DEGREE OF DOCTOR OF PHILOSOPHY IN
COMPUTER ENGINEERING AND SCIENCE

DOCTORATE SCHOOL IN
INFORMATION AND COMMUNICATION TECHNOLOGIES

XXXII CYCLE

UNIVERSITY OF MODENA AND REGGIO EMILIA

"ENZO FERRARI" ENGINEERING DEPARTMENT

Ph.D. DISSERTATION

Techniques for Big Data Integration in Distributed Computing Environments

Candidate:
Luca GAGLIARDELLI
Advisor:
Prof. Sonia BERGAMASCHI
Co-Advisor:
PhD Giovanni SIMONINI
Director of the School:
Prof. Sonia BERGAMASCHI

DOTTORATO DI RICERCA IN
COMPUTER ENGINEERING AND SCIENCE

SCUOLA DI DOTTORATO IN INFORMATION AND
COMMUNICATION TECHNOLOGIES

XXXII CICLO

UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

DIPARTIMENTO DI INGEGNERIA "ENZO FERRARI"

TESI PER IL CONSEGUIMENTO DEL TITOLO DI DOTTORE DI RICERCA

Tecniche per l'Integrazione di Sorgenti Big Data in Ambienti di Calcolo Distribuito

Candidato:
Luca GAGLIARDELLI
Relatore:
Prof. Sonia BERGAMASCHI
Correlatore:
PhD Giovanni SIMONINI
Il Direttore della Scuola:
Prof. Sonia BERGAMASCHI

Keywords:

Data Integration
Big Data
Scalable Entity Resolution
Scalable Similarity Join

Abstract

Data sources that provide a huge amount of semi-structured data are available on Web as tables, annotated contents and Linked Open Data (e.g., RDF). These sources can constitute a valuable source of information for companies, researchers and government agencies, if properly manipulated and integrated with each other or with proprietary data. One of the main problems is that typically these sources are heterogeneous and do not come with explicit keys to perform join operations (i.e., *equi-join*) for effortlessly linking their records. Thus, finding a way to join data sources without keys is a fundamental and critical process of data integration. Moreover, for many applications, the execution time is a critical component (e.g., in finance of national security context) and distributed computing can be employed to significantly improve it. In this dissertation, I present distributed data integration techniques that allow to scale to large volumes of data (i.e., Big Data), in particular: SparkER and GraphJoin. SparkER is an Entity Resolution tool that aims to exploit distributed computing to identify records in data sources that refer to the same real-world entity—thus enabling the integration of the records. This tool introduces a novel algorithm to parallelize the indexing techniques that are currently state-of-the-art. SparkER is a working software prototype that I developed and employed to perform experiments over real data sets; the results show that the parallelization techniques that I have developed are more efficient in terms of execution time and memory usage than those in literature. GraphJoin is a novel technique that allows to find similar records (i.e. to perform similarity join) by applying joining rules on one or more attributes. This technique combines similarity join techniques designed to work on a single rule, optimizing their execution with multiple joining rules, combining different similarity measures both token- and character- based (e.g., Jaccard Similarity and Edit Distance). For GraphJoin I developed a working software prototype and I employed it to experimentally demonstrate that the proposed technique is effective and outperforms the existing similarity join techniques in terms of

execution time.

Sommario

Sorgenti che forniscono grandi quantitativi di dati semi-strutturati sono disponibili sul Web in forma di tabelle, contenuti annotati (e.s. RDF) e Linked Open Data. Questi dati se debitamente manipolati e integrati tra loro o con dati proprietari, possono costituire una preziosa fonte di informazione per aziende, ricercatori e agenzie governative. Il problema principale in fase di integrazione è dato dal fatto che queste sorgenti dati sono tipicamente eterogenee e non presentano chiavi su cui poter eseguire operazioni di join per unire facilmente i record. Trovare un modo per effettuare il join senza avere le chiavi è un processo fondamentale e critico dell'integrazione dei dati. Inoltre, per molte applicazioni, il tempo di esecuzione è una componente fondamentale (e.s. nel contesto della sicurezza nazionale) e il calcolo distribuito può essere utilizzato per ridurlo sensibilmente. In questa dissertazione presento delle tecniche distribuite per l'integrazione dati che consentono di scalare su grandi volumi di dati (Big Data), in particolare: SparkER e GraphJoin. SparkER è un tool per Entity Resolution che mira ad utilizzare il calcolo distribuito per identificare record che si riferiscono alla stessa entità del mondo reale, consentendo così l'integrazione di questi record. Questo tool introduce un nuovo algoritmo per parallelizzare le tecniche di indicizzazione che sono attualmente lo stato dell'arte. SparkER è un prototipo software funzionante che ho sviluppato e utilizzato per eseguire degli esperimenti su dati reali; i risultati ottenuti mostrano che le tecniche di parallelizzazione che ho sviluppato sono più efficienti in termini di tempo di esecuzione e utilizzo di memoria rispetto a quelle già esistenti in letteratura. GraphJoin è una nuova tecnica che consente di trovare record simili applicando delle regole di join su uno o più attributi. Questa tecnica combina tecniche di join similarity pensate per lavorare su una singola regola, ottimizzandone l'esecuzione con più regole, combinando diverse misure di similarità basate sia su token che su caratteri (e.s. Jaccard Similarity e Edit Distance). Per il GraphJoin ho sviluppato un prototipo software funzionante e l'ho utilizzato per eseguire esperimenti che dimostrano che la tecnica proposta è efficace ed è più efficiente di quelle già

esistenti in termini di tempo di esecuzione.

Acknowledgments

Acknowledgments
I owe my deepest gratitude to both my advisor Professor Sonia Bergamaschi, and co-advisor Giovanni Simonini, for their precious support and guide during my Ph.D.

Ringraziamenti
Voglio ringraziare tutti quelli che mi hanno sostenuto durante questi tre anni di dottorato. Ringrazio i miei genitori che mi hanno permesso di raggiungere i miei obiettivi. Ringrazio Giovanni che mi ha aiutato moltissimo nel mio percorso di dottorato. I miei amici e compagni di laboratorio, in particolare Matteo e Paolo. La mia ragazza, Giulia che nell'ultimo anno mi ha sostenuto e spronato.

Contents

1 Introduction **19**
- 1.1 Data Integration . . . 19
- 1.2 Big Data Integration challenges . . . 20
- 1.3 Blocking and meta-blocking . . . 20
- 1.4 Record-level matching rules . . . 22
- 1.5 Structure of the Thesis . . . 24

2 Preliminaries **25**
- 2.1 General concepts . . . 26
- 2.2 Blocking for Entity Resolution . . . 26
  - 2.2.1 Blocking . . . 26
  - 2.2.2 Meta-Blocking . . . 27
  - 2.2.3 Blast . . . 29
- 2.3 Record-level Matching Rules . . . 32
  - 2.3.1 Set Similarity Join . . . 33
- 2.4 MapReduce-like Systems . . . 35

3 Distributed meta-blocking **39**
- 3.1 Distributed Loose Schema Information Extraction . . . 40
- 3.2 Distributed Blocks Generation . . . 40
- 3.3 Distributed Blocking-graph Processing . . . 42
- 3.4 SparkER . . . 46
  - 3.4.1 Blocker . . . 46
  - 3.4.2 Entity Matcher and Clusterer . . . 47
  - 3.4.3 Process debugging . . . 48

4 Record Level Matching Rules **51**
- 4.1 Baseline algorithm . . . 52
- 4.2 GraphJoin . . . 53
- 4.3 RulER . . . 56

5 Experimental Evaluation **59**
5.1 Distributed Blast evaluation 60
5.1.1 Blast vs. Schema-agnostic Meta-blocking 62
5.1.2 Broadcast vs. Repartition Meta-blocking 66
5.1.3 Parallel-blast scalability 67
5.2 GraphJoin evaluation 70
5.2.1 Experimental Setup 70
5.2.2 *GraphJoin* vs *JoinChain* 71
5.2.3 *GraphJoin* scalability 72

6 Related Work **75**

7 Conclusions and Future Work **79**

Bibliography **81**

List of Figures

2.1 Example of *meta-blocking* processing 27
2.2 *Meta-blocking* weight threshold. 29
2.3 *Blast* logical overview 30
2.4 Example of *Blast* processing 30
2.5 Example of prefix filtering 34
2.6 Positional filter example 35
2.7 Prefix filter based Similarity Join process 36

3.1 Repartition meta-blocking example 43
3.2 Broadcast meta-blocking example 45
3.3 *SparkER* architecture. 46
3.4 Blocker module 47
3.5 Entity clusterer. 48
3.6 SparkER result analysis 49

4.1 *GraphJoin* execution model 56
4.2 *RulER* example usage 58

5.1 Meta-blocking execution time 64
5.2 *Blast* vs schema-agnostic meta-blocking recall and precision . 65
5.3 *Repartition* vs. *Broadcast meta-blocking*. 66
5.4 Scalability comparison: *repartition* vs. *broadcast meta-blocking*. 67
5.5 Speedup of *Blast*. 68
5.6 Execution time of *Blast*. 69
5.7 Execution time of the complete ER process. 70
5.8 Execution times of GraphJoin and JoinChain. 72
5.9 Number of comparisons performed by GraphJoin and JoinChain 72
5.10 GraphJoin execution time and speedup. 73

List of Tables

5.1 Dataset characteristics 61
5.2 Acronyms and configurations. 62
5.3 Metrics. 63
5.4 Matching rules employed in the experiments. 71

Chapter 1

Introduction

1.1 Data Integration

Data Integration is the problem of combining data residing at different autonomous sources, and providing the user with a unified view of these data [DHI12; Ber+11]. Our research group, DBGroup[1], has been investigating data integration for more than 20 years and almost all of the research activities have been centered around the MOMIS (Mediator EnvirOnment for Multiple Information Sources) Data Integration System [BCV99; BB04; Ber+11], that follows a traditional approach to Data Integration. An open-source version of the MOMIS system is delivered and maintained by Datariver [2].

Given multiple data sources, the traditional approach to Data Integration is to create a mapping from the sources to a mediated schema (i.e. global schema). The global schema can be queried to obtain an integrated result from the data sources. In this approach, data reside at data sources that are queried in real-time when the global schema is queried, for this reason, it is also called *virtual data integration* [Ber+11]. The advantages of a virtual approach are related to the management of the changes in the sources. Since in database applications schemas do not frequently change, virtual approaches do not require strong updating policies as data are retrieved at run time. To produce a global schema is a hard task because it has to deal with multiple heterogeneous data sources that have different schemata and could have multiple representations of the same data. Moreover, modeling the mapping of the schemata of each source to the mediated schema is a crucial and non-trivial task, especially when dealing with highly heterogeneous data sources. This implies the development of techniques for many difficult tasks: *data*

[1] `http://dbgroup.unimore.it`

[2] `http://www.datariver.it`

cleaning, *reconciliation*, and *fusion* [BN09].

1.2 Big Data Integration challenges

The Web has become a valuable source of information, a huge amount of structured and semi-structured data are available as tables, annotated contents and Linked Open Data. The true potential of these data is expressed when multiple data sources are integrated, to extract and generate new knowledge. Moreover, for companies, researchers and government agencies these data become more valuable if integrated with proprietary data that they already own, and that are typically integrated with standard data integration approaches. In this context, data integration is a non-trivial task, due to the high heterogeneity, volume, and noise of the involved data [DS15; Ber14]. Furthermore, typically the data sources to integrate do not come with keys to perform join operations (i.e., *equi-join*) to effortlessly linking their records. Thus, finding a way to join data without keys is a fundamental and critical process of data integration.

This data integration step is known as *Entity Resolution* (ER), namely the task of matching records from different data sources that refer to the same real-world entity [Chr12b]. Due to the computational complexity ER is a hard and time-consuming task (all records have to be compared with all each other), especially when dealing with *Big Data* sources that can involve billions of comparisons. Moreover, for many applications is critical to retrieve the results in a short time even when working with *Big Data* [DS15]. Thus, it is necessary to use the distributed computing to perform ER on *Big Data* sources with a reasonable execution time [Eft+17].

In this thesis, I present distributed data integration techniques that allow scaling to large volumes of data. In particular, I present two techniques that aim to cover two different scenarios: (i) in the first case the ER process is performed by automatically extract the join keys from the data, without the user intervention; (ii) in the second case the user can express record-level matching rules to join the records that satisfy them. These approaches are introduced in the following sections 1.3 and 1.4 respectively.

1.3 Blocking and meta-blocking

Entity Resolution (ER) is the task to identify if different entity profiles pertain to the same real-world object. Comparing all possible pairs of profiles of an entity collection is a quadratic problem: if the profiles number grows

linearly, the comparisons number grows quadratically. Hence, a naïve approach (i.e., compare all possible pairs of profiles) becomes infeasible for large datasets. To solve this problem, typically, signatures (*blocking keys*) are extracted from the profiles and employed to group similar profiles into *blocks*. Then, only the profiles that appear together in the same block are compared. Traditionally blocking techniques typically rely on a-priori schema knowledge to generate good blocking keys by combining attribute values. Anyway, this approach is not applicable in the Big Data context, due to high heterogeneity, high level of noise and very large volume. For this reason, *schema-agnostic* blocking approaches have been proposed. These approaches rely on redundancy to find the duplicates: each profile is placed in multiple blocks, reducing the probability of missing matches. For example, in Token Blocking each token that appears in the values of the profile is a blocking key. So, all the profiles that share at least one token are placed at least once in the same block. The drawback of schema-agnostic is the degradation of efficiency. In fact, it ensures a high recall (i.e., a high percentage of detected duplicates) but with very low precision, i.e., a high number of superfluous comparisons are performed. To solve this problem *meta-blocking* was introduced [Pap+14; Pap+16].

Meta-blocking is the task of restructuring a set of block to retain only the most promising comparisons. A block collection is represented as a weighted graph, called *blocking graph*, where each entity profile is a node, and two nodes are connected by an edge if the corresponding profiles appear together in at least one block. The edges are weighted to capture the likelihood of a match. After that, a pruning algorithm is applied to the blocking graph to remove the less promising edges. The most effective and accurate strategy to prune the graph is to consider for each node its adjacent edges, and retain only those having a weight higher than the local average [Pap+16]. At the end of the process, each pair of nodes connected by an edge form a new block. Traditional meta-blocking approaches rely only on the schema-agnostic features, Simonini et al. [SBJ16] proposed a novel technique called Blast (Blocking with Loosely-Aware Schema Techniques) that can easily collect significant statistics from the data that approximately describe the data sources schema, performing a loosely-aware schema (meta-)blocking that significantly improves the blocking quality (i.e., the precision) without loosing in recall. In fact, Blast can be considered state-of-the-art for unsupervised (meta-)blocking.

My contribution: In literature, only a few works proposed distributed approaches to scale meta-blocking on very large datasets using MapReduce-like systems [Eft+17]. The existing approaches fully materialize the blocking graph, requiring a high quantity of memory. In this thesis, I propose a

novel approach that do not require to materialize the full blocking graph, while improving the performance in term of execution time and memory requirements. Moreover, no approaches to parallelize Blast were proposed in the literature. In this thesis, I introduce SparkER, a distributed (meta-)blocking tool that supports both schema-agnostic and loosely-schema aware (i.e., Blast) approaches. The proposed approach was described for the first time in [Sim+19].

Here the list of contributions introduced with SparkER:

- an algorithm to efficiently run Blast (and any other graph-based meta-blocking method) on MapReduce-like systems, to take full advantage of a parallel and distributed computation;

- an evaluation of my approach on real-world datasets, comparing it against the meta-blocking state of the art approaches.

1.4 Record-level matching rules

With the increase of the amount of data and the need to integrate multiple data sources, a challenging issue is to find duplicate records efficiently. Different techniques were proposed in the literature to accomplish this task, and one of the most promising is the set similarity join. Different similarity join implementations [MAB16] were proposed, but all of them rely on the idea that there is a single predicate to apply on the records: each record is treated as a single set of elements, and the goal is to find the pairs of sets which have a similarity (i.e., Jaccard Similarity, Edit Distance, etc.) greater than a certain threshold. If the data are (semi-)structured and the user wants to apply multiple predicates, she has to run the similarity join multiple times (one for every predicate) and then intersect the results obtained from each run. Moreover, if the user wants to apply different similarity measures like Edit Distance (ED) and Jaccard Similarity (JS), she has to use different implementations, since they rely on different indexing techniques to perform the similarity join [XWL08; Xia+11].

My contribution: I proposed a novel approach called *GraphJoin* that can efficiently apply multiple matching rules based on different similarity measures in a single step, without any further effort from the user. The main motivation behind this approach is that no one of the proposed similarity joins implementations allows to apply multiple predicates with different similarity measures simply and efficiently, a high human effort is always required, and moreover, the process is inefficient.

For example, suppose to take the IMDB dataset [Das+] that contains records about movies, this dataset has different attributes (e.g., title, actors, plot, etc.), and suppose that we want to apply the following record-level matching rule (i.e., multiple predicates): $JS(plot) \geq 0.9 \wedge ED(title) \leq 2$. Since the plot of a movie is very long and contains a lot of tokens, we want to use the JS to measure the similarity of two plots, while the title is usually a short string, so we want to use the ED as similarity measure. A naïve solution is to accomplish this task using the state-of-the-art algorithms, splitting the work in four main steps: (i) use an algorithm (e.g., PPJoin [Xia+11]) on the plot to collect a set of candidate pairs that can satisfy the first condition; (ii) apply a different algorithm (e.g., EDJoin [XWL08]) on the title to collect a set of candidate pairs that can satisfy the second condition; (iii) intersect the two sets of candidate pairs to obtain the pairs that can satisfy both conditions; finally (iv) verify that the retained candidate pairs effectively satisfies the conditions. The main problem of this solution is that many superfluous comparisons are computed, each predicate is processed independently, without considering the other ones.

GraphJoin considers all the matching predicates together, so when a pair of record is evaluated it is sufficient that a predicate is not respected to discard that pair, avoiding to perform unnecessary comparisons. In practice, if a pair do not respect a predicate, the others are not evaluated, making the process faster. Another advantage is to have a unique framework that let to apply different similarity measures that use different indexing techniques to discover similar records, so it is not necessary to use multiple implementations to accomplish the task. Moreover, the use of a unified system gives room for further improvements, for example it is possible to sort the different predicates based on their cost, so when a pair of record is evaluated the system can start from the cheaper predicate, and then continue with the others. In this way, if a pair do not satisfy the first predicate (i.e., the cheaper to compute) the others are not computed, making the process more efficient.

