UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCES AND ENGINEERING

Entity Resolution in the Web of Data

by

Vasilis Efthymiou

PhD Dissertation

Presented

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

Heraklion, September 2017

UNIVERSITY OF CRETE

DEPARTMENT OF COMPUTER SCIENCE

Entity Resolution in the Web of Data

PhD Dissertation Presented

by **Vasilis Efthymiou**

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

APPROVED BY:

Author: Vasilis Efthymiou

Supervisor: Vassilis Christophides, Professor, University of Crete, Greece

---

Committee Member: Dimitris Plexousakis, Professor, University of Crete, Greece

---

Committee Member: Yannis Tzitzikas, Associate Professor, University of Crete, Greece

---

Committee Member: Yannis Velegrakis, Associate Professor, University of Trento, Italy

---

Committee Member: Manolis Koubarakis, Professor, National and Kapodistrian University of Athens, Greece

---

Committee Member: Fabian Suchanek, Professor, Télécom ParisTech University, France

---

Committee Member: Grigoris Antoniou, Professor, University of Huddersfield, UK

Department Chairman: Angelos Bilas, Professor, University of Crete, Greece

Heraklion, September 2017

Dedicated to my parents, Panagiotis and Magdalini, and my sister, Maria.

Acknowledgments

There are not enough words to express my deepest gratitude and respect to my PhD advisor, Professor Vassilis Christophides. His passion for research, his insight, and ability to always have the big picture, but above all, his personality and ethics, have been a true inspiration to me. I feel extremely lucky and honored to have met and worked with him.

I also thank the members of my advisory committee, Professor Dimitris Plexousakis, and Associate Professor Yannis Tzitzikas, for their valuable comments and suggestions all these years. I am also grateful to the other members of my examination committee, Associate Professor Yannis Velegrakis, Professor Manolis Koubarakis, Professor Fabian Suchanek, and Professor Grigoris Antoniou. Their constructive comments, showing their expertise in the field, helped me improve this thesis. Special thanks to Prof. Suchanek for providing any information that I have asked about the source code and datasets used in his research. This is the right way, if not the only way, for research to progress, but unfortunately his stance was the exception, rather than the rule.

I would like to deeply thank Associate Professor Kostas Stefanidis, the third member of our research team, for his great guidance and support. I owe him a lot and I feel that he has helped me improve in many aspects during our collaboration.

I would also like to express my sincere gratitude to all my co-authors for their help and support throughout our collaboration. This PhD would not be the same without their significant contribution. Especially, I would like to thank Professor Themis Palpanas for his valuable contribution in this PhD and in my career, as well as Dr. George Papadakis for the essential role that he played in our collaboration. I would also like to thank Dr. Oktie Hassanzadeh and Dr. Mariano Rodriguez-Muro from IBM Research for our great collaboration and for providing an ideal working environment during my internship.

A special mention is worth to Professor Timos Sellis for hosting me at RMIT, Melbourne. Seeing how much people within and outside the academia respect him both as a person and as a researcher has been inspiring. Meeting him in person has justified this level of respect.

I would like to acknowledge the support of the Institute of Computer Science of the Foundation of Research and Technology (ICS-FORTH), and especially the Information Systems Laboratory (ISL), for both the financial support and the facilities. In particular, I thank the Head of ISL, Professor Dimitris Plexousakis, as well as all the faculty and staff members of ISL, for creating a highly creative and inspiring environment for me. Furthermore, I would like to thankfully acknowledge the support of DIACHRON (EU FP7-ICT-2011-9), SemData (EU FP7-PEOPLE- 2013-IRSES), and IdeaGarden (EU FP7-ICT-318552). Part of my dissertation was supported and influenced by these projects.

During this PhD, I was fortunate to meet many exceptional and inspiring friends and colleagues: Kostas Petrakis, Dimitra Zografistou, Argyro Avgoustaki, Katerina Papantoniou, Pavlos Fafalios, Yannis Marketakis, Chrysostomos Zeginis, Ioannis Chrysakis, Thanos Yannakis, Nikos Minadakis, Giannis Roussakis, Georgia Troullinou, Konstantina Konsolaki, Vassilis Papakonstantinou, Aleka Seliniotaki, Mingzhao (Kane) Lee, Lance Song, Farhana Choudhury, Xiaolu Lu, Josep Pon Farreny, Siddharth Jain, Varun Chandramouli, Sanjay Surendranath Girija, Bilge Acun, Bogdan Ghit, Nicolas Fauceglia, Victor Han, Ioannis Petromichelakis, Ioannis Liodakis, and Giorgos Lyronis. Special thanks to Panagiotis Papadakos and Vangelis Kritsotakis for their valuable help in setting up and using the ISL cluster for the last part of this PhD. Last but not least, I would like to thank Myrto for her patience, love and support during the last four years.

This PhD is dedicated to my parents, Panagiotis and Magdalini, who have always stood by me and my sister in any possible way, teaching us that life is more about sharing than receiving, and to my sister, Maria, who has been a second mother for me and now a real mother of the most brilliant child I have ever met, my niece Marilia. I hope that one day I will become half as good a parent as they are.

Abstract

Entity resolution (ER) is the problem of identifying descriptions of the same real-world entities among or within knowledge bases (KBs). In this PhD thesis, we study the problem of ER in the Web of data, in which entities are described using graph-structured RDF data, following the principles of the Linked Data paradigm. The two core ER problems are: (a) how can we effectively compute similarity of Web entities, and (b) how can we efficiently resolve sets of entities within or across KBs. Compared to deduplication of entities described by tabular data, the new challenges for these problems stem from the *Variety* (i.e., multiple entity types and cross-domain descriptions), the *Volume* (i.e., thousands of Web KBs with billions of facts, hosting millions of entity descriptions) and *Veracity* (i.e., various forms of inconsistencies and errors) of entity descriptions published in the Web of data.

At the core of an ER task lies the process of deciding whether a given pair of descriptions refer to the same real-world entity i.e., if they *match* (problem a). The matching decision typically depends on the assessment of the similarity of two descriptions, based on their content or their neighborhood descriptions (i.e., of related entity types). This process is usually iterative, as matches found in one iteration help the decisions at the next iteration, via similarity propagation until no more matches are found. The number of iterations to converge clearly depends on the size and the complexity of the resolved entity collections. Moreover, pairwise entity matching is by nature quadratic to the number of entity descriptions, and thus prohibitive at the Web scale (problem b). In this respect, *blocking* aims to discard as many comparisons as possible without missing matches. It places entity descriptions into overlapping or disjoint blocks, leaving to the matching phase comparisons only between descriptions belonging to the same block. For this reason, overlapping blocking methods are accompanied by Meta-blocking filtering techniques, which aim to discard comparisons suggested by blocking that are either repeated (i.e., suggested by different blocks) or unnecessary (i.e., unlikely to result in matches) due to the noise in entity descriptions.

To address ER at the Web-scale, we need to relax a number of assumptions underlying several methods and techniques proposed in the context of database, machine learning and semantic Web communities. Overall, the Big Data characteristics of entity descriptions in the Web of data call for novel ER frameworks supporting: (i) *near similarity* (identify matches with low similarity in their content), (ii) *schema-free* (do not rely on a given set of attributes used by all descriptions), (iii) *no human in the loop* (do not rely on domain-experts for training data, aligned relations, matching rules), (iv) *non-iterative* (avoiding data-convergence methods at several iteration steps), and (v) *scalable* to very large volumes of entity collections (massively parallel architecture needed).

To satisfy the requirements of a Web-scale ER, we introduce the *MinoanER* framework. Our framework exploits new similarity metrics for assessing matching evidence based on both the content and the neighbors of entities, without requiring knowledge or alignment of the entity types. These metrics allow for a compact representation of similarity evidence that can be obtained from different blocking schemes on the names and values of the descriptions, but also on the values of their entity neighbors. This enables the identification of nearly similar matches even from the step of blocking. This composite blocking, accompanied by a novel composite Meta-blocking capturing the similarity evidence from the different types of blocks, set the ground for a non-iterative matching. The matching algorithm, built on a massively parallel architecture, is equipped with computationally cheap heuristics to detect matches in a fixed number of steps. The main contribution of MinoanER is that it achieves at least equivalent results over homogeneous KBs (stemming from common data sources, thus exhibiting strongly similar matches) and significantly better results over heterogeneous KBs (stemming from different sources, thus exhibiting many nearly similar matches) to state-of-the-art ER tools, without requiring any domain-specific knowledge, in a non-iterative and highly efficient way.

Keywords: Entity Resolution, Blocking, Meta-blocking, Linked Data, MinoanER

Supervisor: Vassilis Christophides
Professor
Computer Science Department
University of Crete

Περίληψη

Η ανάλυση οντοτήτων είναι το πρόβλημα της αναγνώρισης περιγραφών των ίδιων οντοτήτων του πραγματικού κόσμου ανάμεσα σε διαφορετικές βάσεις γνώσης. Σε αυτή τη διδακτορική εργασία, μελετάμε το πρόβλημα την ανάλυσης οντοτήτων στον Παγκόσμιο Ιστό των Δεδομένων, στον οποίο οι οντότητες περιγράφονται μέσω RDF γράφων, ακολουθώντας τις αρχές των Διασυνδεδεμένων Δεδομένων. Τα δύο κεντρικά προβλήματα της ανάλυσης οντοτήτων είναι: (α) πώς μπορούμε να υπολογίσουμε την ομοιότητα οντοτήτων αποτελεσματικά, και (β) πώς μπορούμε να αναλύσουμε σύνολα οντοτήτων εντός ή μεταξύ των βάσεων γνώσης αποδοτικά. Σε σχέση με την απαλοιφή διπλοτύπων περιγραφών οντοτήτων σε σχεσιακές βάσεις, οι νέες προκλήσεις για αυτά τα προβλήματα πηγάζουν από την Ποικιλία (πολλαπλοί τύποι οντοτήτων και διαθεματικές περιγραφές), τον Όγκο (χιλιάδες βάσεις γνώσης στον Παγκόσμιο Ιστό με δισεκατομμύρια γεγονότα, που φιλοξενούν εκατομμύρια περιγραφές οντοτήτων), και την Εγκυρότητα (πολλές μορφές ασυνέπειας και λαθών) των περιγραφών οντοτήτων που δημοσιεύονται στον Παγκόσμιο Ιστό των Δεδομένων.

Στον πυρήνα της ανάλυσης οντοτήτων βρίσκεται η διαδικασία λήψης της απόφασης για το αν ένα δοθέν ζευγάρι περιγραφών αναφέρονται στην ίδια πραγματική οντότητα, δηλαδή αν ταιριάζουν (πρόβλημα α). Η απόφαση ταιριάσματος συνήθως εξαρτάται από την εκτίμηση της ομοιότητας δύο περιγραφών, με βάση το περιεχόμενο ή ακόμα και τις γειτονικές τους περιγραφές (για οντότητες διαφορετικών τύπων). Αυτή η διαδικασία είναι συνήθως επαναληπτική, καθώς οι αποφάσεις ταιριάσματος σε μία επανάληψη βοηθούν στη λήψη αποφάσεων σε επόμενες επαναλήψεις, χρησιμοποιώντας διάδοση ομοιότητας, έως ότου να μην βρίσκονται άλλες περιγραφές που ταιριάζουν. Το πλήθος των απαιτούμενων για τη σύγκλιση επαναλήψεων εξαρτάται από το μέγεθος και την πολυπλοκότητα των συλλογών περιγραφών οντοτήτων. Επιπλέον, το ταίριασμα ζευγαριών περιγραφών είναι εκ φύσεως τετραγωνικής πολυπλοκότητας ως προς το πλήθος των περιγραφών και άρα απαγορευτικό στην κλίμακα του Παγκόσμιου Ιστού (πρόβλημα β). Στο πλαίσιο αυτό, η συσταδοποίηση έχει στόχο να αποτρέψει όσο το δυνατόν περισσότερες συγκρίσεις, χωρίς να χαθούν ταιριαστές περιγραφές. Τοποθετεί τις περιγραφές οντοτήτων σε επικαλυπτόμενες ή μη-επικαλυπτόμενες συστάδες, προωθώντας στη φάση ταιριάσματος τις συγκρίσεις μόνο μεταξύ περιγραφών που έχουν τοποθετηθεί σε κάποια κοινή συστάδα. Οι μέθοδοι επικαλυπτόμενης συσταδοποίησης συνοδεύονται από τεχνικές Μετα-συσταδοποίησης, που έχουν ως στόχο την αποτροπή των επαναλαμβανόμενων

συγκρίσεων που προτείνονται από πολλαπλές συστάδες, καθώς και των συγκρίσεων μεταξύ περιγραφών που είναι πιθανότερο να μην ταιριάζουν, αλλά έχουν προταθεί λόγω ύπαρξης θορύβου στις περιγραφές οντοτήτων.

Για να αντιμετωπίσουμε το πρόβλημα της ανάλυσης οντοτήτων στην κλίμακα του Παγκόσμιου Ιστού, χρειάζεται να χαλαρώσουμε ένα πλήθος υποθέσεων που υπόκεινται πολλών μεθόδων και τεχνικών, οι οποίες έχουν προταθεί στις ερευνητικές κοινότητες των βάσεων δεδομένων, της μηχανικής μάθησης και του σημασιολογικού Ιστού. Συνολικά, τα χαρακτηριστικά Μεγάλων Δεδομένων που εμφανίζουν οι περιγραφές οντοτήτων στον Παγκόσμιο Ιστό των Δεδομένων απαιτούν νέα συστήματα ανάλυσης οντοτήτων που να υποστηρίζουν: (i) σχεδόν ομοιότητα περιγραφών (αναγνωρίζουν περιγραφές που ταιριάζουν και έχουν χαμηλή ομοιότητα περιεχομένου), (ii) ανεξαρτησία ύπαρξης σχήματος (δεν στηρίζονται στην ύπαρξη ενός συγκεκριμένου συνόλου γνωρισμάτων που χρησιμοποιούνται από όλες τις περιγραφές), (iii) πλήρη αυτοματοποίηση (δεν στηρίζονται σε ειδικούς της εκάστοτε περιοχής για δεδομένα εκμάθησης, αντιστοίχιση σχέσεων, κανόνες ταιριάσματος), (iv) μη-επαναληπτικότητα (οι επαναληπτικές μέθοδοι συγκλίνουν μετά από υπερβολικά πολλές επαναλήψεις στον Παγκόσμιο Ιστό των Δεδομένων), και (v) κλιμακωσιμότητα σε πολύ μεγάλους όγκους δεδομένων (απαιτούνται μαζικά παραλληλοποιήσιμες αρχιτεκτονικές).

Για να ικανοποιήσουμε τις απαιτήσεις ανάλυσης οντοτήτων στην κλίμακα του Παγκόσμιου Ιστού, εισάγουμε το σύστημα MinoanER. Το σύστημά μας εκμεταλλεύεται νέες μετρικές ομοιότητας για την εκτίμηση των ενδείξεων ταιριάσματος τόσο από το περιεχόμενο όσο και από τις γειτονιές των περιγραφών, χωρίς να απαιτεί πρότερη γνώση ή αντιστοίχιση των τύπων των οντοτήτων. Αυτές οι μετρικές επιτρέπουν μια συμπαγή αναπαράσταση των ενδείξεων ομοιότητας που μπορούν να αποκτηθούν από διαφορετικά σχέδια συσταδοποίησης πάνω στα ονόματα και τις τιμές των περιγραφών, καθώς επίσης και στις τιμές των γειτονικών τους περιγραφών. Αυτό επιτρέπει την αναγνώριση σχεδόν όμοιων περιγραφών που ταιριάζουν νωρίς, από το βήμα της συσταδοποίησης. Η σύνθετη αυτή συσταδοποίηση, ακολουθούμενη από μία νέα σύνθετη Μετα-συσταδοποίηση που αποτυπώνει τις ενδείξεις ομοιότητα από διαφορετικού τύπου συστάδες, θέτουν τις βάσεις για ένα μη-επαναληπτικό ταίριασμα. Ο αλγόριθμος ταιριάσματος, σχεδιασμένος με μία μαζικά παράλληλη αρχιτεκτονική, χρησιμοποιεί υπολογιστικά φτηνές ευριστικές μεθόδους για να αναγνωρίσει περιγραφές που ταιριάζουν σε ένα προκαθορισμένο πλήθος βημάτων. Η κύρια συνεισφορά του MinoanER είναι ότι πετυχαίνει τουλάχιστον ισάξια αποτελέσματα σε ομοιογενείς βάσεις γνώσης (που έχουν κοινές πηγές και συνεπώς περιέχουν πολύ όμοιες περιγραφές οντοτήτων), και σημαντικά καλύτερα αποτελέσματα σε ανομοιογενείς βάσεις γνώσης (που έχουν διαφορετικές πηγές και συνεπώς περιέχουν λιγότερο όμοιες περιγραφές), σε σχέση με συστήματα αιχμής στην ανάλυση οντοτήτων, χωρίς να απαιτεί οποιαδήποτε γνώση ενός συγκεκριμένου πεδίου, με μη-επαναληπτικό και εξαιρετικά

αποδοτικό τρόπο.

Λέξεις κλειδιά: Ανάλυση Οντοτήτων, Συσταδοποίηση, Μετα-συσταδοποίηση, Διασυνδεδεμένα Δεδομένα, MinoanER

Επόπτης: Βασίλης Χριστοφίδης
Καθηγητής
Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Contents

Acknowledgments vii
Abstract ix
Περίληψη (Abstract in Greek) xi
Table of Contents xv
List of Figures xvii
List of Tables xix
1 Introduction 1
1.1 The Value of Entity Resolution 2
1.2 Entity Resolution Workflow 5
1.3 Requirements for a Web-scale Entity Resolution 7
1.4 Contributions and Outline 10
2 Blocking 13
2.1 Introduction 13
2.2 Formal Blocking Model 14
2.3 Related Work 18
2.3.1 Schema-based Blocking 18
2.3.2 Schema-free Blocking 20
2.4 Scaling Blocking Methods to Very Large Entity Collections 25
2.4.1 Token Blocking 25
2.4.2 Attribute Clustering Blocking 25
2.4.3 Prefix-Infix(-Suffix) Blocking 27
2.5 Benchmarking Content-based Blocking Methods in the Web of Data 28
2.5.1 Datasets 29
2.5.2 Quality Results 31
2.5.3 Performance Results 38
2.5.4 Lessons Learned 38
2.6 Conclusion 39
3 Meta-Blocking 41
3.1 Introduction 41
3.2 Formal Meta-blocking Model 44
3.3 Related Work 48
3.4 Scaling Meta-Blocking to Very Large Entity Collections 50
3.4.1 Approach Overview 50

3.4.2 Stage 1: Block Filtering 51
3.4.3 Stage 2: Preprocessing 52
3.4.4 Stage 3: Pruning (WNP) 57
3.5 Load Balancing 60
3.5.1 Default Load Balancing 61
3.5.2 MaxBlock Load Balancing 61
3.6 Experiments 64
3.6.1 Setup 65
3.6.2 Load Balancing 66
3.6.3 Time Efficiency 70
3.6.4 Scalability 74
3.6.5 Qualitative Results 75
3.6.6 Discussion 76
3.7 Conclusion 77
4 Entity Matching 79
4.1 Introduction 79
4.2 Related Work 81
4.3 Basic Definitions 83
4.3.1 Entity similarity based on values 84
4.3.2 Entity similarity based on neighbors 86
4.4 Blocking 88
4.4.1 Composite Blocking Scheme 88
4.4.2 Disjunctive Blocking Graph 88
4.4.3 Graph Weighting and Pruning Algorithms 89
4.5 Non-Iterative Matching 91
4.5.1 Implementation in Spark 94
4.6 Experimental Evaluation 94
4.6.1 Effectiveness Evaluation 98
4.6.2 Efficiency Evaluation 101
4.7 Conclusion 102
5 Conclusion and Future Work 105
5.1 Synopsis of Contributions 105
5.2 Short-term Improvements 106
5.3 Directions for Future Research 107
Bibliography 111
Glossary 119

List of Figures

1.1 A part of the Web of data from two KBs: DBpedia (blue) and Freebase (red). Each table corresponds to an entity description, each header row to the URI of the described entity, and each other row to an attribute (left)- value (right) pair. 3
1.2 Searching for the entity "Stanley Kubrick" in the Web of data. 4
1.3 Outline of the entity resolution process. 6
1.4 Value and neighbor similarity distribution of matching entities in 4 real dataset. . . . 8

2.1 A set of entity description. 21
2.2 Token blocking example. Descriptions having a common token are placed in a common block. 21
2.3 Attribute clustering blocking example. Pairs of most similar attributes are linked (a). Connected attributes form clusters (b). Descriptions with a common token in the values of attributes of the same cluster, are placed in a common block (c). 22
2.4 Prefix-infix(-suffix) blocking example. A set of descriptions (a), their subject URIs (b), and the blocks from their tokens and infixes (c). 23
2.5 Token blocking in MapReduce. 26
2.6 Attribute clustering blocking in MapReduce. 26
2.7 Prefix-infix(-suffix) blocking in MapReduce. 27
2.8 Common tokens (top) and common tokens in common clusters (bottom) per entity description distributions for $D1$-$D7$. 33

3.1 (a) A set of heterogeneous entity description, (b) the overlap-positive block collection derived from them using token blocking, (c) the respective blocking graph that uses Jaccard similarity for edge weights, (d) one of the possible pruned blocking graphs, and (e) the restructured block collection after Meta-blocking. 42
3.2 (a) The serialized workflow of Meta-blocking, and (b) its parallelized counterpart. . . 50
3.3 Pseudo-code interpretation of (a) the basic and (b) the advanced strategy for Block Filtering. They employ a global and a local ordering of blocks, respectively. 51
3.4 An example of the advanced strategy for Block Filtering. 53
3.5 Pseudo-code interpretation of the edge-based Preprocessing strategy, which explicitly creates the blocking graph. 54
3.6 Pseudo-code interpretation of the edge-based Preprocessing strategy for the EJS weighting scheme. 55

3.7 Pseudo-code interpretation of the comparison-based Preprocessing strategy, which creates the blocking graph implicitly, enriching the description of the input block with the necessary information for weight estimation. 56
3.8 An example of the comparison-based strategy for Preprocessing. 56
3.9 Pseudo-code interpretation of the entity-based strategy for Preprocessing, which does not use the blocking graph. 57
3.10 Pseudo-code interpretation of (a) the edge-based and (b) the comparison-based strategy for WNP. They share the same `reduce` function. 58
3.11 An example of the comparison-based strategy for WNP, using the JS weighting scheme. 59
3.12 Pseudo-code interpretation of the entity-based strategy for WNP. 60
3.13 An example of running MaxBlock for load balancing. 63
3.14 Overhead Time in minutes for all configurations of the serialized workflow over DB_C. 73
3.15 Speedup over DB_C of (a) the comparison-based strategy for WEP, and (b) the entity-based strategy for CNP. 74
3.16 Average performance of the four pruning algorithms with respect to (a) Recall, (b) RR, (c) Precision, and (d) $H3R$. 75

4.1 (a) Parts of entity graphs, representing the Wikidata (left) and DBpedia (right) KBs, (b) parts of the corresponding disjunctive blocking graph, and (c) the corresponding graph after pruning. 80
4.2 An example of running our heuristics on a pruned disjunctive blocking graph. 95
4.3 The architecture of MinoanER in Spark. 96
4.4 Scalability of matching in MinoanER w.r.t. running time (left vertical axis) and speedup (right vertical axis) as more cores are involved. 102
4.5 The area of matches from Figure 1.4 targeted by each of the employed heuristics. . . 103

List of Tables

2.1 Quality Measures. 17
2.2 Co-occurrence functions for considering two descriptions candidate match. 24
2.3 Blocking methods with respect to the redundancy attitude and algorithmic attitude. 24
2.4 KBs characteristics. 29
2.5 Datasets characteristics. 31
2.6 Statistics and evaluation of blocking methods. 32
2.7 Characteristics of the missed match of token blocking. 36
2.8 Analysis of 1K sampled match and 1K sampled non-match. 37

3.1 Summary of the notation used in Meta-blocking. 45
3.2 The datasets employed in our experiments. 65
3.3 The block collections that were given as input to Meta-blocking. 66
3.4 The distribution of partition cardinalities produced by the default load balancer of Hadoop, PairRange and MaxBlock. 67
3.5 The wall-clock time (in minutes) of Meta-blocking using the default Hadoop balancer, the two variations of PairRange, and MaxBlock for the entity-based strategy over DB_C, using the CBS weighting scheme across all pruning algorithms. The overhead of executing each load balancing algorithm, compared to the default balancing, is common for all pruning algorithms and is included in the wall-clock times. 69
3.6 The block collections after Block Filtering. 70
3.7 Overhead Time ($OTime$) in minutes for all Meta-blocking techniques across the four real datasets. 71

4.1 KB statistics. 97
4.2 Block statistics. 98
4.3 Evaluation of MinoanER compared to existing methods. 98
4.4 Evaluation of heuristics. 101

Chapter 1
Introduction

An increasing number of government organizations, local bodies, private companies, scientific or citizen communities are currently describing a great variety of real-world *entities* (e.g., persons, places, products, events) as *Linked Data*[1], in the form of RDF triples[2], i.e., subject-predicate-object facts. The emerging *Web of data* aims to support a global data infrastructure, in which real-world entities are described on the Web by data rather than documents. Exhibiting a higher degree of interoperability than documents and ease of reuse both by humans and machines, Linked Data emerges as a prominent paradigm for publishing structured information worldwide.

Comprehensive, machine-readable *entity descriptions* are hosted in *Knowledge Bases* (KBs). Traditionally, KBs are manually crafted by a dedicated team of knowledge engineers (e.g., Wordnet[3] and Cyc[4]); with the explosion of the Web, however, more and more KBs are built from existing Web content using information extraction tools [25]. Such an automated approach offers an unprecedented opportunity to scale-up KB construction and leverage existing knowledge published in HTML documents [50], but it also comes at the cost of a significant degree of *redundancy* in the descriptions provided across domains for the same real-world entities. Such KBs may contain complementary and sometimes conflicting information regarding the same entity, which could be combined in order to provide a more complete picture of the described entities than each individual KB offers and be exploited by a multitude of applications. A prerequisite for merging complementary information or repairing contradicting information is to identify first the descriptions that refer to the same real-world entity (called *matches*). This is the problem of *entity resolution* (ER), on which we focus in this work. This clearly requires an understanding of the similarity among described entities that goes beyond *strong similarity* studied in traditional deduplication and cleaning problems [76]. We essentially need to explore entity descriptions that are *nearly similar* [88], since those descriptions have been created by various extraction tools of different quality, focusing on different aspects of the entities.

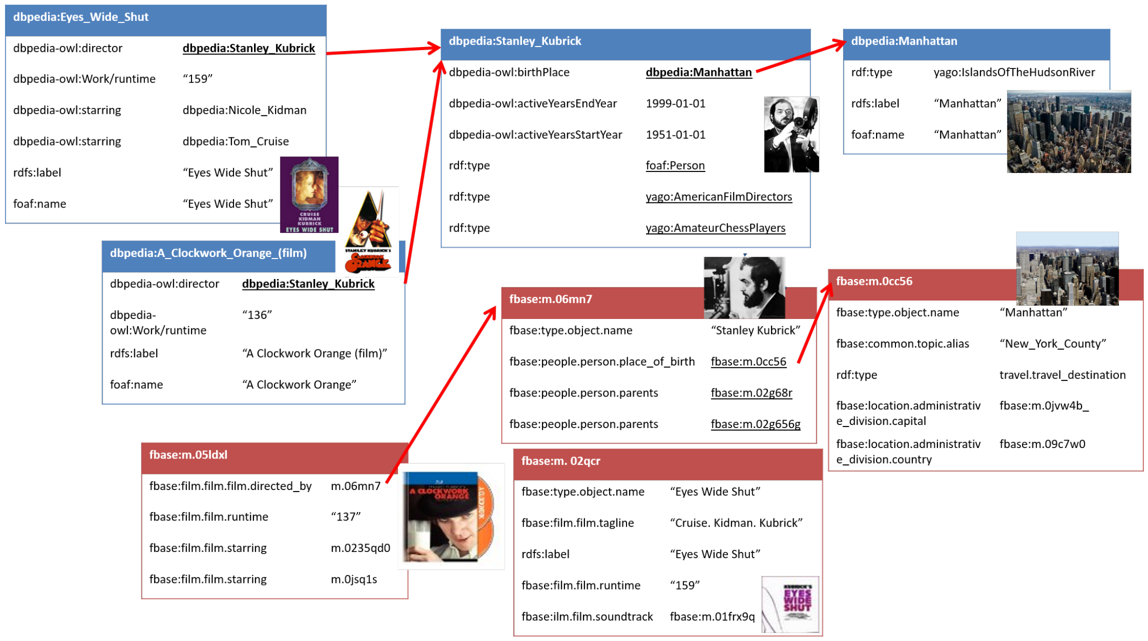
[1]http://linkeddata.org/
[2]http://www.w3.org/RDF
[3]http://wordnet.princeton.edu
[4]http://www.cyc.com

Example 1.1. *Consider the entity descriptions presented in Figure 1.1. An entity description in the Web of data is an identifiable set of attribute-value pairs. In this example, the entity identifiers are given in the header rows and the attribute-value pairs are the remaining rows (attributes left, values right). The example shows entity descriptions hosted in two KBs: DBpedia (blue) and Freebase (red). DBpedia describes two movies,* Eyes Wide Shut *and* A Clockwork Orange*, their director* Stanley Kubrick *and his place of birth* Manhattan*, while Freebase provides alternative descriptions for the same four entities. We say that two descriptions (e.g., Stanley Kubrick and A Clockwork Orange) that are linked through such relations (e.g., director) are* entity neighbors *(see red edges). The sets of attribute-value pairs used for describing these entities essentially group per subject URI (i.e., identifier) a collection of RDF triples. For example the fact that Stanley Kubrick is the director of the movie Eyes Wide Shut, expressed in the first row of the first entity description in this Figure, is expressed in RDF (in N-triples format) as the triple: "<dbpedia:Eyes_Wide_Shut> <dbpedia-owl:director> <dbpedia:Stanley_Kubrick> .", where "dbpedia:" is short for* `http://dbpedia.org/resource/` *and "dbpedia-owl:" is short for* `http://dbpedia.org/ontology/`*. Each triple expresses a fact about an entity, while triples having the attribute rdf:type express the semantic types of an entity. In this example, dbpedia:Stanley_Kubrick is declared to belong to the types Person, AmericanFilmDirectors, and AmateurChessPlayers. Such type declarations do not impose the use of a specific set of attributes in the Web of data, for entities of specific types. Note that different KBs may provide different (complementary or conflicting) facts regarding the same entity. E.g., Freebase states that the runtime of A Clockwork Orange is 137 minutes, while DBpedia suggests 136.*

One can notice that, in our example, there exist both strongly similar and nearly similar descriptions. For example, we can say that the descriptions referring to Eyes Wide Shut *are* strongly similar*, since they have very similar values for semantically equivalent attributes (e.g., name, runtime, cast). However, this is not the case with the descriptions that refer to* Stanley Kubrick*; those descriptions do not use any common words in their descriptions, while their attributes mostly refer to different aspects of this entity (birthplace, active years and semantic types, versus name, birthplace and parents). Hence those descriptions are more heterogeneous and in order to check if they match, we can additionally exploit the similarity of their entity neighbors (such as birthplace and films directed by them). For such descriptions, we can say that they are only* nearly similar.

1.1 The Value of Entity Resolution

To allow better understanding a user's intents, an entity-centric Web infrastructure enables powerful new user experiences, from search results that directly show key facts about people, places and things, to improved refinement interfaces that allow searchers to quickly locate Web documents that mention only the specific people, places or other things they are looking for [55]. We are witnessing a new generation of Web applications that rely on entity descriptions to better serve navigational or information seeking needs of users, namely, *entity-centric search* [7, 8, 15, 69] and *recommendations* [14, 75, 105]. The former semantically enrich the answers of keyword queries



dbpedia:Eyes_Wide_Shut	
dbpedia-owl:director	dbpedia:Stanley_Kubrick
dbpedia-owl:Work/runtime	"159"
dbpedia-owl:starring	dbpedia:Nicole_Kidman
dbpedia-owl:starring	dbpedia:Tom_Cruise
rdfs:label	"Eyes Wide Shut"
foaf:name	"Eyes Wide Shut"

dbpedia:A_Clockwork_Orange_(film)	
dbpedia-owl:director	dbpedia:Stanley_Kubrick
dbpedia-owl:Work/runtime	"136"
rdfs:label	"A Clockwork Orange (film)"
foaf:name	"A Clockwork Orange"

dbpedia:Stanley_Kubrick	
dbpedia-owl:birthPlace	dbpedia:Manhattan
dbpedia-owl:activeYearsEndYear	1999-01-01
dbpedia-owl:activeYearsStartYear	1951-01-01
rdf:type	foaf:Person
rdf:type	yago:AmericanFilmDirectors
rdf:type	yago:AmateurChessPlayers

dbpedia:Manhattan	
rdf:type	yago:IslandsOfTheHudsonRiver
rdfs:label	"Manhattan"
foaf:name	"Manhattan"

fbase:m.05ldxl	
fbase:film.film.film.directed_by	m.06mn7
fbase:film.film.runtime	"137"
fbase:film.film.starring	m.0235qd0
fbase:film.film.starring	m.0jsq1s

fbase:m.06mn7	
fbase:type.object.name	"Stanley Kubrick"
fbase:people.person.place_of_birth	fbase:m.0cc56
fbase:people.person.parents	fbase:m.02g68r
fbase:people.person.parents	fbase:m.02g656g

fbase:m. 02qcr	
fbase:type.object.name	"Eyes Wide Shut"
fbase:film.film.tagline	"Cruise. Kidman. Kubrick"
rdfs:label	"Eyes Wide Shut"
fbase:film.film.runtime	"159"
fbase:ilm.film.soundtrack	fbase:m.01frx9q

fbase:m.0cc56	
fbase:type.object.name	"Manhattan"
fbase:common.topic.alias	"New_York_County"
rdf:type	travel.travel_destination
fbase:location.administrative_division.capital	fbase:m.0jvw4b_
fbase:location.administrative_division.country	fbase:m.09c7w0

Figure 1.1: A part of the Web of data from two KBs: DBpedia (blue) and Freebase (red). Each table corresponds to an entity description, each header row to the URI of the described entity, and each other row to an attribute (left)- value (right) pair.

with references to entities that are mentioned in the queries[5], while the latter also provides recommendations of related entities based on relationships explicitly encoded in a KB [41]. Popular use-cases of such Web applications include Google's search that exploits the ER results of Knowledge Vault [26], and Microsoft's recommender system based on entity search results [11].

Example 1.2. *Consider the query "Stanley Kubrick", as shown in Figure 1.2. The user would probably like to know information about* Stanley Kubrick, *such as his age, birth place and profession, instead of being given a list of relevant documents that, combined, contain this information, or, potentially irrelevant documents that just contain these keywords. To serve this query in the Web of data, the following process would be engaged.*

Initially, a number of entity descriptions related to the entertainment industry (e.g., film makers) have been extracted from semantic annotations of Web pages and/or from domain specific KBs (e.g., LinkedMDB[6]) and cross-domain KBs (e.g., DBpedia, YAGO, Freebase). Such descriptions can be matched and matching descriptions can be linked to each other. Then, the mentions of various entities in the user queries are recognized and matched to the extracted entity descriptions. For example, besides Web documents related to "Stanley Kubrick", an entity search system would enrich

[5]A process known as named-entity extraction [13, 38, 39, 49] and disambiguation [58].
[6]www.linkedmdb.org

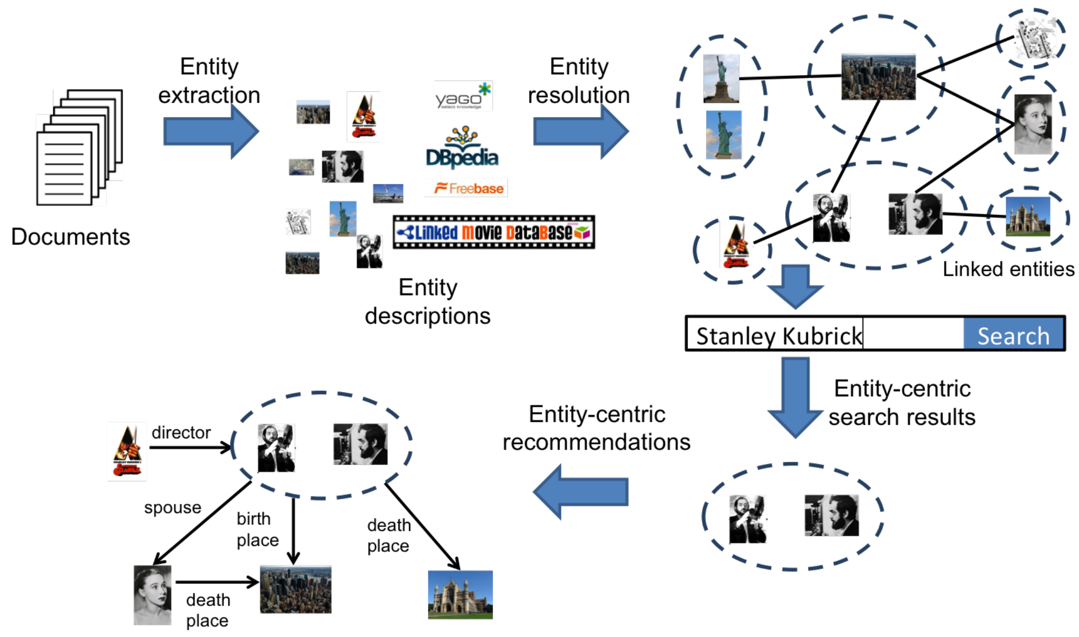


Figure 1.2: Searching for the entity "Stanley Kubrick" in the Web of data.

the answer with the descriptions of Stanley Kubrick in DBpedia and/or Freebase. To serve users willing to extend their knowledge or simply satisfy their curiosity, an entity recommender system could provide additional entities describing information of potential interest for the user. For example, consider the information that Kubrick was born in Manhattan*, extracted from DBpedia, that he was married to Ruth Sobotka, extracted from YAGO, that he was the director of the movie* A Clockwork Orange*, extracted from LinkedMDB, and so forth.*

Given the open and decentralized nature of the Web of data, reliability and usability of entity descriptions need to be constantly improved. Specifically, entity descriptions published in the Web of data can be *incomplete*, i.e., only partially described in KBs, *redundant*, i.e., descriptions of the same real-world entities usually overlap in multiple KBs, *inconsistent*, i.e., real-world entities may have conflicting descriptions across KBs, and *incorrect*, since errors can be propagated from one KB to the other due to manual copying or automated extraction techniques. In this respect, ER improves the quality of Web KBs in terms of *completeness*, since linking nearly similar descriptions will increase coverage of entity facts and relationships, *conciseness*, since merging strongly similar descriptions will reduce duplicate entity facts and relationships, *consistency*, since matching similar descriptions will enable to detect conflicting assertions, and *correctness*, since splitting complex descriptions will facilitate entity repairing. In this work, we will focus on the first two of those issues.

1.2 Entity Resolution Workflow

The general processing steps involved in an ER task are illustrated in Figure 1.3 [35, 95, 96]. The two core ER problems are (a) *how can we effectively compute similarity of Web entities*, and (b) *how can we efficiently resolve sets of entities within or across KBs*.

Regarding problem (a), at the core of an ER task lies the process of making the *matching* decision: for a given pair of descriptions, decide if they refer to the same real-world entity (i.e., if they match). This process aims to place matches at the same partition of the input entity collection $\mathcal{E}$, and all the descriptions placed into the same partition should match. Specifically, the matching decision is typically made by a match function M, mapping each pair of entity descriptions (e_i, e_j) to $\{true, false\}$, with $M(e_i, e_j) = true$ meaning that e_i and e_j are matches, and $M(e_i, e_j) = false$ meaning that e_i and e_j are not matches.

The match function M introduces an equivalence relation among entity descriptions, so it should satisfy the following properties:

- Reflexivity: $\forall e_i \in \mathcal{E}, M(e_i, e_i) = true$,
- Symmetry: $\forall e_i, e_j \in \mathcal{E}, M(e_i, e_j) = M(e_j, e_i)$, and
- Transitivity: $\forall e_i, e_j, e_k \in \mathcal{E}, (M(e_i, e_j) = true) \wedge (M(e_j, e_k) = true) \Rightarrow (M(e_i, e_k) = true)$.

In practice, the match function is defined via a similarity function sim, measuring how similar two entity descriptions are to each other, according to certain comparison criteria. Given a similarity threshold θ:

$$M(e_i, e_j) = \begin{cases} \text{true, if } sim(e_i, e_j) \geq \theta, \\ \text{false, otherwise.} \end{cases}$$

To support the identification of nearly similar matches, existing works perform more than a simple similarity computation on the values of two descriptions; they propagate the similarity of the entity neighbors of two descriptions to the similarity of those descriptions. In this inherently iterative process, the employed match function is based on a similarity that dynamically changes from iteration to iteration, and its results include a third state, the *uncertain* one. Specifically, given two similarity thresholds θ and θ', with $\theta' \leq \theta$, the match function at iteration n is given by:

$$M^n(e_i, e_j) = \begin{cases} \text{true, if } sim^{n-1}(e_i, e_j) \geq \theta, \\ \text{false, if } sim^{n-1}(e_i, e_j) \leq \theta', \\ \text{uncertain, otherwise.} \end{cases}$$

It should be clear from Example 1.1 that finding a similarity function which can perfectly distinguish all matches from non-matches for all entity collections is impossible. Thus, in reality, we seek a similarity function that will be only good enough, i.e., minimize the number of misclassified pairs.

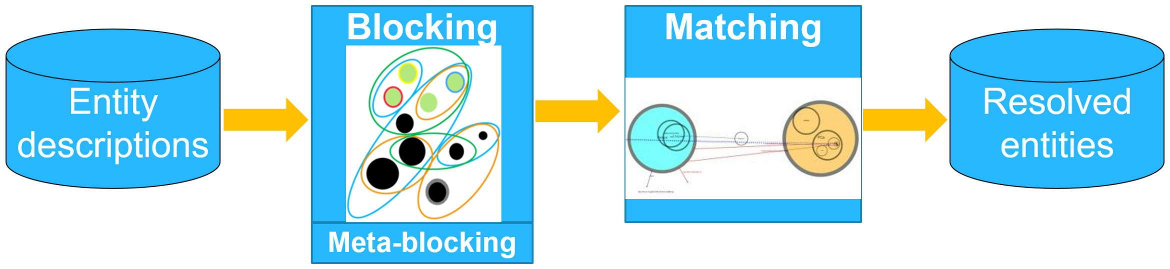


Figure 1.3: Outline of the entity resolution process.

Regarding (b), pairwise entity matching is by nature quadratic to the number of entity descriptions, and thus prohibitive at the Web scale. In this respect, *blocking* aims to discard as many comparisons as possible without missing comparisons that could result into a match. It places similar entity descriptions into blocks, leaving to the matching phase comparisons only between descriptions within the same block, based on some criteria (called *blocking keys*). Specifically, given an entity collection $\mathcal{E}$, blocking creates overlapping or disjoint partitions $B = \{b_1, b_2, \ldots, b_n\}$ of $\mathcal{E}$, called blocks, for which it holds that $\bigcup\limits_{b_i \in B} b_i = \mathcal{E}$. A blocking method is called a partitioning (or disjoint) blocking when $\forall b_i, b_j \in B, b_i \cap b_j = \emptyset$, and overlapping blocking, else. The goal of blocking is to quickly split the input entity collection into blocks that are as close as possible to the final matching results. Hence, following the definition of the match function M, which relies on a similarity function sim, the goal of blocking is for each pair of descriptions e_i, e_j that belong to the same block, it should hold that $sim(e_i, e_j) \geq \theta$.

Overlapping blocking methods are usually accompanied by *Meta-blocking*, which aims to discard comparisons suggested by blocking that are repeated across different blocks, as well as comparisons that are unlikely to result in matches, suggested due to noise in entity descriptions. The core idea for Meta-blocking is that the number and size of blocks that two descriptions share provide matching evidence: the more common blocks two descriptions share, the more similar those descriptions are, while, the smallest the common blocks (i.e., the fewer the descriptions placed in those blocks), the more discriminating they are, thus increasing the matching likelihood for the descriptions that share them. This matching evidence is represented in the form of a *blocking graph*, in which nodes correspond to entity descriptions and edges connect descriptions that co-occur in at least one common block. The weights of the edges, extracted entirely from block statistics, represent the likelihood that connected descriptions match, i.e., how strong the matching evidence for those descriptions is considered to be.

1.3 Requirements for a Web-scale Entity Resolution

ER is challenged by the *Variety*, *Volume* and *Veracity* of the Web of data, across all the steps of the ER workflow.

- *Variety* is mainly due to the descriptive, rather than prescriptive usage of ontologies/vocabularies in entity descriptions (i.e., no DB-like schema), as well as the variety of domains of entity types covered in KBs (there are ~2,600 diverse vocabularies, but only 109 of them are shared by more than one KB[7]).

- *Volume* is related both to the number of KBs and entities in KBs; the LOD cloud alone contains almost 10,000 KBs with ~150B triples describing more than 55M entities[7].

- *Veracity* stems from various forms of inconsistencies and errors in entity descriptions, due to the limitations of the automatic extraction techniques or of the crowd-sourced contributions.

