynamic Similarity-Aware Inverted Index for Real-Time Entity Resolution

As described in Chapter 3, there is a need for blocking-based indexing techniques that work with real-time ER. In this chapter we propose a dynamic blocking-based indexing technique that supports query-based matching in real-time. In Section 5.2 we summarize the notation that we use in this chapter. Then, we describe our proposed approaches in Sections 5.3, and 5.4. In Section 5.5 we provide an analysis of the proposed approach in terms of estimating the number of comparisons required to match query records, and in Section 5.6 we describe the experimental evaluation. Finally, we summarize our findings in Section 5.7.

5.1 Introduction

Blocking-based indexing techniques are commonly used in entity resolution (ER) [33] to reduce the search space by grouping similar records together using a *blocking key* criterion. However, as described in Chapter 3, most existing indexing techniques are static and only work with traditional ER where two or more data sets are matched off-line using batched processing algorithms. Such indexing techniques cannot be used with real-time ER where a stream of query records needs to be matched with an existing data set in real-time.

In this chapter we propose a dynamic indexing technique that works with real-time ER on dynamic data sets. Our proposed technique is based on a similarity-aware inverted index proposed in [35]. We first propose a dynamic inverted index (named DySimII) that is updated after every query record, by adding arriving query records into the index data structures, leaving the index up-to-date at all times. Because this is a memory-based solution, it is important that the full index can fit into available memory. This is challenging for large data sets; therefore, we propose a frequency-based alteration (named DySimII-F) where we reduce the size of the index by only inserting most frequent attribute values into the index data structures. The following sub-sections describe the proposed approaches in more details.

Table 5.1: Summary of the main notations used in this chapter

R	A data set of records about known entities
A	A set of attributes $\{a_1, a_2, \ldots, a_{\|\mathbf{A}\|}\}$ for each $r_i \in \mathbf{R}$
Q	A stream of query records
C	A list of candidate records for a query q_j
D	An inverted index or disk-based data set table
$\mathbf{M}_{q_j}$	A set of all records in **R** that belong to the same entity of a query q_j
r_i	A record in **R**
$r_i.id$	Unique identifier for r_i
$r_i.eid$	Entity identifier for r_i
q_j	A query record in **Q**
$q_j.id$	Unique identifier for q_j
$q_j.eid$	Entity identifier for q_j
n	The size of data set **R**
$sim(.,.)$	A function used to calculate the similarity between two values ($0 \leq sim(.,.) \leq 1$)
BK	A blocking key that is used to partition records in **R** into blocks of similar records
BKV	The blocking key value of an attribute $r_i.a_h \in \mathbf{R}$.
b	The number of the generated blocks using a certain BK.

5.2 Terminology and Notation

The following is a summary for the terminology and the notation that we use in this chapter:

- **Data set:** We assume that data set $\mathbf{R} = \{r_1, r_2, \ldots, r_{|\mathbf{R}|}\}$ contains records of known entities (an entity can be a person, a product, a business or any other object that exists in the real world). Each $r_i \in \mathbf{R}$ has a unique record identifier $r_i.id$ and an entity identifier $r_i.eid$ (note that several records in **R** can represent the same entity). Records in **R** are described by a set of attributes, denoted as $\mathbf{A} = \{a_1, a_2, \ldots, a_{|\mathbf{A}|}\}$. All records in **R** are assumed to have the same attribute structure.

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

- **Blocking Key:** A blocking key (BK) is defined as the attribute(s) that are used to group similar records in **R** together. To improve the quality of the generated blocks we use encoding function such as Soundex, Phonex, or Double-Metaphone [33] to encode the attributes that are selected as BKs. Selecting the BKs, and the encoding functions generally requires domain knowledge.

- **Blocking key value:** A blocking key value (BKV) for a record $r_i \in \mathbf{R}$ is the encoded value of the attribute $r_i.a_h$ (where $a_h \in \mathbf{A}$) that is used as BK using encoding functions [33]. The list of the encoding functions that are used to encode the different attributes of r_i is denoted as $\mathbf{E} = \{e_{a_1}, e_{a_2}, \ldots, e_{a_{|\mathbf{A}|}}\}$. For example, assume that the 'Firstname' attribute from the example records in Figure 5.2 is used as a BK, then the encoded value $c = e_{a_h}(r_i.a_h)$ of record

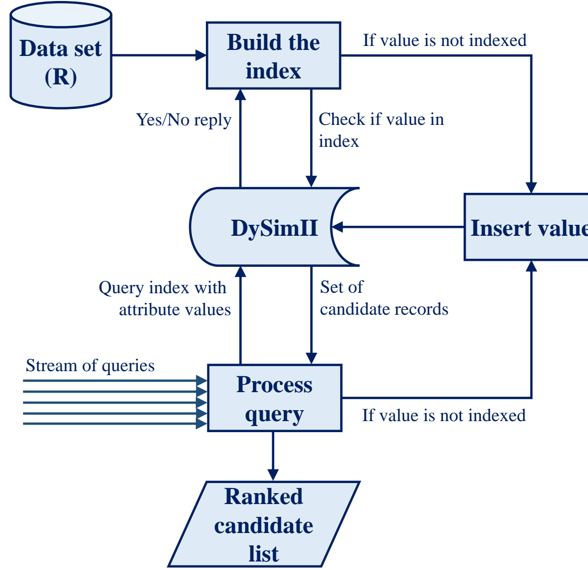


Figure 5.1: The framework for the DySimII technique.

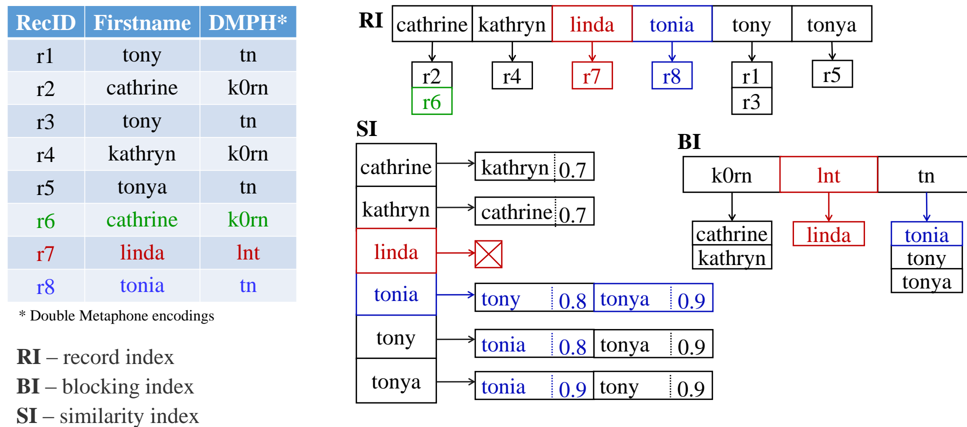
$r1$, with the first name value of '*tony*', will be '*tn*' using the Double-Metaphone encoding function. A BKV can also be generated using the actual attribute values without using encoding functions. However, the aim of using encoding functions is to ensure that similar values are inserted into the same blocks even if they have errors and variations [33]. Note that BKVs for a query record $q_j \in \mathbf{Q}$ are also generated as described above.

The problem of real-time ER is defined as: for each query record q_j in a query stream $\mathbf{Q}$, find all the records in $\mathbf{R}$ that belong to the same entity as q_j, denoted as the set $\mathbf{M}_{q_j}$, in sub-second time, where $\mathbf{M}_{q_j} = \{r_i \mid r_i.eid = q_j.eid, r_i \in \mathbf{R}\}, \mathbf{M}_{q_j} \subseteq \mathbf{R}, q_j \in \mathbf{Q}$. Table 5.1 summarizes the notation that we use.

5.3 Overview of the Approach

The similarity-aware inverted index proposed in [35] is an indexing technique that aims at providing real-time ER for a stream of query records. The main idea behind this technique is to pre-calculate similarities between attribute values that are in the same block. These pre-calculated similarities are stored in main memory to be used later in the query matching process. Avoiding similarity calculations at query time significantly reduces the time required for matching a query record. This technique was shown to be two orders of magnitude faster than standard blocking [35], which makes it suitable for real-time entity resolution.

However, this technique is static and once the index data structures are created, new records and attribute values cannot be added to the index. Therefore, this index dose not work with dynamic data sets. To overcome this limitation we propose a



RecID	Firstname	DMPH*
r1	tony	tn
r2	cathrine	k0rn
r3	tony	tn
r4	kathryn	k0rn
r5	tonya	tn
r6	cathrine	k0rn
r7	linda	lnt
r8	tonia	tn

Figure 5.2: The DySimII created from the example records in the table on the left. The example records contain first name values, and their Double-Metaphone encodings are used as BKVs. r6, r7 and r8 in the table illustrate the different cases for inserting an attribute value to the index as described in Section 5.3.2.

dynamic similarity-aware inverted indexing technique (DySimII) that is more flexible where values can be added to the index data structures whenever a new query record is processed. The framework of the DySimII approach is illustrated in Figure 5.1.

5.3.1 Index Data Structure

The DySimII consists of three index data structures as shown in Figure. 5.2. The first index, called the Block Index (**BI**), is an inverted index that stores unique attribute values and their associated BKVs (generated using encoding techniques such as Soundex, Phonex, or Double-Metaphone [33]). Keys of this index are the BKVs, while each key points to a list of all attribute values that have this BKV. The second index, called the Similarity Index (**SI**), stores the pre-calculated similarities between attribute values that are in the same block (similarities are calculated using approximate string comparison functions, denoted as $sim(.,.)$, such as Jaro-Winkler or Jaccard [33]). Keys for the **SI** are unique attribute values, while each key points to a list of pre-calculated similarities between this value and all other values that are in the same block where $0 < sim(.,.) < 1$. Finally, the Record Index (**RI**) stores all unique attribute values and their associated record identifiers. Keys of this index are the unique attribute values, while each key points to a list of all record identifiers that have the same attribute value.

The indexing process in DySimII is divided into two phases: a *build phase*, and a *query phase*. In the build phase, a data set is loaded into main memory using the three indexes described in [35]. Next, in the query phase, when a query record arrives, its attribute values are inserted into the index data structures. Then, a list of all records that share at least one block with the query record is generated and compared, in detail, with the query record looking for matching records. Unlike the index in [35] (which is static), our DySimII technique is more flexible since arriving query records

Algorithm 5.1: DySimII – *overall*($\mathbf{R}$, $\mathbf{A}$, $\mathbf{E}$, $\mathbf{S}$, $\mathbf{Q}$)

Input:
- Data set: $\mathbf{R}$
- Attributes used: $\mathbf{A} = \{a_1, a_2, \ldots, a_{|\mathbf{A}|}\}$
- Encoding functions: $\mathbf{E} = \{e_{a_1}, e_{a_2}, \ldots, e_{a_{|\mathbf{A}|}}\}$
- Similarity functions: $\mathbf{S} = \{sim_{a_1}, sim_{a_2}, \ldots, sim_{a_{|\mathbf{A}|}}\}$
- Stream of query records: $\mathbf{Q}$

Output:
- Record index: **RI**
- Similarity index: **SI**
- Block index: **BI**
- Ranked list of classified matches: **M**

```
1:  RI := { }, SI := { }, BI := { }
2:  for r_i ∈ R do:                                   // The build phase
3:      for a_h ∈ A do:
4:          insert(r_i.a_h, r_i.id, e_a_h, sim_a_h, RI, SI, BI)   // This method is described in Algorithm 5.2
5:  for q_j ∈ Q do:                                   // The query phase
6:      query(A, q_j, e_a_h, sim_a_h, RI, SI, BI)     // This method is explained in Algorithm 5.3
```

are inserted into the index leading to an up-to-date index at all times. The following sub-sections describe the build and the query phases of the DySimII technique.

5.3.2 Building the Index

The overall process of the DySimII technique is illustrated in Algorithm 5.1). We start with three empty indexes (**RI**, **SI**, and **BI**), then unique attribute values for records are added into the three indexes as described in [35]. Adding attribute values to the inverted indexes is based on two cases: the first case occurs when an attribute value is new and it does not exist in the inverted indexes (like r7 and r8 in Figure 5.2). The second case occurs when an attribute value has been indexed previously and it exists in the inverted indexes (like r6 in Figure 5.2).

1. **New value**: In this case, where an attribute value for a record (denoted as $r_i.a_h$) is new, we first insert the value into the **RI** with its associated record identifier $r_i.id$. Then the encoded value for attribute $r_i.a_h$ is calculated using an encoding function $e_{a_h}()$ to decide into which block it should be added. If the block of that encoded value c = $e_{a_h}(r_i.a_h)$ exists then the attribute value $r_i.a_h$ will be inserted into that block in the **BI**. Otherwise, a new block for the encoded value c = $e_{a_h}(r_i.a_h)$ is created and this attribute value $r_i.a_h$ is inserted into the newly created block. If this value is added into an existing block, the similarities between this new attribute value and all other values within this block are calculated using an approximate string comparison function $sim_{a_h}(.,.)$. These similarities are then stored in the **SI**.

   For example, to insert *r*8 into the index, first we calculate the encoded value for its attribute value as e_{a_h}('tonia') = 'tn'. Because the encoded value 'tn' exists in the **BI** we only insert the record identifier *r*8 into the associated block with the key value of 'tn'. This block now includes three attribute values {'tonia', 'tony',

Algorithm 5.2: DySimII – *insert*($r_i.a_h$, $r_i.id$, e_{a_h}, sim_{a_h}, **RI**, **SI**, **BI**)

Input:
- Attribute value: $r_i.a_h$
- Record identifier: $r_i.id$
- The encoding function used with the inserted attribute: e_{a_h}
- The similarity function used with the inserted attribute: sim_{a_h}
- Indexes: **RI**, **SI**, **BI**

Output:
- Updated indexes: **RI**, **SI**, **BI**

```
1:   Append ri.id to RI[ri.ah]            // Add the record identifier ri.id into RI
2:   if ri.ah ∉ SI then:                  // If the attribute value ri.ah is not indexed before
3:       c := e_ah(ri.ah)                 // Generate the encoding value for ri.ah
4:       b := BI[c]                       // From the BI, get the list of unique attribute values
                                          // that have the same encoded value (c)
5:       Append ri.ah to b                // Add this attribute value ri.ah to b
6:       BI[c] := b                       // Update the block index BI
7:       Initialize inverted index list si := [ ]
8:       for v ∈ b do:
9:           s := sim_ah(ri.ah, v)        // Calculate the similarity between ri.ah and v
10:          Append (v, s) to si          // Add the pair (v, s) to si
11:          oi := SI[v]                  // From the SI, get the pre-calculated similarities for v
12:          Append (ri.ah, s) to oi
13:          SI[v] := oi                  // Update the similarity index SI for v
14:      SI[ri.ah] := si                  // Update the similarity index SI for ri.ah
```

and ‘tonya’}. The next step is to update both the **RI** and the **SI** by inserting the value ‘tonia’ and its record identifier $r8$ into the **RI**, and by pre-calculating the similarity between ‘tonia’ and all other values that are in the same block {‘tony’ and ‘tonya’}, then add these similarities into the **SI**.

2. **Indexed value**: In this case, where an attribute value $r_i.a_h$ has been indexed previously, the only action needed is to add the identifier of this record $r_i.id$ that holds this attribute value into the corresponding record list in the **RI**. For example, to insert $r6$ into the index, first we calculate the encoded value for its attribute value as e_{a_h}(‘cathrine’) = ‘k0rn’. Because this value is indexed previously, there is no need to update the **BI** or the **SI**. The only thing needed is to update the **RI** by adding the record identifier $r7$ into the corresponding attribute value in the**RI**.

The process of loading and indexing attribute values will continue until the last record in data set **R** is inserted into the index data structures. As a result we will have three inverted indexes (**BI, SI**, and **RI**). In the original similarity-aware indexing technique [35], building the indexes will stop at this point. If a new values arrive, these values cannot be added to the indexes. However, this issue is handled in DySimII allowing more values to be added to the indexes. When a query record arrives, it is given a unique identifier $q_j.id$, and its attribute values are encoded. The steps described above will take place adding any new value to the previously built index data structures.

The process of inserting a new attribute value $r_i.a_h$ (shown in Algorithm 5.2) requires the identifier of the record of this attribute, $r_i.id$, the encoding function

Algorithm 5.3: DySimII – $query(\mathbf{A},\ q_j,\ e_{a_h},\ sim_{a_h},\ \mathbf{RI},\ \mathbf{SI},\ \mathbf{BI})$

Input:
- Attributes used: $\mathbf{A} = \{a_1,\ a_2,\ \ldots,\ a_{|\mathbf{A}|}\}$
- Query record: $q_j \in \mathbf{Q}$
- The encoding function used with the inserted attribute: e_{a_h}
- The similarity function used with the inserted attribute: sim_{a_h}
- Indexes: **RI**, **SI**, **BI**

Output:
- Ranked list of matches: **M**

```
1:  Initialize M := [ ]                               // The list of ranked candidate matches
2:  Assign q_j a new unique identifier q_j.id
3:  for a_h ∈ A do:
4:      if q_j.a_h ∉ RI then:                         // If the attribute value q_j.a_h is new
5:          insert(q_j.a_h, q_j.id, e_a_h, sim_a_h, RI, SI, BI)  // Insert q_j.a_h into RI, SI, BI (see Algorithm5.2)
6:      else:                                         // If value q_j.a_h is indexed before
7:          Append q_j.id to RI[q_j.a_h]              // Add identifier q_j.id into RI
8:      ri := RI[q_j.a_h]                             // From the RI, get all identifiers for records that
                                                      // have the same attribute value as q_j.a_h
9:      for r_i.id ∈ ri do:
10:         M[r_i.id] := M[r.id] + 1.0                // Values in ri are exact (similarity =1.0)
11:     si := SI[r_i.a_h]                             // From the SI, retrieve attribute values and their
                                                      // similarities that are in the same block as r_i.a_h
12:     for (r_i.a_h, s) ∈ si do:
13:         ri := RI[r_i.a_h]                         // From the RI, get identifiers for all records
                                                      // that are in the same block as r_i.a_h
14:         for r_i.id ∈ ri do:
15:             M[r_i.id] := M[r_i.id] + s            // Add similarities to accumulator M
16:     Sort M descending based on similarity values
```

$e_{a_h}(.)$, the similarity comparison function $sim_{a_h}(.,.)$, and the inverted indexes **RI**, **SI**, and **BI** as input. The process starts with inserting $r_i.id$ into the **RI** (line 1). If this attribute value does not exist in the **SI** (line 2), the following steps are conducted. First, the encoding value c = $e_{a_h}(r_i.a_h)$ is calculated and all other values in its block are retrieved from the **BI**. The new value is then added into the inverted index list b of this block, and the updated list is stored back into the **BI** (lines 3 to 6). Next the similarities between the new attribute value $r_i.a_h$ and all attribute values that already exist in this block b are calculated (line 9), and inserted into both the new value's similarity list **si** (line 10) and the other value's list **oi** (line 12). Finally, the similarity list **si** of the new value $r_i.a_h$ is added into the **SI** in line 14.

5.3.3 Querying the Index

In the query phase of the proposed DySimII approach, we add any new attribute value from a query record into the indexes. Therefore, if the same attribute value occurs in several query records the encoding and similarity computations that are required to do the matching are performed only the first time it occurs. Algorithm 5.3 illustrates how a query is handled in the DySimII.

First an accumulator **M**, which is a data structure that contains record identifiers and their similarities with the query record, is initialized (line 1). The query record q_j is then assigned a new unique identifier $q_j.id$ in line 2. Next, for every attribute

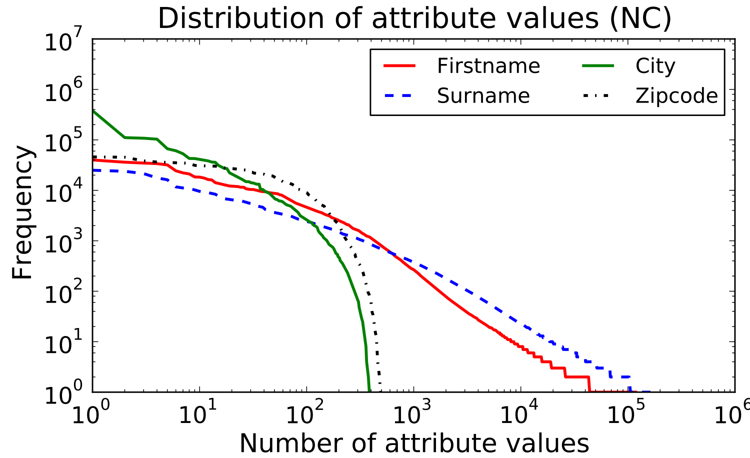


Figure 5.3: The frequency distribution of attribute values for 30% of the NC data set (details about NC data set can be found in Chapter 4).

value in the query, the algorithm checks if this value is in the **RI** (lines 3 and 4). If it is not in the **RI**, then it will be inserted (in line 5) into the three indexes as described in Algorithm 5.2. In lines 6 to 8, the identifiers $r_i.id$ of all other records that have the same attribute value are retrieved and their similarities (exactly 1, as they have the same attribute value) are added into the accumulator **M** (lines 9 and 10). A new element for record identifier $r_i.id$ will be added to the accumulator if it does not exist. Next, all other attribute values in the same block and their similarities with the query attribute value are retrieved from the **SI** (line 11). For each of these values, their record identifiers are retrieved from the **RI** and their similarities are added into the accumulator (lines 12 to 15). Finally, in line 16, the accumulator is sorted such that the records with largest similarities are located at the beginning, and this sorted list is returned.

5.4 Frequency-Based Similarity-Aware Inverted Index

The DySimII technique is based on building the inverted indexes for all unique attribute values in a data set. However, as seen in Figure 5.3 and in Table 5.2 the majority of attribute values are uncommon and have low frequencies. For example, for the ‘Firstname’ attribute in Table 5.2 (b) around 68% of its attribute values occur only once in the data set, and around 94% of attribute values have a frequency that is less than 10 records in the data set.

Indexing such rare values might not be of use. For instance, the first name *John* in the NC data set (described in Chapter 4) is a common name and it has a frequency of 34,141 (around 29% of the unique first names) in this data set, while a first name like *Juvena* is uncommon and only occurs once (around 0.001% of the unique first names) in the same data set. This suggests that the probability of receiving a query with the first name value *John* is much higher than the probability of receiving a query

Table 5.2: Table (a) illustrates a list of the top 10 most frequent first names and surnames for 30% of the NC data set (described in Chapter 4). The number (118,565) is the total number of unique first names and (189,890) is the total number of unique surnames. Table (b) illustrates the percentage of uncommon first names and surnames in 30% of the NC data set.

(a)

Firstname (118,565)		**Surname (189,890)**	
James	41,022 (35%)	Smith	28,243 (15%)
Michael	37,536 (32%)	Williams	23,247 (12%)
John	34,141 (29%)	Johnson	22,100 (12%)
Robert	33,241 (28%)	Jones	19,807 (10%)
William	31,918 (27%)	Brown	16,256 (9%)
David	30,126 (25%)	Davis	15,002 (8%)
Mary	22,993 (19%)	Moore	10,968 (6%)
Christopher	20,590 (17%)	Miller	10,254 (5%)
Jenifer	19,468 (16%)	Wilson	10,254 (5%)
Charles	17,438 (15%)	Harris	9,380 (5%)

(b)

Attribute	**Frequency = 1**	**Frequency <= 10**
Firstname	67.72%	93.63%
Surname	47.14%	90.76%

with the value *Juvena*. Figure 5.3 illustrates the frequency distributions of the four attributes that are in this data set (i.e. 'Firstname', 'Surname', 'City', and 'Zipcode'). It can be seen that only a small number of attribute values have a high frequency, while most have low frequencies.

Many data sets that contain values such as personal details are found to have a frequency distribution that follow Zipf's law [173], which states that in a list of words ranked according to their frequencies, the word at rank rk has a relative frequency that corresponds to $1/rk$. Distributions in which the relative frequency approximates $1/rk^{\alpha}$ are considered Zipfian, even if $\alpha \neq 1$. This means that the number of uncommon values in many data sets is large, and since these values are taking space in the inverted index while they are not queried very often, we suggest filtering the indexing process by only adding to the index the most frequent attribute values. This will reduce the size of the memory required to build the DySimII, and improve the ability of storing larger data sets into the index data structures. Therefore, in this section, we propose a frequency-based alteration (DySimII-F), where we investigate the effect of indexing only $x\%$ of the most frequent attribute values where $0 < x\% < 100$. The following sub-sections describes the build and the query phases of the DySimII-F.

Algorithm 5.4: DySimII-F – *overall*(**R**, **A**, **E**, **S**, **Q**, **F**)

Input:
- Data set: **R**
- Attributes used: $\mathbf{A} = \{a_1, a_2, \ldots, a_{|\mathbf{A}|}\}$
- Encoding functions: $\mathbf{E} = \{e_{a_1}, e_{a_2}, \ldots, e_{a_{|\mathbf{A}|}}\}$
- Similarity functions: $\mathbf{S} = \{sim_{a_1}, sim_{a_2}, \ldots, sim_{a_{|\mathbf{A}|}}\}$
- Stream of query records: **Q**
- List of most x% frequent attribute values: **F**

Output:
- Record index: **RI**
- Similarity index: **SI**
- Block index: **BI**
- Ranked list of classified matches: **M**

```
1:  RI := { }, SI := { }, BI := { }
2:  for r_i ∈ R do:                                  // The build phase
3:      for a_h ∈ A do:
4:          if a_h ∈ F then:                          // Only index attribute values that are among
5:              insert(r_i.a_h, r_i.id, e_a_h, sim_a_h, RI, SI, BI)   // the top x% of the most frequent values
6:  for q_j ∈ Q do:                                  // The query phase is explained
7:      query(A, q_j, e_a_h, sim_a_h, RI, SI, BI, F)  // in Algorithm 5.5
```

5.4.1 Building the Frequency-Based Index

The difference in the build phase between DySimII and DySimII-F is that the later indexes only values that are among the top x% of the most frequent attribute values (see Algorithm 5.4). This requires a list of frequent attribute values (**F**), which can be generated for example from an on-line telephone book. Before we add any value into the index data structures, we check if this value is in the x% of most frequent values (line 4). If this is the case, the value is added into the index data structures (as described in Section 5.3.2), otherwise it will not (line 5). This process is expected to reduce the size of the index structures since we are only indexing the most frequent values. After building the index using only the top x% values, the index is ready to process arriving query records as described next.

5.4.2 Querying the Frequency-Based Index

In the query phase of DySimII-F, illustrated in Algorithm 5.5, we only add the most frequent attribute values to the index data structures. For queries with indexed values (values that exist in the index), the query is processed exactly as described in Section 5.3.3. As for queries with new values (that are not indexed earlier), if the value is within the top x% of the most frequent values in **F**, the query is processed as described in 5.3.3. On the other hand, if a query value is not frequent it will be handled as described in the following paragraph and illustrated in Algorithm 5.5 lines 8 to 19.

We start by encoding this attribute value using the encoding function $e_{a_h}(q_j.a_h)$ (line 9). Next, from the **BI**, we get a list of unique attribute values that have the same encoding value as the query's encoded value (e.i. are in the same block) (line 10). Then, in lines 11 and 12, we calculate the similarity between the attribute values that are with in the same block as the query attribute value. These unique values and

their similarities with the query value are stored into a similarity list **si** (line 13). For each of these values, their record identifiers are retrieved from the **RI** (line 15) and their similarities are added into the accumulator (lines 16 and 17). Finally, in line 18, the accumulator is sorted such that the records with the largest similarities are located at the beginning, and this sorted list is returned.

Algorithm 5.5: DySimII-F – $query(\mathbf{A},\ q_j,\ e_{a_h},\ sim_{a_h},\ \mathbf{RI},\ \mathbf{SI},\ \mathbf{BI},\ \mathbf{F})$

Input:
- Attributes used: $\mathbf{A} = \{a_1,\ a_2,\ \ldots,\ a_{|\mathbf{A}|}\}$
- Query record: $q_j \in \mathbf{Q}$
- The encoding function used with the inserted attribute: e_{a_h}
- The similarity function used with the inserted attribute: sim_{a_h}
- Indexes: **RI**, **SI**, **BI**
- List of most x% frequent attribute values: **F**

Output:
- Ranked list of matches: **M**

```
1:  Initialize M := [ ]                                   // The list of ranked found matches
2:  Initialize inverted index list si := [ ]              // The list of calculated similarities
3:  Assign q_j a new unique identifier q_j.id
4:  for a_h ∈ A do:
5:      if q_j.a_h ∉ RI then:                             // If the attribute value q_j.a_h is new
6:          if a_h ∈ F then:                              // Only index attribute values that are in
7:              insert(q_j.a_h, q_j.id, e_a_h, sim_a_h, RI, SI, BI)  // the top x% of the most frequent values
8:          else:                                         // If value q_j.a_h is not frequent
9:              c := e_a_h(q_j.a_h)                       // Generate the encoding value for q_j.a_h
10:             b := BI[c]                                // Get the list of attribute values that
                                                          // have the same encoded value as (c)
11:             for r_i.a_h ∈ b do:
12:                 s := sim_a_h(q_j.a_h, r_i.a_h)        // Calculate the similarity between
                                                          // q_j.a_h and r_i.a_h that have the same
                                                          // encoded value (c)
13:                 Append (r_i.a_h, s) to si             // Add the calculated similarities to si
14:             for (r_i.a_h, s) ∈ si do:
15:                 ri := RI[r_i.a_h]                     // Get identifiers for all records that
                                                          // are in the same block as r_i.a_h
16:                 for r_i.id ∈ ri do:
17:                     M[r_i.id] := M[r_i.id] + s        // Add similarities to accumulator M
18:             Sort M descending based on similarity values
19:             Break
20:     else:                                             // If value q_j.a_h has been indexed before
21:         Append q_j.id to RI[q_j.a_h]                  // Add identifier q_j.id into the RI
22:     ri := RI[q_j.a_h]                                 // Get all identifiers for records that
                                                          // have the same attribute value as q_j.a_h
23:     for r_i.id ∈ ri do:
24:         M[r_i.id] := M[r.id] + 1.0                    // Values in ri are exact (similarity =1.0)
25:     si := SI[r_i.a_h]                                 // Retrieve attribute values and their
                                                          // that are in the same block as r_i.a_h
26:     for (r_i.a_h, s) ∈ si do:
27:         ri := RI[r_i.a_h]                             // Get identifiers for all records that
                                                          // are in the same block as r_i.a_h
28:         for r_i.id ∈ ri do:
29:             M[r_i.id] := M[r_i.id] + s                // Add similarities to accumulator M
30:     Sort M descending based on similarity values
```

5.5 Estimating the Number of Comparison Required for the Proposed Approach

The comparison step in the ER process is usually the most time consuming step. This is because of the expensive similarity calculations performed when comparing the query record with the generated candidate records in details. Estimating the number of comparisons required to match a query record with an existing data set (with a certain size) beforehand gives users an insight into the expected run time required to match query records. In this section, we provide a way of estimating the number of generated candidate records using the DySimII approach.

The proposed DySimII approach groups similar attribute values into the same block within the **BI**. For each attribute value inserted into the **BI**, a list of the identifiers for all records that have the same attribute value is stored in the **RI**. Therefore, the number of candidate records for a query value depends on the size of the block (in the **BI**) where the query value is inserted, and the length of the list of identifiers (from the **RI**) for each value within the block of that query value. For example, from Figure 5.2 assume that we have a query record r9 that has the value 'tonia'. The number of the candidate records for this query r9 (that have the value 'tonia') is calculated by getting the length of identifier's lists (from the **RI**) for each value in the block 'tn' (i.e. which has 3 values 'tonia', 'tony' and 'tonya') without including the record identifier of the query value. Thus, the list of candidate records includes four records {r8, r1, r3, r5}.

To obtain a better understanding of the number of candidate record that will be generated (using the DySimII approach) for a certain query record, we assume a uniform and a Zipfian frequency distributions of attribute values in a data set. Previous work that estimated the number of candidate record pairs for several indexing techniques also assumed these two distributions [34]. 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 data sets.

Assuming a uniform distribution for the frequency of the attribute values lead to a uniform distribution of the generated record identifier lists in the **RI** (which means that all lists in the **RI** have the same length), and a uniform distribution of the generated blocks in the **BI** (which means that all blocks in the **BI** have the same size). On the other hand, having a Zipfian distribution for the attribute values leads to a Zipfian distribution for the identifier lists in the **RI** and the block sizes in the **BI**. According to the Zipfian law [173], 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 $1/rk$, where rk is the rank of a value.

For the DySimII data structures (described in Section 5.3.1), assuming a uniform distribution for both the **BI** and the **RI**, the number of candidate records for a query record is equal to the product of the size of a block in the **BI** and the size of an identifier's list in the **RI**. Note that we assume that all identifier's lists in the **RI** have the same length that is equal to n/r where n is the number of records in the data set, and r is the number of the generated identifier lists in the **RI** (i.e. the number

of unique attribute values in **RI**). We also assume that all block in the **BI** have the same size that is equal to r/b where b is the number of the generated blocks in the **BI**. Therefore, the number of candidate records **C** can be calculated as:

$$|\mathbf{C}| = \frac{n}{r} * \frac{r}{b} \tag{5.1}$$

which is equal to:

$$|\mathbf{C}| = \frac{n}{b} \tag{5.2}$$

Assuming a Zipfian frequency distribution of the attribute values will lead to a Zipfian distribution of the sizes of blocks in the **BI**, and a Zipfian distribution of the length of the identifier lists in the **RI**. In this case, the number of candidate records will not only be affected by the number of records n in the data set, the number of the generated blocks b in the **BI** and the number of identifier lists in the **RI**, but also by the size of the generated blocks in the **BI** (i.e. the number of values inserted into a block) and the length of identifier list (from the **RI**) for all the values within the block. Assuming we rank the generated blocks in the **BI**, denoted as B_i where $1 \leq i \leq b$, according to their sizes (number of values in a block), the size S_{B_i} of a block is calculated as:

$$S_{B_i} = \frac{1/i}{\sum_{i=1}^{b}(\frac{1}{i})} * r \tag{5.3}$$

where the denominator is the Harmonic number of the partial harmonic sum [34]. Moreover, assuming that we rank the identifier lists (from the **RI**) R_i, $1 \leq i \leq r$, according to their length (number of record identifiers inserted into a list), the size S_{R_i} of an identifier list can be calculated as:

$$S_{R_i} = \frac{1/i}{\sum_{i=1}^{r}(\frac{1}{i})} * n \tag{5.4}$$

The maximum number of candidate records S_q for a query that is inserted into a block in the **BI** which has the size S_{B_i} can be calculated as:

$$S_q = \sum_{i=1}^{i=S_{B_i}} S_{R_i} \tag{5.5}$$

To estimate the minimum and maximum numbers of candidate records for a query q_j we calculate S_q according to the illustration given in Figure 5.4. The maximum number of candidate records occurs when a query is inserted into the largest block in **BI**, and the values in this block have the longest identifier list from **RI**. On the other hand, the minimum number of candidate records occurs when a query is inserted into the smallest block in **BI**, and values in this block have the shortest identifier list from **RI**.

