Deep and Collective Entity Resolution in Parallel

Ting Deng[1] Wenfei Fan[1,2,3] Ping Lu[1] Xiaomeng Luo[1] Xiaoke Zhu[1] Wanhe An[1]
SKLSDE, Beihang University[1] University of Edinburgh[2] Shenzhen Institute of Computing Sciences[3]
wenfei@inf.ed.ac.uk, {dengting@act., luping@, luoxm@act., zhuxk@, anwh@act.}buaa.edu.cn

***Abstract*—This paper studies deep and collective entity resolution (ER). As opposed to a single pass of pairwise comparison of tuples in a single table, deep ER recursively identifies tuples that refer to the same entity by making use of matches in the previous rounds, and collective ER determines matches by correlating information across multiple tables. We propose a fixpoint model for deep and collective ER, by chasing with logic rules that are collectively defined across multiple relations and may embed machine learning classifiers for ER as predicates. While powerful, we show that deep and collective ER is intractable. To scale with large datasets, we develop a data partitioning strategy and a parallel algorithm underlying the fixpoint model, which guarantee to reduce runtime when more processors are used. Using real-life data, we experimentally verify that the approach improves the ER accuracy and is parallelly scalable.**

I. Introduction

Entity resolution (ER) has been a longstanding challenge. Also known as record matching, record linkage and duplicate detection, ER is to identify whether two tuples in a dataset refer to the same real-world entity. There has been a host of work on ER, based on either logic rules (*e.g.,* [30], [16], [10]) or machine learning (ML) models (*e.g.,* [50], [55], [9]).

ER is conceptually conducted on a single relation via a single pass of pairwise comparison of the tuples in the table, possibly with blocking [66], [49] and windowing [39] to speed up the process. However, there is still room to improve its accuracy. (1) We can often identify new matches by making use of matches deduced earlier. This suggests that we conduct ER recursively, referred to as *deep* ER, instead of inspecting tuple pairs only once. (2) While it has long been recognized that accurate ER needs to *collectively* correlate information across multiple tables [17], no data quality rules are in place to express collective ER, and even the complexity of collective ER is not yet settled. In particular, blocking and windowing no longer work for collective ER since they target a table of homogeneous tuples, while collective ER works on multiple (heterogeneous) tables. (3) Rule-based methods and ML models are often taken as separate approaches. Is it possible to benefit from both and improve the ER accuracy by unifying the two?

Example 1: E-commerce companies want to identify merchant accounts that conduct fraudulent behaviors, *e.g.,* account abuse [68] when two merchants boost sales by buying the same products from each other. As an example, consider four relations shown in Tables I–IV, for custermors, shops, products and orders, with the following schemas, respectively:

- Customers(cno, name, phone, addr, pref),
- Shops(sno, sname, owner, email, loc),
- Products(pno, pname, price, desc), and
- Order(ono, buyer, seller, item, IP).

The company finds that shops s_2 and s_4 buy the same product from each other, as follows: (1) customer c_4 (the owner of shop s_4, tuple t_9 in D_2) buys product p_2 from shop s_2 (t_{15} in D_4); (2) customer c_1 buys the same product p_2 from s_4 (t_{18} in D_4); and (3) c_1 is the owner of s_2 (by deduction).

However, the deduction is nontrivial. It needs several steps.

(1) The company finds that c_2 and c_3 are the same customer as they share the same name, phone and addr, by using a rule.

(2) It identifies products p_2 and p_3 in D_3, since they have the same name and semantically similar descriptions. Specifically, it employs an ML model to match p_2.desc and p_3.desc.

(3) It matches shops s_4 and s_5 in D_2 since they have the same email and similar names in D_2, and their owners c_4 and c_5 (tuples t_9 and t_{10} in D_2) have the same phone in D_1 (t_4 and t_5 in D_1). This step is *collectively* checked across D_2 and D_1.

(4) It identifies customers c_1 and c_3 in D_1 since they have the same address and similar names, and moreover, they buy product p_2 (*i.e.,* p_3) in shop s_4 (tuple t_9 in D_2) from the same IP address 113.55.126.9 (tuples t_{16} and t_{17} in D_4). This step is *"deep" ER* and makes use of the prior matches of shops s_4 and s_5 in D_2 (step 3) and products p_2 and p_3 in D_3 (step 2).