The above Big Data characteristics of the Web of data call for novel ER frameworks that relax a number of assumptions underlying several methods and techniques proposed in the context of database, machine learning and semantic Web communities [22, 27]. The first is related to the notion of *similarity* that better characterizes entity descriptions in the Web of data. Clearly, Variety renders inapplicable all schema-based similarity measures, which compare specific attribute values. Similarity evidence of entities inside and across KBs can be obtained only by looking at the bag of literals (mostly strings) contained in descriptions, regardless of the attributes they appear as values. As the *value-based* similarity of a pair of entities may still be weak due to Veracity (e.g., the two descriptions of A Clockwork Orange from DBpedia and Freebase in Figure 1.1 having different values for runtime), we need to consider additional sources of evidence related to the *similarity of neighboring* entities, i.e., connected via semantic relations (see the two descriptions of Eyes Wide Shut in DBpedia and Freebase, and the two descriptions of Manhattan, which are neighbors of Stanley Kubrick in both KBs in Figure 1.1).

Figure 1.4 depicts two types of similarity for entities known to match from 4 benchmark datasets used in the literature (details in Table 4.1). Every dot corresponds to a different matching pair, while its shape denotes its origin dataset. The horizontal axis reports the normalized value similarity based on the descriptions common words in a pair (weighted Jaccard [66]), while the vertical one reports the maximum value similarity of their respective entity neighbors. We can observe that the value-based similarity of matching entities significantly varies across different datasets. For *strongly similar entities* (e.g., with a value-based similarity > 0.5) - typically hosted in *homogeneous KBs* from similar or common data sources - existing duplicate detection techniques work well. However, to resolve *nearly similar entities* (e.g., value similarity < 0.5) - typically hosted in heterogeneous KBs from diverse data sources - which cover a large part of the matching pairs

[7]`http://stats.lod2.eu`

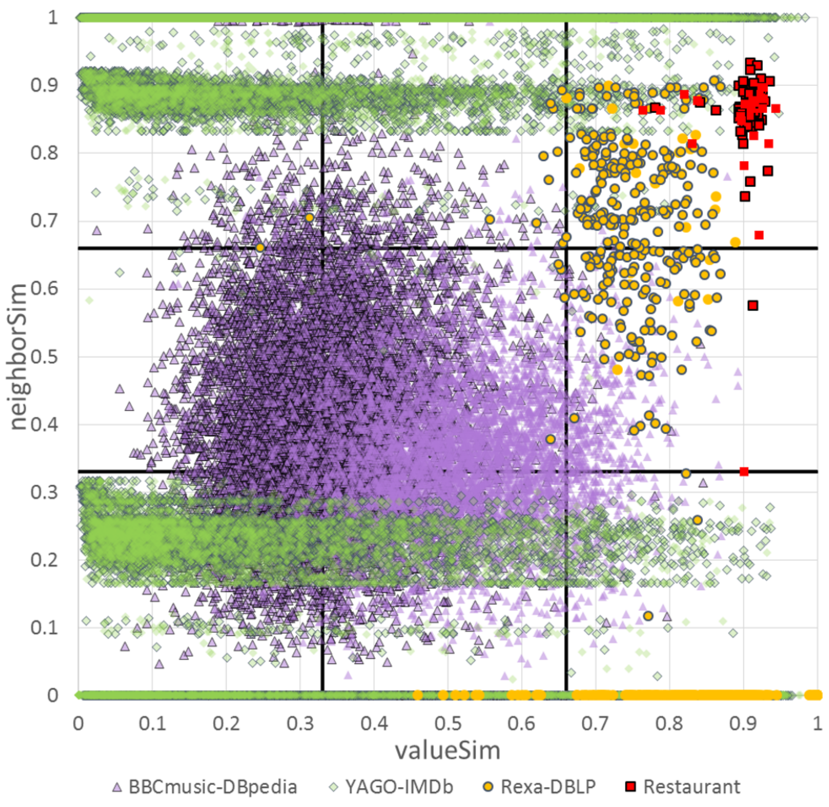


Figure 1.4: Value and neighbor similarity distribution of matching entities in 4 real dataset.

of entities in the Web of data, we need to additionally exploit evidence regarding the similarity of neighboring entities. Existing works in blocking and Meta-blocking in the Web of data are also considering only the content similarity of descriptions, and are thus challenged when dealing with nearly similar entities.

Overall, the main requirements for a Web-scale ER method are the following:

- **Near similarity support.** The heterogeneity of entity descriptions met in the Web of data calls for ER methods that can cope with not only strongly similar, but also nearly similar entities. This means that the blocking phase of an ER workflow should not discard comparisons between descriptions that are nearly similar, as typical blocking methods in databases do, while the matching phase should take into account not only the content, but also the entity neighbors of two descriptions, when deciding if they match.

- **Schema-free.** As the published Web data use a plethora of vocabularies and schemata [29], even within the same KB, it becomes clear that an ER method targeting matches in the Web of data should not rely on a given set of attributes used by all the given entity descriptions[8]. Thus, no step of the ER workflow should rely on the existence and the knowledge of

[8]This is not a restriction on the existence or not of a schema; a Web-scale ER method should work well in either case.

a schema, e.g., blocking cannot operate on the values of a specific attribute only, such as a ZIP code, assuming that all the descriptions will have a value for this attribute.

- **No human in the loop.** The diversity of the cross-domain and multi-type entity descriptions published on the Web does not leave any ground for ER methods relying on domain-experts to create correspondence rules or training sets of labeled matches, as they would on a single domain. Putting humans in the loop of a Web-scale ER is known to pose significant challenges [24]. Thus, matching descriptions in the Web of data should rely entirely on statistics, instead of domain-knowledge, in an unsupervised way.

- **Non-iterative.** Iterative ER methods target nearly similar descriptions, through similarity propagation from their neighbors. This process typically terminates when the iterations converge to a single ER result. However, at the scale of the Web of data, such a process may need too many iteration to converge, making iterative ER inapplicable to our problem.

- **Scalable to massive volumes of data.** It should be clear at this point that only scalable ER methods are applicable at the scale of the Web of data. In this context, only massively parallel implementations of blocking, Meta-blocking and matching can be considered.

To our knowledge, there is no work in ER that satisfies all of these requirements at the same time. Specifically, link discovery tools suggested for the Semantic Web (e.g., LIMES [77], Silk [52, 99]) focus on domain-specific matching rules between entities of a particular type (e.g., on products [45,87]) to infer `owl:sameAs` links. The creation of such rules is labor-intensive and difficult to generalize across domains. On the other hand, learning-based link discovery methods (e.g., [53]) can learn such complex rules, based on a training set, which is often hard to obtain when the number of KBs becomes big.

Iterative methods such as SiGMa [66], LINDA [16] and RiMOM [91] rely on domain knowledge regarding the equivalence of relations between neighboring entities. Initially, they detect strongly similar entities using reasonable heuristics, such as identical literal values. Then, they use these resources as seeds for bootstrapping an iterative algorithm that detects new matches based exclusively on similarity propagation from the neighbors. The more neighboring entities are matching, the stronger is the evidence regarding a candidate entity pair. This process is repeated until converging to a stable solution (i.e., no more matches are identified). Since convergence requires multiple iterations in the Web of data, the employed algorithms cannot scale well to such voluminous datasets.

Finally, blocking methods proposed for structured entities in relational databases [20] (e.g, sorted neighborhood, canopy clustering) rely on blocking keys defined at schema-level. Given the loose structuring and high heterogeneity of entities in the Web of data, we need schema-free blocking methods that could efficiently reduce the number of candidate matches without compromising the effectiveness of matching for entities belonging to multiple types. On the other hand, existing blocking [36] and Meta-blocking [82, 83] methods for the Web of data target only candidate pairs with strong content similarity. To identify such matches, we need disjunctive blocking schemes that exploit different sources of matching evidence.

1.4 Contributions and Outline

To satisfy the requirements of a Web-scale ER, we introduce *MinoanER*, a parallel ER framework that is schema-free, non-iterative, fully automated, i.e., without requiring humans in the loop, targeting not only strongly similar, but also nearly similar matches. Overall, we make the following contributions in this thesis, where each chapter corresponds to one of the ER modules of Figure 1.3 (for a survey of existing works in each module, please refer to our book [22] and tutorials [95, 96]):

- **Blocking.** In Chapter 2, we study the problem of blocking in the context of the Web of data, which enables scaling ER to massive volumes of data in a schema-free way. We make the following contributions, which have been published in [34, 36]:
  - We formalize the notions of *atomic blocking*, operating on a single type of matching evidence (e.g., place two descriptions in the same block, if they have a common word in their values), and *composite blocking*, operating on multiple types of matching evidence (e.g., place two descriptions in the same block, if they have a common word in their values, or a common word in their identifiers).
  - We present the architecture of a massively parallel implementation of blocking methods for Web entities. We explain how our algorithmic design and representation of entity descriptions as (key, value) pairs allows a minimal data exchange between the computational nodes in our cluster, which is a typical bottleneck of such algorithms.
  - We empirically study the behavior of blocking methods for LOD KBs exhibiting different levels of heterogeneity. We are interested in quantifying the *factors that make blocking methods take different decisions* on whether two descriptions from real LOD KBs potentially match or not. We investigate *typical cases of missed matches* of existing blocking methods and examine alternative ways for them to be retrieved. Many matching description pairs have matching entity neighbors even if their content similarity is low. Our analysis shows that a big number of those missed matches could be retrieved if such information was exploited by blocking.
- **Meta-blocking.** In Chapter 3, we study the problem of Meta-blocking, which allows the detection of nearly similar matches in a massively parallel way, operating only on the result of blocking. We make the following contributions, which have been published in [32, 33]:
  - We extend the distinction of blocking methods into atomic and composite, to Meta-blocking: extending the blocking graph, which is the main conceptual model of Meta-

  blocking used with atomic blocking, we further define the *disjunctive blocking graph*, which captures multiple types of matching evidence, allowing the conceptual modeling of composite blocking.

  - We introduce parallel Meta-blocking using *three alternative parallelization strategies*, which provide different advantages when combined with different Meta-blocking edge weighting and pruning strategies, as they feature different I/O costs, number of data-exchange steps and size of exchanged data.
  - We introduce a novel load balancing algorithm called *MaxBlock*, in order to avoid potential bottlenecks associated with the computation-intensive parts of our parallel Meta-blocking. MaxBlock exploits the highly skewed distribution of block sizes in order to split them in partitions of equivalent computational cost (i.e., total number of comparisons). We experimentally compare MaxBlock with state-of-the-art methods and demonstrate that it has significant qualitative and quantitative benefits.

- **Matching.** In Chapter 4, we present our novel non-iterative and scalable matching method for the Web of data, which is fully automated (no human in the loop). We make the following contributions[9]:

  - We define *new similarity metrics* for comparing the values and the neighbors of entities *without requiring knowledge of schema*, the entity types or their correspondences. We rely on simple statistics over the KBs to recognize the most important entity relations involved in neighbor similarity or the most distinctive attributes serving as names of entities. The proposed similarity metrics can be efficiently computed using information provided only by blocking.
  - We propose a *non-iterative matching process* that exploits a disjunctive blocking graph in a *massively parallel* way. Unlike the data-driven convergence of existing iterative systems, our matching method involves a specific number of steps that are independent of data characteristics. Matching entities are found by applying 4 generic *heuristics* to the disjunctive blocking graph, instead of the domain-specific similarity-threshold-based rules employed in state-of-the-art methods. Our experiments show that MinoanER outperforms to a significant extent existing ER tools when matching KBs with high levels of heterogeneity, while it achieves at least equivalent performance over KBs with low levels of heterogeneity, even without making any assumption regarding the alignment of relations in the input.

[9]This work is under submission.

Chapter 2
Blocking

2.1 Introduction

To enhance performance, *blocking* is typically used as a pre-processing step for ER to reduce the number of unnecessary comparisons, i.e., comparisons between descriptions that do not match. After blocking, each description can be compared only to others placed within the same block. The desiderata of blocking are to place (i) matching descriptions in common blocks (*effectiveness*), and (ii) minimize the number of suggested comparisons (*efficiency*). However, efficiency dictates skipping many comparisons, possibly leading to many missing matches, which in turn implies low effectiveness. Thus, the main objective of blocking is to achieve a trade-off between minimizing the number of suggested comparisons, while also minimizing the number of missed matches.

Most blocking methods proposed for structured entities assume both the availability and knowledge of the schema of the input descriptions, i.e., they refer to relational databases. As a typical example, standard blocking [42] would suggest candidate matches in database records of persons, only if those records shared the same ZIP code field (e.g., they live in the same address). To effectively resolve heterogeneous and loosely structured entities across domains, blocking methods proposed for the Web of Data [78, 80, 81] disregard such strong assumptions about schema knowledge and rely on the content, name or identity of descriptions to decide whether they potentially match. For example, token blocking [78] considers two entity descriptions worthy to compare, only if they share at least one common word (token) in their values, regardless of the attribute names for which those values appear. Yet, the effectiveness and efficiency of such blocking methods is not thoroughly studied for LOD KBs exhibiting different levels of heterogeneity in terms of descriptions' content (e.g., number of tokens or frequency distribution of common tokens) and semantics (e.g., number and variety of entity types).

Moreover, most schema-free blocking methods proposed for the Web of Data [80, 81], only take the content of descriptions into account when placing entities in blocks, disregarding any, potentially useful, matching evidence that may be provided by neighboring descriptions, i.e., entities of different types connected via important relations. For example, if two descriptions of the same movie are connected via a "directedBy" relation to two matching descriptions of the same director, then this is an important positive evidence that the movie descriptions also match. We examine whether such neighborhood evidence can be taken into consideration to improve the effectiveness of blocking.

Finally, the process itself of creating the blocks and retrieving the candidate pairs suggested by blocking could raise significant scalability concerns when applied to large volumes of entity collections. Thus, we introduce parallel adaptations of existing blocking methods, which enable blocking in entity collections of massive volumes, without compromising the effectiveness of the original blocking, while minimizing the data exchange between the map and the reduce phase.

In summary, the main contributions of this chapter, which have been published in [34,36], are:

- We formalize the notions of *atomic blocking*, operating on a single type of matching evidence (e.g., place two descriptions in the same block, if they have a common word in their values), and *composite blocking*, operating on multiple types of matching evidence (e.g., place two descriptions in the same block, if they have a common word in their values, or a common word in their identifiers).

- We present the Hadoop architecture of a massively parallel implementation of blocking methods for Web entities. We explain how our algorithmic design and representation of entity descriptions as (key, value) pairs allows a minimal data exchange between the computational nodes in our cluster, which is a typical bottleneck of MapReduce algorithms.

- We empirically study the behavior of blocking methods for LOD KBs exhibiting different levels of heterogeneity. We are interested in quantifying the factors (e.g., frequency distributions of common tokens) that make blocking methods take different decisions on whether two descriptions from real LOD KBs potentially match or not.

- We investigate typical cases of missed matches of existing blocking methods and examine alternative ways for them to be retrieved. Many matching description pairs, given by a ground truth of known matches, have matching entity neighbors even if their content similarity is low. Our analysis shows that a big number of those missed matches could be retrieved if such information was exploited by blocking.

The rest of the chapter is organized as follows: Section 2.2 introduces the formal model of blocking used in this work. Section 2.3 overviews works related to blocking, Section 2.4 presents our implementation of blocking methods in MapReduce. Section 2.5 benchmarks the content-based blocking methods for the Web of Data, and, finally, Section 2.6 summarizes this chapter.

2.2 Formal Blocking Model

Blocking methods are in general defined over key values that can be used to decide whether or not an entity description could be placed in a block using an *indexing function*. The 'uniqueness' of key values determines the number of entity descriptions placed in the same block, i.e., which are considered as *candidate matches*. For entities described in relational databases, *blocking keys* defined by the value of a specific attribute or combination of attributes, i.e., they are *schema-based*. If, for example, the blocking key is defined for the attribute "name", then entity descriptions with same names (or an adequate string transformation function over these names) would end up in the same block. More formally, the building blocks of a blocking method can be defined as [12]:

- An *indexing function* $h_{key} : \mathcal{E} \rightarrow 2^B$ is a unary function that, applied to an entity description using a specific blocking key, returns as a value the set of blocks under which the description will be indexed.

- A *co-occurrence function* $o_{key} : \mathcal{E} \times \mathcal{E} \rightarrow \{true, false\}$ is a binary function that, applied to a pair of entity descriptions, returns 'true' if the intersection of the sets of blocks produced by the indexing function on its arguments, is non-empty, and returns 'false' otherwise; $o_{key}(e_k, e_l) = true$ iff $h_{key}(e_k) \cap h_{key}(e_l) \neq \emptyset$.

It should be stressed that as relational blocking keys have unique values, entity descriptions are placed in at most one block, i.e., the indexing function returns a singular set of blocks. This is not the case of blocking methods for Web entities, given that the employed *schema-free* blocking keys are typically multi-valued. For example, Web entities are usually indexed using the set of tokens appearing in all or a subset of attribute-value pairs. Thus, the same entity description may be placed by the indexing function to several blocks.

The co-occurrence function for every pair of descriptions placed in the same block returns 'true', each pair of descriptions whose co-occurrence function returns 'true' shares at least one common block, and the union of the block elements is the input entity collection. Formally:

Definition 2.1 (Atomic Blocking)**.** *Given an entity collection* $\mathcal{E}$*,* atomic blocking *is defined by an indexing function* h_{key} *for which the generated blocks* $B^{key} = \{b_1^{key}, \ldots, b_m^{key}\}$ *satisfy the following conditions:*

(i) $\forall e_k, e_l \in b_i^{key} : b_i^{key} \in B^{key}, o_{key}(e_k, e_l) = true,$

(ii) $\forall (e_k, e_l) : o_{key}(e_k, e_l) = true, \exists b_i^{key} \in B^{key}, e_k, e_l \in b_i^{key},$

(iii) $\bigcup_{b_i^{key} \in B^{key}} b_i^{key} = \mathcal{E}.$

In general, blocking techniques are characterized by their *redundancy attitude* as: (i) *partitioning*, that place each description into a single block, i.e., $\forall e \in \mathcal{E}, |h_{key}(e)| = 1$, and (ii) *overlapping*, that could place a description in multiple blocks, i.e., $\forall e \in \mathcal{E}, |h_{key}(e)| \geq 1$. When blocking keys fail to uniquely identify an entity, placing a description to a single block according to partitioning approach, would directly result in missed matches, if such matches exist. On the other hand, placing entity descriptions in multiple blocks, as in overlapping approaches, reduces the chances of missing true matches, but entails a greater number of comparisons. As a matter of fact, the occurrence of two descriptions in several blocks, provides evidence regarding their similarity [82]. This way, overlapping approaches can be further divided into: (a) *overlap-positive*, that consider the number of common blocks between two descriptions proportional to the likelihood that they are matches, (b) *overlap-negative*, that consider the number of common blocks between two descriptions inversely proportional to the likelihood that they are matches, and (c) *overlap-neutral*, that consider the number of common blocks between two descriptions irrelevant to the likelihood that they are matches.

Given that using a single key is not enough for building effective and efficient blocking methods, in practice we need to consider several keys that the indexing function exploits to build different sets of blocks. Such a composite blocking method is characterized by a composite co-occurrence function defined as the disjunction or the conjunction of atomic ones. In the sequel, we are interested in disjunctive blocking methods formally defined as follows:

Definition 2.2 (Composite Blocking)**.** *Given an entity collection $\mathcal{E}$,* disjunctive (conjunctive) blocking *is defined by a set of indexing functions H for which the generated blocks $B = \bigcup\limits_{h_{key} \in H} B^{key}$ satisfy the following conditions:*

(i) $\forall e_k, e_l \in b : b \in B, o_H(e_k, e_l) = true,$

(ii) $\forall (e_k, e_l) : o_H(e_k, e_l) = true, \exists b \in B, e_k, e_l \in b,$

where $o_H(e_k, e_l) = \bigvee(\bigwedge)_{h_{key} \in H} o_{key}(e_k, e_l)$ *in disjunctive (conjunctive) blocking.*

Atomic blocking can be seen as a special case of composite blocking, consisting of a singular set of indexing functions, i.e., $H = \{h_{key}\}$.

Measures. The effectiveness and efficiency of a blocking method can be evaluated using the measures described in Table 2.1, with respect to a given *ground truth*, i.e., a set M of known matching pairs of descriptions. Those are the standard measures used to evaluate the quality of the blocking results [21]. The range of all measures is $[0, 1]$, with 1 being the ideal value of a perfect blocking, fulfilling completely both requirements of Definition 2.2. We define the number of True Positives (TP), also referred to as true matches, as

$$TP = |\{(e_k, e_l) | o_H(e_k, e_l) = true \wedge (e_k, e_l) \in M\}|, \tag{2.1}$$

i.e., number of matching pairs that have been placed in a common block, the number of False Positives (FP) as

$$FP = |\{(e_k, e_l) | o_H(e_k, e_l) = true \wedge (e_k, e_l) \notin M\}|, \tag{2.2}$$

i.e., number of non-matching pairs that have been placed in a common block, the number of True Negatives (TN) as

$$TN = |\{(e_k, e_l) | o_H(e_k, e_l) = false \wedge (e_k, e_l) \notin M\}|, \tag{2.3}$$

Table 2.1: Quality Measures.

Name	Formula	Description
Recall	$\frac{TP}{TP+FN}$	Measure what fraction of the known matches are candidate matches.
Precision	$\frac{TP}{TP+FP}$	Measure what fraction of the candidate matches are known matches.
F-measure	$2\frac{Precision \cdot Recall}{Precision+Recall}$	The harmonic mean of precision and recall.
RR	$1 - \frac{\text{comparisons with blocking}}{\text{comparisons without blocking}}$	Returns the ratio of reduced comparisons when blocking is applied.
$H3R$	$2\frac{RR \cdot Recall}{RR+Recall}$	The harmonic mean of recall and reduction ratio.

i.e., number of non-matching pairs that have not been placed in a common block, and the number of False Negatives (FN), also referred to as missed matches, as

$$FN = |\{(e_k, e_l) | o_H(e_k, e_l) = false \wedge (e_k, e_l) \in M\}|, \tag{2.4}$$

i.e., number of matching pairs that have not been placed in a common block.

Intuitively, the recall of blocking measures how many of the known matching pairs of descriptions have been placed in at least one common block, i.e., it captures the *effectiveness* of blocking, while the precision of blocking measures the fraction of matching pairs being placed in common blocks divided by the total number of pairs being placed in common blocks. Reduction Ratio (RR) is the percentage of comparisons that we save if we apply the given blocking method, with respect to an exhaustive comparison of all possible pairs of descriptions, i.e., it captures the *efficiency* of blocking.

In general, a good blocking method should have a low impact on recall, i.e., high effectiveness, and a great impact on the number of required comparisons, i.e., high efficiency. Typically, this trade-off is captured by the F-measure, the harmonic mean of recall and precision. However, in blocking, F-measure is dominated by the values of precision, which are usually many orders of magnitude lower than those of recall, so F-measure cannot be easily used to express this trade-off. Moreover, precision is not as important as recall is for blocking, since precision can only be improved by a non-iterative ER method that follows blocking, whereas the recall of blocking is the upper threshold of such ER methods. Thus, we define $H3R$ as the harmonic mean of recall and RR, a measure which has also been used in [59]. Similar to F-measure, $H3R$ gives high values only when both recall and RR have high values. Unlike F-measure, $H3R$ manages to capture the trade-off between effectiveness and efficiency in a more balanced way. Note that $H3R$ evaluates the actual performance of a blocking method, rather than estimating it, as [80] does. In the sequel, we will explore how different indexing functions are used by various blocking methods to maximize the effectiveness and efficiency of blocking in different contexts.

2.3 Related Work

In this section, we focus on blocking methods proposed in the literature and analyze their applicability to entities met in the Web of Data. We leave out of this review clustering methods which have been proposed for blocking (e.g., [47, 71]).

2.3.1 Schema-based Blocking

The simplest *hash-based* blocking method for relational databases, standard blocking [42], uses a single attribute value as a blocking key and places descriptions in blocks defined for each distinct blocking key. Since each description is placed in exactly one block, standard blocking is a partitioning approach, so each distinct pair of descriptions cannot be compared more than once.

Sort-based blocking methods order entity descriptions according to a sorting criterion and perform blocking based on it. It is expected that matching descriptions will be neighbors after the sorting, so neighbor descriptions constitute candidate matches. Initially, entity descriptions are ordered based on their blocking keys [48]. Then, a window, resembling a block, of fixed length slides over the ordered descriptions, each time comparing only the contents of the window. An adaptive variation of the sorted neighborhood method is to dynamically decide on the size of the window [103]. In this case, adjacent blocking keys in the sorted descriptions that are significantly different from each other, are used as boundary pairs, marking the positions where one window ends and the next one starts. Hence, this variation creates non-overlapping blocks. In a similar line of work, the sorted blocks method [28] allows setting the size of the window, as well as the degree of desired overlap.

Following the intuition of the overlap-positive approaches, *q-gram based blocking* [46] uses a list of q-grams to generate blocking keys, where a q-gram is a substring of *q* characters. For example, the string “Eiffel” can be converted to the list of bi-grams [“ei”,“if”,“ff”,“fe”,“el”]. Sub-lists of this list are generated, by recursively removing one q-gram each time. For instance, some of the sub-lists for the string “Eiffel” are [“ei”,“if”,“ff”,“fe”,“el”], [“if”,“ff”,“fe”,“el”], [“ei”, “ff”,“fe”,“el”], and [“ei”,“ff”,“el”]. Each sub-list is then converted (by concatenation) into a string and used as a blocking key. This way, typographical, or spelling errors are excused. For example, descriptions with the values “Eiffel” and “Eifel”, respectively, will be placed in some common blocks. In a similar way, suffixes of values, i.e., sub-strings produced by removing some of the first characters of the values, can be used for blocking [2], ignoring potential errors in the removed characters. Specifically, each suffix corresponds to a distinct blocking key, and entity descriptions containing this suffix are inserted into the block corresponding to this suffix. To prevent a large number of descriptions being placed into the same block, e.g., when using suffixes of small size, two thresholds are set: (i) a threshold reflecting the minimum length of suffix strings that will be generated and (ii) a threshold reflecting the maximum block size, i.e., number of entity descriptions contained in each block. String-map [57] maps string blocking keys to objects in a d-dimensional Euclidean space. Each dimension is defined by selecting two objects, called pivots, that are chosen to be as dissimilar as possible, using a similarity measure. Blocks are then generated by extracting objects in this space that are close to each other, i.e., within a distance threshold. String-Map is based on FastMap [40], an algorithm with linear complexity to the number of strings.

Finally, [60] introduces a method for building blocks using Maximal Frequent Itemsets (MFI) as blocking keys. Abstractly, each MFI (an itemset can be a set of tokens) of a specific attribute in the schema of a description defines a block, and descriptions containing the tokens of an MFI for this attribute are placed in a common block. Using frequent itemsets to construct blocks may significantly reduce the number of candidates for matching pairs. However, since many matching descriptions share few, or even no common tokens, further requiring that those tokens are parts of frequent itemsets is too restrictive for those pairs of matching descriptions, resulting in many missed matches in the Web of data. Moreover, MFI blocking requires a-priori knowledge of the desired block sizes, and is also based on the notion of a schema, information which is unavailable at the Web of data.

Although blocking has been extensively studied for tabular data, the proposed approaches cannot be used for the Web of data, since their blocking keys rely on the existence of a schema, i.e., a fixed set of attributes, based on which the descriptions are placed into blocks. However, the high heterogeneity of entity descriptions in the Web of data makes the use of schema-based blocking keys inapplicable. In this context, entity descriptions do not follow a fixed schema, and, furthermore, even a single description typically uses attributes defined in multiple LOD vocabularies.

Threshold-based blocking

String-similarity join algorithms (e.g., [9, 18, 102]) construct blocks which are guaranteed to contain all pairs of descriptions whose values' string similarities for a specific attribute are above a certain threshold and potentially some pairs whose string values similarities are below that threshold. To achieve that, without computing the similarity of all pairs of descriptions, this family of algorithms use the tokens of the attribute values of the descriptions as blocking keys. This inverted index is created only by the first non-frequent tokens of each description (i.e., the most discriminating), based on the *prefix filtering* principle [18]. [9] additionally applies a *size filtering* [5] on the sets of tokens to disregard some of the candidate pairs, based on the fact that $Jaccard(x, y) \geq t \Rightarrow t \cdot |x| \leq |y|$. The ppjoin+ algorithm [102] introduces a *positional filtering*, i.e., the position in the ordered set of tokens, in which a token appears, to further reduce the number of candidate pairs. Specifically, it estimates the maximum possible intersection size of two token sets x, y by considering that, if the first common token of x and y is the first token in x and the second token in y, then the maximum intersection that these sets can have is $1 + min(|x| - 1, |y| - 2)$.

Tuning the appropriate similarity threshold is non-trivial and it also affects the performance of the string-similarity join algorithms [56]. Smaller thresholds entail less pruning, and thus, more time. Furthermore, [73] proves experimentally that algorithms based on prefix filtering are only effective when the similarity threshold is extremely high. However, this is not the case in the Web of data, where highly heterogeneous descriptions, yielding very low similarity in their literal values, can refer to the same entity.

In a similar fashion to string-similarity joins, the key idea of (disjunctive) blocking with *Locality-Sensitive Hashing* (LSH) (e.g., [70]) is to hash descriptions multiple times, using a family of indexing functions, in such a way that similar descriptions (e.g., with Jaccard similarity, approximated by minhasing [17]) are more likely (with probabilistic guarantees) to be placed into the same bucket than dissimilar ones. Any two descriptions that hash at least once into the same bucket, for any of the employed indexing functions, are considered to be a candidate pair. This technique assumes an a-priori knowledge of a minimum similarity threshold between entity description pairs, above which, such pairs are considered candidate matches. However, as we will see in our experimental evaluation (see Section 2.5), often, matching descriptions do not share many common tokens and thus, have very low, even zero, similarity when computed only on the values of their attributes. Those matches would not be placed in the same bucket and thus, they would not be considered candidate matches. Effectively choosing a minimum similarity threshold also depends on the KBs. For example, when seeking matches between two homogeneous KBs, a high similarity threshold can be used, since such KBs have more similar values. Using a lower threshold in homogeneous KBs would result in many false candidate pairs. Accordingly, using a high similarity threshold in heterogeneous KBs, in which descriptions have lower similarity values, would yield many missed matches. Consequently, applying LSH across domains is an open research problem, due to the difficulty in knowing or tuning a similarity threshold that can be generalized to identify matches across several domains in an effective and efficient way.

2.3.2 Schema-free Blocking

The simplest blocking method for the Web of Data is *token blocking* [78], which relies on the minimal assumption that matching descriptions should at least share a common token. It indexes descriptions based on the set of all tokens in the values of an entity description. Each distinct token t in the values of a description, defines a new block b_t, essentially building an inverted index of descriptions. Two descriptions are placed in the same block, if they share a token in their values.

Example 2.1. *Given the entity collection of Figure 2.1, Figure 2.2 shows the blocks generated by token blocking. In the generated blocks, we save the comparisons (e_1, e_5), (e_1, e_7), (e_2, e_4), (e_3, e_4), (e_4, e_5), (e_5, e_6) and (e_6, e_7), and we successfully place the matches (e_1, e_6) and (e_2, e_5) in common blocks. Still, pairs, such as (e_1, e_2), (e_1, e_3), and (e_3, e_6), lead to unnecessary comparisons. Note also that the pair (e_1, e_6) is contained in 4 different blocks, which leads to repeated comparisons.*

Next, we present three extensions of token blocking: attribute clustering blocking, in which

e_1 = {(about, Eiffel Tower), (architect, Sauvestre), (year, 1889), (located, Paris)}
e_2 = {(about, Statue of Liberty), (architect, Bartholdi Eiffel), (year, 1886), (located, NY)}
e_3 = {(about, Auguste Bartholdi), (born, 1834), (work, Paris)}
e_4 = {(about, Joan Tower), (born, 1938)}
e_5 = {(work, Lady Liberty), (artist, Bartholdi), (location, NY)}
e_6 = {(work, Eiffel Tower), (year-constructed, 1889), (location, Paris)}
e_7 = {(work, Bartholdi Fountain), (year-constructed, 1876), (location, Washington)}

Figure 2.1: A set of entity description.

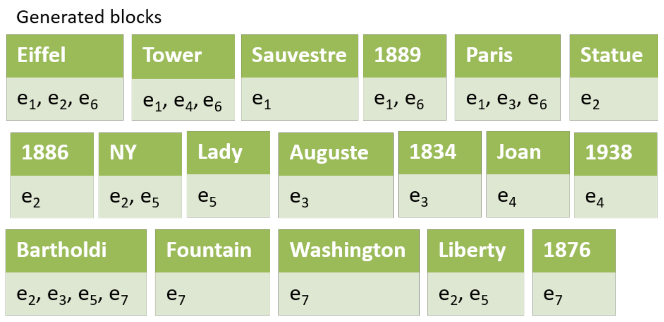


Figure 2.2: Token blocking example. Descriptions having a common token are placed in a common block.

candidate matches should at least share a common token for similar attributes known globally, prefix-infix(-suffix) blocking, in which candidate matches should additionally share a common URI infix, and ppjoin+, in which only a small subset of the tokens in the descriptions' values are used as blocking keys.

To tackle the coarse-grained approach of token blocking, attribute clustering blocking [81] further requires the common tokens of descriptions that should be considered candidate matches to appear for semantically similar attributes. This should improve the low precision of token blocking, at a, hopefully, low cost in recall. In the previous example, it would not place e_1 and e_3 in the same block, for their common token Paris, because this token appears in the values of two semantically different attributes ($located$ and $work$). To achieve this, prior to token blocking, it clusters attributes based on the similarities of their values over the entire dataset. Each attribute from one entity collection is connected to its most similar attribute in the other entity collection and connected attributes, taken by transitive closure, form non-overlapping clusters. Then, each token t in the values of an attribute, belonging to a cluster c, defines a block $b_{c.t}$. Hence, comparisons between descriptions without a common token in a similar attribute, are discarded. Like token blocking, attribute clustering generates overlapping blocks. Compared to the blocks of token blocking, it produces a larger number of smaller blocks.

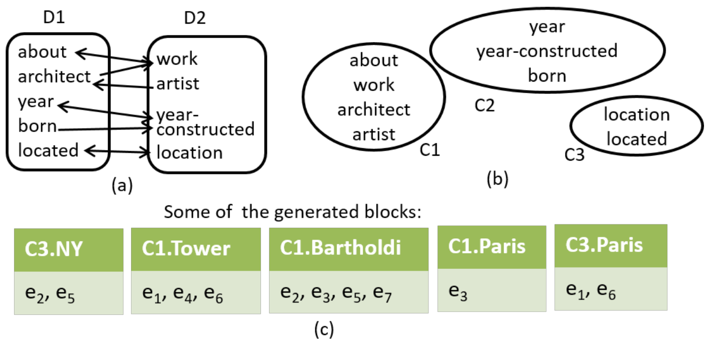
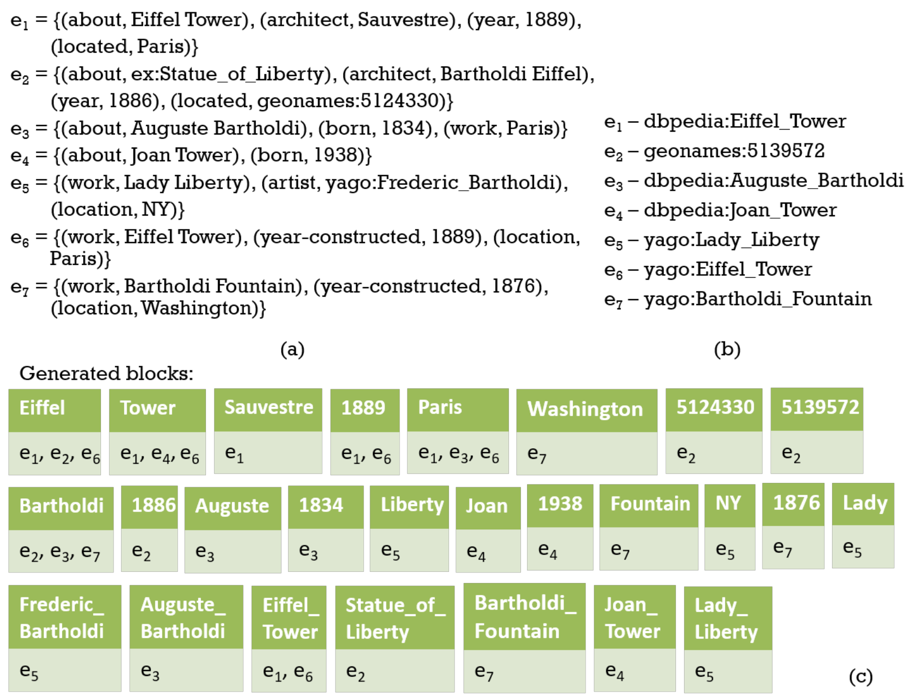


Figure 2.3: Attribute clustering blocking example. Pairs of most similar attributes are linked (a). Connected attributes form clusters (b). Descriptions with a common token in the values of attributes of the same cluster, are placed in a common block (c).

Example 2.2. *As an example, consider that the descriptions of Figure 2.1 consist of two clean entity collections, $D_1 = \{e_1, e_2, e_3, e_4\}$ and $D_2 = \{e_5, e_6, e_7\}$. Using Jaccard similarity, the attribute $work$ (with values: {Lady, Liberty, Eiffel, Tower, Bartholdi, Fountain}) of D_2 is the most similar attribute to $about$ of D_1. Similarly, the transitive closure of the pairs of most similar attributes between D_1 and D_2 (Figure 2.3(a) depicts such pairs), produce the clusters of attribute names (Figure 2.3(b)). A subset of the blocks constructed for each cluster is shown in Figure 2.3(c). This way, the comparisons (e_1, e_3) and (e_3, e_6) that were suggested by token blocking, due to the common token* Paris*, are now discarded, since the token* Paris *appears in different attribute clusters for e_3 than for e_1 and e_6, as shown in the bottom blocks of Figure 2.3(c). Again, both unnecessary (e.g., e_4 and e_6 are both placed in block $C1.Tower$ (Figure 2.3(c))), and repeated (e.g., (e_1, e_3) is still contained in 4 different blocks) comparisons are generated.*

Unlike previous methods analyzing the content of descriptions, prefix-infix(-suffix) blocking [80] exploits the naming pattern in the descriptions' URIs. The prefix describes the domain of the URI, the infix is a local identifier, and the optional suffix contains details about the format, or a named anchor. For example, the prefix of "http://liris.cnrs.fr/olivier.aubert/foaf.rdf#me" is "http://liris.cnrs.fr", the infix is "/olivier.aubert" and the suffix is "/foaf.rdf#me". Given a set of descriptions, this method generates one block collection using as blocking keys the tokens in the descriptions literal values and the URI infixes. It is constrained by the extent to which common naming policies are followed by the KBs. In a favourable scenario, it creates additional blocks than token blocking for the names of the descriptions, which enables to consider matching descriptions, even with no common tokens in their literal values.

Example 2.3. *Figure 2.4(c) shows the blocks produced after applying prefix-infix(-suffix) blocking to the descriptions of Figure 2.4(a) (the descriptions of Figure 2.1, slightly modified to illustrate the*



e_1 = {(about, Eiffel Tower), (architect, Sauvestre), (year, 1889), (located, Paris)}
e_2 = {(about, ex:Statue_of_Liberty), (architect, Bartholdi Eiffel), (year, 1886), (located, geonames:5124330)}
e_3 = {(about, Auguste Bartholdi), (born, 1834), (work, Paris)}
e_4 = {(about, Joan Tower), (born, 1938)}
e_5 = {(work, Lady Liberty), (artist, yago:Frederic_Bartholdi), (location, NY)}
e_6 = {(work, Eiffel Tower), (year-constructed, 1889), (location, Paris)}
e_7 = {(work, Bartholdi Fountain), (year-constructed, 1876), (location, Washington)}

(a)

e_1 – dbpedia:Eiffel_Tower
e_2 – geonames:5139572
e_3 – dbpedia:Auguste_Bartholdi
e_4 – dbpedia:Joan_Tower
e_5 – yago:Lady_Liberty
e_6 – yago:Eiffel_Tower
e_7 – yago:Bartholdi_Fountain

(b)

Generated blocks:

Eiffel	Tower	Sauvestre	1889	Paris	Washington	5124330	5139572
e_1, e_2, e_6	e_1, e_4, e_6	e_1	e_1, e_6	e_1, e_3, e_6	e_7	e_2	e_2

Bartholdi	1886	Auguste	1834	Liberty	Joan	1938	Fountain	NY	1876	Lady
e_2, e_3, e_7	e_2	e_3	e_3	e_5	e_4	e_4	e_7	e_5	e_7	e_5

Frederic_ Bartholdi	Auguste_ Bartholdi	Eiffel_ Tower	Statue_of_ Liberty	Bartholdi_ Fountain	Joan_ Tower	Lady_ Liberty
e_5	e_3	e_1, e_6	e_2	e_7	e_4	e_5

(c)

Figure 2.4: Prefix-infix(-suffix) blocking example. A set of descriptions (a), their subject URIs (b), and the blocks from their tokens and infixes (c).

characteristics of the method), while Figure 2.4(b) presents the URI identifiers of the descriptions.

Summary of schema-free blocking methods

Overall, Table 2.2 summarizes, simplified, the criteria employed by the aforementioned schema-free blocking methods to consider two descriptions as candidate matches, i.e., their co-occurrence functions as defined in Section 2.2. Token blocking makes the simplest assumption about matching pairs of descriptions, i.e., that they share at least one common token in their values, aiming at the maximum possible recall, even if this entails a low precision, since many pairs that share a common word are expected to be non-matches. To tackle this coarse-grained approach, attribute clustering blocking further requires the common tokens of descriptions that should be considered candidate matches to appear for semantically similar attributes. This should improve the low precision of token blocking, at a, hopefully, low cost in recall (attribute clustering blocking cannot have higher recall than token blocking). Finally, as we will see in the next section, it is quite common for matches in heterogeneous data, to not even share a single token. To cope with such cases, prefix-infix(-suffix) blocking assumes that those kinds of pairs should at least have similar parts of their entity identifiers. In the next section we will see if those assumptions are verified or not, for

Table 2.2: Co-occurrence functions for considering two descriptions candidate match.

Method	**Criterion**
Token blocking	The descriptions have a common token in their values.
Attribute clustering blocking	The descriptions have a common token in the values of attributes that have similar values in overall.
Prefix-infix(-suffix) blocking	The descriptions have a common token in their literal values, or a common URI infix.
ppjoin+	The descriptions have a common infrequent token and a close number of tokens overall.
LSH	The descriptions hash at least once into the same bucket, for any of the employed hash functions.

Table 2.3: Blocking methods with respect to the redundancy attitude and algorithmic attitude.

	Redundancy attitude				**Algorithmic attitude**	
		Overlapping				
Blocking approach	Partitioning	Overlap-positive	Overlap-negative	Overlap-neutral	Hash-based	Sort-based
Standard blocking [42, 61]	✓				✓	
Q-grams [46]		✓			✓	
Suffixes [2]		✓			✓	
Sorted neighborhood [48, 63]				✓		✓
Adaptive sorted neighborhood [103]	✓					✓
MFI [60]		✓			✓	
Token blocking [78]		✓			✓	
Attribute clustering blocking [81]		✓			✓	
Prefix-infix(-suffix) blocking [80]		✓			✓	
ppjoin+ [98, 102]		✓			✓	
LSH blocking [70]		✓			✓	

different levels of heterogeneity in the input entity collections. Since ppjoin+ and LSH blocking require a pre-defined similarity threshold for pairs to be considered as candidate matches and there is no generic or efficient way of setting it, we have not included these methods in our experimental study.

The categorization of the blocking methods presented in this chapter with respect to the characteristics of the produced blocks (i.e., partitioning vs. overlapping blocks) and the algorithmic approach (hash-based vs. sort-based) used are presented in Table 2.3. Partitioning approaches are sensitive to typos and erroneous values, since misplaced entity descriptions potentially result in missed matches. Therefore, due to the varying data quality, they are not suited for ER in the Web of data. Data heterogeneity makes sort-based approaches not easily applicable as well, since the missing knowledge of the schema of the data incommodes the sorting process.

2.4 Scaling Blocking Methods to Very Large Entity Collections

Next, we present the MapReduce version of the evaluated methods, designed to cope with Web data. MapReduce [23] offers a fault-tolerant, optimized execution for applications, distributed across independent nodes. Its programs consist of two consecutive procedures grouped together into *jobs*: `Map` receives a (`key`, `value`) pair and transforms it into one or more new pairs; `Reduce` receives a set of pairs that share the same `key` and are sorted according to their `value`, and performs a summary operation on them to produce a new, usually smaller set of pairs. Optionally, a *Combine* function can be provided, to process the output of each mapper, like a local, mini-reducer, and decrease the amount of data transferred through the cluster network.

In our implementation, we try to minimize the size of data transferred from mappers to reducers, by using a minimal representation of entity ids as numerical ids, by using a combiner whenever possible, and by transmitting as little information as necessary for each task. We have also tried to minimize the number of MapReduce jobs required for each method, since each new job bears a significant I/O and setup cost.

2.4.1 Token Blocking

Token blocking is essentially an inverted index of descriptions. Each token is a key in this index, associated with a list of all the descriptions containing it. Our implementation of token blocking in MapReduce is based on the procedure illustrated in Figure 2.5. In the map phase, one entity description of the local input split is processed at a time. For each token t in the values of a description e_i, a (t, e_i) pair is emitted by the mapper. In the reduce phase, all descriptions having a common token will be processed by the same reduce function, i.e., placed in the same block.