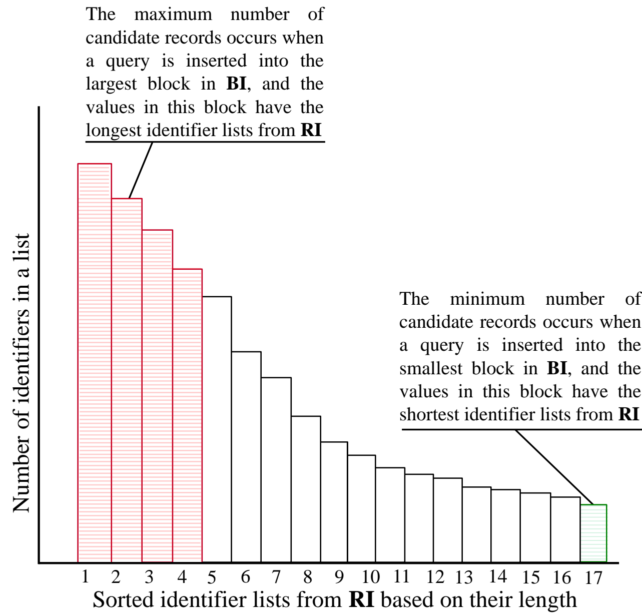


Figure 5.4: Illustration of the situations where the maximum and minimum number of candidate records occurs for the DySimII assuming that the size of the record identifier lists in the **RI** and the blocks in the **BI** follow the Zipfian distributions [173]. In this example, we assume that the largest block size in **BI** has 4 values and the smallest has 1 value. Thus, the maximum number of candidate records occurs when the values in the largest block (with 4 values) from **BI** have the longest identifiers lists from **RI** (the red lists). While the minimum number of candidate records occurs when the values in the smallest block (with 1 value) from **BI** has the shortest identifier list in **RI** (the green list)

5.6 Experimental Evaluation

In this section we describe the experiments conducted to evaluate the proposed DySimII approach and its frequency-based alteration DySimII-F. The data sets used in these experiments are summarized in Table 5.3. More detail about the data sets, the evaluation measures, and the implementation environment are found in Chapter 4. As described in Section 5.3.2, a list of encoding functions **E** is used to block records within a data set. We use the Double-Metaphone [33] phonetic encoding algorithm to encode all attributes (i.e. the encoding values are used as BKs). While for the Zipcode/Postcode attribute we use the last three digits of the Zipcode/Postcode value as a BK [35]. For the list of similarity functions **S**, we used the Jaro-Winlker approximate string comparison function [33] for comparing all attribute values (see Section 4.2), except for the Zipcode/Postcode attribute the similarity is calculated by counting the number of matching digits between compared values [35].

Table 5.3: Data sets summary (described in more details in Chapter 4).

Data set	Type	Number of records
30% of NC	Real	2,399,170
CCA-1	Real	689,928
CCA-3	Real	2,064,823
CCA-10	Real	6,900,163
CCA-30	Real	20,708,303

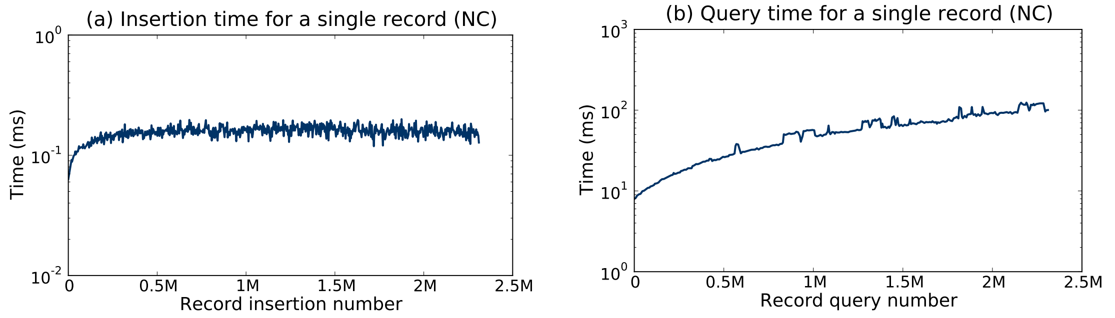


Figure 5.5: Plot (a) shows the average time required for inserting a single record into the index. Plot (b) illustrates the average time required for querying the growing index. A subset of 30% of the NC data set is used (M = Million).

5.6.1 Scalability of the Proposed Solution

In this set of experiments we evaluate whether the proposed DySimII scales to large data sets while facilitating real-time ER. We measure 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 data structure (note that arriving query records are inserted into the index data structure). These experiments are conducted on 2,391,080 records from the NC data sets described in Chapter 4.

As can be seen from Figure 5.5 plot (a), the results show that the average insertion times are not affected by the growing size of the index data structure (almost a constant insertion time), while in plot (b), the query time increases sub-linearly as the index becomes larger. The results also show that the average insertion time is around 0.1 milliseconds (ms) for the growing size of the index, while the average query time is between 10 to 100 (ms).

To investigate how the query time is affected with using larger data sets, we conduct another set of scalability experiments on the different subsets of the CCA data set (described in Chapter 4). The size of these subsets ranges between 689,928 records for the CCA-1 data set and 20,708,303 records for the CCA-30 data set. We build an index using each of the CCA subsets, then we measure the average time

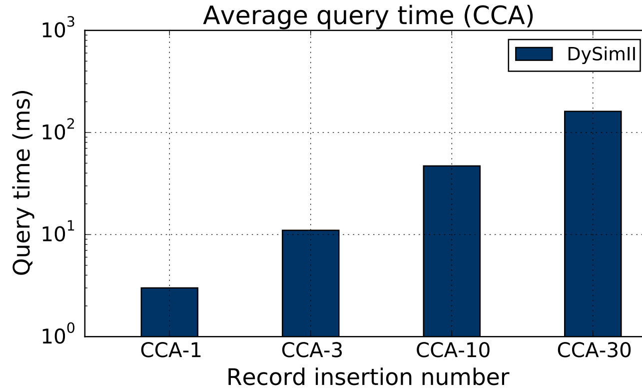


Figure 5.6: Average query time for the different CCA subsets.

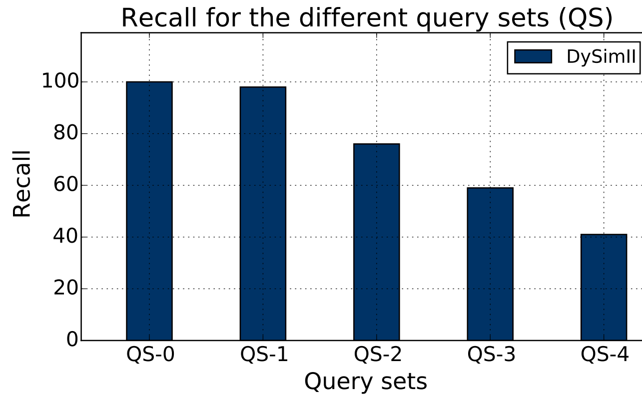


Figure 5.7: Recall for the different query sets using the DySimII approach.

required to query the built indexes. The results (presented in Figure 5.6) show that for the increasing size of the data set the average query time increases sub-linearly yet it is still very fast with an average query time of 161 ms for a data set with over 20 million records. Note that we only use the NC and the CCA data sets for scalability testing since they are larger in size compared to the other data sets described in Section 4.5.

5.6.2 Effects of Having Corrupted Attribute Values in a Query Record on the Quality of the Matching Process

The aim of this set of experiments is to investigate the effect of having different numbers of corrupted (modified) attribute values in a query record on the quality of the matching process. We employ experimental settings that are similar to what is used in [35]. We use 2,391,080 clean records (with no duplicates) from the NC data set to build the index. From those clean records we randomly select 100 records that are used to create five different sets of query records. The first query set (QS-0) is constructed by copying the exact values of the 100 records without any modification. The

Table 5.4: Required memory for the DySimII approach for the different data sets

Data set	Size	Memory (MB)
30% of NC	2,399,170	3,588
CCA-1	689,928	797
CCA-3	2,064,823	2,337
CCA-10	6,900,163	7,874
CCA-30	20,708,303	23,915

second query set (QS-1) is constructed by manually corrupting only one attribute in each query record in the base 100 selected query records (corruptions include adding, deleting, modifying, or swapping characters in the attribute value). For the query set (QS-2) two attributes are corrupted, for (QS-3) three attributes are corrupted, and for QS-4 four attributes (all attributes in the record) are corrupted. These generated query sets are used to query the built index (built with the clean records from the NC data set). The results, presented in Figure 5.7, show (as expected) that recall values decrease with increasing the number of corrupted attributes in the query record. For the QS-0 query set (with no corruptions), the DySimII achieved a recall value of 100%. However, for the other corrupted query sets the recall values achieved are: 98% for QS-1, 76% for QS-2, 59% for QS-3, and 40% for QS-4.

5.6.3 Required Memory Size

Table 5.4 shows the memory requirements for the DySimII approach for the different data sets (these measures are produced using our experimental machine which has 128 GB of main memory). As noted, the memory requirements increases with the increase in the number of recored within a data set. However, for the largest data set (CCA-30) which has more than 20 million records, the required memory for building the index for the full data set is 23,913 MB. This required memory is only around 18% of the amount available on the experimental machine, indicating that our implementation would be viable even for much larger data sets. If the size of the problem increased substantially relative to the physical memory, a possible approach is to prune low-frequency attributes from the index structures, as described in Section 5.4. Another possible alternative is to retain the build indexes on disk(s) and only load into main memory the block(s) that are related to the query record. This option is to be investigated in future work.

5.6.4 Effects of Using Different Values of the Most Frequent Attributes with DySimII-F on the Coverage of the Index

In this set of experiments, we evaluate the DySimII-F approach and investigate the effect of pruning non-frequent attribute values from the index on the ratio between the number of times an attribute value of a query is available in the index (indexed) and the number of times that it is not available (new queries). We call this ratio

Table 5.5: The coverage (the ratio between indexed and new queries) of the DySimII-F while indexing only the top x% of most frequent attribute values.

	x% **of the most frequent attribute values**									
	10%	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
QS-0	0.73	0.85	0.92	0.95	0.98	0.98	0.99	0.99	0.99	1.00
QS-1	0.58	0.68	0.74	0.77	0.80	0.80	0.81	0.81	0.81	0.82
QS-2	0.42	0.50	0.54	0.58	0.60	0.61	0.61	0.61	0.62	0.63
QS-3	0.28	0.32	0.35	0.38	0.40	0.40	0.41	0.42	0.42	0.44
QS-4	0.12	0.15	0.17	0.20	0.21	0.21	0.22	0.23	0.24	0.25

in this chapter the *coverage* of the index. To conduct the experiments, we index the most frequent x% of attribute values where x ranges from 10% to 100%. A value of $x = 10\%$ means that only the most frequent 10% of the attribute values from the NC data set are added to the index structures. We finally apply the DySimII-F approach to investigate its effect on the coverage of the index. Table 5.5 shows that the coverage of the index is very high for all values of x% for the QS-0 query set (i.e. most arriving queries are covered by the index (already indexed)). For example when we index only 30% of the most frequent attribute values from the NC voter data set, 92% of the arriving queries are actually found in the index. However, for the other query sets, the coverage value starts to decrease when more attribute values are corrupted. This indicates that for dirty data sets (with errors and typos in more than one attribute value) the DySimII-F will have a low coverage for arriving query records.

5.6.5 Effects of Using Different Values of the Most Frequent Attribute Values with DySimII-F on Recall

In this set of experiments, we evaluate the DySimII-F approach and investigate the effect of pruning non-frequent attribute values from the index on recall. Similar to the previous set of experiments, we index the top x% of the most frequent attribute values where x ranges from 10% to 100%. Table 5.6 shows that recall drops only slightly when we do not index all attribute values for the QS-0 query set. For example, when we index 50% of the most frequent attribute values, the recall drops only by 2%. The acceptability of this drop in recall depends on who is using the proposed approach. For example, a 2% drop in recall might be acceptable for a general business company, but not for a national security organization. For the other corrupted query sets, the drop in recall ranges between 2% to 4% for when we index 50% of the most frequent attribute values. For indexing only the top 10% of the most frequent attribute values the recall drops 36% for the non-corrupted query set QS-0, while it drops between 22% and 47% for the other corrupted query sets.

Table 5.6: Recall for the DySimII-F approach

	x% of the most frequent attribute values									
	10%	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
QS-0	64	81	92	95	98	99	100	100	100	100
QS-1	51	74	87	90	94	96	97	97	98	98
QS-2	40	57	68	71	73	75	75	76	76	76
QS-3	29	45	51	57	58	59	59	59	60	60
QS-4	19	29	33	38	39	40	40	40	40	41

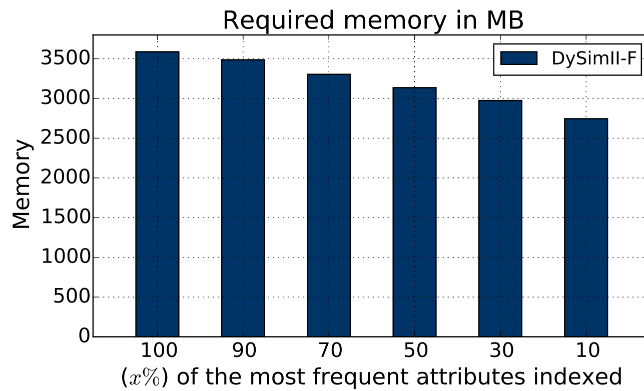


Figure 5.8: The required memory to index x% of the most frequent attribute values using the DySimII-F.

5.6.6 Effects of Using Different Values of the Most Frequent Attribute Values with DySimII-F on Memory Requirements

In this set of experiments, we evaluate the DySimII-F approach and investigate the effect of pruning non-frequent attribute values from the index on the memory required to build the index. We index the top x% of the most frequent attribute values where x ranges from 10% to 100%. The results, presented in Figure 5.8, show that memory requirements drop between 3 % to 23% for the different x% of the most frequent attribute values that are indexed. This drop in memory can be useful when indexing larger data sets.

5.7 Summary

In this chapter we proposed a dynamic similarity-aware inverted indexing technique (DySimII) that can be used for real-time ER on large data sets. We also proposed a frequency-based index (DySimII-F) that is aimed at improving the memory required to build the index data structures by pruning non-frequent attribute values from the index. The results from our experimental evaluation show that the proposed

solution is scalable with large data sets since it has fast insertion and query times, and since the growing size of the index does not affect the time required to insert new records into the index data structures, and it only requires a slight increase in the time required to resolve query records (more results about DySimII can be found in Chapter 9).

The results also show that DySimII achieved high matching quality for data sets with less noise (errors and typos), while the quality of the matching process dropped for dirty data (with more errors and variations within attribute values). Moreover, the results show that the DySimII-F improves the required memory size with only a slight drop in recall values for data set with less noise while it did not perform will with dirty data sets that have more errors and variation within attribute values. Future research directions that can be extended from our work in this chapter are to investigate how to improve the matching quality for data sets with dirty attribute values, address the drop in recall for the F-DySimII, examine how to reduce the size of the F-DySimII, investigate extending the F-DySimII to support disk-based indexing to be used with larger data sets, and investigate the use of this approach in a parallel environment.

Chapter 6

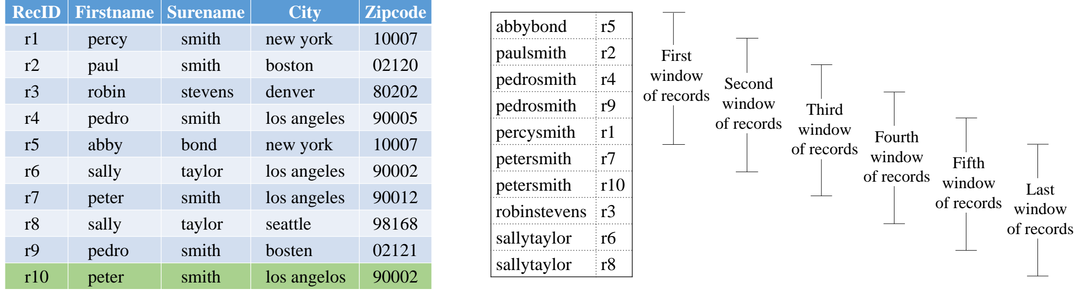
Dynamic Sorted Neighborhood Index for Real-Time Entity Resolution

Sorting-based indexing is very efficient compared to other indexing techniques [34] and is commonly used in entity resolution. In this chapter we propose a dynamic and efficient sorting-based indexing technique that works with real-time entity resolution. We summarize the notation that we use in this chapter in Section 6.2. Then in Sections 6.3 to 6.5 we provide a detailed description of the proposed approach, and in Section 6.6 we analyze the proposed approach with regards to estimating the required number of comparisons. We then evaluate the proposed approach in Section 6.7, and summarize our findings in Section 6.8.

6.1 Introduction

The sorted neighborhood indexing method (SNM) [79] has been successfully used with entity resolution (ER) of large static data sets. It was developed with the aim of reducing the number of comparisons between candidate records in the process of de-duplicating large data sets [79, 80]. The basic SNM (Figure 6.1 shows an example) consists of the following steps: First, if multiple data sets are to be matched, they are merged and a unique identifier is assigned to each record. Then, a sorting key value is generated for every record in the merged data set. Next, the records are sorted according to these sorting key values. 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 data set contains n records, the sorting step of the SNM has a complexity of $O(n\ log(n))$, while the comparison step is $O(w \times n)$ [79].

One of the main drawbacks of the SNM is its sensitivity to data quality of the attributes used as sorting key values. Specifically, if a sorting key value has an error 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 likely not be close to each other in the sorted array if a 'Firstname' attribute is used as a



RecID	Firstname	Surename	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	10007
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

abbybond	r5
paulsmith	r2
pedrosmith	r4
pedrosmith	r9
percysmith	r1
petersmith	r7
petersmith	r10
robinstevens	r3
sallytaylor	r6
sallytaylor	r8

Figure 6.1: 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.

sorting key. A commonly used approach to overcome this drawback is to run the SNM several times using different sorting keys, followed by the calculation of the transitive closure of the identified matching records [79, 80].

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 are conducted. This problem has recently been addressed by two approaches that adaptively adjust the window size according to the characteristics of the sorting key values or data set records. One approach expands the window size if sorting key values are similar with each other according to a minimum similarity threshold [170], while an alternative approach expands a window if a certain minimum number of records are classified as matches within the current window [54]. Both of these approaches are described in more details in Chapter 3.

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 data sets 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 data set consisting of entity records, and where these query records are commonly inserted into the data set and the index data structure after matching. Our proposed technique, described in Section 6.3, provides an indexing technique that facilitates real-time ER, and can handle dynamic data sets.

6.2 Terminology and Notation

In this section we summarize the terminology and the notation used in this chapter:

- **Data set:** We assume that data set $\mathbf{R} = \{r_1, r_2, \ldots, r_{|\mathbf{R}|}\}$ contains records of known entities. Each $r_i \in \mathbf{R}$ has a unique record identifier $r_i.id$ and an entity identifier $r_i.eid$. Records in $\mathbf{R}$ are described by a set of attributes, denoted as $\mathbf{A} = \{a_1, a_2, \ldots, a_{|\mathbf{A}|}\}$. All records in $\mathbf{R}$ are assumed to have the same attribute structure.

Table 6.1: Summary of main notations used in this chapter

$\mathbf{R}$	A data set of records about known entities
$\mathbf{A}$	A set of attributes $\{a_1, a_2, \ldots, a_{\vert\mathbf{A}\vert}\}$ for each $r_i \in \mathbf{R}$
$\mathbf{Q}$	A stream of query records
$\mathbf{C}$	A list of candidate records for a query q_j
$\mathbf{D}$	An inverted index or disk-based data set table
$\mathbf{M}_{q_j}$	A set of all records in $\mathbf{R}$ that belong to the same entity of a query q_j
r_i	A record in $\mathbf{R}$
$r_i.id$	Unique identifier for r_i
$r_i.eid$	Entity identifier for r_i
q_j	A query record in $\mathbf{Q}$
$q_j.id$	Unique identifier for q_j
$q_j.eid$	Entity identifier for q_j
n	The size of data set $\mathbf{R}$, $n = \vert\mathbf{R}\vert$
$sim(.,.)$	A function used to calculate the similarity between two values ($0 \leq sim(.,.) \leq 1$)
w	Window size used to generate candidate records
SK	The sorting key that is used to sort records in $\mathbf{R}$
SKV	The sorting key value of a record $r_i \in \mathbf{R}$
N_i	A node in the index tree data structure
N_{q_j}	The node in the index tree data structure where q_j is inserted
k	The number of the nodes in the index tree data structure
δ	A window expantion threshold that represents the allowed minimum number of candidate records in a window (used with the DySNI-c approach that is described in Section 6.4.2).
θ	A window expansion threshold that represent the similarity between keys of tree nodes (used with the DySNI-s approach that is described in Section 6.4.3).
σ	A window expansion threshold that represents the number of classified matches within a window (used with the DySNI-d approach that is described in Section 6.4.4))

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

- **Sorting Key:** A sorting key (SK) is defined as the list of attributes that are used to sort records in $\mathbf{R}$ alphabetically. SKs are usually generated by concatenating the attributes in the SK list. Selecting SKs generally requires domain knowledge. We propose an automatic key selection algorithm in Chapter 8.

- **Sorting key value:** A sorting key value (SKV) of a record in $\mathbf{R}$ is the value of the attributes used as SK for that record. For example, assuming that a concatenation of the 'Firstname' and 'Surname' attributes from the example records in Figure 6.1 is used as a SK, then the value 'percysmith' would be the SKV generated for $r1$.

The problem of real-time ER is defined as: for each query record q_j in a query stream $\mathbf{Q}$, find all the records in $\mathbf{R}$ that belong to the same entity as q_j, denoted as the set $\mathbf{M}_{q_j}$, in sub-second time, where $\mathbf{M}_{q_j} = \{r_i \mid r_i.eid = q_j.eid, r_i \in \mathbf{R}\}, \mathbf{M}_{q_j} \subseteq \mathbf{R}, q_j \in \mathbf{Q}$. Table 6.1 summarizes the notation that we use.

6.3 Overview of the Approach

The original SNM uses a static array data structure to store the SKV of all records in the data sets that are to be deduplicated or matched [79]. 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 data set. Additionally, finding a certain SKV in a sorted array of length n elements requires $O(nlog(n))$ steps (this includes sorting then searching the array).

Real-time ER on dynamic data sets requires an index data structure with efficient searching, inserting, and retrieving capabilities. Search trees are more efficient than sorted arrays [41] and are commonly used for indexing in different application domains. We propose a tree-based dynamic sorted-neighborhood indexing (**DySNI**) technique that works with real-time ER on dynamic data sets.

6.3.1 Index Data Structure

In this section we describe different possible tree data structures that potentially can be used with our proposed approach.

- A basic binary search tree is a non-balanced tree data structure that consists of nodes and edges to organize data in a hierarchical manner, where each node can have 0, 1 to 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 [41]. Let x and y be nodes in the tree with different key values, i.e. $x.key \neq y.key$ and y being a child node of x, x_L is the left sub-tree of x, and x_R is the right sub-tree 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 required 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 height balanced binary search tree where for each node in the tree the difference between the heights of its left and right sub-trees never exceeds 1 [2, 134, 141]. 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 following operations: insert new node, search for a node, and retrieve next and previous nods in the worst case scenario [2, 134, 141].

- A braided AVL tree is a data structure that combines the properties of both AVL trees and double-linked lists [134]. 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.

Table 6.2: 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	$O(k)$	$O(log(k))$	$O(1)$	$O(log(k))$

- A B-tree is a balanced search tree data structure that is designed to work with secondary storage devices [41]. 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 maximum 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 6 levels, while a binary tree requires 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 (as in B+ tree [40] which is a B-tree that has a link between its leaf nodes). A B+ tree is to be investigated in future work.

Table 6.2 summarizes the complexities of the different operations achieved by the above 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, since this data structure provides efficient retrieval times of neighboring nodes that are required to generate the list of candidate records required to do the matching. Therefore, we use a braided AVL tree in our solution. We define such a tree as:

[**Definition 6.1**] **Braided Tree** (**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_j} if a query record is inserted into the list I of that node. Figure 6.2 illustrates the **BRT** generated based on the small example data set from Figure 6.1.

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 data set **R** unmodified after they are created, since they can provide evidence about earlier queries on in-

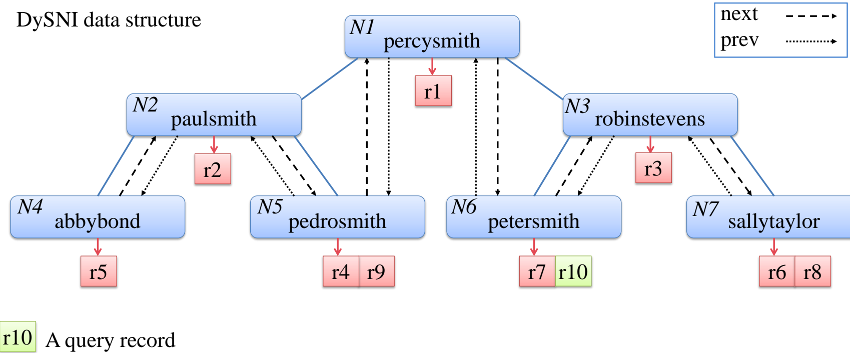


Figure 6.2: The dynamic sorted neighborhood index (DySNI) for the ten records from Figure 6.1 using a **BRT** tree data structure. The sorting key values are the concatenation of 'Firstname' and 'Surname'.

Algorithm 6.1: DySNI – *build*(**R**, SK)

Input:
- Data set: **R**
- Sorting key: SK

Output:
- A **BRT** tree data structure that is filled with all $r_i \in \mathbf{R}$

```
1:  B := createBRT( )                 // Create an empty BRT tree data structure named B
2:  for r_i ∈ R do:
3:     skv := B.generateKey(SK)       // Generate a sorting key value for r_i
4:     N_i := B.findTreeNode(skv)     // Search for skv in B
5:     if N_i == NULL then:
6:        N_i := B.createNode(skv,r_i.id)  // Create a new node N_i with the key skv
7:        B.insert(N_i)               // Insert the created node N_i into B
8:     else:
9:        Append r_i.id to N_i.I      // Append record identifier r_i.id into the list
                                      // of record ids I of the tree node N_i
10: Return B                          // Return B which is filled with all r_i ∈ R
```

dividual 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.

The DySNI has an initial *build phase* where a certain number (possibly none) of records from an existing data set 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.

6.3.2 Building the Index

In this phase (shown in Algorithm 6.1), records are loaded from data set **R** to build the index data structure using the **BRT** tree. First we start, in line 1, by creating an empty **BRT** tree **B**. Then, for all the records in **R** we generate a SKV for each record in **R** (line 3). The SKV become the key value *skv* that is used for creating the nodes in the **BRT** tree.

Algorithm 6.2: DySNI – $query(\mathbf{B},\ q_j,\ \mathbf{S}, \text{SK}, w,\ \mathbf{D})$

Input:
- The built **BRT** data structure: **B**
- Query record: q_j
- Similarity functions: **S**
- Sorting key: SK
- Window size: $w \geq 1$
- Data set table with complete records: **D**

Output:
- Ranked list of matches: **M**

```
1:   skv := B.generateKey(SK, q_j)        // Generate a sorting key value for the query q_j
2:   N_qj := B.findTreeNode(skv)          // Search for skv in B
3:   D[q.id] := q_j                       // Insert the query record q_j into D
4:   if N_qj == NULL then :
5:     N_qj := B.createNode(skv,q_j.id)   // Create a new node N_qj with the key skv
6:   else:
7:     Append q_j.id to N_qj.I            // Append record identifier q_j.id into the list
                                          // of record ids I of the tree node N_qj
8:   C := B.generateWin(N_qj, S, w)       // Generate the candidate records from neighboring
                                          // nodes using a window size w
9:   M := B.compareRecords(C, S, D, q_j)  // Compare query record q_j with all candidate records in C
10:  Sort M according to similarities
11:  Return M
```

In line 4, we search the **BRT** for the generated *skv*. If *skv* has not been indexed earlier, a new node is created and inserted into the **BRT** tree (lines 5-7). On the other hand, if the *skv* already exists in the **BRT** (lines 8-9), we only append the record identifier $r_i.id$ into the identifier's list I of the node N_i with the same key value *skv*. For example, node $N1$ in Figure 6.2 was generated when record $r1$ with SKV 'percysmith' was inserted into the empty index, while for record $r8$ with SKV '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 data set table **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.

6.3.3 Querying the Index

In this phase (shown in Algorithm 6.2), a query record q_j 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 $q_j.id$ is assigned to it (in the *generateKey( )* function). This SKV and $q_j.id$ are then inserted into the **BRT** in the same way as records were inserted during the build phase (lines 4-7). q_j is also added into **D** in line 3.

The window of neighboring nodes can now be created (line 8). The aim of this step is to generate the candidate records that are to be compared with the query record q_j in more detail to find the matching records. The window of neighboring

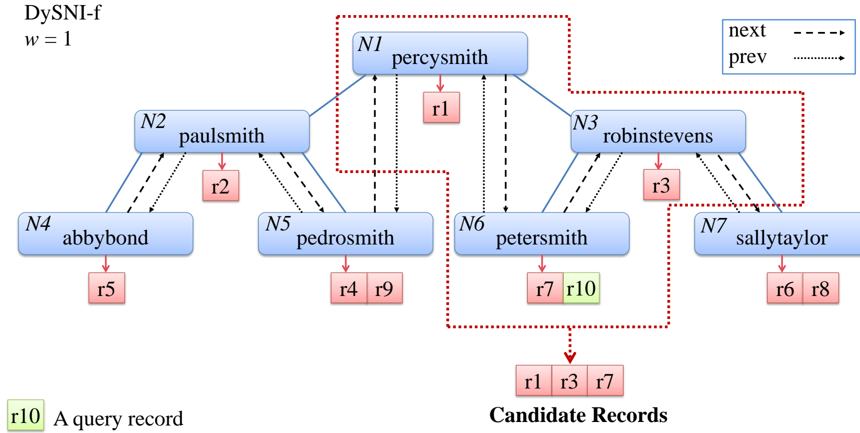


Figure 6.3: The set of candidate records generated using the fixed size window approach (DySNI-f) for query record $r_1 0$, as described in Section 6.4.1 using a window of size $w = 1$.

nodes can be created using different approaches that will be described in the following section. However, in Algorithm 6.2 we are using the fixed window size approach (described in Section 6.4.1). All record identifiers that are stored in the nodes within the window are added to the candidate record set **C**. Whole records (for each record identifier within **C**) are then retrieved from the inverted index or data set table **D**, and the attributes of q_j are compared with the retrieved records using similarity comparison functions [33] appropriate to the content of each attribute (line 9). The compared candidate records are sorted according to their overall similarities with the query record (line 10) and are then returned in the list **M**

6.4 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 [170] where expanding the window is based on the similarities between SKVs, and by [54] where expanding the window is based on the number of classified matches within the window (both [170] and [54] are described in more details in Chapter 3). In the following we describe one static and three adaptive window approaches.

6.4.1 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_j}, the total number of neighboring

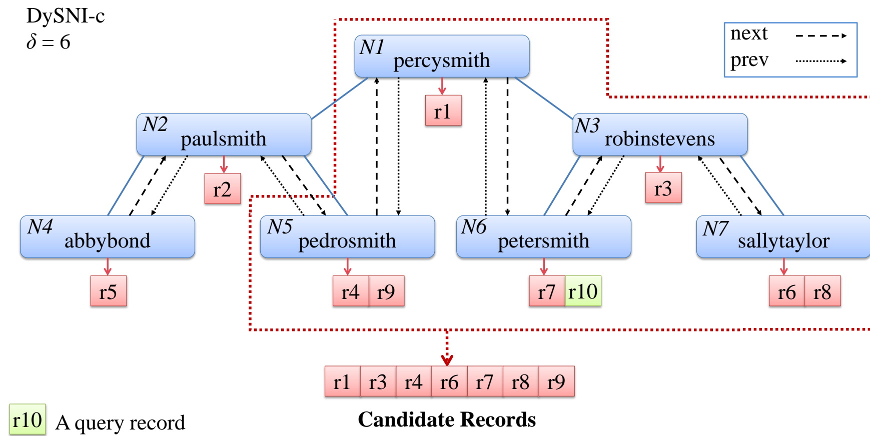


Figure 6.4: The set of candidate records generated using the candidate-based adaptive window approach (DySNI-c) described in Section 6.4.2 using a minimum total number of candidate returned records of $\delta = 6$.

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_j} only (the node where the q_j is inserted).

For the index tree shown in Figure 6.3, 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 $\mathbf{C} = \{r1, r3, r7\}$. 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').

6.4.2 Candidate-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

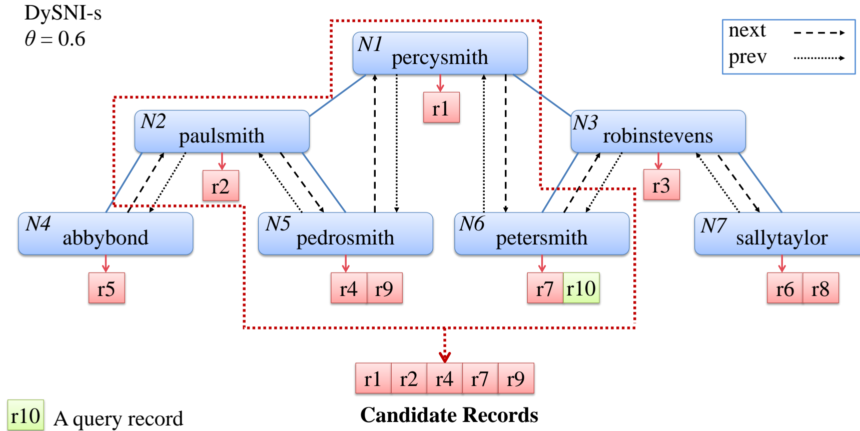


Figure 6.5: The set of candidate records generated using the similarity-based adaptive window approach (DySNI-s) described in Section 6.4.3 using a similarity threshold of $\theta = 0.6$.

N_{q_j} is greater than or equal to the minimum candidate threshold $|\mathbf{C}| >= \delta$, then no expansion is required, 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 required for the total candidate records to reach δ is calculated as $\chi = \delta - |\mathbf{C}|$. A new expansion threshold is set to $\lceil \chi/2 \rceil$ 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 $\lceil \chi/2 \rceil$.

For example, in Figure 6.4, 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 $\chi = 6 - 1 = 5$, and $\lceil 5/2 \rceil = 3$. We therefore need to add a minimum of 3 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 $\mathbf{C} = \{r1, r3, r4, r6, r7, r8, r9\}$. Window expansion stopes at this point since $|\mathbf{C}| >= \delta$.

6.4.3 Similarity-Based Adaptive Window (DySNI-s)

This approach is based on [170] 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 6.3), 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_j} 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

Algorithm 6.3: DySNI-s - *generateWin*$(N_{q_j},\ \theta,\ \mathbf{S})$

Input:
- Query node: N_{q_j}
- Similarity threshold: θ
- Similarity function: **S**

Output:
- Candidate record set: **C**

```
// Window expansion in the (next) direction
1:   C := N_qj.I                               // Add all record ids from N_qj.I into C
2:   next_nd := N_qj.next                      // Get next node
3:   while sim(N_qj.skv, next_nd.skv) > θ do :
4:     C := C ∪ next_nd.I                      // Add all record ids from next_nd.I into C
5:     next_nd := next_nd.next                 // Get next node

// Window expansion in the (previous) direction
6:   prev_nd := N_qj.prev                      // Get previous node
7:   while sim(N_qj.skv, prev_nd.skv) > θ do :
8:     C := C ∪ prev_nd.I                      // Add all record ids from prev_nd.I into C
9:     prev_nd := prev_nd.prev                 // Get previous node
10:  Return C
```

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 6.5 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 Jaro-Winkler string comparison function [162]. This gives us a low value of $sim(\text{`}petersmith\text{'},\text{`}robinstevens\text{'}) = 0.0$. Because $0.0 < \theta$ 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 **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 $sim(\text{`}petersmith\text{'},\text{`}percysmith\text{'}) = 0.89$. Therefore $N1$ and its record identifier $r1$ is added to **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 **C**. The final set of candidate records is $\mathbf{C} = \{r1, r2, r4, r7, r9\}$. As illustrated in this example, the DySNI-s approach avoids unnecessary comparisons (like records in node $N3$) that was included in the window using the DySNI-f approach and includes matches (like records in nodes $N5$ and $N2$) that were missed using the DySNI-f approach.