Here the list of contributions introduced with *GrapJoin*:

- a technique to apply record-level matching rules efficiently, that can use different similarity measures;

- an algorithm to efficiently run record-level matching rules on MapReduce-like systems, to take full advantage of a parallel and distributed computation.

1.5 Structure of the Thesis

This thesis is structured as follows: chapter 2 gives the preliminary concepts and definitions employed throughout the thesis; chapter 3 presents the novel technique to parallelize *Blast* and schema-agnostic meta-blocking for taking full advantage out of parallel and distributed computation; chapter 4 presents the novel technique to efficiently scale record-level matching rules over big datasets; Both chapters include the tools implementing the different proposed techniques. in chapter 5 the experimental outcomes are presented and discussed; chapter 6 reviews the related works; and finally, in chapter 7, I draw the conclusions and present ongoing and future works.

Chapter 2

Preliminaries

This section describes the fundamental concepts and notation employed in this thesis.

2.1 General concepts

An entity *profile* is a tuple composed of a unique identifier and a set of *name-value* pairs $\langle a, v \rangle$. $A_{\mathcal{P}}$ is the set of possible attributes a associated to a profile collection $\mathcal{P}$. A *profile collection* $\mathcal{P}$ is a set of profiles. Two profiles $p_i, p_j \in \mathcal{P}$ are *matching* ($p_i \approx p_j$) if they refer to the same real world object; *Entity Resolution* (ER) is the task of identifying those matches given $\mathcal{P}$. The näive solution to ER implies $|\mathcal{P}_1| \cdot |\mathcal{P}_2|$ comparisons, where $|\mathcal{P}_i|$ is the cardinality of a profile collection $\mathcal{P}_i$.

Dirty ER and *Clean-Clean ER*

Papadakis et al. [Pap+16] have formalized two types of ER tasks: *Dirty ER* and *Clean-Clean ER*. The former refers to those scenarios where ER is applied to a single data source containing duplicates; this problem is also known in the literature as *deduplication* [Chr12b]. In the latter, ER is applied to two or more data sources, which are considered "clean", i.e., each source considered singularly does not contain duplicate. This type of ER is also known as *Record Linkage* [Chr12b]. As in [SBJ16; Pap+16; Eft+17; CES15; Sim+18b], in this thesis the same classification is adopted as well. Notice that, in *Clean-Clean ER* the comparisons among profiles that belong to the same data source are avoided [Pap+16].

2.2 Blocking for Entity Resolution

2.2.1 Blocking

Blocking approaches aim to reduce the ER complexity by indexing similar profiles into *blocks* according to a *blocking key* (i.e., the indexing criterion), restricting the actual comparisons of profiles to those appearing in the same block.

Given the dataset of Figure 2.1(a), an example of *schema-agnostic* blocking key is shown in Figure 2.1(b). Otherwise, a *schema-based* blocking key might be the value of the attribute "name"; meaning that only profiles that have the same value for "name" will be compared (the dataset in Figure 2.1(a) would require a schema-alignment). A set of blocks $\mathcal{B}$ is called *block col-*

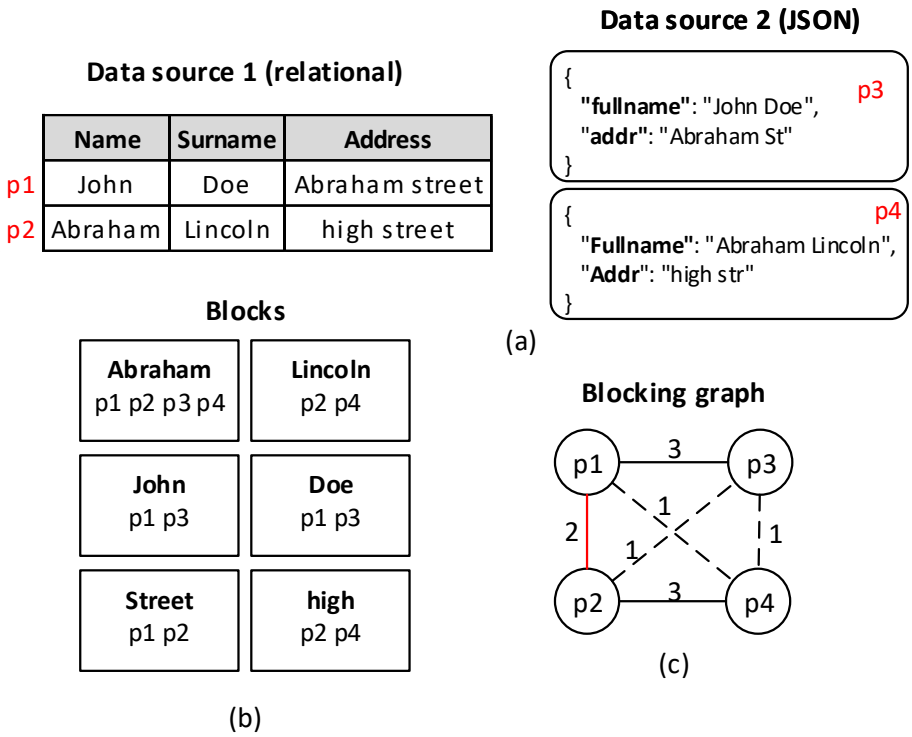

Figure 2.1: (a) a collection of entity *profiles* coming from different data sources. (b) A block collection produced with Token Blocking. (c) The derived blocking graph and the effects of *meta-blocking*: dashed lines represent pruned edges, and red ones the superfluous comparisons not removed.

lection, and its *aggregate cardinality* is $\|\mathcal{B}\| = \sum_{b_i \in \mathcal{B}} \|b_i\|$, where $\|b_i\|$ is the number of comparisons implied by the block b_i.

2.2.2 Meta-Blocking

The goal of *meta-blocking*[Pap+16] is to restructure a collection of blocks, generated by a redundant blocking technique, relying on the intuition that the more blocks two profiles share, the more likely they match.

Definition 1 META-BLOCKING. *Given a block collection $\mathcal{B}$, meta-blocking is the task of restructuring the set of blocks, producing a new block collec-*

tion $\mathcal{B}'$ with significantly higher precision and nearly identical recall, i.e.,: precision($\mathcal{B}'$)$\gg$precision($\mathcal{B}$) and recall($\mathcal{B}'$)$\simeq$recall($\mathcal{B}$).

In *graph-based meta-blocking* (or simply *meta-blocking* from now on), a block collection $\mathcal{B}$ is represented by a weighted graph $\mathcal{G}_\mathcal{B}\{V_\mathcal{B}, E_\mathcal{B}, \mathcal{W}_\mathcal{B}\}$ called ***blocking graph***. V is the set of nodes representing all $p_i \in \mathcal{P}$. An edge between two entity profiles exists if they appear in at least one block together: $E = \{e_{ij} : \exists p_i, p_j \in \mathcal{P} \mid |\mathcal{B}_{ij}| > 0\}$ is the set of edges; $\mathcal{B}_{ij} = \mathcal{B}_i \cap \mathcal{B}_j$, where $\mathcal{B}_i$ and $\mathcal{B}_j$ are the set of blocks containing p_i and p_j respectively. $\mathcal{W}_\mathcal{B}$ is the set of weights associated to the edges. Meta-blocking methods weight the edges to capture the *matching likelihood* of the profiles that they connect. For instance, ***block co-occurrence frequency*** (a.k.a. CBS) [Pap+14; RVC18] assigns to the edge between two profiles p_u and p_v a weight equal to the number of blocks they share, i.e.: $w_{uv}^{CBS} = |\mathcal{B}_u| \cap |\mathcal{B}_v|$. Then, edge-pruning strategies are applied to retain only more promising ones. Thus, at the end of the pruning, each pair of nodes connected by an edge forms a new block of the final, restructured blocking collection. Note that meta-blocking inherently prevents redundant comparisons since two nodes (profiles) are connected at most by one edge.

Two classes of pruning criteria can be employed in meta-blocking: ***cardinality-based***, which aims to retain the *top-k* edges, allowing an a-priori determination of the number of comparisons (the *aggregate cardinality*) and, therefore, of the execution time, at the expense of the recall; and ***weight-based***, which aims to retain the "most promising" edges through a weight threshold. The scope of both pruning criteria can be either ***node-centric*** or ***global***: in the first case, for each node p_i the *top-k_i* adjacent edges (or the edges below a local threshold θ_i) are retained; in the second case, the *top-K* edges (or the edges below a global threshold Θ) are selected among the whole set of edges. The combination of those characteristics leads to four possible *pruning schemas*: *(i) Weight Edge Pruning* (*WEP*) discards all the edges with weight lower than Θ; *(ii) Cardinality Edge Pruning* (*CEP*) sorts all the edges by their weights in descending order, and retains only the first K; *(iii) Weight Node Pruning* (WNP [Pap+16]) considers in turn each node p_i and its adjacent edges, and prunes those edges that are lower than a local threshold θ_i; *(iv) Cardinality Node Pruning* (CNP [Pap+16]) similarly to WNP is node centric, but instead of a weight threshold it employs a cardinality threshold k_i (i.e., retain the top-k_i edges for each node p_i).

In node centric pruning, each edge e_{ij} between two nodes p_i and p_j is related to two thresholds: θ_i and θj (Figure 2.2(a)); where θ_i and θ_j are the threshold associated to p_i and p_j, respectively. Hence, as depicted in Figure 2.2(b), each edge e_{ij} has a weight that can be: (i) lower than both θ_i and

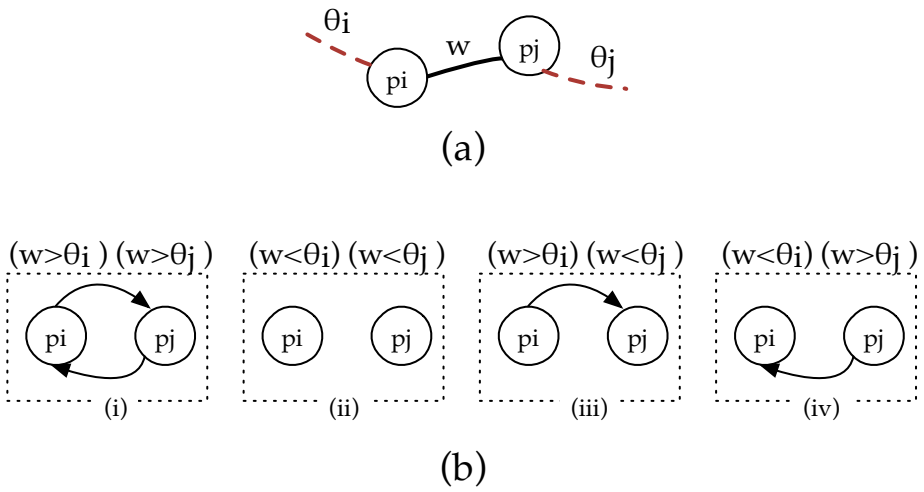

Figure 2.2: Weight threshold. A directed edge from p_i to p_j indicates that the weight of the edge e_{ij} is higher than θ_i; a directed edge from p_j to p_i indicates that the weight of the edge e_{ij} is higher than θ_j.

θ_j, (ii) higher than both θ_i and θ_j, (iii) lower than θ_i and higher than θ_j, or (iv) higher than θ_i and lower than θ_j. Cases (i) and (ii) are not ambiguous, therefore e_{ij} is discarded in the first case, and retained in the second one. But, cases (iii) and (iv) are ambiguous.

Existing *meta-blocking* papers [Pap+16] propose two different approaches to solve this ambiguity: *redefined* WNP/CNP retains e_{ij} if its weight is higher than at least one of the two thresholds for WNP, or if it is in the top-K edges of at least one of the two profiles for CNP (i.e., a logical disjunction, so we call this method $\text{WNP}_{\text{OR}}/\text{CNP}_{\text{OR}}$), while *reciprocal* WNP/CNP retains the edge if its weight is greater than both the threshold for WNP, or if it is in the top-K edges for both profiles for CNP (i.e., logical conjunction, so we call this method $\text{WNP}_{\text{AND}}/\text{CNP}_{\text{AND}}$).

2.2.3 Blast

Blast [SBJ16] is an efficient method to automatically extract loose schema information, which is then used for both blocking and *meta-blocking*. This holistic combination significantly help in producing high-quality candidate pars for ER, compared to other existing *meta-blocking* techniques, which operates only in completely schema-agnostic setting [Pap+14; Pap+16; Eft+17]

The overall workflow of *Blast* is depicted in Figure 2.3: the loose schema information is extracted from the data sources (phase 1) and then it is ex-

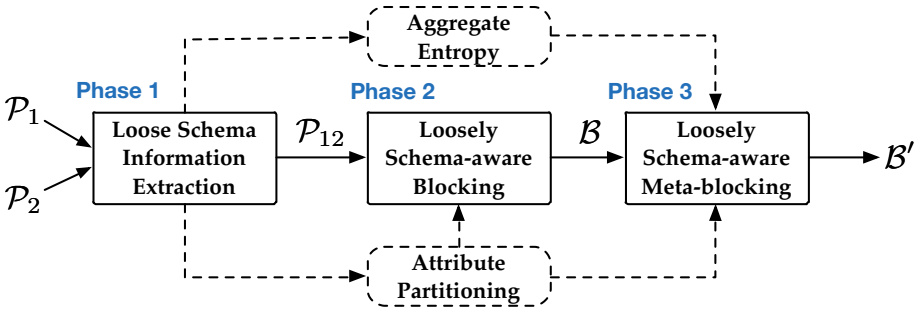

Figure 2.3: *Blast* logical overview.

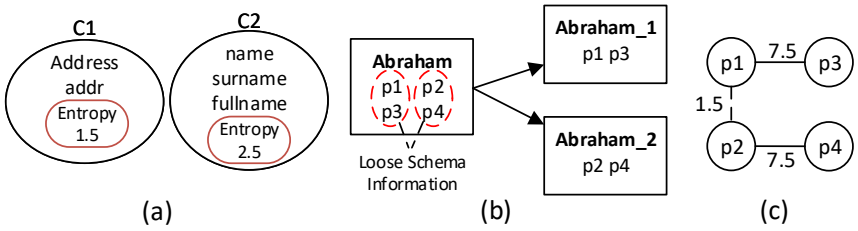

Figure 2.4: (a) Loose schema information extracted from the data: the attributes are clustered together according to the similarity of their values, and for each cluster of attributes the entropy of the values is computed. (b) The blocking key "Abraham" is disambiguated by employing the loose schema information. (c) The effects of the disambiguation on the blocking graph: fewer edges are generated. Moreover, the profiles $p1$ and $p2$ share one less block than before, let to correctly prune the edge e_{1-2}. The edges are weighted according to the entropy of the cluster to which the corresponding attribute belongs. E.g., by using the CBS in combination with the entropy, the edge e_{1-1} is generated by the value *street* of the attribute *address* contained in the *C2* cluster. So its weight is 1.5 due to the cluster entropy.

ploited for building (phase 2) and restructuring (phase 3) a blocking collection. In the following, these three phases are described in more details.

1. The loose schema information is extracted, it is composed of: the *attributes partitioning* and the *aggregate-entropy* (Figure 2.4(a))). The first consists of clusters of attributes (clustered according to their values' similarity); the second measures how *informative* is each of these clusters; together these pieces of information are called: *loose schema information*. For extracting the attributes partitioning, *Blast* exploits an LSH-based [LRU14] algorithm: minhash signatures are employed

to quickly approximate the similarities of the attributes and to build a similarity graph of attributes, which is given as input to a graph-partitioning algorithm (the details of the algorithm can be found in [SBJ16]). *Blast* also extracts the (Shannon) entropy of each cluster of attributes (which is the average of the entropies of its attributes). Basically, the entropy is a measure of how informative is a set of attributes: intuitively, if all the values in the partition are the same, the entropy is equal to zero, and it should not be used indeed for generating blocking keys since all the profiles would be indexed in just one unique block. On the other hand, if the entropy is high, the profiles share values from that attribute partition that are less frequent, hence the derived blocking keys will be more likely useful.

2. A schema-agnostic blocking technique is applied on each attribute partition obtained in the previous phase (Figure 2.4(b)). The resulting method is called Loose-Schema Blocking.

3. This blocking strategy described in the previous phase allows to achieve a high level of recall, but it tends to generate a lot of superfluous comparisons. For this reason, a *meta-blocking* step is performed to generate the final candidate pairs. In particular, *Blast* leverages on the *entropy* extracted in Phase 1 to weight all the candidate pairs generated in Phase 2. The basic idea is to build a graph (the *Blocking Graph*), where each edge corresponds to a set of blocking keys, and each blocking key is associated to an attribute. Then, the edges are weighted accordingly to their entropy (Figure 2.4(c)). For example, consider two independent data sets with people's information. Generally, the attribute *birth date* is less informative than the attribute *surname*. This is because the number of distinct birth dates is typically lower than that of the surnames—and the entropy of the former is lower than the entropy of the latter. Thus, *Blast* assigns a higher weight to edges that represent blocking keys derived from the surnames than those derived from the birth dates. In particular, *Blast* weights the edges computing the (Pearson) chi-square coefficient—the idea is to measure the significance of the co-occurrence of two profiles in a block—multiplied by the aggregate entropy associated to the blocking keys corresponding to that edge. Finally, *Blast* applies a local pruning by computing a threshold for each node in the graph (equal to half of the maximum weight of the adjacent edges[1]).

[1]This is a heuristic that has been shown to work well in practice [SBJ16]

Metrics

Recall and *Precision* are employed to evaluate the quality of a block collection $\mathcal{B}$, as in [Chr12a]. The recall measures the portion of duplicate profiles that are placed in at least one block; while the precision measures the portion of useful comparisons, i.e., those that detect a match. Formally, precision and recall of a blocking method is determined from the block collection $\mathcal{B}$ that it generates:

$$recall = \frac{|\mathcal{D}^{\mathcal{B}}|}{|\mathcal{D}^{\mathcal{P}}|}; \qquad precision = \frac{|\mathcal{D}^{\mathcal{B}}|}{\|\mathcal{B}\|};$$

where $\mathcal{D}^{\mathcal{B}}$ is the set of duplicates appearing in $\mathcal{B}$ and $\mathcal{D}^{\mathcal{P}}$ is the set of all duplicates in the collection $\mathcal{P}$.