(5) Now it concludes that c_1 and c_2 are the same customer since both c_2 and c_1 match c_3 (steps 1 and 4); hence the fraudulence (c_1 owns s_2, which buys product p_2 from s_4). □

The example highlights the need for conducting deep and collective ER, and for using both logic rules and ML models.

Contributions & organization. This paper studies deep and collective ER, by unifying logic rules and ML models.

(1) Embedding ML in rules (Section II). Following [31], we extend matching dependencies (MDs [30], a class of ER rules) by (a) embedding ML classifiers for ER as predicates, and (b) supporting collective ER on multiple tables. We refer to the extended MDs as MRLs (Matching Rules with mL). We show how MRLs improve ER accuracy and interpret ML predictions.

(2) A fixpoint model (Section III). We model deep and collective ER as a form of the chase with a set Σ of MRLs [58]. The chase is Church-Rosser, *i.e.,* it converges at the same set of matches no matter what MRLs in Σ are used and in what order the rules are applied. We show that the improved ER accuracy comes at a price of increased complexity: while deep ER is in polynomial time (PTIME), collective ER becomes intractable in the absence of recursion. To scale with large datasets, we propose a fixpoint model for deep and collective ER, and parallelize the fixpoint computation in the following sections.

(3) Data partitioning (Section IV). To parallelize the fixpoint process, we develop a strategy to partition the data, in place of blocking [66], [49]. The partitioning strategy adapts the Hypercube algorithm of [6], [14] for parallel processing of conjunc-

2022 IEEE 38th International Conference on Data Engineering (ICDE) | 978-1-6654-0883-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDE53745.2022.00200

2375-026X/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDE53745.2022.00200

	cno	name	phone	addr	pref
t_1	c_1	Ford Smith	(213) 243-9856	1st Ave, LA	clothing, makeup
t_2	c_2	F. Smith	(213) 333-0001	1st Ave, LA	clothing
t_3	c_3	F. Smith	(213) 333-0001	1st Ave, LA	dress
t_4	c_4	Tony Brown	(347) 981-3452	9 Ave, NY	sports
t_5	c_5	T. Brown	(347) 981-3452	-	sports

TABLE I
AN INSTANCE D_1 OF SCHEMA Customers

	sno	sname	owner	email	loc
t_6	s_1	Comp. World	c_1	FSm@g.com	1st Ave, LA
t_7	s_2	Smith's Tech shop	c_2	F_Sm@g.com	1st Ave, LA
t_8	s_3	Lap. store	c_3	jp@youp.com	1st Ave, LA
t_9	s_4	T's Store	c_4	T.Brown@ga.com	9 Ave, NY
t_{10}	s_5	Tony's Store	c_5	T.Brown@ga.com	-

TABLE II
AN INSTANCE D_2 OF SCHEMA Shops

	pno	pname	price	desc
t_{11}	p_1	Apple MacBook	$1000	Apple MacBook Air (13-inch, 8GB RAM, 256GB SSD)
t_{12}	p_2	ThinkPad	$2000	ThinkPad X1 Carbon 7th Gen : 14-Inch, 16GB RAM, 512GB Nvme SSD
t_{13}	p_3	ThinkPad	$1800	ThinkPad X1 Carbon 7th Gen 14" - 16 GB RAM - 512 GB SSD
t_{14}	p_4	Acer Laptop	$500	Acer Aspire 5 Slim Laptop, 15.6 inches, 4GB DDR4, 128GB SSD, Backlit Keyboard

TABLE III
AN INSTANCE D_3 OF SCHEMA Products

	ono	buyer	seller	item	IP
t_{15}	o_1	c_4	s_2	p_2	156.33.14.7
t_{16}	o_2	c_3	s_4	p_2	113.55.126.9
t_{17}	o_3	c_1	s_5	p_3	113.55.126.9
t_{18}	o_4	c_1	s_4	p_2	143.32.11.2

TABLE IV
AN INSTANCE D_4 OF SCHEMA Orders

tive queries in one round of communication. We show that it is NP-complete to partition the data with the minimum cost. This said, we provide an effective heuristic partitioning algorithm.