2.4.2 Attribute Clustering Blocking

Given two clean entity collections, our implementation of attribute clustering blocking can be briefly sketched by the following steps, each representing a MapReduce job. Figure 2.6 illustrates a high-level flow of the process.

Attribute Creation. First, we gather the values of each attribute. In the map phase, we emit an (*attribute*, *value*) pair for each attribute-value pair in a description. We also keep the entity collection of this attribute in the *key*. In the reduce phase, all the values of an attribute are grouped together and their concatenation is emitted as the value of this attribute.

Attribute Similarities. In the second job, we compute the pairwise Jaccard similarities be-

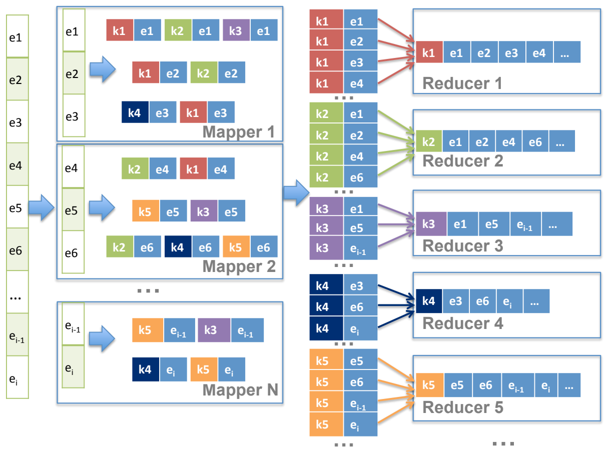


Figure 2.5: Token blocking in MapReduce.

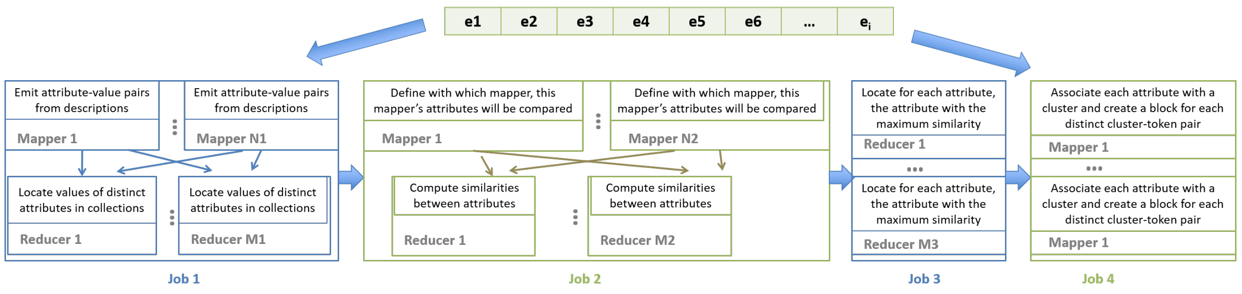

Emit attribute-value pairs from descriptions
Mapper 1
Mapper N1
Locate values of distinct attributes in collections
Reducer 1
Reducer M1
Job 1
Define with which mapper, this mapper's attributes will be compared
Mapper 1
Mapper N2
Compute similarities between attributes
Reducer 1
Reducer M2
Job 2
Locate for each attribute, the attribute with the maximum similarity
Reducer 1
Reducer M3
Job 3
Associate each attribute with a cluster and create a block for each distinct cluster-token pair
Mapper 1
Job 4

Figure 2.6: Attribute clustering blocking in MapReduce.

tween the trigram sets of all attributes. A mapper outputs each input attribute, as many times, as the number of total mappers. Each time, a composite *key*, consisting of the current mapper id and another mapper id, will determine in which reducer the attribute will be placed, and to which other attributes it will be compared. For example, assuming 3 mappers in total, the mapper with id 2, emits for each input attribute, 3 different *keys*: 1_2, 2_2, and 2_3. The keys 1_2 and 2_3 will result in comparing the contents of mapper 2 to the contents of mappers 1 and 3, while 2_2 will result in comparing the contents of mapper 2 to each other. The *value* of each emitted pair is the input attribute with its values and the current mapper id. In the reduce phase, we compute similarities of attributes, ensuring that each comparison is performed once. For each pair of attributes, we emit a (*key*, *value*) pair, with one attribute being the *key* and the second attribute along with their similarity score being the *value*.

Best Match. In the third job, we use an identity mapper, which just forwards its input. A combiner keeps for each attribute of each entity collection, only the attribute of the other entity collection with the local highest similarity score. In the reduce phase, we pick for each attribute of each

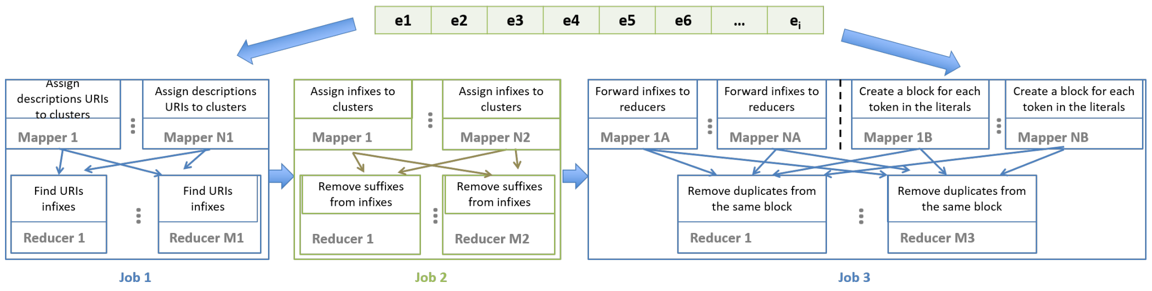

e1
e2
e3
e4
e5
e6
...
e_i
Assign descriptions URIs to clusters
Mapper 1
Assign descriptions URIs to clusters
Mapper N1
Find URIs infixes
Reducer 1
Find URIs infixes
Reducer M1
Job 1
Assign infixes to clusters
Mapper 1
Assign infixes to clusters
Mapper N2
Remove suffixes from infixes
Reducer 1
Remove suffixes from infixes
Reducer M2
Job 2
Forward infixes to reducers
Mapper 1A
Forward infixes to reducers
Mapper NA
Create a block for each token in the literals
Mapper 1B
Create a block for each token in the literals
Mapper NB
Remove duplicates from the same block
Reducer 1
Remove duplicates from the same block
Reducer M3
Job 3

Figure 2.7: Prefix-infix(-suffix) blocking in MapReduce.

entity collection, the attribute with the maximum similarity score, in overall, from the other entity collection. Before this job ends, we start the first step of clustering the most similar attributes together. To accomplish that, we emit for each best-matching attribute pair, two (attribute, clusterId) pairs, one for each attribute, with the same clusterId. Ids of clusters with common attributes are marked, in order to be merged at the next step. This job uses a single reduce task, in which an iterative (sequential) transitive closure algorithm is run, in order to make sure that the clusters to be merged will cover the transitive closure of connected attributes (i.e., if the clusters of connected pairs of attributes are different, those clusters have to be merged into one cluster, iteratively, until all connected attributes belong to the same cluster).

Final Clustering and Blocking. In the final job, we associate each attribute with a final cluster id, according to the marks of the previous step. Then, we perform token blocking (Section 2.4.1), with only difference that in each *key* emitted from a mapper, there is also a cluster prefix, enabling distinctions between blocks for the same token. For example, if the same token t appears in a description e_i for attributes in clusters c_j and c_k, then the mapper will emit the pairs $(c_j.t, e_i)$ and $(c_k.t, e_i)$, instead of a single (t, e_i).

2.4.3 Prefix-Infix(-Suffix) Blocking

Our MapReduce implementation of this method consists of three jobs. The first two are the MapReduce adaptation of the infix extraction algorithm [80]. The third job reads the descriptions, as well as the infixes produced by the second job and creates the blocks. A high-level representation of the process is depicted in Figure 2.7.

Prefix Removal. In the map phase, we output a (*key*, *value*) pair for each URI in a description. The *key* is the second token of the URI (after "http") and the *value* consists of the whole URI and the identifier of the entity description having this URI. This clusters the URIs according to their second token, which usually represents the domain (e.g., "dbpedia"), in the reduce phase. For each URI in a cluster, we find, among all its possible prefixes, the one with the largest set of distinct (immediately) next tokens. The part of the URI following the prefix is the *key* of each output pair, with *value* consisting of the input *key*, i.e., the second token of the URI, and the entity identifier having this URI.

Suffix Removal. We apply Prefix Removal, on each reverse URI (without prefix), to remove the suffix.

Infix&Token Blocking. We create the final blocks, based on the output of Suffix Removal and the initial entity collection. We use two different mappers, operating in parallel; an identity mapper, forwarding the output of Suffix Removal and the mapper of token blocking, operating on the tokens of literal values only of the input descriptions. In the reduce phase, all the descriptions having a common token or infix in their literals or URIs will be placed in the same block.

Summary of parallel blocking

The parallel adaptation of existing blocking methods in MapReduce, enables scaling them to large volumes of entity collections. Sequential token blocking does not exploit the fact that practically all computations can be executed in parallel for different parts of the input, this way reducing the blocking time significantly. Regarding the other two blocking methods, a sequential clustering of the attributes and URI prefixes is too resource-intensive to enable scaling them to big entity collections without an expensive high-end server. Our parallel adaptations of those methods enables a fast and cost-efficient blocking in such entity collections. Finally, we do not claim that our implementation is optimal; we have not examined a better load balancing than the default hash-based, and the same parallelization strategy could be easily adapted in other platforms, such as Apache Spark[1], yielding more efficient results for methods that require more than one MapReduce jobs (i.e., attribute clustering blocking and prefix-infix(-suffix) blocking).

2.5 Benchmarking Content-based Blocking Methods in the Web of Data

In this section, we present the experimental framework we have designed for evaluating existing blocking methods. We describe the datasets and the measures we employed to study the behavior of the blocking methods under different characteristics of entity descriptions in the LOD cloud. We have used a cluster of 15 Ubuntu 12.04.3 LTS servers (1 master, 14 slaves), each with 8 CPUs, 8GB RAM and 60GB of disk, provided by ~okeanos [65]. Each node could run simultaneously 4 map or reduce tasks, each with a heap size of 1250MB, leaving resources required for I/O and communication with the master. We used Apache Hadoop 1.2.0 and Java version 1.7.0_25 from OpenJDK. The source code and datasets used in this study are publicly available[2].

[1] `https://spark.apache.org/`
[2] `csd.uoc.gr/~vefthym/minoanER/`

2.5.1 Datasets

Our study relies on real data from the Billion Triples Challenge 2012 dataset[3] (BTC12), DBpedia, Kasabi[4], the Linked Archives Hub project[5], and OAEI benchmarks[6]. To capture the differences in the heterogeneity and semantic relationships of descriptions, we distinguish between data originating from KBs in the *center* and the *periphery* of the LOD cloud. In general, central KBs, such as DBpedia and Freebase, are derived from a common source, Wikipedia, from which they extract information regarding an entity. Such descriptions often refer to the original wiki page and feature synonym attributes whose values share a significant number of common tokens. Since they have been exhaustively studied in the literature, descriptions across central LOD KBs are heavily interlinked using in their majority *owl:sameAs* links [90], expressing equivalence relations. In our experiments, we used the DBpedia (*BTC12DBpedia*) and Freebase (*BTC12Freebase*) KBs from BTC12, and the raw infoboxes from DBpedia 3.5 (*Infoboxes*), i.e., two different versions of DBpedia. From the OAEI benchmark datasets, we used the one including the *DBLP* and *Rexa* (OAEI 2009) - describing authors and publications - that has been widely used in the literature (e.g., in [66]). We also included a movies dataset, used in [81], extracted from DBpedia movies and *IMDB*, to validate the correctness of our algorithms.

On the other hand, KBs in the periphery of the LOD cloud are highly heterogeneous and sparsely interlinked. In our experiments, we considered the *BTC12Rest*, the *BBCmusic* and the *LOCAH* KBs. *BTC12Rest* originates from BTC12, which consists of multiple KBs, like DBLP, geonames and drugbank. *BBCmusic* originates from Kasabi and contains descriptions regarding music bands and artists, extracted from MusicBrainz and Wikipedia. For *LOCAH*, we used the latest published version at Archives hub (March 2014). This, rather small KB links descriptions of people, from UK archival institutions, with their descriptions in DBpedia.

Table 2.4: KBs characteristics.

	RDF triples	**entity descriptions**	**avg. attribute-value pairs per description**	**attributes**	**entity types**	**attributes/ entity types**	**duplicates (within dataset)**
BTC12DBpedia	102,306,242	8,945,920	11.44	36,354	258,202	0.14	0
Infoboxes	27,011,880	1,638,149	16.49	31,857	5,535	5.76	0
BTC12Rest	849,656	31,668	26.83	518	33	15.7	863
BTC12Freebase	25,050,970	1,849,180	13.55	8,323	8,232	1.01	12,058
BBCmusic	268,759	25,359	10.60	29	4	7.25	372
LOCAH	12,932	1,233	10.49	14	4	3.5	250
DBpedia$_{mov}$	180,680	27,615	6.54	5	1	5	0
IMDB	816,012	23,182	35.20	7	1	7	0
DBLP	12,074,269	1,642,945	7.35	30	10	3	0
Rexa	64,787	14,771	4.39	12	3	4	0

Table 2.4 provides statistics about these KBs, for the number of contained triples, descriptions,

[3]`km.aifb.kit.edu/projects/btc-2012/`
[4]`archive.org/details/kasabi`
[5]`data.archiveshub.ac.uk/`
[6]`oaei.ontologymatching.org/`

attributes, and the average number of attribute-value pairs per description. We have also included the number of entity types, taken as the distinct values of the property *rdf:type*, when provided. Observe that *BTC12DBpedia* contains more types than attributes. This is due to the fact that DBpedia entities may have multiple types from taxonomic ontologies like Yago. *IMDB* is the KB with the highest number of attribute-value pairs per description. Finally, we have included in each KB the number of duplicate descriptions based on our ground truth, i.e., descriptions that have been reported to be equivalent (via *owl:sameAs* links) across all KBs of our testbed. Taking into account the transitivity of equality, those descriptions should be regarded as matches, too.

In this setting, we combine *BTC12DBpedia* with each of the KBs of Table 2.4 to produce the datasets presented in Table 2.5, on which we finally ran our experiments. To combine two KBs, for the dirty ER setting, we simply concatenate them into a singe file, while for clean-clean ER, we seek candidate matches between those KBs.

- ***D*1** combines *BTC12DBpedia* with *Infoboxes*. Since it contains two versions of the same KB, it is considered as a homogeneous dataset. This is the biggest dataset in terms of triples, as well as attributes.
- ***D*2** combines *BTC12DBpedia* with *BTC12Rest*. Since it is constructed by many different KBs, it is the most heterogeneous dataset. Note that *BTC12Rest* has the highest number of attributes per entity type.
- ***D*3** combines *BTC12DBpedia* with *BTC12Freebase*. It is the biggest dataset in terms of entity descriptions, matches, entity types and comparisons.
- ***D*4** combines *BTC12DBpedia* with *BBCmusic*. Note that *BBCmusic* extracts some of its data from MusicBrainz, which, in turn, extracts data from Wikipedia. Also, *BBCmusic* is edited and maintained by users and BBC staff.
- ***D*5** combines *BTC12DBpedia* with *LOCAH*, the smallest KB, both in terms of triples and entity descriptions.
- ***D*6** combines DBpedia movies and *IMDB*, as originally used in [81]. It is the most homogeneous dataset, it only contains descriptions of movies (i.e., a single entity type) using the smallest number of attributes among all datasets. However, the significantly greater (even by six orders of magnitude, compared to the other datasets) ratio of matches to non-matches is not typical of the datasets we can find in the Web of data.
- ***D*7** combines *DBLP* and *Rexa*. Both KBs use the same ontology; *Rexa*'s attributes are a subset of those used by *DBLP*. Also, it is the dataset with the lowest number of attribute-value pairs per description. Note that this dataset is a typical benchmark used to evaluate instance matching algorithms.

Following the distinction of our KBs between central and peripheral, we also distinguish our datasets between central (*D*1, *D*3, *D*6, and *D*7), composed of central KBs, and peripheral (*D*2, *D*4, and *D*5), part of which are peripheral KBs. For all the datasets, we consider both their *clean-clean* and *dirty* versions. In practice, for our datasets, the *clean-clean* and *dirty* versions of a KB are the same; their distinction serves only as means for measuring how well a blocking method can

Table 2.5: Datasets characteristics.

	D1	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**
RDF triples	129,318,122	103,155,898	127,357,212	102,575,001	102,319,174	996,692	12,139,056
entity descriptions	10,584,069	8,977,588	10,795,100	8,971,279	8,947,153	50,797	1,657,716
avg. attribute-value pairs per description	12.22	11.49	11.80	11.43	11.44	19.62	7.32
attributes	68,211	36,872	44,677	36,383	36,368	12	42
entity types	263,737	258,232	266,434	258,206	258,205	1	10
matches	1,564,311	30,864	1,688,606	23,572	1,087	22,405	1,532
matches (incl. duplicates)	1,564,311	31,727	1,700,664	23,944	1,337	22,405	1,532
matches/non-matches	$1.07 \cdot 10^{-7}$	$1.09 \cdot 10^{-7}$	$1.02 \cdot 10^{-7}$	$1.04 \cdot 10^{-7}$	$9.85 \cdot 10^{-8}$	$3.5 \cdot 10^{-5}$	$6.3 \cdot 10^{-8}$
matches/non-matches (dirty)	$2.79 \cdot 10^{-8}$	$7.87 \cdot 10^{-10}$	$2.92 \cdot 10^{-8}$	$5.95 \cdot 10^{-10}$	$3.34 \cdot 10^{-11}$	$1.74 \cdot 10^{-5}$	$1.1 \cdot 10^{-9}$
comparisons (w/o blocking)							
clean-clean	$1.47 \cdot 10^{13}$	$2.83 \cdot 10^{11}$	$1.65 \cdot 10^{13}$	$2.27 \cdot 10^{11}$	$1.1 \cdot 10^{10}$	$6.4 \cdot 10^{8}$	$2.4 \cdot 10^{10}$
dirty	$5.6 \cdot 10^{13}$	$4.03 \cdot 10^{13}$	$5.83 \cdot 10^{13}$	$4.02 \cdot 10^{13}$	$4 \cdot 10^{13}$	$1.29 \cdot 10^{9}$	$1.37 \cdot 10^{12}$

identify links across different KBs and within the same KB.

GroundTruth. Our ground truths were built using a methodology met in the literature (e.g., [80, 81]). For *D*2-*D*5, we consider the *owl:sameAs* links to/from DBpedia 3.7 (the version used in BTC12). For *D*1, we consider the subject URIs of *Infoboxes* that also appear as subjects in *BTC12DBpedia*. The ground truth of *D*6, provided in [81], is made of DBpedia movies connected with IMDB movies through the *imdbId* property. The ground truth of *D*7 is provided by OAEI, since it is a benchmark dataset, containing equivalence links between authors, as well as publications.

Our pre-processing, implemented in MapReduce, parses RDF triples in order to transform them into entity descriptions, which are the input of the methods used in our study. It simply groups the triples by subject, and outputs each group as an entity description, using the subject as the entity identifier, removing triples containing a blank node. Moreover, we kept only the entity descriptions for which we know their linked description in *BTC12DBpedia* and removed the rest. This way, we know that any suggested comparison between a pair of descriptions outside the ground-truth is false.

2.5.2 Quality Results

Identified Matches (TPs)

Token blocking: The premise of this algorithm is that matching descriptions should at least share a common token, disregarding the comparisons between descriptions that do not share common tokens. Therefore, the higher the number of common tokens, i.e., tokens shared by the KBs composing a dataset, a description has, the higher the chances it will be placed in a block with a matching description, increasing recall. Figure 2.8 (left) presents the distributions of common tokens per description, showing that descriptions in central datasets feature many more common

Table 2.6: Statistics and evaluation of blocking methods.

	D1	D2	D3	D4	D5	D6	D7
Token blocking statistics:							
blocks	1,639,962	122,340	1,019,501	57,085	2,109	40,304	18,553
comparisons (clean-clean)	$1.68 \cdot 10^{12}$	$3.74 \cdot 10^{10}$	$6.56 \cdot 10^{11}$	$2.39 \cdot 10^{10}$	$8.72 \cdot 10^{8}$	$2.91 \cdot 10^{8}$	$1.45 \cdot 10^{9}$
RR (clean)	88.51%	86.81%	96.03%	89.48%	92.09%	54.50%	94.04%
comparisons (dirty)	$5.56 \cdot 10^{12}$	$3.68 \cdot 10^{12}$	$4.27 \cdot 10^{12}$	$4.02 \cdot 10^{12}$	$1.01 \cdot 10^{12}$	$2.05 \cdot 10^{9}$	$2.35 \cdot 10^{11}$
RR (dirty)	90.08%	90.87%	92.67%	90.01%	97.48%	−58.85%	82.93%
common tokens per entity (median)	4	3	4	2	0	19	12
Attribute clustering blocking statistics:							
blocks	5,602,644	150,293	1,673,855	39,587	3,724	43,716	19,148
comparisons	$3.22 \cdot 10^{11}$	$4.20 \cdot 10^{9}$	$1.84 \cdot 10^{11}$	$1.43 \cdot 10^{9}$	$7.13 \cdot 10^{8}$	$2.13 \cdot 10^{8}$	$8.38 \cdot 10^{8}$
RR	97.80%	98.52%	98.89%	99.37%	93.54%	66.80%	96.55%
common tokens in common att. clusters per entity (median)	4	0	4	2	0	19	11
attribute clusters	16,886	124	2,106	6	8	4	8
attributes per attribute cluster (median)	2	142	9	4,261	3,946	3	3.5
Prefix-Infix(-Suffix) blocking statistics:							
blocks	3,266,798	141,517	789,723	45,403	2,098	N/A	18,442
comparisons (clean-clean)	$1.10 \cdot 10^{12}$	$1.78 \cdot 10^{10}$	$2.75 \cdot 10^{11}$	$2.30 \cdot 10^{9}$	$4.08 \cdot 10^{8}$	N/A	$1.28 \cdot 10^{9}$
RR (clean)	92.48%	93.72%	98.34%	98.99%	96.30%	N/A	94.72%
comparisons (dirty)	$4.39 \cdot 10^{12}$	$3.45 \cdot 10^{12}$	$5.34 \cdot 10^{12}$	$3.32 \cdot 10^{12}$	$1.76 \cdot 10^{12}$	N/A	$2.23 \cdot 10^{11}$
RR (dirty)	92.16%	91.44%	90.84%	91.76%	95.59%	N/A	83.78%
Recall:							
Token blocking (clean-clean)	98.38%	92.46%	95.52%	87.76%	72.13%	99.92%	99.54%
Token blocking (dirty)	98.38%	89.99%	94.85%	87.95%	77.34%	99.92%	99.54%
Attribute clustering blocking	97.31%	68.42%	92.10%	76.84%	71.11%	99.55%	99.54%
Prefix-Infix(-Suffix) blocking (clean-clean)	100%	91.71%	87.68%	95.44%	68.17%	N/A	99.54%
Prefix-Infix(-Suffix) blocking (dirty)	100%	89.25%	87.06%	95.50%	74.12%	N/A	99.54%
Precision:							
Token blocking (clean-clean)	$1.56 \cdot 10^{-6}$	$1.00 \cdot 10^{-6}$	$2.49 \cdot 10^{-6}$	$1.30 \cdot 10^{-6}$	$1.13 \cdot 10^{-6}$	$1.21 \cdot 10^{-4}$	$1.18 \cdot 10^{-6}$
Token blocking (dirty)	$3.64 \cdot 10^{-7}$	$5.14 \cdot 10^{-9}$	$3.78 \cdot 10^{-7}$	$1.05 \cdot 10^{-8}$	$1.29 \cdot 10^{-9}$	$7.51 \cdot 10^{-5}$	$6.5 \cdot 10^{-9}$
Attribute clustering blocking	$8.51 \cdot 10^{-6}$	$5.76 \cdot 10^{-6}$	$1.01 \cdot 10^{-5}$	$1.41 \cdot 10^{-5}$	$1.35 \cdot 10^{-6}$	$1.52 \cdot 10^{-4}$	$1.97 \cdot 10^{-6}$
Prefix-Infix(-Suffix) blocking (clean-clean)	$1.87 \cdot 10^{-6}$	$2.19 \cdot 10^{-6}$	$5.72 \cdot 10^{-6}$	$1.01 \cdot 10^{-5}$	$2.05 \cdot 10^{-6}$	N/A	$1.19 \cdot 10^{-6}$
Prefix-Infix(-Suffix) blocking (dirty)	$6.04 \cdot 10^{-7}$	$8.21 \cdot 10^{-9}$	$2.77 \cdot 10^{-7}$	$1.23 \cdot 10^{-8}$	$6.99 \cdot 10^{-10}$	N/A	$6.84 \cdot 10^{-9}$
F-measure:							
Token blocking (clean-clean)	$3.13 \cdot 10^{-6}$	$2.00 \cdot 10^{-6}$	$9.72 \cdot 10^{-7}$	$2.06 \cdot 10^{-8}$	$1.94 \cdot 10^{-9}$	$2.42 \cdot 10^{-4}$	$2.35 \cdot 10^{-6}$
Token blocking (dirty)	$7.28 \cdot 10^{-7}$	$1.03 \cdot 10^{-8}$	$7.55 \cdot 10^{-7}$	$2.10 \cdot 10^{-8}$	$2.59 \cdot 10^{-9}$	$1.50 \cdot 10^{-4}$	$1.30 \cdot 10^{-8}$
Attribute clustering blocking	$1.70 \cdot 10^{-5}$	$1.15 \cdot 10^{-5}$	$2.02 \cdot 10^{-5}$	$2.82 \cdot 10^{-5}$	$2.69 \cdot 10^{-6}$	$3.04 \cdot 10^{-4}$	$3.94 \cdot 10^{-6}$
Prefix-Infix(-Suffix) blocking (clean-clean)	$3.75 \cdot 10^{-6}$	$4.38 \cdot 10^{-6}$	$9.98 \cdot 10^{-7}$	$2.02 \cdot 10^{-5}$	$4.11 \cdot 10^{-6}$	N/A	$2.38 \cdot 10^{-6}$
Prefix-Infix(-Suffix) blocking (dirty)	$1.21 \cdot 10^{-6}$	$1.64 \cdot 10^{-8}$	$5.55 \cdot 10^{-7}$	$2.46 \cdot 10^{-8}$	$1.40 \cdot 10^{-9}$	N/A	$1.37 \cdot 10^{-8}$
H3R:							
Token blocking (clean-clean)	93.18%	89.55%	95.77%	88.61%	80.90%	70.53%	97.04%
Token blocking (dirty)	94.05%	90.43%	93.75%	88.97%	86.25%	N/A ($RR < 0$)	90.48%
Attribute clustering blocking	97.55%	80.76%	95.37%	86.66%	80.80%	79.95%	98.16%
Prefix-Infix(-Suffix) blocking (clean-clean)	96.09%	92.70%	92.70%	97.18%	79.83%	N/A	97.07%
Prefix-Infix(-Suffix) blocking (dirty)	95.92%	90.33%	88.91%	93.59%	83.50%	N/A	90.98%

tokens than those in peripheral ones[7]. For example, 41.43% and 44% of descriptions in $D1$ and

[7]We take the median values and not the averages, as the latter are highly influenced by extreme values and our distributions are skewed.

*D*3, respectively, have 2-4 common tokens, while for *D*2, *D*4 and *D*5 the corresponding values are 33.26%, 26.03% and 12.97%. We observe a big difference in the distributions of *D*6 and *D*7, which contain many more common tokens per description, to those of the other datasets. Only 23.75% of the descriptions in *D*6 and 44% of the descriptions in *D*7 have 0 - 10 common tokens. Figure 2.8 (left) also shows that a big number of descriptions in peripheral datasets, do not share any common tokens. Those are hints that the recall of token blocking in central datasets is higher than in peripheral datasets.

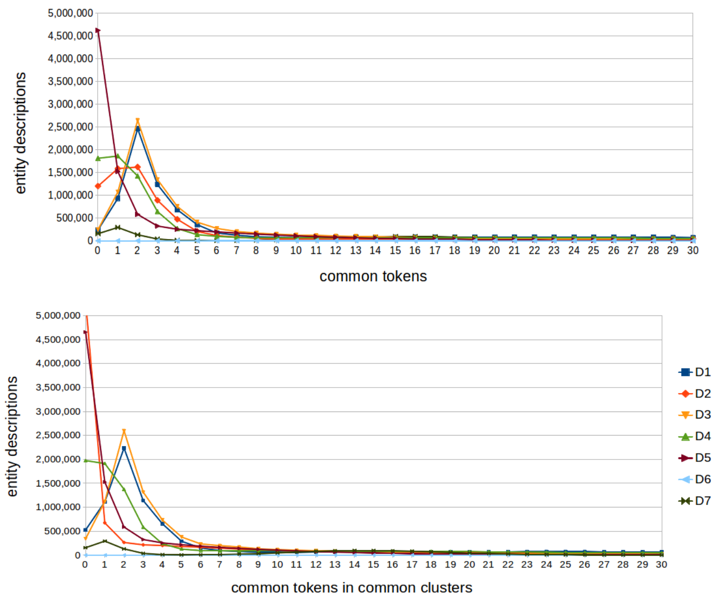


Figure 2.8: Common tokens (top) and common tokens in common clusters (bottom) per entity description distributions for *D*1-*D*7.

Indeed, *D*6 is the dataset with the highest recall (99.92%) and the highest number of common tokens per entity (19), while *D*5 is the dataset with the lowest recall (72.13%) and number of common tokens per entity (0). There is a big difference in the number of common tokens in *D*6, compared to *D*1 and *D*3, which is not reflected by their small difference in recall. Due to the high ratio of matches to non-matches in *D*6 (Table 2.5), descriptions in this dataset have many common tokens and this leads to high recall.

Attribute clustering blocking: The goal of attribute clustering is to improve the precision of token blocking, while retaining its recall as much as possible (it cannot have higher recall). To do this, it restricts the number of attributes on which descriptions, featuring a common token, should be compared. Comparisons between descriptions that do not share a common token in a common attribute cluster, are discarded. Hence, descriptions with many common tokens in common clusters are more likely to be matched. Figure 2.8 (top) presents the distributions of the number of common tokens in common attribute clusters per entity. It shows a clearer distinction between central and peripheral datasets than Figure 2.8 (top); the descriptions in central datasets have many more common tokens in common clusters, while many descriptions in peripheral datasets do not have any common token in a common cluster. This occurs, because values in the descriptions of peripheral datasets are much less similar than those of central datasets, leading to a bad clustering of the attributes and, thus, to lower recall. In fact, $D6$ is the dataset with the highest recall (99.55%) and the highest number of common tokens in common attribute clusters per entity (19). On the other hand, $D2$ and $D5$, which have the lowest recall values (68.42% and 71.11%) also have the lowest number of common token in common attribute clusters per entity (0).

In central datasets ($D1$, $D3$, $D6$, $D7$), many, small clusters of similar attributes are formed, as the values of the descriptions are similar. This leads to a minor (or zero, in $D7$) decrease in recall, compared to token blocking, while it significantly improves its precision (even by an order of magnitude in $D3$). $D1$ forms many (16,886), small attribute clusters (of 2 attributes in the median case), since in most cases there is a 1-1 mapping between the attributes of the KBs that compose it. These clusters contain the same attribute used by the two versions of DBpedia.

However, this approach has a substantial impact on recall in peripheral datasets ($D2$, $D4$, $D5$), even if it still improves precision in all datasets (even by an order of magnitude for $D4$). The descriptions in those datasets have few common tokens, in the first place, which leads to a bad clustering of attributes; few clusters of many attributes, not similar to each other, are formed. Hence, if we make the blocking criterion of token blocking stricter, by also considering attributes, then the more distinct attributes used per entity type, the more difficult it is for an entity description, to be placed in a common block with a matching description. For *BTC12Rest* (part of $D2$), the ratio between attributes and entity types (last row of Table 2.4) is the highest (15.7), leading to a great impact on recall (-24.04%). This dataset has the biggest number of data sources that compose it and many different attribute names can be used for the same purpose; hence, big attribute clusters are formed. *LOCAH* (part of $D5$) only has 3.5 attributes per entity type. Thus, the recall of attribute clustering blocking is insignificantly reduced (-1.02%), compared to that of token blocking.

Prefix-Infix(-Suffix) blocking: Prefix-Infix(-Suffix) blocking is built on the premise that many URIs contain useful information. Its goal is to extend token blocking and improve both its recall, by also considering the subject URIs of the descriptions, and its precision, by disregarding some unneeded tokens in the URI values (either in the prefix or suffix). It achieves good recall values in KBs with similar naming policies in the URIs, as in $D4$, part of which is *BBCmusic*, which also has Wikipedia as a source. However, it misses many matching pairs of descriptions, when the names of the URIs do not contain useful information, as in $D3$ that uses random strings as ids, or have different policies, as in $D5$, which uses concatenations of tokens, without delimiters, as URIs. The recall of $D1$ is 100%, because the dataset is constructed this way; it consists of two versions of the same KB, DBpedia, and the URIs appearing as subjects in *Infoboxes* are only those URIs that also appear as subjects in *BTC12DBpedia*. PIS is not applicable (marked N/A) to $D6$, since URIs have been replaced with numerical ids in the provided dataset. In $D7$, recall is the same as in the other blocking methods, since the matches can be found by tokens in the literal values of the descriptions.

Missed Matches (FNs)

A non-negligible number of matching pairs of descriptions do not share any common tokens at all. Such descriptions, constituting the false negatives of token blocking, should not be assumed faulty, or noisy. We distinguish two different sources of information that can be exploited for successfully placing descriptions of missed matches in common blocks:

1. The matches of their neighbors: Given that a description can have, as one of its values, another description, neighborhoods of related descriptions are formed, spinning the Web of data. The knowledge of matches in the neighbors of a description is valuable for correctly matching this description. For example, if a description e_{10} is related to e_1, e_{20} is related to e_2, and we know that e_{10} and e_{20} match, then we can use this knowledge as a hint that e_1 and e_2 could possibly match, too.

2. A third, matching description: In dirty datasets (typically peripheral), which are composed of KBs that potentially contain duplicate descriptions, a description e_1 could have more than one matching description, e.g., both e_2 and e_3. Identifying one of these matches, e.g., (e_1, e_3), knowing that (e_2, e_3) is a match, leads to also identify the missing match (e_1, e_2).

Table 2.7 provides details about the number and the characteristics of false negative pairs of descriptions, and the set of individual descriptions that constitute these pairs[8].

We focus first on the neighbors of these descriptions, namely descriptions that appear in their values. We found that almost all the descriptions in the false negatives have at least one neighbor (second row of Table 2.7). Looking more thoroughly, we counted the percentage of descriptions in false negatives that have at least one neighbor belonging to the ground truth (third row of Table 2.7). In all cases, this percentage is more than 10% and goes up to 58% for $D4$. This means that, not only do these descriptions have neighbors, but many of these neighbors can be matched to other descriptions in the same entity collection as well. Then, we counted the percentage of descriptions in false negatives that have neighbors, which have already been matched to another description (fourth row of Table 2.7). This percentage is over 20% in most datasets, while it reaches up to 51.84% for $D4$. Finally, we counted the percentage of false negative pairs, whose descriptions have neighbors, which match to each other (fifth row of Table 2.7). This percentage is 0 for $D1$, as matches in this dataset are defined as descriptions that have the same subject URI. However, in some peripheral datasets ($D2$, $D4$), examining the matches of the neighbors of the descriptions is meaningful.

[8] $D6$ is excluded, as it does not contain any descriptions with neighbors and $D7$ is excluded, as it only yields 7 missed matches.

Table 2.7: Characteristics of the missed match of token blocking.

	D1	D2	D3	D4	D5
FNs	25,419	3,176	87,672	2,886	303
descriptions in FNs, with neighbor(s)	99.64%	100%	99.99%	100%	100%
descriptions in FNs, with neighbor(s) in ground truth	22.60%	53.94%	36.43%	58.36%	11.57%
descriptions in FNs, with neighbor(s) with an identified match	20.94%	48.54%	34.05%	51.84%	7.59%
FNs with matching neighbors	0%	24.81%	0.38%	37.63%	0%
FNs with common, identified matches	0%	25.35%	10.54%	0.14%	8.58%

Another useful piece of information for the missed matches of dirty datasets is whether their descriptions have been correctly matched to a third description. The last row of Table 2.7 quantifies this statistic, showing that there are datasets, both peripheral (*D*2, *D*5) and central (*D*3), for which this kind of information could, indeed, be useful.

The information of Table 2.7 is lost when we only consider the tokens in the values of the descriptions to create the blocks in a single round, but it could be useful to an iterative method. Iterative blocking [101], based on some initial blocks, aims to identify matches of type (ii), as well as eliminate redundant comparisons. In our experiments, the recall of iterative blocking, given the blocks of token blocking from the dirty dataset with the smallest number of comparisons (*D*6), was the same as that of token blocking (99.92%), since both of its KBs contain no duplicates (Tables 2.4, 2.5), but the number of comparisons performed was almost half of those suggested by token blocking. We also applied iterative blocking to the dirty dataset with the lowest recall (*D*5), giving the blocks generated by token blocking as input. The process did not terminate within a reasonable amount of time, even so, the recall of iterative blocking was 78.09% after a first pass, whereas the recall of token blocking was 77.34%.

Regarding attribute clustering blocking, it misses the matches that are also missed by token blocking, plus matches that, even if they share common tokens, those tokens appear in the values of attributes in different clusters. The matches missed by prefix-infix(-suffix) blocking are those with no common tokens in their literal values and no common infixes in their URIs.

Non-matches (FPs and TNs)

Next, we examine the ability of blocking methods to identify non-matches, namely their ability to avoid placing non-matching descriptions in the same block. A key statistic for this, regarding the datasets, is the ratio of matches to non-matches (Table 2.5). The higher the ratio, the easier it is for a blocking method to have better precision, as it statistically has better chances of suggesting a correct comparison. *D*6 is the dataset with the highest such ratio and precision, while *D*5 has the lowest ratio and, in most blocking methods, the lowest precision, too. It is clear from Table 2.6 that

Table 2.8: Analysis of 1K sampled match and 1K sampled non-match.

	D1	D2	D3	D4	D5	D7
matches with neighbors	967	956	913	918	859	973
non-matches with neighbors	966	955	912	917	854	973
neighbors of matches (median)	17	80	100	138	121	1
neighbors of non-matches (median)	72	80	105	171	121	1
matches with matching neighbors	862	254	7	766	570	966
non-matches with matching neighbors	32	22	0	0	542	590

attribute clustering is the most precise method, since, in almost every case, it results in the fewest wrong suggestions. On the contrary, the least precise method is token blocking, in all cases. The differences in precision, in some cases even by an order of magnitude, also determine F-measure, since the differences in recall are not that big. All the evaluated methods have very low precision, i.e., the vast majority of suggested comparisons correspond to non-matches. This comes naturally from the fact that matching pairs are only a scintilla of all possible description pairs, as shown in Table 2.5.

Structural Analysis of Matches and Non-matches

To better understand the characteristics of matches versus those of non-matches in the evaluated datasets, we have analyzed sample pairs of matching and non-matching descriptions. In particular, we have taken 1,000 random pairs of matches and non-matches from each dataset and we have focused on their neighbor pairs of descriptions. The results of this analysis are presented in Table 2.8.

First, we counted the number of pairs of descriptions that both have neighbors. We found that those numbers, presented in the first two rows of Table 2.8 for matches and non-matches, respectively, are almost the same. Practically, almost all the pairs of descriptions are linked to other pairs of description, in all datasets. Then, we measured the median number of neighbors (pairs of descriptions) that a match has (Table 2.8, row 3) and the same median number for non-matches (Table 2.8, row 4). Again, there are no significant differences between those two lines. Those numbers vary greatly from dataset to dataset, ranging from 1 (for $D7$) to 171 (for $D4$). Finally, we counted the number of pairs, whose neighbor pairs match. For matches (Table 2.8, row 5), this number is always higher than the corresponding number for non-matches (Table 2.8, row 6). Intuitively, this means that when a match is found, the chances that there is another match in its neighbor pairs are increased.

2.5.3 Performance Results

Table 2.6 shows that all the evaluated methods manage to greatly reduce the number of comparisons that would be required if blocking was not employed, e.g., by one ($D1$-$D4$, $D7$) or two ($D5$) orders of magnitude for token blocking. This is reflected by high RR in all cases. An exception is $D6$, which is much smaller in terms of descriptions and, consequently, comparisons without blocking. Moreover, its descriptions contain many more common tokens than the other datasets, leading to more comparisons per entity. Therefore, token blocking does not save many of the comparisons that would be required without blocking and in $D6$ dirty, it even produces twice as many comparisons.

With respect to $H3R$, we notice that, in general, central datasets have higher scores, i.e., they present a better balance between recall and reduction ratio. This means that in these datasets, comparisons that are discarded by blocking mostly correspond to non-matches, while many of the comparisons discarded by blocking in peripheral datasets correspond to matches. Again, $D6$ has a different behavior, since it initially contains a much smaller number of comparisons and a high ratio of matches to non-matches, so the reduction ratio for this dataset is limited. These measures are not applicable to token blocking, when applied to $D6$ dirty, since in that case the reduction ratio is negative.

2.5.4 Lessons Learned

We now present the key points of our evaluation. *Central* datasets are mostly derived from Wikipedia, from which they extract information regarding an entity. This way, descriptions in such datasets follow similar naming policies and feature many common tokens (Figure 2.8) in the values of semantically similar, or equivalent attributes (see the small size of clusters in Table 2.6). Those are exactly the premises on which the evaluated blocking methods are built.

For these reasons, the recall achieved by token blocking in central datasets is very high (ranges from 99.92% to 94.85%). With the exception of $D6$ (featuring a higher ratio of matching to non-matching descriptions), the precision achieved by token blocking in these datasets ranges from $2.49 \cdot 10^{-6}$ to $3.64 \cdot 10^{-7}$. The gains in precision brought by attribute clustering blocking in central datasets are up to one order of magnitude (for $D3$), with a minor cost on recall (from 0% to 3.42%). Prefix-infix(-suffix) blocking can improve both recall and precision of token blocking for central datasets, as in $D1$, but, it can also deteriorate these values, as in the dirty case of $D3$, which uses random identifiers as URIs, in which recall drops by 7.79% and precision by 26.72%. In a nutshell, many redundant comparisons are suggested by blocking methods in all datasets (see precision and F-measure in Table 2.6), due to the small ratio of matches to non-matches in the datasets (Table 2.5). However, as $H3R$ reveals, the comparisons that are discarded by blocking in central datasets mostly correspond to non-matches.

On the contrary, descriptions in *peripheral* datasets are more diverse, following different naming policies and sharing few common tokens (Figure 2.8), since they stem from various sources.

The lack of similar values in those descriptions leads to a bad clustering of attributes; big clusters of attributes not similar to each other are formed (Table 2.6).

For these reasons, the recall of token blocking for peripheral datasets drops even to 72.13%, while precision ranges from $1.3 \cdot 10^{-6}$ to $1.29 \cdot 10^{-9}$. The gains in precision brought by attribute clustering blocking (up to one order of magnitude) in peripheral datasets, come at the cost of a drop in recall up to 24.04% (corresponding to 7,421 more missed matches). Prefix-infix(-suffix) blocking can improve the precision of token blocking in peripheral datasets, even by an order of magnitude (for $D4$), or decrease it by an order of magnitude (for $D5$), while it decreases recall from 0.74% to 3.96%, i.e., more matches are missed. In the case of $D4$, in which both KBs use Wikipedia as a source, recall is improved by up to 7.68%. Overall, however, $H3R$ reveals that many of the comparisons that are discarded by blocking in peripheral datasets correspond to matches.

Nevertheless, information for the missed matches, e.g., from the neighborhoods of their descriptions (Table 2.7), sets the ground for a new generation of ER algorithms, which will exploit this information to identify more matches, in an iterative fashion. In Table 2.8, we have shown that even a single match in the neighborhood of a candidate pair is a good match-indication for that pair, too.

2.6 Conclusion

In this chapter, we have first reviewed a wide spectrum of works focusing on blocking. We have divided those works into schema-based, whose blocking keys rely on the existence of a fixed schema, and schema-free, which do no make any assumptions about the schema of the entity descriptions to be matched. Since entity descriptions in the Web of data fall in the latter case, we focus only on schema-free blocking methods. In detail, we evaluated, for the first time, blocking methods for highly heterogeneous entity descriptions in the Web of data. To make this evaluation possible for datasets of large volumes, we have introduced massively parallel adaptations of those methods in MapReduce. Our experimental evaluation shows that entity descriptions met in central LOD datasets feature many common tokens in the values of common attributes, while descriptions met in peripheral datasets have significantly fewer common tokens in attributes that are not necessarily semantically related (see Figure 2.8). Hence, the former can be compared only on their content, i.e., values, (see Table 2.6), while the latter require new blocking methods which can exploit contextual information, e.g., the similarity of neighbor descriptions, linked via different types of relations (see Tables 2.7,2.8).