2.3 Record-level Matching Rules

Combining data sets that bare information about the same real-world objects is an everyday task for practitioners that work with structured and semi-structured data. Frequently (e.g., when dealing with data lakes or when integrating open data with proprietary data) data sets do not have explicit keys that can be used for a traditional *equi-join*. When this happens, a common solution is to perform a *similarity join* [MAB16], i.e., to join records that have an attribute value similar above a certain threshold, according to a given similarity measure, as in the following example:

Example 2.3.1 (Similarity Join) *Given two product data sets, join all the record pairs with the Jaccard similarity of the product names above 0.8.*

A plethora of algorithms have been proposed in the last decades to efficiently execute the similarity join considering a single attribute, i.e., *attribute-level matching rules* (see [MAB16] for a survey). At their core, all these algorithms try to prune the candidate pairs of records, on the basis of a single-attribute predicate—to alleviate the quadratic complexity of the problem.

Interestingly, only a few works had been focused on studying how to execute *record-level matching rules*, i.e., the combination of multiple similarity join predicates on multiple attributes (see section 2.3). Yet, this kind of rules allows to specify more flexible rules to match records, as in the following example:

Example 2.3.2 (Record-level matching rule) *Given two product data sets, join all the record pairs that have a Jaccard similarity of the product names above 0.8, or that have a Jaccard similarity of the description that is above 0.6 and the edit distance of the manufacturer is lower than 3.*

Furthermore, record-level matching rules can be used to represent *decision trees* [ADA+18], hence learned with machine learning algorithms when training data is available. As a matter of fact, a decision tree for binary classification (i.e., classification of *matching/not-matching* records) can be naturally represented with DNF (disjunctive normal form) predicates—the same consideration can be done for a forest of trees.

We define a matching rule $\mathcal{R}$ as a *disjunction* (logical *OR*) of *conjunctions* (logical *AND*) of similarity join predicates on multiple attribute (i.e., at the *record level*). This design choice is driven by the fact that DNF matching rules are easy to read and thus to debug, in practice. Moreover, DNFs can be employed to represent the trained model of a decision tree (or of a random forest), hence suitable for exploiting labeled data. In this thesis, we focus on how to scale DNF matching rules and we do not investigate how to generate *good* DNFs (i.e., decision trees/random forests) starting from training data.

2.3.1 Set Similarity Join

In this scenario, a profile p_i is considered as a set of elements identified by a unique identifier. Different techniques can be employed to generate the elements from the values of a profile, for example, each word can be considered as a token or it is possible to generate the n-grams, etc. Formally, given a collection of profiles, a similarity function sim and a similarity threshold t, the goal of set similarity join is to find all the possible pairs of profiles $\langle p_i, p_j \rangle$ such that $sim(p_i, p_j) \geq t$.

A naïve solution to perform the set similarity join is to enumerate and compare every pair of records, but this process is highly inefficient and not feasible in the Big Data context. To reduce the task complexity different approaches were proposed in literature [CGK06; BMS07; Xia+11; XWL08]. All these approaches adopt a filter-verification approach: (1) first an index is used to obtain a set of pre-candidates; (2) the pre-candidates are filtered using a set of pre-defined filters; (3) the resulting candidate pairs are probed with the similarity function to generate the final results.

The most used filters are: prefix filter, length filter, and positional filter. All these filters can be adapted to work with different similarity measures: Dice, Cosine, Jaccard Similarity, Edit Distance and Overlap Similarity [MAB16; XWL08; Xia+11].

Prefix filter

A key technique to perform the set similarity join efficiently is the *prefix filter* [CGK06]. First of all, given a collection of profiles (i.e., sets of elements)

their elements are sorted according to a global order $\mathcal{O}$, usually the document frequency of the tokens (i.e., how many documents contain that token) that is a heuristic that helps to reduce the number of comparisons [CGK06]. Then, for each sorted set, only the first π elements are considered, i.e., the *prefixes*. A pair $\langle p_i, p_j \rangle$ can be safely pruned if their prefixes have no common elements. The prefix size depends on the similarity threshold and the similarity function. For example, the prefix filter for the overlap similarity is defined as follows: given two sets, p_i and p_j, and an overlap threshold t; if $|p_i \cap p_j| \geq t$, then there is at least one common token within the π_{p_i}-prefix of p_i and the π_{p_j}-prefix of p_j, where $r = |p_j| - t + 1$ and $s = |p_j| - t + 1$.

An example of how prefix filter works is reported in Figure 2.5. The prefixes for overlap threshold $t = 4$ are highlighted in grey. Since the two prefixes do not share any token, the pair $\langle p_i, p_j \rangle$ can be pruned. The intuition behind this is that the 3 remaining tokens to check can provide at most a similarity of 3, that is not enough to reach the requested threshold t.

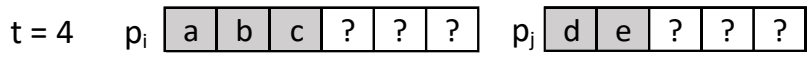

Figure 2.5: Prefix filter.

Length filter

A filter that is commonly used in conjunction with the prefix filter is the length filter [AGK06]. Normalized similarity functions (e.g., Jaccard, Cosine, Dice, ED) depend on the set size, thus it is possible to exploit it to prune the pairs generated with the prefix filter. For the Jaccard Similarity the length filter is defined as: a set of elements r can reach Jaccard threshold t only with a set s of size $lb_r \leq |s| \leq ub_r$ ($lb_r = t \cdot |r|, ub_r = \frac{|r|}{|t|}$); for example, if $|r| = 10$ and $t = 0.8$, then $8 \leq |s| \leq 12$ is required.

Positional filter

The positional filter [Xia+11] reasons over the matching position of tokens in the prefix. Given a pair of sets of sorted elements it checks the positions of their common tokens in the prefix, if the remain tokens to check are not enough to reach the threshold, it prunes the pair. Since it needs to scan the tokens in the prefix, this filter is more expensive than prefix and length filters, so usually it is applied only on the pairs that already passed them.

An example of how positional filter works is provided in Figure 2.6. The pair $\langle p_i, p_j \rangle$ passes both length and prefix filters. The first match in p_i occurs in position 1 (counting from 0), thus only 8 tokens of p_j are left to match tokens of p_i, and the pair can be filtered because it can never reach the requested threshold $t = 9$.

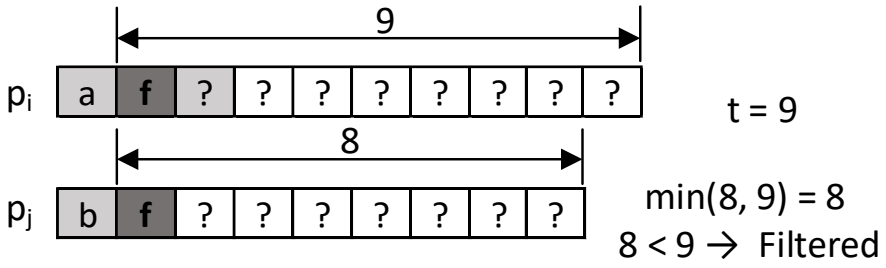

Figure 2.6: Positional filter example.

Prefix filter based set similarity join

An example of how a prefix filter based set similarity join works is outlined in Figure 2.7. Starting from a document collection, the documents are transformed in sets of elements (e.g., tokens, n-grams, etc.) and sorted according to a global order (1). Then, using the prefixes (highlighted in gray) an inverted index is built, i.e., the prefix index (2). From the prefix index, a set of pre-candidate pairs is built (3), i.e., each pair of profiles that appear together in at least one entry of the prefix index. The pre-candidate pairs are filtered using different filters (e.g., length filter, positional filter, etc.) that are fast to compute and let to discard the pairs that cannot reach the threshold (4). Finally, the pairs that pass all the filters (i.e., candidate pairs) are probed with the similarity function, and only those that have a similarity above the threshold are retained (5).

2.4 MapReduce-like Systems

In MapReduce-like Systems, programs are written in functional style and automatically executed in parallel on a cluster of machines. These systems also

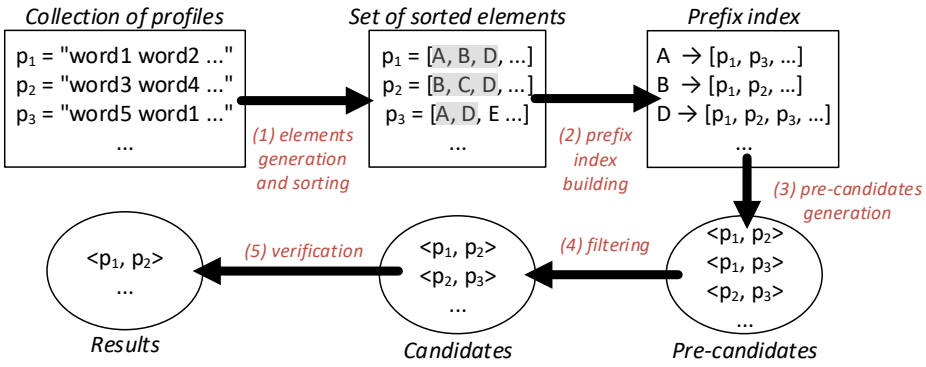

Figure 2.7: Prefix filter based Similarity Join process.

provide automatic mechanisms for load balancing and to recover from machine failures without recomputing the whole program by leveraging on the functional programming abstraction (e.g., *lazy evaluation* in Apache Spark [Zah+12]). In the following, we present the main *functions* employed to formalize MapReduce-like algorithms in this thesis with a concise and *Spark-like* syntax. These functions are defined w.r.t. *Resilient Distributed Dataset* (RDD [Zah+12]), which are the basic data structure in Apache Spark. In a nutshell, an RDD is a distributed and resilient collection of objects (e.g.: *integers*, *strings*, etc.).

Basic Functions for MapReduce-like Algorithms

- `map` (`map` in MapReduce [DG08]) applies a given function to all elements of the RDD returning a new RDD.
- `flatMap` applies a given function that returns multiple elements to all elements of the RDD, than concatenates all the results returning a new RDD.
- `mapPartitions`: applies a given function to each RDD partition returning a new RDD.
- `reduceByKey` (`reduce` in MapReduce [DG08]) reduces the elements for each key of an RDD using a specified commutative and associative binary function.
- `groupByKey`: groups the values for each key in the RDD into a single

collection.

- `join`: performs a hash join between two RDDs.
- `intersection`: performs the intersection between two RDDs.
- `broadcast`: broadcasts a read-only variable to each node in the cluster (which cache it).

We employ this set of functions for the sake of presentation of our algorithms for MapReduce-like systems (Chapters 3, 4). Yet, the algorithms discussed in this thesis employing such functions are general enough to run on any MapReduce-like systems.

In MapReduce-like systems implementations, functions like `intersection`, `join` and `groupByKey` are notoriously expensive, due to the so-called `shuffling` of data across the network [Spa]. In fact, they involve a redistribution of the data across partitions with the consequent overheads: data serialization/deserialization, transmission of data across the network, disk I/O operations. For instance, `join` implies that all the records that have the same key are sent to the same node. Whereas, `flatMap`, `map` and `mapPartitions` are usually fast to compute, because data is locally processed in memory, and no shuffling across the network is required [Spa].

Chapter 3

Distributed meta-blocking

In this chapter, we describe what is needed to parallelize *Blast* (and schema-agnostic meta-blocking) for taking full advantage out of parallel and distributed computation.

3.1 Distributed Loose Schema Information Extraction

To perform the *loose information extraction* (Phase 1 in Figure 2.3) with MapReduce paradigm it is necessary to implement a MapReduce-based LSH algorithm. Algorithm 1 describes the distributed *loose information extraction* process using the LSH [LRU14]. The algorithm takes as input the profile collection P, the similarity threshold t, the number of hashes n_h needed for the LSH, and provides as output the clusters of similar attributes.

First, the attributes with all their tokens (i.e., single words) are extracted from the profile collection (line 1), producing a set of pairs $\langle attribute, [tokens] \rangle$. Then, it is built an index that given a token provides all its hashes (line 2). The index is sent in broadcast to all worker nodes (line 3). Then, for each attribute the signatures are generated using the previously generated hashes (lines 5-9). The width of the bands is computed according to the number of signatures and the threshold (line 11), as explained in [LRU14]. The signatures are split in chunks of *bw* size (lines 12-13), each chunk represents the identifier of a bucket, if two attributes appear together in the same bucket (i.e., their signatures generate the same chunk) they are considered similar. The buckets are generated by grouping the attributes by the bucket id (line 16). Then, for each pair of attributes that appear together in the same bucket, is computed the Jaccard Similarity using their signatures (lines 18-21). For each attribute a list of its similar ones is computed with their similarities (line 23). Then, for each attribute only the most similar one is kept (lines 24-26), for example if there are the pairs $\langle A_i, A_j, sim_{i,j} \rangle$, $\langle A_i, A_k, sim_{i,k} \rangle$ and $sim_{i,j} > sim_{i,k}$ only the first one is retained. Finally, the clusters of attributes are built by applying the transitive closure on the retained pairs (line 27).

3.2 Distributed Blocks Generation

For *loosely schema-aware blocking* (Phase 2 in Figure 2.3), adapting the proposed solution of Section 2.2.3 to the MapReduce paradigm is straightforward. Then, adapting Token Blocking to the MapReduce paradigm is straightforward as well (it essentially builds an inverted index).

Algorithm 1 Loose information extraction

Input: P, the profile collection
Input: t, similarity threshold
Input: n_h, number of hash
Output: AC, the list of clusters of attributes

1: $A_T \leftarrow extractAttributeValues(P)$ //$\langle attribute, [tokens]\rangle$
2: $H \leftarrow generateHashes(A_T, n_h)$ //Generates the hashes for each token
3: **broadcast**$(hashes)$
4: $S \leftarrow \{\}$ //Attribute's signatures
5: **map** $A_i \in A_T$
6: $\quad H_i \leftarrow \{\}$
7: $\quad$ **for each** $t_j \in A_i.tokens$ **do**
8: $\quad\quad H_i \leftarrow H_i \cup H(t_j)$
9: $\quad S \leftarrow S \cup \langle A_i.attribute, minHash(H_i)\rangle$ //Attribute signature
10: $B \leftarrow \{\}$ //Map attributes to buckets
11: $bw \leftarrow getBandWidth(n_h, t)$
12: **map** $S_i \in S$
13: $\quad B_{ids} \leftarrow split(S_i.sig, bw)$
14: $\quad$ **for each** $id_j \in B_{ids}$ **do**
15: $\quad\quad B \leftarrow B \cup \langle id_j, \langle S_i.attr, S_i.sig\rangle\rangle$
16: $buckets \leftarrow$ **groupByKey** (B)
17: $E \leftarrow \{\}$ //Pairs of attributes with similarity
18: **map** $b_i \in buckets$
19: $\quad$ **for each** $\langle A_j, A_k\rangle (j \neq k) \in b_i$ **do**
20: $\quad\quad sim_{j,k} \leftarrow sim(A_j.sig, A_k.sig)$
21: $\quad\quad E \leftarrow E \cup \langle A_j.attr, (A_k.attr, sim_{j,k})\rangle \cup \langle A_k.attr, (A_j.attr, sim_{j,k})\rangle$
22: $E_{max} \leftarrow \{\}$ //Most similar attribute for each attribute
23: $simAttr \leftarrow$ **groupByKey** (E)
24: **map** $simAttr_i \in simAttr$
25: $\quad mostSimilar_i \leftarrow getMostSimilar(simAttr_i.attrList)$
26: $\quad E_{max} \leftarrow E_{max} \cup \langle simAttr_i.attr, mostSimilar_i\rangle$
27: $AC \leftarrow transitiveClosure(E_{max})$
28: **return** AC

The main challenge for the parallelization of *Blast* is related to the graph-based meta-blocking step. In fact, the blocking-graph, defined in Chapter 2, is an abstract model useful to formalize and devise meta-blocking methods. However, materializing and processing the whole blocking-graph may be challenging in the context of big data due to the size of such a graph. For this reason, algorithms for processing the blocking-graph have been proposed to scale meta-blocking to large datasets on MapReduce-like systems [Eft+17].

Algorithm 2 *Repartition Meta-blocking* [Eft+17]

Input: P, the profile collection
Output: C, the list of retained comparisons

```
1: P^K ← ∅
2: C ← {}    // retained comparisons
3: map ⟨profile⟩ ∈ P
4:     for each k ∈ getKeys(profile) do
5:         P^K ← P^K ∪ ⟨key, profile⟩
6: P^J ← P^K join P^K on key    // self-join
7: P^G ← groupByKey (P^J)
8: map ⟨profileNeighborhood⟩ ∈ P^G
9:     C_p ← prune(profileNeighborhood)
10:    C.append(C_p)
```

Their basic idea is to distribute the blocking-graph processing on multiple machines, trading a fast execution for high resource occupation.

In the following, firstly we revise the state-of-the-art blocking-graph processing algorithm, i.e., *repartition meta-blocking*[1] [Eft+17], discussing its limitations; then, we present our novel algorithm called *broadcast meta-blocking*, which overcome these limitations.

3.3 Distributed Blocking-graph Processing

Repartition meta-blocking—At the core of *repartition meta-blocking* [Eft+17] there is a full materialization of the blocking graph.

Algorithm 2 describes the *repartition meta-blocking* with pseudocode. Firstly, for each $profile$ and for each of its blocking key, a pair $\langle key, profile \rangle$ is generated (Lines 3-5). The result can be seen as a table P^K with two columns: *key* and *profile*. Then, a self-join on P^K (Line 6) and a group by profile (Lines 7) are performed. In practice, this corresponds to a graph materialization, since each node is associated with a copy of its local neighborhood. As a matter of fact, each element of P^G (Line 7) is a set of pairs $\langle p_i,\ p_j \rangle$, where p_i is fixed and p_j is a profile sharing at least one blocking key with p_i.