(4) Parallel algorithm (Section V). On the partitioned data, we develop a parallel algorithm, denoted by DMatch, to support the fixpoint process for deep and collective ER. We propose a strategy to reduce repeated checking, and implement DMatch based on partial evaluation and incremental computation, to reduce both computation cost and communication cost. We show that DMatch is parallelly scalable [47], *i.e.,* it guarantees to reduce runtime when more processors are used.

(5) Experimental study (Section VI). Using real-life and synthetic data, we empirically find the following. (a) By supporting deep and collective ER and by unifying ML and logic, DMatch is 23% and 38% more accurate than ML models and logical methods for ER, respectively; it outperforms deep ER and collective ER by 21% and 32%, respectively. Its F-measure is 0.95 on average. (b) DMatch scales well with large datasets. It takes 505s on datasets of 30M tuples using 16 machines. It is even faster than 7 out of 8 state-of-the-art ER baselines. (c) DMatch is parallelly scalable with the number n of processors used. When n varies from 4 to 32, DMatch is 3.56 times faster.

Related work. We categorize the related work as follows.

Entity resolution. There has been a host of work on ER, classified as follows: (1) ML-based, *e.g.,* deep learning [25], [50], [67], [48], active learning [9], [55], and transfer learning [43], (2) Rule-based, *e.g.,* uniqueness constraints [38], matching dependencies (MDs) [30], [29], [16], [11], [46], and datalog-like rules [10], [65]; and (3) hybrid, *e.g.,* [12], by employing MDs as blocking keys and ML for classification. JedAI [53] is a toolkit that combines various state-of-the-art ER methods. Collective ER was proposed in [17], [10]; [17] also advocated "recursive" ER by leveraging entity co-occurrence and aggregating similarity scores from neighboring entities.

ER is conceptually conducted as a single pass of pairwise comparisons of tuples in a single table, in quadratic-time [28]. To speed it up, windowing [39] and blocking [52], [51] have been commonly used. Windowing first sorts the tuples in a table; it then employs a sliding window and compares only tuples in the same window. Blocking first clusters similar entities into "disjoint" blocks via blocking keys, and then conducts pairwise comparisons only within each block.

This work differs from the prior work as follows. (1) It makes a first attempt to formalize deep ER and collective ER in a uniform logical framework in terms of logic rules with embedded ML predicates, beyond similarity score aggregation. (2) It settles the complexity of deep ER and collective ER. (3) To support deep and collective ER, it develops a fixpoint computation model and parallelly scalable algorithms.

Closer to this work are [31] and [32]. [31] proposes a class of entity enhancing rules (REEs) that may embed ML classifiers as predicates. Using REEs, [32] develops a parallel algorithm for error detection (entity resolution and conflict resolution). MRLs studied in this paper are a special case of REEs, and the correctness of our chase (Section III) follows from the Church-Rosser property verified in [31].

In contrast to [31], [32], (1) we focus on ER by extending MDs with ML predicates, so that we can develop efficient algorithms for ER (Section V); (2) we settle the complexity of deep ER and collective ER, which is not studied in [31]; and (3) we study deep ER with a fixpoint model, to make new matches by using matches in the prior rounds, while [32] only detects violations of REEs in a single pass, without recursion.

Parallel ER. Parallel ER algorithms have been studied under MapReduce [45], [56], [22], [35], [26], [61], [8], [27] or MPC [62]. In particular, a blocking strategy under MapReduce was developed [22] by revising the HypeCube algorithm [13].

In contrast to the prior work, (1) we provide the first parallel algorithm for deep and collective ER, beyond conventional ER on a single table. (2) We propose a fixpoint model for parallel ER; as opposed to MapReduce that shuffles data between mappers and reducers, no raw data is sent between different workers during the fixpoint computation, and hence incurs less communication cost. (3) We develop a data partitioning algorithm in place of blocking. It extends the HyperCube algorithm of [13] to handle a set of MRLs, for deep and collective ER across multiple tables rather than on a single table [22]. (4) To the best of our knowledge, we provide the first (deep) ER algorithm with provable parallel scalability.