Moreover, the same candidate matches suggested by blocking, may originate from multiple blocks of different size. This means that we will have to perform the same comparison multiple times, or, find a mechanism that can identify repeated comparisons and perform each of them only once. Even better, this block post-processing mechanism could take advantage of the overlap-positive characteristic of blocking methods and turn this overlap into an advantage: the more the common blocks between two entity descriptions and/or the smaller those blocks are (i.e., the less frequent the common tokens between two descriptions are), the higher the chances that they match. We will explore such mechanism, called meta-blocking, in the next chapter.

The datasets and source code used in this study are publicly available[9], facilitating the benchmarking of blocking methods for entities described in the Web of Data. Existing works in ER benchmarks [43, 51] and evaluation frameworks [47, 64] focus on the similarity of descriptions and how these similarities affect the matching decision of entity resolution; not on blocking, explicitly. In all cases, datasets are built from central KBs of a single domain, e.g., only bibliographic. Those data variations are not adequate to evaluate the blocking algorithms suitable for cross-domain ER involving a large number of entity types. Finally, many works on ontology and instance matching, e.g., [66, 97], have been using the OAEI benchmarks in their evaluations. Typically, those datasets are composed of two ontologies with a 1-1 mapping in their attributes, or even a single ontology, whose instances, i.e., entity descriptions, have some modifications in their values. We have included and analyzed one of those benchmarks in this study.

[9] `csd.uoc.gr/~vefthym/minoanER/`

Chapter 3
Meta-Blocking

3.1 Introduction

The main characteristic of overlap-positive blocking methods is that they trade a large number of repeated and unnecessary comparisons between non-matching entities in their effort to achieve high recall. Meta-blocking [82] is a post processing of a collection of blocks aiming to balance the tradeoff between the achieved reduction ratio and recall. It essentially discards all repeated comparisons from a collection of blocks as well as reduces the number of unnecessary comparisons.

In more detail, the functionality of Meta-blocking consists of two logical steps[1]. The first one transforms the input block collection B into the *blocking graph* G, whose nodes correspond to the entities grouped in B, and edges connect the co-occurring entities, denoting suggested comparisons. Only one edge between two entities is maintained regardless of the number of blocks they co-occur in, thus eliminating all repeated comparisons. For instance, applying token blocking to the entities in Figure 3.1(a), yields the blocks of Figure 3.1(b), in which all matches, e_1-e_3 and e_2-e_4, are placed in at least one common block. The blocking graph G extracted from those blocks is shown in Figure 3.1(c). The repeated comparison e_1-e_3, suggested by blocks b_1 and b_2, is discarded in G, since there is only one edge connecting e_1 with e_3.

The second step associates every edge with a weight, proportional to the likelihood that the adjacent entities are matching, an evidence given by the degree of overlap between the sets of blocks in which those entities have been placed. Low-weighted edges are less likely to correspond to matches, so they are pruned. The pruned blocking graph G' is then transformed into a new block collection B' by creating a new block for every retained edge. For instance, G' in Figure 3.1(d) is derived from G in Figure 3.1(c) by discarding the edges with (Jaccard) weight lower than the average one (1/4). Given that the only matches are e_1-e_3 and e_2-e_4, the blocks b_5, b_6, b_7 and b_8 in Figure 3.1(b) contain 8 unnecessary comparisons. In total, the resulting blocks in Figure 3.1(e) contain 5 comparisons, of which only 3 are unnecessary. Compared to the initial blocks B in Figure 3.1(b), the comparisons entailed by the final blocks B' of Figure 3.1(e) were reduced by 62% without any impact on recall.

[1]Those steps do not need to be explicitly implemented, as we will explain in the following sections; they only describe the logic of this process.

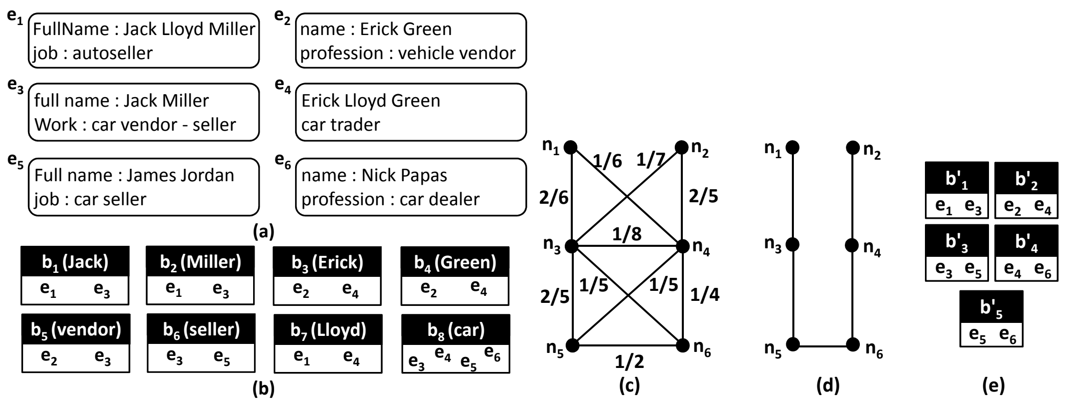


Figure 3.1: (a) A set of heterogeneous entity description, (b) the overlap-positive block collection derived from them using token blocking, (c) the respective blocking graph that uses Jaccard similarity for edge weights, (d) one of the possible pruned blocking graphs, and (e) the restructured block collection after Meta-blocking.

The time complexity of serialized Meta-blocking is quadratic to the size of the input blocks; it relies on the number of comparisons suggested by the input block collection, which define the number of edges that need to be weighted and then pruned in the blocking graph [82]. The reason is that each edge $< v_i, v_j >$ is weighted after computing the intersection of the sets of blocks associated with the descriptions e_i and e_j, while additional computations (e.g., the total number of edges, the number of comparisons suggested by the common blocks) may be required by different weighting schemes, as we will see in the next section. Thus, even as a preprocessing step for ER, sequential Meta-blocking is a heavy computational task with serious scalability limitations at the scale of Web data.

To overcome these limitations, we adopt the MapReduce programming model for parallelizing Meta-blocking and scaling its techniques to voluminous entity collections met in the Web of data, providing exactly the same qualitative results. The basic idea of parallel Meta-blocking is that we process each block of the input block collection in parallel, allowing us to incrementally compute the edge weights, for different node-, or edge-partitions of the blocking graph, depending on the parallelization strategy. When two large blocks are processed by the same computational node, the processing of those blocks will probably become the bottleneck of parallel Meta-blocking. Therefore, we try to distribute the load of each computational node in a balanced way, optimizing the usage of the available computational resources. Our goal is to minimize the total execution time of Meta-blocking, making it as close as possible to the time required to process the largest block (still quadratic with respect to the size of the largest block). In our experiments, we show how parallel Meta-blocking can reduce the time required to process the blocks and get the final matching results from 14 days to 95.5 minutes.

In summary, the main contributions of this chapter, which have been published in [32,33], are:

- We extend the distinction of blocking methods into atomic and composite, to Meta-blocking: extending the blocking graph, we define the *disjunctive blocking graph*, which captures multiple types of matching evidence, allowing the conceptual modeling of composite blocking.

- We implement Meta-blocking in MapReduce using 3 alternative parallelization strategies. The first one explicitly targets the blocking graph, which builds and stores all the edges along with their weights. This is the most intuitive approach, but it bears a significant I/O cost that becomes the bottleneck, when building very large blocking graphs. The second strategy offers a more efficient implementation, by enriching the input block collection with indexing information used for computing the weights of the edges, without building and storing any of them explicitly. Still, enriching the block collection with such information requires an additional MapReduce job, which bears an additional cost. The third strategy is independent of the blocking graph. For every entity, it aggregates the bag of all entities that co-occur in at least one block, and then, it derives on the fly the edge weight that corresponds to each neighbor from its frequency in the co-occurrence bag. Since the weights are computed on the fly, this is the most efficient strategy, when the edge weights do not need to be re-computed before pruning.

- We introduce a novel load balancing algorithm, called *MaxBlock*, in order to avoid potential bottlenecks associated with the computation-intensive parts of our MapReduce functions. MaxBlock exploits the highly skewed distribution of block sizes in overlap-positive collections in order to split them in partitions of equivalent computational cost (i.e., total number of comparisons). We experimentally compare MaxBlock with existing approaches, including a state-of-the-art algorithm that serves a similar purpose, and demonstrate that our approach has significant qualitative and quantitative benefits.

- We verify the scalability of our techniques through a thorough experimental evaluation over the four largest, real datasets that have been applied to Meta-blocking. We show that the speedup of our parallel implementation is close to the ideal, linear case, in which doubling the available resources results in half the execution time.

The rest of the chapter is organized as follows: In Section 3.2, we introduce the formal model of Meta-blocking. In Section 3.3, we overview the state-of-the-art works in the field. In Section 3.4, we provide the overview of our parallelization strategies for Meta-blocking. Section 3.5 introduces our MaxBlock load balancing algorithm. In Section 3.6, we experimentally evaluate the parallelization strategies and MaxBlock, and we conclude the chapter in Section 3.7.

3.2 Formal Meta-blocking Model

The functionality of Meta-blocking consists of two logical steps. The first one transforms the input block collection B into the *blocking graph* G. The second step associates every edge $< v_i, v_j >$ with a weight $w_{i,j} \in \Re$ that is proportional to the likelihood that the adjacent entities are matching. Note that this is only the logical representation of Meta-blocking, which does not need to be implemented as such. Formally:

Definition 3.1 (Blocking Graph)**.** *Given a block collection B^{key}, produced by an indexing function h_{key}, the blocking graph for an entity collection $\mathcal{E}$, is a graph $G = (V, E, w)$, where there is a node $v_i \in V$ for each description $e_i \in \mathcal{E}$, and an edge $< v_i, v_j >\in E$ for each pair $e_i, e_j \in \mathcal{E}$ for which $o_{key}(e_i, e_j) = `true'$. $w : E \rightarrow \Re$ is a labeling function applied to the edges of G.*

Table 3.1 summarizes the notation used in this chapter. The tokens used in an entity collection $\mathcal{E}$ is given by $tokens(\mathcal{E}) = \bigcup_{e_i \in \mathcal{E}} tokens(e_i)$, where $tokens(e_i)$ is the set of tokens used in the values of an entity description $e_i \in \mathcal{E}$. The number of edges $|E|$ of the blocking graph, corresponds to the number of unique comparisons suggested by blocking, i.e., at most $\sum_{b_k \in B} ||b_k||$, where $||b_k||$ is the number of comparisons suggested by a block b_k, which is quadratic to the size of b_k. When b_k has been created by token blocking for a token t, $||b_k|| = EF_{\mathcal{E}}(t) \cdot (EF_{\mathcal{E}}(t) - 1)/2$, where $EF_{\mathcal{E}}(t) = |\{e_i | e_i \in \mathcal{E}, t \in tokens(e_i)\}|$ is the Entity Frequency of t in an entity collection $\mathcal{E}$. The use of EF is inspired by the Document Frequency DF, which has been used in Information Retrieval to define the Inverse Document Frequency (IDF), i.e., the specificity of a term t in a document corpus D, as $IDF(t, D) = \log \frac{|D|}{DF(t)}$ [94], where $DF(t)$ is the number of documents containing the term t. We will refer to the set of blocks $h_{key}(e_i)$ in which e_i has been placed as B_i, and to the common blocks of e_i and e_j as $B_{i,j}$, for brevity.

Next, we will present alternative ways for weighting (i.e., labeling) the edges of a blocking graph (i.e., pairs of entity descriptions), which have been introduced in [82, 84]. All of them have the nice property that they only rely on the input blocks in order to define the weights, without any additional information.

• *Common Blocks Scheme* (CBS) captures the fundamental property of overlap-positive block collections that the more blocks two entities share, the more likely they are matching. The labeling function used by CBS is:

$$w_{CBS}(e_i, e_j) = |B_{i,j}|. \tag{3.1}$$

Then, CBS corresponds to the overlap similarity measure between two descriptions e_i, e_j, unnormalized as in [102], when token blocking is used:

$$sim_{CBS}(e_i, e_j) = |tokens(e_i) \cap tokens(e_j)|.$$

Intuitively, two descriptions are likely to match, if they share many common tokens.

• *Enhanced Common Blocks Scheme* (ECBS) improves CBS by discounting the contribution

Table 3.1: Summary of the notation used in Meta-blocking.

Name	**Symbol**
Set of tokens used in the values of an entity description e_i	$tokens(e_i)$
Set of tokens used in the values of any description in an entity collection $\mathcal{E}$	$tokens(\mathcal{E})$
Entity Frequency of a token t in an entity collection $\mathcal{E}$	$EF_{\mathcal{E}}(t)$
Block collection (a set of blocks)	B
Block collection size (number of blocks)	$\|B\|$
Block collection cardinality (number of comparisons)	$\|\|B\|\|$
Blocks containing e_i	B_i
Number of blocks containing e_i	$\|B_i\|$
Block with id i	b_i
Block size (number of entities)	$\|b_i\|$
Block cardinality (number of comparisons)	$\|\|b_i\|\|$
Blocks shared by e_i and e_j	$B_{i,j}$
Number of blocks shared by e_i and e_j	$\|B_{i,j}\|$
Blocking graph	G
Node in G corresponding to e_i	v_i
Edge in G	$< v_i, v_j >$
Set of edges in G	E
Weight of $< v_i, v_j >$	$w_{i,j}$
Degree of v_i (number of adjacent nodes)	$\|v_i\|$

of entities participating in many blocks. The labeling function used by ECBS is:

$$w_{ECBS}(e_i, e_j) = w_{CBS}(e_i, e_j) \cdot \log\frac{|B|}{|B_i|} \cdot \log\frac{|B|}{|B_j|}. \tag{3.2}$$

Again, when token blocking is used, ECBS corresponds to the following similarity measure between e_i and e_j:

$$sim_{ECBS}(e_i, e_j) = sim_{CBS}(e_i, e_j) \cdot \log\frac{|tokens(\mathcal{E})|}{|tokens(e_i)|} \cdot \log\frac{|tokens(\mathcal{E})|}{|tokens(e_j)|}.$$

Intuitively, two descriptions are likely to match, if they share many common tokens, and each of the descriptions doesn't have too many tokens.

• *Aggregate Reciprocal Comparisons Scheme* (ARCS) is based on the assumption that the fewer the comparisons suggested by the common blocks of two entities, the more likely it is that those entities are matching. The labeling function used by ARCS is:

$$w_{ARCS}(e_i, e_j) = \sum_{b_k \in B_{i,j}} \frac{1}{||b_k||}. \tag{3.3}$$

In the extreme case in which all the blocks in the input block collection contain two entities only, i.e., $\forall b_k \in B, ||b_k|| = 1$, then w_{ARCS} yields the same weights as w_{CBS}. We can further refine the contribution of each common block, by introducing a log in the denominator, which makes the size of each common block less important than in Equation 3.3 and the number of common blocks more important:

$$w_{ARCS}(e_i, e_j) = \sum_{b_k \in B_{i,j}} \frac{1}{\log_2(||b_k|| + 1)}. \tag{3.4}$$

Then, ARCS corresponds to the following similarity measure between two descriptions e_i and e_j, when token blocking is used:

$$sim_{ARCS}(e_i, e_j) = \sum_{t \in tokens(e_i) \cap tokens(e_j)} \frac{1}{\log_2(EF_{\mathscr{E}}(t) \cdot (EF_{\mathscr{E}}(t) - 1)/2 + 1)}.$$

This is similar in logic to using the sum of the IDFs of the common tokens between two documents in a document corpus. Intuitively, two descriptions are likely to match, if they share many, infrequent tokens.

• *Jaccard Scheme* (JS) estimates the portion of blocks shared by two entities. The labeling function used by JS is:

$$w_{JS}(e_i, e_j) = \frac{|B_{i,j}|}{|B_i| + |B_j| - |B_{i,j}|}. \tag{3.5}$$

When token blocking is used, JS corresponds to the Jaccard similarity [54] of the token sets of two entities e_i, e_j:

$$sim_{JS}(e_i, e_j) = \frac{|tokens(e_i) \cap tokens(e_j)|}{|tokens(e_i) \cup tokens(e_j)|}.$$

Intuitively, two descriptions are likely to match, if most of their individual tokens are common.

• *Enhanced Jaccard Scheme* (EJS) improves JS by discounting the contribution of entities involved in too many non-repeated comparisons (i.e., high node degree). The labeling function used by EJS is:

$$w_{EJS}(e_i, e_j) = w_{JS}(e_i, e_j) \cdot \log \frac{|E|}{|v_i|} \cdot \log \frac{|E|}{|v_j|}. \tag{3.6}$$

EJS can be approximated by the following similarity measure between two entities e_i and e_j, when token blocking is used:

$$sim_{EJS}(e_i, e_j) = sim_{JS}(e_i, e_j) \cdot \log \frac{A}{|comp(e_i, \mathscr{E})|} \cdot \log \frac{A}{|comp(e_j, \mathscr{E})|},$$

where $A = \sum_{t \in tokens(\mathscr{E})} EF_{\mathscr{E}}(t) \cdot (EF_{\mathscr{E}}(t) - 1)/2$, and $comp(e_l, \mathscr{E}) = \{e_k | e_k \in \mathscr{E}, tokens(e_l) \cap tokens(e_k) \neq \emptyset\}$ is the subset of descriptions from $\mathscr{E}$ having at least one common token with e_l. Intuitively, two descriptions are likely to match, if most of their common tokens are infrequent (low EF), and their non-common tokens are few and infrequent.

The time efficiency of each labeling function depends on the number and the nature of the computations that need to be performed in order to evaluate the edge weights. w_{CBS} is the easiest function to evaluate, as it only needs to compute the intersection of two sets (of numerical block ids). Additionally, w_{JS} also computes the union of those sets, while w_{ECBS} also needs the total size of the input block collection. Thus, those three labeling functions are very close in terms of computational cost. w_{ARCS}, needs to compute two things: the intersection of two block sets, as well as the cardinality of each block belonging to the intersection, so it is heavier computationally than the previous functions. Finally, w_{EJS} is the heaviest function to evaluate; on top of computing the intersection and the union of two sets of blocks, it also needs to compute the node cardinality of the two entities, i.e., how many other entities share a common block with the entities to be compared.

On the other hand, the *pruning scheme* relies on a *pruning criterion* which can be either weight- or cardinality-based; the former specifies the minimum weight of the retained edges (i.e., a dynamic similarity threshold) and the latter the maximum number of retained edges (i.e., a top-K functionality). Both the minimum weight and the number of retained edges are set dynamically, driven by data statistics. The selected criterion is then combined with a *pruning algorithm*, which is either edge-centric (i.e., global) or node-centric (i.e., local); the former iterates over all edges of the graph to retain the globally best ones and the latter over all edges of the neighborhood to retain the locally best ones. Overall, the main pruning schemes are the following:

• *Weighted Edge Pruning* (WEP) combines the edge-centric algorithm with a global weight threshold that amounts to the average edge weight of the entire blocking graph. That is, it retains all edges with a weight higher than the overall mean one.

• *Cardinality Edge Pruning* (CEP) couples the edge-centric algorithm with a global cardinality threshold equal to: $K = \lfloor \frac{\sum_{b_i \in B} |b_i|}{2} \rfloor$. Thus, it retains the top-K edges of the entire blocking graph.

• *Weighted Node Pruning* (WNP) combines the node-centric pruning algorithm with a local weight threshold that amounts to the average edge weight of each neighborhood.

• *Cardinality Node Pruning* (CNP) combines the node-centric pruning algorithm with a global cardinality threshold equal to: $k = \lfloor \sum_{b_i \in B} |b_i| / |E| - 1 \rfloor$. Thus, it retains the top-k edges of each neighborhood.

The above definitions consider only one type of blocks. However, as we have seen in Chapter 2, composite blocking schemes may also be constructed on different types of blocks. Thus, edges in the corresponding blocking graph may be determined according to a composite co-occurrence conditions while their weights may be extended accordingly.

Definition 3.2 (Disjunctive Blocking Graph). *Given a block collection $B = \bigcup_{h_{key} \in H} B^{key}$, produced by a set of indexing functions H, the disjunctive blocking graph for an entity collection $\mathcal{E}$, is a graph $G = (V, E, \lambda)$, where each node $v_i \in V$ represents a distinct description $e_i \in \mathcal{E}$, and each edge $< v_i, v_j > \in E$ represent a pair $e_i, e_j \in \mathcal{E}$ for which $\mathcal{F}(e_i, e_j) = 'true'$. $\mathcal{F}(e_i, e_j)$ is a disjunction of the atomic co-occurrence functions o^k defined along with H. $\lambda : E \rightarrow R^n$ is a labeling function assigning a tuple $[w^1, \ldots, w^n]$ to each edge $\in E$, where w^k is a weight associated with each co-occurrence function o^k of H.*

We will see an example of applying disjunctive Meta-blocking in Chapter 4.

3.3 Related Work

In this section, we first overview works related to block processing, i.e., processing the results of blocking, before providing the final matches, and then we discuss about how existing works address the challenge of balancing the load of block processing in a distributed environment.

Block Processing

Numerous studies have focused on the problem of *block processing*, whose goal is to discard redundant (both unnecessary and repeated) comparisons in order to enhance the precision of blocking collections. Most of the relevant techniques involve a functionality that operates at the block level, based on coarse-grained characteristics of the input blocking collection, such as the size of blocks: Block Purging [78] a-priori discards oversized blocks like b_8 in Figure 3.1(b), while Block Pruning [78] orders blocks from smallest to largest and terminates their processing as soon as the cost of identifying new matches exceeds a predefined threshold. Both of these methods are equivalent, in logic, to discarding stopwords, i.e., very frequent words which do not offer much information about an entity, such as 'the', 'a', 'to', etc. Those words would only add a big computational cost, without offering much in the similarity evidence that two entities match (they have high EF, as defined in the previous section). [79] proposes a method for discarding all repeated comparisons from any set of blocks. In essence, when two descriptions are compared in a block, this comparison is not performed again in any other block this pair appears.

Such techniques are efficient, but lack in accuracy, as their crude processing cannot control its impact on recall (in terms of matching comparisons).

Similar to Meta-blocking, *Block Filtering* [85], drastically reduces the size of the blocking graph by transforming an overlap-positive block collection B into a new one B' that involves a lower number of comparisons. Instead of using a graph, though, it simply removes every entity from the least important of its blocks. The main assumption is that the larger a block is (i.e., higher $||b_i||$), the less important it is for its entities. In more details, Block Filtering orders the blocks of B in ascending order of cardinality, i.e., suggested comparisons, and retains every entity e_i in the top N_i blocks of B_i (i.e., the N_i smallest blocks that contain e_i). For every entity e_i, this threshold is locally defined as $N_i = \lfloor r \times |B_i| \rfloor$, where $r \in [0,1]$ is the *ratio* of Block Filtering. In this work, we employ Block Filtering as an integral part of our parallelized approach, setting $r = 0.8$. This value was experimentally verified to increase efficiency to a significant extent, pruning at least 50% of the blocking graph's edges, while having a negligible impact on recall [85].

[83] formalizes Meta-blocking as a binary classification task, targeting at identifying edges that correspond to matches and non-matches between their adjacent entity descriptions. To extend the simple pruning rules of the form "if weight < threshold then discard edge" for removing comparisons, [83] works towards assigning representative weights to edges and choosing appropriate thresholds for removing edges by learning composite pruning models from the data. Supervised Meta-blocking composes information about the co-occurring entities into comprehensive

feature vectors. As an example, consider that each edge is associated with a feature vector [a_1, a_2], where a_1 is the number of common blocks shared by the adjacent descriptions, and a_2 is the total number of comparisons contained in these blocks. The resulting feature vectors are fed into a classification algorithm that learns composite rules to effectively distinguish matching and non-matching edges. In our example, a composite rule could be "if $a_1 \leq 2$ and $a_2 > 5$ then discard edge", capturing the intuition that the more blocks two descriptions share and the smaller these blocks are, the more likely the descriptions match. Concerning the set of features annotating the edges of the blocking graph, clearly, using more features may help make the pruning of the non-matching edges more accurate. However, the computational cost for Meta-blocking gets higher.

Load Balancing

A crucial aspect of MapReduce-based ER methods is the load balancing algorithm that distributes evenly the overall workload among the available nodes. Several recent works examine this aspect, with PairRange constituting the best solution so far [62]. In essence, PairRange splits evenly the comparisons of a block collection into a predefined number of partitions, by assigning every comparison to a particular partition id. To this end, it involves a single MapReduce job, whose mapper associates every entity e_i in block b_k with the output key $rid.k.i$, where rid denotes the index of the comparison range, i.e., the partition id. Then, the reducer groups together all entities that have the same rid and block id k, reproducing the comparisons corresponding to partition rid. In our experimental evaluation, we compare MaxBlock with PairRange.

Another approach is BlockSplit [62]. As its name suggests, it splits the bigger blocks into smaller sub-blocks and processes them in parallel, ensuring that every entity is compared to all entities in its sub-block, as well as to the entities of its super-block. BlockSplit has been proven to be less scalable and less generic than PairRange [62]: it needs to process multiple times the entity descriptions of blocks that are split, creating an additional network and I/O overhead. Additionally, it may still lead to unbalanced workload, due to sub-blocks of different size.

A similar approach is followed by the dynamic blocking algorithm in [72]. Instead of perfectly balancing the load, though, its goal is to split large blocks into sub-blocks, "until they are all of tractable size". Yet, we already achieve this goal through Block Filtering, which completely removes large blocks (instead of splitting them into sub-blocks), as it considers them to be of lower importance.

Finally, two more load balancing algorithms were presented in [104]. Both rely on sketches in order to minimize memory consumption; the one aims to improve the space requirements of BlockSplit and the other of PairRange. In our case, though, all load balancing algorithms that were compared in Section 3.6.2 fit easily to the limited memory that is available to a single node. The reason is the optimized representation model, which represents every entity by an integer that denotes its id, while every block consists of a list of integers and is itself identified by a unique integer id.

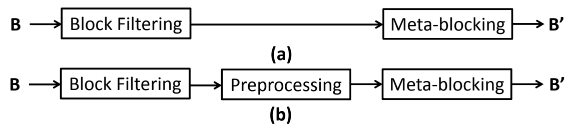


Figure 3.2: (a) The serialized workflow of Meta-blocking, and (b) its parallelized counterpart.

In summary, existing works in block processing suffer from serious accuracy and scalability issues, or they rely on training data, while little work has been done in balancing the load of processing block collections in distributed environments. In this sequel, we will describe a parallel adaptation of unsupervised Meta-blocking in MapReduce, using a novel load balancing algorithm with significant benefits over state-of-the-art algorithms.

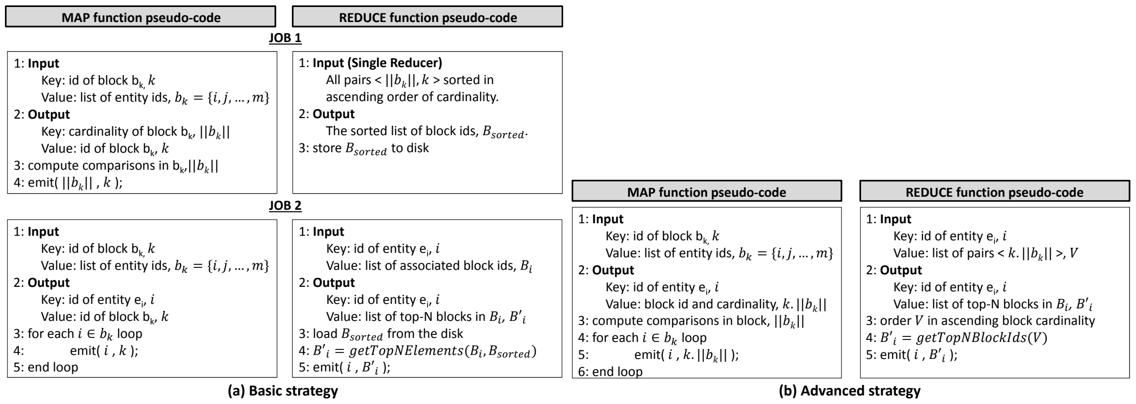
3.4 Scaling Meta-Blocking to Very Large Entity Collections

3.4.1 Approach Overview

In the following, we elaborate on the adaptation of Meta-blocking to MapReduce. The serialized workflow we want to parallelize is depicted in Figure 3.2(a) and consists of two consecutive stages: the first one applies Block Filtering to the input block collection B, while the second one applies Meta-blocking to yield the final, restructured collection B'. The parallelized counterpart is presented in Figure 3.2(b) and consists of three stages. Again, the first one applies Block Filtering to the input block collection and the last one implements Meta-blocking. The only difference is in the second stage, which preprocesses the blocks in order to transform them into a suitable form for parallel Meta-blocking.

We analyze every stage of the parallelized workflow separately, proposing at least two different approaches in each case. The first one applies a basic strategy that relies on a straightforward adaptation, but involves more jobs and higher I/O between the nodes. The other approach(es) correspond(s) to more advanced strategy(ies), reducing the overhead of data exchange through more elaborate processing. In all cases, we provide the pseudo-code of the strategy's functionality and, for the most important strategies, we accompany it with an example that facilitates its understanding.

Section 3.4.2 presents two strategies for the first stage (Block Filtering), while Section 3.4.3 introduces three strategies for the second stage (Preprocessing). The last stage of the parallel workflow applies one of the four pruning algorithms to the output of Preprocessing and yields a set of retained edges; every edge corresponds to a new block that is part of the final, restructured block collection. We examine one of the four pruning algorithms, in Section 3.4.4, and the other three can be found in [33]. Given that the functionality and the complexity of their parallelization depend on the preprocessing strategy, we present three adaptations in every section, each of them corresponding to the output of the previous stage. Finally, that applies to all strategies of the last



MAP function pseudo-code
REDUCE function pseudo-code
JOB 1
JOB 2
(a) Basic strategy
(b) Advanced strategy

(a) Basic strategy — JOB 1, MAP function pseudo-code:

```
1: Input
      Key: id of block b_k, $k$
      Value: list of entity ids, $b_k = \{i, j, \dots, m\}$
2: Output
      Key: cardinality of block b_k, $||b_k||$
      Value: id of block b_k, $k$
3: compute comparisons in b_k, $||b_k||$
4: emit( $||b_k||$ , $k$ );
```

JOB 1, REDUCE function pseudo-code:

```
1: Input (Single Reducer)
      All pairs < $||b_k||, k$ > sorted in
      ascending order of cardinality.
2: Output
      The sorted list of block ids, $B_{sorted}$.
3: store $B_{sorted}$ to disk
```

JOB 2, MAP function pseudo-code:

```
1: Input
      Key: id of block b_k, $k$
      Value: list of entity ids, $b_k = \{i, j, \dots, m\}$
2: Output
      Key: id of entity e_i, $i$
      Value: id of block b_k, $k$
3: for each $i \in b_k$ loop
4:        emit( $i$ , $k$ );
5: end loop
```

JOB 2, REDUCE function pseudo-code:

```
1: Input
      Key: id of entity e_i, $i$
      Value: list of associated block ids, $B_i$
2: Output
      Key: id of entity e_i, $i$
      Value: list of top-N blocks in $B_i$, $B'_i$
3: load $B_{sorted}$ from the disk
4: $B'_i = getTopNElements(B_i, B_{sorted})$
5: emit( $i$ , $B'_i$ );
```

(b) Advanced strategy — MAP function pseudo-code:

```
1: Input
      Key: id of block b_k, $k$
      Value: list of entity ids, $b_k = \{i, j, \dots, m\}$
2: Output
      Key: id of entity e_i, $i$
      Value: block id and cardinality, $k.||b_k||$
3: compute comparisons in block, $||b_k||$
4: for each $i \in b_k$ loop
5:        emit( $i$ , $k.||b_k||$ );
6: end loop
```

REDUCE function pseudo-code:

```
1: Input
      Key: id of entity e_i, $i$
      Value: list of pairs < $k.||b_k||$ >, $V$
2: Output
      Key: id of entity e_i, $i$
      Value: list of top-N blocks in $B_i$, $B'_i$
3: order $V$ in ascending block cardinality
4: $B'_i = getTopNBlockIds(V)$
5: emit( $i$ , $B'_i$ );
```

Figure 3.3: Pseudo-code interpretation of (a) the basic and (b) the advanced strategy for Block Filtering. They employ a global and a local ordering of blocks, respectively.

two stages.

Note that in every stage, special care was taken to minimize the I/O between the independent nodes. Part of this effort focused on *optimizing our representation model.* Apparently, we could use the actual blocking keys and URIs to identify the blocks and the entities, respectively. However, the binary representation of these textual values is much larger than that of numerical identifiers. For this reason, our model relies exclusively on numbers: we enumerate every block and entity, so that they are uniquely identified by an integer id, and represent the edges by the concatenation of the adjacent entity ids. Their weights are naturally represented by real numbers.

3.4.2 Stage 1: Block Filtering

The first stage applies Block Filtering to the input block collection in order to reduce the size of the blocking graph. Central to this procedure is the sorting of blocks in ascending order of cardinality, from the smallest to the largest one. Depending on how this sorting is performed, we present two possible approaches for adapting Block Filtering to MapReduce.

The basic strategy orders once and globally all input blocks, using two MapReduce jobs that exploit the automatic sorting of the input to the `reduce` function. The advanced strategy employs a single MapReduce job that orders locally the blocks associated with every entity at the cost of repeating some computations across the independent nodes.

For both strategies, every `(key, value)` pair of the input corresponds to a block b_k; the key stands for the id of the block, while the value contains the list of the entity ids placed in b_k: `key`=k and `value`=$\{i, j, \dots, m\}$ for b_k=$\{e_i, e_j, \dots, e_m\}$. The output of both strategies comprises the N most important blocks associated with the individual entities. Every key denotes the id of an entity e_i, while the corresponding value contains the list of ids of the blocks still containing e_i: `key`=i and `value`=B'_i.

Basic Strategy

This strategy employs two MapReduce jobs. The first one sorts all blocks globally in ascending order of cardinality, producing the sorted list B_{sorted}. The second job uses B_{sorted} in order to identify the most important blocks for each entity.

The functionality of the first job is outlined in the upper part of Figure 3.3(a). The `map` function receives a block id k along with the entities contained in b_k. It computes the corresponding cardinality, $||b_k||$, and emits a $(||b_k||,\ k)$ pair. All pairs are sorted in descending order of their keys (i.e., cardinalities), before they are forwarded as input to the single `reduce` function. The reducer extracts and stores to the disk the values of the sorted input, i.e., the block ids that form B_{sorted}.

The pseudo-code interpretation of the second job is presented in the lower part of Figure 3.3(a). The `map` function gets the same input as the first job: the id of a block along with the entity ids it contains. For every entity e_i contained in the given block b_k, it emits as output a pair (i, k). MapReduce groups together all pairs having the same key so that the `reduce` function receives as input all block ids assigned to a specific entity e_i (i.e., `key`=i, `value`=B_i). It loads from the disk the sorted list of block ids, B_{sorted}, and uses it to get the ranking position of every block. The N blocks with the highest ranking positions form the list of retained block ids B'_i, which are the emitted as output: `key`=i, `value`=B'_i.

Advanced Strategy

The rationale behind the advanced strategy is to use a single MapReduce job that provides the `reduce` function with the necessary information for sorting the blocks of each entity locally. Its functionality is outlined in Figure 3.3(b). The `map` function gets as input the id and the entities of a block b_k and computes its cardinality, $||b_k||$. For every entity $e_i \in b_k$, it emits a pair with the entity id as the key, while the (composite) value concatenates the id and the cardinality of block b_k: `key`=i and `value`=$k.||b_k||$. The `reduce` function gathers all blocks associated with an entity e_i along with their cardinality. It sorts them in ascending number of comparisons and extracts the top N elements from the resulting list to form B'_i. Similar to the basic strategy, it then emits a pair (i, B'_i).

Example 3.1. *Figure 3.4 illustrates the functionality of the advanced strategy of Block Filtering. For the three entities e_1, e_2 and e_3 of b_1, we emit in the* `Map` *phase a pair with each of them as the* `key` *and $b_1.3$ as* `value`*, since there are three comparisons in this block. In the* `Reduce` *phase, we gather all four pairs having e_1 as key and keep only the top-3 blocks for this entity. Thus, we discard b_7 from the blocks of e_1.*

3.4.3 Stage 2: Preprocessing

The second stage of the parallel Meta-blocking workflow prepares the input that will be processed by the selected pruning algorithm in the third stage. It plays a crucial role, as its output determines

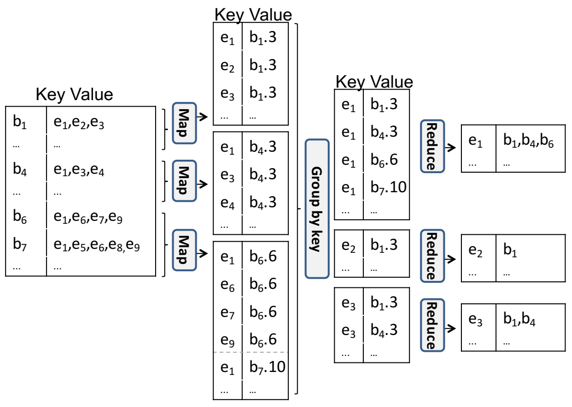


Figure 3.4: An example of the advanced strategy for Block Filtering.

the complexity of the pruning algorithm: the more computations are performed by Preprocessing and are integrated into its output, the simpler is the functionality of the pruning algorithms and vice versa.

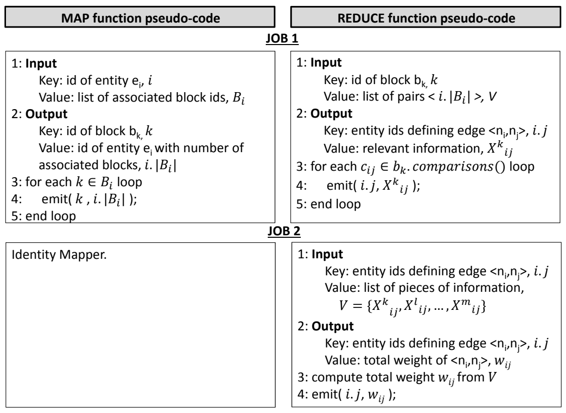
This trade-off gives rise to three different strategies for Preprocessing, which share the same input (i.e., the outcome of Block Filtering), but differ in their output. The *edge-based* strategy explicitly creates the blocking graph, performing all weight computations in order to simplify the functionality of the pruning algorithm. On the flip side, it involves two MapReduce jobs with high I/O that store all edges to the disk. The *comparison-based* strategy defers all weight computations and simply facilitates them by enriching the input of the pruning algorithms with all the necessary information. The *entity-based* strategy facilitates a different approach for weight estimation that does not require any preprocessing. Thus, it simply receives the output of Block Filtering (i.e., the block ids retained per entity) and transforms it into a new block collection. Due to their simplicity, the last two strategies require just one job.

Edge-based Strategy: Explicit Blocking Graph

The pseudo-code interpretation of the edge-based strategy is depicted in Figure 3.5. The first MapReduce job transforms the output of Block Filtering into a block collection. Its `map` function receives as key the id of an entity e_i and as value the list of associated blocks, B_i. It swaps values and keys, emitting for every block $b_k \in B_i$ a pair $(k, i.|B_i|)$, where k and i are the block and the entity id, respectively, while $|B_i|$ denotes the number of blocks containing e_i after Block Filtering. The reason is that $|B_i|$ is the cornerstone for most weighting schemes.

The `reduce` function of the first job groups together all entities contained in a block b_k and is able to reproduce all its comparisons.[2] For every comparison between entities e_i and e_j (c_{ij}), it emits the concatenation of their ids as key and some local information X^k_{ij} as value: `key=`$i.j$ and

[2]Note that a block with just one remaining entity contains no comparison and, thus, no processing is performed.



MAP function pseudo-code | **REDUCE function pseudo-code**

JOB 1

MAP:
```
1: Input
    Key: id of entity $e_i$, $i$
    Value: list of associated block ids, $B_i$
2: Output
    Key: id of block $b_k$, $k$
    Value: id of entity $e_i$ with number of
    associated blocks, $i.|B_i|$
3: for each $k \in B_i$ loop
4:     emit( $k$ , $i.|B_i|$ );
5: end loop
```

REDUCE:
```
1: Input
    Key: id of block $b_k$, $k$
    Value: list of pairs $< i.|B_i| >$, $V$
2: Output
    Key: entity ids defining edge <$n_i$,$n_j$>, $i.j$
    Value: relevant information, ${X^k}_{ij}$
3: for each $c_{ij} \in b_k.comparisons()$ loop
4:     emit( $i.j$, ${X^k}_{ij}$ );
5: end loop
```

JOB 2

MAP:
```
Identity Mapper.
```

REDUCE:
```
1: Input
    Key: entity ids defining edge <$n_i$,$n_j$>, $i.j$
    Value: list of pieces of information,
      $V = \{{X^k}_{ij}, {X^l}_{ij}, \dots, {X^m}_{ij}\}$
2: Output
    Key: entity ids defining edge <$n_i$,$n_j$>, $i.j$
    Value: total weight of <$n_i$,$n_j$>, $w_{ij}$
3: compute total weight $w_{ij}$ from $V$
4: emit( $i.j$, $w_{ij}$ );
```

Figure 3.5: Pseudo-code interpretation of the edge-based Preprocessing strategy, which explicitly creates the blocking graph.

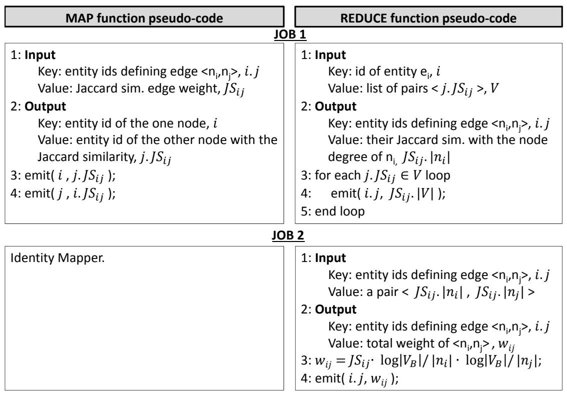
`value`=X^k_{ij}. The information in X^k_{ij} is necessary for estimating the corresponding edge weight and varies, depending on the selected weighting scheme. For ARCS, it comprises the cardinality of block b_k (i.e., X^k_{ij}=$||b_k||$), while for all other schemes it concatenates $|B_i|$ and $|B_j|$ (i.e., X^k_{ij}=$|B_i|.|B_j|$); for CBS, though, it can be empty.

The second job consists of an identity mapper and a `reduce` function that estimates the weight for every edge of the blocking graph. The value list of its input, V, clusters together all local information pertaining to the edge $<v_i, v_j>$ that is specified by the input key. Based on them, the reducer computes the corresponding edge weight w_{ij} from Equations 3.3-3.6. For example, we simply have $w_{ij} = |V|$ for CBS, as the size of the value list equals the number of common blocks, $|B_{ij}|$. As output, the reducer emits a pair with the id and the weight of the edge: `key`=$i.j$ and `value`=w_{ij}.

There is an exception to this strategy, as the EJS weighting scheme requires two additional jobs to be applied to the output of JS. Their goal is to estimate the node degree $|v_i|$ of every entity e_i. The first job counts the edges that are adjacent to each entity, while the second one reassembles all neighboring entities in order to estimate the weight of their adjacent edge according to EJS formula (Equation 3.6). Their functionality is outlined in Figure 3.6.

The first job continues from the Preprocessing of the JS weighting scheme. Its `map` function receives as input an individual edge from the respective blocking graph; the concatenated ids of the adjacent entities form the key, while the value contains the corresponding edge weight. The mapper performs no processing, but just emits two pairs: for each of the two entities, it uses its id as the key and concatenates the id of the other entity with the edge weight to form the value.

In this way, the `reduce` function gathers all edges that correspond to a specific entity e_i. Its input value actually comprises a list with the ids of all neighboring entities appended to the weight of the respective edge. The size of this list equals the degree $|v_i|$ of node v_i that corresponds



MAP function pseudo-code | **REDUCE function pseudo-code**

JOB 1

MAP:
```
1: Input
      Key: entity ids defining edge <n_i,n_j>, $i.j$
      Value: Jaccard sim. edge weight, $JS_{ij}$
2: Output
      Key: entity id of the one node, $i$
      Value: entity id of the other node with the
      Jaccard similarity, $j.JS_{ij}$
3: emit( $i$ , $j.JS_{ij}$ );
4: emit( $j$ , $i.JS_{ij}$ );
```

REDUCE:
```
1: Input
      Key: id of entity e_i, $i$
      Value: list of pairs < $j.JS_{ij}$ >, $V$
2: Output
      Key: entity ids defining edge <n_i,n_j>, $i.j$
      Value: their Jaccard sim. with the node
      degree of n_i, $JS_{ij}.|n_i|$
3: for each $j.JS_{ij} \in V$ loop
4:     emit( $i.j$, $JS_{ij}.|V|$ );
5: end loop
```

JOB 2

MAP:
```
Identity Mapper.
```

REDUCE:
```
1: Input
      Key: entity ids defining edge <n_i,n_j>, $i.j$
      Value: a pair < $JS_{ij}.|n_i|$ , $JS_{ij}.|n_j|$ >
2: Output
      Key: entity ids defining edge <n_i,n_j>, $i.j$
      Value: total weight of <n_i,n_j> , $w_{ij}$
3: $w_{ij} = JS_{ij} \cdot \log|V_B|/|n_i| \cdot \log|V_B|/|n_j|$;
4: emit( $i.j$, $w_{ij}$ );
```

Figure 3.6: Pseudo-code interpretation of the edge-based Preprocessing strategy for the EJS weighting scheme.

to e_i. The reducer emits this information so that the corresponding EJS weights can be computed in the second job: for each of the neighboring entities e_j, it emits a pair with `key`=$i.j$ and `value`=$JS_{ij}.|v_i|$.