6.4.4 Duplicate-Based Adaptive Window (DySNI-d)

This third adaptive approach is based on [54]. 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

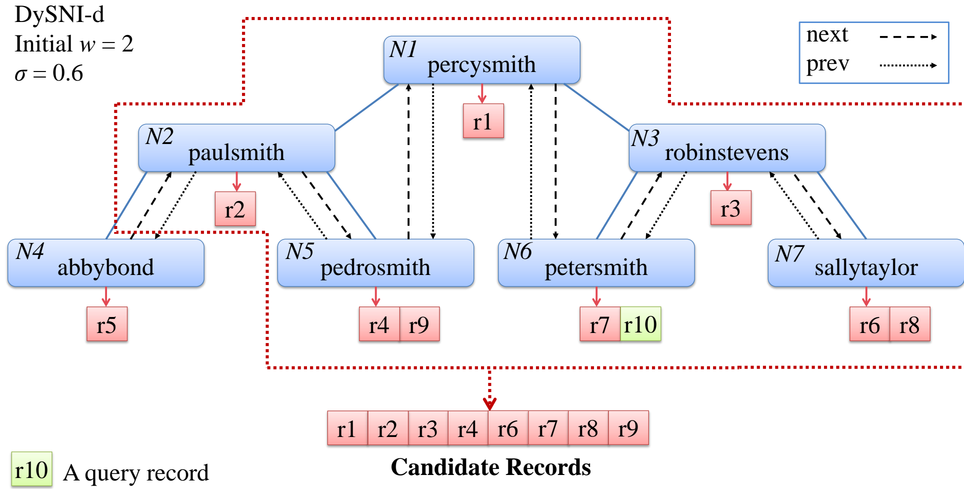


Figure 6.6: The set of candidate records generated using the duplicate-based adaptive window approach (DySNI-d) described in Section 6.4.4 using an expansion threshold of $\sigma = 0.6$.

the last record in the index to match records in the whole data set. The more matches are found, the larger the window size becomes. However, if no or only a small number of matches for a query record are found in the window, then this approach assumes that there are no more matches further away (based on the fact that all records are sorted alphabetically according to a sorting key and similar records are likely closer to each other), and therefore there is no need to increase the size of the window. In our approach (shown in Algorithm 6.4), 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 non-matching record pairs. The query record is compared with all candidate records that are in the initial window. Records that have a similarity above a certain threshold with the query record are classified as matches, all others as non-matches. Assume that the number of classified matches is m out of a total of c candidate records compared with the query record, and assume that the expansion threshold (expansion ratio) is σ [54]. A window is expanded to the next tree node if the following holds:

$$\frac{m}{c} \geq \sigma \tag{6.1}$$

In the same way as the similarity-based adaptive approach expands the window in each direction independently, the duplicate-based approach also calculates Equation 6.1 independently in the forward (next) and backward (prev) direction (as shown in Algorithm 6.4).

Algorithm 6.4: DySNI-d - *generateWin*(q_j, N_{q_j}, w, $\mathbf{S}$, σ)

Input:
- Query record: q_j
- Query node: N_{q_j}
- Initial window size: $w \geq 1$
- Similarity functions: $\mathbf{S}$
- Expansion threshold: σ

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

```
// Window expansion in the (next) direction
1:  C := N_{q_j}.I                              // Add record id's I within N_{q_j} to C
2:  c_n := getNextCandidates(N_{q_j}, w)        // Get candidates from the next w nodes to N_{q_j}
3:  C := C ∪ c_n                                // Add candidates from the next w nodes to C
4:  c_next := |c_n|                             // The number of candidates in the next direction
5:  m_next := getNumMatches(c_n, q_j, S)        // The number of matches in the next direction
6:  next_nd := N_{q_(j+w)}.next                 // Get the next node after w neighboring nodes
                                                // from N_{q_j} (in the next direction)
7:  while m_next/c_next ≥ σ do :                // Expand the window in the
                                                // (next) direction while condition is true
8:    C := C ∪ next_nd.I                        // Add record ids I of next_nd to C
9:    next_nd := next_nd.next                   // Get the node in the next direction of next_nd
10:   m_next := getNumMatches(next_nd, q_j, S)  // Get the number of matches in the new next_nd
11:   c_next := getNumCandidates(next_nd)       // Get the number of candidates in the new next_nd

// Window expansion in the (previous) direction
12: c_p := getPrevCandidates(N_{q_j}, w)        // Get candidates from the previous w nodes to N_{q_j}
13: C := C ∪ c_p                                // Add candidates from the previous w nodes to C
14: c_prev := |c_p|                             // The number of candidates in the prev direction
15: m_prev := getNumMatches(c_p, q_j, S)        // The number of matches in the previous direction
16: prev_nd := N_{q_(j−w)}.prev                 // Get the previous node before w neighboring
                                                // nodes from N_{q_j} (in the previous direction)
17: while m_prev/c_prev ≥ σ do :                // Expand the window from the
                                                // (previous) direction while condition is true
18:   C := C ∪ prev_nd.I                        // Add record ids I of prev_n to C
19:   prev_nd := prev_nd.prev                   // Get the node in the previous direction of prev_nd
20:   m_prev := getNumMatches(prev_nd, q_j, S)  // Get the number of matches in the new prev_nd
21:   c_prev := getNumCandidates(prev_nd)       // Get the number of candidates in the new prev_nd
22: Return C
```

Let us use the example in Figure 6.6, and assume that the initial window size is $w = 2$, the expansion threshold $\sigma = 0.6$, and $r10$ is 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.

As for the window expansion in the forward (next) direction, the window cannot expand since the last node in the tree, N_7, is already included in the initial window size. However, 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 6.1. Based on the full example records in Figure 6.1, assume that both $r1$ and $r4$ are matching records (so number of matching records in the previous window is

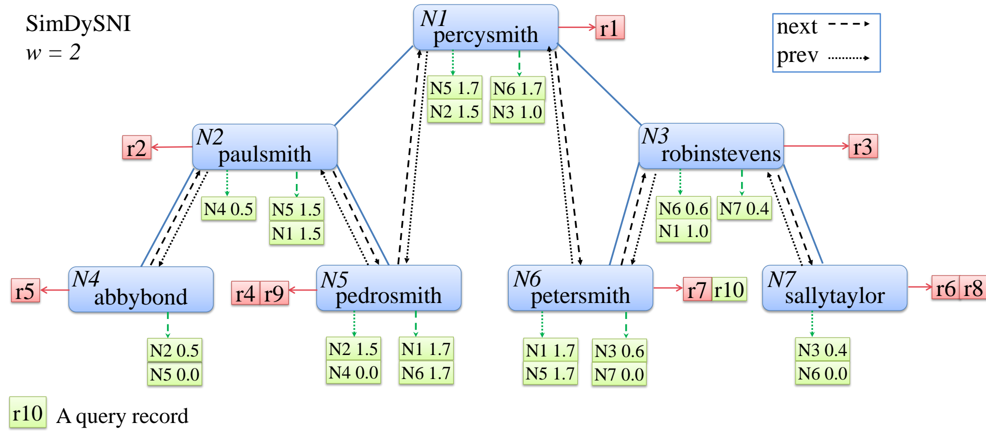


Figure 6.7: Example of the similarity-based dynamic sorted neighborhood index (SimDySNI) for the ten records from the table in Figure 6.1. The sorting key values are the concatenation of 'Firstname' and 'Surname' values, and the window size for pre-calculation of similarities is set as $w = 2$. The pre-calculated similarities are generated using the Jaro-Winkler similarity function [33].

$m = 2$). Because $2/3 \geq \sigma$, this means that the window will expand in the backward (previous) direction to include $N2$. The expansion process will continue until $m/c < \sigma$ which is reached after including node $N2$ in the window. Therefore, the final set of candidate records is $\mathbf{C} = \{r1, r2, r3, r4, r6, r7, r8, r9\}$.

6.5 Similarity-Based Dynamic Sorted Neighborhood Index

The idea behind the similarity-based dynamic sorted neighborhood index (SimDySNI) is to pre-calculate the similarities between the attribute values used to generate the SKVs, and to store these similarities in the tree. These pre-calculated 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 pre-calculated similarities are stored within nodes of the tree index.

[Definition 6.2] Similarity-based BRT (S-BRT): is a **BRT** tree with two extra lists attached to its nodes as illustrated in Figure 6.7. These lists contain the pre-calculated 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 pre-calculated similarities between this node's SKV and the SKVs of all neighboring nodes within the window in the previous and following (next) directions respectively, while the other node elements are the same as in Definition 1.

6.5.1 Build and similarity calculation phases

The build phase is the same as the build phase of the DySNI described in Section 6.3.2. However, after the build phase finishes, a similarity calculation phase is conducted where the pre-calculated 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 sum of the pre-calculated similarities of all attributes used as SK are stored in the similarity lists S_p and S_n. For example, to calculate the similarity between the SKVs 'petersmith' and 'percysmith', we first calculate the similarity between the first attribute values $sim(\text{'}peter\text{'}, \text{'}percy\text{'}) = 0.7$, then the similarity between the second attribute values $sim(\text{'}smith\text{'}, \text{'}smith\text{'}) = 1.0$. The sum of those calculations $sum = 1.7$ is then stored in the similarity lists.

The calculated similarities are stored into the lists S_p and S_n for each tree node. Figure 6.7 gives an example for SimDySNI when using a fixed window of size $w = 2$ and the Jaro-Winkler similarity function [33]. 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 6.4.3) which means that in this case the pre-calculated 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.

6.5.2 Query phase

In the query phase (shown in Algorithm 6.5) of the SimDySNI approach we benefit from the pre-calculated similarities that are stored in each node to reduce the time required to resolve a query. Querying the built index (from the build phase) is based on two cases (as shown in Algorithm 6.5): the first case occurs when the SKV of a query record q_j is new and it has not been indexed earlier. The second case occurs when the SKV of a query record q_j has been indexed previously and it already exists in the index data structure.

1. **New SKV:** In this case (lines 4-10), because the SKV of a query record q_j is new, we create a new node N_{q_j} for this query and we resolve it using the original DySNI approach as described in Section 6.3.3 (i.e. without benefiting from any pre-calculated similarities). After resolving the query, we generate the two pre-calculated similarity lists (i.e. S_p and S_n) for both directions for N_{q_j} 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) (line 8). Next, we update the similarity lists for all w previous and next tree nodes of the newly inserted tree node (lines 9-10). This step ensures that the pre-calculated similarities are up-to-date at any time.

Algorithm 6.5: SimDySNI (fixed window) – *query*(**B**, q_j, **S**, SK, w, **D**)

```
Input:
- The built S-BRT data structure: B
- Query record: q_j
- Similarity functions: S
- Sorting key attributes: SK
- Window size: w ≥ 1
- Database table with complete records: D
Output:
- Ranked list of matches: M
1:  skv := generateKey(SK, q_j)              // Generate a sorting key skv for q_j
2:  N_qj := B.findTreeNode(skv)              // Search for skv in S-BRT
3:  D[q_j.id] := q_j                         // Insert the query record q_j into D

// New SKVs
4:  if N_qj := NULL then :
5:    N_qj := B.createNode(skv, q_j.id)      // Create a new node N_qj with the key skv
6:    C := B.generateWin(N_qj, w)            // Generate the candidate records from
                                             // neighboring nodes using a window size w
7:    M := B.compareRecords(C, S, D, q_j)    // Compare query record q_j with all
                                             // candidate records in C
8:    B.preCalcNodeSimilarities(N_qj, w, S)  // Calculate similarities for node N_qj
9:    B.updateSimNextNodes(N_qj, w, S)       // Calculate similarities for neighboring
                                             // nodes in the next direction
10:   B.updateSimPreviousNodes(N_qj, w, S)   // Calculate similarities for neighboring
                                             // nodes in the previous direction
// Indexed SKV
11: else:
12:   Append q_j.id to N_qj.I                // Append record identifier q_j.id into N_qj.I
13:   C := N_qj.I ∪ N_qj.Get_I_from_nxt_wind()   // Generate candidate records
            ∪ N_qj.Get_I_from_prv_wind()
14:   M := B.comparePreCalcRec(C, S, D, q_j) // Compare query record q_j with all
                                             // candidate records in C while benefiting
                                             // from the stored pre-calculated similarities
15: Sort M according to similarities
16: Return M
```

2. **Indexed SKV:** In this case (lines 11-14), 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 pre-calculated 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 pre-calculated 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 pre-calculated, but at the cost of a larger tree (as likely a larger number of distinct SKVs will be generated). In our experi-

mental 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 pre-calculated similarities (i.e. indexed queries above), 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 6.1 and the index tree shown in Figure 6.7, 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 $\mathbf{C} = \{r1, r3, r4, r6, r7, r8, r9\}$. To compare query $r10$ with these candidate records we first retrieve the actual records from the inverted index or database table $\mathbf{D}$ of record details, and for each candidate record we retrieve the pre-calculated similarity from the SimDySNI index. For example, the pre-calculated similarity between query record $r10$ and candidate record $r1$ is $sim = 1.7$ as retrieved from the previous list of node $N6$. This similarity corresponds to the sum of the pre-calculated Jaro-Winkler similarities of the 'Firstname' and 'Surname' attributes (each is between 0 and 1). To calculate the overall similarity between $r10$ and $r1$ we only have to calculate the similarities for the 'City' and the 'Zipcode' attribute values of these two records (the attributes that are not used as SKs), then these calculated similarities are summed with the pre-calculated similarity of (1.7) that is stored within the index. The overall similarity is then used to decide if a candidate record is a match or non-match. As we show in Section 6.7, this process improves query time by reducing the time required to calculate the overall similarity between record pairs (which is achieved by pre-calculating the similarities of attributes that are used as SKs).

6.6 Estimating the Number of Comparisons Required for the Proposed Approach

As described in Section 5.5, the comparison step in the ER process is usually the most time consuming step because of the calculations performed when candidate records are compared. Thus, estimating the number of comparisons beforehand gives users an insight into the expected run time required to match a query record with a data set of a certain size. In this section, we provide a way of estimating the number of generated candidate records using DySNI.

The DySNI approach groups records in the data set that have the same SKVs into one tree node (i.e. block) of the index data structure. To estimate 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 6.8, the possible SKs that can be used to build the index (using the different attributes in the NC data set described in Chapter 4) either have a Zipfian like distribution (like 'Firstname' and 'Surname'), or a distribution

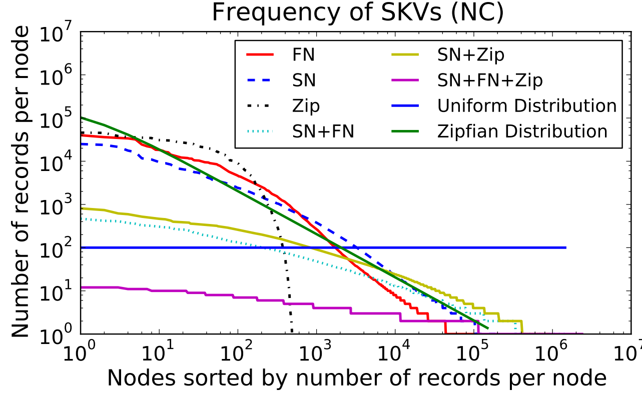


Figure 6.8: Number of records in tree nodes generated using different SKVs for the NC data set described in Chapter 4 (FN=Firstname, SN= Surname, Zip = Zipcode).

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 maximum and the minimum number of comparisons based on uniform and Zipfian distributions (similar to what we do in Chapter 5).

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 data set 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 of size $2w + 1$ (for a fixed-window approach) and the number of records n in the data set. Therefore, the estimated number of generated candidate records in **C** is:

$$|\mathbf{C}| = \frac{n(2w+1)}{k} \tag{6.2}$$

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 data set, 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:

$$S_i = \frac{1/i}{\sum_{i=1}^{k}(\frac{1}{i})} * n, \tag{6.3}$$

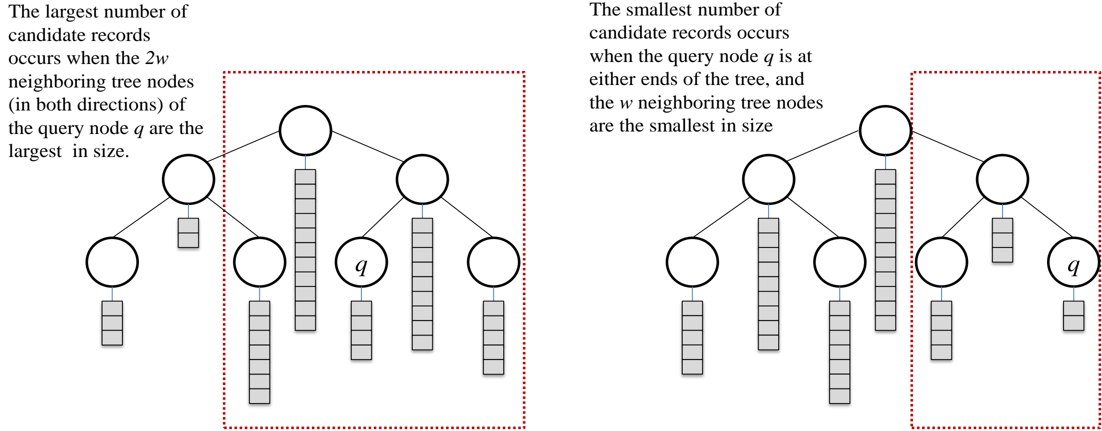


Figure 6.9: Illustration of the situations where the maximum (left) and minimum (right) number of candidate records are obtained for the DySNI. The window size is set to $w = 2$.

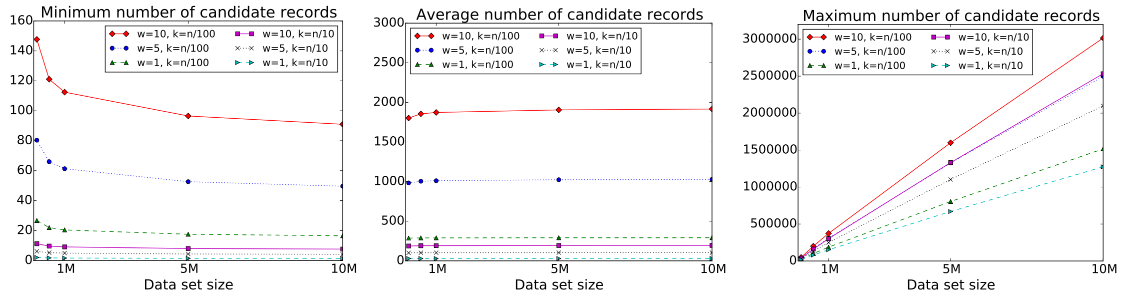


Figure 6.10: The minimum, average, and maximum of number of candidate records estimated based on Equations 6.3 and 6.4, with n the number of records in a data set and k the number of nodes in the tree index.

where the denominator is the Harmonic number of the partial harmonic sum [34]. The number of candidate record S_w in a window that includes $2w + 1$ nodes is then calculated as:

$$S_w = \sum_{i-w}^{i+w} S_i \tag{6.4}$$

To estimate the minimum and maximum numbers of candidate records for a query q_j generated using a fixed-size window approach (assuming window size w) we calculate S_w according to the illustration given in Figure 6.9. 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 candidates record are generated if the query node N_{q_j} is at either end of the tree, and the w neighboring tree nodes are the smallest in size.

Figure 6.10 shows estimates of the minimum, average, and maximum number of candidate records for increasing sizes of data sets based on Equations 6.3 and 6.4. From the figure we can see that the maximum estimated number of candidate records

Table 6.3: Data sets summary (described in more details in Chapter 4).

Data set	Type	Number of records	Number of entities
30% of NC	Real	2,399,170	2,282,834
OZ-1	Real-world (modified)	345,876	172,938
Febrl-5	Synthetic	100,000	20,000
Febrl-10	Synthetic	100,000	10,000
Febrl-20	Synthetic	100,000	5,000

increases linearly with the growing size of the data set, while the minimum estimated number decreases with larger data sets, 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 data set size. 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 6.13 where the average query time is nearly constant with the growing size of the index.

6.7 Experimental Evaluation

In this section we describe the experiments conducted to evaluate the proposed DySNI approach and all its variations. The data sets used in these experiments (summarized in Table 6.3), the evaluation measures, and the implementation environment are all described in details in Chapter 4. To evaluate our approaches we conducted several sets of experiments as described below. The SK used for conducting all of the following experiments is the concatenation of 'Surname+Firstname' attribute values. This SK is selected based on domain knowledge and experimental investigation using different SKs. However, an automatic key selection approach is proposed in Chapter 8.

6.7.1 Efficiency of Generating Candidate Records Using Different Tree Data Structures

We evaluate the efficiency of the first three data structures described in Section 6.3.1 with regard to generating the candidate records by comparing the average time required 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 6.11 illustrate 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 a braided AVL tree (**BRT**) in the DySNI.

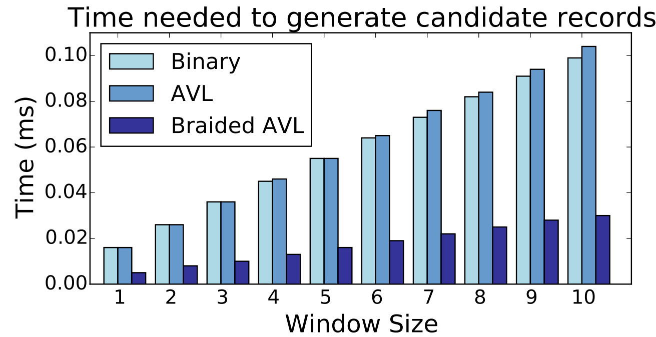


Figure 6.11: The Average time required to generate candidate records for different types of trees. The plot is generated using the OZ-1 data set described in Chapter 4.

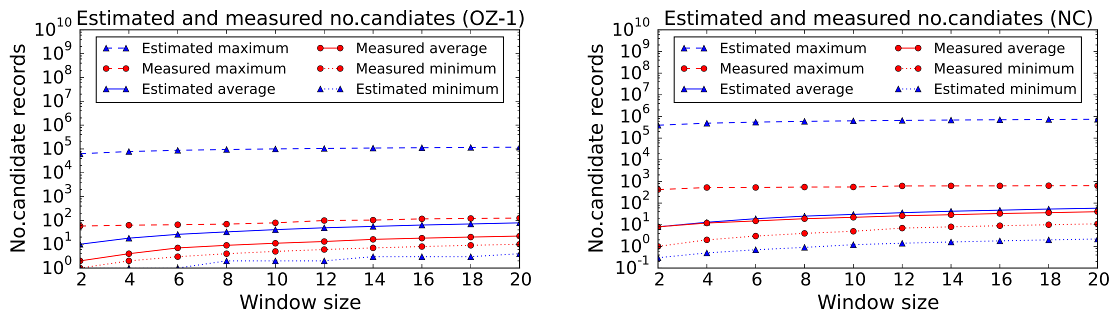


Figure 6.12: The estimated number of candidate records for the OZ-1 and 30% of the NC data sets using Equation 6.4, compared with the measured number of candidate records required for running DySNI-f (described in Section 6.4.1) on both the NC and OZ-1 data sets using a concatenation of 'Surname+Firstname' as SK for different window sizes.

6.7.2 Estimation of the Number of Candidate Records

We conduct this set of experiments to evaluate the estimation functions proposed in Section 6.6. We collected the minimum, maximum, and average number of candidate records used when running the DySNI-f approach on 30% of the NC data set (with $n = 2,567,638$ and $k = 1,634,650$) and on the OZ-1 data set (with $n = 345,876$ and $k = 160,058$) using different window sizes ($2 \leq w \leq 10$). Then we used the proposed estimation function (in Equation 6.4) 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 data sets for window sizes ($2 \leq w \leq 10$) and plotted the results in Figure 6.12.

The results show that the measured and the estimated average number of candidate records are highly similar in both the OZ-1 and the NC data sets. 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 6.2 can successfully estimate the number of candidate records for both OZ-1 and NC data sets.

Table 6.4: Number of nodes generated using various SKs for 30% of the NC data set with a total number of records of $n = 2,567,639$. (FN=Firstname, SN= Surname, Zip = Zipcode).

SK	Number of nodes (k)
FN	120,632
SN	194,090
Zip	508
SN + FN	1,634,650
SN + Zip	897,215
SN + FN + Zip	2,237,069

6.7.3 Effects 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 data set. Figure 6.8 and Table 6.4 show that using single attribute values as SKs results in trees with smaller number of 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 data sets.

6.7.4 Scalability of the Proposed Solutions

In this set of experiments we evaluate whether the proposed DySNI scales to large data sets while facilitating real-time ER. We measure 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 data structure. These experiments are conducted on 2,399,170 records from the NC data sets. The four window approaches (proposed in Section 6.4) for generating the set of candidate records are evaluated. Threshold selection for the various proposed approaches is based on achieving similar recall values (around 73%) and similar average number of comparisons (which is the number of records compared with the query record in order to be resolved) for all evaluated approaches. Note that the achieved recall value is affected by the used thresholds. The affect of using different thresholds on the effectiveness and the efficiency of the different window approaches is discussed in Section 6.7.5.

As can be seen from Figure 6.13, 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. 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,

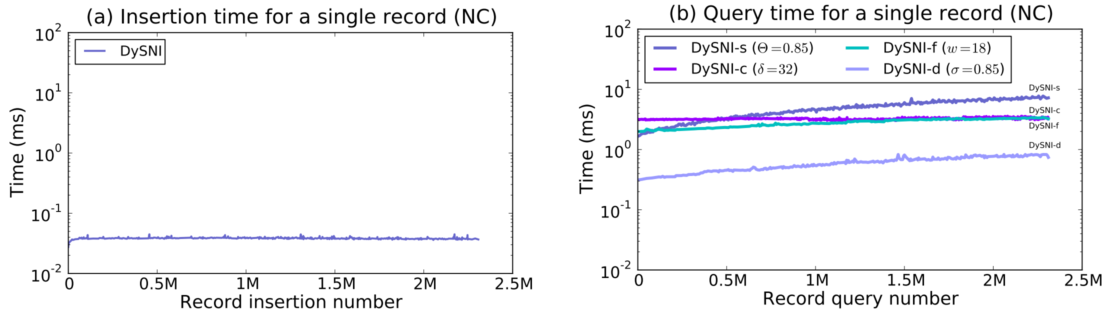


Figure 6.13: Plot (a) shows the average time required for inserting a single record into the index. Plot (b) illustrates the average time required for querying the growing index. A subset of 30% of the NC data set is used (M = Million). All compared approaches give similar recall values.

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 3.0 milliseconds (ms) per query record.

For the insertion times achieved by the DySNI approach (shown in Figure 6.13), the worst case complexity is equal to $\log(k)$, where k is the number of nodes (i.e. unique key values) in the tree data structure. However, for personal information data sets (which we used in our scalability experiments) the number of unique attribute values is small. For instance, the number of unique values for the 'Firstname' attribute in 30% of the NC data set (2,399,170 records) is 120,632, for the 'Surname' attribute is 194,090, and for the 'Zipcode' attribute is 508. This mean that if we, for example, selected the 'Firstname' attribute as a SK, the insertion times will flatten after inserting 120,632 records into the index data structure because the value of k will not increase more than 120,632 while inserting the remaining of the 2,399,170 records.

6.7.5 Effects of Using Different Thresholds on Quality and Efficiency

In this set of experiments we investigate the effect of using different window sizes and thresholds on the quality of the obtained results and the efficiency of the approaches. The OZ-1 data set (with only one duplicate per record) is used for this set of experiments.

- Fixed size window (DySNI-f): We investigate using different window sizes for this approach by measuring recall and average query times required to resolve query records. As shown in Figure 6.14 (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 required to resolve queries will also increase. This means, as one would expect,

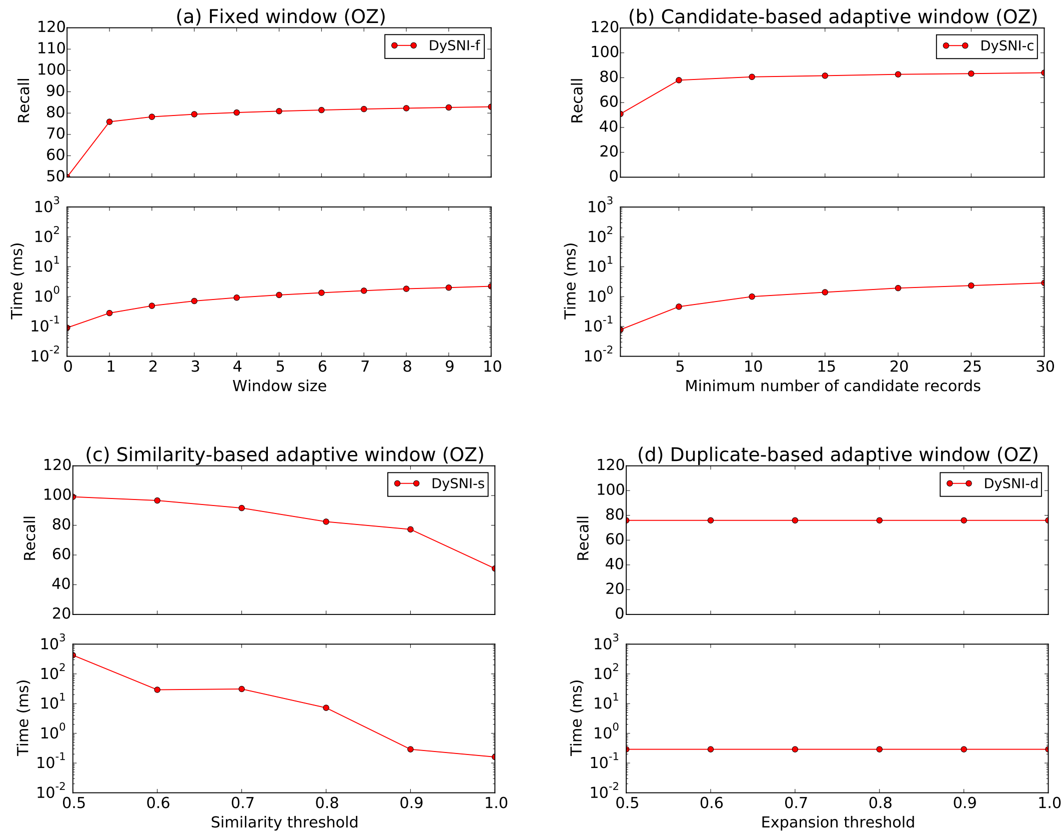


Figure 6.14: Recall and time measures achieved by the different DySNI window approaches. All plots are generated using the OZ-1 data set. Each entity can have one duplicate.

that larger window sizes can achieve better recall values but at the cost of increasing query time.

- Candidate-based adaptive window (DySNI-c): We investigate using different minimum number of candidate records for the candidate-based adaptive window approach; the results in Figure 6.14 (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 required to resolve queries.

- Similarity-based adaptive window (DySNI-s): We investigate using different similarity thresholds to expand the window on both sides for this adaptive window approach. Figure 6.14 (c) shows that smaller threshold values achieve better recall values but more time is required 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.

- Duplicate-based adaptive window (DySNI-d): We investigate using different expansion thresholds for the duplicate-based adaptive window approach. Figure 6.14 (d) shows that both recall and average query times are almost constant

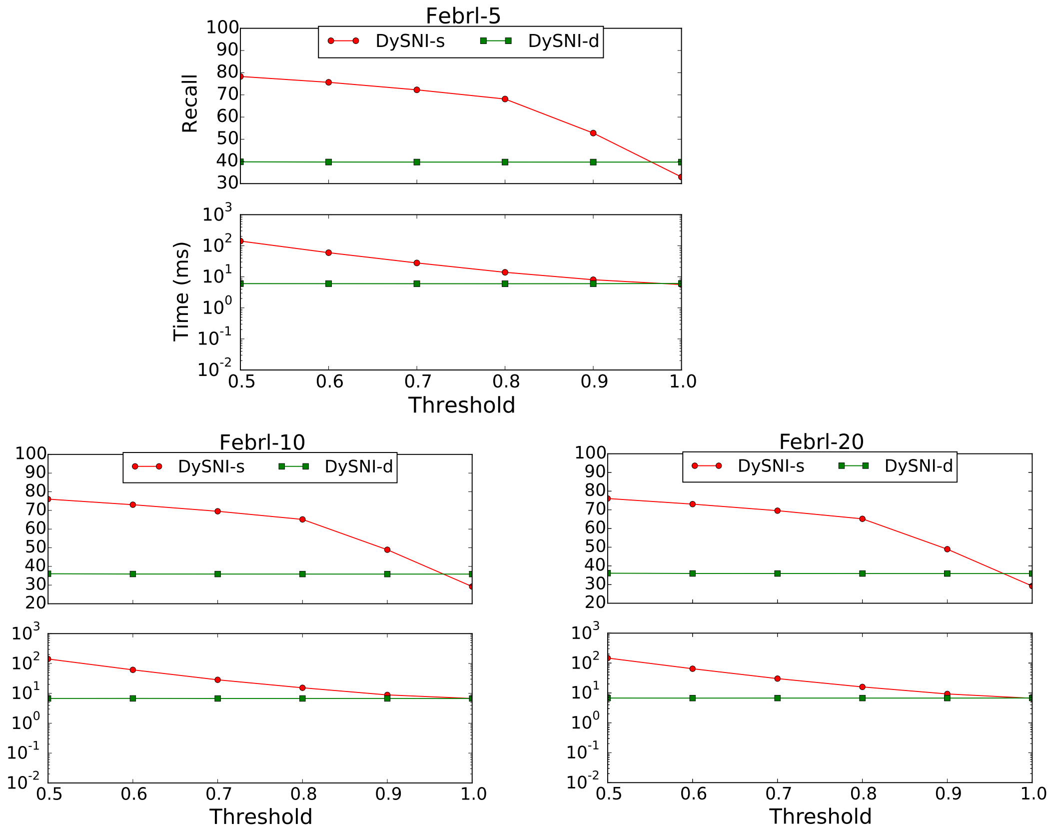


Figure 6.15: Recall and time measures for similarity and duplicate adaptive approaches using the Febrl data sets. Note that the threshold is different between all approaches; for DySNI-s it represents the similarity threshold between the SKV of the query's node and neighboring nodes, while for DySNI-d it represents the ratio between the found true matches and the number of candidate records within the window.

when using different expansion thresholds. This means that the expansion of the window is very limited. The reason for this is that each tree node in the DySNI-d contains all records with the same SKV and most of the duplicates are found in the node of the query record 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, based on [54], is not suitable for DySNI.

6.7.6 Effects of Having Different Number of Duplicates

The aim of this set of experiments is to investigate the effect of the number of duplicate records on recall, query times, and on the expansion of the adaptive window in both the DySNI-s and DySNI-d approaches. These experiments are conducted using the Febrl data sets described in Chapter 4. From the results shown in Figure 6.15 we can see that in general the recall values achieved by the two adaptive approaches are less than the recall values achieved in Figure 6.14 in the previous set of experiments.

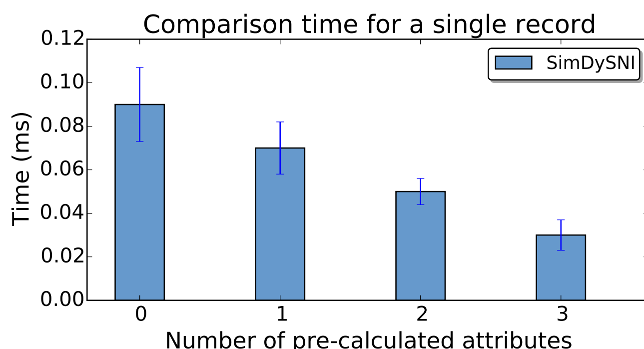


Figure 6.16: The average times required to compare a query record with a single candidate record for the SimDySNI approach using different numbers of attributes as sorting keys on the OZ-1 data set.

This is due to the fact that for the Febrl data sets the average number of duplicates ranges between 5 and 20 per entity, while for the data set used in Figure 6.14 each entity only has a maximum of one duplicate. Having a larger number of modified duplicates in the data sets 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 (proposed in Chapter 7) to increase the chance of having records that represent the same entities close to each other. However, from the results we can see that the DySNI-s approach achieved better recall values than the DySNI-d approach.

In addition, the results show that having a different number of duplicates in the Febrl data sets does not affect window expansion for DySNI-d, and similar to the results from the OZ-1 data set 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.