II. Embedding ML Models in Rules

In this section, we define MRLs, an extension of matching dependencies (MDs [30]) with embedded ML predicates.

Datasets. Consider a database schema $\mathcal{R} = (R_1, \ldots, R_m)$, where each R_i is a relation schema $(A_1 : \tau_1, \ldots, A_n : \tau_n)$, and each A_i is an attribute of type (domain) τ_i. A dataset $\mathcal{D}$ of $\mathcal{R}$ is $(D_1, \ldots, D_m)$, where D_i is a relation of R_i for $i \in [1, m]$. In particular, we assume *w.l.o.g.* a designated attribute id for each R_i, such that a tuple of R_i represents an entity with identity id.

Predicates. *Predicates* over $\mathcal{R}$ are defined as follows:

$$p ::= R(t) \mid t.A = c \mid t.A = s.B \mid \mathcal{M}(t[\bar{A}], s[\bar{B}]).$$

Here following tuple relational calculus [5], (1) $R(t)$ is a *relation atom* over $\mathcal{R}$, where $R \in \mathcal{R}$, and t is a tuple variable *bounded by* $R(t)$. (2) When t is bounded by $R(t)$ and A is an attribute of R, $t.A$ denotes the A-attribute of t. (3) In $t.A = c$, c is a constant in the domain of attribute A in R. (4) In $t.A = s.B$, $t.A$ and $s.B$ are *compatible*, *i.e.,* $R(t)$ and $R'(s)$ are specified, and $A \in R$ and $B \in R'$ have the same type. In particular, $t.\mathsf{id} = s.\mathsf{id}$, referred to as id *predicate*, denotes that the entities represented by t and s match. Moreover, (5) $\mathcal{M}$ is an ML classifier for ER, where $\bar{A}$ and $\bar{B}$ are vectors of pairwise compatible attributes, and $t[\bar{A}]$ and $s[\bar{B}]$ are vectors of the values of attributes $\bar{A}$ and $\bar{B}$ in tuples t and s, respectively. That is, ML predicates are applied to vectors of multiple attributes, *e.g.,* DeepER [25] can be even applied to entire tuples taken as vectors.

Intuitively, $\mathcal{M}(t[\bar{A}], s[\bar{B}])$ can be any well-trained ML classifier for ER, *e.g.,* [9], [55], [50], [25], [12], [40], [43], [41], [42], or for semantic similarity checking, *e.g.,* [54], [15], [64]. We refer to such $\mathcal{M}$ as an *ML predicate*, which returns true if it predicts that $t[\bar{A}]$ and $s[\bar{B}]$ "match", and false otherwise.

Observe the following. (1) ML predicates can take more than two relations. This said, since most ML predicates in practice use only two relations (*e.g.,* DeepER [25], ERBlox [12], fasttext [19]), we focus on binary ML classifiers to simplify the presentation. (2) One can take a probabilistic model as an ML predicate as follows: it returns true if the probability is above a predefined threshold and returns false otherwise. (3) ML classifiers are able to check semantic similarity, *e.g.,* "US" and "America"; moreover, ML predicates include classifiers for NLP (*e.g.,* NER [34]), ER (*e.g.,* ditto [48] and DeepER [25]) and conflict detection (*e.g.,* HoloClean [57]).

Matching rules. We define MRLs φ over $\mathcal{R}$ as

$$X \rightarrow l.$$

Here (1) X is a conjunction of predicates over $\mathcal{R}$, and (2) l is a predicate of the form either $t.\mathsf{id} = s.\mathsf{id}$ or $\mathcal{M}(t[\bar{A}], s[\bar{B}])$, where t and s are tuple variables bounded in X. We refer to X and l as the *precondition* and *consequence* of φ, respectively.