Finally, for each profile p_i and its neighborhood (Lines 8-10), a pruning

[1] In [Eft+17] this algorithm is called *entity-based parallel meta-blocking* (an example is shown in Figure 14 of [Eft+17]) and it is the state-of-the-art (i.e., fastest and efficient) algorithm for performing node-centric pruning on the blocking graph; we coined the term *repartition meta-blocking* for the analogy with the *repartition join* algorithm [Bla+10].

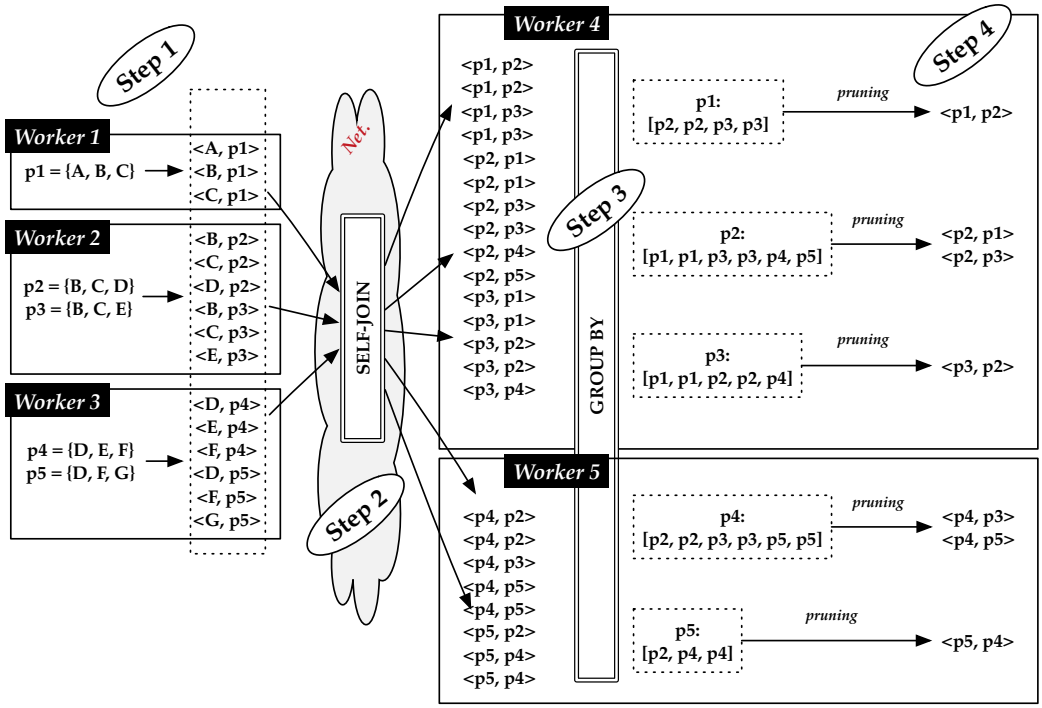

Figure 3.1: Repartition meta-blocking example

function computes a local threshold θ_i and retains only the edges with a weight higher than θ_i (Lines 9)[2].

Optimization note—When implementing *repartition meta-blocking*, for alleviating the network communication bottleneck, blocks and profiles are represented by their ids, as proposed in [Eft+17]. This means that, for Algorithm 2, the pair $\langle key,\ profile \rangle$ (in Line 5) is a pair of identifiers: the first id represents the *key* (i.e., the block), the second id represents the entity profile.

An example of the execution steps of *repartition meta-blocking* is shown in Figure 3.1. Five profiles are grouped in three partitions: $\{p_1\}$, $\{p_2; p_3\}$ and $\{p_4; p_5\}$. Each partition is assigned to a *worker* (i.e., a physical computational node) that computes the $\langle key, profile \rangle$ pairs (*Step 1*). The resulting set of pairs P^K is then employed for a self *join* in order to yield the bag of all the comparison pairs $\langle p_i, p_j \rangle$; this step (Step 2) requires a shuffling of the

[2]Some pruning functions requires as input both the local threshold of the current node p_i and the local threshold of its neighbors; in this case, (Lines 8-10) are executed two times: first, for computing all the thresholds (which are then broadcasted); then, for the actual pruning.

data (P^K) through the network (note that only the ids of the profiles are sent around the network). The comparison pairs are assigned to a *worker* according to their keys, so the *group by* operator partitions them to materialize the neighborhoods within each worker (Step 3). Thus, in parallel, each neighborhood can be processed to generate the final restructured block collection (*Step 4*).

The bottleneck of *repartition meta-blocking* is the `join` (Line 6 in Algorithm 2). In fact, Efthymiou et al. [Eft+17] describe it as a *standard repartition join* [Bla+10] (a.k.a. *reduce-side join*), a notoriously expensive operator for MapReduce-like systems[3]. A workaround for this issue could be the employment of *broadcast join* [Bla+10], a join operator for MapReduce-like systems that is very efficient if one of the join tables can fit in main memory. Unfortunately, P^K (Line 6 in Algorithm 2) typically cannot fit in memory with large dataset (e.g., those employed in our experiments in Section 5.1). Thus, *broadcast join* cannot be employed in Algorithm 2.

Broadcast meta-blocking—To avoid the repartition join bottleneck, we propose a novel algorithm for parallel meta-blocking inspired by the broadcast join. The key idea of our algorithm is the following: instead of materializing the whole blocking graph, only a portion of it is materialized in parallel. This is possible by partitioning the nodes of the graph and sending in broadcast (i.e., to each partition) all the information needed to materialize the neighborhood of each node one at a time. Once the neighborhood of a node is materialized, the pruning functions that can be applied are the same employed in *repartition meta-blocking* [Eft+17], and (non-parallel) *meta-blocking* [Pap+16; Pap+14].

The pseudocode of *broadcast meta-blocking* is shown in Algorithm 3 and described in the following. Given the profile collection $\mathcal{P}$ the block index I_B is generated (Lines 1-2): it is an inverted index listing the profile ids of each block (blocks are represented through ids as well). When executing *Blast*, the functions *buildBlocks* and *buildBlockIndex* also extract the loose schema information—i.e., they basically perform what is described in Section 3.2. Then, I_B is broadcasted to all workers (Line 4), in order to make it available to them. On each partition, an index I_P is built (Lines 5-6): for each profile it lists the block identifiers in which it appears. Then, for each partition and for each profile, by using the I_P and I_B indexes, a profile's neighborhood at a time is built locally (Lines 7-9): for each block id contained in I_P it is possible to obtain from I_B the list of profile ids (the neighbors). Finally, it

[3]We make explicit the *join* operator: Efthymiou et al. present their algorithms in [Eft+17] by using a only `map` and `reduce` functions.

Algorithm 3 *Broadcast Meta-blocking*

Input: P, the profile collection
Output: C, the list of retained comparisons

1: $B \leftarrow buildBlocks(P)$
2: $I_B \leftarrow buildBlockIndex(B)$
3: $C \leftarrow \{\}$ `// retained comparisons`
4: **broadcast**(I_B)
5: **map partition** $\langle part \rangle \in P$
6: $I_P \leftarrow buildProfileBlockIndex(I_B)$
7: **for each** $profile \in part$ **do**
8: $B_{ids} \leftarrow I_P[profile.id]$
9: $profileNeighborhood \leftarrow buildLocalGraph(B_{ids}, I_B)$
10: $C_p \leftarrow$ **prune(*profileNeighborhood*)**
11: $C.append(C_p)$

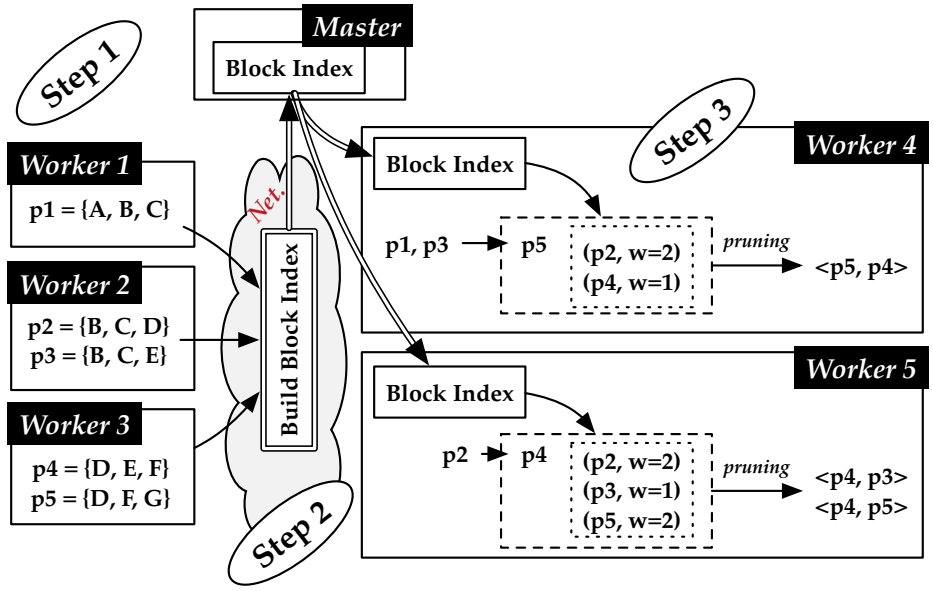

Figure 3.2: Broadcast meta-blocking example

performs the pruning (Lines 10-11)[4].

Note that the `prune` function employed in Algorithm 2 (Line 9) and Algorithm 3 (Line 10) takes as input a profile's neighborhood and can be any node-centric pruning function, e.g., the one described in Section 2.2.2.

An example of the execution steps of *broadcast meta-blocking* is shown in

[4]As for Algorithm 2, for some pruning functions, this last iteration has to be performed twice: the first time for computing all the thresholds, the second for the actual pruning.

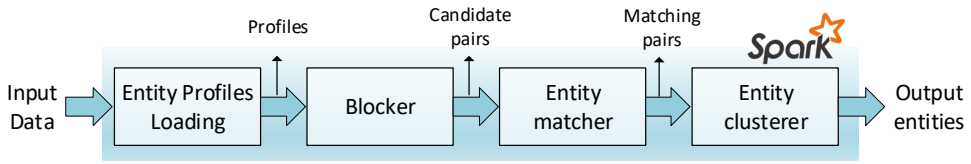

Figure 3.3: *SparkER* architecture.

Figure 3.2. In *Step 1* the profiles are partitioned and assigned to the workers. Then, in *Step 2*, the inverted index of blocks (the *Block Index*) is built—for the sake of the example, the intermediate steps to build the inverted index are not depicted. This step requires a shuffling of data through the network, but at a significantly lower extent compared to that needed for the self-join operation of *repartition meta-blocking*. Then, the *Block Index* is broadcasted to all the workers that perform the last phase of the processing (*Step 2*). Finally, in *Step 3*, each worker processes a partition of the profile set: it materializes a neighborhood at a time by exploiting the local instance of the *Block Index*, and performs pruning to yield the final restructured block collection.

3.4 SparkER

We implemented the previous described broadcast meta-blocking (see Section 3.3) in *SparkER* [Gag+19; Sim+18a; Gag+18; Gag+17]. *SparkER* is a distributed entity resolution tool, composed by different modules designed to be parallelizable on Apache Spark. *SparkER* is freely available on my GitHub repository[5]. Figure 3.3 shows the architecture of our system. There are 3 main modules: (1) **blocker**: takes the input profiles and performs the blocking phase, providing as output the candidate pairs; (2) **entity matcher** takes the candidate pairs generated by the blocker and label them as match or no match; (3) **entity clusterer** takes the matched pairs and groups them into clusters that represents the same entity. Each of these modules can be seen as black box: each one is independent from the other.

3.4.1 Blocker

Figure 3.4 shows the `blocker`' sub-modules implementing the *Loose-Schema Meta-Blocking* method described in the introduction.

[5] `https://github.com/Gaglia88/sparker/`

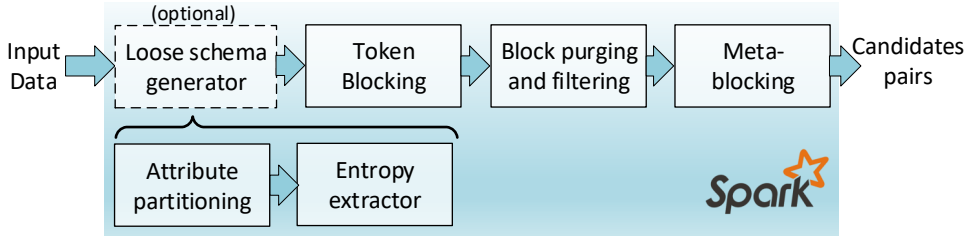

Figure 3.4: Blocker module

Loose Schema Generator-Attribute Partitioning: attributes are partitioned in cluster using a *Locality-sensitive Hashing* (LSH) based algorithm. Initially, LSH is applied to the attributes values, in order to group them according to their similarity. These groups are overlapping, i.e., each attribute can compare in multiple clusters. Then, for each attribute only the most similar one is kept, obtaining pairs of similar attributes. Finally, the transitive closure is applied to such attributes pairs and then attributes are partitioned into nonoverlapping clusters. All the attributes that do not appear in any cluster are put in a *blob* partition.

Loose Schema Generator-Entropy Extractor: computes the Shannon entropy for each cluster.

Block Purging and Filtering : the block collection is processed to remove/shrink its largest blocks [Pap+16]. *Block Purging* discards all the blocks that contain more than half of the profiles in the collection, corresponding to highly frequent blocking keys (e.g., stop-words). *Block Filtering* removes each profile from the largest 20% blocks in which it appears, increasing the precision without affects the recall.

Meta-Blocking: Finally, the *meta-blocking* method 2.2.2 is applied.

The output of the `blocker` module are profile pairs connected by an edge, which represent candidate pairs that will be processed by the *entity matcher* module.

3.4.2 Entity Matcher and Clusterer

Regarding Entity Matching, any existing tool can be used, for example Magellan [Kon+16]. The `Entity Matcher` produces *matching pairs* of similar profiles with their similarity score (*similarity graph*). The user can select from a wide range of similarity (or distance) scores, e.g.: Jaccard similarity,

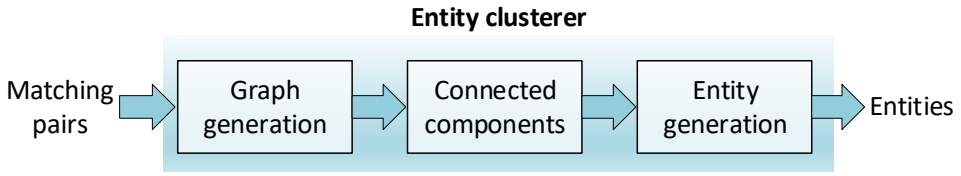

Figure 3.5: Entity clusterer.

Edit Distance.

The `Entity Clusterer` receives as input the similarity graph, in which the profiles are the nodes and the matching pairs represent the edges, and partition its nodes into *equivalence clusters* such that every cluster contains all profiles that correspond to the same entity. Several entity clustering algorithms have been proposed in literature [Has+09]; at the moment, we use the *connected component* algorithm[6], based on the the assumption of *transitivity*, i.e., if p_1 matches with p_2, p_2 matches with p_3, then p_1 matches with p_3. At the end of this step, the system produces clusters of profiles: the profiles in the same cluster refer to the same real-world entity.

3.4.3 Process debugging

The tool can work in a completely unsupervised mode, i.e. the user can use a default configuration and performs the process on its data without taking care of the parameters tuning. Otherwise, the user can supervise the entire process, in order to determine which are the best parameters for its data, producing a custom configuration. Given the iterative nature of this process (e.g., the user try a configuration, if it is not satisfied changes it, and repeat the step again), it is not feasible to process the entire input data, as the user should waste too much time. Thus, it is necessary to sample the input data, reducing the size. The main problem is to take a sample that represents the original data, and also contains matching and non matching profiles. This problem was already addressed in [Kon+16], where the authors proposed to pick up some random K profiles P^K, then for each profile $p_i \in P^K$ pick up $k/2$ profiles that could be a match (i.e. shares a high number of token with p_i) and $k/2$ profiles randomly. K and k are two parameters that can be set by the user based on the time that she wants to spend (e.g., selecting more

[6]This approach is implemented by using the GraphX library of Spark (`https://spark.apache.org/graphx/`) that natively implement the *connected component* approach.

Figure 3.6: (a) the task analysis presents a summary of the results through four charts (precision, recall, F1 score, execution time), also let to see the detailed information about all steps; (b) if the loose schema-aware token blocking is used, it is possible to see and edit the generated attribute clusters; (c) it is possible to explore the identified as duplicate records.

records requires a higher computation time).

Each step can be assessed using precision and recall, if a ground-truth is available; otherwise the system selects a sample of the generated profile pairs (e.g., pairs after blocking, matching pairs after matching, etc.) and shows them to the user who, on the basis of his experience evaluates whether the system is working well or not.

In the `blocker` each operation (blocking, purging, filtering, and meta-blocking) can be fine tuned in order to obtain better performances, e.g., the purging/filtering are controlled through parameters that can change the aggressiveness of filters, or the *meta-blocking* can use different types of pruning strategies, etc. Moreover, if the *Loose Schema Blocking* is used, it is possible to see how the attributes are clustered together, and how to change the clustering parameters in order to obtain better clusters.

In the entity matching phase, it is possible to try different similarity techniques (e.g., Jaccard, cosine, etc.) with different thresholds.

At present no tuning activity is possible in the *clustering* step since the

connected component algorithm used does not have any parameters. At the end of the process, the system allows to explore the generated entities and to store the obtained configuration. Then, the optimized configuration can be applied to the whole data in a batch mode, in order to obtain the final result.

At the end, through the GUI the user can analyze the obtained results as Figure 3.6 show.

Chapter 4

Record Level Matching Rules

In this chapter, we present our method *GraphJoin* to efficiently scale record-level matching rules over big data sets. The presented algorithm is the self-join version for the sake of the presentation; adapting it for joining two different data sets is straightforward. To the best of our knowledge, no technique has been proposed to leverage on distributed and parallel computing for scaling record-level matching rules. The benefit is twofold: *(i)* distributed computation allows to scale to large data sets that cannot be handled with a single machine; *(ii)* parallel execution reduces the execution time. As a matter of fact, being able to efficiently execute similarity join is crucial when time is a critical component, e.g., when users are involved in the process. For instance, in exploratory search in a data lake [Nar+19], users typically look for related data sets and low latency in performing similarity join is required for enabling the user's interactive exploration. Also, when debugging record-level matching rules, users typically try different configurations of similarity metrics, thresholds, and attributes. Hence, enabling fast execution of such rules can significantly save user's time. Recap that we define a matching rule $\mathcal{R}$ as a *disjunction* (logical *OR*) of *conjunctions* (logical *AND*) of similarity join predicates on multiple attribute (i.e., at the *record level*).