6.7.7 Effects of Pre-calculating Similarities on Comparison Time

Here we investigate how the SimDySNI is able to improve query time. First, we measure the average time required 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 approach (as shown in Algorithm 6.5) with a different number of attributes being used as the SK. We ran the experiments using the OZ-1 data set with 1, 2 and 3 attributes used as SKs for different possible combinations of the four attributes: 'Firstname', 'Surname', 'Suburb', and 'Postcode'. The average

Table 6.5: The average total query time (in ms) for indexed queries (in Section 6.5.2) using various SKs. The OZ-1 data set was used (FN=Firstname, SN= Surname, Post = Postcode).

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 %

comparison times over these combinations are shown in Figure 6.16. The results show that for queries where the node is pre-existing, 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 result show that for a one-attribute sorting key the reduction in the comparison time 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 indexed queries (where the SKV of the query record already exists in the index) for both DySNI and SimDySNI approaches. Table 6.5 presents the average overall query time required for indexed queries using various SKVs for of the OZ-1 data set. The results show that using SimDySNI for indexed queries has improved the overall average query time by 44% to 93% for the various SKVs. Table 6.6 provides the percentages of both new and indexed queries when using various SKs for the NC data set. Results show that between 13% and 99% of arriving queries can benefit from query time improvement by SimDySNI for indexed queries for the different SKs.

6.7.8 Required Memory Size

Table 6.6 shows the memory requirements of the **BRT** data structure used in the DySNI approach, and the **SBRT** data structure used in the SimDySNI approach 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.

6.7.9 General Discussion

The experimental results described in this chapter 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 data sets makes 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.

Table 6.6: Required memory for different tree types using various SKs for 30% of the NC data set. (FN=Firstname, SN= Surname, Zip = Zipcode).

SK	Number of nodes	New SKVs(%)	Indexed SKVs(%)	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

The similarity-based (DySNI-s) window approach shows higher 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 its window expansion and reduces 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. From the presented results we conclude that the DySNI-s is suitable for applications that require high quality query matching results, and DySNI-f and DySNI-c approaches can be used with applications that requires a controlled time for resolving queries.

Moreover, our results illustrate that using the similarity-based SimDySNI reduces the average comparison time between 20-70% (based on the number of attributes used to generate SKV) while it increases the memory footprint between 13% and 40% for various SKs. 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 data set. However, like any sorting-based indexing technique, the DySNI has the drawback of sensitivity to errors and variations at the beginning of SKVs. This drawback is addressed in the next chapter where we propose a forest-based index with multiple tree data structures that uses multiple distinct SKs to build the index.

6.8 Summary

In this chapter we proposed a dynamic tree-based sorted neighborhood indexing technique that can be used for real-time ER on large data sets. The technique was shown to be scalable with large data sets as it has fast insertion and query times. We improved query times by proposing a variation where we pre-calculate the similarities between the attribute values that are used to generate the sorting key values. We investigated several approaches to generated candidate records using both a

fixed size window and various adaptive window techniques. Our evaluation results 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.

Some future research directions that can be extended from our work in this chapter include extend the proposed DySNI by using a combination of a B+ tree and an AVL braided tree to create a disk-based memory solution that allows indexing of very large data sets that do not fit into main memory, and explore how DySNI can be integrated with classification and clustering techniques [72] to make the complete ER pipeline applicable for real-time matching.

Chapter 7

Forest-Based Dynamic Sorted Neighborhood Index for Real-Time Entity Resolution

As described in Chapter 3, sorting-based indexing techniques are sensitive to variations and errors that occur at the beginning of attribute values that are used as sorting keys. To overcome this problem, in this chapter we propose a forest-based indexing technique that uses multiple distinct trees in the index data structure where each tree has a unique sorting key. In Section 7.2 we summarize the notation that we use in this chapter. Then, we describe our proposed approach in Section 7.3, and in Section 7.4 we analyze the approach with regard to estimating the number of comparisons that are required to resolve query records. We describe the experimental evaluation that we use to evaluate the approach in Section 7.5, and summarize our findings in Section 7.6.

7.1 Introduction

Sorting-based indexing techniques, whether static such as the sorted neighborhood method (SNM) [79] or dynamic such as the dynamic sorted neighborhood index (DySNI) proposed in Chapter 6, aim to reduce the search space by bringing similar records closer to each other [33] and only comparing records that are within a certain distance from each other. This is achieved by sorting records within data sets alphabetically using a *sorting key* criterion and then using a window of a certain size to generate candidate records to be compared and matched. However, a major drawback of sorting-based techniques is their sensitivity towards errors and variations that occur at the beginning of attribute values that are used as a sorting key. This can potentially place similar records far away from each other in the index and can affect the quality of the matching process.

This problem was addressed, for the static SNM, in [79] by running the indexing process several times using different sorting keys. However, this technique is not suitable for use with real-time entity resolution (ER) as it only works with batch

Table 7.1: Summary of main notations used in this chapter

$\mathbf{R}$	A data set of records about known entities
$\mathbf{A}$	A set of attributes $\{a_1, a_2, \ldots, a_{\vert\mathbf{A}\vert}\}$ for each $r_i \in \mathbf{R}$
$\mathbf{Q}$	A stream of query records
$\mathbf{C}$	A list of candidate records for a query q_j
$\mathbf{D}$	An inverted index or disk-based data set table
$\mathbf{M}_{q_j}$	A set of all records in $\mathbf{R}$ that belong to the same entity of a query q_j
r_i	A record in $\mathbf{R}$
$r_i.id$	Unique identifier for r_i
$r_i.eid$	Entity identifier for r_i
q_j	A query in $\mathbf{Q}$
$q_j.id$	Unique identifier for q_j
$q_j.eid$	Entity identifier for q_j
n	The size of data set $\mathbf{R}$
w	Window size used to generate candidate records
SK	The sorting key that is used to build a single tree
SKL	The list of SKs that are used to build all trees in the index data structure
SKV	The sorting key value of a record $r_i \in \mathbf{R}$
k	The number of the nodes in the index tree data structure
t	Number of trees in the index data structure

processing ER algorithms that match and resolve all records in one or more data set(s) rather than resolving those relating to a single query record. In this chapter, we propose a multiple tree dynamic indexing technique that is tailored for real-time ER. The aim of this approach is to overcome the drawback discussed above and improve the matching quality by reducing the effect of errors and variations that occur at the beginning of attribute values that are used as sorting keys. We also investigate the effect using different numbers of trees in the index data structure on its effectiveness and efficiency. Moreover, we examine using different SKs to build the index data structure and investigate which SKs are more suitable for use with real-time ER.

7.2 Terminology and Notation

In this section we summarize the terminology and the notation that we use in this chapter:

- **Data set:** We assume that data set $\mathbf{R} = \{r_1, r_2, \ldots, r_{|\mathbf{R}|}\}$ contains records of known entities. Each $r_i \in \mathbf{R}$ has a unique record identifier $r_i.id$ and an entity identifier $r_i.eid$. Records in $\mathbf{R}$ are described by a set of attributes, denoted as $\mathbf{A} = \{a_1, a_2, \ldots, a_{|\mathbf{A}|}\}$. All records in $\mathbf{R}$ are assumed to have the same attribute structure.

- **Query stream:** We assume that a stream of query records $\mathbf{Q} = \{q_1, q_2, \ldots, q_{|\mathbf{Q}|}\}$ is to be matched with $\mathbf{R}$. Each $q_j \in \mathbf{Q}$ is given a unique identifier $q_j.id \neq r_i.id$, $\forall r_i \in \mathbf{R}$; and has the same attribute structure as records in $\mathbf{R}$. It is assumed that q_j is to be added to $\mathbf{R}$ 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 $\mathbf{R}$ alphabetically. SKs are usually generated by concatenating

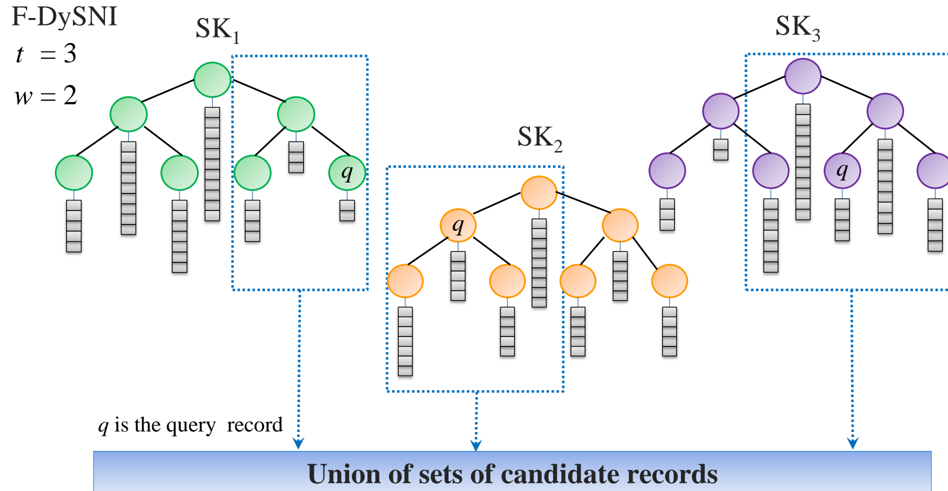


Figure 7.1: The forest-based dynamic sorted neighborhood index (F-DySNI). $t = 3$ is the number of trees used in the index. $w = 2$ is number of the nodes included in the window in each side. SK_1, Sk_2, and Sk_3 are the sorting keys used to build the trees in the index.

the attributes in the SK list. Selecting SKs generally requires domain knowledge. However, we propose an automatic key selection algorithm in Chapter 8.

- **Sorting key value:** A sorting key value (SKV) of a record in **R** is the value of the attributes used as SK for that record. For example, assume we have a record with the following attribute values r = {Firstname = 'john', Surname = 'smith', City = 'sydney', Postcode = '2000'}, and assume that a concatenation of the 'Firstname' and 'Surname' attributes is used as a SK, then the value 'johnsmith' will be the SKV generated for r.

The problem of real-time ER is defined as: for each query record q_j in a query stream $\mathbf{Q}$, find all the records in $\mathbf{R}$ that belong to the same entity as q_j, denoted as the set $\mathbf{M}_{q_j}$, in sub-second time, where $\mathbf{M}_{q_j} = \{r_i \mid r_i.eid = q_j.eid, r_i \in \mathbf{R}\}, \mathbf{M}_{q_j} \subseteq \mathbf{R}, q_j \in \mathbf{Q}$. Our aim is to improve the quality of real-time ER. Table 7.1 summarizes the main notation that we use in this chapter.

7.3 Overview of the Approach

The forest-based dynamic sorted neighborhood index (F-DySNI) is an index that facilitates real-time ER and that can be used with dynamic data sets. The index (illustrated in Figure 7.1) consists of multiple distinct trees where each tree is built using a unique sorting key (SK) to help improve the quality of results in cases where errors and variations occur at the beginning of attribute values. For example, assume that we have the following two records in a data set, r_1 = {'christine', 'jones', 'sydney', '2000'}, and r_2 = {'khristine', 'jones', 'sydney', '2010'}. If we construct the F-DySNI index by building only one tree using the 'Firstname' as a SK, these two records will

not be inserted into the same tree node (or neighboring tree nodes) which means that they will not be classified as matches. However, to improve this result, we can build another two trees in the index using other SKs (e.g. the 'Surname' and 'City' attributes). In this case, although records are not inserted into the same tree node in the first tree (that uses the 'Firstname' attribute as a SK), they will be inserted into the same tree node in the other two trees (that uses the 'Surname' and 'City' attributes as SKs). This means that these two records will be distinguished as candidate records although they have an error at the beginning of their first name values. The following sub-sections describe in more detail the data structure of the F-DySNI, and how it is built and queried.

7.3.1 Index Data Structure

As shown in Figure 7.1, the F-DySNI consists of multiple distinct trees. A single tree in the F-DySNI uses the same braided tree data structure (**BRT**) that is used in the dynamic sorted neighboring index (DySNI) which is proposed in Chapter 6). A **BRT** is a height balanced AVL braided search tree that combines the propriety of an AVL tree and a double-linked lists where every tree node is linked to its predecessor and successor nodes based on the alphabetical sorting of the key values of the tree nodes (refer to Chapter 6 for more details on the structure of a **BRT**).

The difference between the structure of the DySNI and the F-DySNI is that the later consists of multiple tree data structure for the purpose of improving the quality of matching query results by using multiple unique SKs to build the different trees in the index. The records in a data set are inserted into each tree in the index using the SK that is used to build that tree. We define the data structure that we use to construct the F-DySNI as:

[Definition 7.1] Multiple Braided Tree (M-BRT): is a multiple tree index that consists of t distinct **BRT** trees (defined in Chapter 6), where $t > 1$. Each tree in the index uses a unique SK to sort records from data set **R** into its tree nodes. Therefore, a record $r_i \in \mathbf{R}$ will be inserted into each tree in the index data structure.

The F-DySNI has an initial *build phase* where a certain number (possibly none) of records from an existing data set **R** are inserted into the **M-BRT** (i.e. a record $r_i \in \mathbf{R}$ is inserted into each tree in the **M-BRT**). Then, the built index is used to generate candidate records to resolve query records during the *query phase*. The F-DySNI is dynamic since query records are added into the **M-BRT** as they arrive. The build and the query phases are described in more details in the following sub-sections.

7.3.2 Building the Index

Building a single tree in the F-DySNI is similar to the build phase of the DySNI (described in Chapter 6). During the build phase of F-DySNI, illustrated in Algorithm 7.1, we start (in line 1) by creating an empty **M-BRT** data structure that con-

Algorithm 7.1: F-DySNI – *build*(**R**, t, SKL)

Input:
- Data set: **R**
- Number of trees: t
- List of sorting keys: SKL = [SK_1, SK_2, ..., SK_t]

Output:
- A **M-BRT** data structure that contains t trees: $\mathbf{B}_b = \{ b_1, b_2, \ldots, b_t \}$, where b_j is a single tree in $\mathbf{B}_b$

```
1:   B_b = createMBRT(t)                  // Create an empty M-BRT with t empty trees
2:   for r_i ∈ R do:
3:     for SK_j ∈ SKL do:
4:       skv := generateKey(SK_j)         // Generate a sorting key value for r_i
5:       N_i := B_bj.findTreeNode(skv)    // Search for skv in B_bj
6:       if N_i == NULL then:
7:         N_i := B_bj.createNode(skv, r_i.id)  // Create a new node N_i with the key skv
8:         B_bj.insert(N_i)               // Insert the created node N_i into B_bj
9:       else:
10:        Append r_i.id to N_i.I
11:  Return B_b                           // Return the generated M-BRT which has t trees
                                          // and which is filled with all r_i ∈ R
```

tains t trees. Then, in lines 2 to 10, we insert each record from data set **R** into the multiple t trees using the list of the unique SKs (SKL) that are used to build the different trees in the index. To insert a record r_i into a single tree of the index we first generate a SKV for r_i (line 4) using the SK which is used to build that tree. Then, we search for the generated SKV in the tree that we want to insert r_i into (line 5). If the SKV is not found in the tree (line 6), we create a new node with the record's identifier added to its identifier's list (line 7) and insert this new node into the tree data structure (line 8). On the other hand, if the generated SKV is found in the tree (line 9) we only append the record's identifier into the node that have the same SKV (line 10). In the same way r_i is inserted into the other trees in the index using the rest of the SKs from the SKL. This process continues until all records from **R** are inserted into the t trees of the index.

An example of using multiple SKs to build the F-DyNI data structure for a data set that has the following attributes {'Firstname', 'Surname', 'Suburb', 'Postcode'} is to use the 'Firstname' attribute as a SK to build the first tree in the index, the 'Surname' attribute to build a second tree, the 'Postcode' attribute to build a third tree, and so on. After all trees in the F-DySNI are built, it will be ready for resolving queries as described next.

7.3.3 Querying the Index

In this phase, shown in Algorithm 7.2, a query record q_j is first inserted into the index data structure **M-BRT** and then it is matched in real-time against all trees that were constructed in the build phase. As shown in the algorithm, for each SK in the sorting key list that is used to build the F-DySNI we do the following (lines 3 to 11): we first generate the SKV for the query record q_j in line 4. Then, in lines 5

Algorithm 7.2: F-DySNI – *query*($\mathbf{B}_b$, q_j, **S**, SKL, w, **D**)

Input:
- The built index data structure: $\mathbf{B}_b = \{ b_1, b_2, \ldots, b_t\}$
- Query record: q_j
- Similarity functions: **S**
- Sorting key list: SKL = [SK_1, SK_2, ..., SK_t]
- Window size: $w \geq 1$
- Data set table with complete records: **D**

Output:
- Ranked list of matches: **M**

1:	$\mathbf{C} := \{\}$	// Initialize the set of candidate records to be empty
2:	$\mathbf{D}[q.id] := q_j$	// Insert the query record q_j into **D**
3:	**for** $SK_j \in$ SKL **do:**	
4:	$skv := generateKey(SK, q_j)$	// Generate a sorting key value for the query q_j
5:	$N_{q_j} := \mathbf{B}_{b_j}.findTreeNode(skv)$	// Search for skv in $\mathbf{B}_{b_j}$
6:	**if** N_{q_j} == NULL **then:**	
7:	$N_{q_j} := \mathbf{B}_{b_j}.createNode(skv, q_j.id)$	// Create a new node N_{q_j} with the key skv
8:	**else:**	
9:	Append $q_j.id$ to $N_{q_j}.I$	
10:	c:= $\mathbf{B}_{b_j}.generateWin(N_{q_j}, \mathbf{S}, w)$	// Generate the candidate records from neighboring // nodes using a window size w
11:	**C**:= **C** + c	// Add the candidate record generated from this // tree to **C**
12:	**M** :=$\mathbf{B}_{b_j}.compareRecords(\mathbf{C}, \mathbf{S}, \mathbf{D}, q_j)$	// Compare query record q_j with all candidate // records in **C** and return found matches
13:	Sort **M** according to similarities	
14:	Return **M**	

to 9, we insert the query record q_j into the tree (as described in the build phase in Section 7.3.2). Next, in line 10, we create a window of nodes that are neighbors to the query record q_j (using any of the adaptive of fixed window approaches proposed in Chapter 6), and in line 11 the candidate records generated from this tree are added to a set of the overall candidate records (i.e. this set contains the candidate records that are generated from all trees in the index). In line 12, the query record is compared with all generated candidate records and the found matching records are added to **M** which is then sorted in line 13.

The process of retrieving candidate records from a single tree in the F-DySNI is similar to what was described in Chapter 6, which is achieved by using any of the different proposed window approaches (fixed or adaptive). However, the candidate records in the F-DySNI approach are retrieved from every tree 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_j is compared with all unique records in **C** in detail using similarity comparison functions [33] 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 in DySNI where only one tree is used.

7.4 Estimating the Number of Comparisons Required for the Proposed Approach

Estimating the number of candidate records that are required to resolve a query record using a single tree is discussed in Section 6.6 assuming both a uniform and a Zipfian [173] distributions of the frequency of attribute values in a data set. These estimates (for both uniform and Zipfian distributions) as given in Equations 6.2, 6.3 and 6.4 (in Chapter 6) also apply to F-DySNI since the difference between DySNI and F-DySNI is that the latter 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 number of candidate records for F-DySNI assuming a uniform distribution can be calculated as:

$$|\mathbf{C}| = \sum_{i=1}^{i=t} \frac{n(2w+1)}{k_{t_i}} \tag{7.1}$$

where n is the number of records in data set $\mathbf{R}$, w is the size of the window that is used to generate the candidate records, k_{t_i} is the number of nodes in a tree, and t is the number trees in the F-DySNI data structure. As for the Zipfian distribution, the maximum number of candidate records using multiple trees can be calculated as:

$$|\mathbf{C}| = \sum_{i=1}^{i=t} S_{w_{t_i}} \tag{7.2}$$

where t is the number of trees in the F-DySNI and $S_{w_{t_i}}$ is the number of candidate records returned from a single tree, and how its calculated is described in Chapter 6. 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 the all 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 [41].

7.5 Experimental Evaluation

In this section we describe the experiments conducted to evaluate the proposed F-DySNI approach. The data sets used in these experiments are summarized in Table 7.2. More detail about the data sets, the evaluation measures, and the implementation environment are found in Chapter 4. To evaluate our approach we conducted several sets of experiments described below.

Table 7.2: Data sets summary (described in more details in Chapter 4).

Data set	Type	Number of records
NC	Real	7,997,234
OZ-(1,2,3,4)	Real (modified)	345,876
CCA-1	Real	689,928
CCA-3	Real	2,064,823
CCA-10	Real	6,900,163
CCA-30	Real	20,708,303

7.5.1 Effects of Using Multiple Trees and Different SK Combinations on Quality and Efficiency

In this set of experiments we evaluate the effect of using multiple trees and the effect of using different SK combinations on recall and average query time. We run the experiments several times using different numbers of trees (with one, two, and three trees in the index). The SKs that we use to build the different trees in the index include all possible single attributes (for example 'Firstname', 'Surname', 'City', and so on) and all possible combinations of concatenated pairs of attributes (for example 'Firstname+City', 'Firstname+Surname', and so on). To generate the set of candidate records we use the similarity-based adaptive window approach (described in Chapter 6) with a similarity threshold between 0.5 and 1.0. The four OZ-(1,2,3,4) data sets (with different corruption ratios) are used to conduct the experiments (details about the data sets are found in Chapter 4). A single record in any of these data sets has an average of 3.5 corrupted duplicates. The results for this set of experiments are illustrated in Figure 7.2.

The results show that building more trees in the index increases recall values at the cost of a slight increase in the average query time for all OZ data sets. Recall values, when using three trees in the index compared to using a single tree, increase between 42% and 136% for SKs that are generated using single attribute values, and between 48% and 120% for SKs that are generated using concatenated attribute values for the different OZ data sets. It is also noted that single attribute values (such as 'Firstname') achieve better recall values than concatenated attribute pairs (such as 'Firstname+Postcode'). This is because using single attributes as SKs results in having nodes with large sizes which leads to having large number of candidate records (which in turn increases recall values).

We can also see that although SKs generated from single attribute values give better recall results they require longer time to resolve queries (nevertheless still achieve the sub-second query time required for real-time ER). On the other hand, SKs that are generated from a concatenation of two attribute values reduce the average query time significantly and still achieve high recall values. 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 achieving high matching quality.

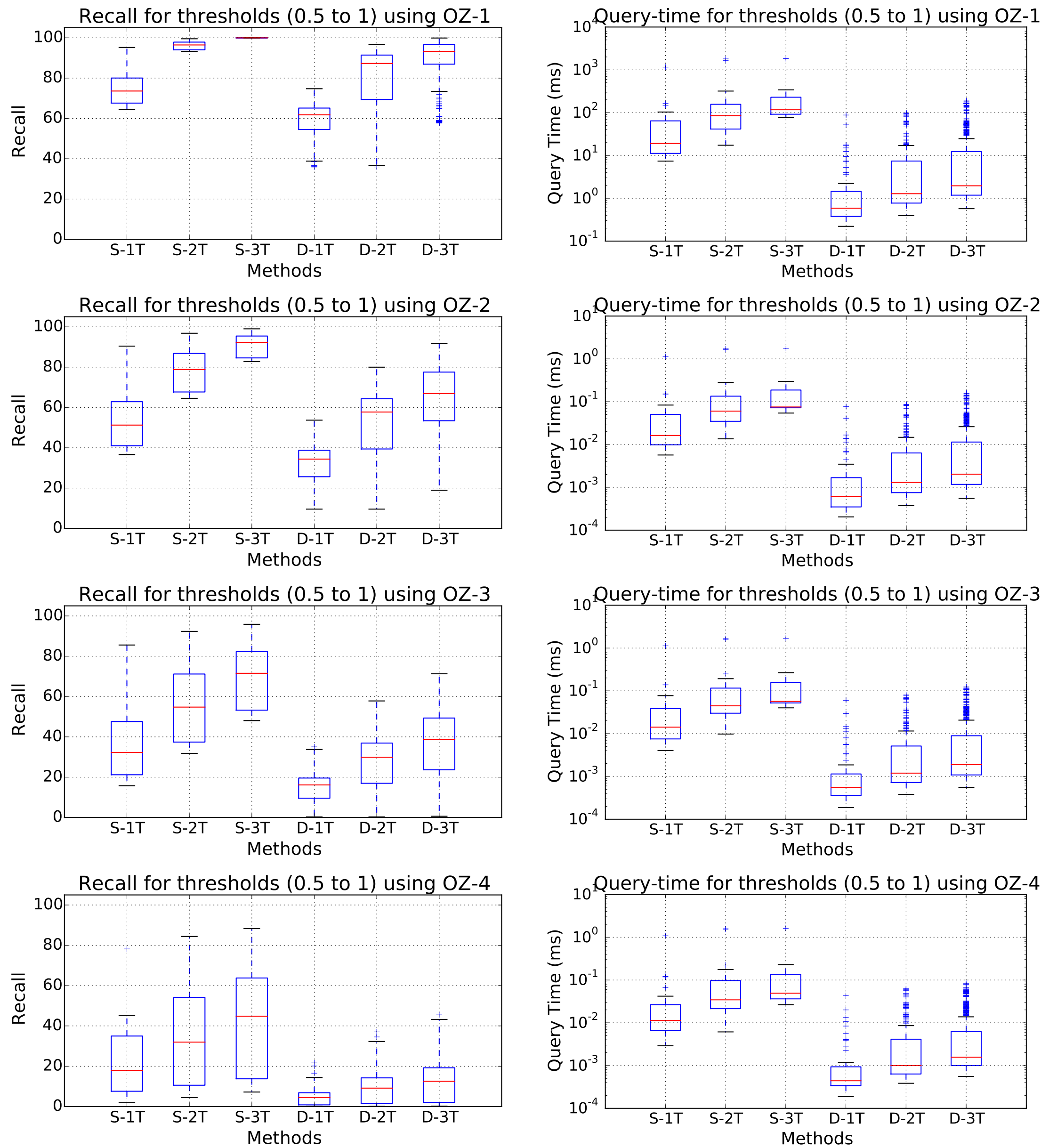


Figure 7.2: Recall and query times achieved by the F-DySNI using different numbers of trees on the OZ-(1,2,3,4) data sets. The data sets have various corruption ratios (refer to Chapter 4 for more details on the data sets). The similarity-based window approach described in Section 6.4 is used to generate the candidate records based on the similarity between the SKs of neighboring nodes using similarity thresholds between 0.5 and 1.0.

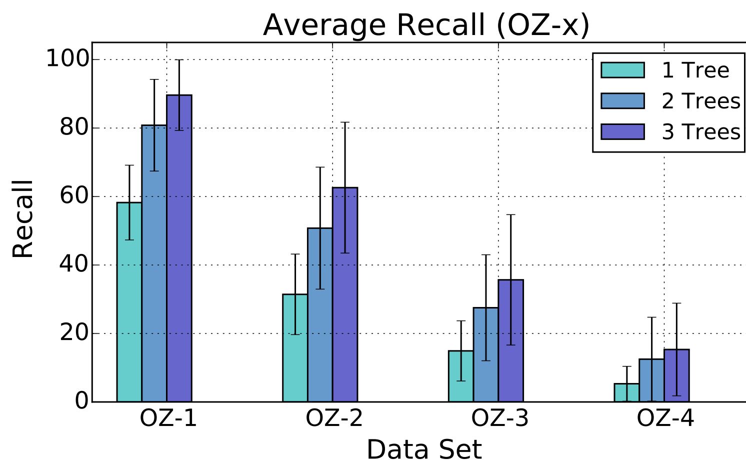


Figure 7.3: Recall for the different OZ data sets with different corruption ratio using the F-DySNI approach. The results include recall values for all possible SK combinations that are generated using a concatenation of two attribute values (which are shown, in Section 7.5.1, to be suitable for use with real-time ER).

7.5.2 Effects of Having Corrupted Attribute Values in a Query Record on the Quality of the Matching Process

The aim of this set of experiments is to investigate the effect of having different number of corrupted (modified) attribute values in records on the quality of the matching process. As one would expect, the results presented in Figure 7.3 show that recall values decrease with increasing the number of corrupted attributes in the query record. For the OZ-1 data set (with one corrupted attribute value), the F-DySNI achieves (using three trees) an average recall value of around 90%. However, for the other corrupted data sets the recall values achieved are around: 62% for OZ-2, 38% for OZ-3, and 18% for OZ-4 (where all attributes values are corrupted).

It is important to note that although in general recall values are expected to be lower for data sets with more corrupted attribute values, still, using multiple trees achieved a significant increase in recall values. This is observed clearly in Figure 7.3 where the achieved recall values have increased by 48% for OZ-1, 97% for OZ-2, 116% for OZ-3, and 120% for OZ-4 when using three trees compared to recall values achieved using only a single tree.

7.5.3 Scalability of the Proposed Solution

In this set of experiments we evaluate whether the proposed F-DySNI with a different number of trees scales to large data sets while facilitating real-time ER. We measure the average time required to insert a single record into the index data structure, and the average time required to resolve a single query record across the growing size of the index data structure. These experiments are conducted on the full NC data set which contains around 8 million records (described in Chapter 4). The similarity-

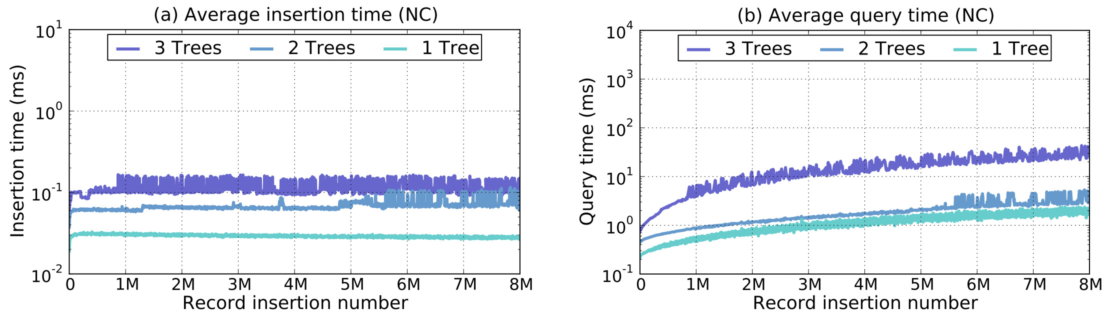


Figure 7.4: Plot (a) shows the average time required for inserting a single record into the index using different number of trees. Plot (b) illustrates the average time required for querying the growing index using different number of trees. The full NC data set (described in Chapter 4) was used to build the indexes (M = Million).

based adaptive window approach (described in Chapter 6) is used to generate the set of candidate records (since it achieved better quality results compared to the other window approaches proposed in Chapter 6).

We run the experiments using different numbers of trees with different SKs based on two concatenated attributes. First, we build an index using only one tree in the index data structure and we use a concatenation of the 'Surname+Firsname' attributes as a SK. Then we build another index with two trees in the index; the same SK from the previous index is used to build the first tree and for the second tree we use a concatenation of the 'Firstname+Surname' attributes as a SK. Finally, we build a third index with three trees in its data structure using the same SKs from the previous index to build the first two trees, and for its third tree we use a concatenation of the 'Firstname+City' attributes as a SK. The measured insertion and query times for the three built indexes are plotted in Figure 7.4. We selected these SKs based on the results achieved in the experiments described in Section 7.5.1 where we show that SKs generated from the concatenation of two attribute values are more suitable for real-time ER than SKs generated from single attribute values.

As can be seen from Figure 7.4, the results show that in plot (a) the average insertion times using the various numbers of trees is not affected by the growing size of the index data structure, while in plot (b), the results show 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) when building three trees in the index.

To investigate how the query time is affected with using larger data sets, we conduct another set of scalability experiments on the different subsets of the CCA data set (described in Chapter 4). The size of these subsets ranges between 689,928 records for the CCA-1 data set and 20,708,303 records for the CCA-30 data set. We build three indexes (with one, two, and three trees) using each of the CCA subsets, then we measure the average time required to query the built indexes. The results

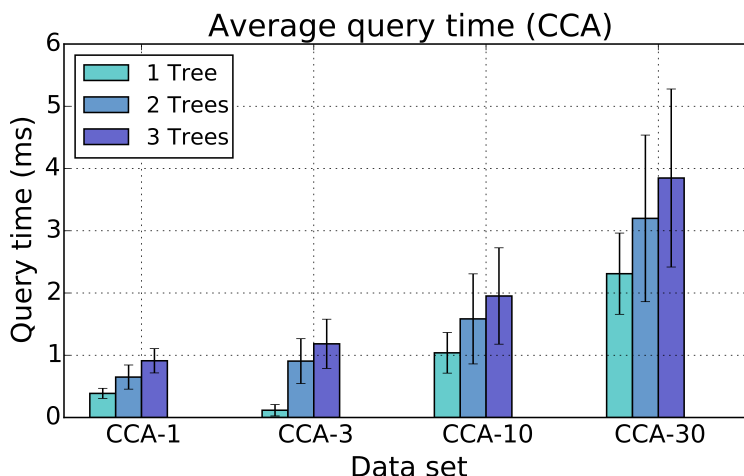


Figure 7.5: Average query time for the different CCA subsets.

Table 7.3: Required memory (in MB) for the F-DySNI approach for the different data sets. The SK 'Firstname+Surname' is used to build the first tree, 'Surname+Firstname' is used to build the second tree, and 'Surname+Postcode' is used to build the second tree.

		Memory (MB)		
Data set	**Size**	**1 Tree**	**2 Trees**	**3 Trees**
NC	7,997,234	1,843	3,686	4,403
OZ-(1,2,3,4)	345,876	114	227	448
CCA-1	689,928	112	224	414
CCA-3	2,064,823	310	621	1,238
CCA-10	6,900,163	913	1,826	3,607
CCA-30	20,708,303	2,337	4,675	8,957

(presented in Figure 7.5) show that for the increasing size of the data set the average query time increases sub-linearly yet it is still very fast with an average query time of around 4 ms for a data set with over 20 million records.

7.5.4 Required Memory Size

Table 7.3 shows the memory requirements for the F-DySNI approach using one, two, and three trees in the index data structure. The keys used for building the different trees are 'Firstname+Surname' for the first tree, 'Surname+Firstname' for the second tree, and 'Surname+Postcode' for the third tree. These keys are selected based on our results in Section 7.5.1 which confirms that SKs generated using concatenated attribute values are more suitable for real-time ER than SKs generated using single attribute values.

As shown in Table 7.3, the memory requirements increases with the increase in the number of the trees in the index (for all data sets). Moreover, it increases with the increase in the number of records in the data set. However, for the largest data set (CCA-30) which has more than 20 million records, the required memory for building the index (with 3 trees) is 8,957 MB. This required memory is around 6.7% of the amount of main memory available on the experimental machine (which is 128 GB), indicating that our implementation would be viable even for much larger data sets.

7.6 Summary

We proposed a dynamic forest-based index that can be used for real-time ER. The index uses multiple trees with different sorting keys to reduce the effect of errors and variations at the beginning of attribute values on the quality of matching results. Our evaluation shows that F-DySNI is scalable with respect to the size of large data sets and that using multiple trees has a noticeable improvement on matching quality with a small increase in query time when using sorting keys based on several concatenated attribute values. The results also showed that sorting keys which are based on a concatenation of more than one attribute value are more suitable for real-time ER as they require shorter query times than sorting keys generated from single attribute values while still achieving high matching accuracy.

As a follow-up to what we have presented in this chapter, we propose (in the next chapter), an automatic key selection algorithm which selects sorting keys that are suitable for real-time entity resolution without human intervention. Other research directions that can follow our work in this chapter are to investigate the effect of using secondary SKs (i.e. attribute values that are different than the SK used to build the index tree) that sort records within tree nodes on the efficiency and the effectivity of the DySNI, and to investigate how to parallelize the multiple-tree index to improve its efficiency. Moreover, to investigate how our proposed approach can be extended to work as a disk-based indexing solution to be used with larger data sets that cannot fit into main memory. Further comparative evaluation can be conducted to compare matching records produced by our proposed indexing technique with different indexing techniques using the same data sets for individual query records.