Note that MDs [30] can be expressed as MRLs $X \rightarrow l$ in which X consists of two relation atoms $R_1(t_1)$ and $R_2(t_2)$, equality atoms $t.A = s.B$, $\mathcal{M}(t_1[\bar{A}_1], t_2[\bar{A}_2])$ that simulates similarity checking, and l is $t_1.\mathsf{id} = t_2.\mathsf{id}$. MRLs extend MDs by supporting (a) ML predicates $\mathcal{M}(t[\bar{A}], s[\bar{B}])$, (b) constant predicates $t.A = c$, and (c) collective ER across multiple relations, while MDs are defined on at most two relations.

Example 2: Over the dataset of Example 1 we define MRLs.

(1) φ_1: $\mathsf{Customers}(t_c) \wedge \mathsf{Customers}(t'_c) \wedge t_c.\mathsf{name} = t'_c.\mathsf{name} \wedge t_c.\mathsf{phone} = t'_c.\mathsf{phone} \wedge t_c.\mathsf{addr} = t'_c.\mathsf{addr} \rightarrow t_c.\mathsf{id} = t'_c.\mathsf{id}$. The MRL says that if customers t_c and t'_c share the same name, phone and address, then they refer to the same person.

(2) φ_2: $\mathsf{Products}(t_p) \wedge \mathsf{Products}(t'_p) \wedge t_p.\mathsf{pname} = t'_p.\mathsf{pname} \wedge \mathcal{M}_1(t_p.\mathsf{desc}, t'_p.\mathsf{desc}) \rightarrow t_p.\mathsf{id} = t'_p.\mathsf{id}$. It matches two products if they have the same name and similar descriptions. It uses ML model $\mathcal{M}_1$ to check the similarity of long text descriptions.

(3) φ_3: $\mathsf{Customers}(t_c) \wedge \mathsf{Customers}(t'_c) \wedge \mathsf{Shops}(t_s) \wedge \mathsf{Shops}(t'_s) \wedge \mathcal{M}_2(t_s.\mathsf{name}, t'_s.\mathsf{name}) \wedge t_s.\mathsf{email} = t'_s.\mathsf{email} \wedge t_s.\mathsf{owner} = t_c.\mathsf{cno} \wedge t'_s.\mathsf{owner} = t'_c.\mathsf{cno} \wedge t_c.\mathsf{phone} = t'_c.\mathsf{phone} \wedge \rightarrow t_s.\mathsf{id} = t'_s.\mathsf{id}$. It matches two shops if they have the same email and similar names, and if their owners have the same phone number.

(4) φ_4: $\mathsf{Customers}(t_c) \wedge \mathsf{Customers}(t'_c) \wedge \mathsf{Orders}(t_o) \wedge \mathsf{Orders}(t'_o) \wedge \mathsf{Products}(t_p) \wedge \mathsf{Products}(t'_p) \wedge \mathsf{Shops}(t_s) \wedge \mathsf{Shops}(t'_s) \wedge t_c.\mathsf{cno} = t_o.\mathsf{buyer} \wedge t'_c.\mathsf{cno} = t'_o.\mathsf{buyer} \wedge t_o.\mathsf{item} = t_p.\mathsf{pno} \wedge t'_o.\mathsf{item} = t'_p.\mathsf{pno} \wedge t_o.\mathsf{seller} = t_s.\mathsf{sno} \wedge t'_o.\mathsf{seller} = t'_s.\mathsf{sno} \wedge \mathcal{M}_3(t_c.\mathsf{name}, t'_c.\mathsf{name}) \wedge t_c.\mathsf{addr} = t'_c.\mathsf{addr} \wedge t_o.\mathsf{IP} = t'_o.\mathsf{IP} \wedge t_p.\mathsf{id} = t'_p.\mathsf{id} \wedge t_s.\mathsf{id} = t'_s.\mathsf{id} \rightarrow t_c.\mathsf{id} = t'_c.\mathsf{id}$. The MRL identifies two customers if they have the same address and similar names, and moreover, if they buy the same product from the same shop at the same IP address.