4.1 Baseline algorithm

Given a matching rule $\mathcal{R}$ a naïve solution to perform it is to process each predicate as a single similarity join and then intersect/merge the obtained results according to the requirements. In particular, we adopted two algorithms to perform the similarity joins: *PPJoin* [Xia+11] and *EDJoin* [XWL08]. Both algorithms employ *prefix filter* (see Section 2.3) to find candidate pairs. *PPJoin* is considered one of the best performing similarity join algorithm [MAB16], also its parallel implementation (i.e., Vernica Join) has demonstrated to be one of the best performing for distributed computing [Fie+18]. It can work with different similarity measures like Jaccard Similarity, Dice and Cosine. *EDJoin* adapts the *PPJoin* concepts to work with the Edit Distance. We adapted both algorithms to work on Spark as proposed in [VCL10] for *PPJoin*.

The distributed algorithm to perform PPJoin/EDJoin is presented in Algorithm 4. First, the records are transformed into sets of sorted elements according to the predicate requirements (line 1). The prefix index (see Subsection 2.3.1) is built generating an inverted index that groups all the records that share at least one token in their prefix. Then, the algorithm iterates over each entry of the prefix index B_i, probing each pair of records $\langle r_i, r_j \rangle \in B_i$

with the appropriate filters according to the predicate $\mathcal{P}$.

Algorithm 5 outlines the baseline algorithm (i.e., *JoinChain*). A rule in DNF format is composed of different blocks P_i of predicates that are in logical *OR*, each of these blocks contains one or more predicates p_i that are in logical *AND*. First, the algorithm iterates over the P_i blocks (line 2) and for each of them initializes a set of candidates C_{P_i} (line 3). Then, each simple predicate $p_j \in P_i$ is used to apply a similarity join on the record collection R according to requirements (lines 6-9). The result of a P_i block is given by the intersection of the results provided by each similarity join applied with the predicate $p_j \in P_i$ (lines 10-13). The final candidate set is computed by merge the results of each P_i block (line 14). In the end, the candidates are verified with a verify function that ensures that all predicates are respected (line 15).

Algorithm 4 PPJoin/EDJoin

Input: R collection of records to join
Input: $\mathcal{P}$ predicate that contains the join attribute, the threshold, and the elements pattern (i.e., tokens, n-grams, etc.)
Output: C, the pairs of records that can satisfy $\mathcal{P}$

1: $R_T \leftarrow getSortedElements(R, \mathcal{P})$ //Transforms the records in set of sorted elements (i.e., tokens, n-grams, etc.)
2: $I \leftarrow buildPrefixIndex(R_T, \mathcal{P})$ //Build prefix index
3: $C \leftarrow$ **flatMap** $B_i \in I$ //For each entry of the prefix index
4: **for each** $\langle r_j, r_k \rangle \in B_i (r_j \neq r_k)$ **do**
5: **if** $passLengthFilter(r_j, r_k, \mathcal{P})$ **then**
6: **if** $passPositionalFilter(r_j, r_k, \mathcal{P})$ **then**
7: $emit(\langle r_j, r_k \rangle)$

4.2 GraphJoin

Algorithm 5 has three main drawbacks:

(i) the *intersect* operation (line 13) is expensive in MapReduce-like systems, because it generates *shuffle* (see Subsection 2.4);

(ii) a predicate is independently checked by the others. For example, given a matching rule $M = (C1 \wedge C2) \vee (C3 \wedge C4)$ in which each Cx is a similarity join predicate (e.g., Jaccard Similarity $title \geq 0.8$). A pair $\langle r_i, r_j \rangle$ is probed with all predicates even if it fails/passes one of them. For example, if the pair passes the predicate $(C3 \wedge C4)$ it is not necessary to probe it with $(C1 \wedge C2)$. Or, if it fails with the predicate $C1$ it is not necessary to probe it with $C2, C3$.

Algorithm 5 JoinChain

Input: R collection of records to join
Input: $\mathcal{M}$ matching rule in DNF form
Output: M, the pairs of records that satisfy $\mathcal{M}$

```
1: C ← {}
2: for each P_i ∈ M do //For each block of predicates in or
3:     C_{P_i} ← {}  //Set of candidate pairs for P_i
4:     for each p_j ∈ P_i do //For each single predicate in and
5:         C_{p_j} ← {}  //Set of candidate pairs for p_j
6:         if p_i.type = ED then
7:             C_{p_j} ← EDJoin(R, p_j) //Get candidate pairs with EDJoin
8:         else
9:             C_{p_j} ← PPJoin(R, p_j) //Get candidate pairs with PPJoin
10:        if C_{P_i}.isEmpty then //Intersects candidates with previous ones
11:            C_{P_i} ← C_{p_i}
12:        else
13:            C_{P_i} ← C_{P_i} ∩ C_{p_j}
14:    C ← C ∪ C_{P_i} //Merge candidates with previous ones
15: return verify(C, M)
```

(iii) Vernica Join, employed in the implementation of *JoinChain* algorithm (Algorithm 5 lines 7, 9), produces duplicates [Fie+18] that have to be removed. If a pair of records appears in more prefix entries, it is processed and emitted multiple times (Algorithm 4 lines 3-7).

We solved these problems in our GraphJoin algorithm. The main intuition of GraphJoin is to exploit the prefix indexes—one prefix index for each predicate of the matching rule—to build a graph structure, which is then employed to iterate over the records (the nodes of the graph), efficiently applying the rules and to keep only the candidates (the edges of the graph) that pass the whole rule. In other words, GraphJoin adopts a record-based parallelization approach; in contrast to the existing algorithms, which adopt a prefix-based parallelization approach on a single predicate at a time.

The GraphJoin matching rule execution algorithm is outlined in Algorithm 6.

The algorithm takes as input a collection of records and a record-level matching rule $\mathcal{M}$ and gives as output the set of record pairs that satisfy $\mathcal{M}$. Recall that $\mathcal{M}$ is in DNF, i.e., it is composed of sets of predicates P_j in *logical or*, each set P_j contains predicates p_k in *logical and*. First of all, the values of attributes are converted into sets of elements (Line 1) according to the matching rule requirements (e.g., n-grams, trigrams, tokens, etc.); then the prefix indexes are built to find the candidate pairs (line 2)—one prefix index is needed for each predicate p_k of the matching rule. The prefix indexes are

Algorithm 6 GraphJoin

Input: R collection of records to join
Input: $\mathcal{M}$ matching rule in DNF
Output: C, the pairs of records that satisfy $\mathcal{M}$

```
1: R_T ← getElements(R, M)
2: I ← buildPrefixIndexes(R_T, M)
3: broadcast(I)
4: C ← {} //Candidate pairs
5: map partition part ∈ R_T
6:     for each r_i ∈ part do
7:         C_{r_i} ← {} //Candidates for r_i
8:         for each P_j ∈ M do //For each set of predicates in logical or
9:             C_{P_j} ← {} //Candidates that satisfy P_j
10:            for each p_k ∈ P_j do //For each predicate in logical and
11:                C_{r_i,p_k} ← I(p_k, r_i) //Gets the candidates from the prefix index
12:                /*Removes candidates that already passed previous predicates in or
13:                and those that did not pass previous predicates in and*/
14:                C_{r_i,p_k} ← C_{r_i,p_k} − C_{r_i}
15:                if C_{P_j} ≠ ∅ then
16:                    C_{r_i,p_k} ← C_{r_i,p_k} ∩ C_{P_j}
17:                /*Applies filters (length, positional, ...)*/
18:                C_{P_j} ← applyFilters(r_i, C_{r_i,p_k}, p_k)
19:            C_{r_i} ← C_{r_i} ∪ C_{P_j}
20:        C.append(C_{r_i})
21: return verify(C, M)
```

sent in broadcast to each node (line 3) to be available to each computational node (called *worker*). Then, each worker iterates over its portion of records (lines 5-6), and performs the following operations for each record r_i. First, a set of candidates for r_i is initialized as an empty set C_{r_i} (line 7). Second, for each set P_j, a set of candidates C_{P_j} is initialized as an empty set (lines 8-9) and for each $p_k \in P_j$ the candidates C_{r_i,p_k} that can match with r_i are extracted using the prefix indexes (lines 10-11). Third, the candidates C_{r_i,p_k} are pruned by removing those that already passed one of the previous P_j set of predicates (line 14), and those that did not passed previous $p_k \in P_j$ predicates (lines 15-16). Fourth, the retained candidates are probed with other filters that further improve the efficiency of the overall process (e.g., length filter, position filter, etc. [Xia+11; XWL08]) according to the rule (line 18). Since p_k is in *logical and* with the previous predicates, only the candidates that pass the filters are kept. The resulting candidates from P_j are added to C_{r_i} (line 20). Finally, the candidates are verified (line 21).

Given a matching rule $R = (C1 \wedge C2 \wedge C3) \vee (C4 \wedge C5)$, in which each Cx is a similarity join predicate (e.g., Jaccard Similarity $title \geq 0.8$); an example

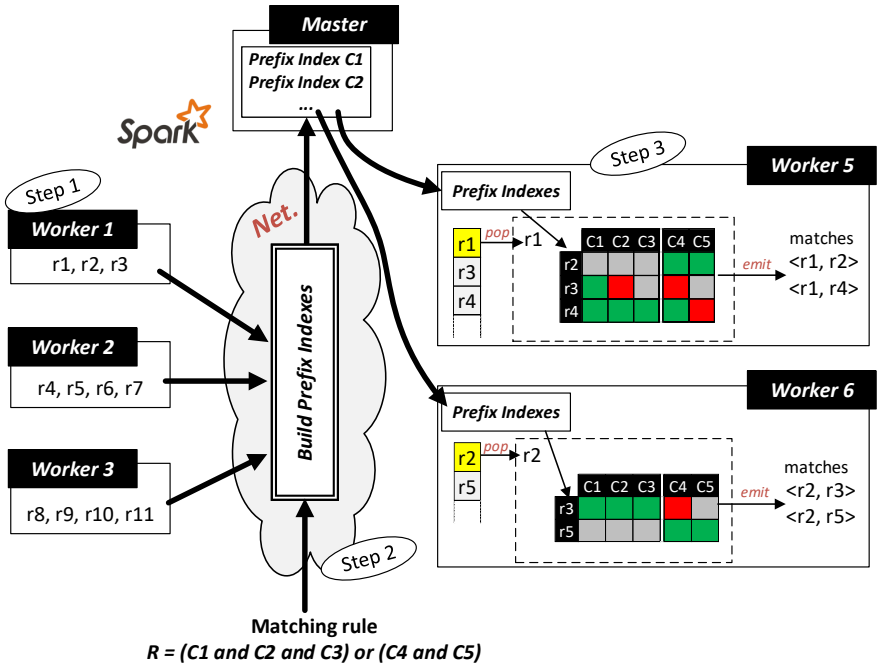

Figure 4.1: *GraphJoin* execution model: green cells represents executed and passed rules; red cells executed that do not pass the rules; grey cells not executed rules.

of how *GraphJoin* executes R is outlined in Figure 4.1. First, a prefix index is built on the basis of the record-level matching rules expressed in the main matching rule R. Then, the index is distributed to each worker. Each worker iterates over each record in its partition extracting the possible candidates from the prefix index. The rules are applied to each candidate. If more rules are in *or* it is possible to avoid computing the other rules when one of them is verified, e.g., with $\langle r1, r2\rangle$ the rule $(C1 \wedge C2 \wedge C3)$ is not verified since the pair passes the rule $(C4 \wedge C5)$. Otherwise, if more rules are in *and*, it is possible to avoid the computation when one of them fails, for example for the pair $\langle r1, r3\rangle$, C2 fails, so C3 has not to be computed.

4.3 RulER

RulER [GSB20b; GSB20a], is a tool that employ the previously described *GraphJoin* to efficiently execute record-level matching rules on parallel and distributed systems, developed on top of Apache Spark. The tool is freely

available on my GitHub repository[1]. It can be easily employed for data preparation tasks (i.e., to join data sets to be consumed by data analytic tasks) and to support the user in debugging record-level matching rules.

RulER it is very simple to use, as Figure 4.2 shows. For example, suppose to have two movies data sets like `Rotten Tomatoes`[2] and `Roger Erbert`[3]. The former gather users' ratings about movies, the latter critics' ratings. There is no foreign key between the two data sets. We want to discover if there is a correlation between the ratings given by the users with the ones given by the critics. To do that, we need to define a matching rule to integrate the two data sets, then we compute the Pearson correlation on the obtained results. A matching rule that can be employed can be: $(\text{"}movie\_name\text{"}, \text{"}Title\text{"}, JS, 0.8) \wedge (\text{"}actors\text{"}, \text{"}Cast\text{"}, JS, 0.5)$, That means that the JS between the names of the movies must be greater or equal than 0.8 and the JS between the actors of the movies must be greater or equal than 0.5. After the matching, it is possible to obtain a scatter plot of the ratings, and compute the Pearsons correlation rating given by critics and the rating given by users on the same movie.

The user just has to include RulER library (line 1), load the data as Spark Dataframe[4] (lines 3-4), to define the rules (lines 6-7) and combines them to obtain the final matching rule (line 9). Finally, the matching rule can be used to join the two dataframes with the *joinWithRules* method (line 11).

[1] `https://github.com/Gaglia88/ruler/`

[2] https://www.rottentomatoes.com

[3] https://www.rogerebert.com

[4] `https://spark.apache.org/docs/latest/sql-programming-guide.html#datasets-and-dataframes`

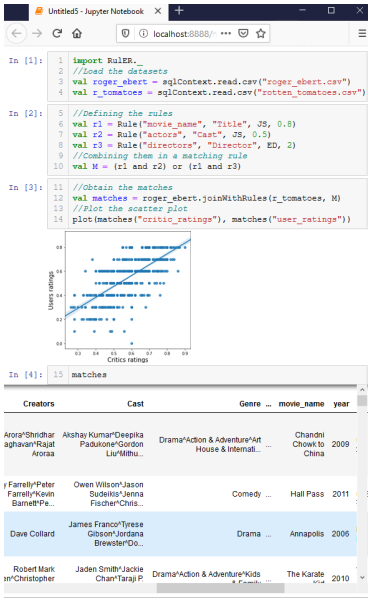

Figure 4.2: *RulER* example usage.

Chapter 5

Experimental Evaluation

In this chapter we evaluate the novel approaches (i.e. parallel *Blast* and *GraphJoin*) proposed in this thesis.

All the experiments are performed on a ten-node cluster; each node has two Intel Xeon E5-2670v2 2.50 GHz (20 cores per node) and 128 GB of RAM, running Ubuntu 14.04. All the software is implemented in Scala 2.11.8 and available at [Gag+17]. To assess the performance of the state-of-the-art meta-blocking methods we re-implemented all of them for running on Apache Spark as well. We employ Apache Spark 2.1.0, running 3 executors on each node, reserving 30 GB of memory for the master node. We set the default parallelism to twice the number of cores as suggested by best practice[1].

5.1 Distributed Blast evaluation

In this section we evaluate *Blast* and schema-agnostic meta-blocking. In particular the experimental evaluation aims to answer the following questions:

$Q1$: *What is the performance of Blast in terms of precision, recall, and execution time compared to the state-of-the-art [Pap+16]?* (Section 5.1.1)

$Q2$: *How efficient is broadcast meta-blocking, compared to repartition meta-blocking [Eft+17]?* (Section 5.1.2)

$Q3$: *How does Blast (with broadcast meta-blocking) scale when varying the number of machines available for the ER processing?* (Section 5.1.3)

Experimental Setup

Datasets—Table 5.1 lists the 7 real-world datasets employed in our experiments. They have different characteristics and are from a variety of domains. The small datasets (i.e., `articles1`, `articles2`, `products`, and `movies`) are used only when evaluating the performance in terms of recall and precision, since their time performance on distributed setting is not significant. (Table 5.3 reports the definition of precision and recall from Section 2.2.3.)

All the datasets match two different data sources for which the ground truth of the real matches is known. From [KTR10]: `articles1` matches scientific articles extracted from dblp.org and dl.acm.org; `articles2` matches scientific articles extracted from dblp.org and scholar.google.com. `products` matches products extracted from Abt.com and Buy.com. From [Pap+13]: `movies` matches movies extracted from imdb.com and dbpedia.org; `dbpedia`

[1] https://spark.apache.org/docs/latest/tuning.html

	Size	$\lvert\mathcal{P}_1\rvert - \lvert\mathcal{P}_2\rvert$	$\lvert\mathcal{A}_1\rvert - \lvert\mathcal{A}_2\rvert$	$\lvert\mathcal{D}_P\rvert$
`articles1` (*)	small	2.6k - 2.3k	4 - 4	2.2k
`articles2` (*)	small	2.5k - 61k	4 - 4	2.3k
`products` (*)	small	1.1k - 1.1k	4 - 4	1.1k
`movies`	small	28k - 23k	4 - 7	23k
`articles3` (*)	large	1.8M - 2.5M	7 - 7	0.6M
`dbpedia`	large	1.2M - 2.2M	30k - 50k	0.9M
`freebase`	large	4.2M - 3.7M	37k - 11k	1.5M

Table 5.1: Dataset characteristics: number of entity profiles, number of *attribute names*, and number of existing matches. An exact schema alignment can be achieved only on starred "(*)" datasets.

matches entity profiles from two different snapshots of DBpedia (2007 and 2009)[2]. From [Das+]: `articles3` matches scientific articles extracted from Citeseer and DBLP. Finally, `freebase` is derived from the Billion Triple Challenge 2012 Dataset [Har12]: it is composed by two datasets, one contains the data of DBpedia 3.7, the other one the data of Freebase; we cleaned these two datasets keeping only the information in English, removing other languages; the ground truth is represented by the *owl:sameAs* relationships between them.