Chapter 8

Unsupervised Blocking Key Selection for Real-Time Entity Resolution

As described in Chapter 3, selecting blocking/sorting keys is crucial for the effectiveness and efficiency of the real-time entity resolution process. Indexing techniques require domain knowledge for optimal key selection. However, to make the real-time entity resolution process less dependent on human domain knowledge, automatic selection of optimal blocking/sorting keys is required. In this chapter we propose an unsupervised learning technique that automatically selects optimal blocking/sorting keys for building indexes that can be used with real-time ER. We summarize the notation that we use in this chapter is Section 8.2, then in Section 8.3 we provide a detailed description of the proposed approach. In Section 8.4 we describe the experimental evaluation that we use to evaluate the proposed approach. Finally, we summarize our findings in Section 8.5.

8.1 Introduction

Indexing is a vital step in the entity resolution (ER) process especially for large data sets as it reduces the number of candidate records to be compared in detail to find matching records. This can be achieved by two main approaches (as detailed in Chapter 3). The first is to partition records in a data set into several blocks according to a blocking key criterion, where only records that are inserted into the same block are compared with each other [62]. The second approach is to sort the records in a data set according to a sorting key criterion that brings similar records close to each other, so that only records that are close to each other will be compared [79].

A good indexing technique should group similar records into one block or close to each other in the index [34]. This depends mainly on the blocking/sorting key used to partition/sort the records in a data set. An optimal key needs to find all true matching records, while keeping to a minimum the number of true non-matching records. However, an optimal key for one domain will likely not work for another domain [34]. Moreover, an optimal key for traditional static ER with batch processing

Table 8.1: Summary of the main notations used in this chapter

Notation	Description
$\mathbf{R}$	A data set of records about known entities
$\mathbf{A}$	A set of attributes $\{a_1, a_2, \ldots, a_{\lvert\mathbf{A}\rvert}\}$ for each $r_i \in \mathbf{R}$
$\mathbf{Q}$	A stream of query records
$\mathbf{M}_{q_j}$	A set of all records in $\mathbf{R}$ that belongs to the same entity of a query q_j
r_i	A record in $\mathbf{R}$
$r_i.id$	Unique record identifier for r_i
$r_i.eid$	Entity identifier for r_i
q_j	A query in $\mathbf{Q}$
$q_j.id$	Unique record identifier for q_j
$q_j.eid$	Entity identifier for q_j
$k_{h,l}$	A blocking key that is consisting of a pair $\langle a_h, f_l \rangle$, which contains an attribute $a_h \in \mathbf{A}$ and a blocking function $f_l \in \mathbf{F}$
$\mathbf{F}$	A set of candidate blocking functions
$\mathbf{K}$	A set of candidate blocking keys
b	A block of records that have the same blocking key value
$\mathbf{B}_{h,l}$	A set of all blocks generated using a blocking key $k_{h,l}$ on all records in $\mathbf{R}$
$TFIDF(.,.)$	A weighting scheme used to calculate the similarity between two record pairs (r_i, r_x)
$\mathbf{O}$	A set of optimal blocking keys, where $\mathbf{O} \subset \mathbf{K}$
$\mathbf{R}_U$	Unlabeled data set of record pairs
$\mathbf{R}_N$	A negative training data set
$\mathbf{R}_P$	A positive training data set
$\mathbf{V}_P$	A blocking key vector generated from applying all keys in $\mathbf{K}$ on $\mathbf{R}_P$
$\mathbf{V}_N$	A blocking key vector generated from applying all keys in $\mathbf{K}$ on $\mathbf{R}_N$
ut, lt	An upper and lower thresholds where $0.0 < lt < ut < 0.1$
c_k	The coverage of a blocking key
v_k	The variance of a blocking key
s_b	The size of a block (number of records within a block b)
ck_k	The overall score of a key

algorithms might not be suitable for real-time ER which requires small block sizes to achieve fast query matching. Selecting an optimal key needs expert knowledge of the nature of the data and the requirements of the domain.

To the best of our knowledge, there is no existing automatic blocking key selection technique for indexing that considers real-time ER. Therefore, there is a need for novel techniques that learn optimal keys for different real-time ER domains without the need for manual intervention. In this chapter, we propose a general learning technique that automatically selects optimal keys for building indexes to be used with real-time ER in order to find matches in a data set effectively and efficiently. Our approach can be used with different indexing techniques. We demonstrate how our automatic key selection approach can be used with the F-DySNI indexing technique proposed in Chapter 7 since it a dynamic indexing technique that is designed to work with real-time ER.

8.2 Terminology and Notation

The following is a summary of the terminology and the notation that we use in this chapter:

- **Data set:** We assume that data set $\mathbf{R} = \{r_1, r_2, \ldots, r_{\lvert\mathbf{R}\rvert}\}$ contains records of known entities. Each $r_i \in \mathbf{R}$ has a unique record identifier $r_i.id$ and an entity

identifier $r_i.eid$. Records in $\mathbf{R}$ are described by a set of attributes, denoted as $\mathbf{A} = \{a_1, a_2, \ldots, a_{|\mathbf{A}|}\}$. An attribute of r_i is denoted as $r_i.a_h$ where $a_h \in \mathbf{A}$. All records in $\mathbf{R}$ are assumed to have the same attribute structure.

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

- **Blocking Key:** A blocking key (BK), denoted as $k_{h,l} = \langle a_h, f_l \rangle$, is a pair consisting of an attribute $a_h \in \mathbf{A}$ and a blocking function $f_l \in \mathbf{F}$, with $\mathbf{F}$ being the set of candidate blocking functions. We assume the functions in $\mathbf{F}$ are manually selected by domain and ER experts. Examples of such functions include exact value (isExact), same first character (sameFirst1Char), or same last three characters (sameFirst3Char), and so on.

- **Blocking key value:** The blocking function f_l is applied on attribute a_h, and the resulting value for a record r_i is called a blocking key value (BKV) and denoted as $k_{h,l}(r_i) = f_l(r_i, a_h)$. For example, for $k_{h,l} = \langle a_h, f_l \rangle$, assume that a_h='Firstname', f_l = first3Char, and $r_i.a_h$ = 'peter', then the BKV that is generated from evaluation function $f_l(r_i, a_h)$, will be 'pet'.

- **Candidate blocking keys:** The set of all candidate BKs selected by an expert, denoted as $\mathbf{K}$. $\mathbf{K}$ is generated by applying all $f_l \in \mathbf{F}$ on all $a_h \in \mathbf{A}$. For example, assume we have two attributes $\mathbf{A}$ = {'Firstname', 'Surname'} in our data set, and assume we have two blocking functions $\mathbf{F}$ = {sameFirst2Char, sameFirst3Char}. Our list of candidate BKs will be:
  $\mathbf{K} = \{\langle$Firstname, sameFirst2Char$\rangle$, $\langle$Firstname, sameFirst3Char$\rangle$,
  $\langle$Surname, sameFirst2Char$\rangle$, $\langle$Surname, sameFirst2Char$\rangle\}$.

- **Block:** A block $b \in \mathbf{B}_{h,l}$ is a set of records $\mathbf{R}_b \subseteq \mathbf{R}$ where all $r_i \in \mathbf{R}_b$ have the same BKV: $\mathbf{R}_b = \{r_i \in \mathbf{R} : k_{h,l}(r_i) = f_l(r_i, a_h)\}$. $\mathbf{B}_{h,l}$ is the set of all blocks generated by a BK $k_{h,l}$ on all records in $\mathbf{R}$.

The problem of real-time ER is defined as: for each query record q_j in a query stream $\mathbf{Q}$, find all the records in $\mathbf{R}$ that belong to the same entity as q_j, denoted as the set $\mathbf{M}_{q_j}$, in sub-second time, where $\mathbf{M}_{q_j} = \{r_i \mid r_i.eid = q_j.eid, r_i \in \mathbf{R}\}, \mathbf{M}_{q_j} \subseteq \mathbf{R}, q_j \in \mathbf{Q}$. The goal of this chapter is to automatically select blocking/sorting keys to build indexes that can be used to carryout the real-time ER process efficiently and effectively. Table 8.1 summarizes the notation that we use.

8.3 Overview of the Approach

Current automatic BK selections algorithms (discussed in Chapter 3) do not learn keys that are suitable for real-time ER. In real-time ER, the selected BK(s) should generate block sizes within a controllable range to make sure that the number of

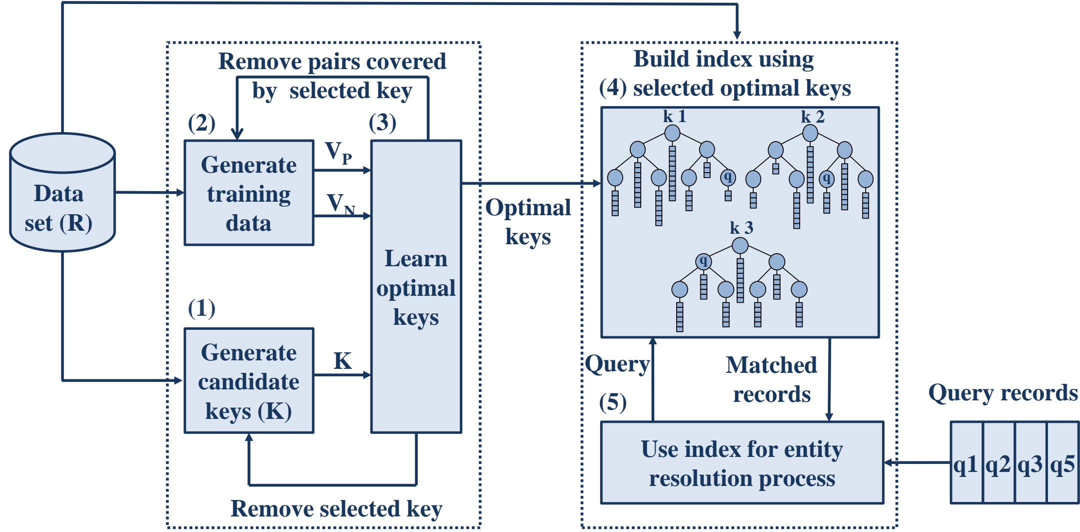


Figure 8.1: Framework of the proposed automatic blocking key selection approach.

detailed comparisons needed to match a query record (q_j) is within an allocated time. Also, keys that generate blocks of similar sizes are more suitable for real-time ER than keys that generate blocks of different sizes, as the time required to resolve different query records will be similar [42].

In our work, we aim to learn a set of optimal blocking/sorting keys, $\mathbf{O} \subset \mathbf{K}$, to perform ER and deliver high quality matching results in real-time. The selected optimal keys can also be used with any multi-pass indexing technique [128] (where the index is built several times using different blocking/sorting keys). Following [89], our approach does not require existing training data sets to learn these optimal keys.

The overall framework of the proposed approach contains the following steps, as illustrated in Figure 8.1. In step (1), we generate the set of candidate BKs $\mathbf{K}$ based on domain knowledge. Then, we generate positive and negative training data sets ($\mathbf{R}_P$ and $\mathbf{R}_N$) in step (2). Both $\mathbf{R}_P$ and $\mathbf{R}_N$ are converted into a set of BK vectors ($\mathbf{V}_P$ and $\mathbf{V}_N$) as described in Section 8.3.2 to be used in the learning process. In step (3), we employ the proposed learning algorithm using the generated BK vectors $\mathbf{V}_P$ and $\mathbf{V}_N$ to select a set of optimal BKs $\mathbf{O} \subset \mathbf{K}$. In step (4), we use the optimal keys $\mathbf{O}$ selected in the previous step to index (block) all records from data set $\mathbf{R}$. Any real-time indexing technique can be used for this step [128]. Finally, in step (5), we use the index that was built in step (4) for matching arriving query records with records within the index in real-time.

8.3.1 Generating Candidate Keys

The set of candidate keys $\mathbf{K}$ contains all keys which we select our optimal keys from. The candidate BKs can differ based on the domain, the used indexing technique, and the data sets to be matched. Because we are evaluating our key selection algorithm using a sorted neighbourhood indexing technique [130] (as will be

Algorithm 8.1: *generateTrainingDS*($\mathbf{R}, \mathbf{A}, ut, lt$)

Input:
- Data set: $\mathbf{R}$
- Set of attributes: $\mathbf{A}$
- Upper threshold: ut
- Lower threshold: lt

Output:
- Set of positive pairs: $\mathbf{R}_P$
- Set of negative pairs: $\mathbf{R}_N$

```
1:  R_U := { }, R_P := { }, R_N := { }    // Initialize the sets of unlabeled, positive,
                                          // and negative record pairs to empty
2:  B := { }                              // Initialize the set of generated blocks to empty
3:  for r_i ∈ R do:
4:     for a_h ∈ A do:
5:        tokens := generateTokens(a_h)   // Tokenize attribute values
6:        blks := generateBlocks(tokens)  // Each token becomes a block
7:        insertToBlocks(r_i,tokens)      // Insert r_i into all blocks that are
                                          //  the same as its generated tokens
8:        B.add(blks)                     // Add generated blocks into B
9:  stats = generateIDFStats(R, B)        // Generate the TF-IDF statistics required to
                                          // calculate the TF-IDF similarity in line 14
10: for b ∈ B do:
11:    P := generatePairs(b)              // Generate all possible record pairs from
                                          // the records in b
12:    R_U.add(P)                         // Add P to the Unlabeled data set R_U
13: for (r_x, r_y) ∈ R_U do:
14:    s := TFIDF(r_x, r_y)               // Calculate TF-IDF similarity between (r_x, r_y)
15:    if s ≥ ut then:
16:       R_P.add(r_x, r_y)               // Add record pair (r_x, r_y) to R_P
17:    else if s ≤ lt then:
18:       R_N.add(r_x, r_y)               // Add record pair (r_x, r_y) to R_N
19: Return R_P,R_N                        // Return generated training data sets
```

discussed in Section 8.4), we generate a list of candidate BKs that capture the beginning of attribute values. This is achieved by using the following blocking functions $\mathbf{F} = \{\mathrm{isExact},\ \mathrm{sameFirst1Char},\ \mathrm{sameFirst2Char},\ \mathrm{sameFirst3Char},\ \mathrm{sameFirst4Char},\ \mathrm{sameFirst5Char},\ \mathrm{concatenatedIsExact}\}$ on all attributes in $\mathbf{A}$. For data sets with long attribute values we use the first 1 to 5 tokens (i.e. individual characters, separate words, terms, or abbreviations) in the attribute value instead of characters as blocking functions (since attribute values contain multiple words). The generated set of BKs $\mathbf{K}$ is given to the proposed learning algorithm (described in Section 8.3.3) to be used in the learning process.

8.3.2 Generating Training Data Sets

In most practical applications of ER no training data sets (gold standard data) are available. As an alternative, training data sets can be generated using classification functions as in [65, 89]. In this step we use Kejriwal's [89] approach to generate weakly labeled training data sets using a TF-IDF weighting scheme to calculate the similarity between record pairs $(r_x, r_y) \in \mathbf{R}$ as described in Algorithm 8.1.

We start the algorithm (lines 1 and 2) by initializing the sets of unlabeled record pairs $\mathbf{R}_U$, positive record pairs $\mathbf{R}_P$ (pairs that are classified as positive matches), and

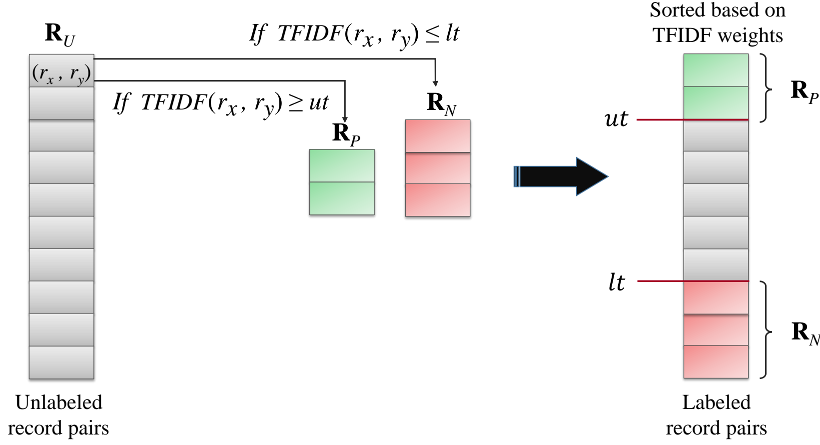


Figure 8.2: Generating training data sets for the proposed automatic blocking key selection approach.

negative record pairs $\mathbf{R}_N$ (pairs that are classified to non-matches) to be empty. We also initialize the set $\mathbf{B}$ of all generated blocks to be empty. Then, in lines 3 to 7, we convert attribute values for all $r_i \in \mathbf{R}$ into tokens, and insert r_i into all blocks that corresponds to its generated tokens. We consider a token to be limited to its attribute; this means that the same token generated from another attribute will be considered as a different block.

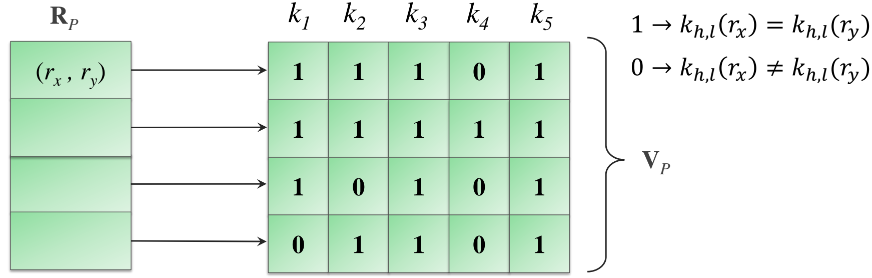
It is important to mention that for data sets with short attribute values which contain only one word we convert attribute values into q-grams (sub-strings of length q) instead of words tokens, and the generated q-grams becomes the blocks that are used to partition records in $\mathbf{R}$. Then, in line 8, the new blocks (blks) generated for an attribute value $a_h \in \mathbf{A}$ are added to the set of all generated blocks $\mathbf{B}$. After tokanization of attribute values, and after generating all possible blocks $\mathbf{B}$, in line 9, we calculate the TF-IDF weights described in Equation 8.3 that is required to calculate the TF-IDF similarity between record pairs. Next, in lines 10 to 12, we iterate through all blocks in $\mathbf{B}$ and generate a set of all possible record pairs from the records in each block. The generated record pairs are added into $\mathbf{R}_U$ (a set of the unlabeled record pairs).

The following step is to classify the record pairs in $\mathbf{R}_U$ into positive and negative pairs to generate the training data sets $\mathbf{R}_P$ and $\mathbf{R}_N$. This is achieved by calculating the TF-IDF similarity value (in line 14), denoted as $TFIDF(.,.)$, for all $(r_x, r_y) \in \mathbf{R}_U$ as follows [19, 89]:

$$TFIDF(r_x, r_y) = \sum_{tkn \in r_x \cap r_y} wt(r_x, tkn) \cdot wt(r_y, tkn) \tag{8.1}$$

where wt is:

$$wt(r, tkn) = \frac{wt'(r, tkn)}{\sqrt{\sum_{tkn \in r} wt'(r, tkn)^2}} \tag{8.2}$$



Example:
For a blocking key $k_{h,l} = \langle a_h, f_l \rangle$, assume that a_h is the 'FirstName' attribute, and f_l is the *sameFirst2Char*( ) function. $r_x.\ a_h$ = 'peter', $r_y.\ a_h$ = 'pedro'. Then:
$k_{h,l}(r_x) = k_{h,l}(r_y)$ since the first two characters of 'peter' and 'pedro' are the same

Figure 8.3: An example on generating the blocking key vectors that are used in the process of learning optimal blocking keys (described in Section 8.3.3) of the proposed automatic blocking key selection approach.

and wt' is:

$$wt'(r, tkn) = log(tf_{r,tkn} + 1) \cdot log(\frac{|\mathbf{R}|}{df_{tkn}} + 1) \tag{8.3}$$

where $r_x \cap r_y$ is the set of common tokens between r_x and r_y, $wt(r, tkn)$ is the normalized TF-IDF weight for a token tkn in a record r, $tf_{r,tkn}$ is the term frequency of tkn in r, $|\mathbf{R}|$ is the number of records in $\mathbf{R}$, and df_{tkn} is the number of records in $\mathbf{R}$ that contain the term tkn.

Then, in lines 15 to 18, an upper and a lower thresholds $0.0 < lt < ut < 1.0$ are used to generate the training data sets. We generate a positive training set $\mathbf{R}_P \subset \mathbf{R}$ where the similarity between record pairs is greater than or equal to the upper threshold ut: $\mathbf{R}_P = \{r_x, r_y \in \mathbf{R} : TFIDF(r_x, r_y) \geq ut\}$; and a negative training set $\mathbf{R}_N \subset \mathbf{R}$ where the similarity between record pairs is less than or equal to the lower threshold lt: $\mathbf{R}_N = \{r_x, r_y \in \mathbf{R} : TFIDF(r_x, r_y) \leq lt\}$ with $\mathbf{R}_P \cap \mathbf{R}_N = \{\ \}$. Figure 8.2 illustrate how the training data sets are generated.

Both $\mathbf{R}_P$ and $\mathbf{R}_N$ are then converted into a set of *blocking key vectors*, $\mathbf{V}_P$ and $\mathbf{V}_N$ respectively (to be used in the learning process), by applying all keys $k_{h,l} \in \mathbf{K}$ on the record pairs in $\mathbf{R}_P$ and $\mathbf{R}_N$ [89]. Each record pair is converted into a vector of Boolean values (i.e. 0 or 1 bits), with one value for each candidate key $k_{h,l} \in \mathbf{K}$. If a record pair (r_x, r_y) in $\mathbf{R}_P$ or $\mathbf{R}_N$ for a certain candidate key $k_{h,l}$ results in having the same key value, i.e. $k_{h,l}(r_x) = k_{h,l}(r_y)$, then the corresponding element in the pair's vector is set to 1 and the pair is said to be *covered* by this key. Otherwise, the corresponding vector element is set to 0, and the pair is said to be *uncovered* by that key. An optimal key should cover as many record pairs as possible from $\mathbf{V}_P$ while keeping the number of covered record pairs in $\mathbf{V}_N$ to the minimum. Figure 8.3 illustrates an example on how $\mathbf{V}_P$ and $\mathbf{V}_N$ are generated.

The generated set of key vectors $\mathbf{V}_P$ and $\mathbf{V}_N$ are then used in our key selection algorithm to learn the optimal keys, as described in the following section. Alterna-

tively, if a truth training set is available, step (2) of our framework is not required. The rest of the steps of our framework are described in more detail in the following sections.

8.3.3 Learning Optimal Keys

The indexing step of real-time ER should group similar records together while maintaining small block sizes to be able to match query records in real-time. The BKs used in the indexing step have an impact on the quality and efficiency of query matching. To make sure that the keys we select are suitable for real-time ER we use three criteria:

- ***Key coverage:*** The coverage c of a key $k_{h,l}$ that is applied on record pairs (r_x, r_y) in data set $\mathbf{R}$ is defined as the number of record pairs that evaluate to the same key value: $c_{k_{h,l}} = |\{r_x, r_y \in \mathbf{R} : k_{h,l}(r_x) = k_{h,l}(r_y)\}|$. A key with a high coverage value in $\mathbf{V}_P$, and a low coverage value in $\mathbf{V}_N$ leads to grouping of a high number of true positive matches into the blocks generated using that key while having a small number of negative matches in the blocks. We use the blocking key vectors $\mathbf{V}_P$ and $\mathbf{V}_N$ (described in Section 8.3.2) to measure the coverage of a BK by calculating its Fisher score [89] as follows:

$$c_k = \frac{|\mathbf{V}_P|(\mu_{p,k} - \mu_k)^2 + |\mathbf{V}_N|(\mu_{n,k} - \mu_k)^2}{|\mathbf{V}_P|\sigma^2_{p,k} + |\mathbf{V}_N|\sigma^2_{n,k}} \tag{8.4}$$

  where $\mu_{p,k}$ and $\mu_{n,k}$ are the mean of all binary bits generated from evaluating the key $k_{h,l} \in \mathbf{K}$ on all pairs in $\mathbf{V}_P$ and $\mathbf{V}_N$ respectively (for example, in Figure 8.3 the binary bits generated from evaluating k_1 on all pairs in $\mathbf{V}_P$ are {1, 1, 1, 0}). $\sigma^2_{p,k}$ and $\sigma^2_{n,k}$ are the variance of corresponding binary bits of $k_{h,l}$ in $\mathbf{V}_P$ and $\mathbf{V}_N$ respectively, and μ_k is the mean of the corresponding binary bits of key $k_{h,l}$ in $\mathbf{V}_P \cup \mathbf{V}_N$. Note that a key will have a high Fisher score if it has high coverage in $\mathbf{V}_P$ and low coverage in $\mathbf{V}_N$. The aim of using this measure is to select keys that produce high quality blocks (with mostly true matches, and only few negative matches grouped within the generated blocks).

- ***Block Size:*** The size of a block b is the number of records that are inserted into that block and it is denoted as $s_b = |\mathbf{R}_b|$. In this criterion we use two measures: the maximum block size denoted as $s_{b_{(max)}} = max\{s_b : b \in \mathbf{B}_{h,l}\}$, and the average block size denoted as $s_{b_{(ave)}} = ave\{s_b : b \in \mathbf{B}_{h,l}\}$. The aim of using the size criterion is to control the number of candidate records that are required to be compared with a query record within a desired time range.

- ***Distribution of blocks:*** For the set of blocks $\mathbf{B}_{h,l}$ generated from applying a key $k_{h,l}$ on all records in data set $\mathbf{R}$, the distribution of $k_{h,l}$ is measured by calculating the variance v_k of the sizes of all blocks in $\mathbf{B}_{h,l}$ (which reflects how far the generated block sizes are spread) using:

$$v_k = \frac{\sum_{b=1}^{|\mathbf{B}_{h,l}|}(s_b - \mu_s)^2}{|\mathbf{B}_{h,l}|} \tag{8.5}$$

where s_b is the size of a block $b \in \mathbf{B}_{h,l}$, and $\mathbf{B}_{h,l}$ is the set of blocks generated using the key $k_{h,l}$. The mean of all block sizes in $\mathbf{B}_{h,l}$ is denoted as μ_s. A variance value that is equal to 0.0 means that all generated blocks have exactly the same size. For real-time ER it is better to generate blocks of similar sizes (small variance in block sizes) where the time required to match a query record is similar for different query records. Therefore, a BK is more suitable for real-time ER if its variance of the sizes of the generated blocks is close to 0.0.

Based on the above description, an optimal blocking key is defined as follows:

[Definition] 8.1 Optimal blocking key: In a list of candidate blocking keys $\mathbf{K}$, a key $k_i \in \mathbf{K}$ is an optimal key k_o if it has a minimum overall score sc_{k_i} where $k_o = min\{sc_{k_i} : k_i \in \mathbf{K}\}$. A minimum sc_{k_i} score is achieved when the following criteria are fulfilled:

$$\begin{aligned}
v_{k_{i(min)}} &= min\{v_{k_i} : k_i \in K\} \\
s_{b_{k_{i(min)}}} &= min\{s_{b_{k_i}} : k_i \in K\} \\
cp_{k_{i(max)}} &= max\{count(1) : k_{i_{(\mathbf{V}_P)}} \in \mathbf{V}_P\} \\
cn_{k_{i(min)}} &= min\{count(1) : k_{i_{(\mathbf{V}_N)}} \in \mathbf{V}_N\}
\end{aligned}$$

where v_{k_i} is the distribution of the size of the blocks generated by evaluating k_i, $s_{b_{k_i}}$ is the average block size generated by evaluating k_i, cp_{k_i} is the count of the record pairs from $\mathbf{V}_P$ that are covered by k_i, and cn_{k_i} is the count of the record pairs from $\mathbf{V}_N$ that are covered by k_i.

Our learning algorithm (see Algorithm 8.2) automatically selects the list of optimal keys $\mathbf{O}$ based on the three criteria discussed above (key coverage, generated block sizes, and distribution of block sizes) to ensure that the selected keys can be used with real-time ER to provide matching results efficiently. We start the algorithm, in line 1, by initializing the set of optimal keys to be empty. The set of valid keys ($\mathbf{K}_v$), and the scores (which holds the keys and their overall score) are also initialized to be empty (lines 3 and 4). All keys that cover less than n_m pairs in $\mathbf{V}_N$ are added to the set of valid keys $\mathbf{K}_v$ in lines 5 to 7, where n_m is the maximum allowed number of covered vectors from the negative key vectors $\mathbf{V}_N$. Then, in lines 8 to 11, for each key in the set of valid keys $\mathbf{K}_v$, if the key has a maximum block size $s_{b_{max}}$ that is greater than the maximum allowed block size s_m, it is removed from $\mathbf{K}_v$. In lines 12 to 16, for all keys left in $\mathbf{K}_v$, we calculate an overall score sc_k to determine which keys should be added to the optimal key list $\mathbf{O}$ based on the following equation:

$$sc_k = \alpha \cdot (1 - c_k) + \beta \cdot s_{b_{(ave)_k}} + (1 - \alpha - \beta) \cdot v_k \tag{8.6}$$

where c_k is the coverage of $k_{h,l}$ (as calculated in Equation 8.4), $s_{b_{(ave)_k}}$ and v_k are the average block size and the variance between the block sizes, respectively. We assume

that blocks are generated by applying the BK $k_{h,l}$ on all records in $\mathbf{R}$. The aim is to select a set of BKs that have high coverage, low average block size, and low variance between block sizes (note that keys with large maximum block size $s_{b(max)}$ were removed earlier from $\mathbf{K}_v$ in line 11). The parameters α and β are used to control the weights of the three criteria based on the domain and application area. Each weight parameter is a value between 0.0 and 1.0 where the sum of all weights is equal to 1.0. Regardless of the weight parameters used, the lower the overall score for a key is, the more this key is suited for real-time ER. An example on how to calculate the overall score sc_k is found in Example 8.1 at the end of this chapter.

After calculating the overall score sc_k for all keys in $\mathbf{K}_v$, these scores are sorted in an ascending order, since a lower overall score is better (line 17). The first key in the overall score list is then added to the optimal key list $\mathbf{O}$ (lines 18 and 19). In line 21, we remove all positive vectors that are covered by the selected optimal key from $\mathbf{V}_P$ because we want the selected keys to be complementary to each other with regard to the coverage of the vectors in $\mathbf{V}_P$ where each key covers different positive vectors from $\mathbf{V}_P$. This should achieve better matching quality when using the selected optimal keys in building the index that is used to resolve query records. We also remove the selected optimal key from $\mathbf{K}$ (in line 22) since we need to select distinct keys to be used for building the multiple trees in the index data structure. This process continues until the required number of optimal keys l_k is reached or until there are no positive vectors left in $\mathbf{V}_P$. The selected optimal keys $\mathbf{O}$ are used to build an index that is used to perform the ER process on data set $\mathbf{R}$ in real-time. An example of calculations that are required to select the first optimal key using OZ-1 data set (described in Chapter 4) is found in Table 8.5 at the end of this chapter.

8.4 Experimental Evaluation

We evaluate our learning algorithm with regard to the efficiency and the effectiveness of the selected BKs. The aim is to examine if the selected BKs are suitable for building indexes that can by used with real-time ER. We also compare our algorithm with a recent BK learning algorithm named (FDJ) proposed by Mayank Kejriwal and Daniel P. Miranker [89]. The FDJ algorithm consists of two phases. In the first phase, it generates a weakly labeled training data sets as described in Section 8.3.2. In the second phase, it uses the generated labeled training data sets to learn the optimal BKs using a Fisher discrimination criterion [55]. This Fisher score is used to rank the candidate BKs, then the algorithm selects the optimal BK (the key with the highest Fisher score). The FDJ algorithm only considers key coverage (described in Section 8.3.3) when calculating Fisher scores and selecting optimal BKs.

We use the data sets summarized in Table 8.2 to conduct our evaluation experiments. More information about these data sets can be found in Chapter 4. To conduct the evaluation we use the keys selected by our approach and the keys selected by the FDJ approach to build indexes (the keys selected by both approaches are summarized in Table 8.3). Then, these indexes are used to resolve query records in real-time. The

Algorithm 8.2: *learnOptimalBK*($\mathbf{V}_P, \mathbf{V}_N, \mathbf{K}, n_m, s_m, t$)

Input:
- Positive key vectors: $\mathbf{V}_P$
- Negative key vectors: $\mathbf{V}_N$
- Candidate blocking keys: $\mathbf{K}$
- Maximum allowed covered vectors from negative key vectors: n_m
- Maximum allowed block size: s_m
- Number of blocking keys to be selected: l_k

Output:
- Optimal blocking keys: $\mathbf{O}$

1:	$\mathbf{O} := \{\ \}$	// Initialize the set of optimal blocking keys
2:	**while** $i \leq l_k$ **do:**	// l_k is the number of optimal keys to be selected
3:	$\mathbf{K}_v := \{\ \}$	// Initialize the set of valid keys
4:	$scores := [\ ]$	// Initialize the list of keys and their scores
5:	**for** $k \in \mathbf{K}$ **do:**	
6:	**if** k covers pairs in $\mathbf{V}_N$ that are $< n_m$ **then:**	
7:	$\mathbf{K}_v$.add(k)	// Add k to the valid key list $\mathbf{K}_v$
8:	**for** $k \in \mathbf{K}_v$ **do:**	
9:	$s_{b_{(max)}} := getMaxBlockSize(k)$	// Get the maximum block size for k
10:	**if** $s_{b_{(max)}} > s_m$ **then:**	// Remove keys with large block size
11:	$\mathbf{K}_v$.remove(k)	
12:	**for** $k \in \mathbf{K}_v$ **do:**	
13:	$v_k := getVariance(k)$	// Get the variance for k
14:	$s_{b_{(ave)}} := getAveBlockSize(k)$	// Get the average block size for k
15:	$sc_k := calcScore(\mathbf{V}_P, \mathbf{V}_N, s_{b_{(ave)}}, v_k)$	// Calculate overall score for k
16:	$scores.append((k, sc_k))$	// Append a pair of this key and its score (k,sc_k) // to the list of scores
17:	scores.sort( )	// Sort the list of scores ascending based // on the score values
18:	$o := scores[0]$	// This optimal key has the lowest score value
19:	$\mathbf{O}.add(o)$	// Add this optimal key to optimal key list
20:	$\mathbf{V}_c := getCoveredPairs(k, \mathbf{V}_P)$	// Get pairs from $\mathbf{V}_P$ that are covered by k
21:	$\mathbf{V}_P$.remove($\mathbf{V}_c$)	// Remove all pairs covered by k from $\mathbf{V}_P$
22:	$\mathbf{K}$.remove(o)	// Remove the selected optimal key from $\mathbf{K}$
23:	$i := i + 1$	
24:	Return $\mathbf{O}$	// Return the optimal key list

F-DySNI (proposed in Chapter 7 and illustrated in Figure 8.4) is used for this purpose. This is because the F-DySNI is a dynamic indexing technique that can be used with real-time ER and since it performed better than the other proposed indexing techniques as shown in Chapter 7. The index consists of multiple tree data structures where each tree is built using a different key.

When a query record arrives it is inserted into all trees in the index. Then, in each tree, a window of size w is used to generate a set of candidate records from the tree node that contains the query record and the neighboring tree nodes that fall within the window w. The query is then compared using an approximating string similarity function [33] with the generated candidate records. Candidate records with similarities above a specific threshold (we used a threshold of $th = 0.75$) are considered to be matches. We use 50% of the records in each data set to build the indexes, and the remaining records are used as query records. We have $t = 3$ trees in the index, and we generate the candidate records using a window of size $w = 2$. Note that any real-time indexing technique can be used in this experimental evaluation.

Table 8.2: Data sets summary (described in more details in Chapter 4).

Data set	Type	Number of records	Number of entities
10% of OZ-1	Real-world (modified)	34,588	30,292*
Cora	Real-world	1,295	112
DBLP/ACM	Real-world	2,616/2,294	2,686

* With a maximum of one duplicate per entity

Table 8.3: Keys selected by the proposed and the FDJ approaches for the different data sets.