(5) φ_5: $\mathsf{Customer}(t_c) \wedge \mathsf{Customer}(t'_c) \wedge \mathsf{Orders}(t_o) \wedge \mathsf{Orders}(t'_o) \wedge t_c.\mathsf{cno} = t_o.\mathsf{buyer} \wedge t'_c.\mathsf{cno} = t'_o.\mathsf{buyer} \wedge t_o.\mathsf{item} = t'_o.\mathsf{item} \rightarrow \mathcal{M}_4(t_c.\mathsf{pref}, t'_c.\mathsf{pref})$. It interprets ML prediction $\mathcal{M}_4(t_c.\mathsf{pref}, t'_c.\mathsf{pref})$ with logic characteristics (see below). □

Remark. In an MRL $X \rightarrow \mathcal{M}(t[\bar{A}], s[\bar{B}])$, X provides a logic "explanation" of ML predictions: it is because logic conditions X hold that $\mathcal{M}$ predicts true on $(t[\bar{A}], s[\bar{B}])$. For example, φ_5 says that $\mathcal{M}_4$ predicts customers t_c and t'_c to have similar preferences because the two have bought the same product.

Semantics. Consider a dataset $\mathcal{D}$ of schema $\mathcal{R}$. A *valuation* h of tuple variables of φ in $\mathcal{D}$, or simply *a valuation of* φ, is a mapping that instantiates t in each relation atom $R(t)$ of φ with a tuple in the relation of R in $\mathcal{D}$.

We say that h *satisfies* a predicate p, written as $h \models p$, if the following conditions are satisfied. (1) If p is $R(t)$, $t.A = c$ or $t.A = s.B$, then $h \models p$ is interpreted as in tuple relational calculus following the standard semantics of first-order logic [5]. (2) If p is $\mathcal{M}(t[\bar{A}], s[\bar{B}])$, then $h \models p$ if the ML classifier $\mathcal{M}$ predicts true when provided with $(h(t)[\bar{A}], h(s)[\bar{B}])$.

For a conjunction X of predicates over $\mathcal{R}$, we write $h \models X$ if $h \models p$ for *all* predicates p in X. A dataset $\mathcal{D}$ of $\mathcal{R}$ *satisfies* an MRL φ, denoted by $\mathcal{D} \models \varphi$, if *for all* valuations h of φ in $\mathcal{D}$, if $h \models X$, then $h \models l$. We say that $\mathcal{D}$ *satisfies* a set Σ of MRLs, denoted by $\mathcal{D} \models \Sigma$, if for all $\varphi \in \Sigma$, $\mathcal{D} \models \varphi$.

III. A Fixpoint Model for Deep and Collective ER

Below we first present deep and collective ER, and settle their complexity (Section III-A). We then propose a fixpoint model for deep and collective ER in parallel (Section III-B).

A. Deep and Collective Entity Resolution

Deep and collective ER. Consider a database schema $\mathcal{R}$, a dataset $\mathcal{D}$ of $\mathcal{R}$, and a set Σ of MRLs. Deep and collective ER deduces a set Γ of "matches" and validated ML predictions by applying a rule in Σ as follows. Initially, Γ consists of pairs $(t.\mathsf{id}, t.\mathsf{id})$ for all tuples t in $\mathcal{D}$. For each MRL $\varphi = X \rightarrow l \in \Sigma$ and valuation h of φ in $\mathcal{D}$, when $h \models X$,

- if l is $t.\mathsf{id}{=}s.\mathsf{id}$, then add $(h(t).\mathsf{id}, h(s).\mathsf{id})$ to Γ, referred to as a *match*; *i.e.,* the two denote the same entity; and
- if l is $\mathcal{M}(t[\bar{A}], s[\bar{B}])$, then mark $\mathcal{M}(h(t)[\bar{A}], h(s)[\bar{B}])$ as a *validated prediction*, and add it to Γ.

We verify that $h \models p$ for predicates $p \in X$

- if p is $t.\mathsf{id} = s.\mathsf{id}$ and $(h(t).\mathsf{id}, h(s).\mathsf{id})$ is in Γ; or
- if p is $\mathcal{M}(t[\bar{A}], s[\bar{B}])$ and is validated in earlier steps.

The process proceeds until Γ cannot be further extended.

Observe the following about the process.