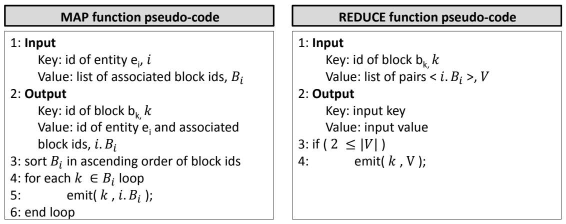
The second job involves an identity mapper so that the reducer gathers both values that pertain to an individual edge $<v_i, v_j>$, namely $JS_{ij}.|v_i|$ and $JS_{ij}.|v_j|$. Having this information, the EJS weight can be derived from the Equation 3.6. This forms the output value, while the ids of the adjacent entities form the output key.

Comparison-based Strategy: Implicit Blocking Graph

This strategy creates the blocking graph implicitly: it enriches the description of the input block collection with the information that is required for detecting all edges and estimating their weights according to the selected scheme. The key to this approach is the idea that every edge $<v_i, v_j>$ of the blocking graph G corresponds to a non-repeated comparison c_{ij} in the block collection B.

A comparison c_{ij} in b_k is *non-repeated* only if it satisfies the Least Common Block Index condition (LeCoBI for short). That is, if the id of b_k equals the least common block id of the entities e_i and e_j: $k = min(B_i \cap B_j)$ [79]. To assess the LeCoBI condition for two entities e_i and e_j, we need to compare the lists of associated blocks, B_i and B_j; for higher efficiency, their elements should be sorted in ascending order of block ids. The comparison-based strategy integrates this information to its output, so that the pruning algorithms carry out all edge and weight computations in the third stage.

This functionality is performed by one MapReduce job, which is outlined in Figure 3.7. The `map` function receives as input the outcome of Block Filtering: the id of an entity e_i as key and the associated blocks B_i as values. First, it sorts B_i in ascending order of block ids. Then, for every block $b_k \in B_i$, it emits its id as the key, while the value concatenates the id of e_i with the



MAP function pseudo-code

```
1: Input
      Key: id of entity e_i, $i$
      Value: list of associated block ids, $B_i$
2: Output
      Key: id of block b_k, $k$
      Value: id of entity e_i and associated
      block ids, $i.B_i$
3: sort $B_i$ in ascending order of block ids
4: for each $k \in B_i$ loop
5:         emit( $k$ , $i.B_i$ );
6: end loop
```

REDUCE function pseudo-code

```
1: Input
      Key: id of block b_k, $k$
      Value: list of pairs $< i.B_i >$, $V$
2: Output
      Key: input key
      Value: input value
3: if ( $2 \leq |V|$ )
4:         emit( $k$ , V );
```

Figure 3.7: Pseudo-code interpretation of the comparison-based Preprocessing strategy, which creates the blocking graph implicitly, enriching the description of the input block with the necessary information for weight estimation.

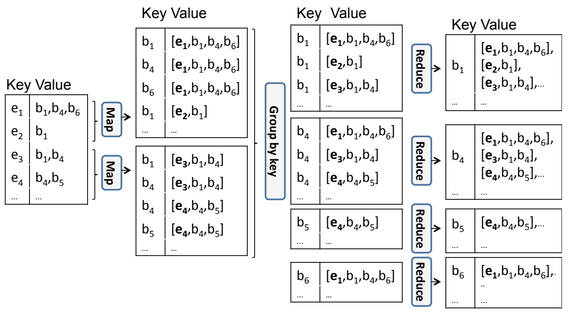
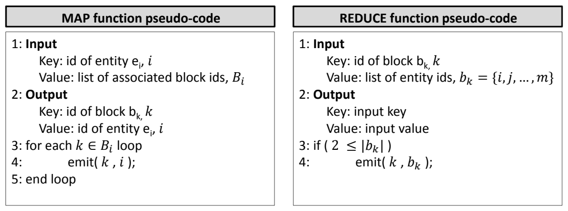


Figure 3.8: An example of the comparison-based strategy for Preprocessing.

entire sorted list B_i: `key`=k, `value`=$i.B_i$. MapReduce then reassembles all blocks, by grouping together all pairs with the same key. The `reduce` function receives as input the entity list of a specific block along with the blocks that are associated with every individual entity; provided that there are at least two entities, it emits the same `(key, value)` pair as output: `key`=k and `value`=$\{i.B_i, j.B_j, \ldots, m.B_m\}$.

Example 3.2. *Figure 3.8 provides an example of this functionality. For each block b_1, b_4 and b_6, to which e_1 belongs, we emit a pair with their block ids as* `key` *and e_1, concatenated with b_1, b_4, b_6 as* `value` *in the* `Map` *phase. In the* `Reduce` *phase, all the entities of b_1 are grouped together (i.e., e_1, e_2 and e_3), each accompanied with the block ids in which it belongs. We just concatenate them and emit them as the* `value` *of the* `key`*=b_1.*

There are two exceptions to this functionality, because the weighting schemes ARCS and EJS need additional information for estimating their edge weights. The former requires the cardinality of every block contained in B_i. This can be easily embedded into the output of the previous stage (Block Filtering), such that for ARCS, the input is not only a list of associated blocks for each entity, but also the cardinality of each such block. Then, the rest of the process is the same as in the other weighting schemes. In contrast, the information required by EJS can only be derived from an elaborate processing (see [33] for details).



MAP function pseudo-code

```
1: Input
      Key: id of entity e_i, $i$
      Value: list of associated block ids, $B_i$
2: Output
      Key: id of block b_k, $k$
      Value: id of entity e_i, $i$
3: for each $k \in B_i$ loop
4:          emit( $k$ , $i$ );
5: end loop
```

REDUCE function pseudo-code

```
1: Input
      Key: id of block b_k, $k$
      Value: list of entity ids, $b_k = \{i, j, \dots, m\}$
2: Output
      Key: input key
      Value: input value
3: if ( $2 \le |b_k|$ )
4:          emit( $k$ , $b_k$ );
```

Figure 3.9: Pseudo-code interpretation of the entity-based strategy for Preprocessing, which does not use the blocking graph.

Entity-based Strategy: No Blocking Graph

This strategy is fundamentally different from the others in the sense that it does not require the blocking graph. Its functionality revolves around the individual entities such that for every entity e_i, it estimates the weights of its co-occurring entities with a single iteration over the contents of the associated blocks B_i. To facilitate this procedure, the Preprocessing stage simply transforms the output of Block Filtering into a block collection.

This is performed with a single MapReduce job that is presented in Figure 3.9. The `map` function receives the id of an entity e_i as input key and the associated blocks B_i as input value. For every block $b_k \in B_i$, it simply emits its id k as the key and the id of the entity i as value. Then, MapReduce groups together all pairs with the same key, thus reassembling all blocks. In more detail, the `reduce` function receives as input key the id k of a specific block $b_k = \{e_i, e_j, \dots, e_m\}$, while the input value comprises the ids of the corresponding entities: `value`=$\{i, j, \dots, m\}$. Without any further processing, it emits the same `(key, value)` pair as output.

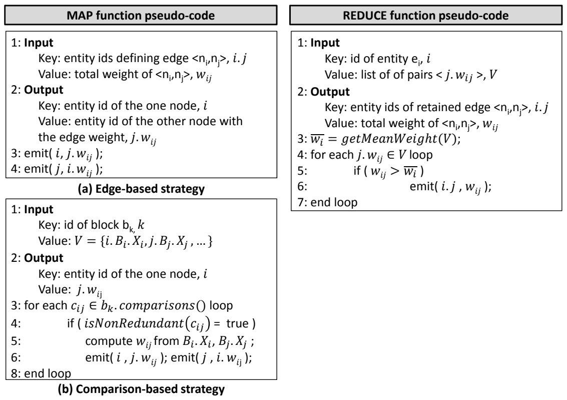
3.4.4 Stage 3: Pruning (WNP)

WNP refines the blocking graph by processing every node neighborhood independently of the others. For every node, it discards the incident edges that have a weight lower than the average edge weight of the neighborhood. We propose three parallelization strategies for this algorithm – one for every Preprocessing strategy. They all require a single MapReduce job. You can refer to [33] for the rest of the pruning schemes, as well as to [37] for the scalable top-k MapReduce algorithm, used for the implementation of CEP.

Edge-based Strategy

The functionality of this strategy is outlined in Figure 3.10 – together with the comparison-based strategy. They share the same reducer, but they differ in the mapper, due to the different input they receive.

In more detail, Figure 3.10(a) depicts the edge-based `map` function. It takes as input key the id of an individual edge $<v_i, v_j>$ $(i.j)$ and as input value the corresponding weight (w_{ij}). To ensure



MAP function pseudo-code

```
1: Input
      Key: entity ids defining edge <n_i,n_j>, $i.j$
      Value: total weight of <n_i,n_j>, $w_{ij}$
2: Output
      Key: entity id of the one node, $i$
      Value: entity id of the other node with
      the edge weight, $j.w_{ij}$
3: emit( $i, j.w_{ij}$ );
4: emit( $j, i.w_{ij}$ );
```

(a) Edge-based strategy

```
1: Input
      Key: id of block b_k, $k$
      Value: $V = \{i.B_i.X_i, j.B_j.X_j, \ldots\}$
2: Output
      Key: entity id of the one node, $i$
      Value: $j.w_{ij}$
3: for each $c_{ij} \in b_k.comparisons()$ loop
4:         if ( $isNonRedundant(c_{ij})$ = true )
5:             compute $w_{ij}$ from $B_i.X_i, B_j.X_j$ ;
6:             emit( $i$ , $j.w_{ij}$ ); emit( $j$ , $i.w_{ij}$ );
8: end loop
```

(b) Comparison-based strategy

REDUCE function pseudo-code

```
1: Input
      Key: id of entity e_i, $i$
      Value: list of of pairs < $j.w_{ij}$ >, $V$
2: Output
      Key: entity ids of retained edge <n_i,n_j>, $i.j$
      Value: total weight of <n_i,n_j>, $w_{ij}$
3: $\overline{w_i} = getMeanWeight(V)$;
4: for each $j.w_{ij} \in V$ loop
5:         if ( $w_{ij} > \overline{w_i}$ )
6:                 emit( $i.j$ , $w_{ij}$ );
7: end loop
```

Figure 3.10: Pseudo-code interpretation of (a) the edge-based and (b) the comparison-based strategy for WNP. They share the same `reduce` function.

that each reducer gathers all edges adjacent to a specific node, it emits two `(key, value)` pairs – one for each of the adjacent entities. In each case, the key contains one of the entity ids (i or j), while the value concatenates the other entity id with the edge weight ($j.w_{ij}$ or $i.w_{ij}$).

The `reduce` function in Figure 3.10 receives as input key the id i of an entity e_i that defines a neighborhood in the blocking graph G. Its input value comprises the adjacent node/entity ids concatenated with the respective edge weights. From them, it estimates the average weight of the neighborhood, $\bar{w}_i$, in Line 3. Then, in Lines 4-7, it iterates over all adjacent edges and for every edge $<v_i, v_j>$ with a weight higher than the average one, it emits a pair ($i.j$, w_{ij}).

Comparison-based Strategy

The `map` function of this strategy appears in Figure 3.10(b). It operates on the enriched description of an individual block b_k: the input key contains its id (k), while the input value is of the form `value` = $\{i.B_i.X_i, j.B_j.X_j, \ldots\}$; that is, it comprises the corresponding entity ids, the blocks ids associated with each entity and the local information required by the selected weighting scheme. For EJS, this information contains the node degree (X_i=$|v_i|$), for ARCS it contains the cardinalities of the blocks in B_i (X_i=$\{||b_j|| : b_j \in B_i\}$) and for all other weighting schemes it is empty. The mapper iterates over all comparisons in the given block (Line 3). For every non-repeated comparison, it computes the corresponding edge weight from the associated block ids and the local information X_i of the weighting scheme (Lines 4-5). Then, it emits two `(key, value)` pairs, one for each of the adjacent entities (Line 6) – just like the mapper in Figure 3.10(a).

Example 3.3. *Figure 3.11 shows an example that applies this functionality in combination with the JS weighting scheme to the output of Figure 3.8. In the `map` function, for the comparison e_1-e_2, we emit the pairs $(e_1, e_2.1/3)$ and $(e_2, e_1.1/3)$. In the `reduce` function, we group all the pairs with `key`=e_1 and calculate, for this group, a local weight threshold (e.g., 1/3). Then, for the group of e_1,*

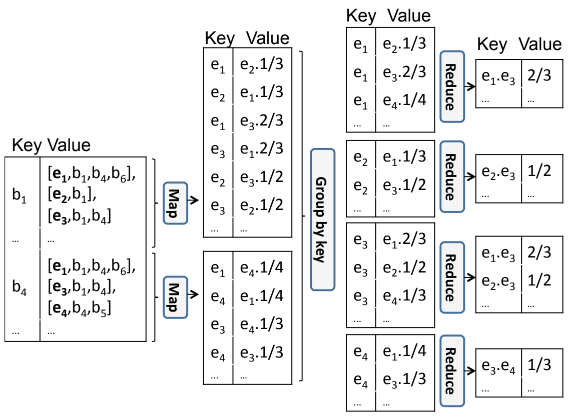


Figure 3.11: An example of the comparison-based strategy for WNP, using the JS weighting scheme.

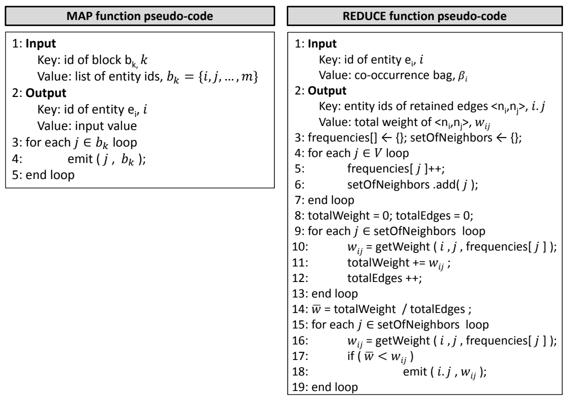
we emit only the pairs with a weight higher than 1/3, i.e., e_1*-*e_3*, which has a weight of 2/3.*

Entity-based Strategy

The outline of this strategy appears in Figure 3.12. Its `map` function receives as input key the id k of block b_k and as input value the entity ids contained in b_k. For every entity $e_i \in b_k$, it simply emits its id as key (i.e., `key`=i) and the entire block b_k as value (i.e., `value`=b_k). In this way, the reducer aggregates the *co-occurrence bag* of entity e_i, i.e., the ids of all entities that share at least one block with e_i. The frequency of an entity e_j in this bag amounts to $|B_{ij}|$, the number of blocks it shares with e_i. This is the core information required by all weighting schemes for estimating the corresponding edge weight w_{ij}.

Based on this rationale, the `reduce` function estimates the edge weights using two data structures, which are initialized in Line 3: the array $frequencies$, which gathers the number of appearances of each entity, and the set $setOfNeighbors$, which aggregates the ids of the distinct co-occurring entities. For every entity in the co-occurrence bag, the reducer updates its frequency in the array and adds it to the set of neighbors (Lines 4-7). Subsequently, it estimates the weights of the edges incident to e_i from the distinct neighbors in $setOfNeighbors$ (Lines 9-10). At the same time, it derives the average weight of the neighborhood, $\bar{w}$, with the help of two counters (Lines 8 & 11-14). The final loop in Lines 15-19 repeats the estimation of edge weights and retains those exceeding $\bar{w}$; the ids of their adjacent entities are emitted as keys and their weights as values.

Note that after the loop in Lines 4-7, $setOfNeighbors$ contains the id of the neighborhood's center, e_i. Given that e_i co-occurs with itself in all blocks, its edge weight would be equal (or close) to 1 for all weighting schemes. To avoid retaining the meaningless comparison $c_{i,i}$ and to avoid distorting the weight threshold $\bar{w}$, we remove i from $setOfNeighbors$ at the end of the loop. For ease of presentation, we have excluded this operation from the outline of the reducer in



MAP function pseudo-code

```
1: Input
      Key: id of block b_k, $k$
      Value: list of entity ids, $b_k = \{i, j, \dots, m\}$
2: Output
      Key: id of entity e_i, $i$
      Value: input value
3: for each $j \in b_k$ loop
4:          emit ( $j$ , $b_k$ );
5: end loop
```

REDUCE function pseudo-code

```
1: Input
      Key: id of entity e_i, $i$
      Value: co-occurrence bag, $\beta_i$
2: Output
      Key: entity ids of retained edges <n_i,n_j>, $i.j$
      Value: total weight of <n_i,n_j>, $w_{ij}$
3: frequencies[] ← {}; setOfNeighbors ← {};
4: for each $j \in V$ loop
5:          frequencies[ $j$ ]++;
6:          setOfNeighbors .add( $j$ );
7: end loop
8: totalWeight = 0; totalEdges = 0;
9: for each $j \in$ setOfNeighbors loop
10:         $w_{ij}$ = getWeight ( $i$ , $j$ , frequencies[ $j$ ] );
11:         totalWeight += $w_{ij}$ ;
12:         totalEdges ++;
13: end loop
14: $\overline{w}$ = totalWeight / totalEdges ;
15: for each $j \in$ setOfNeighbors loop
16:         $w_{ij}$ = getWeight ( $i$ , $j$ , frequencies[ $j$ ] );
17:         if ( $\overline{w} < w_{ij}$ )
18:                 emit ( $i.j$ , $w_{ij}$ );
19: end loop
```

Figure 3.12: Pseudo-code interpretation of the entity-based strategy for WNP.

Figure 3.12.

For the same reason, we have simplified the use of the function $getWeight()$ in Lines 10 and 16. In practice, its arguments depend on the selected weighting scheme:

• For CBS, it simply needs the array of frequencies as input, since w_{ij}=$frequencies[j]$.

• For ECBS and JS, it additionally requires the number of blocks containing e_i and e_j. This information is provided by an array that contains the number of blocks for all input entities. Due to its small size, this array can be loaded in memory in all available nodes.

• For EJS, $getWeight()$ additionally requires the node degree corresponding to every entity. This is equal to the number of non-repeated comparisons involving every entity and is computed through an additional MapReduce job. This job has almost the same functionality as Figure 3.12, but its `reduce` function stops at Line 7, only emitting the size of the set of neighbors for each entity (without counting the frequencies).

• For ARCS, $getWeight()$ requires only the cardinality of the blocks shared by every pair of entities, and not the frequency of their co-occurrence. Given that the reducer receives a list of whole blocks in its input value, the cardinality of each such block and the weight of each co-occurring entity can be directly computed in the first for loop (starting at Line 4). The rest of the process remains the same.

3.5 Load Balancing

A typical bottleneck in MapReduce algorithms is the unbalanced workload that is assigned to the map or reduce tasks in each MapReduce job. In practice, data follow a skewed distribution, which results in groups of data being significantly larger than others. The map or reduce tasks that process these larger groups need substantially more time to finish, determining the efficiency of the whole job. Load balancing, indeed, affects both phases of the MapReduce job, as the reduce phase cannot start processing the output of the map phase, until all map tasks have finished, and the job is not finished unless all reduce tasks are completed.

3.5.1 Default Load Balancing

The default load balancing implementation of Hadoop is a hash-based algorithm, which assigns each group of data, determined by the output key of the map phase, to a bucket in a hash table. The buckets correspond to data partitions, each of which is input to a distinct reduce task. Consequently, the number of data partitions is equal to the number of reduce tasks. Notice that a reduce task can be assigned to more than one keys, since a bucket in a hash table corresponds to more than one hashed keys. This means that the default hashing-based load balancer of Hadoop, given a good hash function, can achieve a very good distribution in the number of keys that each partition (reduce task) will receive. However, in skewed data, this does not guarantee a balanced workload.

For example, assume that we have $p = 10$ partitions, $i = 100$ distinct keys $k_1, ..., k_i$, each corresponding to a word, ordered in descending frequency, and each key k_i has $|k_i|$ values, where the distribution of $|k_i|$ abides by Zipf's law. Assuming we have 1,000 values in total, i.e., $\sum |k_i| = 1,000$, then $|k_1| = 1,000 \cdot \frac{1}{\sum_i 1/i} \approx 192$, $|k_2| = |k_1|/2 \approx 96, \ldots, |k_{100}| = |k_1|/100 \approx 2$. The default balancer of Hadoop would ideally assign $i/p = 10$ keys per partition. Thus, the partition that will receive the most frequent key k_1, associated with 1/5 of the total values, will also receive 9 more keys, and it is highly likely that this partition will be one of the slowest to process.

In the case of entity resolution algorithms, the imbalance of the workload is even greater, as the keys typically correspond to block ids, and the values correspond to entities in those blocks, which have to be compared. Hence, the workload of each reduce task is quadratic to the number of input values it receives. In the previous example, the total number of comparisons would be $\sum |k_i| \cdot (|k_i| - 1)/2 \approx 64,500$, while the biggest block k_1 would yield 18,336 comparisons, i.e., approximately 1/3 of the total comparisons.

3.5.2 MaxBlock Load Balancing

To address this issue, we developed a specialized algorithm for load balancing, named *MaxBlock*. Our goal is to split the input blocks into partitions with a balanced number of comparisons. In order to ensure better results, the number of partitions is determined dynamically. Intuitively, our load balancing strategy is to assign the biggest block to a partition of its own and set the number of comparisons in this partition as the upper threshold of comparisons for every other partition. Then, we create a new partition and keep adding blocks to this new partition, until this threshold is reached. When the threshold is reached, we create a new partition, and continue this process until all blocks have been assigned to a partition.

The functionality of MaxBlock is outlined in Algorithm 1. It sorts the block collection in descending cardinality (Line 1) and removes the first and largest block, b_0 (Line 2). The maximum computational cost of each partition, $maxCost$, is set equal to the cardinality of b_0 (Line 3). A par-

Algorithm 1: MaxBlock

```
Input: $B$ the current block collection
Output: $P$ the set of block partitions
$B' \leftarrow$ sort($B$);    // sort in descending cardinality
$b_0 \leftarrow B'$.remove(0); // remove largest block
$maxCost \leftarrow ||b_0||$; // max comparisons per partition
$P_0 \leftarrow \{b_0\}$;     // first partition
$Q \leftarrow \{P_0\}$;    // priority queue, sorting partitions in ascending cost
while $B' \neq \{\}$ do      // while not empty
    $b_i \leftarrow B'$.remove(0);  // remove first block
    $P_{head} \leftarrow Q$.poll();   // get lowest cost partition
    $totalCost \leftarrow ||b_i|| + P_{head}$.currentCost();
    if $totalCost \leq maxCost$ then
        $P_{head} \leftarrow P_{head} \cup \{b_i\}$;  // add to partition
    else
        $P_i \leftarrow \{b_i\}$;   // create new partition
        $Q$.add($P_i$);  // add to queue
    $Q$.add($P_{head}$);  // place back to queue
    if $B' = \{\}$ then // if all blocks were processed
        $P_{head} \leftarrow Q$.poll();  // get smallest partition
        if isRemnantCluster($P_{head}$) = $true$ then
            $B' \leftarrow B' \cup P_{head}$;  // re-process its blocks
            $maxCost \leftarrow maxCost + P_{head}$.currentCost() / $|Q|$;
        else
            $Q$.add($P_{head}$);
return $Q$;
```

tition is created for b_0 (Line 4) and placed in a priority queue Q, keeping partitions in ascending order of comparisons (Line 5). Subsequently, our algorithm iterates over the remaining blocks and examines whether the current block fits into the partition at the head of the queue, P_{head} (Lines 6-10); that is, it checks whether their combined cardinality is lower than $maxCost$. If so, the current block is added to P_{head} (Line 11); otherwise, it is placed in a new partition that is added to the queue (Lines 13-14). Then, P_{head} is placed back to Q (Line 15).

Example 3.4. *Figure 3.13 abstractly presents the functionality of MaxBlock. After sorting the input blocks in descending order (Figure 3.13 (a)), we create a new partition and place the largest block (block 1) in this new partition (Figure 3.13 (b)). The number of comparisons in this partition sets the threshold for the maximum number of comparisons allowed in the remaining partitions. Then, moving to the next biggest block (block 2), since it doesn't fit in the existing partition, as it violates the maximum comparisons threshold (Figure 3.13 (c)), we create again a new partition and place block 2 in it (Figure 3.13 (d)). The same process continues (Figure 3.13 (e)-Figure 3.13 (i)), until all the blocks have been placed in a partition. At the end, the number of comparisons per partition is similar.*

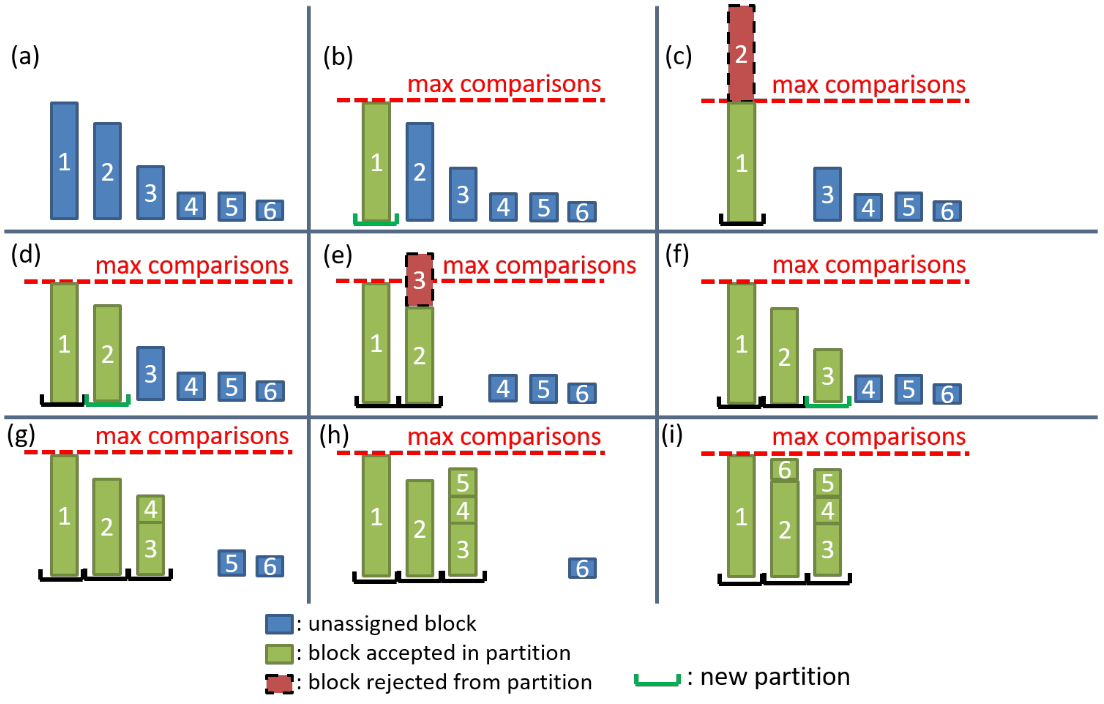


Figure 3.13: An example of running MaxBlock for load balancing.

As we demonstrate in the experimental evaluation, all partitions share practically the same computational cost, except for the smallest one, which merely covers a small fraction of $maxCost$. Yet, it contains the vast majority of the blocks, with each one involving a handful of comparisons. This is called *remnant cluster* and it corresponds to the tail of the power-law distribution of block cardinalities. To achieve a perfectly flat distribution of costs, our algorithm distributes the blocks of the remnant cluster to the other partitions.

This functionality is performed by the second `if` statement in Algorithm 1. Line 16 checks whether all blocks have been placed into a partition, thus terminating the first iteration. Lines 17 and 18 examine whether the smallest partition is a remnant cluster, i.e., whether it contains more than 50% of all blocks and their total computational cost is lower than 90% of $maxCost$. In case both conditions are satisfied, the blocks of the remnant cluster are put back into the processing list (Line 19); in addition, $maxCost$ is updated so that their computational cost is evenly split among the other partitions (Line 20). In case of a negative check, the smallest partition is placed back into the priority queue and the process is terminated.

On the whole, the time complexity of MaxBlock is determined by the sorting of blocks and the use of the priority queue, whose operations cost $O(\log|B|)$ per block. Therefore, the overall time complexity is $O(|B|\log|B|)$, which means that MaxBlock scales well to large block collections, involving a negligible overhead, as shown in Section 3.6. For example, the actual cost of sorting the blocks is the cost for sorting up a few million of integer values, each representing a block cardinality. This operation does not require more than a few seconds.

MaxBlock Implementation

The results of MaxBlock are fed to the Partitioner class, which is responsible for assigning a reduce task to each key. We have overridden the default Partitioner to just send each key to the partition that MaxBlock has defined for this key. This approach is primarily used to balance the functions with quadratic time or space complexity. The former case involves functions that iterate over all comparisons in a block. For the edge-based strategy, this is the `reduce` function in the first job for Stage 2 - Preprocessing (Figure 3.5). For the comparison-based strategy, this case applies to all `map` functions for Stage 3 - Meta-blocking (e.g., Figure 3.10(b)).

Quadratic space complexity appears in the case of the entity-based strategy, where the bottleneck is the I/O overhead of its `map` function in Stage 3: the size of its output is quadratic with respect to number of entities in the input value, since the whole block is emitted for each of the contained entities. As a result, most mappers have to write only a few intermediate key-value pairs, while those that deal with the bigger blocks have to emit a much larger bulk of data. To address this issue, we use MaxBlock to balance the output of entity-based Preprocessing (Figure 3.9). Our goal is to split the blocks into partitions with equal size of representation in bytes. This can be easily done by redefining the cost of a block as the number of bytes that are required for the compressed representation of its entities (i.e., after sorting them in ascending id and replacing every id by its difference with the previous one).

3.6 Experiments

The goal of our experimental analysis is threefold: *(i)* to demonstrate that our approaches scale well to large block collections stemming from Web data, *(ii)* to compare the relative time efficiency of the edge-, comparison- and entity-based strategies, and *(iii)* to assess the relative time efficiency of the various Meta-blocking configurations.

We begin with the setup of our experimental analysis in Section 3.6.1. In Section 3.6.2, we present a comparison between the default balancer and MaxBlock. In Section 3.6.3, we show the time efficiency of all strategies for the four pruning schemes in combination with the five weighting schemes; we also discuss their relative time efficiency in view of a similar comparison in the case of the serialized workflow. Section 3.6.4 analyzes the scalability of the comparison-based and entity-based strategies, while Section 3.6.5 elaborates on the qualitative performance of the Meta-blocking techniques. We conclude with a discussion on the findings of our experiments in Section 3.6.6.

Table 3.2: The datasets employed in our experiments.

	$D_{dbpedia}$		$D_{freebase}$	
	D_1	D_2	D_1	D_2
Entities $\lvert\mathscr{E}\rvert$	1,190,733	2,164,040	3,157,726	4,204,942
Triples	$1.69{\cdot}10^7$	$3.50{\cdot}10^7$	$1.42{\cdot}10^8$	$3.90{\cdot}10^7$
Attribute Names	30,757	52,554	37,825	11,108
Triples per Entity	14.19	16.18	44.84	9.29
Matches	892,579		1,347,266	
BF Comparisons	$2.58{\cdot}10^{12}$		$1.33{\cdot}10^{13}$	

3.6.1 Setup

All approaches were implemented in Java, version 7, using Apache Hadoop[3] version 1.2.0 on a cluster[4] with 15 Ubuntu 12.04.3 LTS servers, one master and 14 slaves, each having 8 AMD 2.1 GHz CPUs and 8 GB of RAM. Each node can run 4 map or reduce tasks simultaneously, assigning 1024 MB to each task. The available disk space amounted to 4 TB and was equally partitioned among the 15 nodes. For Load Balancing, we employed the default mechanism of Hadoop for the `map` and `reduce` functions that involve a processing of linear complexity. For those involving a quadratic complexity, we distributed the relevant blocks to the available nodes using MaxBlock, as explained in Section 3.5.2.

Datasets. To evaluate the performance of our approaches, we employ the largest datasets that have been applied to Meta-blocking. Their technical characteristics appear in Table 3.2.

$D_{dbpedia}$ involves entities stemming from two snapshots of the DBpedia[5] Infoboxes in English, which chronologically differ by 2 years – D_1 corresponds to version 3.0rc and D_2 to version 3.4. In total, they comprise 3.3 million entities, of which less than 900,000 are common (i.e., they have the same URL). This dataset has been previously employed in the literature [78, 80–82, 92]. The second dataset, $D_{freebase}$, contains entities from the Billion Triple Challenge 2012[6]. In this case, D_1 encompasses the entities from DBpedia and D_2 the entities from Freebase[7]. For both KBs, we have disregarded all URIs that appear in just one triple so as to avoid noisy entity descriptions. In total, there are 7.4 million entities, of which 1.3 million are common according to the *owl:sameAs* statements.

Given that both datasets comprise two individually clean (i.e., duplicate-free) entity collections, D_1 and D_2, they are inherently suitable for *Clean-Clean ER*. In our experiments, we use both datasets for *Dirty ER*, as well, by merging D_1 and D_2 into a single dirty entity collection that contains matches in itself. The ground truth is provided by existing owl:sameAs links between Freebase and DBpedia for $D_{freebase}$. Since $D_{dbpedia}$, involves two different snapshots of DBpedia, we

[3]They are also compatible with more advanced frameworks, such as Apache Spark and Apache Flink.
[4]provided by GRNET's ~okeanos (`https://okeanos.grnet.gr`)
[5]`http://dbpedia.org`
[6]`https://km.aifb.kit.edu/projects/btc-2012`
[7]`https://www.freebase.com`

Table 3.3: The block collections that were given as input to Meta-blocking.

	$D_{dbpedia}$		$D_{freebase}$	
	DB_C	DB_D	FR_C	FR_D
Task	Clean-Clean ER	Dirty ER	Clean-Clean ER	Dirty ER
$\|B\|$	1,239,424	1,499,534	1,309,145	4,522,222
$\|\|B\|\|$	$4.23 \cdot 10^{10}$	$8.00 \cdot 10^{10}$	$1.05 \cdot 10^{11}$	$2.19 \cdot 10^{11}$
BPE	15.30-16.08	14.79	75.55-4.43	40.12
$Recall$	0.999	0.999	0.979	0.944
$Precision$	$2.11 \cdot 10^{-5}$	$1.12 \cdot 10^{-5}$	$1.26 \cdot 10^{-5}$	$5.82 \cdot 10^{-6}$

consider matching descriptions those having the same subject URI.

We used token blocking [78, 81] in order to derive overlap-positive block collections from the entity descriptions of the two datasets. We also applied Block Purging [81] to the original blocks in order to discard the extremely large ones that contain almost half the input entities. The technical characteristics of the resulting blocks appear in Table 3.3. In total, we have four block collections, two for each ER task, that vary significantly in their characteristics, for example, in the number of blocks per entity (BPE).

Measures. To assess the *effectiveness* and *efficiency* of the (restructured) block collections, we employ the same measures as the ones in Chapter 2, i.e., Recall, Precision, RR, and $H3R$. To assess the *time efficiency* of (Meta-)blocking methods, we use the *Overhead Time* ($OTime$). This is the time in minutes that intervenes between receiving an overlap-positive block collection as input and returning the restructured blocks as output. The lower its value is, the more time-efficient is the corresponding method.

3.6.2 Load Balancing

In this section, we examine the performance of load balancing with respect to the computationally most intensive functions of the three strategies for parallel Meta-blocking, i.e., the functions with quadratic time or space complexity.

Remember that quadratic time complexity appears in the `reduce` function of the first Preprocessing job for the edge-based strategy (see Figure 3.5) as well as in all `map` functions of Stage 3 for the comparison-based strategy (see Figure 3.10(b)). All these functions iterate over all comparisons in the input blocks in order to estimate the corresponding edge weights. As a result, load balancing aims to split the original block collection into disjoint partitions with (ideally) the same *partition cardinality*, i.e., the same total number of comparisons in the blocks of the partition; every partition is then assigned to one of the available nodes for its processing.

We compare the performance of MaxBlock with two baseline methods: the default balancer of Hadoop and *PairRange*. To compare the two baseline methods with MaxBlock, we consider the distribution of the partition cardinalities they produce. We actually summarize these distributions through their minimum, maximum, median and mean partition cardinalities. The closer

Table 3.4: The distribution of partition cardinalities produced by the default load balancer of Hadoop, PairRange and MaxBlock.

		Partitions	Min Card.	Max Card.	Median Card.	Average Card.	**St. Dev. Card.**
	Default	223	$4.18 \cdot 10^7$	$8.20 \cdot 10^7$	$5.83 \cdot 10^7$	$5.87 \cdot 10^7$	$\mathbf{7.59 \cdot 10^6}$
DB_C	PairRange	442	$2.71 \cdot 10^7$	$2.71 \cdot 10^7$	$2.71 \cdot 10^7$	$2.71 \cdot 10^7$	**0.48**
	MaxBlock	442	$2.71 \cdot 10^7$	$2.71 \cdot 10^7$	$2.71 \cdot 10^7$	$2.71 \cdot 10^7$	**0.48**
	Default	223	$7.88 \cdot 10^7$	$1.48 \cdot 10^8$	$9.59 \cdot 10^7$	$9.68 \cdot 10^7$	$\mathbf{9.98 \cdot 10^6}$
DB_D	PairRange	378	$5.74 \cdot 10^7$	$5.74 \cdot 10^7$	$5.74 \cdot 10^7$	$5.74 \cdot 10^7$	**0.19**
	MaxBlock	378	$5.74 \cdot 10^7$	$5.74 \cdot 10^7$	$5.74 \cdot 10^7$	$5.74 \cdot 10^7$	**0.19**
	Default	1,674	$3.21 \cdot 10^4$	$2.35 \cdot 10^8$	$2.02 \cdot 10^7$	$3.14 \cdot 10^7$	$\mathbf{3.01 \cdot 10^7}$
FR_C	PairRange	2,042	$1.45 \cdot 10^7$	$1.45 \cdot 10^7$	$1.45 \cdot 10^7$	$1.45 \cdot 10^7$	**0.48**
	MaxBlock	2,042	$1.45 \cdot 10^7$	$1.45 \cdot 10^7$	$1.45 \cdot 10^7$	$1.45 \cdot 10^7$	**0.48**
	Default	1,119	$9.81 \cdot 10^6$	$9.81 \cdot 10^7$	$9.01 \cdot 10^7$	$5.84 \cdot 10^7$	$\mathbf{4.64 \cdot 10^7}$
FR_D	PairRange	1,735	$3.76 \cdot 10^7$	$3.76 \cdot 10^7$	$3.76 \cdot 10^7$	$3.76 \cdot 10^7$	**0.27**
	MaxBlock	1,735	$3.76 \cdot 10^7$	$3.76 \cdot 10^7$	$3.76 \cdot 10^7$	$3.76 \cdot 10^7$	**0.27**

these measures are to each other, the more balanced is the workload assigned to each node. We applied all approaches to the input of Stage 2 of the parallelized workflow, i.e., after applying Block Filtering to the original block collections (see Figure 3.2(b)). The outcomes of our experiments appear in Table 3.4.

Note that PairRange receives the number of ranges (partitions) as input from the user. This requires the user to manually inspect the data at hand, which is cumbersome. In our experiments, we gave PairRange an unfair advantage by using the same number of ranges as those in MaxBlock. As a result, we observe that the two algorithms produce identical sets of partitions. Most importantly, though, their partitions exhibit a practically constant distribution of cardinalities across all datasets: all four measures have identical values, while the standard deviation of the distribution is lower than 1. This means that the partitions differ by a handful of comparisons in the worst case.

In contrast, the default balancer yields distributions with much larger variance. For DB_C and DB_D, it yields a normal distribution, as the median and the average cardinalities almost coincide, lying close to the middle of the maximum and the minimum ones. The standard deviation is an order of magnitude lower than the other measures, thus indicating minor differences in the computational cost of the various partitions. However, the performance of the default balancer aggravates in the case of FR_C and FR_D, where the standard deviation is almost equal to the average cardinality. Another indication is that the difference between the minimum and the maximum cardinality raises to 4 and 1 orders of magnitude, respectively. Their medians suggest that the distribution of FR_C is dominated by partitions smaller than the mean cardinality, and vice versa for FR_D.

These patterns indicate that serious bottlenecks are expected to rise in the case of the default load balancer of Hadoop. For this reason, we did not measure the actual running time it yields.

Neither do we present the running time of PairRange. The reason is that it is almost identical with that of MaxBlock, which appears in Section 3.6.3. In fact, PairRange is slower than MaxBlock by a couple of minutes, due to the higher overhead it involves: to adapt it to the functions of quadratic time complexity, an additional MapReduce job is required for both the edge- and the comparison-based strategy.

In more detail, we can integrate PairRange into the edge-based strategy by modifying the first `reduce` function of Figure 3.5 so that a global counter estimates the total number of comparisons, while the input is emitted without any further processing. Thus, the `map` function of a second, new job receives as input an individual block and applies the mapper of PairRange to it. The second reducer receives a balanced comparison range as input and estimates the corresponding edge weights (i.e., it applies the `reduce` function of Job 1 in Figure 3.5). Finally, the third job applies Job 2 of Figure 3.5 without any modifications.

For the comparison-based strategy, PairRange needs to extend the `reduce` function of Preprocessing in Figure 3.7 so that it estimates the total number of comparisons. Then, we need to add a new MapReduce job to every pruning algorithm; the `map` function receives individual blocks and applies the mapper of PairRange to them, while the `reduce` function receives a balanced comparison range as input and applies the functionality of the `map` functions in Figure 3.10(b),i.e., it estimates the corresponding edge weights. Finally, a second MapReduce job is required for every pruning algorithm; it consists of an identity mapper and the `reduce` functions in Figure 3.10(b).

Regarding the entity-based strategy, the goal of load balancing is to address the quadratic space complexity that appears in all `map` functions of Stage 3 (see Figure 3.12).This can be achieved by balancing the output of Preprocessing in Figure 3.9. Yet, among the three load balancing mechanisms, only the default one provided by Hadoop applies to this task without any modifications. As explained in Section 3.5.2, MaxBlock needs to adopt a new cost function, which expresses the disk space that is occupied by the compressed representation of the entities contained in every block.

However, PairRange cannot consider alternate cost functions, as it is inherently crafted for balancing comparisons. To adapt it to the entity-based strategy, we need to modify its functionality so that every block is entirely contained in a single comparison range (partition). In other words, we need to ensure that the comparisons of no block are spread across multiple partitions; otherwise, we have to alter the functionality of the entity-based mappers of Stage 3, which is out of the scope of this evaluation. To meet this requirement, the number of comparison ranges should be equal to or less than those of MaxBlock. We actually consider two configurations: using the same number of partitions as MaxBlock (PairRangeI) and using half the partitions of MaxBlock (PairRangeII). Note that PairRangeI does not necessarily produce the same distributions as PairRange in Table 3.4, because some blocks are larger than the remaining space in their partition, but are not broken into smaller chunks.

To evaluate the performance of load balancing for the entity-based strategy, we do not consider the distribution of comparisons among partitions. Instead, we are more interested in the

Table 3.5: The wall-clock time (in minutes) of Meta-blocking using the default Hadoop balancer, the two variations of PairRange, and MaxBlock for the entity-based strategy over DB_C, using the CBS weighting scheme across all pruning algorithms. The overhead of executing each load balancing algorithm, compared to the default balancing, is common for all pruning algorithms and is included in the wall-clock times.

	Overhead	CNP	WNP	CEP	WEP
Default	0	88	73	163	147
PairRangeI	2	77	68	145	130
PairRangeII	1	83	74	156	139
MaxBlock	3	74	65	145	123

compressed representation of blocks in bytes and the corresponding I/O overhead. We indirectly evaluate this aspect through the overhead time of all entity-based pruning algorithms. Table 3.5 presents the corresponding performance on top of DB_C, using the CBS weighting scheme for each algorithm. The rest of the datasets and weighting schemes yield similar results and are omitted for brevity.

We observe that MaxBlock exhibits the highest overhead, compared to the default balancer, due to its cost function, which compresses the representations of blocks before clustering them into partitions. PairRangeII is faster than PairRangeI, due to the lower number of partitions it involves, while the default balancer has 0 overhead, as it is the baseline of the overhead of the load balancing algorithms. Regarding the overall time, we observe that MaxBlock consistently provides the best execution times, with the default balancer being the least efficient one in most cases: it yields slower times than MaxBlock by 12% (CEP) to 20% (WEP). The two variations of PairRange fluctuate between these two extremes, with PairRangeI being consistently more efficient, because the larger number of partitions it employs ensures a more balanced I/O overhead across the nodes. Note that PairRangeII appears to be less time-efficient than the default load balancer over WNP, but their difference should be attributed to its execution overhead.

On the whole, we conclude that MaxBlock consistently outperforms the default mechanism of Hadoop across all parallelization strategies and pruning algorithms, even if it comes with a small execution overhead, compared to the default balancer. Moreover, MaxBlock is scalable – $O(|B| \cdot \log|B|)$ – and terminates within a few minutes for all datasets, as shown in Table 3.5.