Proposed	FDJ
OZ-1	
⟨(Firstname, Suburb), concatenatedIsExact⟩	⟨Firstname, sameFirst2Char⟩
⟨(Surname, Suburb), concatinatedIsExact⟩	⟨Surname, sameFirst2Char⟩
⟨(Surname, Postcode), concatinatedIsExact⟩	⟨Postcode, sameFirst3Char⟩
Cora	
⟨(Authors, Venue), concatenatedIsExact⟩	⟨Authors, sameFirst2Token⟩
⟨(Title, Venue), concatenatedIsExact⟩	⟨Title, sameFirst2Token⟩
⟨(Authors, Year), *concatenatedIsExact*⟩	⟨Venue, sameFirst4Token⟩
DBLP-ACM	
⟨(Authors, Year), concatenatedIsExact⟩	⟨Authors, sameFirst3Token⟩
⟨Title, sameFirst2Token⟩	⟨Title, sameFirst3Token⟩
⟨(Venue, Year), concatinatedIsExact⟩	⟨Venue, sameFirst3Token⟩

For generating the training data sets (described in Section 8.3.2) all records in the data sets (described in Table 8.2) are used. We use a lower threshold $lt = 0.1$ and an upper threshold $ut = 0.7$ to weakly label record pairs into positive and negative pairs. The generated training data sets are then used in our learning algorithm as described in Section 8.3.3 to learn the optimal BKs $\mathbf{O}$. We use $n_{max} = 100$ for the maximum allowed number of covered vectors from $\mathbf{V}_N$, we use a maximum block size of $sm = 100$, and for the weight parameters we use $\alpha = 0.2$ and $\beta = 0.4$. Weights and thresholds that we use are selected based on an experimental investigation of using different values to achieve better average query time while maintaining similar recall values compared to the baseline. We aim to investigate learning these values to produce blocks with high quality and small size in our future work.

8.4.1 Efficiency of Selected Blocking Keys

To evaluate the efficiency of the BKs selected by our learning algorithm we use query time (which is the time required to resolve a query record), number of candidate records (which is the number of the generated candidate records that are required to resolve a query record), and BK distribution (which is the frequency distribution of the sizes of all blocks generated using a BK) as measures to examine whether our

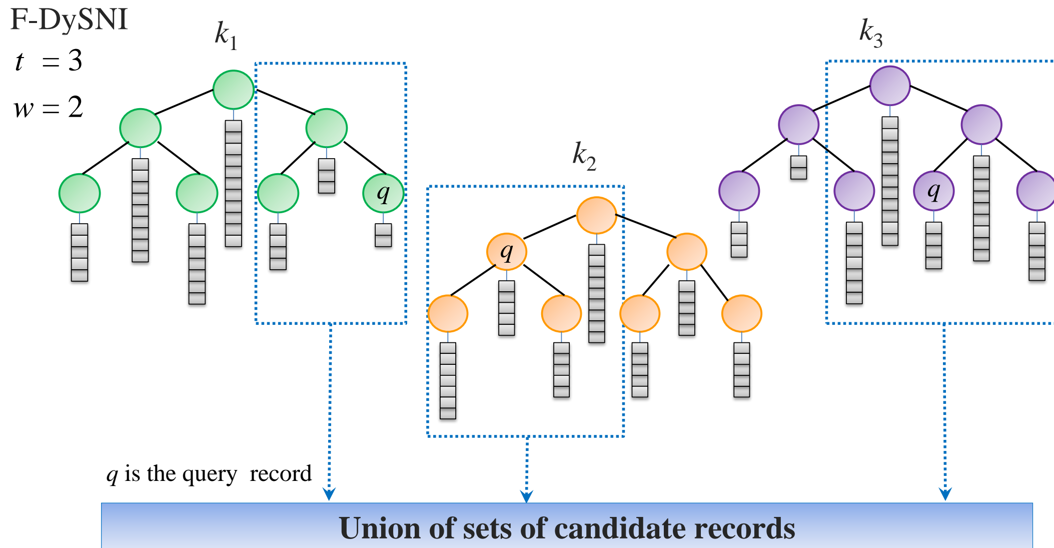


Figure 8.4: The F-DySNI used in the experimental evaluation. $t = 3$ is the number of trees used in the index. $w = 2$ is number of the nodes included in the window in each side. $k1$, $k2$, and $k3$ are the keys used to build the trees in the index.

proposed BK learning algorithm can learn BKs that are suitable for building indexes to be used with real-time ER. We build the F-DySNI (as described above) first using the keys selected by our algorithm, then using the keys selected by the FDJ. Then, we perform real-time ER resolution using those two indexes to capture the results for the measures described above and compare the values as follows:

The left plot in Figure 8.5 presents the query time required to resolve a single query in milliseconds (ms). The results show that the BKs selected by our approach improve the efficiency of query matching significantly. The selected keys using our proposed approach achieve an average query time of 1.08, 7.82, and 9.29 ms for the OZ-1, Cora, and DBLP-ACM respectively, while the selected keys using the FDJ approach achieve an average query time of 206.0, 175.4, and 937.7 ms for OZ-1, Cora and DBKP-ACM respectively. As can be seen from these values, our selected BKs are around two orders of magnitude faster than the FDJ approach for the OZ-1 and the DBLP/ACM data sets while around one order of magnitude faster for the Cora data set. Note that these timing measures are produced while using the same indexing technique, comparison functions, thresholds, and weights for the proposed and the FDJ approaches. Also note that these results are achieved while maintaining the quality of the matching process as will be described in Section 8.4.2.

The right plot in Figure 8.5 shows the distribution of the block sizes generated using the keys selected by the proposed and the FDJ approaches on the OZ-1 data set (the other two data sets were too small to clearly show how the blocks are distributed). The results show that the keys selected by our approach lead to block sizes that do not exceed the maximum allowed block size. It also shows that the generated blocks using our approach have similar sizes. In contrast, the keys selected by the

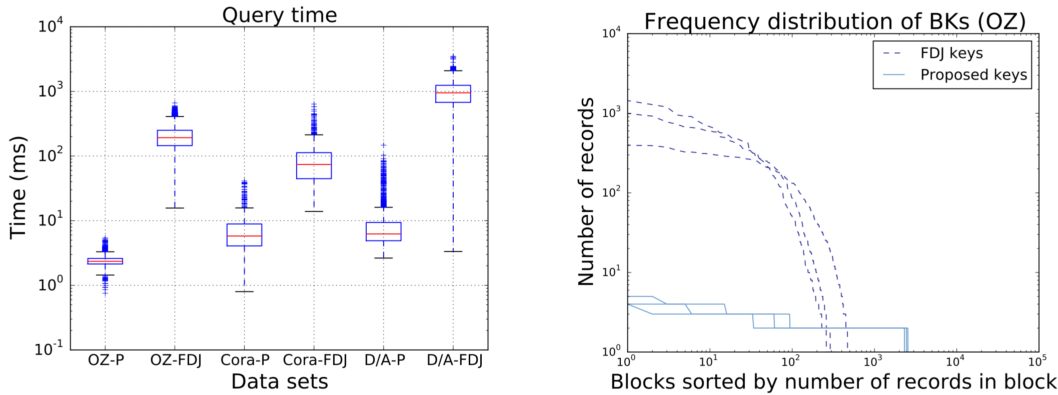


Figure 8.5: Plot on the left shows time measures for the different data sets generated using the keys selected by our approach (presented by the first, third, and fifth box plots) and the FDJ approach (presented by the second, fourth, and sixth box plots). D/A refers to DBLP-ACM and P refers to our proposed approach. Plot on the right shows the frequency distribution of the sizes of the blocks generated using the BKs selected by the proposed approach (presented by the solid lines) and the FDJ approach (presented by the dashed lines) on the OZ-1 data set.

Table 8.4: Statistics for the number of candidate records generated using F-DySNI with $t = 3$ trees and a window $w = 2$. The keys selected by our learning approach and by the FDJ approach are used to build the trees in the index.

	OZ-1		**Cora**		**DBLP-ACM**	
	Proposed	FDJ	Proposed	FDJ	Proposed	FDJ
Average	10	2,041	28	274	30	2,643
Median	10	1,936	26	268	17	2,853
St.deviation	2	800	11	107	30	1,170
Minimum	6	162	10	54	10	13
Maximum	18	6,134	70	583	235	4,389

FDJ approach lead to blocks of various sizes. These results are also supported by Table 8.4 where the standard deviation values (which measure the amount of variation from the average) of the number of candidate records required using our selected keys are small (compared to the values of the FDJ approach) for the three data sets. This means that the block sizes tend to be close to the average block size.

Table 8.4 show various statistical measures for the number of candidate records generated using the proposed and the FDJ approaches. It is clear from values in the table that the proposed approach has decreased the number of candidate records greatly which is the reason behind the significant drop in query times for the proposed approach compared to the FDJ approach.

From the results discussed above, we can conclude that the BKs selected by our BK learning algorithm can be used to build fast and efficient indexes that can be used with real-time ER.

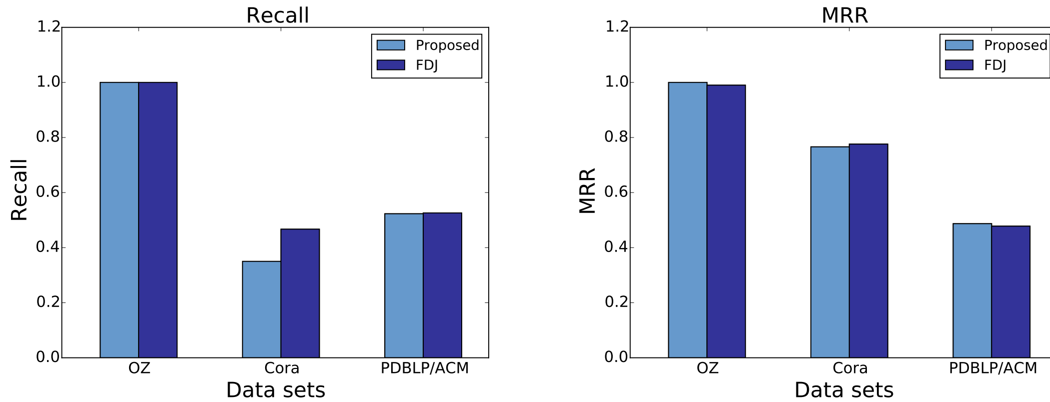


Figure 8.6: Recall and MRR measures for the different data sets generated using the keys selected by our approach and the FDJ approach.

8.4.2 Effectiveness of Selected Blocking Keys

To evaluate the effectiveness of the BKs selected by our learning algorithm we use recall (which is the proportion of the actual true matches that have been classified correctly) and the mean reciprocal rank (MRR) (which is the average of the reciprocal of the rank of the first true matching record in the returned result) as measures to examine whether our proposed BK learning algorithm can learn BKs that are suitable for building indexes that can be used with real-time ER without affecting the quality of the matching process (details about the used measures can be found in Chapter 4).

Figure 8.6 shows the recall and the MRR values that are measured when building the F-DySNI using both BKs that are selected by our proposed learning algorithm and by the FDJ algorithm. The results from the left plot if the figure show that our approach achieves similar recall values compared to the FDJ approach. For both the OZ-1 and the DBLP-ACM data sets our approach achieved the same recall value as the FDJ, while it achieved a 5% lower recall value for the Cora data set. This is because Cora contains entities with high frequency duplicates in the data set which could lead to generating large block sizes when building the index. In our algorithm, keys that generate blocks with a size that is above a maximum number of records s_m are removed from $\mathbf{K}$ during the key learning process which cause the drop in recall values since the true matches that are within the removed large blocks are not found. As for the MRR, our proposed learning algorithm achieved similar results as the FDJ approach for the three different data sets used. From this we can conclude that our approach managed to maintain the matching quality achieved by the FDK approach while improving the query time significantly (around two orders of magnitude).

Notice that recall and MRR values achieved for the Cora and the PDBLP/ACM data sets are lower than those achieved for the OZ data set for both the proposed and FDJ techniques. The main reason for missing true matches for the Cora and the DBLP/ACM data sets is the generated list of candidate keys. As discussed in

Section 8.3.1, this list contains all keys which we select our optimal keys from and can differ based on the domain, the indexing technique and the matched data sets. The currently generated candidate keys did not work well with the bibliographic data (which have long attribute values) in both Cora and PDBLP/ACM since the current candidate keys are based on exact matching of tokens (i.e. individual characters, separate words, terms, or abbreviations) generated from attribute values which lead to missing a high number of true matches. The following example illustrates this issue: Consider the following two records, that represent the same entity, from the Cora data set:

Record ID	**Entity ID**	**Authors**
108	drucker1992	h. drucker, r. schapire, and p. simard,
109	drucker1992	drucker, h., schapire, r., simard, p.

and consider using the key <Authors, sameFirst2Tokens> to build the F-DySNI for the Cora data set. In this case, these two records will not be inserted in the same tree node and will not be considered as matching records. This is because the first two tokens of the 'Authors' attribute for both records are swapped. This issue can be solved by carefully selecting candidate keys that consider the nature of long attribute values of bibliographical data sets (for instance using the number of common tokens or n-grams in the attribute value to calculate a similarity between two records). Investigating more possible candidate key functions is to be studied in future work.

8.5 Summary

In this chapter we proposed a general unsupervised blocking key selection algorithm that automatically selects optimal BKs for building indexes that can be used with real-time ER. Our algorithm takes into consideration both effectiveness and efficiency of the real-time ER process when learning optimal keys. We evaluated our approach using three real-world data sets and compared it with an existing automatic blocking key selection technique. The results show that our approach can learn keys that are suitable for real-time ER. The keys selected by our approach reduced query times significantly while maintaining matching quality.

Some future research directions that can be extended from our work in this chapter is to investigate how to automatically identify candidate blocking functions based on the content of the data sets, investigate learning the weights that are used in our key selection algorithm to produce blocks with high quality and small size, and compare our proposed approach with other existing blocking key selection techniques.

Example 8.1: To calculate an overall score for candidate blocking keys $\mathbf{K} = \{k_1, k_2, k_3\}$, assume that the average block size produced by each blocking key is $\mathbf{B} = \{12, 5, 14\}$. Also assume that the sets of blocking key vectors $\mathbf{V}_P$ and $\mathbf{V}_N$ presented in Figure 8.7 are generated from the positive and negative training data sets (as described in Section 8.3.2). Figure 8.7 also shows the mean, and variance values calculated for each blocking key in both vector sets ($\mathbf{V}_P$ and $\mathbf{V}_N$) and in their union $\mathbf{V}_P \cup \mathbf{V}_N$.

$\mathbf{V}_P$

	k_1	k_2	k_3
	1	1	0
	0	1	0
	1	1	1
	0	1	0
	0	0	0
$\mu_{k,p}$	0.40	0.80	0.20
$\sigma_{k,p}$	0.24	0.16	0.16
$\sigma^2_{k,p}$	0.05	0.02	0.02

$\mathbf{V}_N$

	k_1	k_2	k_3
	1	1	1
	1	0	1
	0	0	1
	1	1	0
	0	0	1
$\mu_{k,n}$	0.60	0.40	0.80
$\sigma_{k,n}$	0.24	0.24	0.16
$\sigma^2_{k,n}$	0.05	0.05	0.02

$\mathbf{V}_P \cup \mathbf{V}_N$

	k_1	k_2	k_3
	1	1	0
	0	1	0
	1	1	1
	0	1	0
	0	0	0
	1	1	1
	1	0	1
	0	0	1
	1	1	0
	0	0	1
μ_k	0.50	0.60	0.50

Figure 8.7: Blocking key vectors $\mathbf{V}_P$ and$\mathbf{V}_N$ that are used in Example 8.1 to calculate the overall scores for the blocking keys $\mathbf{K} = \{k_1, k_2, k_3\}$. $\mathbf{V}_P \cup \mathbf{V}_N$ represents the union of both key vectors.

Based on the vectors in Figure 8.7, we calculate the overall score sc_k for each candidate blocking key in $\mathbf{K}$ using Equation 8.6 which relies on three criteria (key coverage, average block size for the blocks generated by the key, and the distribution of the size for the blocks generated using a blocking key). Moreover, assume having the weight parameters $\alpha = 0.2$ and $\beta = 0.4$. First we need to calculate the coverage c_k of each key using Equation 8.4 as follows:

Key	**Key coverage (c_k)**
k_1	$\frac{5(0.4-0.5)^2+5(0.6-0.5)^2}{5*0.05+5*0.05}=0.2$
k_2	$\frac{5(0.8-0.6)^2+5(0.4-0.6)^2}{5*0.02+5*0.05}=1.14$
k_3	$\frac{5(0.2-0.5)^2+5(0.8-0.5)^2}{5*0.02+5*0.02}=4.5$

We also calculated the distribution of blocks generated by each blocking key using Equation 8.5, assuming the block sizes in the following table as follows:

Key	**Block sizes**	**Average block size ($s_{b_{(ave)_k}}$)**	**Distribution of blocks (v_k)**
k_1	10, 12, 15	12	$\frac{(10-12)^2+(12-12)^2+(18-12)^2}{3}=13.3$
k_2	3, 5, 7	5	$\frac{(5-5)^2+(3-5)^2+(7-5)^2}{3}=2.66$
k_3	10, 13, 20	14	$\frac{(10-14)^2+(13-14)^2+(20-14)^2}{3}=17.6$

Then we calculate the overall score sc_k using our proposed equation (see Equation 8.6), and weight parameters $\alpha = 0.2$ and $\beta = 0.4$ as follows:

Key	**Coverage (c_k)**	**Average block size ($s_{b_{(ave)_k}}$)**	**Distribution of blocks (v_k)**	**Overall score**
k_1	0.2	12	13.3	10.29
k_2	1.14	5	2.66	3.03
k_3	4.5	10	17.6	11.94

Table 8.5: List of candidate keys and their coverage (c_k), average block size (s_b), variance (v_k), and overall score (sc_k) for the OZ-1 data set. Note that keys with smaller overall scores are more suitable for real-time ER. The overall key score sc_k is calculated using Equation 8.6. c_k, s_b, v_k values are normalized into a value between 0 and 1 before calculating the overall key scores. The values in the table represent the calculations required to select the first optimal key (which is highlighted in red bold font). Note that after selecting the first optimal key it will be removed from the candidate key list and all records covered by this key are also removed from the training data sets. To select a second optimal key, all calculations are repeated for all remaining keys on the new training data set (that does not include records covered by the first selected key). The three keys selected by our approach at the end of the key selection iterations are: ⟨(Firstname, Suburb), concatenatedIsExact⟩, ⟨(Surname, Suburb), concatenatedIsExact⟩, and ⟨(Surname, Postcode), concatenatedIsExact⟩.

No.	Key	c_k	s_b	v_k	sc_k
1	⟨Firstname, isExact⟩	9.28	4.24	4.33	0.59
2	⟨Firstname, sameFirst1Char⟩	84.32	1330.3	0.77	0.32
3	⟨Firstname, sameFirst2Char⟩	252.01	97.4	2.35	0.21
4	⟨Firstname, sameFirst3Char⟩	95.09	17.4	3.67	0.46
5	⟨Firstname, sameFirst4Char⟩	26.90	7.6	4.03	0.55
6	⟨Firstname, sameFirst5Char⟩	13.6	5.4	4.22	0.58
7	⟨Surname, isExact⟩	6.07	2.1	2.57	0.42
8	⟨Surname, sameFirst1Char⟩	38.16	1330.3	0.77	0.36
9	⟨Surname, sameFirst2Char⟩	54.89	95.02	1.78	0.32
10	⟨Surname, sameFirst3Char⟩	36.49	12.5	2.35	0.38
11	⟨Surname, sameFirst4Char⟩	18.26	4.02	2.59	0.41
12	⟨Surname, sameFirst5Char⟩	9.91	02.6	2.63	0.43
13	⟨Suburb, isExact⟩	10.53	3.93	1.78	0.34
14	⟨Suburb, sameFirst1Char⟩	27.33	1383.5	0.78	0.37
15	⟨Suburb, sameFirst2Char⟩	135.36	149.09	1.47	0.22
16	⟨Suburb, sameFirst3Char⟩	178.74	27.67	1.92	0.22
17	⟨Suburb, sameFirst4Char⟩	164.32	12.03	2.15	0.25
18	⟨Suburb, sameFirst5Char⟩	72.33	7.75	2.34	0.35
19	⟨Postcode, isExact⟩	12.10	11.94	1.33	0.30
20	⟨Postcode, sameFirst1Char⟩	7.95	3843.1	1.07	0.67
21	⟨Postcode, sameFirst2Char⟩	110.33	494.11	1.3	0.26
22	⟨Postcode, sameFirst3Char⟩	31.66	64.65	1.35	0.29
23	⟨Postcode, sameFirst4Char⟩	12.10	11.94	1.33	0.3
24	⟨Postcode, sameFirst5Char⟩	12.10	11.94	1.33	0.3
25	⟨(Firstname, Surname), concatenatedIsExact⟩	9.28	1.08	0.27	0.19
26	⟨(**Firstname**, **Suburb**), concatenatedIsExact⟩	**9.28**	**1.08**	**0.27**	**0.19**
27	⟨(Firstname, Postcode), concatenatedIsExact⟩	9.28	1.10	0.3	0.20
28	⟨(Surname, Suburb), concatenatedIsExact⟩	6.07	1.07	0.25	0.20
29	⟨(Surname, Postcode), concatenatedIsExact⟩	6.07	1.08	0.26	0.20
30	⟨(Suburb, Postcode), concatenatedIsExact⟩	10.53	2.87	1.78	0.34

Chapter 9

Comparative Evaluation

In the previous chapters we have proposed several indexing techniques that are tailored for real-time entity resolution, and one learning technique that automatically selects blocking/sorting keys to be used with real-time entity resolution. In this chapter, we comparatively evaluate the effectiveness and efficiency of the different proposed solutions. In Section 9.2 we describe the comparative evaluation setup and framework. In Section 9.3 we present the comparative evaluation results for the different indexing techniques and in Section 9.4 the results for the automatic blocking key selection technique. Finally, we summarize our findings in Section 9.5.

9.1 Introduction

In Chapter 3, we identified the need for developing novel efficient indexing techniques that are suitable for use with real-time entity resolution (ER) on dynamic databases. We also identified the need for novel techniques that learn optimal blocking/sorting keys for different real-time ER domains without the need for human intervention. We have addressed the first problem in Chapters 5, 6, and 7 by proposing several efficient dynamic indexing techniques that are tailored for real-time ER. In addition, we addressed the second problem in Chapter 8 by proposing a general learning technique that automatically selects optimal keys for building indexes that can be used with real-time ER.

In this chapter we provide an empirical comparative evaluation of the different proposed solutions with regard to their effectiveness and efficiency. We first compare our indexing solutions (presented in Chapters 5, 6, and 7) with a current indexing technique that is used in ER [11, 34, 107] (as described in Section 4.3). Then, we compare our automatic blocking/sorting key selection algorithm (proposed in Chapter 8) with manual key selection that requires expert knowledge. The results of both evaluations are presented in the following sections. Table 9.1 summarize the abbreviations that we use in this Chapter.

Table 9.1: Summary of abbreviations used in this chapter

DySimII	The similarity-aware inverted index (proposed in Chapter 5)
DySNI	The dynamic sorted neighborhood index (proposed in Chapter 6)
F-DySNI(2)	The forest-based dynamic sorted neighborhood index / with 2 trees (proposed in Chapter 7)
F-DySNI(3)	The forest-based dynamic sorted neighborhood index / with 3 trees (proposed in Chapter 7)
QGI	Q-gram based indexing technique that is used in entity resolution (described in Chapter 4)

Table 9.2: Data sets summary. The selected data sets and why they are used are described in more details in Section 4.5.

Data sets	Provenance	No. Records	No. Duplicates	No. Entities	Ave Duplicates
NC	Real	7,997,234	150,089	7,847,145	3.0
OZ-(1,2,3,4)	Real (modified)	345,876	172,938	172,938	3.5
Febrl-5	Synthetic	100,000	80,000	20,000	5.0
Febrl-10	Synthetic	100,000	90,000	10,000	10.0
Febrl-20	Synthetic	100,000	95,000	5,000	20.0

9.2 Comparative Evaluation Setup

To conduct the comparative evaluation we use a large real-world data set and several synthetic data sets with different characteristic. Table 9.2 provides a summary for the used data sets. We use the NC data set, which has around 8 million records, to conduct the scalability evaluation of the compared techniques. The four OZ data sets have an average of 3.5 duplicate per record, however, they differ in their corruption ratio. In OZ-1, duplicates have been corrupted by modifying only one of their attributes, while in OZ-4 duplicates have been corrupted by modifying all of their four attributes. These data sets are used to evaluate the effect of having different number of corruption ratios on the quality of the various indexing techniques. On the other hand, the three Febrl data sets differ in the average number of duplicates for a single record. Febrl-(5,10,20) have an average of 5, 10, and 20 duplicates per record respectively. These data sets are used to evaluate the effect of having different number of duplicates on the quality of the evaluated techniques.

We use the average insertion and query times to evaluate the efficiency and scalability of the different indexing techniques. In addition, we use recall (the proportion of the actual true matches that have been classified correctly) and the mean reciprocal rank (MRR) (the average of the reciprocal of the rank of the first true matching record in the returned result) to evaluate the matching quality. More details about the data sets, the evaluation measures, the used baseline, and the implementation environment are found in Chapter 4.

9.3 Indexing Techniques

In this section we comparatively evaluate all proposed indexing solutions together with an indexing technique that is used in ER [11, 34, 107]. The compared techniques include the similarity-aware inverted index proposed in Chapter 5 (labeled as DySimII), the dynamic sorted neighborhood index proposed in Chapter 6 (labeled as DySNI), the forest-based sorted neighborhood index using two and three trees proposed in Chapter 7 (labeled as F-DySNI(2) and F-DySNI(3) respectively), and the q-gram based index [11, 34, 107] described in Section 4.3 (labeled as QGI).

The QGI technique is a q-gram based inverted index [11, 34, 107] that converts the attribute values of each record in a data set into a list of q-grams (sub-strings of length q). Each unique q-gram becomes a key in the inverted index where its corresponding value is the list of all records in a data set that have this q-gram in their attribute values. More details on how the QGI approach operates are found in Section 4.3.

9.3.1 Scalability

We comparatively evaluate the scalability of the different indexing techniques on the NC data set (The CCA data sets are not used for scalability testing since the QGI technique is shown to be slow to run on very large data sets). We measure the average time required to insert a single record into an index data structure, and the average query time required to resolve a single query record across the growing size of the index structure. 10% of the NC data set is used to build the indexes, and the rest of the records are considered as query records (note that query records are inserted into the index upon arrival).

- **DySimII:** We use the Double-Metaphone [33] encoding function to encode the attribute values when building the index as described in Section 5.3.2.

- **DySNI:** We use a concatenation of the 'Surname' and 'Firstname' attributes as a sorting key (SK) to build the index data structure, and we use the similarity-based adaptive window approach to generate the set of candidate records using a similarity threshold of $\theta = 0.8$, where $0 \leq \theta \leq 1$. Building the index and generating the set of candidate records are described in detail in Sections 6.3.2 and 6.4.3 respectively. Note that we did not use single attribute values as SKs because they are not suitable for real-time ER (based on the results presented in Chapter 7).

- **F-DySNI(2):** We build two trees in the index data structure. For the first tree we use the 'Firstname+Surname' concatenation as a SK (the same SK used for DySNI), and for the second tree we use the 'Surname+Firstname' concatenation. We use the same window approach and similarity threshold as in the DySNI to generate the set of candidate records.

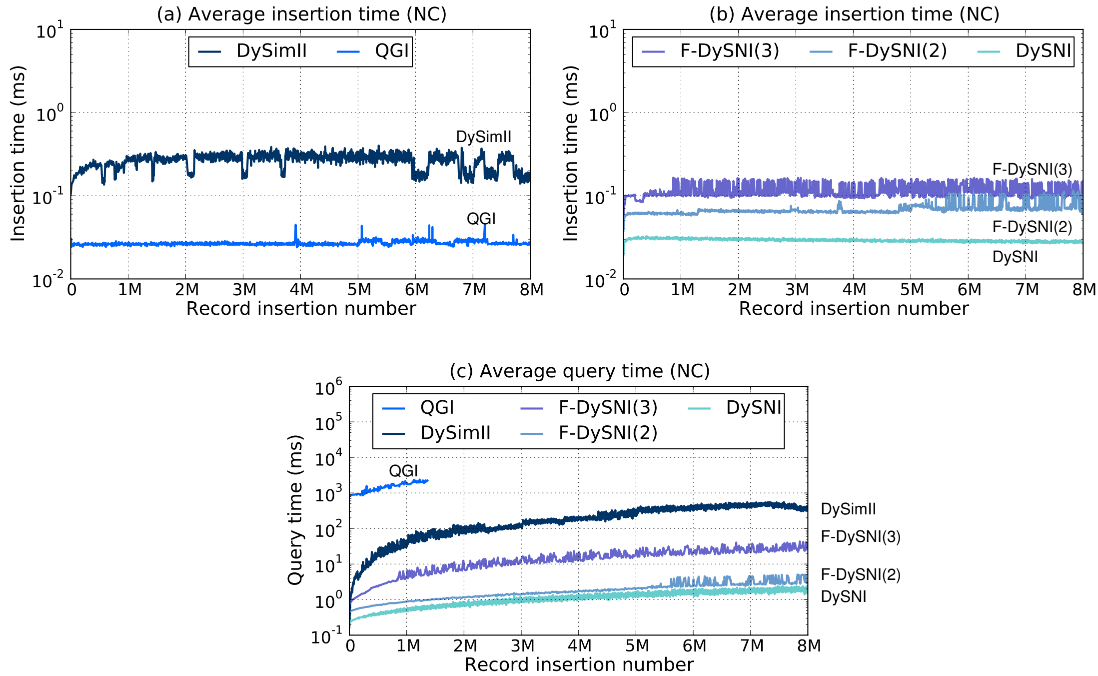


Figure 9.1: Plots (a) and (b) show the average time required to insert a single record into the index using the compared indexing techniques (the results are split over two plots to improve readability). Plot (c) illustrates the average time required to query the growing index. The results for the QGI technique are not complete because the technique was slow and it was not feasible to finish running the experiment. The Full NC data set (described in Section 4.5) is used to build the indexes (M = million).

- **F-DySNI(3):** We build three trees in the index data structure. For the first and second trees we use the same SKs as in F-DySNI(2), and for the third tree we use the 'Firstname+City' concatenation as a SK. We use the same window approach and similarity threshold as in the DySNI to generate the set of candidate records.
- **QGI:** We use q-grams of length $q = 2$ to convert the attribute values of each record in the data set into a list of q-grams and each unique q-gram becomes a key in the inverted index. How the index operates is explained in detail in Section 4.3

Figure 9.1, illustrates the scalability results for all compared techniques. Note that the results for the average query times for the QGI technique (in plot (c)) are not complete as it was not feasible to run the ER process for the full 8 million records. This is because the technique was slow and required high query times (an average of around 1.5 seconds for the first 1.5 million records).

Plots (a) and (b) in Figure 9.1 present the average insertion time required by all compared techniques (we split the results over two plots to improve readability). The results show that the average insertion times for all compared techniques are

not affected by the growing size of the index data structure (almost constant). The average insertion times for the compared techniques ranges between 0.05 to 0.4 milliseconds (ms). The DySimII achieves an average insertion time of around 0.4 ms, the F-DySNI(3) around 0.1 ms, the F-DySNI(2) around 0.08 ms, and both DySNI and QGI techniques achieve around 0.05 ms. The results presented in Figure 9.1 (a) and (b) confirm that the process of inserting a record into the index data structure is scalable to large data sets for the compared techniques.

The results for the average query times achieved by all compared techniques are presented in Figure 9.1 (c). From the plot it is clear that the average query times for all techniques increases sub-linearly as the index becomes larger. However, the QGI technique has a high average query time (around 1.5 second) that makes it not suitable for real-time ER and not scalable with large data sets.

The fastest technique was the DySNI, which achieved an average query time of 1.15 ms, followed by the F-DySNI(2) and F-DySNI(3) techniques that achieved an average query time of 1.9 ms and 15.7 ms respectively. The reason behind the increase in query times when we use more trees in the index data structure is that having more trees leads to an increase in the number of candidate records which then leads to an increase in the average query time. The slowest among all proposed solutions is the DySimII that achieved an average query time of 225 ms (although it is the slowest, it is still fast enough to be used with real-time ER).

The presented results confirm that the proposed indexing techniques are suitable for real-time ER (where query records need to be matched in sub-second times) and are scalable to large data sets. The effectiveness and efficiency of the compared techniques are evaluated next.

9.3.2 Effectiveness and Efficiency

We next evaluate the compared indexing techniques (DySimII, DySNI, F-DySNI(2), F-DySNI(3), and QGI) with regard to their effectiveness (matching quality) and efficiency using the OZ-x data sets (with different corruption ratios) and the Febrl data sets (with different number of duplicates per entity). More details about the data sets are found in Section 4.5.

We measure recall and MRR values to comparatively evaluate the effectiveness of the compared techniques, and we measure the average query times to comparatively evaluate the efficiency of the compared techniques. Details about this evaluation are described in the following sub-sections.

9.3.2.1 The Effect of Having Different Corruption Ratios in a Data Set

In this set of experiments we evaluate the effect of having different error rates in a data set on the effectiveness and efficiency of the compared indexing techniques. We use the OZ-x data sets (with different corruption ratios) to conduct this set of experiments. The settings used for running the different techniques are:

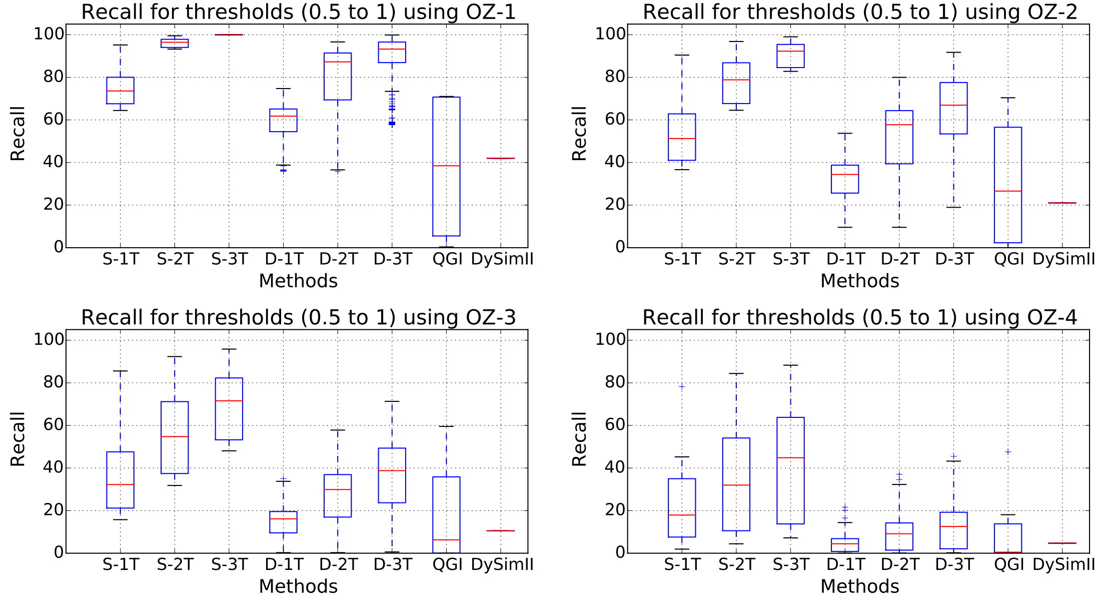


Figure 9.2: Recall values for all compared indexing techniques using the OZ-x data sets. S refers to using single attribute values as SKs and D refers to using a concatenation of two attribute values as SKs. 1T refers to the DySNI which uses a single tree, 2T refers the F-DySNI(2) which uses two trees, and 3T refers to the F-DySNI(3) which uses three trees. Note that the threshold used to generate the plots is different between the techniques; for the DySNI and F-DySNI it represents the similarity threshold between the SKV of the query's node and its neighboring nodes and it is used to generate the candidate records. For QGI the threshold represents the minimum Jaccard similarity between a query record and a candidate record and it is used to classify record pairs into matches and non-matches.