Methods Configurations and Results Analysis—For each dataset, the initial block collection is extracted through a redundant blocking technique (either Token Blocking or Loose Schema Blocking). Then, the block collection is processed with Block Purging and Block Filtering [Pap+16], which aim to remove/shrink the largest blocks in the collection. Block Purging discards all the blocks that contain more than half of the entity profiles in the collection, corresponding to highly frequent blocking keys (e.g., stop-words). Block Filtering removes each profile p_i from the largest 20% blocks in which it appears[3]. The time required by both Block Purging and Block Filtering is negligible compared to the meta-blocking phase, thus not listed in the experimental results.

The schema-agnostic meta-blocking methods can be executed on blocks generated with both Token Blocking and with Loose Schema Blocking, while *Blast* is compatible with the latter only, since it exploits the loose schema information.

For the schema-agnostic meta-blocking methods, we report the average

[2]Only 25% of the name-value pairs are shared among the two snapshots, due to the constant changes in DBpedia, therefore the ER is not trivial.

[3]This heuristic has shown to not affect recall in practice, while lighting the blocking-graph handling [Pap+16].

Blocking

TB	Token Blocking [Pap+13] (see Section 2.2.1)
LSB	Loose-Schema Blocking (see Section 2.2.3)

Meta-blocking

WNP	Weight Node Pruning [Pap+16] (see Section 2.2.2)
CNP	Cardinality Node Pruning [Pap+16] (see Section 2.2.2)
$WNP_{OR}(CNP_{OR})$	The *redefined* WNP (CNP) approach [Pap+16] (see Section 2.2.2). An edge is not pruned if it weight is greater than any of its adjacent node's local thresholds (OR condition)
$WNP_{AND}(CNP_{AND})$	The *reciprocal* WNP (CNP) approach [Pap+16] (see Section 2.2.2). An edge is not pruned if it weight is greater than both of its adjacent node's local thresholds (AND condition)

Table 5.2: Acronyms and configurations.

values of recall, precision, F1-score[4] and time obtained by executing each method in combination with each of the five weighting schemas proposed in [Pap+13][5]. We also report that no traditional weighting schema and pruning strategy combination performs better than the other on the considered datasets, confirming the results of [Pap+13].

Finally, for the time measurement, we report the values obtained by averaging the times recorded for five runs.

Table 5.2 summarizes the acronyms used in this Section.

5.1.1 Blast vs. Schema-agnostic Meta-blocking

Table 5.2 summarizes the acronyms and configurations employed in this experiment. WNP and CNP is applied on block collections generated both with Token Blocking (TB) and Loose Schema Blocking (LSB), and employing both *redefined* ($\text{WNP}_{\text{OR}}/\text{CNP}_{\text{OR}}$) and *reciprocal* ($\text{WNP}_{\text{AND}}/\text{CNP}_{\text{AND}}$) approaches (see Section 2.2.2).

[4] Hand et al. [HC18] have recently discussed how F1-score may be an unreliable measure for comparing different ER algorithms. We report F1-score for the sake of completeness—it has been used in many related works [Kon+16; Ebr+18; Mud+18]—yet we draw conclusions on the basis of precision and recall only.

[5] Among the weighting schemas proposed in [Pap+13], we did not identify an overall best performer and an overall worst performer, confirming the results reported in [Eft+17], for this reason we report the average precision, recall, F1-score and execution time.

$\|\mathcal{B}\|$	Number of comparisons entailed by a block collection $\mathcal{B}$
$\|\mathcal{D}^{\mathcal{P}}\|$	Number of duplicates (matches) in a profile collection $\mathcal{P}$
$\|\mathcal{D}^{\mathcal{B}}\|$	Number of duplicates (matches) indexed in at least one block $b \in \mathcal{B}$
$recall(\mathcal{B})$	$\|\mathcal{D}^{\mathcal{B}}\|/\|\mathcal{D}^{\mathcal{P}}\|$
$precision(\mathcal{B})$	$\|\mathcal{D}^{\mathcal{B}}\|/\|\|B\|\|$

Table 5.3: Metrics.

Figure 5.2 shows the result of the execution of *Blast* and traditional meta-blocking on all the datasets. Compared to WNP approaches, *Blast* achieves significantly higher precision and basically the same level of recall on all the datasets, confirming the results reported in [SBJ16]. In particular *Blast* always outperforms LSB+$\text{WNP}_{\text{OR/AND}}$, demonstrating that the *Blast* weight-based pruning is actually more effective than the traditional ones.

Compared to TB+$\text{CNP}_{\text{OR/AND}}$, *Blast* achieves higher precision on all the datasets, with the exception of `articles2` and `freebase`, where CNP_{AND} has a higher precision (Figure 5.2(i) and Figure 5.2(n)). Notice though that on `articles2` and on `freebase` *Blast* achieves a recall significantly higher (Figure 5.2(b) and Figure 5.2(g)). On all the other datasets, the recall of *Blast* is almost the same of TB+$\text{CNP}_{\text{OR/AND}}$ (Figure 5.2(a-g)), or slightly higher (Figure 5.2(b) and Figure 5.2(g)). Similarly, *Blast* outperforms LSB+$\text{CNP}_{\text{OR/AND}}$ in terms of precision on all the datasets but `articles2` and `freebase` (Figure 5.2(i) and Figure 5.2(n)). Yet, on these datasets *Blast* yields a higher recall (Figure 5.2(b) and Figure 5.2(g)).

We also considered the overall execution time of the methods. For the comparison, we employed our Spark implementation of them, employing *broadcast-meta-blocking* as core blocking-graph processing algorithm, running on a single node (for scalability and performance on multiple nodes see Section 5.1.3). In such a configuration, for the small datasets the results are not reported: the overhead introduced by Spark in each execution does not allow to properly record the actual time efficiency of such configuration when the size of the data is small[6]. The results are shown in Figure 5.1. *Blast* is always significantly faster than $\text{CNP}_{\text{OR/AND}}$ on all the considered datasets and all the configurations (up to 3.8× on `dbpedia` in Figure 5.1(b)). It is also faster than TB+$\text{WNP}_{\text{OR/AND}}$ on `dbpedia` (2.8× in Figure 5.1(b)) and `freebase` (1.6× in Figure 5.1(c)); while, on `articles3` is slightly slower (Figure 5.1(a)). Compared to LSB+$\text{WNP}_{\text{OR/AND}}$, *Blast* has almost the same

[6]In [**DBLP:journals/pvldb/SimoniniBJ16**] the time for these datasets are reported for the Java implementation and the results are analogous.

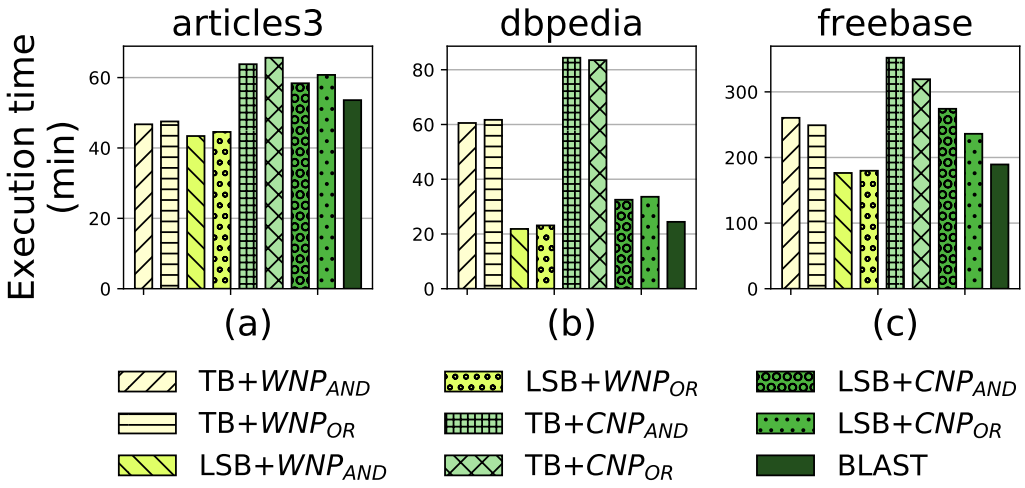

Figure 5.1: Execution time of the different methods applied on blocks obtained with the Token Blocking (TB+WNP$_{\text{ADN/OR}}$/CNP$_{\text{ADN/OR}}$) and with the Loose Schema Blocking (LSB+WNP$_{\text{ADN/OR}}$/CNP$_{\text{ADN/OR}}$). The execution time is referred to the *meta-blocking*, and it was taken on a single node on the biggest datasets.

execution time on `dbpedia` (Figure 5.1(b)) and `freebase` (Figure 5.1(c)); while on `articles3` is slightly slower (Figure 5.1(a)).

Overall, we conclude that *Blast* yields the same recall and a significantly higher precision of the best performing schema-agnostic meta-blocking methods [Pap+16], on each dataset [7]. The only exception is LSB+CNP$_{\text{OR/AND}}$, which achieves higher recall than *Blast* on two of the seven considered datasets (Figure 5.2(i) and Figure 5.2(n)), but at the same time has lower recall (Figure 5.2(b) and Figure 5.2(g)) and is always slower than *Blast* Figure 5.1. Finally, we also observe that *Blast* has time performance similar to the fastest schema-agnostic method.

[7] The differences between *Blast* and WNP/CNP are statistically significant according to Student's T-Test (with p-value < 0.05).

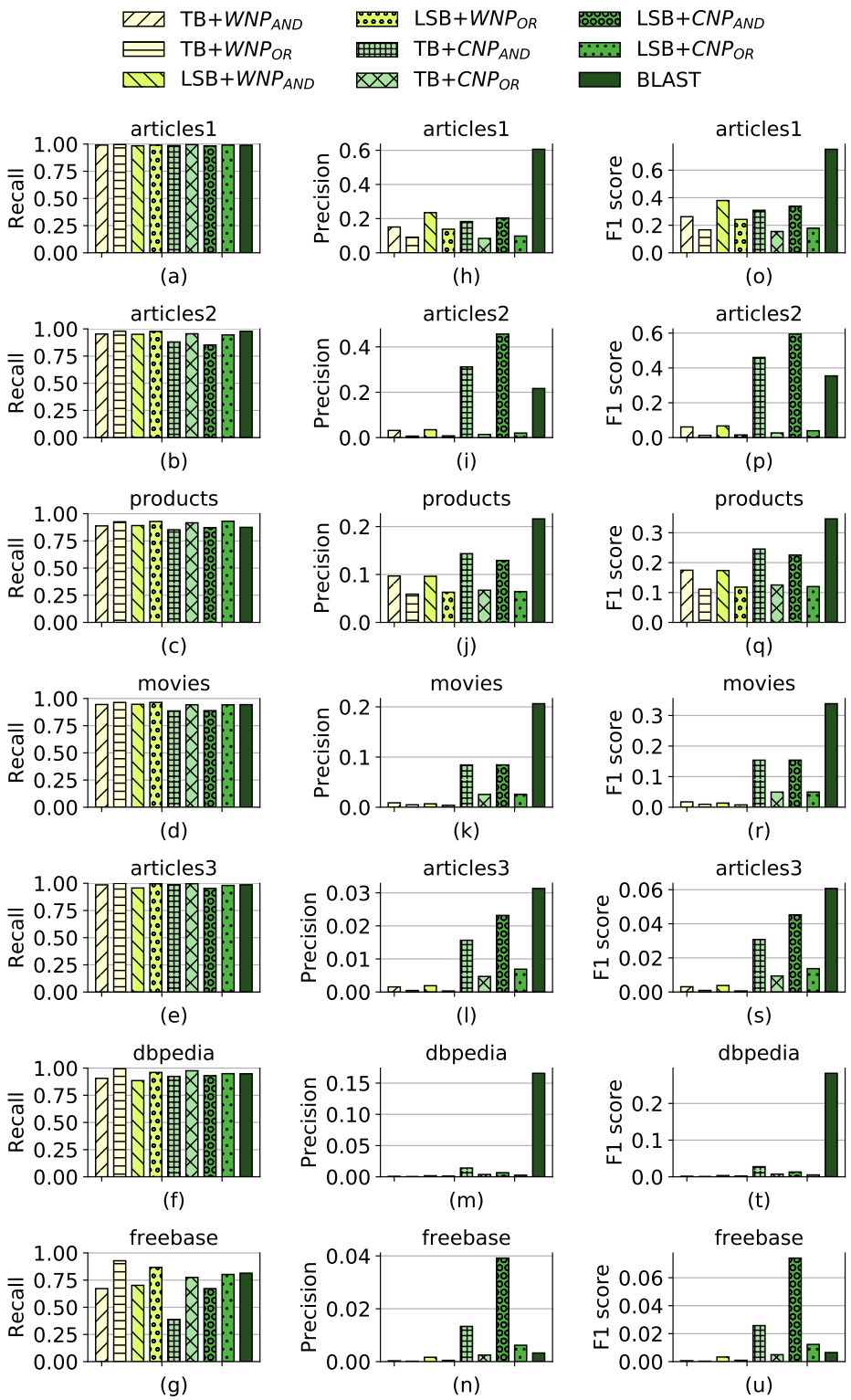

Figure 5.2: Recall and precision achieved by the considered methods on all the datasets. Traditional meta-blocking ($\text{WNP}_{\text{ADN/OR}}$ and $\text{CNP}_{\text{ADN/OR}}$) has been combined both with Token Blocking (TB+$\text{WNP}_{\text{ADN/OR}}$/$\text{CNP}_{\text{ADN/OR}}$) and Loose Schema Blocking (LSB+$\text{WNP}_{\text{ADN/OR}}$/$\text{CNP}_{\text{ADN/OR}}$). *Blast* is based on Loose Schema Blocking for the extraction of the loose schema information, thus it is not applicable on block collection generate with Token Blocking.

5.1.2 Broadcast vs. Repartition Meta-blocking

The goal of this experiment is to compare the efficiency of *broadcast meta-blocking* (Algorithm 2) and *repartition meta-blocking* (Algorithm 3). Both the algorithms can be employed as core graph-processing algorithms for any meta-blocking method. Thus, we evaluate them in combination with WNP and CNP, in order to analyze how they perform on both family of meta-blocking, i.e., those based on weight-threshold, and those based on cardinality-threshold (see Section 2.2.2). To minimize additional overhead, we run them in combination with the computationally cheapest weighting function, i.e., *block co-occurrence frequency* (we record analogues trends with other weighting functions). The experiment was performed on 10 nodes. We consider only the large datasets since the overhead introduced by Spark does not pay off on the small ones on multiple nodes. Notice that both algorithms perform the same logical operation, that is the final recall and precision are the same on all the datasets, hence not reported here.

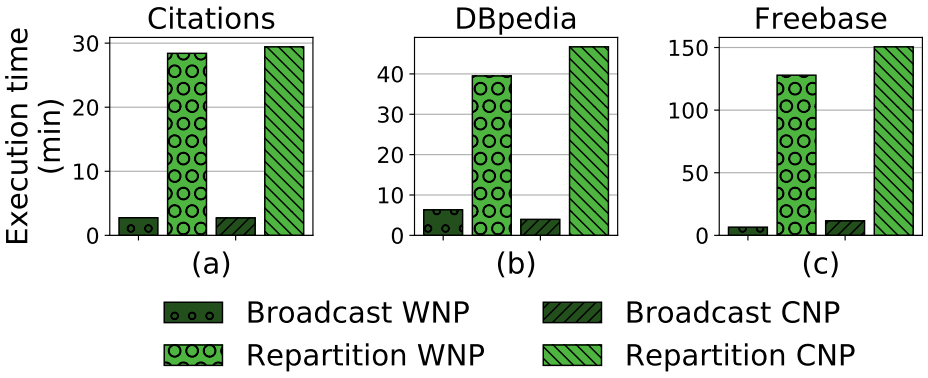

Figure 5.3: *Repartition* vs. *Broadcast meta-blocking*. For each dataset we report two different strategies for the ***prune*** functions, i.e., the weight- and cardinality-based pruning. This times was taken on 10 nodes.

The results are reported in Figure 5.3: *broadcast meta-blocking* is faster than *repartition meta-blocking* from 4.9 to 12.7 times for WNP, and from 7.7 to 10.1 times for CNP. To analyze the scalability of the algorithms, we report in Figure 5.4 their execution times in function of the number of nodes (from 1 to 10) on `freebase` (the largest dataset). In our setting, *repartition meta-blocking* is not able to run with less than 7 nodes; whereas *broadcast meta-blocking* on a single node is 3 to 4 times faster than the execution time of the *repartition meta-blocking* on 10 nodes.

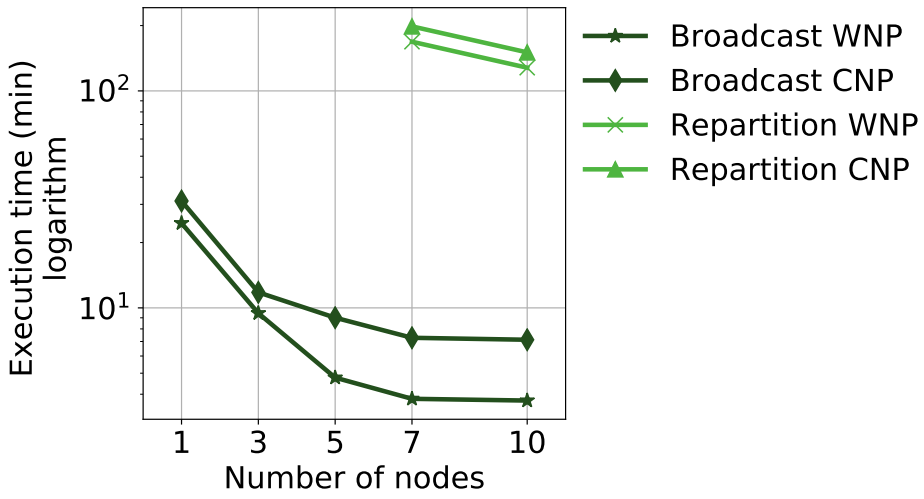

Figure 5.4: Scalability comparison: *repartition* vs. *broadcast meta-blocking* on `freebase`.

We conclude that the *broadcast meta-blocking* is always faster than the *repartition meta-blocking*.