Compared to PairRange, MaxBlock has three advantages: *(i)* It determines the number of partitions automatically, through a data-driven procedure. Instead, PairRange receives this parameter as input, requiring the user to specify it, after manually inspecting the data at hand. *(ii)* For the edge- and comparison-based parallelization strategies, MaxBlock consistently yields lower overall execution times than PairRange, as it saves a whole MapReduce job. *(iii)* MaxBlock is more flexible and generic than PairRange. Thus, it can be easily adapted to the entity-based strategy, by incorporating a cost function that tackles quadratic space complexity. Instead, PairRange is only

Table 3.6: The block collections after Block Filtering.

	$D_{dbpedia}$		$D_{freebase}$	
	DB_C	DB_D	FR_C	FR_D
Blocks $\|B\|$	1,239,315	1,499,422	1,308,970	4,521,129
Block comparisons $\|\|B\|\|$	$1.20{\cdot}10^{10}$	$2.17{\cdot}10^{10}$	$2.96{\cdot}10^{10}$	$6.53{\cdot}10^{10}$
BPE	12.12-12.68	11.72	57.28-3.86	19.70
Recall	0.998	0.998	0.961	0.907
Precision	$7.44{\cdot}10^{-5}$	$4.11{\cdot}10^{-5}$	$4.38{\cdot}10^{-5}$	$1.87{\cdot}10^{-5}$

suitable for balancing functions that suffer from quadratic time complexity, due to the number of comparisons they process.

3.6.3 Time Efficiency

We applied the three parallelization strategies of all Meta-blocking techniques to the four datasets 2 times and measured the corresponding average Overhead Time. The outcomes are presented in Table 3.7. Note that the edge-based strategy was inapplicable to FR_C and FR_D, as its space requirements exceeded the available 4 TB of disk space. For the other two datasets, we terminated prematurely the processes that ran for more than 6,000 minutes (100 hours), all of which were still far from completion. Below, we analyze the performance of each stage of the parallelized workflow of Meta-blocking.

Stage 1. The goal of this stage is to apply Block Filtering to the input block collection. In Table 3.7, we observe that the basic and the advanced strategy exhibit practically equivalent overhead times. Remember that the former involves two jobs that order once and globally the input blocks, whereas the advanced strategy entails a single job that sorts repeatedly and locally the input blocks. We can conclude, therefore, that the basic strategy offsets the cost of using two jobs by avoiding the computations that are repeated by the advanced one. However, the main reason for the equivalent overhead times is the linear time complexity of Block Filtering and its simple functionality that processes very large block collections at a negligible cost.

It should be stressed here that the exemplary performance of Block Filtering justifies the lack of a specialized load balancing algorithm for the functions with linear complexity.

Also worth noting is the qualitative performance of Block Filtering, which is presented in Table 3.6. We observe that despite its simple functionality, Block Filtering conveys significant enhancements in precision at a minor cost in recall. The total cardinality of all block collections is reduced by more than 60%, while their recall drops by less than 2%. As a result, the precision raises by 3 times, on average. The number of blocks remains almost intact, but the average number of blocks per entity (BPE) is significantly reduced. In this way, the computation of edge weights is accelerated to a considerable extent.

Stages 2 & 3. To compare the parallelization strategies for Meta-blocking on an equal basis, Table 3.7 considers the performance of Stages 2 and 3 as a whole; note that the edge-based strategy

Table 3.7: Overhead Time ($OTime$) in minutes for all Meta-blocking techniques across the four real datasets.

		DB_C			DB_D			FR_C		FR_D	
Block Filt.		2			2			3		6	
	Basic Adv.	2			2			3		6	
		Edge Based	Comp. Based	Entity Based	Edge Based	Comp. Based	Entity Based	Comp. Based	Entity Based	Comp. Based	Entity Based
CEP	ARCS	252	89	184	>6,000	135	431	1,319	1,244	3,359	3,065
	CBS	222	55	145	250	87	363	781	1,343	2,556	2,842
	ECBS	240	78	210	278	110	487	841	1,652	2,663	3,257
	JS	223	60	190	279	94	466	777	1,480	2,574	2,832
	EJS	1,996	116	>6,000	>6,000	180	>6,000	1,166	>6,000	4,090	>6,000
CNP	ARCS	554	370	73	>6,000	625	191	2,109	605	3,934	970
	CBS	491	301	74	559	527	186	1,488	643	2,514	995
	ECBS	555	383	76	639	633	197	1,949	665	3,058	1,187
	JS	534	363	83	618	620	210	1,637	656	2,546	977
	EJS	2,645	430	142	>6,000	733	382	2,319	1,069	5,222	1,993
WEP	ARCS	203	65	389	>6,000	99	319	520	1,006	1,802	1,967
	CBS	220	50	123	250	76	338	501	1,088	1,414	2,031
	ECBS	219	54	123	254	83	342	555	1,164	1,438	1,945
	JS	219	54	132	254	84	340	540	1,097	1,431	2,093
	EJS	1,993	81	204	>6,000	124	517	837	1,555	2,419	3,025
WNP	ARCS	562	363	63	>6,000	647	185	2,068	685	3,904	977
	CBS	498	304	65	569	539	196	1,534	541	2,671	1,313
	ECBS	568	389	73	658	647	193	1,971	588	3,046	1,238
	JS	553	373	74	641	644	202	1,636	690	2,790	1,176
	EJS	2,626	411	142	>6,000	700	379	2,317	1,041	5,214	2,211

was applied only to DB_C and DB_D, because its space requirements over the two larger datasets exceeded the available disk space (4 TB). Special care has been taken to highlight the relative efficiency not only of the three strategies, but also of the pruning and weighting schemes. For this reason, we examine these aspects separately.

Parallelization Strategies. We observe that when moving from left to right in Table 3.7, i.e., from the smallest block collection to the largest one, the Overhead Time increases analogously for all parallelization strategies. Even for the largest dataset, though, most Meta-blocking methods require less than 2 days (~3,000 minutes), thus being much faster than the serial processing, which requires almost 8 days over the high-end server described in Section 3.1. Most importantly, though, there is a considerable discrepancy among the time-efficiency of the three parallelization strategies, which designates that the parallelization of Meta-blocking is not a trivial task.

In more detail, the edge-based strategy is consistently slower than the comparison-based one. Their difference is particularly intense in the case of edge-centric pruning schemes, but is significantly reduced for the node-centric ones. There are two exceptions that prove this rule: for CNP and WNP in combination with JS, the edge-based strategy is faster (by less than 3 minutes) than the comparison-based one over DB_D.

Regarding the entity-based strategy, it is significantly faster than the other strategies in the case of the node-centric pruning schemes across all datasets. For DB_C, for instance, it is 5 times faster than the comparison-based strategy of WNP in combination with all weighting schemes. Compared to the edge-based strategy, it is 9 times faster, on average, for the same pruning scheme and dataset. In the case of the edge-centric algorithms, though, the entity-based strategy outperforms only the edge-based one; compared to the comparison-based strategy, it requires at least twice as much time.

There are two factors that determine the relative performance of the parallelization strategies. The first one is the number of MapReduce jobs they involve. The larger this number is, the higher the overhead becomes and the less efficient is the corresponding strategy. This explains the inferior performance of the edge-based strategy, when compared to the comparison-based one: its Preprocessing involves one more job in order to calculate the weights of all edges in the blocking graph. The same holds for the entity-based strategy, when it is combined with the edge-centric pruning schemes; in this case, the entity-based strategy employs one more job than the comparison-based one in order to calculate the global pruning criterion in the absence of preprocessing computations in Stage 2.

The second important factor for the time efficiency of the parallelization strategies is the I/O they involve between the independent nodes of the cluster. The higher the I/O of a strategy is, the higher is its overhead and the lower is its time efficiency. Comparing the edge- and comparison-based strategies in this respect, the former involves a higher I/O, because it materializes an edge for every comparison in the input blocks – even the repeated ones. In contrast, the comparison-based approach creates a distinct edge only for the non-repeated comparisons. In our datasets, the latter approach yields around 30% less comparisons. An even more time-efficient approach is implemented by the entity-based strategy, which sends no edges through the network. Instead, it exchanges the nodes of the blocking graph, as their number is typically orders of magnitude lower than the number of edges. By attaching the necessary information to every graph node, the edges can be created, weighted, and pruned locally, inside the independent nodes of the cluster.

Overall, we recommend using the entity-based strategy for node-centric pruning algorithms, and the comparison-based strategy for edge-centric ones.

Pruning Schemes. WEP is the most efficient method for the edge- and comparison-based strategies across all datasets, because it involves the simplest processing. For these strategies, the second fastest method is CEP, since it merely adds one job to the functionality of WEP in order to convert the cardinality pruning criterion into a weight one. For the entity-based strategy, though, WEP and CEP are the least efficient methods, as they require 1 and 2 additional jobs, respectively, in order to compute their pruning criteria.

For this strategy, the node-centric pruning schemes, CNP and WNP, are the most efficient ones, involving a single job. In contrast, they are the most time-consuming schemes for the other strategies, since they process every edge twice, inside the neighborhoods of both adjacent nodes.

It is interesting to compare these patterns with the relative time efficiency of pruning schemes

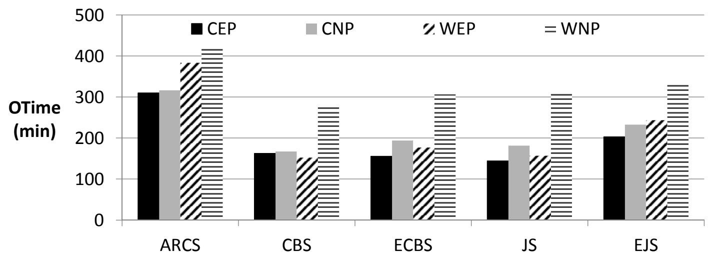


Figure 3.14: Overhead Time in minutes for all configurations of the serialized workflow over DB_C.

in the case of serial processing. To this end, Figure 3.14 presents the Overhead Time of all serialized workflows over the DB_C dataset. The measurements were performed using the high-end server mentioned in Section 3.1. Similar patterns were exhibited for the other datasets and are omitted for brevity. First of all, we observe that the overhead of serial processing is significantly higher than that of parallel processing in the vast majority of cases. Second, the pruning schemes exhibit a similar behavior as in the case of the edge- and comparison-based strategies: the edge-centric ones, CEP and WEP, are significantly faster than their node-centric counterparts, CNP and WNP. However, the relations are different between cardinality- and weight-based schemes: CEP and CNP are faster than WEP and WNP, respectively, because the latter involve an additional iteration over the edges in order to estimate their pruning criterion. Thus, the most time-efficient serial algorithm overall is CEP, while WNP remains the most time-consuming one.

Weighting Schemes. For the edge-based strategy, CBS is the fastest weighting scheme, as the output value of its first `reduce` function in Stage 2 is empty. ARCS, ECBS, JS add information to this output value and, thus, require more time and I/O in order to process it. Given that they involve the same number of jobs, they exhibit similar overhead times. EJS requires two additional jobs in order to estimate the degree of every node, thus being the most time-consuming weighting scheme.

For the comparison-based strategy, we observe slightly different patterns. CBS, ECBS and JS yield similar overhead times, because they basically perform the same computation: for each pair of entities, they estimate the intersection of the associated block lists. They are faster than ARCS and EJS, as they rely exclusively on the information contained in the enriched input (i.e., the ids of the blocks associated with every entity). In contrast, EJS requires two additional jobs and is the most time-consuming weighting scheme in all cases. In most cases, ARCS lies between these two extremes, as it requires additional information and, thus, involves higher I/O than the most efficient schemes.

For the entity-based strategy, the differences between the weighting schemes are minor, except for EJS, which again requires an additional job and is, thus, the most time-consuming scheme. Among the other schemes, CBS and ARCS are slightly faster, since they do not load in memory

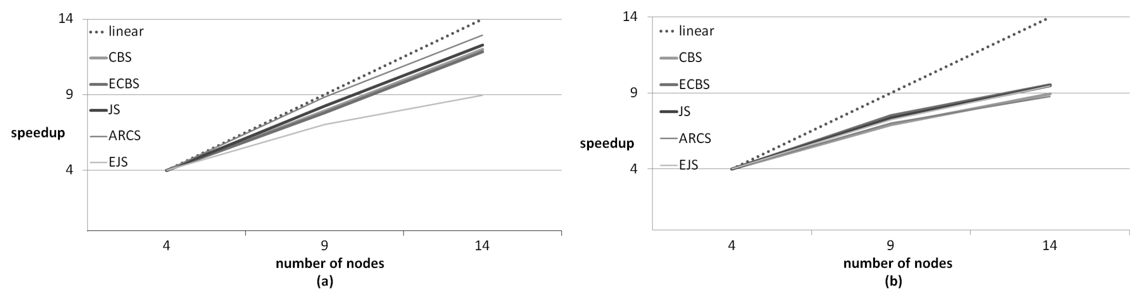


Figure 3.15: Speedup over DB_C of (a) the comparison-based strategy for WEP, and (b) the entity-based strategy for CNP.

the array with the number of blocks per entity, unlike JS and ECBS.

In the case of the serialized workflow, Figure 3.14 shows that ARCS is consistently the most time-consuming weighting scheme, because it produces very low values as edge weights (with tens of decimal digits). It is followed by EJS, which again involves higher computational cost in order to estimate the degree of every node. The remaining schemes share almost the same overhead, as their processing is very similar, computing the intersection of block lists.

3.6.4 Scalability

To assess the scalability of the comparison- and entity-based strategies, we estimate the *speedup* of their most time-efficient pruning schemes. That is, we measure the extent to which their overhead time decreases as we increase the number of available cluster nodes. Specifically, we apply the comparison-based WEP and the entity-based CNP to DB_C in combination with all weighting schemes. We increase the number of slave nodes from 4 to 9 and 14; in every case, there is an independent master node. The outcomes are presented in Figures 3.15(a) and (b) for WEP and CNP, respectively. In every figure, there is a dotted diagonal line, which illustrates the ideal case, where the speedup is linear to the number of nodes.

In Figure 3.15(a), we observe that all the weighting schemes show a speedup close to the ideal, with the exception of EJS. ARCS seems to be the weighting scheme that best exploits the available resources, showing a speedup of 12.92 when using 14 nodes. ECBS, CBS and JS have almost identical speedup values, ranging from 11.8 to 12.3, when using 14 nodes. This is because they basically perform the same computations, as explained previously. For EJS, the speedup is constantly lower than that of the other weighting schemes, because of the quadratic complexity of its additional jobs.

Regarding CNP, Figure 3.15(b) indicates that the deviation in the speedup of the various weighting schemes is much smaller than for WEP. Indeed, the speedup for 14 nodes fluctuates between 8.8 for ARCS and 9.5 for ECBS. This time EJS does not yield the worst speedup, as its additional job involves a linear complexity instead of a quadratic one.

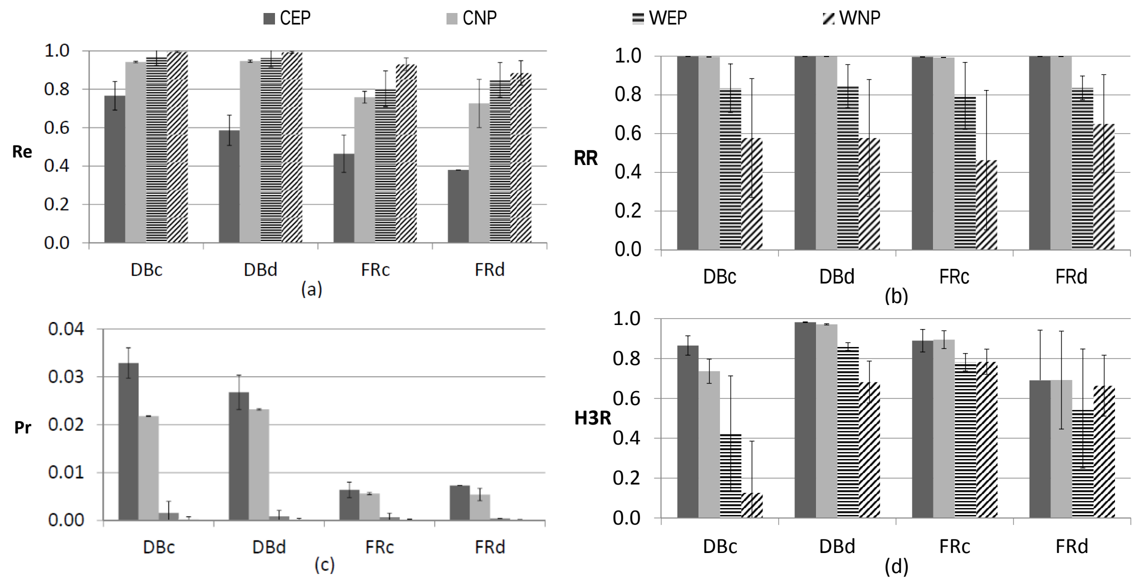


Figure 3.16: Average performance of the four pruning algorithms with respect to (a) Recall, (b) RR, (c) Precision, and (d) $H3R$.

In practice, these patterns indicate that the more cluster nodes we used, the faster was the execution of both strategies. For WEP, the improvement in time was almost as much as the the number of additional resources (until a certain point), while for CNP, n additional nodes improved the Overhead Time by $2n/3$ times. The reason is that WEP is able to balance the workload of its nodes right after Preprocessing, i.e., before applying Meta-blocking in Stage 3. In the case of CNP, though, this is impossible, since the workload of every `reduce` function in Stage 3 is not known a-priori.

3.6.5 Qualitative Results

To assess the quality of the restructured blocks produced by Meta-blocking, we consider their performance with respect to the four relevant measures of Section 3.6.1. For every pruning scheme and dataset, we estimated the average value and the standard deviation of every measure across the five weighting schemes. The outcomes are presented in Figures 3.16(a) to (d). Remember that in all diagrams, the higher a bar is, the better is the corresponding performance. We should also note that all the MapReduce implementations are exact adaptations of their serialized counterparts, which means that the qualitative results of the serialized and the parallel implementations are identical.

Starting with Figure 3.16(a), we observe that the relative recall of the pruning schemes remains the same across all datasets: the node-centric ones, CNP and WNP, are more robust and detect more matches than their edge-centric counterparts, CEP and WEP. The cardinality-based schemes, CEP and CNP, consistently achieve lower recall than the weight-based ones, WEP and WNP, which exceed 0.8 across all datasets. In fact, CEP and CNP they reduce the original recall by less than 10%, despite the significant enhancements in efficiency they convey.

Indeed, Figure 3.16(b) shows that WEP consistently achieves an RR close to 0.8, thus saving 80% of the original comparisons. The pruning of WNP is more shallow, as it retains at least one edge per node. Its RR fluctuates between 0.46 and 0.65, thus saving around half the original comparisons. For CEP and CNP, RR is consistently higher than 0.99. In fact, they perform such a deep pruning that they reduce the pairwise comparisons by 2 to 3 orders of magnitude across all datasets. This explains their poor recall.

Yet, Figure 3.16(c) demonstrates that CEP and CNP achieve significantly higher precision across all datasets. Compared to the input blocks, the restructured ones, produced by those schemes, increase precision by 2 to 3 orders of magnitude. For WEP and WNP, the improvement is slightly higher than an order of magnitude. This pattern actually indicates a clear trade-off between precision and recall: the higher precision is for a specific method and dataset, the lower is the corresponding recall and vice versa.

To identify the scheme that achieves the overall best balance between the identified matches and the executed comparisons, we use $H3R$, which is presented in Figure 3.16(d). We observe that the cardinality-based methods, CEP and CNP, exhibit the highest values across all datasets, fluctuating between 0.97 and 0.67. The difference between the two methods is small, even though CNP retains twice as many comparisons as CEP, on average. Still, CNP should be preferred, since it retains the best comparisons per entity and, thus, is more robust to recall.

These patterns are in accordance with earlier findings about the relative performance of the four pruning schemes [82].

3.6.6 Discussion

The results of our experimental analysis demonstrate that the proposed strategies for parallel Meta-blocking yield significant improvements in the execution time, thus enabling ER in voluminous datasets. However, simple strategies cannot give us the full benefit: we observed that the edge-based strategy leads to significantly higher space requirements and is consistently slower than the comparison- and entity-based ones. The experiments also showed that our load balancing algorithm consistently outperforms the default balancer of Hadoop, assigning an almost identical workload to all the nodes of the cluster.

Among the four pruning schemes, the overall winner is CNP, as it involves the most time-efficient functionality (when using the entity-based strategy) and achieves the best balance between effectiveness and efficiency in terms of $H3R$ (CEP exhibits similar $H3R$ values, but is significantly less robust to recall than CNP). The five weighting schemes exhibit similar quality results and are almost equivalent with respect to time efficiency, with the exception of EJS, which is much slower and less scalable than the rest.

In summary, the edge-based strategy should not be used in practice. Instead, parallel Meta-blocking should be applied using the comparison-based and entity-based strategies. The edge-centric algorithms, CEP and WEP, should always be combined with the comparison-based strategy, while the node-centric algorithms, CNP and WNP, should always be used with the entity-based strategy.

To demonstrate in a more intuitive way the actual benefit of Meta-blocking, we have estimated the times required to get the final matching results with and without Meta-blocking, given a block collection. In the first case, we sum the times needed for the three stages of Meta-blocking and the time required to perform the resulting comparisons of Meta-blocking. In the latter case, we only estimate the time required to perform all the comparisons suggested by the input block collection. To estimate the time required for the comparisons, we performed 1 billion comparisons, using the Jaccard similarity of the tokens in the values of the DB_C collection. The average time required to get the similarity of 1 pair of entity descriptions was $4.9 \cdot 10^{-7}$ minutes. Based on this number and taking as an example the CBS weighting scheme and the CNP pruning scheme, using the entity-based strategy, we estimate that the time required to perform Meta-blocking (including Block Filtering) and then the comparisons suggested by Meta-blocking is 76 minutes + $3.96 \cdot 10^7$ comparisons × 4.9 $\cdot 10^{-7}$ minutes/comparison ≈ 95.5 minutes. The corresponding time required to perform the comparisons suggested by blocking, without using Meta-blocking, would be $4.23 \cdot 10^{10}$ comparisons × 4.9 $\cdot 10^{-7}$ minutes/comparison = 20,727 minutes = 345.45 hours ≈ 14 days. The cost of using Meta-blocking, in this case, is a loss of 3.79% in recall.

3.7 Conclusion

In the previous chapter, we saw how blocking can be scaled to Web of data, without any qualitative cost. The scalability of blocking for Web data would be vain, if the processes that follow blocking remained non-scalable. In this chapter, we parallelized Meta-blocking using MapReduce and enhanced dramatically the time efficiency of its serialized implementation. We proposed 3 parallelization strategies: *(i)* The edge-based one implements a straightforward approach that materializes the blocking graph; hence, it involves high I/O and high space requirements that do not scale well to large datasets. *(ii)* The comparison-based strategy offers a more elaborate implementation that uses the blocking graph implicitly. This way, it reduces the overhead of data exchange and the number of required MapReduce jobs, leading to significant performance gains, especially for the edge-centric pruning schemes, CEP and WEP. *(iii)* The entity-based strategy is completely independent of the blocking graph, minimizing the data exchange and the overhead of MapReduce job chains. This approach offers an optimized implementation for the node-centric pruning schemes, CNP and WNP. All these strategies do not affect the qualitative results of the Meta-blocking algorithms suggested in [82]. We observe that the cardinality-based methods, CEP and CNP, exhibit the best qualitative results across all datasets. The difference between the two methods is small, even though CNP retains twice as many comparisons as CEP, on average. Still, CNP should be preferred, since it retains the best comparisons per entity and, thus, is more robust to recall, while it is also the most time-efficient pruning method, when the entity-based strategy is employed. All these strategies were combined with *MaxBlock*, a purpose-built load balancing algorithm that distributes the workload evenly among the cluster nodes. We have thoroughly evaluated our parallelization strategies for Meta-blocking using real datasets from the Web of Data. The datasets and the implementation of our techniques are publicly available[8].

Even if the simple Meta-blocking model presented in this chapter can achieve a very efficient computation of the most promising candidate matches, based on the similarity of their values, it still fails to identify matching entity descriptions that have low value similarity, but are related with another pair of similar, or even matching entities. In the next chapter, we will see how the composite Meta-blocking model that we introduced in Section 3.2 can improve not only the efficiency, but also the effectiveness of atomic blocking, and, consequently, of the whole ER workflow, by utilizing a disjunctive blocking graph, exploiting multiple matching evidence given from the input block collections as well as the relatedness of entities in the original entity graph.

[8]`https://github.com/vefthym/ParallelMetablocking`

Chapter 4
Entity Matching

4.1 Introduction

To the best of our knowledge, no existing ER framework simultaneously accomplishes: (a) support for matching *highly heterogeneous* entities within or across domains, (b) *full automation* avoiding any human intervention or presuming domain knowledge, and (c) *massive parallelization* of blocking and matching computations.

As we have seen in the previous chapter, similarity evidence for matching entities inside and across KBs in the Web of data can be obtained only by looking at the bag of literals (mostly strings) contained in descriptions, regardless of the attributes they appear as values. As the *value-based* similarity of a pair of entities may still be weak due to high veracity, we need to consider additional sources of matching evidence related to the *similarity of neighboring* entities (i.e., connected via semantic relations).

The id of an entity description may appear in the values of another entity description, this way forming an *entity graph*. Figure 4.1(a) presents parts of the Wikidata (left) and DBpedia (right) KBs, showing the entity graph that captures connections inside them. For example, Restaurant2 and Jonny Lake are neighbor entities in this graph, connected via a "headChef" relation. If we compare John Lake A to Jonny Lake based on their values only, it is easy to infer that those descriptions are matching; they are *strongly similar*. However, we cannot be that sure about Restaurant1 and Restaurant2, if we only look at their values. Those descriptions are *nearly similar* and we have to look further, at the similarity of their neighbors (e.g, John Lake A and Jonny Lake) in order to verify that they match.

In state-of-the-art systems, such as SiGMa [66], LINDA [16] and RiMOM [91], this is typically done through an *iterative process* that relies on *domain knowledge* regarding the equivalence of relations between neighboring entities. Initially, they detect strongly similar entities using reasonable heuristics, such as identical literal values. Then, they use these resources as seeds for bootstrapping an iterative algorithm that detects new matches based exclusively on neighbor similarity. The more neighboring entities are matching, the stronger is the evidence regarding a candidate entity pair. This process is repeated until converging to a stable solution (i.e., no more matches are identified) or until no comparison exceeds the minimum similarity threshold. In this chapter,

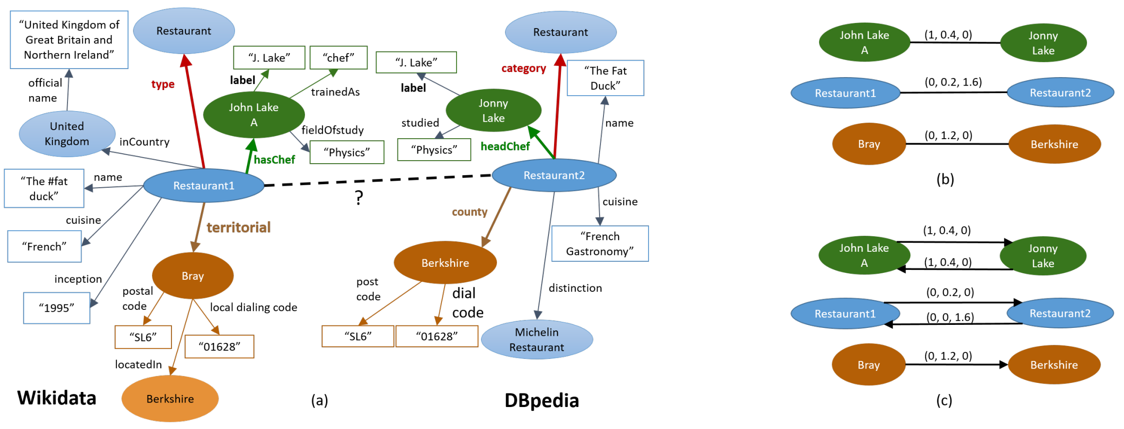


Figure 4.1: (a) Parts of entity graphs, representing the Wikidata (left) and DBpedia (right) KBs, (b) parts of the corresponding disjunctive blocking graph, and (c) the corresponding graph after pruning.

we argue that to assess the impact of neighbor similarity in a candidate pair, we do not actually need an iterative process, while an estimation of which entity relations in this neighborhood are important to consider can be guided by simple statistics over KBs.

This opens new perspectives in reducing the number of required comparisons due the high volume of entities via blocking and massively parallel computing techniques. Rather than blocking entities based on the values of specific attributes (as in SiGMa [66] and RiMOM [91]), we consider as blocking keys the entities' tokens [82]. To avoid restricting our candidates for matching exclusively on strongly similar entities, we consider a composite blocking scheme that allows to assess both *value-* and *neighbor-based* similarity of candidate entities in conjunction with evidence provided by the strong similarity of the entities names (e.g., `rdfs:labels`). This composite scheme can be naturally implemented (via hashing) in a Big Data computing platform (e.g., Spark [106]).

Overall, this chapter makes the following contributions[1]:

- In Section 4.3, we define *new similarity metrics* for comparing the values and the neighbors of entities without requiring knowledge of the entity types or their correspondences. We rely on simple statistics over the KBs to recognize the most important entity relations involved in neighbor similarity or the most distinctive attributes serving as names of entities. The proposed similarity metrics can be efficiently computed using information provided only by the blocks of entities.

- In Section 4.4, we show how our value and neighbor similarity metrics can be computed based only on the information provided by token blocking. We combine the matching evidence from the values and the neighbor of entities stemming from token blocking with an

[1]This work is under submission.

additional blocking on the names of the entities, in a disjunctive blocking graph. We present an efficient algorithm for weighting and then pruning the edges of this graph, corresponding to candidate matches.

- In Section 4.5, we propose a *non-iterative matching process*. Unlike the data-driven convergence of existing systems (e.g., LINDA [16], SiGMa [66] and RiMOM [91]), our matching method involves a specific number of steps that are independent of data characteristics. Matching entities are found by applying 4 heuristics to the blocking graph. Initially, we recognize matching entities based on their name. Then, the value similarity is exploited to find matching entities with a large number of common and not frequent tokens. When value similarity is not high, entities are matched based on both value and neighbors' similarity using a rank aggregation function. Finally, reciprocal evidence of matching entities is exploited: only entities that are mutually ranked in the top positions of their unified ranking lists are considered as matches. MinoanER heuristics can be implemented in a massively parallel system like Spark.

- In Section 4.6, we experimentally compare the effectiveness of our approach against the state-of-the-art methods using real datasets from KBs involved in benchmarking efforts in the field. The main conclusions drawn from our experiments is that MinoanER achieves at least equivalent performance over KBs with a small number of attributes and entity types (i.e., *low levels of heterogeneity*), even without making any assumption regarding the alignment of relations in the input. Yet, it outperforms to a significant extent existing ER tools when matching KBs with a big number of attributes and entity types (i.e., *high levels of heterogeneity*).

We overview the main differences with the state-of-the-art ER methods in Section 4.2 and conclude this chapter in Section 4.7.

4.2 Related Work

In this section, we position MinoanER with respect to state-of-the-art techniques proposed for linked entity descriptions.

Value-based similarities (e.g., Jaccard, Dice) usually assess the similarity of two descriptions based on the values of specific attributes. Our value similarity is a variation of ARCS (Equation 3.4), which drops any schema information and considers descriptions as a bag of words. Compared to ARCS, though, we focus more on the *number than the frequency of common tokens between two descriptions*. *Relational similarity* measures additionally consider neighbor similarity by exploiting the value similarity of all or some of the entities' neighbors. For example, SiGMa [66] and RiMOM-IM [91] consider the similarity of "compatible" neighbors, linked with pre-aligned relations, while LINDA [16] considers only neighbors linked via relations with similar labels (small edit distance).

MinoanER does not aggregate different similarities in one similarity score; instead, it uses a disjunction of the different evidence coming from the values, neighbors and names of the descriptions. The most important neighbors are detected automatically from dataset statistics.

Based on the nature of the matching decision, ER can be characterized as *pairwise* or *collective*. In pairwise ER (e.g., [61]), we only need to know the value similarity of descriptions to decide if they match. Collective ER (e.g., [10]) iteratively updates the matching decision for entities by dynamically assessing the similarity of their neighbors. The starting point for this similarity propagation is a set of seed matches identified by a value-based blocking. *MinoanER is a static collective ER approach*, in which all sources of similarity are assessed only once per candidate pair. By considering a composite blocking not only on the value but also the neighbors similarity, we discover in a non-iterative way most of the matches returned by the data-driven convergence of existing systems, or even more (see Section 4.6). Next, we explore how recent works in collective ER dynamically update their similarity assessment based on neighborhood evidence.

To capture this inherently iterative intuition, [10] performs hierarchical agglomerative clustering, where, at each iteration, the two most similar (according to a relational similarity function) clusters, i.e., groups of matching descriptions, are merged, until the similarity of the most similar clusters is below a threshold. When two clusters are merged, the similarities of their related clusters, i.e., the clusters corresponding to descriptions which are related to the descriptions in the merged cluster, are updated.

SiGMa [66] starts with seed matches having identical entity names. Then, it propagates the matching decisions on the compatible neighbors of existing matches. Unique Mapping Clustering is applied for detecting matches. First, it places all pairs into a priority queue, in decreasing (relational) similarity. At each iteration, the top pair is considered a match, if none of its entities has been already matched and their similarity exceeds a given threshold t. For every new matched pair, the similarities of the neighbors are recomputed and their position in the priority queue is updated. The process ends when the top pair has a lower similarity than t.

LINDA [16] follows a very similar approach, which differs from SiGMa only in the similarity functions used and the lack of a manual relation alignment. Instead, LINDA relies on the edit distance of the relations names used in the two KBs to determine if they are equivalent or not. This alignment method makes a strong assumption that entity descriptions in KBs use meaningful names for relations and similar names for equivalent relations, which is rarely true in the Web of Data. Finally, rather than using a similarity threshold, the resolution process in LINDA terminates when the priority queue is empty, or after performing a predetermined number of iterations.

RiMOM-IM [68, 91] initially considers as matches entities placed in blocks of size 2 (this is more generic than heuristic H1, as it considers all attribute value tokens). It also uses a heuristic called "one-left object": if two matched descriptions e_1, e'_1 are connected via aligned relations r and r' and all their entity neighbors via r and r', except e_2 and e'_2, have been matched, then e_2 and e'_2 are also considered matches. Finally, similar to SiGMa, RiMOM-IM employs a complex similarity score, which requires the alignment of relations among the KBs.

Three are the main differences of MinoanER to SiGMa, LINDA and RiMOM-IM. First, the matching process iterates over the disjunctive blocking graph, instead of the initial KBs. Second, MinoanER employs statistics to automatically discover distinctive entity names and important relations. Third, MinoanER exploits different sources of matching evidence (values, names and neighbors) to statically identify candidate matches already from the step of blocking.

On another line of research, PARIS [97] uses a probabilistic model to identify matching evidence, based on previous matches and the functional nature of entity relations. Specifically, a relation is considered to be functional if, for a given source entity, there is only one destination entity (e.g., `wasBornIn`). The basic matching idea is that if $r(x, y)$ is a function in one KB and $r(x, y')$ is a function in another KB, then y and y' are considered to be matches. The *functionality*, i.e., degree by which a relation is close to being a function (considering only its discriminability, not its support), and the alignment of relations along with previous matching decisions determine the decisions in subsequent iterations. Specifically, the functionality of each relation is computed at the beginning of the algorithm and remains unchanged. Then, at the first iteration, instances with identical values (for all attributes) are considered matches and based on those matches, an alignment of relations takes place. At the next iteration, instances can be now compared based on the newly aligned relations, and this process continues until convergence. In the last step, an alignment of classes (i.e., entity types) also takes place. *Unlike MinoanER, PARIS cannot deal with structural heterogeneity, while it targets both ontology and instance matching.*

Finally, [89] parallelizes the collective ER of [10], relying on a black-box matching and exploits a set of heuristic rules for structured entities. It essentially runs multiple instances of the matching algorithm in subsets of the input entities (similar to blocks), also keeping information for all the entity neighbors, needed for similarity propagation. Since some rules may require the results of multiple blocks, an iterative message-passing framework is employed. *Rather than a block-level synchronization, the MinoanER parallel computations in Spark require synchronization only across the 4 threshold-free and schema-agnostic matching heuristics* (see Section 4.5.1).

Regarding the heuristics, the ones employed by MinoanER based on values and names are similar to heuristics that have been already employed in the literature individually (e.g., in [66, 68, 91]), while the idea of reciprocity has been applied to enhance the results of Meta-blocking [84], but was never used in matching. In this work, we use a combination of those heuristics for the first time, also introducing a novel rank aggregation heuristic to incorporate value and neighbor matching evidence.

4.3 Basic Definitions

The relations of an entity description $e_i \in \mathcal{E}$, are defined as $relations(e_i) = \{p|(p, j) \in e_i \wedge e_j \in \mathcal{E}\}$, while its neighbors as $neighbors(e_i) = \{e_j|(p, j) \in e_i \wedge e_j \in \mathcal{E}\}$. In the example of Figure 4.1(a), $relations(Restaurant1)$ = {hasChef, territorial, inCountry}, and $neighbors(Restaurant1)$ = {John Lake A, Bray, United Kingdom}.

In the remaining of this chapter, we will focus only on clean-clean ER, i.e., the sub-problem of ER in which we only seek for matches among two clean entity collections. We only use two entity collections for an easier presentation of the problem, but the proposed techniques can be easily generalized for more than two clean entity collections. This is the problem that is typically met in the Web of data, as opposed to dirty ER, i.e., seeking for matches within a single entity collection, which is typically met in data warehouses.

4.3.1 Entity similarity based on values

Traditionally, similarities between entity descriptions are computed based on their values. In our work, we apply a similarity measure based on the intuition that if two entity descriptions share many, infrequent tokens, then they have high value similarity (this is an adaptation of ARCS presented in Equation 3.4 as sim_{ARCS}, and of Adamic/Adar similarity measure [1])[2]. Formally:

Definition 4.1 (Value similarity)**.** *Let $\mathcal{E}_1$, $\mathcal{E}_2$ be two entity collections. The* ***value similarity*** *of two entity descriptions $e_i \in \mathcal{E}_1, e_j \in \mathcal{E}_2$ is:*

$$valueSim(e_i, e_j) = \sum_{t \in tokens(e_i) \cap tokens(e_j)} \frac{1}{log_2(EF_{\mathcal{E}_1}(t) \cdot EF_{\mathcal{E}_2}(t) + 1)},$$

where $EF_{\mathcal{E}}(t) = |\{e_i | e_i \in \mathcal{E} \wedge t \in tokens(e)\}|$ stands for "Entity Frequency", which is the number of entity descriptions in $\mathcal{E}$ having token t in their values.

Note that $valueSim$ is not a normalized measure, since it can take any value in $[0, +\infty)$, with 0 $valueSim$ denoting the existence of no common tokens in the values of the compared descriptions.

Proposition 4.1. *$valueSim$ is a* similarity metric*, as it satisfies the following properties [19]:*

- $valueSim(e_i, e_i) \geq 0,$ (4.1)
- $valueSim(e_i, e_j) = valueSim(e_j, e_i),$ (4.2)
- $valueSim(e_i, e_i) \geq valueSim(e_i, e_j),$ (4.3)
- $valueSim(e_i, e_i) = valueSim(e_j, e_j) = valueSim(e_i, e_j) \Leftrightarrow e_i = e_j,$ (4.4)
- $valueSim(e_i, e_j) + valueSim(e_j, e_z) \leq valueSim(e_i, e_z) + valueSim(e_j, e_j).$ (4.5)

Property 4.1 states that the self-similarity of any description e_i is non-negative. Although it is not mandatory to set this lower bound at zero (e.g., the similarity measure used in LINDA [16] can have negative values), this is a common and reasonable choice. Since $valueSim$ is not normalized, the self-similarity of any description is not bound to a specific number; it merely depends on the number and frequency of its tokens. Property 4.2 states that $valueSim$ is symmetric.

[2]Currently, we handle numbers and dates in the same way as strings, assuming string-dominated entity descriptions.

Property 4.3 states that for any description e_i the self-similarity is no less than the similarity between e_i and any other description e_j. Property 4.4 states that the statements $valueSim(e_i,e_i) = valueSim(e_j,e_j) = valueSim(e_i,e_j)$ and $e_i = e_j$ are equivalent. Property 4.5 states that the similarity between e_i and e_z through e_j is no greater than the direct similarity between e_i and e_z plus the self-similarity of e_j. This property is the equivalent of the triangle inequality in distance metrics [19].

Proof. **Property 4.1:** If $tokens(e_i) \cap tokens(e_j) = \emptyset$, then $valueSim(e_i,e_j) = 0$. Else, for any common token $t \in tokens(e_i) \cap tokens(e_j)$, it holds that $EF_{\mathscr{E}_1}(t) \geq 1$, and $EF_{\mathscr{E}_2}(t) \geq 1$ (since $t \in tokens(e_i)$, and $t \in tokens(e_j)$). Thus,

$$log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t) + 1) \geq log_2(2) = 1 \Rightarrow \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t) + 1)} > 0.$$

Since $valueSim(e_i,e_j)$ is the sum of positive numbers, it is also a positive number.

Property 4.2: $tokens(e_i) \cap tokens(e_j) = tokens(e_j) \cap tokens(e_i)$. If $tokens(e_i) \cap tokens(e_j) = \emptyset$, then $valueSim(e_i,e_j) = valueSim(e_j,e_i) = 0$. Else, for any common token $t \in tokens(e_i) \cap tokens(e_j)$, it holds that $EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t) = EF_{\mathscr{E}_2}(t) \cdot EF_{\mathscr{E}_1}(t) \Rightarrow$
$\frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} = \frac{1}{log_2(EF_{\mathscr{E}_2}(t) \cdot EF_{\mathscr{E}_1}(t)+1)} \Rightarrow valueSim(e_i,e_j) = valueSim(e_j,e_i)$.

Property 4.3: $valueSim(e_i,e_i)$ refers to the similarity of two identical entity descriptions in two entity collections $\mathscr{E}_1, \mathscr{E}_2$. Then, $valueSim(e_i,e_i) = \sum_{t \in tokens(e_i)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)}$.
By definition, $tokens(e_i) \cap tokens(e_j) \subseteq tokens(e_i) \Rightarrow \sum_{t \in tokens(e_i)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} \geq$
$\sum_{t \in tokens(e_i) \cap tokens(e_j)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} \Rightarrow valueSim(e_i,e_i) \geq valueSim(e_i,e_j)$.

Property 4.4: Proof of "$\Leftarrow$":
$e_i = e_j \Rightarrow tokens(e_i) = tokens(e_j) \Rightarrow tokens(e_i) \cap tokens(e_j) = tokens(e_i) \Rightarrow valueSim(e_i,e_j) = \sum_{t \in tokens(e_i)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} = valueSim(e_i,e_i) = valueSim(e_j,e_j)$.
Proof of "$\Rightarrow$":
$valueSim(e_i,e_i) = valueSim(e_i,e_j) \Rightarrow$
$\sum_{t \in tokens(e_i)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} = \sum_{t \in tokens(e_i) \cap tokens(e_j)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} \Rightarrow$
$tokens(e_i) = tokens(e_i) \cap tokens(e_j) \Rightarrow tokens(e_i) \subseteq tokens(e_j)$.
Accordingly, $valueSim(e_j,e_j) = valueSim(e_i,e_j) \Rightarrow tokens(e_j) \subseteq tokens(e_i)$.
Since, for the needs of $valueSim$, we represent an entity description as a set of tokens only, from the last two equations, it holds that $tokens(e_i) = tokens(e_j) \Rightarrow e_i = e_j$.

Property 4.5: We know that $tokens(e_i) \cap tokens(e_j) = (tokens(e_i) \cap tokens(e_j) \cap tokens(e_z)) \cup ((tokens(e_i) \cap tokens(e_j)) \setminus tokens(e_z))$. We also know that $(tokens(e_i) \cap tokens(e_j) \cap tokens(e_z)) \cap ((tokens(e_i) \cap tokens(e_j)) \setminus tokens(e_z)) = \emptyset$. Based on those properties, we have that:
$valueSim(e_i,e_j) = \sum_{t \in tokens(e_i) \cap tokens(e_j) \cap tokens(e_z)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} +$
$\sum_{t \in (tokens(e_i) \cap tokens(e_j)) \setminus tokens(e_z)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)}$,
and that:

$valueSim(e_j,e_z) = \sum_{t \in tokens(e_i) \cap tokens(e_j) \cap tokens(e_z)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} +$
$\sum_{t \in (tokens(e_j) \cap tokens(e_z)) \setminus tokens(e_i)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)}$.

Then, $valueSim(e_i,e_j) + valueSim(e_j,e_z) =$
$\sum_{t \in tokens(e_i) \cap tokens(e_j) \cap tokens(e_z)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} +$
$\sum_{t \in (tokens(e_i) \cap tokens(e_j)) \setminus tokens(e_z)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} +$
$\sum_{t \in tokens(e_i) \cap tokens(e_j) \cap tokens(e_z)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} +$
$\sum_{t \in (tokens(e_j) \cap tokens(e_z)) \setminus tokens(e_i)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)}$.