- **DySimII:** We use the Double-Metaphone [33] encoding function to encode the attribute values when building the index as described in Section 5.3.2. We use an overall similarity threshold of $t = 0.75$ to classify record pairs into matches or non-matches.

- **DySNI:** We use all possible single attribute values as SKs and we refer to such keys as (**S**). We also use all possible combinations of the concatenations of two (double) attribute values as SKs and we refer to such keys as (**D**). We use the similarity-based adaptive window approach to generate the candidate records with all similarity thresholds between $0.5 \leq \theta \leq 1.0$. Moreover, we use an overall similarity threshold of $t = 0.75$ to classify record pairs into matches and non-matches.

- **F-DySNI(2):** The same settings from the DySNI apply for this technique but with two trees in the index data structure. We use all possible key combinations for the two trees for both single and double keys (generated from a concatenation of two attribute values).

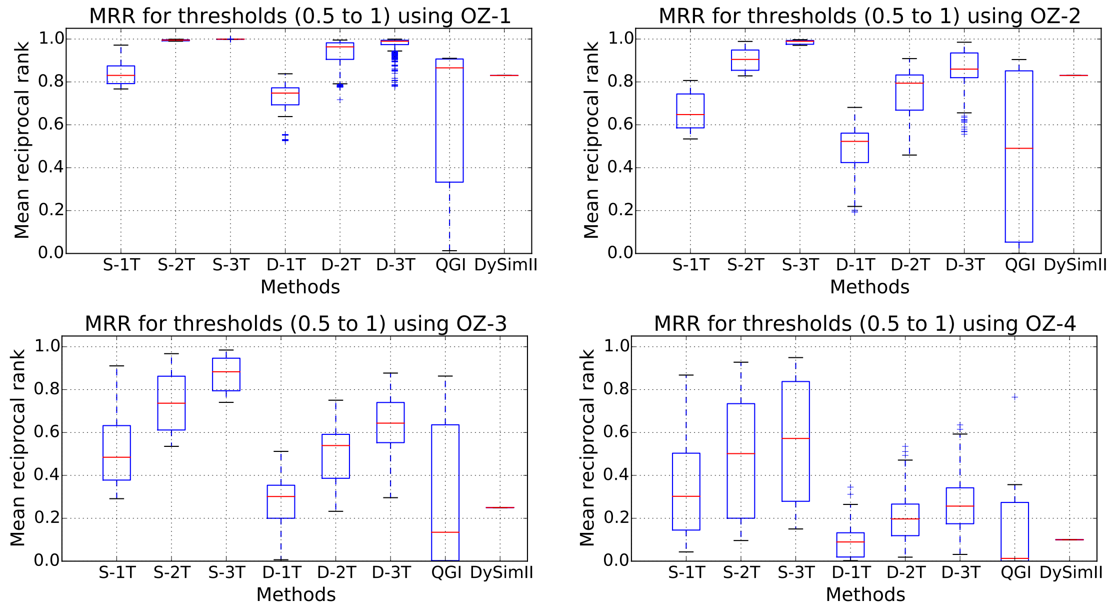


Figure 9.3: MRR values for all compared indexing techniques using the OZ-x data sets. The description in the caption of Figure 9.2 also applies here.

- **F-DySNI(3):** The same settings from the DySNI apply for this technique but with three trees in the index data structure. All possible key combinations for the three trees are used for both single and double keys (generated from a concatenation of two attribute values).

- **QGI:** We use a q-gram of length $q = 2$ to convert the attribute values of each record in a data set into a list of q-grams and each unique q-gram becomes a key in the inverted index. We use a minimum Jacard similarity threshold that is $0.5 \leq t \leq 1.0$. Details about how the QGI operates are found in Section 4.3

Recall: We measure recall values for all compared approaches. The results, presented in Figure 9.2, show that recall values for all compared techniques, as one would expect, are affected by the number of corrupted attribute values in the data sets. Higher corruption ratios lead to lower recall values. All techniques achieved better results on OZ-1 (where only one attribute value is corrupted) while the lowest recall values are achieved on OZ-4 (where all attribute values are corrupted). For example, the median of recall values achieved by the F-DySNI(3) technique using double SKs (labeled as D-3T in the plots) has dropped from around 92% on OZ-1 to around 17% on OZ-4 (a decrease by around 80%). A similar decrease in recall values between OZ-1 to OZ-4 data sets apply to all compared techniques as well.

The F-DySNI(3), F-DySNI(2), and DySNI using single attribute values as SKs achieved, respectively, the highest recall values compared to the other techniques. However, as discussed in Section 7.5.1, and as will be discussed in Section 9.3.2.2, although single attribute values when used as SKs achieve better recall values, they are not suitable for real-time ER as they require longer times to match query records.

As for using concatenated attribute values as SKs (double keys) with DySNI, F-DySNI(2), and F-DySNI(3), they all achieve better recall values than both DySimII and QGI. For example, for the OZ-1 data set the median of achieved recall values for the F-DySNI(3) is around 92%, for F-DySNI(2) it is around 89%, and for DySNI it is around 62%, while both DySimII and QGI techniques achieved a recall median of around 40%. This is noticeable in all other OZ-x data sets as well. All proposed techniques achieve better recall values than the QGI baseline for all OZ data sets except for the OZ-2 where QGI achieves slightly better recall value than DySimII. The DySimII technique did not perform well on the OZ data sets compared to other proposed techniques. This is because, as discussed in Chapter 5, similar attribute values with errors and variations are not inserted into the right blocks as they should which increases the number of missed true matches.

MRR: We measure the MRR values for all compared techniques. The results, presented in Figure 9.3, show that these MRR values are affected by the number of corrupted attribute values in the data sets. Higher corruption ratios lead to lower MRR values. All compared techniques achieve higher MRR values on the data set with the lower corruption ratio (OZ-1), and lower MRR values on the data set with the highest corruption ratio (OZ-4). For example the median MRR value for the F-DySNI(3) technique using double SKs (labeled as D-3T in the plots) is around 0.99 on the OZ-1 data set. This median drops to around 0.24 on the OZ-4 (a decrease of around 75%). This applies for the other approaches as well. All proposed techniques achieve MRR values that are higher than the QGI technique for all OZ data sets. The F-DySNI(3) technique achieves the highest MRR values, followed by F-DySNI(2), DySNI, and DySimII.

Query times: We measure the average time required to match a query record using the OZ-x data sets for all compared techniques. The results presented in Figure 9.4 show that all proposed techniques outperform the QGI technique with regard to query times. The results also show that for DySNI and F-DySNI using single attribute values as SKs requires longer query times (but still achieve the sub-second query time required for real-time ER). However, using concatenated SKs (generated from a concatenation of two attribute values) improves the time by more than one order of magnitude (compared to single SKs). The fastest index among all compared techniques is the DySNI when using concatenated SKs (labeled as D-1T in the plots), followed by the F-DySNI(2) and F-DySNI(3). The reason behind the increase in query times when we use more trees in the index (for the DySNI and F-DySNI) is that having more trees leads to an increase in the number of candidate records which in turn increases the average query time. The query time achieved by the DySimII technique is similar to the median of the achieved query times by the F-DySNI(3), while QGI is the slowest among all techniques.

A summary of the above results is presented in Figure 9.5 (the results include using both single and double SKs for the DySNI and F-DySNI solutions). From this figure we can summarize that all proposed indexing techniques achieve better average query times than the QGI technique. The DySimII is around two orders of magnitude faster than the QGI, the DySNI is around one and a half orders of mag-

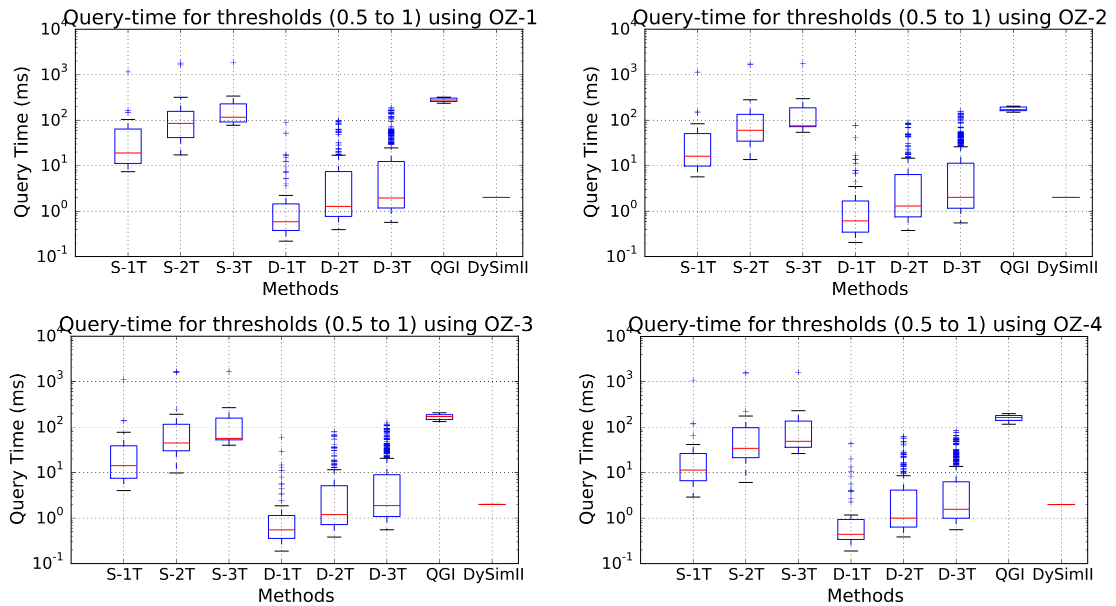


Figure 9.4: Query times for all compared indexing techniques using the OZ-x data sets. The description in the caption of Figure 9.2 also applies here.

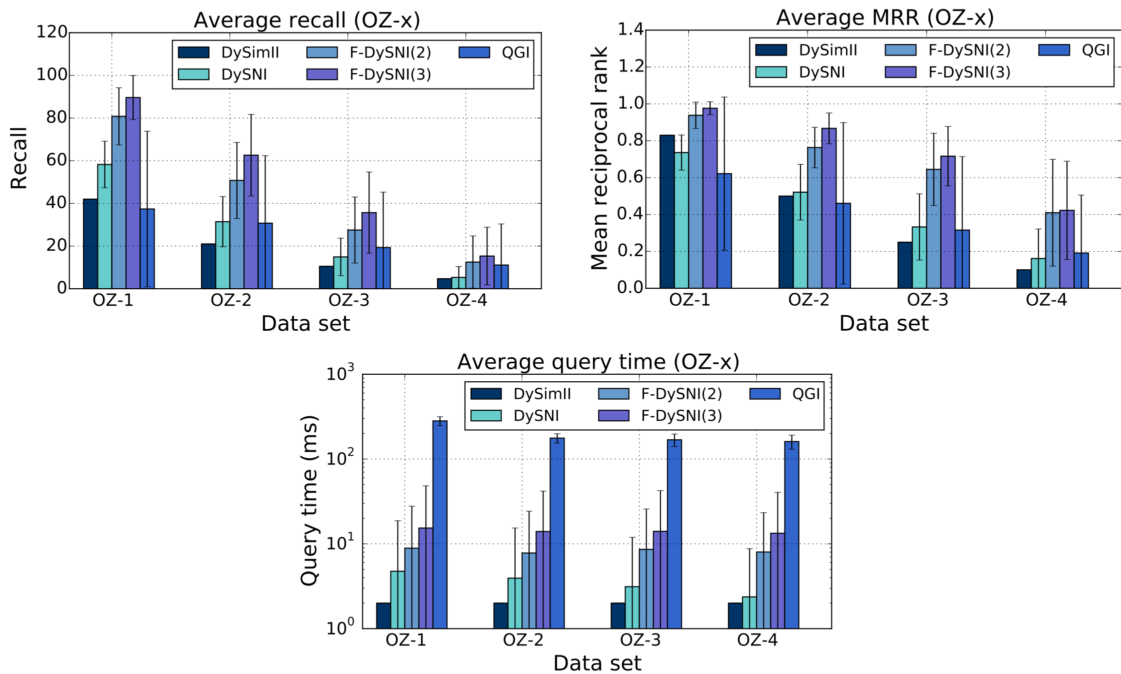


Figure 9.5: Summary of the average recall, MRR, query time values for the different techniques using the OZ-x data sets. The results for both DySNI and F-DySNI are generated using all possible SK combinations (both single and concatenated attribute values).

nitude faster, while the F-DySNI(3) is more than one orders of magnitude faster than the QGI technique. The average query time achieved by the proposed techniques ranges between 2 to 11 ms which is suitable for matching query records in real-time.

The figure also shows that the DySNI and F-DySNI(3) approaches outperform QGI on all OZ data sets with regard to matching quality (recall and MRR), and that the F-DySNI(3) achieves the highest quality results of all compared approaches. It outperforms the QGI technique (with regard to recall values) by 28% to 57% for the different OZ-x data sets, and by 22% to 50% with regard to MRR values. The F-DySNI(3) technique is shown to perform better than all other compared approaches on noisy data sets that contains errors and variations.

9.3.2.2 The Effect of Having Different Number of Duplicates in a Data Set

In the second set of experiments we evaluate the effect of having different number of duplicate records in a data set on the effectiveness and efficiency of the compared indexing techniques. We us the Febrl data sets (with different number of duplicates) to conduct this set of experiments. More details about the Febrl data sets can be found in Sections 4.5. The settings for building the DySimII and QGI techniques are similar to the settings used in the previous section. For DySNI and F-DySNI we also use the same settings as in the previous section but for the SKs we use the 'Firstname+Surname', the 'Surname+Firstname', and the 'Firstname+Postcode' to build the first, second, and third trees in the indexes. We select these keys since it was shown in the previous experiments that SKs generated from single attribute values are not suitable for real-time ER.

Recall and MRR: We measure both recall and MRR values using the different Febrl data sets. The results presented in Figure 9.6 show that all techniques achieve similar quality (recall and MRR) results on the different Febrl data sets except for the DySimII technique where its recall value drops from 32.1% on Febrl-5 to 12.6% on Febrl-20 (a decrease by around 60%). The MRR values for the same approach are not affected by the increase in number of duplicates.

The results, when using the Febrl data sets, also confirm the results in the previous sections (using the OZ-x data sets) since the F-DySNI(3) achieve the highest median of recall values for all Febrl data sets, followed by F-DySNI(2) and DyDNI. The DySimII technique did not perform well (with regard to recall) on all Febrl data sets but achieved relatively good MRR results. For the median of the MRR values for all approaches, the DySimII achieved the highest value followed by F-DySNI(3) and f-DySNI(2).

Query times: We measure the average query time required to match a query record using the Febrl data sets for all compared techniques. The results presented in Figure 9.7 show that all proposed techniques outperform the QGI technique. The results also show that the DySimII is the fastest among all compared techniques, followed by the DySNI, F-DySNI(2), and F-DySNI(3) respectively .

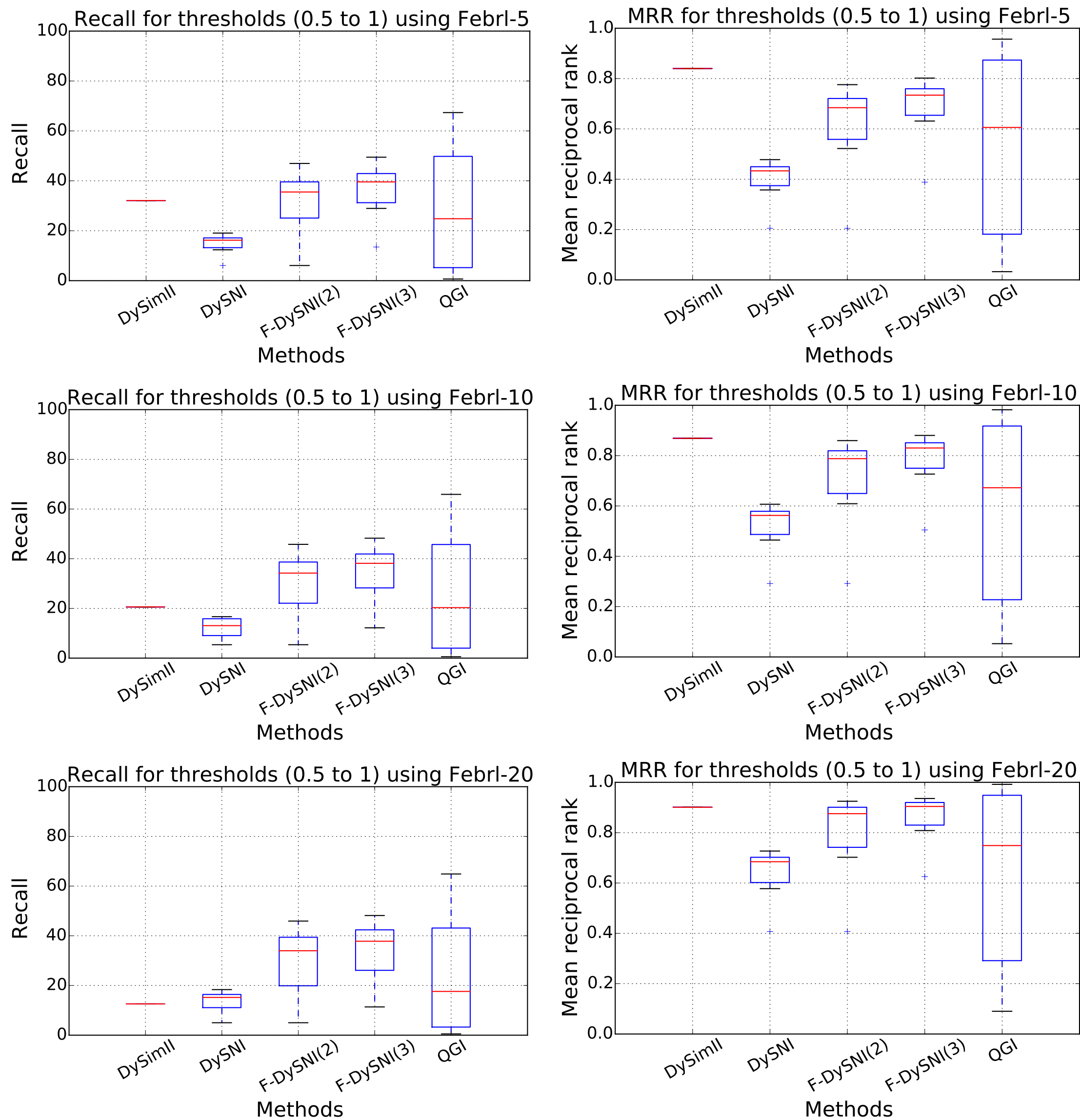


Figure 9.6: Recall and MRR values for all compared approaches using the Febrl data sets. The similarity-based adaptive window approach is used to generate candidate records using thresholds between 0.5 to 1.0.

A summary of all Febrl results is presented in Figure 9.8. From the figure we can see that all proposed indexing solutions achieve better average query times than the QGI technique. The DySimII is around two orders of magnitude faster than the QGI, while the F-DySNI(3) is around one order of magnitude faster. Moreover, all proposed solutions achieve better MRR values compared to the QGI technique for all Febrl data sets. As for the average recall values, both F-DySNI(2) and F-DySNI(3) outperform the QGI technique on all Febrl data sets while the DySimII outperform QGI only on Febrl-5 as its recall values is affected by the increased number of duplicates in the data sets.

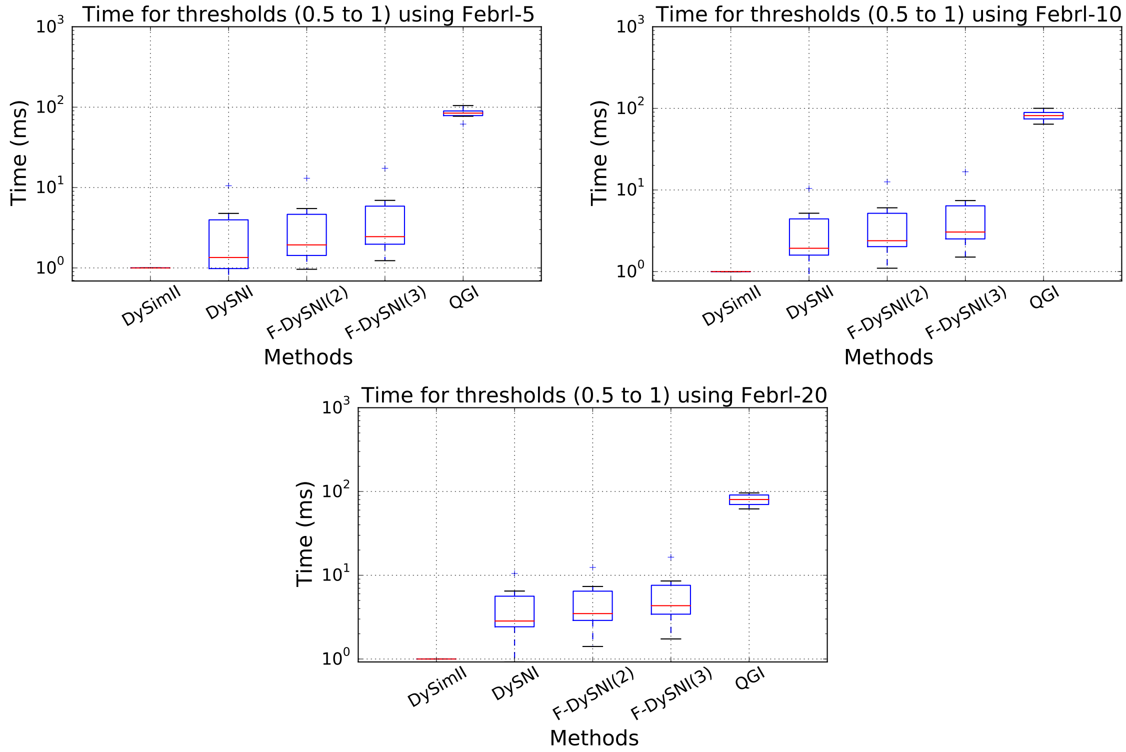


Figure 9.7: Average query times for all compared approaches on the Febrl data sets.

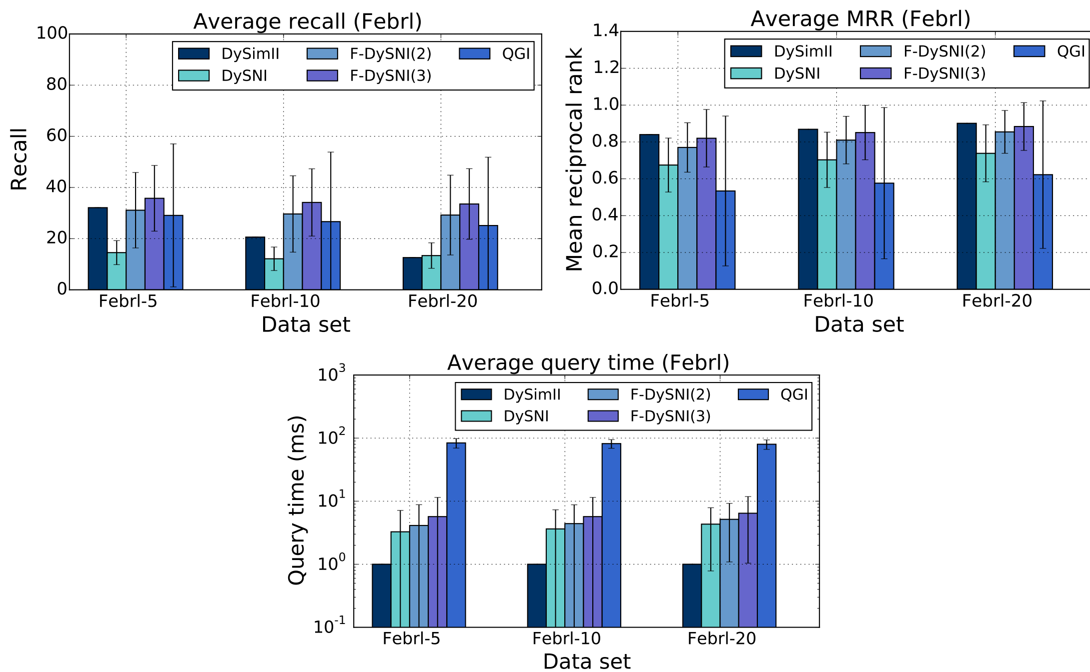


Figure 9.8: Summary of the average recall, MRR, query time values for the different techniques using the Febrl data sets.

From the presented results for both OZ-x and Febrl we can conclude that all proposed techniques DySimII, DySNI, and F-DySNI are suitable for real-time ER as they can match query records with sub-second times. However, DySNI achieves better matching quality and efficiency results compared to DySimII. This is because unlike the DySimII technique that generates larger block sizes (because of using encoded attribute values as blocking keys), the DySNI technique is based on using a tree-based index that generates small size nodes (blocks) when using suitable sorting keys (like using a concatenation of attribute values). Moreover, the DySNI can generate candidate records based on the similarities between the neighboring nodes (blocks) which lead to improving its matching quality.

The F-DySNI improves the matching quality achieved by the DySNI approach with the cost of an increase in query times because it uses multiple trees with distinct sorting keys. Using multiple trees improves matching quality as it reduces the effect of having errors and variation at the beginning of attribute values that are used as sorting keys. The increase in query times, for the F-DySNI, is caused by the increased number of candidate records that are collected from multiple trees (unlike the DySNI that have only a single tree and less number of candidate records).

9.4 Blocking Key Learning Technique

In Chapter 3 we identified the need for new novel techniques that learn optimal blocking keys which can be used in the indexing step of real-time ER. We addressed this problem in Chapter 8 by proposing a general automatic blocking/sorting key selection algorithm that automatically selects blocking/sorting keys that are suitable for building indexes to be used with real-time ER. We evaluated the proposed learning algorithm in Chapter 8 by comparing it with an existing state-of-the-art blocking key learning algorithm [89]. The results from Chapter 8 showed that our approach can learn keys that are suitable for real-time ER and that the keys selected by our approach reduced query times significantly (around one order of magnitude) compared to the baseline [89] while maintaining matching quality.

In this section we compare our automatic key selection algorithm with manual key selection based on expert knowledge. The aim of this evaluation is to investigate whether or not our automatic technique can substitute manual key selection while achieving similar results. To conduct the comparison, we use the keys selected manually by an expert and the keys selected by our automatic selection algorithm (proposed in Chapter 8). We build an index using the keys selected automatically by our learning approach, and a second index using the manually selected keys. Then we conduct the real-time ER process on both built indexes and compare their effectiveness and efficiency. We used both the OZ-1 and Febrl-5 data sets in this experiment, where 50% of the records in the data sets are used to build two indexes and the remaining 50% are considered as query records. The similarity-based adaptive window approach (presented in Section 6.4.3) is used to generate candidate records using similarity thresholds that range between $0.5 \leq \theta \leq 1.0$.

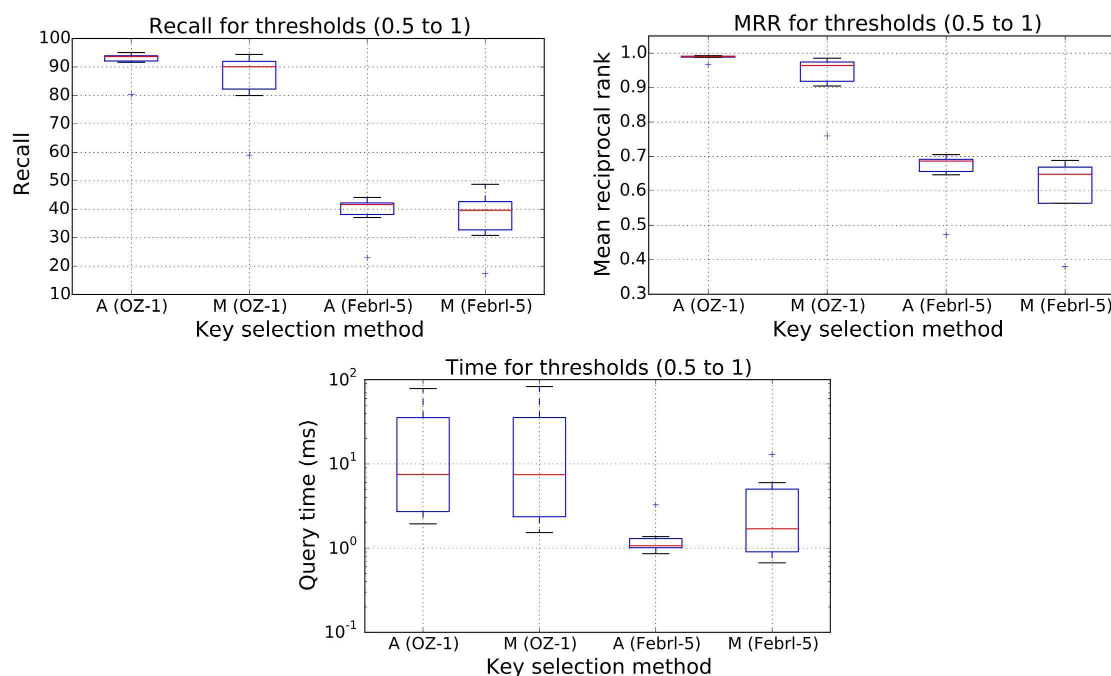


Figure 9.9: Comparative results between manual and automatic key selection for the OZ-1 and Febrl-5 data sets. (A) refers to automatic keys while (M) refers to manual keys. The F-DySNI technique is used with three trees. The similarity-based adaptive window approach is used to generate the candidate records, and similarity thresholds that range between 0.5 and 1.0 are used. The keys selected manually by an expert for both data sets include: 'Firstname+Surname', 'Surname+Firstname', and 'Surname+Postcode'. Manual keys were selected based on the results presented in Chapter 7 which confirmed that keys generated from concatenated attribute values are more suitable for real-time ER than single attribute values. The automatic keys (that were selected by our learning algorithm) for the OZ-1 data set are 'Firstname+Suburb', 'Surname+Suburb', and 'Surname+Postcode', and for the Febrl-5 data set they are 'Firstname+suburb', 'Surname+Suburb', and 'Surname+Suburb' (details on how automatic keys are selected are given in Chapter 8)

Figure 9.9 presents the recall, MRR, and query time values achieved by using both automatic keys (selected by our learning algorithm) and manual keys (selected by an expert) to build the indexes. Note that all other settings are exactly the same. The results show that, for the OZ-1 data set, the automatic keys selected by our learning algorithm slightly outperforms the keys selected manually with regard to recall and MRR while maintaining similar query times. The results also show that for the Febrl-5 data set the automatic keys selected by our learning algorithm slightly outperforms the keys selected manually with regard to MRR and average query time while achieving similar recall values.

The presented results confirm that our automatic blocking key selection algorithm can reduce human intervention by substituting manual key selection while maintaining the effectiveness and efficiency of the matching process.

9.5 Summary

In this chapter we compared our proposed indexing solutions together with a q-gram based indexing technique [11, 34, 107] that is used in ER, and we compared our automatic blocking key selection technique with manual key selection that is based on expert knowledge. The results presented in this chapter confirm that our proposed indexing solutions are suitable for real-time ER and that they outperform the q-gram based indexing technique. In addition, the results show that the F-DySNI with multiple trees achieves better recall and MRR values on all data sets compared to the other proposed solutions at the cost of a slight increase in query time. The reason behind this increase is that more trees lead to an increase in the number of candidate records to be compared with the query record. The DySimII did not perform as good as the other two solutions (with regard to matching quality) as it does not work well with noisy data sets that contains errors and variations. The F-DySNI technique achieves the best results with regard to matching quality on noisy data sets.

The results presented in this chapter also confirm that our blocking/sorting key selection algorithm can successfully learn blocking/sorting keys that are suitable for real-time ER. Our learning algorithm outperforms manual key selection (that selects keys based on expert knowledge) with regard to matching quality while maintaining similar efficiency results. The presented results confirm that our blocking/sorting key selection algorithm can substitute human intervention that is required for manually selecting blocking/sorting keys that are suitable for use with real-time ER.

The comparative evaluation of the different proposed techniques can be extended in the future by using other data sets with different characteristics, for instance, data sets with longer attribute values or data sets with attributes other than names and addresses) to investigate if the developed indexing techniques are suitable for different data sets. Moreover, additional indexing techniques from the literature, besides the QGI, could be used as baselines and compared with the proposed approaches. Another future evaluation extension is to experimentally investigate how the proposed indexing techniques would handle receiving multiple query records during a specific time interval (for instance one second interval).

Chapter 10

Conclusions and Future Directions

In previous chapters we described the entity resolution process in general, identified the problem of real-time entity resolution, where we need to match query records with large and dynamic data sets in real-time, and we proposed various solutions that address some of the challenges related with real-time entity resolution. In this chapter we conclude the work presented in this thesis and provide possible future research directions. Section 10.1 provides a summary of our conclusions. Sections 10.2 and 10.3 provide a summary of the contributions and future research directions. Finally, Section 10.4, provides our closing remarks.

10.1 Conclusions

The work presented in this thesis addresses the problem of real-time entity resolution (ER) and focuses on the indexing step in particular. We have proposed three dynamic indexing techniques that are tailored for real-time ER, and an automatic blocking/sorting key selection technique that learns blocking/sorting keys that can be used with real-time ER.

First, in Chapter 5, we proposed a dynamic similarity-aware inverted index (named DySimII) which is a blocking-based indexing technique that is updated whenever a new query record arrives to facilitate query matching in real-time. Then, in Chapter 6, we proposed a dynamic sorted neighborhood index (named DySNI), which is a tree-based indexing technique that is tailored for real-time ER. This technique uses static and adaptive window approaches to retrieve candidate records. Moreover, in Chapter 7 we proposed a forest-based sorted neighborhood index (named F-DySNI) which is a multi-tree technique that is based on DySNI and uses multiple distinct trees in the index data structure where each tree has a unique sorting key. Finally, in Chapter 8, we proposed an unsupervised learning algorithm that automatically selects optimal blocking/sorting keys for building indexes that are suitable for real-time ER.

We conducted an empirical evaluation to evaluate the proposed indexing solutions with regard to their efficiency and effectiveness. We used various real-world data sets with millions of records and synthetic data sets with different data characteristics. The results showed that, for the growing sizes of our indexing solutions, no

appreciable increase occurs in both record insertion and query times. Record insertion times were almost constant for the growing size of the index while query times increased sub-linearly. Our proposed indexing solutions are shown to be scalable with large data sets.

We compared our indexing solutions together with an existing q-gram based indexing technique [11, 34, 107] (labeled as QGI) that is used in ER. Our solutions outperformed the QGI with regard to their effectiveness and efficiency, and the QGI technique was shown to be not suitable for real-time ER because of the long time it requires to match query records. Amongst the proposed techniques the DySNI (which consists of only one tree) was the fastest and it achieved an average query time that is less than 3 ms for a data set with around 20 million records. F-DySNI, using two and three trees in the index, achieved an average query time of around 3, and 4 ms respectively, while the DySimII achieved an average of 161 ms for the same data set.

As for matching quality, the F-DySNI outperformed the DySNI and DySimII approaches on all data sets. Including more trees in the F-DySNI data structure leads to better matching quality with the cost of an increase in query times. This is because having more trees leads to an increase in the number of candidate records which then leads to an increase in the average query time. We aim to address the increase in query times, when using multiple trees in future work by building and querying the trees in the index data structure in parallel. The DySimII did not perform as good as the DySNI and F-DySNI techniques with regard to matching quality and did not work well with dirty data sets (that contain errors and variations). This problem is to be addressed in future work were we aim to investigate how to improve matching quality for dirty data sets in the ER process in general and for our proposed indexing solutions in specific.

Moreover, we compared our blocking key learning algorithm with an existing stat-of-the-art blocking key learning algorithm [89] and the results showed that our learning solution achieved an average query time that is up to two orders of magnitude faster than the baseline while maintaining similar matching quality. The results also showed that our algorithm can learn keys that are similar to those selected manually by an expert where our keys slightly outperformed manual blocking keys with regard to matching quality while maintaining similar query times.