5.1.3 Parallel-blast scalability

Finally, we assess the scalability of parallel *Blast* by varying the number of nodes in the cluster (1, 3, 5, 7 and 10 nodes). For this experiment we employ `freebase`, which is the heaviest dataset to process due to the huge number of comparisons yielded by the blocking phase (2.23×10^{13} comparisons), and to its large number of attributes ($47,945$ distinct attributes).

Figure 5.6 shows the scalability of each blocking step, i.e.: Loose Schema Blocking (LSB, which is composed of Loose attribute-Match Induction in combination of Token Blocking), and Loose Schema Meta-Blocking (LS-MB). Figure 5.5 shows the speedup of each blocking step, which is sub-linear to the number of nodes in the cluster (i.e. 10x nodes, the overall speedup do not reach 5). For each step, we observe at least a 50% reduction of execution time from 1 to 3 nodes. Then, the execution times continuously decrease until reaching an overall speedup on 10 nodes of 4.2×.

The time and speedup reported so far only consider the blocking and meta-blocking phase of an ER process. In practice, all the comparisons generated through any blocking process have to be compared by means of an *Entity Resolution Algorithm*, which is a binary function that takes as input two profiles and decides whether or not they are matching [Ben+09; Kon+16]. Such a function is typically expensive, e.g., involving string sim-

ilarity computations, calls to external resources or even human intervention (i.e., crowdsourcing). Thus, the more the employed function is expensive, the more useful a good blocking (and meta-blocking) method is; in other words: the resources saved avoiding superfluous comparisons are proportional to the complexity of the *Entity Resolution Algorithm*. Hence, we now compare *Blast* and WNP using a naïve (i.e., cheap) *Entity Resolution Algorithm* for showing that *Blast* significantly reduce the overall execution time of a complete ER process. We employ as *Entity Resolution Algorithm* the computation of the *Jaccard Similarity* of the two profiles involved in each comparison[8].

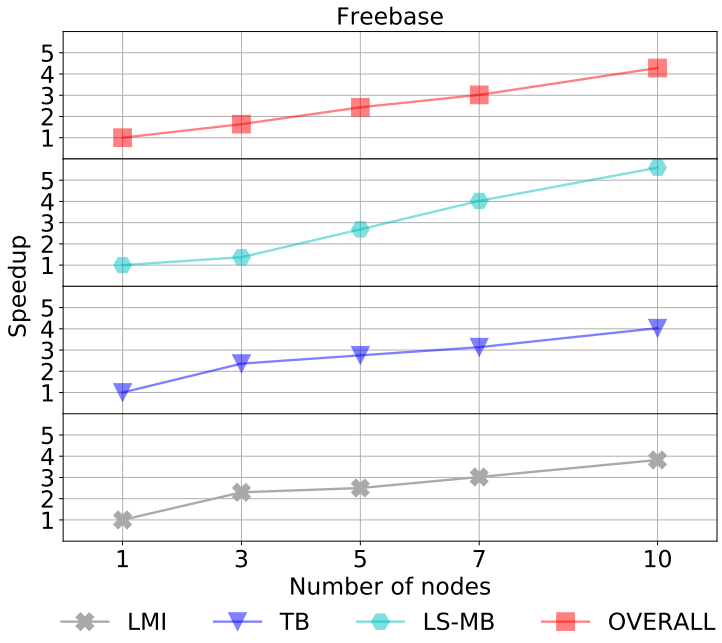

Figure 5.5: Speedup of *Blast* on `freebase`.

[8]In a real-world scenario, a threshold would be required to discriminate between matching and non-matching pairs.

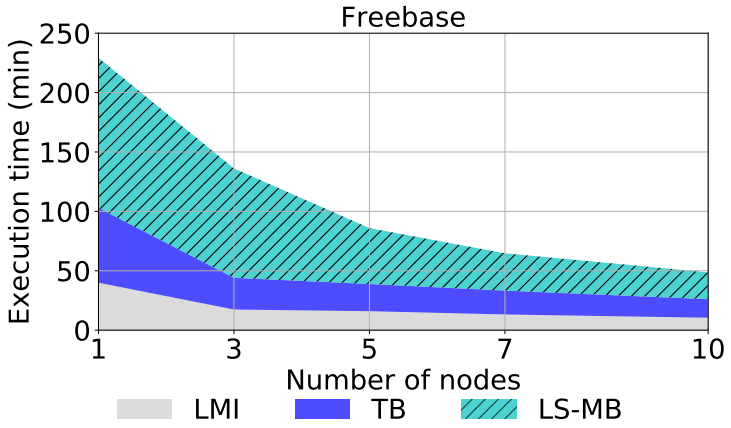

Figure 5.6: Execution time of *Blast* on `freebase`.

Figure 5.7 shows the execution time of *Blast* and WNP in combination with the naïve *Entity Resolution Algorithm*[9] and by varying the number of nodes. We observe that the meta-blocking phase of *Blast* is slower than standard schema-agnostic WNP. This is not surprising, since *Blast* performs an additional step compared to WNP (i.e., *Loose attribute-Match Induction*). Yet, the overall ER process employing *Blast* is significantly faster that employing WNP, since it retains much fewer comparisons ($3.80 \cdot 10^8$ of *Blast* vs. $2.17 \cdot 10^{10}$ of WNP). Please, recall that *Blast* and WNP, on `freebase`, achieve the same recall (Figure 5.2(g)).

[9]The average comparison time on `freebase` is 0.05 ms.

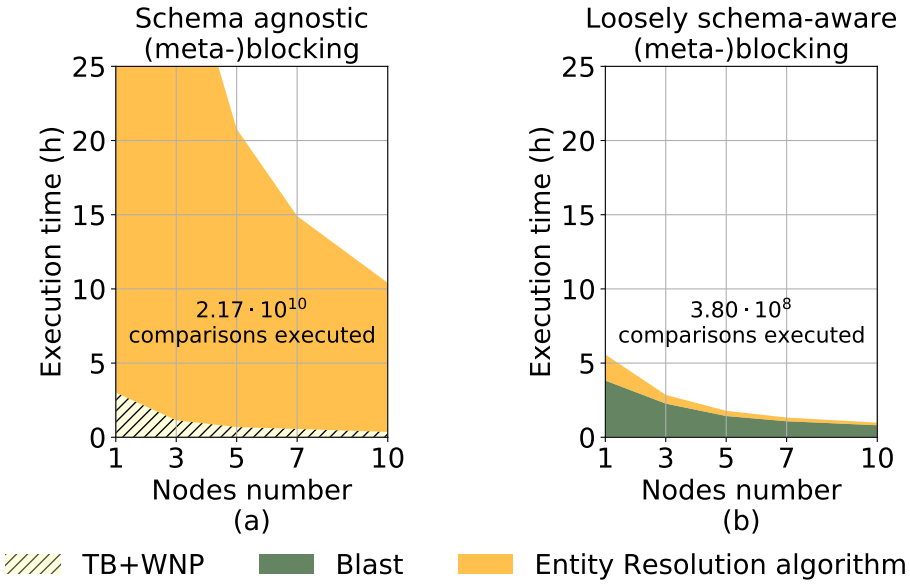

Figure 5.7: Execution time of the complete ER process on `freebase`, varying the number of execution nodes in the cluster. The whole ER process is composed of a blocking phase, which generates candidate pairs that are compared through an Entity Resolution Algorithm. In (a), the blocking method employed is Token Blocking in combination with WNP meta-blocking. In (b), the blocking method employed is *Blast*.

5.2 GraphJoin evaluation

In this section we evaluate *GraphJoin* with respect to *JoinChain* (see Chapter 4). In particular, the experimental evaluation aims to answer the following questions:

$Q1$: *What is the performance of GraphJoin in terms of execution time compared to JoinChain (i.e., the naïve solution)?* (Section 5.2.2)

$Q2$: *How does GraphJoin scale when varying the number of machines available for the record-level matching rule processing?* (Section 5.2.3)

5.2.1 Experimental Setup

We have chosen the `ombd` dataset [Das+] to evaluate the performance of *GraphJoin* and *JoinChain*. It contains 2.3 millions of records about movies

	Rule	**Candidates**	**Matches**
$\mathcal{M}_1$	$(Title, 0.9, JS) \wedge (Cast, 0.8, JS)$	1725885	53023
$\mathcal{M}_2$	$(Title, 0.9, JS) \vee (Cast, 0.8, JS)$	990278774	253593213
$\mathcal{M}_3$	$((Cast, 0.9, JS) \wedge (Director, 0.9, JS) \wedge (Writer, 0.9, JS)) \vee (Title, 0.9, JS)$	61987445	34798235
$\mathcal{M}_4$	$(Director, 0.9, JS) \wedge (Title, 2, ED)$	1133134	252935

Table 5.4: Matching rules employed in the experiments. For each rule the number of candidate pairs obtained after the filters (i.e., prefix filter, length filter, positional filter, see Section 2.3) is reported, together with the number of final matching pairs.

collected from *omdbapi.com*. We choose this dataset because it has a good variety of attributes (e.g., title, cast, director, writer, plot, etc.) on which it is possible to define a different kind of record-level matching rules. Moreover, it contains a sufficient number of records that make it suitable to test the performance with Spark.

The goal of our experiments is to compare the efficiency of the algorithms, not to find good matching rules. Moreover, with *GraphJoin* it would be possible to define an order for the application of the predicates that minimize the execution time and the number of performed comparisons, but we do not explore this aspect in this work.

5.2.2 *GraphJoin* vs *JoinChain*

The goal of this experiment is to compare the efficiency of *GraphJoin* (Algorithm 6) and *JoinChain*(Algorithm 5). Both algorithms can be employed to apply a record-level matching rule $\mathcal{M}$. In this experiment we apply the rules presented in Table 5.4 on the `omdb` dataset. Since both algorithms use the same functions to extract the elements from the records, to generate the prefix indexes and to verify the candidate pairs, we analyze here only the time requested to perform the *join* operation. All the experiments are performed on a single node.

Figure 5.8 reports the execution times of *GraphJoin* and *JoinChain*with the rules reported in Table 5.4. *GraphJoin* is always significantly faster than *JoinChain*: 24× with $\mathcal{M}_1$ (Figure 5.8(a)), 5× with $\mathcal{M}_2$ (Figure 5.8(b)), 7× with $\mathcal{M}_3$ (Figure 5.8(c)), 27× with $\mathcal{M}_4$ (Figure 5.8(d)). Moreover, Figure 5.9 shows the number of comparisons performed by both algorithms. Also in this case, *GraphJoin* works better than *JoinChain* performing less comparisons: $21.6 \cdot 10^6$ less for $\mathcal{M}_1$ (Figure 5.9(a)), $23.0 \cdot 10^6$ less for $\mathcal{M}_2$ (Figure 5.9(b)),

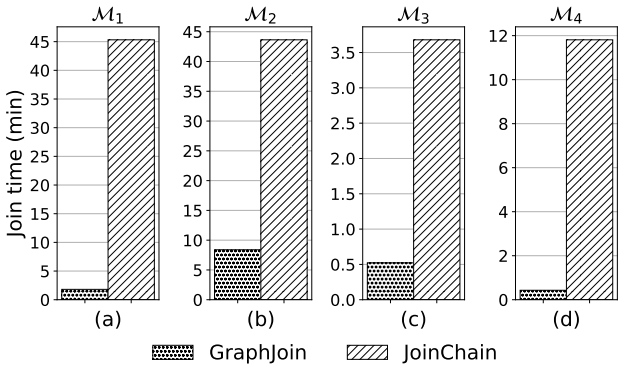

Figure 5.8: Execution times of *GraphJoin* and *JoinChain* with the rules reported in Table 5.4.

$3.8 \cdot 10^6$ less for $\mathcal{M}_3$ (Figure 5.9(c)), $45.1 \cdot 10^6$ less for $\mathcal{M}_3$ (Figure 5.9(d)).

We conclude that *GraphJoin* is always faster than *JoinChain*.

5.2.3 *GraphJoin* scalability

Finally, we assess the scalability of *GraphJoin* by varying the number of nodes in the cluster (1, 3, 5, 7 and 10 nodes). For this experiment we apply the rules described in Table 5.4 on the `omdb` dataset.

Figure 5.10 shows the scalability and the speedup of the whole process. For each step, we observe at least a 50% reduction of execution time from 1 to 3 nodes. Then, the execution times continuously decrease until reaching an overall speedup on 10 nodes of: 5.0× for $\mathcal{M}_1$, 7.3× for $\mathcal{M}_2$, and 4.16× for $\mathcal{M}_3$. $\mathcal{M}_2$ is the rule that takes more advantage in the increase of worker'

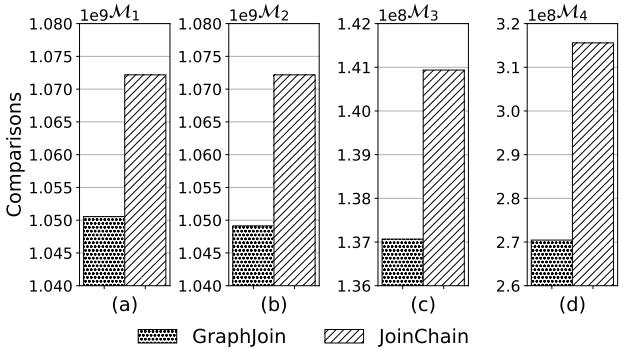

Figure 5.9: Number of comparisons performed by *GraphJoin* and *JoinChain* with the rules reported in Table 5.4.

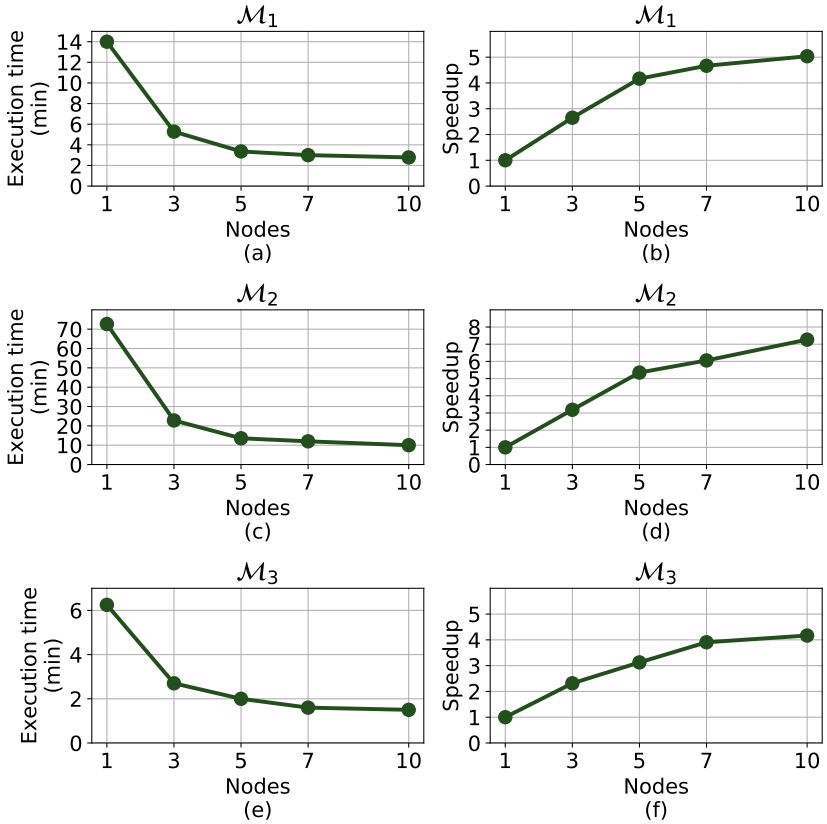

Figure 5.10: Execution time and speedup of *GraphJoin* with the rules reported in Table 5.4.

nodes because it performs more comparisons than the others, as shown in Table 5.4.

Chapter 6

Related Work

Blocking—Blocking techniques have been commonly employed in Entity Resolution (ER) [KR10; NH10; Chr12b; Kon+16; Sto+13; Pap+18; Eft+18; Sar+11; DN11], and can be classified into two broad categories: *schema-based* (Suffix Array [DV+09], q-grams blocking [Gra+01], Canopy Clustering [MNU00]), and *schema-agnostic* (Token Blocking [Pap+13], Progressive ER [Sim+18b; Sim+18c; WMG13; PHN15; FSS16; Fir+18], and *Attribute-match induction* [Pap+13; MT13]).

Attribute-match induction—Among the schema-agnostic techniques, *Attribute Clustering* (AC) [Pap+13] and *TYPiMatch* [MT13] try to extract statistics to define efficient blocking keys. AC relies on the comparison of all possible pairs of attribute profiles of two datasets to find the pairs of those most similar; this is an inefficient process because the vast majority of comparisons are superfluous. The LSH-based preprocessing step adopted in *Blast* aims to address this specific issue. *TYPiMatch* tries to identify the latent *subtypes* from generic attributes (e.g., description, info, etc.) frequent on generic dataset on the Web, and uses this information to select blocking keys; but it cannot efficiently scale to large dataset.

Block manipulation—In this thesis, we work the problem of meta-blocking, i.e., how to restructure (*manipulate*) an existing blocking collection, for improving the quality of the overall ER process. The state-of-the-art, *unsupervised* and schema-agnostic meta-blocking has been presented in [Pap+16]. *Blast* was shown to outperform them in Section 5.1.1. *Supervised* meta-blocking [PPK14; BGD18] extends the blocking graph model by representing each edge as a vector of schema-agnostic features (e.g., graph topological measures), and treats the problem of identifying most promising edge as a *classification* problem; hence, a training set of labeled data (matching/non-matching pairs) is required. *Blast* exploits the *loose schema information* and does not require any training set (i.e., it is completely unsupervised).

Entity Resolution with MapReduce-like Systems—Parallel and distributed versions of traditional (schema-based) blocking techniques have been extensively studied in [KTR12; Das+17]. Altowim and Mehrota [AM17] have investigated how to generate candidate profile pairs on MapReduce-like systems in *pay-as-you-go* (i.e., progressive) fashion. Their proposed solution relies on the definition of schema-based blocking keys. Finally, Efthymiou et al. [Eft+17] have proposed the *repartition meta-blocking* algorithm to run graph-based meta-blocking methods on MapReduce. In Sections 3 and 5.1.2, we extensively compare it against our proposed *broadcast meta-blocking* algorithm.