However, $tokens(e_i) \cap tokens(e_j) \cap tokens(e_z) \subseteq tokens(e_i) \cap tokens(e_z) \Rightarrow$
$\sum_{t \in tokens(e_i) \cap tokens(e_j) \cap tokens(e_z)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} \leq \sum_{t \in tokens(e_i) \cap tokens(e_z)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)}$, and
$(((tokens(e_i) \cap tokens(e_j)) \setminus tokens(e_z)) \cup$
$(tokens(e_i) \cap tokens(e_j) \cap tokens(e_z)) \cup$
$((tokens(e_j) \cap tokens(e_z)) \setminus tokens(e_i))) \subseteq tokens(e_j) \Rightarrow$
$\sum_{t \in (tokens(e_i) \cap tokens(e_j)) \setminus tokens(e_z)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} +$
$\sum_{t \in tokens(e_i) \cap tokens(e_j) \cap tokens(e_z)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} +$
$\sum_{t \in (tokens(e_j) \cap tokens(e_z)) \setminus tokens(e_i)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)} \leq \sum_{t \in tokens(e_j)} \frac{1}{log_2(EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t)+1)}$. Thus, it holds that $valueSim(e_i,e_j) + valueSim(e_j,e_z) \leq valueSim(e_i,e_z) + valueSim(e_j,e_j)$.

□

Note that an interesting property of $valueSim$ is that the maximum contribution of a single common token between two descriptions is 1, in the case this common token does not appear in the values of any other entity description, i.e., when $EF_{\mathscr{E}_1}(t) \cdot EF_{\mathscr{E}_2}(t) = 1$. Note also that $valueSim$ is a *schema-free similarity metric*, as it completely disregards any schema or domain knowledge.

4.3.2 Entity similarity based on neighbors

In addition to value similarity, we exploit the relations between descriptions to find the matching entities of the compared KBs. This can be done by aggregating the value similarity of all pairs of descriptions that are neighbors of the target descriptions. Formally, we define the neighbor similarity for two descriptions $e_i, e_j \in \mathscr{E}$ as follows:

$$neighborSim(e_i,e_j) = \sum_{\substack{ne_i \in neighbors(e_i)\\ ne_j \in neighbors(e_j)}} valueSim(ne_i,ne_j).$$

Given the potentially high number of neighbors that a description might have, we propose considering only the most valuable neighbors for computing the neighbor similarity between two target descriptions. These are neighbors that are connected with the target descriptions via important relations, i.e., relations that exhibit high *support* and *discriminability*. Intuitively, high support for a particular relation p indicates that p appears in many entity descriptions, while high discriminability for p indicates that it has many distinct values:

Definition 4.2 (Support of relation)**.** *The **support of a relation** $p \in \mathscr{P}$ in an entity collection $\mathscr{E}$ is: $support(p) = \frac{|instances(p)|}{|\mathscr{E}|^2}$, where $instances(p) = \{(i,j) | e_i, e_j \in \mathscr{E}, (p,j) \in e_i\}$.*

Definition 4.3 (Discriminability of relation)**.** *The **discriminability of a relation** $p \in \mathscr{P}$ in an entity collection $\mathscr{E}$ is: $discriminability(p) = \frac{|objects(p)|}{|instances(p)|}$, where $objects(p) = \{j | (i,j) \in instances(p)\}$.*

Overall, we combine support and discriminability via the F-measure in order to locate the most important relations.

Definition 4.4 (Importance of relation)**.** *The **importance of a relation** $p \in \mathscr{P}$ in an entity collection $\mathscr{E}$ is: $importance(p) = 2 \cdot \frac{support(p) \cdot discriminability(p)}{support(p) + discriminability(p)}$.*

Furthermore, we identify the most valuable relations and neighbors for every single entity description. We use $topNrelations(e_i)$ to denote the N relations in $relations(e_i)$ with the maximum importance scores. Then, the **best neighbors** for e_i are:

$$topNneighbors(e_i) = \{ne_i | (p, ne_i) \in e_i, p \in topNrelations(e_i)\}.$$

Intuitively, strong matching evidence (high $valueSim$) for the important neighbors leads to strong matching evidence for the target pair of descriptions. Hence, we define neighbor similarity as:

Definition 4.5 (Neighbor similarity)**.** *Let $\mathscr{E}_1$, $\mathscr{E}_2$ be two entity collections. The **neighbor similarity** of two entity descriptions $e_i \in \mathscr{E}_1, e_j \in \mathscr{E}_2$ is:*

$$neighborNSim(e_i, e_j) = \sum_{\substack{ne_i \in topNneighbors(e_i) \\ ne_j \in topNneighbors(e_j)}} valueSim(ne_i, ne_j).$$

Proposition 4.2. *$neighborNSim$ is a similarity metric.*

Proof. $neighborNSim$ is the sum of similarity metrics ($valueSim$), so it is also a similarity metric [19]. □

Example 4.1. *Continuing our example in Figure 4.1, assume that the best 2 relations for $Restaurant1$ and $Restaurant2$ are: $top2relations(Restaurant1)$ = {hasChef, territorial} and $top2relations(Restaurant2)$ = {headChef, county}. Then, $top2neighbors(Restaurant1)$ = {John Lake A, Bray} and $top2neighbors(Restaurant2)$ = {Jonny Lake, Berkshire}, and $neighbor2Sim(Restaurant1, Restaurant2) = valueSim(Bray, Jonny\ Lake) + valueSim(John\ Lake\ A, Berkshire) + valueSim(Bray, Berkshire) + valueSim(John\ Lake\ A, Jonny\ Lake)$.*

From every entity collection, we derive the k attributes of highest importance, with their values acting as names for any description e_i that contains any of these attributes. Their support is simply defined as $support(p) = |subjects(p)|/|\mathscr{E}|$, where $subjects(p) = \{i|(i,j) \in instances(p)\}$ [93]. Based on these statistics, function $name(e_i)$ returns the names of e_i, and $\mathscr{N}_x$ denotes all names in an entity collection $\mathscr{E}_x$.

4.4 Blocking

4.4.1 Composite Blocking Scheme

To achieve a good trade-off between effectiveness and efficiency, our *schema-free composite blocking scheme* assesses name- and value-based similarities of the candidate matches in conjunction with evidence provided by comparing value-wise their neighbors on the most important relations. We consider the blocks constructed for all entities $e_i \in \mathscr{E}$ using the indexing function $h_i(\cdot)$ both over entity names $(\forall n_j \in names(e_i) : h_N(n_j))$ and tokens $(\forall t_j \in tokens(e_i) : h_T(t_j))$. The composite blocking scheme of MinoanER is defined by the following disjunctive co-occurrence condition of any two entities $e_i, e_j \in \mathscr{E}$:

$$\mathscr{F}(e_i, e_j) = o_N(e_i, e_j) \vee o_T(e_i, e_j) \vee \Big(\bigvee_{(e'_i, e'_j) \in topNneighbors(e_i) \times topNneighbors(e_j)} o_T(e'_i, e'_j) \Big).$$

It is worth noticing that token blocking (i.e., h_T) allows for deriving $valueSim$ from the size of blocks that are shared by two descriptions. As a result, no additional blocks are needed to assess neighbor similarity of candidate entities: token blocking is sufficient also for estimating $neighborNsim$ according to Definition 4.5.

4.4.2 Disjunctive Blocking Graph

The disjunctive blocking graph G is an abstraction of the disjunctive co-occurrence condition of candidate matches in blocks. Nodes represent candidates from our input entity descriptions, while edges represent pairs of candidates for which at least one of the co-occurrence conditions is 'true'. Each edge is labeled with three weights, quantifying similarity evidence on names, tokens and neighbors of candidate entities. Specifically, the disjunctive blocking graph of MinoanER is a graph $G = (V, E, \lambda)$ (see Definition 3.2), where λ assigns to each edge a label (α, β, γ), where α is '1' if $o_N(e_i, e_j)$ is true and the name block in which e_i, e_j co-occur is of size 2, and '0' otherwise, $\beta = valueSim(e_i, e_j)$, and $\gamma = neighborNSim(e_i, e_j)$. Definition 3.2 covers the cases of an entity collection $\mathscr{E}$ being composed of one, two, or more KBs. When matching k KBs, assuming that each is clean, the disjunctive blocking graph is k-partite, with each of the k KBs corresponding to a different independent set of nodes, i.e., there are only edges between descriptions from different KBs. The only information needed to match multiple KBs is to which KB each description belongs, so as to add it to the corresponding independent set.

Example 4.2. *Consider the graph of Figure 4.1(b), which is part of the disjunctive blocking graph generated from Figure 4.1(a). John Lake A and Jonny Lake have a common name ("J. Lake"), and there is no other entity having this name, so there is an edge connecting them with $\alpha = T$. Bray and Berkshire have some common, quite infrequent tokens in their values, so their value similarity, reflected in the β score of the edge that connects them, is quite high (1.2). Since Bray is a top neighbor of Restaurant1 in Figure 4.1(a), and Berkshire is a top neighbor of Restaurant 2, there is also an edge with a non-zero γ connecting Restaurant1 with Restaurant2. Specifically, the γ score of this edge (1.6) is the sum of the β scores of the edges connecting Bray with Berkshire (1.2), and John Lake A with Jonny Lake (0.4), the other two neighbors of Restaurant1 and Restaurant2.*

4.4.3 Graph Weighting and Pruning Algorithms

Each edge in the blocking graph represents a suggested comparison between two descriptions. To reduce the number of comparisons suggested by the disjunctive blocking graph, we keep for each node the K edges with the highest β and the K edges with the highest γ weights, while *pruning* edges with trivial weights (i.e., with α=0, β=0 and γ=0), since they connect descriptions unlikely to match. Given that nodes v_i and v_j may have different top K edges based on β or γ, we consider each undirected edge in G as two directed ones, with the same initial weights, and perform pruning on them.

Example 4.3. *Figure 4.1(c) shows the pruned version of the graph in Figure 4.1(b). Note that the blocking graph is only a conceptual model, which we do not actually materialize; instead, we retrieve any necessary information from computationally cheap entity indices.*

The process of weighting and pruning the edges of the disjunctive blocking graph is described in Algorithm 2. Initially, the graph contains no edges. We start adding edges by checking the name blocks (Lines 5-9). For each name block that contains exactly two entities, one from each KB, we create an edge with $\alpha = 1$ linking those entities[3]. Then, we compute the β weights (Lines 10-20) by running a variation of Meta-blocking [82], adapted to our value similarity metric (Definition 4.1). We keep for each entity, its connected nodes from the K edges with the highest β. In Line 20, we compute the $topNneigbors$ of each entity, and get their reverse, i.e., for each entity e_i we get the entities $inNeighbors[i]$ that have e_i as one of their $topNneighbors$. To avoid re-computing the value similarities that are necessary for the γ computations, we exploit the already computed βs. Thus, we assign to each pair of $inNeighbors$ (in the entity graph) in_i, in_j of the entities e_i, e_j connected with an edge with $\beta > 0$, a partial γ equal to this β (Lines 20-26). After summing the partial γs computed for each pair of entities (in_i, in_j) from all its out-neighbor pairs (e_i, e_j) in the entity graph, we get the correct $\gamma = neighborNsim(in_i, in_j)$ (Definition 4.5).

The time complexity of Algorithm 2 is dominated by the processing of value evidence, which iterates twice over all comparisons in the token blocks B_T. In the worst-case, this results in one

[3]When we add a new edge, we initially set its weight to $(\alpha = 0, \beta = 0, \gamma = 0)$ and update its α, β or γ weight next. Also, b_k^l, $l \in \{1,2\}$, denotes the sub-block of b_k that contains all entities from $\mathcal{E}_l$, i.e., $b_k^l \subset \mathcal{E}_l$.

Algorithm 2: Disjunctive Blocking Graph Construction.

```
Input: $\mathcal{E}_1, \mathcal{E}_2$, set of blocks from name and token blocking, $B_N$ and $B_T$, resp.
Output: A disjunctive blocking graph $G$.
procedure CompositeBGraph($\mathcal{E}_1, \mathcal{E}_2, B_N, B_T$)
    $V \leftarrow \mathcal{E}_1 \cup \mathcal{E}_2$;
    $E \leftarrow \emptyset$;
    $W \leftarrow \emptyset$;                                   // init. to (0,0,0)
    // Name Evidence
    for $b_k \in B_N$ do
        if $|b_k^1| \cdot |b_k^2| = 1$ then                     // only one comparison in $b_k$
            $e_i \leftarrow b_k^1.get(0)$, $e_j \leftarrow b_k^2.get(0)$;   // get entity descriptions in block
            $E \leftarrow E \cup \{< v_i, v_j >\}$;
            $W \leftarrow W.set(\alpha, < v_i, v_j >, T)$;
    // Value Evidence
    for $e_i \in \mathcal{E}_1$ do
        $\beta[] \leftarrow \emptyset$;                         // value weights wrt all $e_j \in \mathcal{E}_2$ init. to 0
        for $b_k \in B_T \wedge b_k \cap e_i \neq \emptyset$ do
            for $e_j \in b_k^2$ do                              // $e_j \in \mathcal{E}_2$
                $\beta[j] \leftarrow \beta[j] + 1/log_2(|b_k^1| \cdot |b_k^2| + 1)$;   // $valueSim$
        $ValueCandidates \leftarrow getTopCandidates(\beta[], K)$;
        for $e_j \in ValueCandidates$ do
            $E \leftarrow E \cup \{< v_i, v_j >\}$;
            $W \leftarrow W.set(\beta, < v_i, v_j >, \beta[j])$;
    for $e_i \in \mathcal{E}_2$ do ...;                         // ...do the same for $\mathcal{E}_2$
    // Neighbor Evidence
    $inNeighbors[] \leftarrow getTopInNeighbors(\mathcal{E}_1, \mathcal{E}_2)$;
    $\gamma[][] \leftarrow \emptyset$;                          // neighbor weights wrt all $e_i, e_j \in V$ init. to 0
    for $e_i \in \mathcal{E}_1$ do
        for $e_j \in \mathcal{E}_2$, s.t. $W.get(\beta, < v_i, v_j >) > 0$ do
            for $in_j \in inNeighbors[j]$ do
                for $in_i \in inNeighbors[i]$ do                // $neighborNSim$
                    $\gamma[in_i][in_j] \leftarrow \gamma[in_i][in_j] + W.get(\beta, < n_i, n_j >)$;
    for $e_i \in \mathcal{E}_2$ do ...;                         // ...do the same for $\mathcal{E}_2$
    for $e_i \in \mathcal{E}_1$ do
        $NeighborCandidates \leftarrow getTopCandidates(\gamma[i], K)$;
        for $e_j \in NeighborCandidates$ do
            $E \leftarrow E \cup \{< v_i, v_j >\}$;
            $W.set(\gamma, < v_i, v_j >, \gamma[v_i][v_j])$;
    for $e_i \in \mathcal{E}_2$ do ...;                         // ...do the same for $\mathcal{E}_2$
    return $G = (V, E, W)$;
procedure getTopInNeighbors($\mathcal{E}_1, \mathcal{E}_2$)
    $topNeighbors[] \leftarrow \emptyset$;                      // one list for each entity
    $r1Sorted \leftarrow$ sort $\mathcal{E}_1$'s relations by importance;
    $r2Sorted \leftarrow$ sort $\mathcal{E}_2$'s relations by importance;
    for $e \in \mathcal{E}_1$ do
        $sortedRel(e) \leftarrow relations(e).sortBy(r1Sorted)$;
        $topNrelations(e) \leftarrow sortedRel(e).topN$;
        for $(p, o) \in e$, where $p \in PQ$ do
            $topNeighbors[e].add(o)$;
    for $e_i \in \mathcal{E}_2$ do ...;                         // ...do the same for $\mathcal{E}_2$
    $topInNeighbors[] \leftarrow \emptyset$;                    // the reverse of topNeighbors
    for $e \in \mathcal{E}_1 \cup \mathcal{E}_2$ do
        for $ne \in topNeighbors[e]$ do
            $topInNeighbors[ne].add(e)$;
    return $topInNeighbors$;
```

computation for every pair of entities, i.e., $O(|\mathcal{E}_1|\cdot|\mathcal{E}_2|)$. In practice, though, we bound the number of computations by removing excessively large blocks that correspond to highly frequent tokens (e.g., stop-words). Following [82], this is carried out by Block Purging, which ensures that the resulting blocks involve two orders of magnitude fewer comparisons than the brute-force approach, without any significant impact on recall. This complexity is higher than that of name and neighbor evidence, which are both linearly dependent on the number of input entities. The former involves a single iteration over the name blocks B_N, which amount to $|\mathcal{N}_1 \cap \mathcal{N}_2|$, as there is one block for every name shared by $\mathcal{E}_1$ and $\mathcal{E}_2$. For neighbor evidence, Algorithm 2 checks all pairs of N neighbors between every entity e_i and its K most value-similar descriptions, performing $K \cdot N^2 \cdot (|\mathcal{E}_1| + |\mathcal{E}_2|)$ operations; the cost of estimating the top in-neighbors for each entity is dominated by the ordering of all relations in $\mathcal{E}_1$ and $\mathcal{E}_2$ (i.e., $|R_{max}| \cdot log|R_{max}|$), where $|R_{max}|$ stands for the maximum number of relations in one of the KBs.

4.5 Non-Iterative Matching

Our matching method receives as input the disjunctive blocking graph G and performs four steps – unlike most existing works, which involve a data-driven iterative process. In every step, a heuristic is applied with the goal of extracting new matches from the edges of G by analyzing their weights. The functionality of our algorithm is outlined in Algorithm 3. Next, we describe its heuristics in the order they are applied:

Name Heuristic (H1). The matching evidence of H1 comes from the entity names. It assumes that *two candidate entities match, if they, and only they, have the same name.* Thus, H1 traverses G and for every edge $< v_i, v_j >$ with $\alpha = \text{T1}$, it updates the set of matches M with the corresponding descriptions (Lines 2-4). All candidates matched by H1 are not examined by the remaining heuristics.

Value Heuristic (H2). It presumes that *two entities match, if they, and only they, share a common token, or, if they share many infrequent tokens.* Based on Definition 4.1, H2 identifies pairs of descriptions with high value similarity (Lines 5-9). To this end, it goes through every node v_i of G and checks whether the corresponding description stems from the smaller in size KB, for efficiency reasons, e.g., $\mathcal{E}_1$, but has not been matched yet. In this case, it locates the adjacent node v_j with the maximum β weight (Line 7). If $\beta \geq 1$, H2 considers the pair (e_i, e_j) to be a match. Matches identified by H2 will not be considered in the sequel.

Rank Aggregation Heuristic (H3). This heuristic identifies further matches for candidates whose value similarity is low ($\beta < 1$), yet their neighbor similarity (γ) could be high. In this respect, the order of candidates rather than their similarity values are used. Its functionality appears in Lines 10-23 of Algorithm 3. In essence, H3 traverses all nodes of G that correspond to a description that has not been matched yet. For every such node v_i, it retrieves two lists: the first one contains adjacent edges with a non-zero β weight, sorted in descending order (Line 13), while the second one includes the adjacent edges sorted in decreasing non-zero γ weights (Line 18). Then, H3

Algorithm 3: Evidence-based Matching.

```
Input: $\mathcal{E}_1$,$\mathcal{E}_2$, Disjunctive Blocking Graph $G$.
Output: A set of matches $M$.
1  $M \leftarrow \emptyset$;                                                  // The set of matches
   // Name Heuristic (H1)
2  for $< v_i, v_j > \in G.E$ do
3      if $G.W.get(\alpha, < v_i, v_j >) = T$ then
4          $M \leftarrow M \cup (e_i, e_j)$;
   // Value Heuristic (H2)
5  for $v_i \in G.V$ do
6      if $e_i \in \mathcal{E}_1 \setminus M$ then
7          $v_j \leftarrow argmax_{v_k \in G.V} G.W.get(\beta, < v_i, v_k >)$ ;        // top candidate
8          if $G.W.get(\beta, < v_i, v_j >) \geq 1$ then
9              $M \leftarrow M \cup (e_i, e_j)$;
   // Rank Aggregation Heuristic (H3)
10 for $v_i \in G.V$ do
11     if $e_i \in \mathcal{E}_1 \cup \mathcal{E}_2 \setminus M$ then
12         $agg[] \leftarrow \emptyset$;                    // Aggregate candidate scores, init. zeros
13         $valCands \leftarrow G.valCand(e_i)$ ;                // nodes linked to $e_i$ in decr. $\beta$
14         $rank \leftarrow |valCands|$;
15         for $e_j \in valCands$ do
16             $agg[e_i].update(e_j, \theta \cdot rank/|valCands|)$;
17             $rank \leftarrow rank - 1$;
18         $ngbCands \leftarrow G.ngbCand(e_i)$ ;                // nodes linked to $e_i$ in decr. $\gamma$
19         $rank \leftarrow |ngbCands|$;
20         for $e_j \in ngbCands$ do
21             $agg[e_i].update(e_j, (1-\theta) \cdot rank/|ngbCands|)$;
22             $rank \leftarrow rank - 1$;
23     $M \leftarrow M \cup (e_i, getTopCandidate(agg[e_i])$;
   // Reciprocity Heuristic (H4)
24 for $(e_i, e_j) \in M$ do
25     if $< v_i, v_j > \notin G.E \vee < v_j, v_i > \notin G.E$ then
26         $M \leftarrow M \setminus (e_i, e_j)$;
27 return $M$;
```

aggregates the two lists by considering the normalized ranks of their elements: assuming the size of a list is K, the first candidate gets the score K/K, the second one $(K-1)/K$, while the last one $1/K$. Overall, each adjacent node of v_i takes a score equal to the weighted summation of its normalized ranks in the two lists, as determined through the trade-off parameter $\theta \in (0,1)$ (Lines 16 & 21): the scores of the β list are weighted with θ and those of the γ list with 1-θ. At the end, we keep for v_i, its top-1 candidate match v_j, i.e., the one with the highest aggregate score (Line 23). Intuitively, H3 *matches e_i with e_j, when, based on all available evidence, there is no better candidate for e_i than e_j.*

Reciprocity Heuristic (H4). It aims to clean the matches identified by H1, H2 and H3 by exploiting the reciprocal edges of G. Given that G becomes a directed graph after pruning, a pair of nodes v_i and v_j are reciprocally connected when there are two edges between them, i.e., an edge from v_i to v_j and an edge from v_j to v_i. Hence, H4 aims to improve the precision of our algorithm based on the rationale that two entities are unlikely to match, when one of them does not even consider the other to be a candidate for matching. Intuitively, *two entity descriptions match, only if both of them "agree" that they are likely to match.* H4 essentially iterates over all matches detected by the above heuristics and discards those missing any of the two directed edges (Lines 24-26).

Given a pruned disjunctive blocking graph, every heuristic can be formalized as a function that receives a pair of entities and returns true (T) if the entities match according to the heuristic's rationale, or false (F) otherwise, i.e., : $Hn : \mathcal{E}_1 \times \mathcal{E}_2 \rightarrow \{T, F\}$. In this context, we formally define the MinoanER matching process as follows:

Definition 4.6. *The **non-iterative matching** of two KBs $\mathcal{E}_1$, $\mathcal{E}_2$, denoted by the Boolean matrix $M(\mathcal{E}_1, \mathcal{E}_2)$, is defined as a filtering problem of the pruned disjunctive blocking graph G: $M(e_i, e_j) = (H1(e_i, e_j) \vee H2(e_i, e_j) \vee H3(e_i, e_j)) \wedge H4(e_i, e_j)$.*

The time complexity of Algorithm 3 is dominated by the size of the pruned blocking graph G it receives as input, since H1, H2 and H3 essentially go through all directed edges in G (in practice, though, H1 reduces the edges considered by H2 and H3, and so does H2 for H3). In the worst case, G contains $2K$ directed edges for every description in $\mathcal{E}_1 \cup \mathcal{E}_2$, i.e., $|V|_{max}$=$2K \cdot (|\mathcal{E}_1| + |\mathcal{E}_2|)$. Thus, the overall complexity is linear with respect to the number of input descriptions, i.e., $O(|\mathcal{E}_1| + |\mathcal{E}_2|)$, which indicates high scalability.

Example 4.4. *To illustrate our matching algorithm, consider the pruned disjunctive blocking graph of Figure 4.2 (a, left), in which nodes $e1 - e3$ represent three entity descriptions from a collection $\mathcal{E}_1$, and nodes $e4 - e9$ represent 6 entity descriptions from another collection $\mathcal{E}_2$. Assuming that we want to keep the top-2 candidates per node based on the β and γ weights, the corresponding candidate lists per entity are shown in Figure 4.2 (a, right), along with the pruned candidates. First, we treat the edges with $\alpha = 1$ as matches and remove them from the graph, along with the nodes they connect (H1). This would return ($e1, e7$) as a match and remove $e1$ and $e7$ from the remaining lists of candidates. Next, we consider the edges with $\beta \geq 1$ as matches (H2) and update the graph accordingly, as shown in Figure 4.2 (b). This would return ($e2, e6$) as a match and remove $e2$ and $e6$ from the remaining lists of candidates. For each remaining node of $\mathcal{E}_1$, we take an aggregate score from its edges and return the adjacent node with the maximum score as a match (H3). At this point, only $e3$ has been left in $\mathcal{E}_1$, with its only candidates being $e8$ and $e9$, as shown in Figure 4.2 (c). To illustrate the aggregation points for each element of the candidate list, we have marked the first-ranked candidate of each list in Figure 4.2 (c, right) with value 1 and each second-ranked candidate with value 1/2 (since element lists are of maximum size 2 in this example). Assuming an equally weighted aggregation for the sake of simplicity here, we see that the aggregate score of the comparison* $(e3, e8)$

is 2 (1 from $e3$'s candidates list from β, plus 1 from $e3$'s candidates list from γ, in both of which $e8$ is the top candidate), while the aggregate score of the comparison $(e3, e9)$ is 1.5 (0.5 from $e3$'s candidates list from β, in which $e9$ is the second candidate, plus 1 from $e9$'s candidates list from γ, in which $e3$ is the top candidate). This means that the rank aggregation heuristic would return $e3 - e8$ as a match. Finally, from the discovered matches, we keep only those with reciprocal edges (H4). This constraint filters out $(e3, e8)$, since there is an edge from $e3$ to $e8$, but no edge from $e8$ back to $e3$. As a result, the final matches would be $(e1, e7)$ and $(e2, e6)$.

4.5.1 Implementation in Spark

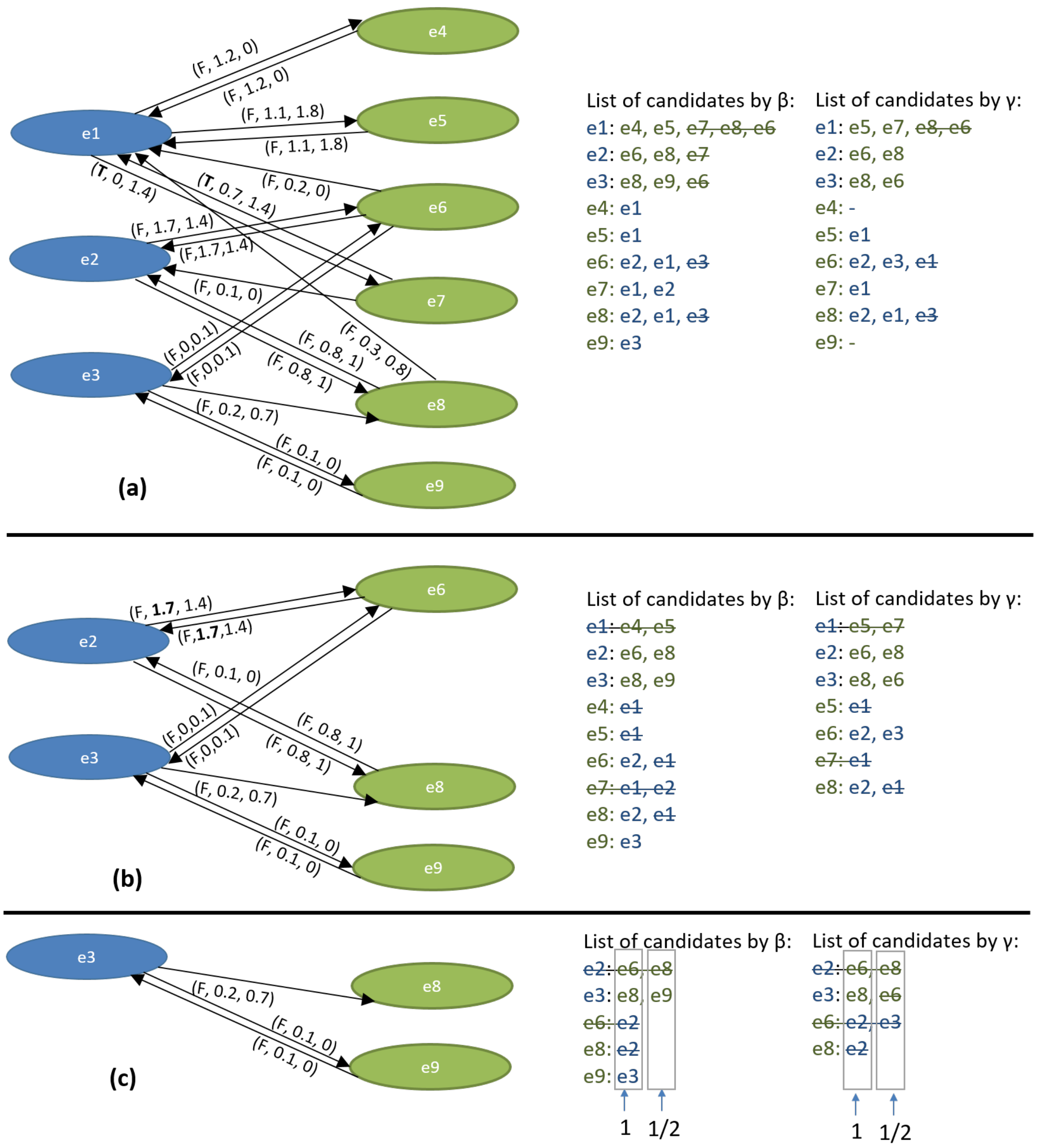
Figure 4.3 shows the architecture of MinoanER's implementation in Spark. Each process is executed in parallel for different chunks of input, in different Spark workers. Each dashed edge represents a synchronization point, at which the process has to wait for results produced by different data chunks (and different Spark workers). As discussed in Section 4.4, our framework employs a composite blocking scheme that is based on two types of blocks: (a) Name blocking applies the name indexing function h_N to place into the same block all entities that have identical names; blocks with exactly 1 entity from each KB produce an edge with $\alpha = 1$. (b) Token blocking applies the token indexing function h_T to place into the same block all entities with a common token in their values; after processing these blocks, all value similarities, i.e., the β weights of the edges, have been computed.

In more detail, we apply name blocking, while running token blocking and the extraction of top neighbors per entity. Then, we synchronize the results of the last two processes: we combine the value similarities computed by token blocking (the β weights) with the top neighbors per entity to estimate the neighbor similarities (the γ weights) for all entity pairs with neighbors co-occurring in at least one block. To minimize the overall run-time, H1 starts right after name blocking, H2 after H1 and token blocking, H3 after H2 and the computation of neighbor similarities, while H4 runs last, providing the final, filtered set of matches. During the execution of every heuristic, each Spark worker contains only the partial information of the disjunctive blocking graph that is necessary to find the match of a specific node (i.e., the corresponding lists of candidates based on names, values, or neighbors).

4.6 Experimental Evaluation

In this section, we compare the effectiveness of MinoanER with state-of-the-art tools and a custom, heavily fine-tuned baseline method.

Experimental Setup. All experiments were performed with Apache Spark v2.1.0 and Java 8, on a cluster of 4 Ubuntu 16.04.2 LTS servers. Each server has 236GB RAM and 36 Intel(R) Xeon(R) E5-2630 v4 @2.20GHz CPU cores. Preliminary experiments have indicated that the following parameter configuration yields the best results for MinoanER across all datasets: K=15 (candidate



(a)

List of candidates by β:	List of candidates by γ:
e1: e4, e5, ~~e7, e8, e6~~	e1: e5, e7, ~~e8, e6~~
e2: e6, e8, ~~e7~~	e2: e6, e8
e3: e8, e9, ~~e6~~	e3: e8, e6
e4: e1	e4: -
e5: e1	e5: e1
e6: e2, e1, ~~e3~~	e6: e2, e3, ~~e1~~
e7: e1, e2	e7: e1
e8: e2, e1, ~~e3~~	e8: e2, e1, ~~e3~~
e9: e3	e9: -

(b)

List of candidates by β:	List of candidates by γ:
~~e1: e4, e5~~	~~e1: e5, e7~~
e2: e6, e8	e2: e6, e8
e3: e8, e9	e3: e8, e6
e4: ~~e1~~	e5: ~~e1~~
e5: ~~e1~~	e6: e2, e3
e6: e2, ~~e1~~	~~e7: e1~~
~~e7: e1, e2~~	e8: e2, ~~e1~~
e8: e2, ~~e1~~	
e9: e3	

(c)

List of candidates by β:	List of candidates by γ:
~~e2: e6, e8~~	~~e2: e6, e8~~
e3: e8, e9	e3: e8, ~~e6~~
~~e6: e2~~	~~e6: e2, e3~~
e8: ~~e2~~	e8: ~~e2~~
e9: e3	

Figure 4.2: An example of running our heuristics on a pruned disjunctive blocking graph.

matches per entity from values and neighbors), N=3 (most important relations per entity), k=2 (most important attributes per KB, whose values act as names), and θ=0.6 (trade-off between value-based over neighbor-based candidates). Regarding the number of most important relations and names, we interpret that those small numbers work well, because there are only few charac-

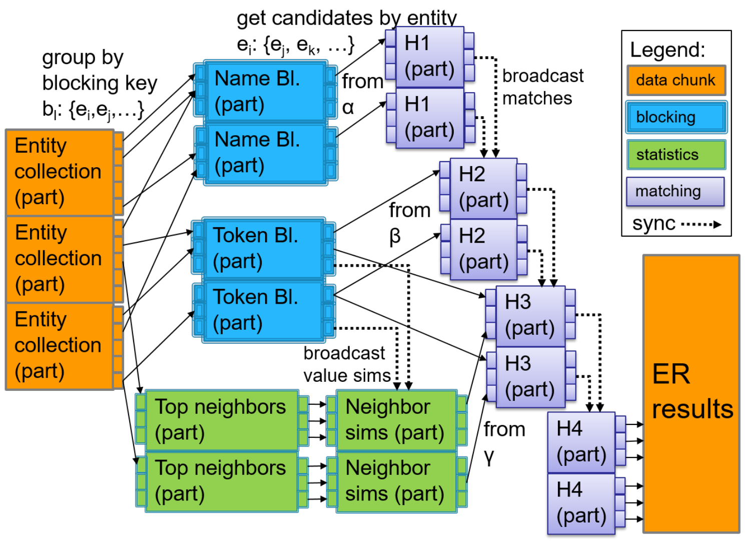


Figure 4.3: The architecture of MinoanER in Spark.

teristics of an entity that make it unique (e.g., its name, and its relation to very few unique other entities, e.g., parents, director). We are currently considering setting those parameters dynamically, based on the local similarity distributions of each node's candidates.

Datasets. In our experiments, we use 4 benchmark datasets with real data commonly used in the literature. Table 4.1 presents their technical characteristics. All KBs contain relations between the described entities.

Restaurant[4] contains descriptions of restaurants and their addresses from two different KBs. The ground truth contains matches only between restaurants, with 23 out of the 112 matches in the original ground truth referring to missing entities; as a result, we consider only 89 verified matches. This is the dataset with the highest value and neighbor similarity between matches (Figure 1.4). It is also the smallest dataset in terms of the number of entities, triples, entity types[5], as well as the one using the smallest number of vocabularies. Hence, it involves the easiest pair of KBs to resolve.

Rexa-DBLP[6] contains descriptions of publications and their authors. The ground truth contains matches between both publications and authors. This dataset contains strongly similar matches in terms of values and neighbors (Figure 1.4). Although it is relatively easy to resolve, Table 4.1 shows that it exhibits the greatest difference with respect to the size of the KBs to be matched (DBLP is 2 orders of magnitude bigger than Rexa in terms of descriptions, and 3 orders of magnitude in terms of triples). We do not distinguish relations to those between entities with global URIs and those between entities with local URIs (i.e., blank nodes), hence, the big number

[4]http://oaei.ontologymatching.org/2010/im/
[5]Extracted using the values of the attribute `w3.org/1999/02/22-rdf-syntax-ns\#type`.
[6]http://oaei.ontologymatching.org/2009/instances/

Table 4.1: KB statistics.

	Restau-rant	Rexa-DBLP	BBCmusic-DBpedia	YAGO-IMDb
$\mathscr{E}_1$ **entities**	339	18,492	58,793	5,208,100
$\mathscr{E}_2$ **entities**	2,256	2,650,832	256,602	5,328,774
$\mathscr{E}_1$ **triples**	1,130	87,519	456,304	27,547,595
$\mathscr{E}_2$ **triples**	7,519	14,936,373	8,044,247	47,843,680
$\mathscr{E}_1$ **av. tokens**	20.44	40.71	81.19	15.56
$\mathscr{E}_2$ **av. tokens**	20.61	59.24	324.75	12.49
$\mathscr{E}_1$ / $\mathscr{E}_2$ **attributes**	7 / 7	114 / 145	27 / 10,953	65 / 29
$\mathscr{E}_1$ / $\mathscr{E}_2$ **relations**	2 / 2	103 / 123	9 / 953	4 / 13
$\mathscr{E}_1$ / $\mathscr{E}_2$ **types**	3 / 3	4 / 11	4 / 59,801	11,767 / 15
$\mathscr{E}_1$ / $\mathscr{E}_2$ **vocab.**	2 / 2	4 / 4	4 / 6	3 / 1
Matches	89	1,309	22,770	56,683

of relations in this dataset.

BBCmusic-DBpedia [34] containsdescriptions of musicians, bands and their birthplaces, from BBCmusic and the BTC2012 version of DBpedia[7]. In our experiments, we consider only entities appearing in the ground truth, as well as their immediate in- and out-neighbors. The most challenging characteristic of this dataset is the high heterogeneity between its two KBs in terms of both schema and values: DBpedia contains ~11,000 different attributes, ~60,000 entity types, 953 relations, the highest number of different vocabularies (6), while using on average 4 times more tokens to describe an entity than the average entity described in BBCmusic.

Based only on the latter feature, all normalized, set-based similarity measures like Jaccard fail to identify such matches, since a big difference in the token set sizes yields low similarity values (Figure 1.4). We have previously shown in Section 2.5.2 that in the median case, an entity description in this dataset contains only 2 words in its values that are used by both KBs.

YAGO-IMDb [97] contains descriptions of movie-related entities (e.g., actors, directors, movies) from YAGO and IMDb[8]. Figure 1.4 shows that a large number of matches in this dataset has low value similarity, while a significant number of them has high neighbor similarity. Moreover, this is the biggest dataset in terms of entities and triples, challenging the scalability of ER tools, while it is the most balanced pair of KBs with respect to their relative size.

Baselines. In our experiments, we compare MinoanER against four state-of-the-art methods: SiGMa, PARIS, LINDA and RiMOM. We also consider a custom baseline method, called BSL. This method receives as input the disjunctive blocking graph G, before its pruning, and compares every pair of descriptions that are connected by an edge in G. The resulting similarities are then processed by Unique Mapping Clustering. Unlike our approach, though, BSL disregards all evidence from neighboring descriptions. Instead, it relies exclusively on value similarity, but optimizes its performance through a series of well-established string matching methods that undergo extensive fine-tuning on the basis of the ground-truth.

[7]`datahub.io/dataset/bbc-music`, `km.aifb.kit.edu/projects/btc-2012/`
[8]`http://www.yago-knowledge.org/`, `http://www.imdb.com/`

Table 4.2: Block statistics.

	Restaurant	Rexa-DBLP	BBCmusic-DBpedia	YAGO-IMDb
$\|B_N\|$	83	15,912	28,844	580,518
$\|B_T\|$	625	22,297	54,380	495,973
$\|\|B_N\|\|$	83	$6.71 \cdot 10^7$	$1.25 \cdot 10^7$	$6.59 \cdot 10^6$
$\|\|B_T\|\|$	$1.80 \cdot 10^3$	$6.54 \cdot 10^8$	$1.73 \cdot 10^8$	$2.28 \cdot 10^{10}$
$\|\mathscr{E}_1\| \cdot \|\mathscr{E}_2\|$	$7.65 \cdot 10^5$	$4.90 \cdot 10^{10}$	$1.51 \cdot 10^{10}$	$2.78 \cdot 10^{13}$
Pr / Re	4.95 / 100	$1.81 \cdot 10^{-4}$ / 99.77	0.01 / 99.83	$2.46 \cdot 10^{-4}$ / 99.35
F1	9.43	$3.62 \cdot 10^{-4}$	0.02	$4.92 \cdot 10^{-4}$

Table 4.3: Evaluation of MinoanER compared to existing methods.

		Restau-rant	Rexa-DBLP	BBCmusic-DBpedia	YAGO-IMDb
SiGMa [66]	Pr / Re	99 / 94	97 / 90	- / -	98 / 85
	F1	97	94	-	91
PARIS [97]	Pr / Re	95 / 88	93.95 / 89	19.40 / 0.29	94 / 90
	F1	91	91.41	0.51	**92**
LINDA [16]	Pr / Re	100 / 63	- / -	- / -	- / -
	F1	77	-	-	-
RiMOM [68]	Pr / Re	86 / 77	80 / 72	- / -	- / -
	F1	81	76	-	-
BSL	Pr / Re	100 / 100	96.57 / 83.96	85.20 / 36.09	11.68 / 4.87
	F1	**100**	89.82	50.70	6.88
MinoanER	Pr / Re	100 / 100	96.74 / 95.34	91.44 / 88.55	91.02 / 90.57
	F1	**100**	**96.04**	**89.97**	90.79

In more detail, we examine the performance of BSL using a large number of parameter configurations to detect the best performing one. Four parameters are fine-tuned to maximize its F-measure: *(i)* The schema-free representation of the values in every entity. BSL uses token n-grams for this purpose, with $n \in \{1, 2, 3\}$, thus representing every resource by the token uni-/bi-/tri-grams that appear in its values. *(ii)* The weighting scheme that assesses the importance of every token. We consider TF and TF-IDF weights. *(iii)* The similarity measure, for which we consider the following well-established similarities: Cosine, Jaccard, Generalized Jaccard and SiGMa. All measures are normalized to $[0, 1]$ and SiGMa similarity applies exclusively to TF-IDF weights, by definition [66]. *(iv)* The similarity threshold that prunes the entity pairs processed by Unique Mapping Clustering. We use all thresholds in $[0, 1)$ with a step of 0.05. In total, we consider 420 different configurations for BSL, reporting the one with the highest F-Measure.

4.6.1 Effectiveness Evaluation

Table 4.2 reports the performance of the blocks used by BSL and MinoanER. The number of comparisons in token blocks ($||B_T||$) is at least 1 order of magnitude larger than those of name blocks ($||B_N||$), even if the latter may involve more blocks ($|B_N|>|B_T|$ over YAGO-IMDb). In fact, the comparisons suggested by names seem to depend linearly on the number of input descriptions, whereas the comparisons suggested by tokens seem to depend quadratically on that number. Nevertheless, the overall comparisons in $B_T \cup B_N$ are at least 2 orders of magnitude lower than the Cartesian product $|E_1| \cdot |E_2|$, even though recall (Re) is consistently higher than 99%. Yet, both precision (Pr) and F-Measure (F1) remain rather low.

Table 4.3 reports the performance of MinoanER and the baselines. For every method, we report its Pr, Re and F1 with respect to the descriptions in the first KB appearing in the ground truth. Since PARIS [97] is openly available, we were able to run it on Rexa-DBLP and BBCmusic-DBpedia. For the rest of the tools, we report their performance from the original publications[9].

Table 4.3 shows that MinoanER offers competitive performance when matching KBs with few attributes and entity types, even if it requires no domain-specific input, while significantly outperforming state-of-the-art ER methods for highly heterogeneous KBs. Specifically, it achieves 100% F1 in Restaurant, which is 3% higher than SiGMa, 9% higher than PARIS, and ~20% higher than LINDA and RiMOM. Note that BLS also achieves perfect F1, due to the strongly similar matching entities (Figure 1.4). In Rexa-DBLP, MinoanER also outperforms all existing ER methods. It is 2% better than SiGMa in F1, 4.6% better than PARIS, 20% better than RiMOM, and 6% better than BSL. As explained previously, BBCmusic and DBpedia are, by far, the most heterogeneous KBs. For this reason, PARIS struggles to identify the matches, with BLS performing significantly better, but still very poorly in absolute numbers. In contrast, MinoanER succeeds in identifying 89% of matches with 91% precision, achieving a 90% F1. In YAGO-IMDb, MinoanER achieves similar performance with SiGMa (91% F1), with more identified matches (91% vs 85%), but lower precision (91% vs 98%). Compared to PARIS, its F1 is 1% lower, due to 3% lower precision, even if our recall is better by 1%. Finally, BSL exhibits the worst performance by far, due to the very low value similarity between matching entities in this KB (Figure 1.4).

Comparing the performance of MinoanER (Table 4.3) to that of its input blocks (Table 4.2), precision raises by several orders of magnitude at the cost of slightly lower recall. The lower recall is caused by missed matches close to the lower left corner of Figure 1.4, i.e., with very low value and neighbor similarities. This explains why the impact on recall is larger for BBCmusic-DBpedia and YAGO-IMDb.

Evaluation of Heuristics

Table 4.4 summarizes the individual contribution of each heuristic in Algorithm 3, when executed alone, as well as the overall contribution of neighbor similarity evidence in the matching results.