Our proposed solutions are shown to be suitable for use with real-time ER. A summary of the major contributions presented in this thesis and future directions are described in the following sections.

10.2 Contribution Summary

This section provides a list of main contributions presented in this thesis:

1. **Several efficient indexing techniques that are tailored for real-time ER:** Based on the identified need for novel efficient indexing techniques that are suitable

for real-time ER we proposed three different dynamic indexing techniques that are specifically designed to work with real-time ER.

- **A similarity-aware inverted index:** We proposed a dynamic index that is based on the static inverted index from [35]. The aim of this technique is to provide a dynamic index that can be used with real-time ER. We also reduced the size of the index by proposing a frequency-filtered variation that only indexes the most frequent attribute values.
- **A sorted neighborhood index:** We proposed a tree-based dynamic index that is based on the sorted neighborhood method [80]. The aim of our technique is to provide a dynamic index that is suitable for query matching in real-time. We investigated using various fixed and adaptive window techniques for retrieving candidate records. We also proposed a variation that aims to improve query matching times by pre-calculating the similarities of attribute values between neighboring tree nodes.
- **A forest-based sorted neighborhood index:** We proposed a multi-tree dynamic indexing technique that aims to improve the quality of query matching by using multiple distinct trees in the index data structure. This technique reduces the effects of errors and variations at the beginning of attribute values (that are used as sorting keys) on matching quality. We investigated using different numbers of trees and different sorting keys with single attribute values and concatenation of multiple attribute values to examine which sorting keys are more suitable for real-time ER.
- **A conceptual analysis:** We conducted a theoretical analysis of all proposed indexing techniques with regard to estimating the expected number of record comparisons required by each technique. The aim of this analysis is to give users an insight into the expected run time required to match a query record with a data set of a certain size.

2. **A blocking key learning technique:** We proposed a general unsupervised learning technique that automatically selects optimal blocking/sorting keys based on three criteria: key coverage (the number of record pairs that evaluate to the same key value), generated block sizes (the number of records that are inserted into a block), and distribution of block sizes (the variance of the sizes of all generated blocks) to ensure that the selected keys can be used with real-time ER to provide matching results efficiently. Our technique can learn multiple blocking/sorting keys that can be used with any multi-pass indexing technique (where several indexes are built using different blocking/sorting keys). Note that our technique can also learn a single blocking/sorting key for single runs of indexing techniques.

3. **A comprehensive evaluation:** We conducted an empirical evaluation using multiple large real-world data sets and multiple synthetic data sets in terms of quality and efficiency. The aim of our empirical evaluation is to examine if

the proposed approaches are suitable for use with real-time ER. In addition, we conducted a comparative evaluation between the different proposed techniques and existing techniques that are currently used in ER.

10.3 Future Directions

The work presented in this thesis can be improved or extended in different ways as described in the following:

1. **Improve matching quality for data sets with dirty attribute values:** Dirty data sets can include errors, missing values, or inconsistent values [10, 82]. Errors and variations in attribute values are usually caused by data entry errors or changes in attribute values like changes in names and addresses. These errors and variations can reduce the matching quality and cause incorrect classification of record pairs [57]. Our experimental evaluation of the different proposed indexing techniques showed that the quality of real-time ER is affected by how dirty the queried data set is. Our results showed that the more dirty a queried data set is, the lower the matching quality will be. It is important to investigate how the quality of real-time query matching can be improved for data sets with dirty attribute values.

2. **Disk-based indexing techniques for real-time ER:** All proposed indexing techniques are memory-based techniques that require the index to reside in main memory during the matching process. Although main memory capabilities have increased substantially in recent years and can retain large indexes with millions of records, as shown in our experimental evaluation, it is always important to have disk-based options that allow indexing of very large data sets that do not fit into main memory. A possible research direction in the future is to investigate using a combination of a B+ tree (that is usually used as a desk-based indexing data structure) and the braided tree (**BRT**) proposed in Chapter 6 (which can be used as a memory-based index) where the full data set can be indexed on a hard desk using the B+ tree while the data required for matching a stream of query records can reside in main memory (while it is needed) using the DySNI or F-DySNI approaches as described in Chapters 6 and 7. This include investigating when, how, and what data can move from the B+ tree on the hard desk to the BRT in main memory to enable the matching process in real-time. Another research direction is to investigate enhancing the DySimII approach to work as a disk-based index that can be used with larger data sets. A possible approach is to investigate retaining the build indexes on disk(s) and only load into main memory block(s) that are similar to the matched query record.

3. **Parallelize the indexing and record pairs comparison steps:** The results presented in Chapters 7 and 9 confirm that using multiple trees with distinct sorting keys in the F-DySNI improves the quality of query matching with the cost

of an increase in query times. A possible solution to a avoid the increase in insertion and query times (that is caused by using multiple trees in the index data structure) is to investigate using parallelized environment to process query matching in real-time. The building phase, where the multiple trees are constructed, can be distributed over multiple processors since each tree is distinct and independent from other trees in the index data structure. In addition, querying the multiple trees in the index can be conducted independently using multiple processors where a processor can be responsible for generating a set of candidate recored from a single tree. The overall set of candidate records (which is generated from the union of candidate records returned from all single trees) can then be compared with the query record to be classified into matches and non-matches. Since the comparison step is computationally expensive it can also be parallelized using multi-threading or multi-processing to generate the final list of matching records. This process should improve average insertion and query times when using multiple trees in the index data structure.

4. **Investigate other steps in ER with regard to real-time ER:** Since fast and dynamic indexing techniques are required to enable conducting the ER process in real-time, our focus in this thesis was directed towards the indexing step of the real-time ER process. However, it is important to investigate how other steps of the ER process can be tailored to make the complete ER pipeline applicable for real-time matching. We can investigate which of the existing comparison and classification techniques are more suitable for use with real-time ER based on their complexity and speed (Simple and fast techniques are more suitable for real-time ER).

5. **A fully automated blocking/sorting key selection:** Our blocking key learning algorithm automatically selects blocking/sorting keys from a list of candidate blocking keys that is generated manually by an expert. Moreover, the algorithm uses several weights that can be adjusted by an expert to produce keys that are most suitable for real-time ER. A future research direction can be to investigate how to automatically identify candidate blocking keys based on the content of the data sets without human intervention. Different characteristics of data set attributes (like completeness, frequency, accuracy, and so on) can be used to automatically select candidate blocking keys. Another possible future research direction is to learn the weights that are used in our key selection algorithm to produce blocks with high quality and small size.

6. **Investigate multiple entity types:** The proposed indexing techniques in Chapters 5, 6, and 7 assume dealing with data sets that contain one entity type only. However, real-world data sets can have multiple entity types (for instance bibliographic data sets can have multiple entity types such as author, paper, and venue entities). For future work, an investigation on how to extend the proposed indexing techniques to handle multiple entity types can be conducted.

10.4 Closing Remarks

The work presented in this thesis provides an insight into the importance of ER in general, and into the role of the indexing step particularly in tolerating the real-time ER process. We believe that the proposed techniques can be used in real world scenarios where real-time ER is required.

References

1. E. D. Acheson et al. Medical record linkage. *Medical record linkage.*, 1967. (cited on page 11)

2. G. Adelson-Velskii and E. M. Landis. An information organization algorithm. In *Doklady Akademia Nauk SSSR*, volume 146, pages 263–266, 1962. (cited on pages 27 and 76)

3. N. Adly. Efficient record linkage using a double embedding scheme. In *DMIN*, pages 274–281, 2009. (cited on page 32)

4. A. Aizawa and K. Oyama. A fast linkage detection scheme for multi-source information integration. In *WIRI*, pages 30–39, Tokyo, 2005. (cited on page 37)

5. H. Aizenstein and L. Pitt. On the learnability of disjunctive normal form formulas. *Machine Learning*, 19(3):183–208, 1995. (cited on pages 41 and 43)

6. M. Aly, M. Munich, and P. Perona. Indexing in large scale image collections: Scaling properties and benchmark. In *Applications of Computer Vision (WACV), 2011 IEEE Workshop on*, pages 418–425. IEEE, 2011. (cited on page 28)

7. V. Athitsos, M. Potamias, P. Papapetrou, and G. Kollios. Nearest neighbor retrieval using distance-based hashing. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 327–336. IEEE, 2008. (cited on page 33)

8. S. Azman et al. Efficient identity matching using static pruning q-gram indexing approach. *Decision Support Systems*, 73:97–108, 2015. (cited on page 31)

9. R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999. (cited on page 28)

10. C. Batini and M. Scannapieco. *Data quality: Concepts, methodologies and techniques*. Data-Centric Systems and Applications. Springer, 2006. (cited on pages 1, 3, 12, 15, 16, and 156)

11. R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *ACM SIGKDD Workshop on Data Cleaning, Record Linkage and Object Consolidation*, pages 25–27, Washington DC, 2003. (cited on pages 8, 9, 20, 31, 48, 137, 139, 151, and 154)

12. R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th international conference on World Wide Web*, pages 131–140. ACM, 2007. (cited on page 28)

13. O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *The VLDB Journal The International Journal on Very Large Data Bases*, 18(1):255–276, 2009. (cited on page 33)

14. D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*, pages 1–9. IEEE, 2009. (cited on page 11)

15. I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *ACM Transactions on Knowledge Discovery from Data*, 1(1), 2007. (cited on page 22)

16. I. Bhattacharya and L. Getoor. Query-time entity resolution. *Journal of Artificial Intelligence Research*, 30:621–657, 2007. (cited on pages 22 and 39)

17. M. Bilenko, S. Basil, and M. Sahami. Adaptive product normalization: Using online learning for record linkage in comparison shopping. In *IEEE 5th International Conference on Data Mining (ICDM)*. IEEE, 2005. (cited on pages 14, 41, 43, and 50)

18. M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *IEEE ICDM*, Hong Kong, 2006. (cited on pages 41 and 42)

19. A. Bilke and F. Naumann. Schema matching using duplicates. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 69–80. IEEE, 2005. (cited on page 122)

20. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. (cited on pages 28 and 38)

21. R. Bolton and D. Hand. Statistical fraud detection: A review. *Statistical Science*, pages 235–249, 2002. (cited on page 14)

22. D. G. Brizan and A. U. Tansel. A. survey of entity resolution and record linkage methodologies. *Communications of the IIMA*, 6(3):5, 2015. (cited on page 12)

23. A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 327–336. ACM, 1998. (cited on page 33)

24. E. Brook, D. Rosman, and C. Holman. Public good through data linkage: measuring research outputs from the western australian data linkage system. *Australian and New Zealand Journal of Public Health*, 32(1):19–23, 2 2008. (cited on page 13)

25. E. Brook, D. Rosman, C. Holman, and B. Trutwein. Summary report: Research outputs project, WA data linkage unit (1995-2003). Technical report, Western Australian Data Linkage Unit, Department of Health, 2005. (cited on page 13)

26. Y. Cao, Z. Chen, J. Zhu, P. Yue, C.-Y. Lin, and Y. Yu. Leveraging unlabeled data to scale blocking for record linkage. In *IJCAI*, Barcelona, 2011. (cited on page 42)

27. R. D. Carr, S. Doddi, G. Konjevod, and M. V. Marathe. On the red-blue set cover problem. In *SODA*, pages 345–353. Citeseer, 2000. (cited on page 41)

28. S. Chaudhuri, B. Chen, V. Ganti, and R. Kaushik. Example-driven design of efficient record matching queries. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 327–338. VLDB Endowment, 2007. (cited on page 12)

29. S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference*, pages 5–5. IEEE, 2006. (cited on page 28)

30. H. Chen, W. Chung, J. Xu, G. Wang, Y. Qin, and M. Chau. Crime data mining: a general framework and some examples. *IEEE Computer*, 37(4):50–56, 2004. (cited on page 14)

31. P. Christen. A comparison of personal name matching: Techniques and practical issues. In *Workshop on Mining Complex Data, held at IEEE ICDM*, Hong Kong, 2006. (cited on pages 3, 15, 17, 19, and 20)

32. P. Christen. Towards parameter-free blocking for scalable record linkage. Technical Report TR-CS-07-03, The Australian National University, 2007. (cited on page 32)

33. P. Christen. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer, 2012. (cited on pages 1, 2, 4, 11, 12, 13, 15, 16, 17, 19, 20, 21, 22, 23, 29, 32, 34, 41, 47, 49, 53, 54, 55, 56, 66, 80, 86, 87, 103, 108, 127, 139, and 142)

34. P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Transactions on Knowledge and Data Engineering*, 2012. (cited on pages 8, 9, 12, 18, 28, 29, 48, 64, 65, 73, 91, 117, 137, 139, 151, and 154)

35. P. Christen, R. Gayler, and D. Hawking. Similarity-aware indexing for real-time entity resolution. In *ACM CIKM*, pages 1565–1568, Hong Kong, 2009. (cited on pages 4, 19, 39, 40, 53, 55, 56, 57, 58, 66, 68, and 155)

36. P. Christen and K. Goiser. Quality and complexity measures for data linkage and deduplication. In F. Guillet and H. Hamilton, editors, *Quality Measures in Data Mining*, volume 43 of *Studies in Computational Intelligence*, pages 127–151. Springer, 2007. (cited on pages 3, 17, 22, 23, 45, and 46)

37. P. Christen and A. Pudjijono. Accurate synthetic generation of realistic personal information. In *PAKDD, LNAI*, volume 5476, pages 507–514, Bangkok, Thailand, 2009. Springer. (cited on page 52)

38. C. E. H. Chua, R. H. Chiang, and E.-P. Lim. Instance-based attribute identification in database integration. *The VLDB Journal*, 12(3):228–243, 2003. (cited on page 11)

39. W. W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Workshop on Information Integration on the Web, held at IJCAI*, pages 73–78, Acapulco, 2003. (cited on page 20)

40. D. Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979. (cited on pages 27 and 77)

41. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 3nd edition, 2009. (cited on pages 27, 76, 77, and 109)

42. A. Das Sarma, A. Jain, A. Machanavajjhala, and P. Bohannon. An automatic blocking mechanism for large-scale de-duplication tasks. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1055–1064. ACM, 2012. (cited on pages 42 and 120)

43. T. De Vries, H. Ke, and S. Chawla. *Record Linkage in the Industry: Applications of an Improved Suffix Array Blocking Method*. School of Information Technologies, University of Sydney, 2009. (cited on page 37)

44. T. De Vries, H. Ke, S. Chawla, and P. Christen. Robust record linkage blocking using suffix arrays. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 305–314. ACM, 2009. (cited on page 37)

45. T. De Vries, H. Ke, S. Chawla, and P. Christen. Robust record linkage blocking using suffix arrays and bloom filters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 5(2):9, 2011. (cited on page 37)

46. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. (cited on pages 30, 33, 34, 36, 39, and 42)

47. Department of Innovation Industry, Science and Research. Investing in Australia's population health data linkage infrastructure consultation. Technical report, Department of Innovation, 2011. (cited on page 13)

48. D. Dey, V. Mookerjee, and D. Liu. Efficient techniques for online record linkage. *IEEE Transactions on Knowledge and Data Engineering*, 23(3):373–387, 2010. (cited on page 39)

49. X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *ACM SIGMOD*, pages 85–96, Baltimore, 2005. (cited on page 12)

50. X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity search for web services. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 372–383. VLDB Endowment, 2004. (cited on page 23)

51. X. L. Dong and D. Srivastava. Big data integration. In *ICDE*, pages 1245–1248, Brisbane, 2013. IEEE. (cited on page 2)

52. U. Draisbach and F. Naumann. A comparison and generalization of blocking and windowing algorithms for duplicate detection. In *Workshop on Quality in Databases, held at VLDB*, Lyon, 2009. (cited on page 37)

53. U. Draisbach and F. Naumann. A generalization of blocking and windowing algorithms for duplicate detection. In *Data and Knowledge Engineering (ICDKE), 2011 International Conference on*, pages 18–24. IEEE, 2011. (cited on page 37)

54. U. Draisbach, F. Naumann, S. Szott, and O. Wonneberg. Adaptive windows for duplicate detection. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1073–1083. IEEE, 2012. (cited on pages 35, 36, 74, 80, 83, 84, and 97)

55. R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification*. John Wiley & Sons, 2012. (cited on pages 43, 50, and 126)

56. V. Efthymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, and T. Palpanas. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. *IEEE Big Data*, 2015. (cited on page 39)

57. A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007. (cited on pages 1, 2, 11, 12, 16, 17, 23, 28, and 156)

58. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing a fast access method for dynamic files. *ACM Transactions on Database Systems (TODS)*, 4(3):315–344, 1979. (cited on page 27)

59. C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems (TOIS)*, 2(4):267–288, 1984. (cited on page 27)

60. C. Faloutsos and S. Christodoulakis. Description and performance analysis of signature file methods for office filing. *ACM Transactions on Information Systems (TOIS)*, 5(3):237–257, 1987. (cited on page 27)

61. W. Fan, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. *Journal of Data and Information Quality (JDIQ)*, 4(4):16, 2014. (cited on page 12)

62. I. Fellegi and A. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969. (cited on pages 11, 12, 28, 31, 39, and 117)

63. A. Ferro, R. Giugno, P. L. Puglisi, and A. Pulvirenti. An efficient duplicate record detection using q-grams array inverted index. In *Data Warehousing and Knowledge Discovery*, pages 309–323. Springer, 2010. (cited on page 31)

64. J. Fisher, P. Christen, Q. Wang, and E. Rahm. A clustering-based framework to control block sizes for entity resolution. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 279–288. ACM, 2015. (cited on page 29)

65. P. H. Giang. A machine learning approach to create blocking criteria for record linkage. *Health care management science*, 2014. (cited on pages 42, 43, and 121)

66. A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB Conference*, pages 518–529, Stanford, Canada, 1999. Stanford University. (cited on pages 2, 23, and 33)

67. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics, 2003. (cited on page 27)

68. L. Gu and R. A. Baxter. Adaptive filtering for efficient record linkage. In *SDM*, pages 477–481. SIAM, 2004. (cited on page 29)

69. H. Hajishirzi, W.-t. Yih, and A. Kolcz. Adaptive near-duplicate detection via similarity learning. In *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 419–426. ACM, 2010. (cited on page 14)

70. P. A. Hall and G. R. Dowling. Approximate string matching. *ACM Computing Surveys*, 12(4):381–402, 1980. (cited on page 20)

71. J. Han and M. Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann, 3 edition, 2011. (cited on pages 12, 21, and 22)

72. O. Hassanzadeh, F. Chiang, H. C. Lee, and R. J. Miller. Framework for evaluating clustering algorithms in duplicate detection. *PVLDB*, 2(1):1282–1293, 2009. (cited on page 101)

73. D. Haussler. *Probably approximately correct learning*. University of California, Santa Cruz, Computer Research Laboratory, 1990. (cited on page 43)

74. T. H. Haveliwala, A. Gionis, D. Klein, and P. Indyk. Evaluating strategies for similarity search on the web. In *Proceedings of the 11th international conference on World Wide Web*, pages 432–442. ACM, 2002. (cited on page 23)

75. D. Hawking, T. Rowlands, and M. Adcock. Improving rankings in small-scale web search using click implied descriptions. *Austr. J. Intelligent Information Processing Systems*, 9(2):17–24, 2006. (cited on page 46)

76. Q. He, Z. Li, and X. Zhang. Data deduplication techniques. In *Future Information Technology and Management Engineering (FITME), 2010 International Conference on*, volume 1, pages 430–433. IEEE, 2010. (cited on page 11)

77. J. W. Hector Garcia-Molina, Jeffrey D. Ullman. *Database systems: the complete book*. Pearson Prentice Hall, 2nd edition, 2009. (cited on page 27)

78. M. A. Hernández. A generalization of band joins and the merge-purge problem. 1995. (cited on page 11)

79. M. A. Hernandez and S. J. Stolfo. The merge/purge problem for large databases. In *ACM SIGMOD*, pages 127–138, San Jose, 1995. (cited on pages 4, 11, 18, 31, 34, 35, 36, 73, 74, 76, 103, and 117)

80. M. A. Hernandez and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998. (cited on pages 11, 34, 36, 73, 74, and 155)

81. M. Herschel, F. Naumann, S. Szott, and M. Taubert. Scalable iterative graph duplicate detection. *Knowledge and Data Engineering, IEEE Transactions on*, 24(11):2094 –2108, nov. 2012. (cited on pages 12 and 22)

82. T. Herzog, F. Scheuren, and W. Winkler. *Data quality and record linkage techniques*. Springer Verlag, 2007. (cited on pages 1, 3, 11, 12, 15, 16, 17, 19, 20, 29, and 156)

83. G. Howe and J. Lindsay. A generalized iterative record linkage computer system for use in medical follow-up studies. *Computers and Biomedical Research*, 14(4):327–340, 1981. (cited on page 11)

84. P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998. (cited on page 33)

85. E. Ioannou, W. Nejdl, C. Niederée, and Y. Velegrakis. On-the-fly entity-aware query processing in the presence of linkage. *Proceedings of the VLDB Endowment*, 3(1), 2010. (cited on pages 40 and 43)

86. E. Ioannou, O. Papapetrou, D. Skoutas, and W. Nejdl. Efficient semantic-aware detection of near duplicate resources. In *The Semantic Web: Research and Applications*, pages 136–150. Springer, 2010. (cited on page 33)

87. M. A. Jaro. Advances in record-linkage methodology a applied to matching the 1985 Census of Tampa, Florida. *Journal of the American Statistical Association*, 84:414–420, 1989. (cited on pages 20, 28, 29, 31, and 39)

88. L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. In *DASFAA*, pages 137–146, Tokyo, 2003. (cited on pages 32 and 33)

89. M. Kejriwal and D. P. Miranker. An unsupervised algorithm for learning blocking schemes. In *IEEE 13th International Conference on Data Mining (ICDM)*, pages 340–349. IEEE, 2013. (cited on pages 8, 9, 42, 43, 50, 120, 121, 122, 123, 124, 126, 149, and 154)

90. M. Kejriwal and D. P. Miranker. On the complexity of sorted neighborhood. *arXiv preprint arXiv:1501.01696*, 2015. (cited on page 34)

91. A. Kent, R. Sacks-Davis, and K. Ramamohanarao. A signature file scheme based on multiple organizations for indexing very large text databases. *Journal of the American Society for Information Science*, 41(7):508, 1990. (cited on page 27)

92. H. Kim and D. Lee. HARRA: fast iterative hashed record linkage for large-scale data collections. In *International Conference on Extending Database Technology*, pages 525–536, Lausanne, Switzerland, 2010. (cited on page 33)

93. L. Kolb, A. Thor, and E. Rahm. Dedoop: efficient deduplication with hadoop. *Proceedings of the VLDB Endowment*, 5(12):1878–1881, 2012. (cited on page 12)

94. L. Kolb, A. Thor, and E. Rahm. Multi-pass sorted neighborhood blocking with mapreduce. *Computer Science-Research and Development*, 27(1):45–63, 2012. (cited on pages 34 and 36)

95. L. Kolb, A. Thor, and E. Rahm. Don't match twice: redundancy-free similarity computation with mapreduce. In *Proceedings of the Second Workshop on Data Analytics in the Cloud*, pages 1–5. ACM, 2013. (cited on page 30)

96. H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *VLDB Endowment*, 3(1-2), 2010. (cited on pages 28 and 52)

97. S. Lafon, Y. Keller, and R. R. Coifman. Data fusion and multicue data matching by diffusion maps. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(11):1784–1797, 2006. (cited on page 11)

98. H.-S. Lee, M.-Y. Lee, and M.-J. Kim. Method and system for indexing and searching high-dimensional data using signature file, Oct. 4 2011. US Patent 8,032,534. (cited on page 27)

99. T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proc. VLDB*, Kyoto, 1986. (cited on page 27)

100. C. Li, L. Jin, and S. Mehrotra. Supporting efficient record linkage for large data sets using mapping techniques. *World Wide Web*, 9(4):557–584, 2006. (cited on page 33)

101. L. Li, J. Li, and H. Gao. Rule-based method for entity resolution. *Knowledge and Data Engineering, IEEE Transactions on*, 27(1):250–263, 2015. (cited on pages 12 and 21)

102. W. Litwin. Linear hashing: a new tool for file and table addressing. In *VLDB*, volume 80, pages 1–3, Montreal, 1980. (cited on page 27)

103. Y. Ma and T. Tran. Typimatch: Type-specific unsupervised learning of keys and key values for heterogeneous web data integration. In *ACM WSDM*, Rome, 2013. (cited on pages 42 and 43)

104. P. Malhotra, P. Agarwal, and G. Shroff. Graph-parallel entity resolution using lsh & imm. In *EDBT/ICDT Workshops*, pages 41–49, 2014. (cited on page 33)

105. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *The SIAM Journal on Computing*, 22(5):935–948, 1993. (cited on page 27)

106. C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008. (cited on page 28)

107. A. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178. ACM, 2000. (cited on pages 8, 9, 22, 32, 41, 48, 137, 139, 151, and 154)

108. R. A. McCallum, G. Tesauro, D. Touretzky, and T. Leen. Instance-based state identification for reinforcement learning. *Advances in Neural Information Processing Systems*, pages 377–384, 1995. (cited on page 11)

109. D. G. Mestre, C. E. Pires, and D. C. Nascimento. Adaptive sorted neighborhood blocking for entity matching with mapreduce. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 981–987, 2015. (cited on pages 35 and 36)

110. M. Michelson and C. A. Knoblock. Learning blocking schemes for record linkage. In *The Twenty-first National Conference on Artificial Intelligence (AAAI)*, Boston, 2006. (cited on pages 41, 42, 43, and 50)

111. T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997. (cited on pages 21 and 41)

112. A. D. Morris, D. I. Boyle, R. MacAlpine, A. Emslie-Smith, R. T. Jung, R. W. Newton, and T. M. MacDonald. The diabetes audit and research in tayside scotland (darts) study: electronic record linkage to create a diabetes register. *Bmj*, 315(7107):524–528, 1997. (cited on page 11)

113. F. Naumann and M. Herschel. *An Introduction to Duplicate Detection*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010. (cited on pages 1, 2, 12, 14, 16, 19, 20, 22, 23, 45, 46, and 48)

114. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001. (cited on pages 19 and 48)

115. H. Newcombe, J. Kennedy, S. Axford, and A. James. Automatic linkage of vital records. *Science*, 130(3381):954–959, 1959. (cited on pages 11 and 12)

116. Oxford University Press. Oxford english dictionary. http://www.oed.com/. Accessed online on 27/01/2013. (cited on page 14)

117. G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, and W. Nejdl. Eliminating the redundancy in blocking-based entity resolution methods. In *Proceedings of the 11th annual international ACM/IEEE joint conference on Digital libraries*, pages 85–94. ACM, 2011. (cited on page 30)

118. G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. Meta-blocking: Taking entity resolutionto the next level. *Knowledge and Data Engineering, IEEE Transactions on*, 26(8):1946–1960, 2014. (cited on pages 38 and 39)

119. G. Papadakis, G. Papastefanatos, and G. Koutrika. Supervised meta-blocking. *Proceedings of the VLDB Endowment*, 7(14):1929–1940, 2014. (cited on page 39)

120. T. Papenbrock, A. Heise, and F. Naumann. Progressive duplicate detection. *Knowledge and Data Engineering, IEEE Transactions on*, 27(5):1316–1329, 2015. (cited on pages 12 and 36)

121. C. Phua, V. Lee, K. Smith, and R. Gayler. A comprehensive survey of data mining-based fraud detection research. *arXiv preprint arXiv:1009.6119*, 2010. (cited on page 14)

122. J. J. Pollock and A. Zamora. Automatic spelling correction in scientific and scholarly text. *Communications of the ACM*, 27(4):358–368, 1984. (cited on page 20)

123. P. Ponniah. *Data Warehousing Fundamentals: A Comprehensive Guide for IT Professionals*. Paulraj Ponniah, 2001. (cited on page 12)

124. Population Health Research Network Program Office. Annual review 2010-2011. Technical report, 2011. (cited on page 13)

125. Population Health Research Network Program Office. Benefits of data linkage. http://www.phrn.org.au/about-us/data-linkage/benefits/, Jan 2013. Accessed on 27-January-2013]. (cited on page 13)

126. E. H. Porter and W. E. Winkler. Approximate string comparison and its effect on an advanced record linkage system. Technical Report RR97/02, US Bureau of the Census, 1997. (cited on page 14)

127. E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000. (cited on pages 12, 16, and 17)

128. B. Ramadan and P. Christen. Forest-based dynamic sorted neighborhood indexing for real-time entity resolution. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 1787–1790, 2014. (cited on pages 7 and 120)

129. B. Ramadan and P. Christen. Unsupervised blocking key selection for real-time entity resolution. In *Databases Theory and Applications*, page Pages to be added. Springer, 2015. (cited on page 8)

130. B. Ramadan, P. Christen, and H. Liang. Dynamic sorted neighborhood indexing for real-time entity resolution. In *Databases Theory and Applications*, pages 1–12. Springer, 2014. (cited on pages 7 and 120)

131. B. Ramadan, P. Christen, H. Liang, R. W. Gayler, and D. Hawking. Dynamic similarity-aware inverted indexing for real-time entity resolution. In *Trends and Applications in Knowledge Discovery and Data Mining*, pages 47–58. Springer, 2013. (cited on page 7)

132. Reserve Bank of Australia. C1 credit and charge card statistics. Online from the Reserve Bank of Australia, Jan 2013. Accessed on 29/01/2013. (cited on page 14)

133. E. K. Rezig, E. C. Dragut, M. Ouzzani, and A. K. Elmagarmid. Query-time record linkage and fusion over web databases. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 42–53. IEEE, 2015. (cited on page 40)

134. S. V. Rice. Braided AVL trees for efficient event sets and ranked sets in the simscript III simulation programming language. In *Western Multiconference on Computer Simulation*, pages 150–155, San Diego, 2007. Woodhead Publishing. (cited on page 76)

135. R. L. Rivest. Partial-match retrieval algorithms. *SIAM Journal on Computing*, 5(1):19–50, 1976. (cited on page 27)

136. F. Saıs, N. Pernelle, and M.-C. Rousset. L2r: A logical method for reference reconciliation. In *Proc. AAAI*, pages 329–334, 2007. (cited on page 12)

137. F. Saïs, N. Pernelle, and M.-C. Rousset. Combining a logical and a numerical method for data reconciliation. In *Journal on Data Semantics XII*, pages 66–94. Springer, 2009. (cited on page 12)

138. S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *ACM SIGKDD*, pages 269–278, Edmonton, 2002. (cited on page 12)

139. K.-U. Sattler. Data quality dimensions. In *Encyclopedia of Database Systems*, pages 612–615. Springer, 2009. (cited on page 16)

140. M. Scannapieco, I. Figotin, E. Bertino, and A. K. Elmagarmid. Privacy preserving schema and data matching. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 653–664. ACM, 2007. (cited on page 11)

141. R. Sedgewick and P. Flajolet. *An introduction to the analysis of algorithms*. Addison-Wesley, 2nd edition, 2013. (cited on page 76)

142. P. Singla and P. Domingos. Entity resolution with markov logic. In *Data Mining, 2006. ICDM'06. Sixth International Conference on*, pages 572–582. IEEE, 2006. (cited on page 12)

143. D. Song and J. Heflin. Automatically generating data linkages using a domain-independent candidate selection approach. In *The Semantic Web–ISWC 2011*, pages 649–664. Springer, 2011. (cited on pages 42 and 43)

144. M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller. Secure data deduplication. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 1–10, 2008. (cited on page 11)

145. J. Talburt. *Entity Resolution and Information Quality*. Morgan Kaufmann, 2011. (cited on page 11)

146. G. K. Tayi and D. P. Ballou. Examining data quality. *Communications of the ACM*, 41(2):54–57, 1998. (cited on page 16)

147. The Australian Bureau of Statistics. Information paper: Evaluation of administrative data sources for use in quarterly estimation of interstate migration. Technical Report 3127.0.55.001, 2002. (cited on page 14)

148. The Data Linkage Branch of the Department of Health. Data linkage in Western Australia. http://www.datalinkage-wa.org/, Jan 2013. Accessed on 23-January-2013. (cited on page 13)

149. K.-N. Tran, D. Vatsalan, and P. Christen. Geco: an online personal data generator and corruptor. In *ACM CIKM*, pages 2473–2476, San Francisco, 2013. (cited on page 51)

150. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. (cited on page 27)

151. V. S. Verykios, A. K. Elmagarmid, and E. N. Houstis. Automating the approximate record-matching process. *Information sciences*, 126(1):83–98, 2000. (cited on page 12)

152. T. Vogel and F. Naumann. Automatic blocking key selection for duplicate detection based on unigram combinations. In *VLDB workshops*, Istanbul, 2012. (cited on page 42)

153. G. A. Wang, H. Chen, J. J. Xu, and H. Atabakhsh. Automatically detecting criminal identity deception: an adaptive detection algorithm. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 36(5):988–999, 2006. (cited on page 31)

154. R. Y. Wang and D. M. Strong. Beyond accuracy: What data quality means to data consumers. *Journal of management information systems*, pages 5–33, 1996. (cited on page 16)

155. Y. R. Wang and S. E. Madnick. The inter-database instance identification problem in integrating autonomous systems. In *Data Engineering, 1989. Proceedings. Fifth International Conference on*, pages 46–55. IEEE, 1989. (cited on page 11)

156. P. Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973. (cited on page 27)

157. M. Weis, F. Naumann, U. Jehle, J. Lufter, and H. Schuster. Industry-scale duplicate detection. *Proceedings of the VLDB Endowment*, 1(2):1253–1264, 2008. (cited on page 21)

158. S. Whang and H. Garcia-Molina. Entity resolution with evolving rules. *Proceedings of the VLDB Endowment*, 3(1-2):1326–1337, 2010. (cited on page 40)

159. S. E. Whang, D. Marmaros, and H. Garcia-Molina. Pay-as-you-go entity resolution. *Knowledge and Data Engineering, IEEE Transactions on*, 25(5):1111–1124, 2013. (cited on page 40)

160. S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *ACM SIGMOD*, pages 219–232, Providence, Rhode Island, 2009. (cited on page 29)

161. W. Winkler. Frequency-based matching in the fellegi-sunter model of record linkage. In *Proceedings of the Section on Survey Research Methods, American Statistical Association*, volume 778, page 783, 1989. (cited on page 14)

162. W. Winkler. String comparator metrics and enhanced decision rules in the Fellegi-Sunter model of record linkage. In *Proceedings of the Section on Survey Research Methods*, pages 354–359. American Statistical Association, 1990. (cited on pages 20 and 83)

163. W. Winkler. Matching and record linkage. *Business survey methods*, 1:355–384, 1995. (cited on pages 13, 14, and 17)

164. W. Winkler. Approximate string comparator search strategies for very large administrative lists. *Statistics*, page 02, 2005. (cited on pages 14 and 43)

165. W. Winkler. Overview of record linkage and current research directions. In *Bureau of the Census*. Citeseer, 2006. (cited on page 11)

166. W. E. Winkler. Advanced methods for record linkage. 1994. (cited on page 11)

167. W. E. Winkler. The state of record linkage and current research problems. Technical report, US Census Bureau, Washington, DC,, 1999. (cited on page 12)

168. I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005. (cited on page 22)

169. C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, 36(3):15, 2011. (cited on pages 28 and 43)

170. S. Yan, D. Lee, M. Y. Kan, and L. C. Giles. Adaptive sorted neighborhood methods for efficient record linkage. In *ACM/IEEE-CS joint conference on Digital Libraries*, pages 185–194, 2007. (cited on pages 14, 35, 37, 74, 80, and 82)

171. W. E. Yancey. Evaluating string comparator performance for record linkage. Technical Report RR2005/05, US Bureau of the Census, 2005. (cited on page 20)

172. J. J. Zhu and L. H. Ungar. String edit analysis for merging databases. In *KDD workshop on text mining, held at ACM SIGKDD*, Boston, 2000. (cited on page 20)

173. G. K. Zipf. Human behavior and the principle of least effort. 1949. (cited on pages 61, 64, 66, and 109)

174. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys (CSUR)*, 38(2):6, 2006. (cited on page 28)