Araújo et al. [APN17] have proposed a novel schema-agnostic pruning strategy called *Global Weighted Node Pruning* (GWPN) that combines a lo-

cal threshold with a global one. The local threshold is computed for each profile as for the WNP, while the global one is computed as the average of all the edges weights. This strategy aims to discard the edges with a low weight that connects only profiles with a very low local threshold. Compared to traditional WNP, GWNP improves the precision of 0.01%, while achieving the same recall, on DBpedia dataset [APN17]. Araújo et al. also discuss a Spark implementation for their strategy, which is based on the MapReduce parallel meta-blocking proposed in [Eft+17], and suffers from the same limitations (see Section 3.3).

Record Level Matching-Rules—In this thesis, we tackle the problem of execute *record-level matching rules* written in DNF (see Section 2.3). To the best of our knowledge, the only related work tackling this problem is [ADA+18], which focuses on single-node execution, borrowing optimization techniques from the traditional relational database approaches. Similarly, [Li+15] focuses on how to optimize multi-attribute similarity join, but only for conjunctions of predicates (i.e., not for DNF). We were not able to find any work about the parallelization of these techniques.

Chapter 7

Conclusions and Future Work

Conclusion. In this thesis, I tackled the problem of joining records without explicit join keys, presenting distributed data integration algorithms that allow to scale to large volumes of data (i.e., Big Data): *repartition meta-blocking* and *GraphJoin*. Both of the algorithms are implemented to efficiently run on Spark and in general on MapReduce-like Systems. In particular, *repartition meta-blocking* allows to efficiently running any meta-blocking technique, while *GraphJoin* allows to efficiently apply *record-level matching rules*.

I implemented two tools of the developed algorithms: *SparkER* an ER framework for Apache Spark, based on *repartition meta-blocking*; *RulER* a tool that employs *GraphJoin* to efficiently join Spark Dataframes by using *record-level matching rules*.

Finally, I experimentally evaluated the proposed approaches, demonstrating how *repartition meta-blocking* can outperform the state-of-the-art parallel *meta-blocking* implementations (i.e., *repartition meta-blocking*), and how *GraphJoin* can be used to efficiently apply *record-level matching rules*. *SparkER* and *RulER* are available on my GitHub repository[1].

Ongoing and future Work—My ongoing research is focused on the extension of *RulER*, to optimize the execution of the predicates, i.e., given a matching rule to find the optimal order to execute the predicates that compose it to minimize the number of comparisons [Li+15; ADA+18].

Moreover, at the DBGRoup we are working in a joint project with colleagues from Paris and Athens that are the authors of `JedAI` [Pap+18] to integrate their toolkit with *SparkER*, to create a complete ER system that runs seamlessly both on stand-alone computers and clusters of computers. In particular, my contribution is to integrate the parallel *meta-blocking* pipeline (available in *SparkER*) with `JedAI`. Moreover, to parallelize the similarity join and the clustering algorithms included in `JedAI`, and to perform the scalability tests of the ER system by using Apache Spark. The final ER system will be published in the `JedAI` repository on GitHub[2].

[1] `https://github.com/Gaglia88/`

[2] `https://github.com/scify/JedAIToolkit`

Research Products Related to This Thesis

[Gag+17] Luca Gagliardelli, Giovanni Simonini, Song Zhu, and Sonia Bergamaschi. *SparkER: an Entity Resolution tool for Apache Spark*. 2017. URL: `https://github.com/Gaglia88/sparker`.

[GSB20a] Luca Gagliardelli, Giovanni Simonini, and Sonia Bergamaschi. *RulER: a tool for scaling up Record-level Matching Rules*. 2020. URL: `https://github.com/Gaglia88/ruler`.

Publications Related to This Thesis

[Gag+18] Luca Gagliardelli, Song Zhu, Giovanni Simonini, and Sonia Bergamaschi. "Bigdedup: a Big Data integration toolkit for duplicate detection in industrial scenarios". In: *25th International Conference on Transdisciplinary Engineering (TE2018)*. Vol. 7. 2018, pp. 1015–1023.

[Gag+19] Luca Gagliardelli, Giovanni Simonini, Domenico Beneventano, and Sonia Bergamaschi. "SparkER: Scaling Entity Resolution in Spark". In: *EDBT 2019: 22nd International Conference on Extending Database Technology*. 2019.

[GSB20b] Luca Gagliardelli, Giovanni Simonini, and Sonia Bergamaschi. "RulER: Scaling Up Record-level Matching Rules". In: *EDBT 2020: 23nd International Conference on Extending Database Technology*. 2020.

[Sim+18a] Giovanni Simonini, Luca Gagliardelli, Song Zhu, and Sonia Bergamaschi. "Enhancing Loosely Schema-aware Entity Resolution with User Interaction". In: *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2018, pp. 860–864.

[Sim+19] Giovanni Simonini, Luca Gagliardelli, Sonia Bergamaschi, and HV Jagadish. "Scaling entity resolution: A loosely schema-aware approach". In: *Information Systems* 83 (2019), pp. 145–165.

Bibliography

[ADA+18] Adel Ardalan, AnHai Doan, Aditya Akella, et al. "Smurf: self-service string matching using random forests". In: *PVLDB* 12.3 (2018), pp. 278–291.

[AGK06] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. "Efficient exact set-similarity joins". In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment. 2006, pp. 918–929.

[AM17] Yasser Altowim and Sharad Mehrotra. "Parallel Progressive Approach to Entity Resolution Using MapReduce". In: *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. 2017, pp. 909–920. DOI: 10.1109/ICDE.2017.139.

[APN17] Tiago Brasileiro Araújo, Carlos Eduardo Santos Pires, and Thiago Pereira da Nóbrega. "Spark-based Streamlined Metablocking". In: *Computers and Communications (ISCC), 2017 IEEE Symposium on*. IEEE. 2017, pp. 844–850.

[BB04] Domenico Beneventano and Sonia Bergamaschi. "The MOMIS methodology for integrating heterogeneous data sources". In: *Building the Information Society*. Springer, 2004, pp. 19–24.

[BCV99] Sonia Bergamaschi, Silvana Castano, and Maurizio Vincini. "Semantic integration of semistructured and structured data sources". In: *ACM Sigmod Record* 28.1 (1999), pp. 54–59.

[Ben+09] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang, and Jennifer Widom. "Swoosh: a generic approach to entity resolution". In: *The VLDB Journal—The International Journal on Very Large Data Bases* 18.1 (2009), pp. 255–276.

[Ber+11] Sonia Bergamaschi, Domenico Beneventano, Francesco Guerra, and Mirko Orsini. "Data integration". In: *Handbook of Conceptual Modeling*. Springer, 2011, pp. 441–476.

[Ber14] Sonia Bergamaschi. "Big data analysis: Trends & challenges". In: *2014 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2014, pp. 303–304.

[BGD18] Guilherme dal Bianco, Marcos André Gonçalves, and Denio Duarte. "BLOSS: Effective meta-blocking with almost no effort". In: *Information Systems* 75 (2018), pp. 75–89.

[Bla+10] Spyros Blanas, Jignesh M Patel, Vuk Ercegovac, Jun Rao, Eugene J Shekita, and Yuanyuan Tian. "A comparison of join algorithms for log processing in mapreduce". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 975–986.

[BMS07] Roberto J Bayardo, Yiming Ma, and Ramakrishnan Srikant. "Scaling up all pairs similarity search". In: *Proceedings of the 16th international conference on World Wide Web*. ACM. 2007, pp. 131–140.

[BN09] Jens Bleiholder and Felix Naumann. "Data fusion". In: *ACM Computing Surveys (CSUR)* 41.1 (2009), p. 1.

[CES15] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. "Entity resolution in the web of data". In: *Synthesis Lectures on the Semantic Web* 5.3 (2015), pp. 1–122.

[CGK06] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. "A primitive operator for similarity joins in data cleaning". In: *22nd International Conference on Data Engineering (ICDE'06)*. IEEE. 2006, pp. 5–5.

[Chr12a] Peter Christen. "A survey of indexing techniques for scalable record linkage and deduplication". In: *IEEE transactions on knowledge and data engineering* 24.9 (2012), pp. 1537–1555.

[Chr12b] Peter Christen. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Data-Centric Systems and Applications. Springer, 2012. ISBN: 978-3-642-31163-5. DOI: 10.1007/978-3-642-31164-2.

[Das+] Sanjib Das, AnHai Doan, Paul Suganthan G. C., Chaitanya Gokhale, and Pradap Konda. *The Magellan Data Repository*. https://sites.google.com/site/anhaidgroup/projects/data.

[Das+17] Sanjib Das, Paul Suganthan G. C., AnHai Doan, Jeffrey F. Naughton, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, Vijay Raghavendra, and Youngchoon Park. "Falcon: Scaling Up Hands-Off Crowdsourced Entity Matching to Build Cloud Services". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 2017, pp. 1431–1446. DOI: 10.1145/3035918.3035960.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[DHI12] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of data integration*. Elsevier, 2012.

[DN11] Uwe Draisbach and Felix Naumann. "A generalization of blocking and windowing algorithms for duplicate detection". In: *2011 International Conference on Data and Knowledge Engineering, ICDKE 2011, Milano, Italy, September 6, 2011*. 2011, pp. 18–24. DOI: 10.1109/ICDKE.2011.6053920.

[DS15] Xin Luna Dong and Divesh Srivastava. "Big data integration". In: *Synthesis Lectures on Data Management* 7.1 (2015), pp. 1–198.

[DV+09] Timothy De Vries, Hui Ke, Sanjay Chawla, and Peter Christen. "Robust record linkage blocking using suffix arrays". In: *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM. 2009, pp. 305–314.

[Ebr+18] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. "Distributed representations of tuples for entity resolution". In: *Proceedings of the VLDB Endowment* 11.11 (2018), pp. 1454–1467.

[Eft+17] Vasilis Efthymiou, George Papadakis, George Papastefanatos, Kostas Stefanidis, and Themis Palpanas. "Parallel meta-blocking for scaling entity resolution over big heterogeneous data". In: *Information Systems* 65 (2017), pp. 137–157.

[Eft+18] Vasilis Efthymiou, George Papadakis, Kostas Stefanidis, and Vassilis Christophides. "Simplifying Entity Resolution on Web Data with Schema-Agnostic, Non-Iterative Matching". In: *34th IEEE International Conference on Data Engineering, ICDE 2018,*

Paris, France, April 16-19, 2018. 2018, pp. 1296–1299. URL: https://doi.org/10.1109/ICDE.2018.00134.

[Fie+18] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. "Set similarity joins on mapreduce: An experimental survey". In: *Proceedings of the VLDB Endowment* 11.10 (2018), pp. 1110–1122.

[Fir+18] Donatella Firmani, Sainyam Galhotra, Barna Saha, and Divesh Srivastava. "Robust Entity Resolution Using a CrowdOracle". In: *IEEE Data Eng. Bull.* 41.2 (2018), pp. 91–103. URL: http://sites.computer.org/debull/A18june/p91.pdf.

[FSS16] Donatella Firmani, Barna Saha, and Divesh Srivastava. "Online Entity Resolution Using an Oracle". In: *PVLDB* 9.5 (2016), pp. 384–395. DOI: 10.14778/2876473.2876474. URL: http://www.vldb.org/pvldb/vol9/p384-firmani.pdf.

[Gra+01] 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 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. 2001, pp. 491–500.

[Har12] Andreas Harth. *Billion triples challenge data set*. 2012.

[Has+09] O. Hassanzadeh, F. Chiang, H. C. Lee, and R.ée J Miller. "Framework for evaluating clustering algorithms in duplicate detection". In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 1282–1293.

[HC18] David Hand and Peter Christen. "A note on using the F-measure for evaluating record linkage algorithms". In: *Statistics and Computing* 28.3 (2018), pp. 539–547.

[Kon+16] Pradap Konda, Sanjib Das, Paul Suganthan GC, AnHai Doan, Adel Ardalan, Jeffrey R Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, et al. "Magellan: Toward building entity matching management systems". In: *Proceedings of the VLDB Endowment* 9.12 (2016), pp. 1197–1208.

[KR10] Hanna Köpcke and Erhard Rahm. "Frameworks for entity matching: A comparison". In: *Data & Knowledge Engineering* 69.2 (2010), pp. 197–210.

[KTR10] Hanna Köpcke, Andreas Thor, and Erhard Rahm. "Evaluation of entity resolution approaches on real-world match problems". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 484–493.

[KTR12] Lars Kolb, Andreas Thor, and Erhard Rahm. "Dedoop: Efficient Deduplication with Hadoop". In: *PVLDB* 5.12 (2012), pp. 1878–1881. DOI: 10.14778/2367502.2367527.

[Li+15] Guoliang Li, Jian He, Dong Deng, and Jian Li. "Efficient similarity join and search on multi-attribute data". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1137–1151.

[LRU14] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

[MAB16] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. "An empirical evaluation of set similarity join techniques". In: *Proceedings of the VLDB Endowment* 9.9 (2016), pp. 636–647.

[MNU00] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. "Efficient clustering of high-dimensional data sets with application to reference matching". In: *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, MA, USA, August 20-23, 2000*. 2000, pp. 169–178. DOI: 10.1145/347090.347123.

[MT13] Yongtao Ma and Thanh Tran. "Typimatch: Type-specific unsupervised learning of keys and key values for heterogeneous web data integration". In: *Proceedings of the sixth ACM international conference on Web search and data mining*. ACM. 2013, pp. 325–334.

[Mud+18] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. "Deep Learning for Entity Matching: A Design Space Exploration". In: *Proceedings of the 2018 International Conference on Management of Data*. ACM. 2018, pp. 19–34.

[Nar+19] Fatemeh Nargesian, Erkang Zhu, Renée J Miller, Ken Q Pu, and Patricia C Arocena. "Data lake management: challenges and opportunities". In: *PVLDB* 12.12 (2019), pp. 1986–1989.

[NH10] Felix Naumann and Melanie Herschel. *An Introduction to Duplicate Detection*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010. DOI: 10.2200/S00262ED1V01Y201003DTM003.

[Pap+13] George Papadakis, Ekaterini Ioannou, Themis Palpanas, Claudia Niederee, and Wolfgang Nejdl. "A blocking framework for entity resolution in highly heterogeneous information spaces". In: *IEEE Transactions on Knowledge and Data Engineering* 25.12 (2013), pp. 2665–2682.

[Pap+14] George Papadakis, Georgia Koutrika, Themis Palpanas, and Wolfgang Nejdl. "Meta-blocking: Taking entity resolution to the next level". In: *IEEE Transactions on Knowledge and Data Engineering* 26.8 (2014), pp. 1946–1960.

[Pap+16] George Papadakis, George Papastefanatos, Themis Palpanas, and Manolis Koubarakis. "Scaling Entity Resolution to Large, Heterogeneous Data with Enhanced Meta-blocking." In: *EDBT*. 2016, pp. 221–232.

[Pap+18] George Papadakis, Leonidas Tsekouras, Emmanouil Thanos, George Giannakopoulos, Themis Palpanas, and Manolis Koubarakis. "The return of JedAI: End-to-End Entity Resolution for Structured and Semi-Structured Data". In: *PVLDB* 11.12 (2018), pp. 1950–1953. DOI: 10.14778/3229863.3236232.

[PHN15] Thorsten Papenbrock, Arvid Heise, and Felix Naumann. "Progressive Duplicate Detection". In: *IEEE Trans. Knowl. Data Eng.* 27.5 (2015), pp. 1316–1329. DOI: 10.1109/TKDE.2014.2359666.

[PPK14] George Papadakis, George Papastefanatos, and Georgia Koutrika. "Supervised Meta-blocking". In: *PVLDB* 7.14 (2014), pp. 1929–1940. DOI: 10.14778/2733085.2733098.

[RVC18] Thilina Ranbaduge, Dinusha Vatsalan, and Peter Christen. "A Scalable and Efficient Subgroup Blocking Scheme for Multi-database Record Linkage". In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2018, pp. 15–27.

[Sar+11] Anish Das Sarma, Ankur Jain, Ashwin Machanavajjhala, and Philip Bohannon. "CBLOCK: An Automatic Blocking Mechanism for Large-Scale De-duplication Tasks". In: *CoRR* abs/1111.3689 (2011). arXiv: 1111.3689. URL: http://arxiv.org/abs/1111.3689.

[SBJ16] Giovanni Simonini, Sonia Bergamaschi, and HV Jagadish. "BLAST: a loosely schema-aware meta-blocking approach for entity resolution". In: *Proceedings of the VLDB Endowment* 9.12 (2016), pp. 1173–1184.

[Sim+18b] Giovanni Simonini, George Papadakis, Themis Palpanas, and Sonia Bergamaschi. "Schema-agnostic Progressive Entity Resolution". In: *IEEE Trans. Knowl. Data Eng. (2018)* (2018). DOI: 10.1109/TKDE.2018.2852763.

[Sim+18c] Giovanni Simonini, George Papadakis, Themis Palpanas, and Sonia Bergamaschi. "Schema-Agnostic Progressive Entity Resolution". In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. 2018, pp. 53–64. DOI: 10.1109/ICDE.2018.00015.

[Spa] URL: https://spark.apache.org/docs/2.1.0/programming-guide.html#shuffle-operations.

[Sto+13] Michael Stonebraker, Daniel Bruckner, Ihab F. Ilyas, George Beskales, Mitch Cherniack, Stanley B. Zdonik, Alexander Pagan, and Shan Xu. "Data Curation at Scale: The Data Tamer System". In: *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. 2013.

[VCL10] Rares Vernica, Michael J Carey, and Chen Li. "Efficient parallel set-similarity joins using MapReduce". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 495–506.

[WMG13] Steven Euijong Whang, David Marmaros, and Hector Garcia-Molina. "Pay-As-You-Go Entity Resolution". In: *IEEE Trans. Knowl. Data Eng.* 25.5 (2013), pp. 1111–1124. DOI: 10.1109/TKDE.2012.43.

[Xia+11] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. "Efficient similarity joins for near-duplicate detection". In: *ACM Transactions on Database Systems (TODS)* 36.3 (2011), p. 15.

[XWL08] Chuan Xiao, Wei Wang, and Xuemin Lin. "Ed-join: an efficient algorithm for similarity joins with edit distance constraints". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 933–944.

[Zah+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28. ISBN: 978-931971-92-8.