Name Heuristic (H1). This heuristic achieves both high precision (> 97% in all cases) and a decent recall (> 66% in all cases). Hence, given no other matching evidence, H1 alone yields good matching results, emphasizing on precision, with only an insignificant number of its suggested

[9]RiMOM-IM [91] is also openly available, but without execution instructions.

matches being false positives. To illustrate the importance of this similarity evidence in real KBs, we have marked the matches with identical names in Figure 1.4 as bordered points. Thus, we observe that matches may agree on their names, regardless of their value and neighbor similarity evidence.

Value Heuristic (H2). This heuristic is also very precise (> 90% in all cases), but exhibits a lower recall (> 30%). Nevertheless, even this low recall is not negligible, especially when it complements the matches found from H1. In the case of strongly similar entity descriptions as in the Restaurant dataset, this heuristic alone can identify all the matches with perfect precision.

Rank Aggregation Heuristic (H3). This heuristic is the only one that exploits neighbor similarity evidence and its contribution in terms of matches is not the same in all KBs. For KBs with low value similarity (left part of Figure 1.4), this heuristic is the only solution for finding matches having no/different names. In BBCmusic-DBpedia and YAGO-IMDb, it has the highest contribution in recall and F1 among all other heuristics, with the results for YAGO-IMDb being almost equivalent to those of Table 4.3 (YAGO-IMDb features the lowest value similarities in Figure 1.4). For KBs with medium value similarity (middle part of Figure 1.4), but not sufficient enough to find matches with H2, aggregating neighbor with value similarity is very effective. In Rexa-DBLP, H3 yields almost perfect results. Overall, H3 is the heuristic with the greatest F1 in 3 out of 4 datasets.

Reciprocity Heuristic (H4). Since this is a filtering heuristic, i.e., it does not add new results, we measure its contribution by running the full workflow without it. Thus, the results of this heuristic in Table 4.4 should be compared to the results in Table 4.3. This comparison shows that this heuristic increases the precision of MinoanER, with a small, or no impact on recall. Specifically, it increases the precision of BBCmusic-DBpedia by 1.51%, while its recall is decreased by 1.38%, and in the case of YAGO-IMDb, it improves precision by 0.44% with no negative impact on recall. This results in an increase of 0.04% and 0.21% in F1 for BBCmusic-DBpedia and YAGO-IMDb, respectively. Overall this heuristic is the weakest one, yielding only a minor improvement in the results of MinoanER.

Contribution of neighbors. To evaluate the contribution of neighbor evidence in the matching results, we have repeated Algorithm 3, without heuristic H3. Note that this experiment is not the same as our baseline; here, we use all the other heuristics, also operating on the pruned disjunctive blocking graph, while the baseline does not use our heuristics and operates on the unpruned graph. The results show that neighbor evidence play a minor or even no role in KBs with strongly similar entities, such as Restaurant and Rexa-DBLP, while having a bigger impact in KBs with nearly similar matches, such as in BBCmusic-DBpedia and YAGO-IMDb (see Figure 1.4). Specifically, compared to the results of Table 4.3, the use of neighbor evidence improves precision by 2.22% and recall by 3.19% in BBCmusic-DBpedia, while, in YAGO-IMDB, precision is improved by 2.97% and recall by 3.15%.

Table 4.4: Evaluation of heuristics.

		Restau-rant	Rexa-DBLP	BBCmusic-DBpedia	YAGO-IMDb
H1	Pr / Re	100 / 68.54	97.36 / 87.47	99.85 / 66.11	97.55 / 66.53
	F1	81.33	92.15	79.55	79.11
H2	Pr / Re	100 / 100	96.15 / 30.56	90.73 / 37.01	98.02 / 69.14
	F1	100	46.38	52.66	81.08
H3	Pr / Re	98.88 / 98.88	94.73 / 94.73	81.49 / 81.49	90.51 / 90.50
	F1	98.88	94.73	81.49	90.50
¬H4	Pr / Re	100 / 100	96.03 / 96.03	89.93 / 89.93	90.58 / 90.57
	F1	100	96.03	89.93	90.58
No Nei-ghbors	Pr / Re	100 / 100	96.59 / 95.26	89.22 / 85.36	88.05 / 87.42
	F1	100	95.92	87.25	87.73

4.6.2 Efficiency Evaluation

To evaluate the scalability of matching in MinoanER[10], we present in Figure 4.4 the running times and speedup of matching for each dataset, as we change the number of available processors in our cluster, i.e., the number of tasks that can run at the same time. In each diagram, the left vertical axis shows the running time and the right vertical axis shows the speedup, as we increase the number of available processors (from 1 to 72) shown in the horizontal axis[11]. Across all experiments, we have kept the same total number of tasks, which was defined as the number of all cores in the cluster multiplied by a parallelism factor of 3, i.e., 3 tasks are assigned to each core, when all cores are available. This was to ensure that each task would require the same amount of resources (e.g., memory and disk), regardless of the number of available cores. We observe that the running times decrease as more processors become available, and this decrease is steeper when using a small number of processors. For example, the matching of Rexa-DBLP with 6 cores is 6 times faster than with 1 core, while it is 10 times faster with 12 cores than with 1 core (top-right of Figure 4.4). Overall, we observe a sub-linear speedup in all cases, which is expected when synchronization is required for different steps (see Section 4.5.1), while bigger datasets have a speedup closer to linear than smaller tasks, since the overhead of Spark is smaller with respect to the processing times in such cases.

It is not possible to directly compare the efficiency of MinoanER with the competitive tools of Table 4.3; most of them are not publicly available, while the available ones do not support parallel execution using Spark. Note that scalability, i.e., a massively parallel architecture, is one of the Web-scale ER requirements that we have set in Section 1.3. Additionally, the running times reported in the original works are about sequential algorithms executed in machines with a different setting than ours. However, we can safely argue that our fixed-step process, as opposed to

[10]The scalability of blocking and Meta-blocking have been already evaluated [33, 36].

[11]We could not run MinoanER on the YAGO-IMDb dataset with only 1 processor, due to limited space in a single machine, so we report its running time starting with a minimum of 4 processors. This means that the linear speedup for 72 tasks would not be 72, but 18 (72/4).

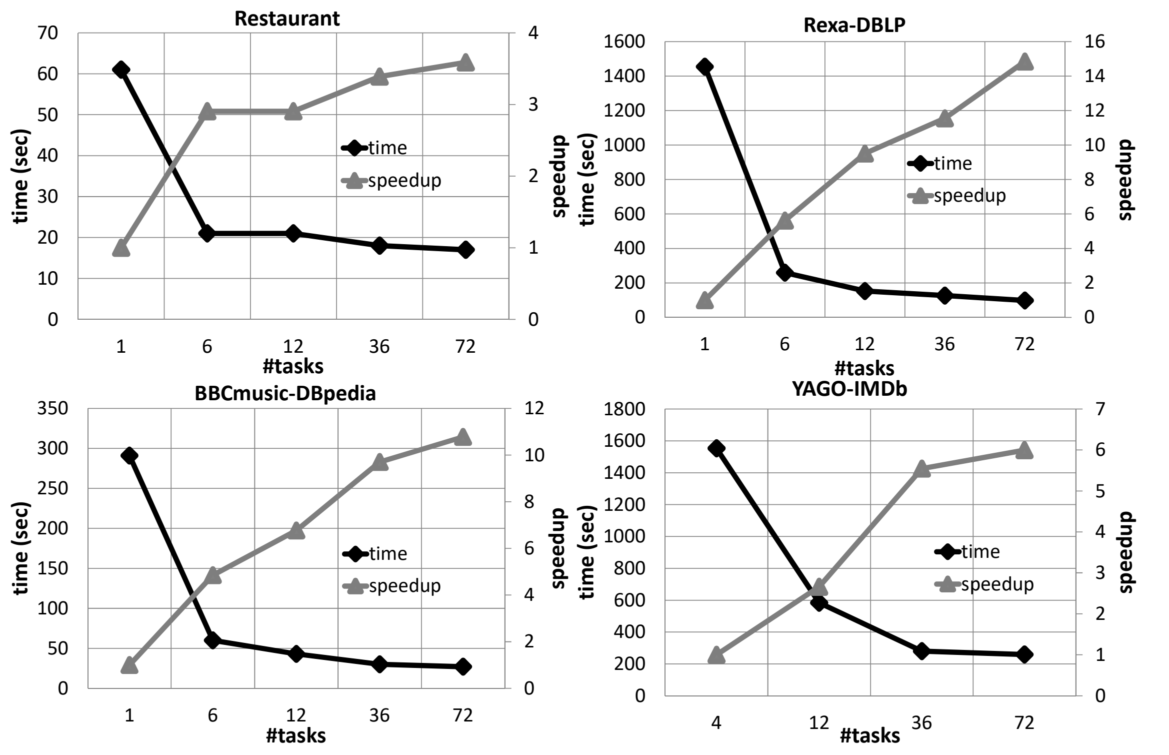


Figure 4.4: Scalability of matching in MinoanER w.r.t. running time (left vertical axis) and speedup (right vertical axis) as more cores are involved.

the data-iterative processes of existing works, boosts the efficiency of MinoanER at no cost in (or, in most cases, with even better) effectiveness. Indicatively, the running time of our framework for Rexa-DBLP was 3.5 minutes (it took PARIS 11 minutes on one of our cluster nodes for the same dataset), for BBCmusic-DBpedia it was 69 seconds (it took PARIS 3.5 minutes on one of our cluster nodes), while the running time for YAGO-IMDb was 28 minutes (SiGMa reports 70 minutes, and PARIS reports 51 hours). In small datasets like Restaurant, our framework can be slower than other tools, as Spark has a setup overhead, which is significant for such cases (it took MinoanER 27 seconds to run this dataset, while PARIS needed 6 seconds).

4.7 Conclusion

In this chapter, we have presented MinoanER, a fully automated, schema-free and massively parallel framework for ER in the Web of data. To resolve highly heterogeneous entities met in this context, we define schema-free similarity metrics that consider both the content and the neighbors of entities. We exploit these metrics in a composite blocking scheme and conceptually build a disjunctive blocking graph, a novel graph-based abstraction of the similarity evidence obtained by blocking. This graph of candidate matches is processed by a non-iterative matching method with linear cost to the number of entity descriptions. This means that we can now identify the matches with low similarity (left part of the diagram in Figures 1.4, 4.5) based on their neighbors and names, even from the step of blocking, without any iteration over previously found matches.

The results show that neighbor evidence plays a minor role in KBs with strongly similar entities, such as Restaurant and Rexa-DBLP, while having a bigger impact in KBs with nearly similar entities, such as in BBCmusic-DBpedia and YAGO-IMDb. MinoanER achieves at least equivalent performance with state-of-the-art ER tools over homogeneous KBs, even without requiring any domain-specific knowledge, e.g., regarding the alignment of relations in the input, or training data. Yet, it outperforms to a significant extent existing ER tools when matching highly heterogeneous KBs, while its parallel implementation in Spark allows it to easily scale to voluminous datasets, as the ones met in the Web of data. The employed heuristics manage to cover a wide area of matches in the diagram of Figure 1.4, as abstractly annotated in Figure 4.5 (H1 covers a big part of the whole diagram, H2 focuses on the right part, and H3 targets the middle-top part), but still some areas of the diagram are not covered sufficiently (e.g., the bottom-left part), or the covered areas are not handled in a perfect manner, since the recall of blocking is still better than that of matching. The difficulty in identifying those matches is also reflected by the overlap of the heuristics in this figure. If for example, a match is missed from H1 in the top-right area, this match has two more chances to be identified (by H2 or H3), so it is easier to identify such matches. If, on the other hand, H1

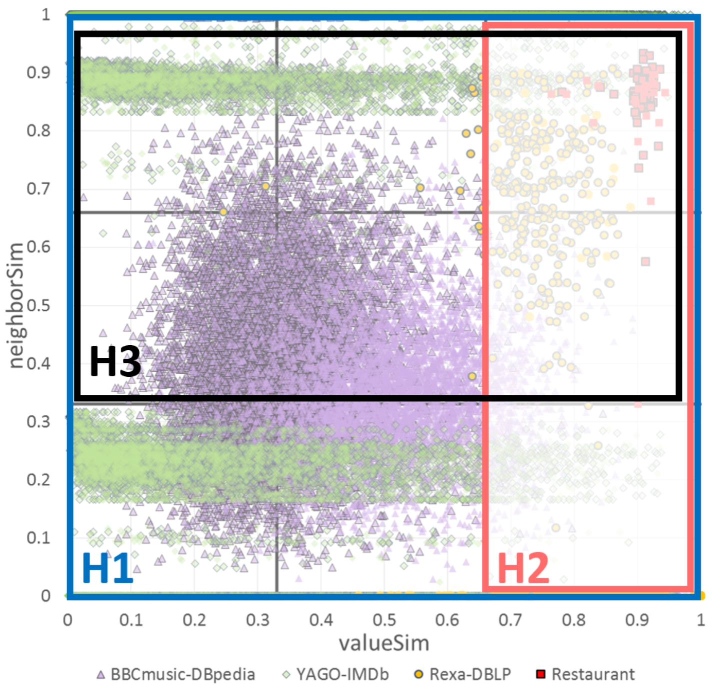


Figure 4.5: The area of matches from Figure 1.4 targeted by each of the employed heuristics.

misses one match in the bottom-left corner of Figure 4.5, then this match will not be identified by any other heuristic. This means that there is still some room for improvement in the heuristics used, that could increase the number of cases covered.

Furthermore, we are currently working on dynamically setting the three parameters that we have discussed in Section 4.6 (number of candidate matches per entity, most important relations per entity, and most important properties per KB acting as names), based on the local similarity distributions of each node's candidates will further increase the flexibility and effectiveness of our framework. For example, when many of the candidate matches have a high value similarity, we expect that increasing the number of candidate matches kept per entity will yield better recall results, while, when the neighbor similarity of candidate matches is low, reducing the number of most important relations should improve precision. The datasets and source code used in MinoanER are publicly available[12].

[12]http://csd.uoc.gr/~vefthym/minoanER/datasets.html

Chapter 5
Conclusion and Future Work

5.1 Synopsis of Contributions

Although Entity Resolution (ER) has been studied for more than three decades in different computer science communities, it still remains an active area of research. In particular, the scale, diversity, and graph structuring of entity descriptions published according to the Linked Data paradigm challenge the core ER tasks, namely, (i) how resolution algorithms can efficiently filter the candidate description pairs that need to be compared and (ii) how descriptions can be effectively compared for similarity. In this thesis, *we introduce the MinoanER framework*, which deals with these challenging ER tasks in the Web of data in the following ways.

Regarding the first task, ER in the Web of data involves a large number of KBs (in the order of hundreds) and even a larger number of entity types in different domains (in the order of thousands) whose published descriptions could be potentially resolved (Chapter 1). Aiming at efficiency, MinoanER employs *blocking* (Chapter 2) and *Meta-blocking* (Chapter 3) to reduce the number of required comparisons. The indexing functions of blocking are *schema-free*, based on the tokens and the names of the entity descriptions, disregarding any assumptions regarding the way entities are described in various KBs. Meta-blocking relies on a novel *disjunctive blocking graph* capturing similarity evidence provided by several atomic blocking methods (i.e., on the content, name and neighbors of descriptions) that can be then efficiently built and pruned, using only on the blocking results. Both blocking and Meta-blocking have been implemented in a massively parallel architecture, to allow scaling to the volumes of KBs met in the Web of data. The experimental results show that our composite blocking and Meta-blocking techniques achieve for both strongly and nearly similar entities a good tradeoff between the *efficiency* and the *effectiveness* of the ER process, by reducing the number of suggested comparisons by more than 90%, while also suggesting the comparisons between more than 90% of the actual matches.

Regarding the second task, MinoanER relies on new similarity metrics, that can *effectively* compare descriptions of different entity types using simple statistics on terms, attributes or relations employed to describe entities even in different domains without involving domain experts (Chapter 4). The similarity evidence provided by these metrics and captured by the disjunctive blocking graph, are exploited by generic heuristics in a non-iterative matching. The matching heuristics are also implemented in a massively parallel architecture, enabling MinoanER to handle ER at the scale of the Web of data. The experimental results show that our non-iterative matching can successfully identify not only strongly similar, but also nearly similar matches met in highly heterogeneous Web KBs.

To our knowledge, MinoanER is the first ER framework that can identify nearly similar matches in a schema-free, fully automated, non-iterative and massively parallel way. The main contribution of this framework to the field, is that it achieves at least equivalent results over homogeneous KBs (stemming from common data sources, thus exhibiting strongly similar matches), and significantly better results over heterogeneous KBs (stemming from different sources, thus exhibiting many nearly similar matches), to state-of-the-art ER tools, without requiring any domain-specific knowledge, e.g., regarding the schema or the alignment of relations in the input, or training data, in a non-iterative and highly efficient way.

5.2 Short-term Improvements

The following improvements could potentially enhance flexibility and performance of MinoanER:

- Dynamically setting the fixed parameters of the matching phase (i.e., number of candidates kept per node after pruning the blocking graph, number of most important relations per entity, and most important properties per KB acting as names) based on the local similarity distribution of each node would strengthen the flexibility of our matching to better adapt to each unique case of matching. This extension aims to achieve the maximum possible recall for matching, which is the recall of blocking. This is similar in logic to the global vs. local pruning of the blocking graph edges: in our current design, we are using only a global strategy, however, considering a local one, we can set different parameter values per entity. Our intuition is that when many of the candidate matches have a high value similarity, increasing the number of candidate matches kept per entity will result in higher recall, while, when the neighbor similarity of candidate matches is low, reducing the number of most important relations should improve precision. Also, when many candidates are tied at the candidate lists, increasing the number of important relations would help breaking those ties, with respect to the neighbor evidence, and increasing the number of important attributes used as entity names could also help in this direction.

- Optimizing the parallel algorithms employed for blocking, Meta-blocking and matching, tailoring them to the specific parallelization environment used is an unexplored field in this work. In our preliminary examples, we have seen a drastic reduce in the running times of our algorithms when moving from MapReduce to Spark. In one setting of Meta-blocking, in which multiple MapReduce jobs were employed, the transition from MapReduce to Spark reduced the running time from 70 minutes to only 10, without any advanced tuning. At the moment, we have only implemented in Spark the Meta-blocking methods that were utilized by MinoanER. We believe that further tuning our algorithms for Spark, and proving that their parallelization is optimal, would yield significant efficiency benefits.

- Extending the matching evidence to indirect neighbors. In this work, we consider only the similarity of direct (hop-1) neighbors to influence the matching decision for a candidate pair. Extending the radius of important neighbors (e.g., to hop-2) may improve the effectiveness of a matching method, probably bringing an overhead in efficiency. Specifically, [67] studies this problem for matching anonymous entity descriptions (blank nodes) with a varying radius, using a textual signature of the entity description and its neighbors (the signature is a string concatenation of all the RDF triples in which the identifier of an entity description participates). It shows that extending the similarity evidence to indirect neighbors can yield better qualitative results, while the signature-based algorithm allows an efficient search even when the radius is greater than 1. As noted in [67], a parallel version of this algorithm is worth noticing, and we believe that a signature-based extension of our work for indirect neighbors could improve the effectiveness of MinoanER, with a small efficiency overhead.

- Support of numerical comparisons, hierarchical data, and stemming. In some cases, treating all values as tokens may have disadvantages. Those are domain-specific problems, such as matching geographical data, which contain many numeric values (e.g., co-ordinates, polygons) and an exact match of values or even tokens of those values may be ineffective. In such cases, we may want to compare values and suggest candidate matches based, not only on the exact match of tokens, but instead, on the proximity of the numeric values (e.g., co-ordinates very close to each other), a containment measure (for polygons describing locations), or a hierarchical comparison (e.g., the facts "bornIn Manhattan" and "bornIn NewYork" should not be considered dissimilar). This could easily integrated to our framework as an additional indexing function (i.e., type of blocking) and the inclusion of the generated blocks in our disjunctive co-occurrence function. Another improvement on the comparison of values could be to employ stemming before blocking, this way handling typos in a better way. The way this additional evidence could be exploited in blocking is also interesting; for this reason, a domain-specific heuristic could be employed.

5.3 Directions for Future Research

There are several aspects that are worth further work and research. Here, we discuss our ongoing and future work regarding Web table annotation, streaming ER, and the benefit of ER.

Web table annotation. In this thesis, we have studied the problem of resolving entities whose descriptions are published in one or two RDF KBs. An interesting research question is how can we resolve entities when one of the KBs is published in a different format, such as in the form of Web tables? Could we exploit some of MinoanER's components to this respect? In our ongoing work, we are trying to provide answers to these questions for the problem of annotating the contents of Web tables with matching descriptions contained in a target RDF KB. The main difference with what we have seen in this thesis is that a typical Web table contains entity descriptions of a single domain with a fixed schema, while the target KB does not. We only consider horizontal tables, in which each row describes a different entity and each column corresponds to a specific attribute, which can also be a relation, or the attribute corresponding to the entity name. We make the assumption that all the entities described in the same column are of the same type and that column headers, typically existing in the first row of a horizontal Web table, do not contain meaningful names which we could exploit for matching them to properties in a KB, as this is an assumption holding for the majority of Web tables [6].

We are currently evaluating different approaches to this problem, whose contextual information vary from poor (in Web tables) to rich (in KBs). First, we examine a *lookup-based method*, which exploits the columns of the Web tables recognized as entity names. It essentially detects correspondences using the minimal contextual information available in Web tables, which is then refined (based on frequently occurring terms in entity descriptions) or enriched (by exploiting relationships with other entities) with respect to the context of entities available in the KB. To do that, it creates an index for the target KB, using token blocking, and then searches this index for the tokens contained in the name column, to create candidate matches. The final matches are then selected using similar heuristics to the ones we used in Chapter 4. In the opposite direction, the *embeddings-based method* exploits a vectorial representation of the rich entity context in a KB (using word2vec [74] algorithm) to identify the most relevant subset of entities in the Web table. Again, blocking is utilized to quickly store and retrieve the names of the entities, which is required to generate candidate matches. The candidate matches are stored as nodes in a graph, in which the cosine similarity of their vectors are used to weight the edges connecting them. The most coherent result-set, which is the set of most visited nodes found by several iterations of a weighted PageRank algorithm [107], is the final annotation result. Our experiments show that the best results are acquired by a combination of these two methods [30, 31].

Progressive ER. Works in progressive ER [3,4,86,100] focus on maximizing the reported matches, given a limited computational budget, by potentially exploiting the partial matching results obtained so far in an iterative ER process. Essentially, they extend the typical ER workflow with a scheduling phase, which is responsible for selecting which candidate matches, suggested by blocking, will be compared in the matching phase and in what order. The goal of this new phase is to favor more promising comparisons, i.e., those that are more likely to result in matches. This way, those comparisons are executed before less promising ones and thus, more matches are identified early on in the process. The partial results of matches are then propagated such that a new scheduling phase will promote the comparison of pairs that were influenced by the previous matches. This iterative process continues until the pre-defined computing budget is consumed.

We believe that the quality of the resulting entity graph after merging the matches, rather than the number of matches, should determine the benefit of the ER process under resource constraints. In this respect, we are interested in measuring the complementary knowledge (similar to the notion of diversity used in information retrieval) that an ER process could achieve in the resulting entity graph. Our intuition is that merges resulting from nearly similar entity descriptions are more beneficial in this respect compared to merges from strongly similar descriptions (i.e., duplicates). Thus, given a constraint in the number of possible merges, we would like to perform those that contribute most in diversifying the knowledge encoded in the resulting entity graph. Complementary knowledge can be measured by the degree of a merged node in an entity graph, excluding overlapping edges. Intuitively, each edge represents a fact or relationship, which we use as a unit of knowledge increase. When two edges overlap, they represent the same knowledge unit, so we do not gain anything by knowing both of them, whereas, when two edges represent two different knowledge units, then they are both useful. Thus, when we merge two matching entities, we want to count only the number of unique knowledge units that the merging brought to our graph. In that sense, nearly similar descriptions provide complementary knowledge units about the entity that they describe and thus maximize the benefit of progressive ER, whereas merging strongly similar descriptions comes with a zero benefit. Thus, we would need to reconsider the order of applying our heuristics during the matching phase, placing the rank aggregation heuristic (H3), targeting nearly similar matches, before the value heuristic (H2), targeting strongly similar matches, or combining the name (H1) and value (H2) heuristics, such that we return first the matches with identical names (i.e., a subset of those returned by H1) that are not strongly similar (i.e., they are not returned by H2).

Streaming ER. We are finally interested in a streaming version of MinoanER, in which we are not asked to find all the matching descriptions between two entity collections, but the matches of descriptions arriving in a streaming fashion against a stored collection of entities (e.g., [3,44,100]). For example, consider an application resolving the entities described across news feeds. A journalist using this application could be provided with several facts regarding a breaking news story, as they get published by different agencies or witnesses, enabling him/her to form a complete picture of the events as they occur, in real-time. This would require storing only some parts of the blocking graph, and discarding the rest, as more descriptions are fed to the system. To evaluate which parts of the graphs are more useful to keep, we can design different strategies. For example, we may want to keep the latest nodes of the graph, since new input entities are more likely to be connected to them, and thus, their resolution is more likely to be helped by those latest nodes. Another strategy would be to keep the nodes with the highest in-degrees, since they are more likely to influence the matching decision of their in-neighbors and new entities appearing are more likely to be connected to those nodes.

The MinoanER framework already considers local matching decisions taken for each node, using only its name, tokens and most important direct neighbors. Hence, the matching of a single node does not require the identification of all existing matches. Also, the indexing functions used by our blocking method are schema-free and non-iterative, which means that entities can be placed in blocks as they come, regardless of the set of attributes that they use. Moreover, the most effective heuristic in terms of matching cases, the name heuristic (H1), can be run immediately with a close-to-zero cost. The value heuristic (H2) could also run at real-time considering the number and size of common token blocks between two descriptions, which, together with H1, would result in a decent streaming ER framework. The biggest challenge for a streaming version of MinoanER would be the incorporation of the rank aggregation heuristic (H3), which requires matching evidence from the neighbors. Streaming ER would probably imply the withdrawal of the reciprocity heuristic (H4), sacrificing a small fraction of precision for the sake of faster processing. Finally, scalability still demands a parallel architecture, since the search space of candidate matches would incrementally increase. Such a streaming version of ER could be supported by distributed stream-processing frameworks such as Spark streaming[1] and Apache Flink[2].

[1]https://spark.apache.org/streaming/
[2]https://flink.apache.org/

Bibliography

[1] Lada A. Adamic and Eytan Adar. Friends and neighbors on the web. *Social Networks*, 25(3):211–230, 2003.

[2] Akiko N. Aizawa and Keizo Oyama. A fast linkage detection scheme for multi-source information integration. In *WIRI*, pages 30–39, 2005.

[3] Yasser Altowim, Dmitri V. Kalashnikov, and Sharad Mehrotra. Progressive approach to relational entity resolution. *PVLDB*, 7(11):999–1010, 2014.

[4] Yasser Altowim and Sharad Mehrotra. Parallel progressive approach to entity resolution using mapreduce. In *ICDE*, pages 909–920, 2017.

[5] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *VLDB*, 2006.

[6] Sreeram Balakrishnan, Alon Y. Halevy, Boulos Harb, Hongrae Lee, Jayant Madhavan, Afshin Rostamizadeh, Warren Shen, Kenneth Wilder, Fei Wu, and Cong Yu. Applying WebTables in Practice. In *CIDR*, 2015.

[7] Krisztian Balog, Marc Bron, and Maarten de Rijke. Category-based query modeling for entity search. In *ECIR*, pages 319–331, 2010.

[8] Krisztian Balog, Edgar Meij, and Maarten de Rijke. Entity search: Building bridges between two worlds. In *SEMSEARCH*, 2010.

[9] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.

[10] Indrajit Bhattacharya and Lise Getoor. Collective entity resolution in relational data. *TKDD*, 1(1), 2007.

[11] Bin Bi, Hao Ma, Bo-June Paul Hsu, Wei Chu, Kuansan Wang, and Junghoo Cho. Learning to recommend related entities to search users. In *WSDM*, pages 139–148, 2015.

[12] Mikhail Bilenko, Beena Kamath, and Raymond J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *ICDM*, 2006.

[13] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. Dbpedia - A crystallization point for the web of data. *J. Web Sem.*, 7(3):154–165, 2009.

[14] Roi Blanco, Berkant Barla Cambazoglu, Peter Mika, and Nicolas Torzec. Entity recommendations in web search. In *ISWC*, pages 33–48, 2013.

[15] Roi Blanco, Peter Mika, and Sebastiano Vigna. Effective and efficient entity search in RDF data. In *ISWC*, pages 83–97, 2011.

[16] Christoph Böhm, Gerard de Melo, Felix Naumann, and Gerhard Weikum. LINDA: distributed web-of-data-scale entity matching. In *CIKM*, pages 2104–2108, 2012.

[17] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci.*, 60(3):630–659, 2000.

[18] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

[19] Shihyen Chen, Bin Ma, and Kaizhong Zhang. On the similarity metric and the distance metric. *Theor. Comput. Sci.*, 410(24-25):2365–2376, 2009.

[20] Peter Christen. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Data-Centric Systems and Applications. Springer, 2012.

[21] Peter Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Trans. Knowl. Data Eng.*, 24(9):1537–1555, 2012.

[22] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. *Entity Resolution in the Web of Data.* Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool Publishers, 2015.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[24] AnHai Doan, Adel Ardalan, Jeffrey R. Ballard, Sanjib Das, Yash Govind, Pradap Konda, Han Li, Sidharth Mudgal, Erik Paulson, Paul Suganthan G. C., and Haojun Zhang. Human-in-the-loop challenges for entity matching: A midterm report. In *HILDA*, pages 12:1–12:6, 2017.

[25] AnHai Doan, Jeffrey F. Naughton, Raghu Ramakrishnan, Akanksha Baid, Xiaoyong Chai, Fei Chen, Ting Chen, Eric Chu, Pedro DeRose, Byron Gao, Chaitanya Gokhale, Jiansheng Huang, Warren Shen, and Ba-Quy Vuong. Information extraction challenges in managing unstructured data. *SIGMOD Rec.*, 37(4):14–20, March 2009.

[26] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge vault: a web-scale approach to probabilistic knowledge fusion. In *SIGKDD*, pages 601–610, 2014.

[27] Xin Luna Dong and Divesh Srivastava. *Big Data Integration*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2015.

[28] Uwe Draisbach and Felix Naumann. A generalization of blocking and windowing algorithms for duplicate detection. In *ICDKE*, pages 18–24, 2011.

[29] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *SIGMOD*, pages 145–156, 2011.

[30] Vasilis Efthymiou, Oktie Hassanzadeh, Mariano Rodriguez-Muro, and Vassilis Christophides. Matching web tables with knowledge base entities: From entity lookups to entity embeddings. In *ISWC (to appear)*, 2017.

[31] Vasilis Efthymiou, Oktie Hassanzadeh, Mohammad Sadoghi, and Mariano Rodriguez-Muro. Annotating web tables through ontology matching. In *OM*, pages 229–230, 2016.

[32] Vasilis Efthymiou, George Papadakis, George Papastefanatos, Kostas Stefanidis, and Themis Palpanas. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *IEEE Big Data*, pages 411–420, 2015.

[33] Vasilis Efthymiou, George Papadakis, George Papastefanatos, Kostas Stefanidis, and Themis Palpanas. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Inf. Syst.*, 65:137–157, 2017.

[34] Vasilis Efthymiou, Kostas Stefanidis, and Vassilis Christophides. Big data entity resolution: From highly to somehow similar entity descriptions in the web. In *IEEE Big Data*, 2015.

[35] Vasilis Efthymiou, Kostas Stefanidis, and Vassilis Christophides. Minoan ER: progressive entity resolution in the web of data. In *EDBT*, pages 670–671, 2016.

[36] Vasilis Efthymiou, Kostas Stefanidis, and Vassilis Christophides. Benchmarking blocking algorithms for web entities. *IEEE Transactions on Big Data*, (to appear), 2017.

[37] Vasilis Efthymiou, Kostas Stefanidis, and Eirini Ntoutsi. Top-k computations in MapReduce: A case study on recommendations. In *IEEE Big Data*, pages 2820–2822, 2015.

[38] Oren Etzioni, Michael J. Cafarella, Doug Downey, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. Unsupervised named-entity extraction from the web: An experimental study. *Artif. Intell.*, 165(1):91–134, 2005.

[39] Pavlos Fafalios, Manolis Baritakis, and Yannis Tzitzikas. Exploiting linked data for open and configurable named entity extraction. *International Journal on Artificial Intelligence Tools*, 24(2), 2015.

[40] Christos Faloutsos and King-Ip Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *SIGMOD*, pages 163–174, 1995.

[41] Lujun Fang, Anish Das Sarma, Cong Yu, and Philip Bohannon. REX: explaining relationships between entity pairs. *PVLDB*, 5(3):241–252, 2011.

[42] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64:1183–1210, 1969.

[43] Alfio Ferrara, Stefano Montanelli, Jan Noessner, and Heiner Stuckenschmidt. Benchmarking matching applications on the semantic web. In *ESWC*, 2011.

[44] Donatella Firmani, Barna Saha, and Divesh Srivastava. Online entity resolution using an oracle. *PVLDB*, 9(5):384–395, 2016.

[45] Vishrawas Gopalakrishnan, Suresh Parthasarathy Iyengar, Amit Madaan, Rajeev Rastogi, and Srinivasan H. Sengamedu. Matching product titles using web-based enrichment. In *CIKM*, 2012.

[46] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[47] Oktie Hassanzadeh, Fei Chiang, Renée J. Miller, and Hyun Chul Lee. Framework for evaluating clustering algorithms in duplicate detection. *PVLDB*, 2(1):1282–1293, 2009.

[48] Mauricio A. Hernàndez and Salvatore J. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, pages 127–138, 1995.

[49] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artif. Intell.*, 194:28–61, 2013.

[50] Eduard H. Hovy, Roberto Navigli, and Simone Paolo Ponzetto. Collaboratively built semi-structured content and artificial intelligence: The story so far. *Artif. Intell.*, 194:2–27, 2013.

[51] Ekaterini Ioannou, Nataliya Rassadko, and Yannis Velegrakis. On generating benchmark data for entity matching. *J. Data Semantics*, 2(1):37–56, 2013.

[52] Robert Isele and Christian Bizer. Learning expressive linkage rules using genetic programming. *PVLDB*, 5(11):1638–1649, 2012.

[53] Robert Isele and Christian Bizer. Active learning of expressive linkage rules using genetic programming. *J. Web Sem.*, 23:2–15, 2013.

[54] Paul Jaccard. Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines. *Bull. Soc. Vaud. Sci. Nat.*, 37:241–272, 1901.

[55] Bernard J. Jansen and Amanda Spink. How are we searching the world wide web? A comparison of nine search engine transaction logs. *Inf. Process. Manage.*, 42(1):248–263, 2006.

[56] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[57] Liang Jin, Chen Li, and Sharad Mehrotra. Efficient record linkage in large data sets. In *DASFAA*, pages 137–146, 2003.

[58] Yuzhe Jin, Emre Kiciman, Kuansan Wang, and Ricky Loynd. Entity linking at the tail: sparse signals, unknown entities, and phrase models. In *WSDM*, pages 453–462, 2014.

[59] Mayank Kejriwal and Daniel P. Miranker. An unsupervised algorithm for learning blocking schemes. In *ICDM*, 2013.

[60] Batya Kenig and Avigdor Gal. MFIBlocks: An effective blocking algorithm for entity resolution. *Inf. Syst.*, 38(6):908–926, 2013.

[61] Lars Kolb, Andreas Thor, and Erhard Rahm. Dedoop: Efficient deduplication with hadoop. *PVLDB*, 5(12):1878–1881, 2012.

[62] Lars Kolb, Andreas Thor, and Erhard Rahm. Load balancing for mapreduce-based entity resolution. In *ICDE*, pages 618–629, 2012.

[63] Lars Kolb, Andreas Thor, and Erhard Rahm. Multi-pass sorted neighborhood blocking with mapreduce. *Computer Science - R&D*, 27(1):45–63, 2012.

[64] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1):484–493, 2010.

[65] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. ~okeanos: Building a cloud, cluster by cluster. *IEEE Internet Computing*, 17(3):67–71, 2013.

[66] Simon Lacoste-Julien, Konstantina Palla, Alex Davies, Gjergji Kasneci, Thore Graepel, and Zoubin Ghahramani. Sigma: simple greedy matching for aligning large knowledge bases. In *KDD*, 2013.

[67] Christina Lantzaki, Panagiotis Papadakos, Anastasia Analyti, and Yannis Tzitzikas. Radius-aware approximate blank node matching using signatures. *Knowl. Inf. Syst.*, 50(2):505–542, 2017.

[68] Juanzi Li, Jie Tang, Yi Li, and Qiong Luo. Rimom: A dynamic multistrategy ontology alignment framework. *IEEE Trans. Knowl. Data Eng.*, 21(8):1218–1232, 2009.

[69] Thomas Lin, Patrick Pantel, Michael Gamon, Anitha Kannan, and Ariel Fuxman. Active objects: actions for entity-centric search. In *WWW*, pages 589–598, 2012.

[70] Pankaj Malhotra, Puneet Agarwal, and Gautam Shroff. Graph-parallel entity resolution using LSH & IMM. In *EDBT/ICDT Workshops*, 2014.

[71] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *SIGKDD*, pages 169–178, 2000.

[72] W.P. McNeill, Hakan Kardes, and Andrew Borthwick. Dynamic record blocking: efficient linking of massive databases in mapreduce. In *QDB*, 2012.

[73] Ahmed Metwally and Christos Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8):704–715, 2012.

[74] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[75] Iris Miliaraki, Roi Blanco, and Mounia Lalmas. From "selena gomez" to "marlon brando": Understanding explorative entity search. In *WWW*, pages 765–775, 2015.

[76] Felix Naumann and Melanie Herschel. *An Introduction to Duplicate Detection.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[77] Axel-Cyrille Ngonga Ngomo and Sören Auer. LIMES - A time-efficient approach for large-scale link discovery on the web of data. In *IJCAI*, 2011.

[78] George Papadakis, Ekaterini Ioannou, Claudia Niederée, and Peter Fankhauser. Efficient entity resolution for large heterogeneous information spaces. In *WSDM*, pages 535–544, 2011.

[79] George Papadakis, Ekaterini Ioannou, Claudia Niederée, Themis Palpanas, and Wolfgang Nejdl. Eliminating the redundancy in blocking-based entity resolution methods. In *JCDL*, pages 85–94, 2011.

[80] George Papadakis, Ekaterini Ioannou, Claudia Niederée, Themis Palpanas, and Wolfgang Nejdl. Beyond 100 million entities: large-scale blocking-based resolution for heterogeneous data. In *WSDM*, pages 53–62, 2012.

[81] George Papadakis, Ekaterini Ioannou, Themis Palpanas, Claudia Niederée, and Wolfgang Nejdl. A blocking framework for entity resolution in highly heterogeneous information spaces. *IEEE Trans. Knowl. Data Eng.*, 25(12):2665–2682, 2013.

[82] George Papadakis, Georgia Koutrika, Themis Palpanas, and Wolfgang Nejdl. Meta-blocking: Taking entity resolutionto the next level. *IEEE Trans. Knowl. Data Eng.*, 26(8):1946–1960, 2014.

[83] George Papadakis, George Papastefanatos, and Georgia Koutrika. Supervised meta-blocking. *PVLDB*, 7(14):1929–1940, 2014.

[84] George Papadakis, George Papastefanatos, Themis Palpanas, and Manolis Koubarakis. Boosting the efficiency of large-scale entity resolution with enhanced meta-blocking. *Big Data Research*, 6:43–63, 2016.

[85] George Papadakis, George Papastefanatos, Themis Palpanas, and Manolis Koubarakis. Scaling entity resolution to large, heterogeneous data with enhanced meta-blocking. In *EDBT*, pages 221–232, 2016.

[86] Thorsten Papenbrock, Arvid Heise, and Felix Naumann. Progressive duplicate detection. *IEEE Trans. Knowl. Data Eng.*, 27(5):1316–1329, 2015.

[87] Petar Petrovski, Volha Bryl, and Christian Bizer. Integrating product data from websites offering microdata markup. In *WWW*, 2014.

[88] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[89] Vibhor Rastogi, Nilesh N. Dalvi, and Minos N. Garofalakis. Large-scale collective entity matching. *PVLDB*, 4(4):208–218, 2011.

[90] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. Adoption of the linked data best practices in different topical domains. In *ISWC*, pages 245–260, 2014.

[91] Chao Shao, Linmei Hu, Juan-Zi Li, Zhichun Wang, Tong Lee Chung, and Jun-Bo Xia. Rimom-im: A novel iterative framework for instance matching. *J. Comput. Sci. Technol.*, 31(1):185–197, 2016.

[92] Giovanni Simonini, Sonia Bergamaschi, and H. V. Jagadish. BLAST: a loosely schema-aware meta-blocking approach for entity resolution. *PVLDB*, 9(12):1173–1184, 2016.

[93] Dezhao Song and Jeff Heflin. Automatically generating data linkages using a domain-independent candidate selection approach. In *ISWC*, 2011.

[94] Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[95] Kostas Stefanidis, Vassilis Christophides, and Vasilis Efthymiou. Web-scale blocking, iterative and progressive entity resolution. In *ICDE*, pages 1459–1462, 2017.

[96] Kostas Stefanidis, Vasilis Efthymiou, Melanie Herschel, and Vassilis Christophides. Entity resolution in the web of data. In *WWW Companion Volume*, pages 203–204, 2014.

[97] Fabian M. Suchanek, Serge Abiteboul, and Pierre Senellart. PARIS: probabilistic alignment of relations, instances, and schema. *PVLDB*, 5(3):157–168, 2011.

[98] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010.

[99] Julius Volz, Christian Bizer, Martin Gaedke, and Georgi Kobilarov. Silk - A link discovery framework for the web of data. In *LDOW*, 2009.

[100] Steven Euijong Whang, David Marmaros, and Hector Garcia-Molina. Pay-as-you-go entity resolution. *IEEE Trans. Knowl. Data Eng.*, 25(5):1111–1124, 2013.

[101] Steven Euijong Whang, David Menestrina, Georgia Koutrika, Martin Theobald, and Hector Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD*, pages 219–232, 2009.

[102] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[103] Su Yan, Dongwon Lee, Min-Yen Kan, and C. Lee Giles. Adaptive sorted neighborhood methods for efficient record linkage. In *JCDL*, pages 185–194, 2007.

[104] Wei Yan, Yuan Xue, and Bradley Malin. Scalable load balancing for mapreduce-based record linkage. In *IPCCC*, pages 1–10, 2013.

[105] Xiao Yu, Hao Ma, Bo-June Paul Hsu, and Jiawei Han. On building entity recommender systems using user click log and freebase knowledge. In *WSDM*, pages 263–272, 2014.

[106] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[107] Stefan Zwicklbauer, Christin Seifert, and Michael Granitzer. DoSeR - A Knowledge-Base-Agnostic Framework for Entity Disambiguation Using Semantic Embeddings. In *ESWC*, pages 182–198, 2016.

Glossary

block (symb. b) a subset of entity descriptions from an entity collection (typically created using an indexing function). 6, 10, 11, 13–22, 24, 25, 27, 28, 31, 34–36, 39, 41–77, 80, 82, 88, 89, 91, 94, 98, 99

blocking graph (symb. G) a graph whose nodes represent entity descriptions and edges connect descriptions that share a common block, denoting the candidate matches suggested by blocking. 6, 10, 11, 41–44, 47–49, 51, 53–55, 57, 72, 81, 83, 88, 89, 91, 93, 94, 97, 100, 105, 106, 109

dataset a set of entity collections, whose entity descriptions can be matched (usually referring to a benchmark dataset). 7, 9, 21, 28–31, 33–40, 43, 65, 66, 69–72, 75–78, 96, 97, 100, 101, 103

entity collection (symb. $\mathcal{E}$) a set of entity descriptions, either from a single source (e.g., a single KB or database), or from multiple sources. 5, 6, 14–16, 20–22, 24–28, 35, 42, 44, 45, 47, 65, 84, 87, 88, 109

entity description (symb. e) (or simply "description", when clear from the context) an identifiable set of attribute-value pairs, used to describe a real-world entity, including physical (e.g., a book) and non-physical (e.g., a character in a book) objects. 1–4, 6–10, 13–15, 18–20, 25, 27, 28, 30, 31, 34, 39, 40, 44, 48, 49, 65, 66, 77, 79, 81–87, 93, 100, 105, 107, 108

ER (Entity Resolution) the problem of identifying entity descriptions that refer to the same real-world entity. 1, 4, 5, 7–11, 13, 17, 25, 30, 39, 40, 49, 66, 76, 79, 81, 82, 84, 97, 99, 101, 105, 106, 108–110

ground truth a set entity description pairs which are known to refer to the same real-world entity (i.e., a given set of matches), used for the evaluation of an ER task. 14, 16, 30, 31, 35, 36, 65, 96, 97, 99

KB (Knowledge Base) a single data source, containing descriptions of real-world entities (the main difference to a database, is that a KB does not necessarily follow a specific schema). 1–5, 7–11, 13, 14, 20, 22, 29–31, 34–36, 39, 40, 65, 79–82, 86, 88, 89, 91, 96, 97, 100, 103, 105–108

match (noun) a pair of entity descriptions that refer to the same real-world entity; (verb) refer to the same real-world entity. 1, 5–10, 13–21, 23–25, 30, 31, 33, 35–39, 41, 48, 65, 75, 76, 79, 81–84, 88, 91–97, 99, 100, 106, 108, 109