(1) Formally, the process can be modeled as an extension of the chase [58] with MRLs [31]. Following [31], one can verify that the chase has the Church-Rosser property [5], *i.e.,* it always converges at the same set Γ of matches no matter what MRLs in Σ are used and in what order they are applied. In particular, MRLs may embed all the ML models listed in Section II while the extended chase still retains the Church-Rosser property.

Corollary 1: *Deep and Collective ER is Church-Rosser.* □

As a result, deep and collective ER and Γ are well defined. We refer to Γ as *the set of matches* deduced by Σ in $\mathcal{D}$. We say that tuples t_1 and t_2 in $\mathcal{D}$ are matched by Σ, denoted by $(\mathcal{D}, \Sigma) \models (t_1.\mathsf{id}, t_2.\mathsf{id})$, if $(t_1.\mathsf{id}, t_2.\mathsf{id}) \in \Gamma$.

(2) Deep and collective ER is a recursive process. It employs matches of Γ found in earlier steps, as opposed to conventional ER with blocking and windowing. Moreover, a valuation may span across multiple relations, as opposed to MDs and other quality rules that are defined on at most two relations.

(3) It is not possible in principle to express recursion with joins; indeed, recursion is not expressible in first-order logic (and join in relational algebra); in practice it is not known in advance how many levels a recursive computation needs to go.

Complexity. We study the *deep and collective ER problem*.

- *Input*: A database schema $\mathcal{R}$, a set Σ of MRLs over $\mathcal{R}$, a dataset $\mathcal{D}$ of $\mathcal{R}$ and two tuples t and s in $\mathcal{D}$.
- *Question*: Does $(\mathcal{D}, \Sigma) \models (t.\mathsf{id}, s.\mathsf{id})$?

We also study two of its special cases: (1) *collective* ER when MRLs in Σ may be defined with an unbounded number of relations (collective) but do not carry id predicates in their preconditions (not deep); and (2) *deep* ER when all MRLs in Σ are defined with *a fixed constant* of relation atoms (not collective)) but may have id predicates in their preconditions (deep).

Theorem 2: *(1) Collective ER is* NP*-complete. (2) Deep ER is in* PTIME*. (3) Deep and collective ER is* NP*-complete.* □

That is, the deep and collective ER problem is already intractable even without recursion. Here we assume *w.l.o.g.* that testing ML predictions with pretrained ML models in MRLs is in PTIME, as commonly found in practice.

Proof sketch: We show that (1) deep and collective ER is in NP; (2) collective ER is NP-hard; (3) deep ER is in PTIME.

(1) We define a notion of proof graphs and show a small model property: $(\mathcal{D}, \Sigma) \models (t.\mathsf{id}, s.\mathsf{id})$ iff there is a proof graph of size at most $\|\Sigma\|(|\Sigma|{+}1)|\mathcal{D}|^2$ to encode $(\mathcal{D}, \Sigma) \models (t.\mathsf{id}, s.\mathsf{id})$. Here $\|\Sigma\|$ (resp. $|D|$) is the number of rules in Σ (resp. tuples in $|\mathcal{D}|$), and $|\Sigma|$ refers to the maximum number of tuple variables of rules in Σ. Then we give an NP algorithm: guess a graph $\mathcal{T}$ with at most $O(\|\Sigma\|(|\Sigma|{+}1)|\mathcal{D}|^2)$ nodes, and check whether $\mathcal{T}$ is a proof graph of $(\mathcal{D}, \Sigma) \models (t.\mathsf{id}, s.\mathsf{id})$, in PTIME.

(2) We show that collective ER is NP-hard by reduction from the Boolean conjunctive query evaluation problem, which is NP-complete [20]. It is to decide, given a Boolean conjunctive query Q and a dataset $\mathcal{D}'$, whether $Q(\mathcal{D}') = \mathsf{true}$. The construction uses no id predicate in the preconditions of MRLs.

(3) For deep ER, we develop an algorithm to check whether $(t.\mathsf{id}, s.\mathsf{id})$ can be deduced by Σ in $\mathcal{D}$. It is in PTIME since (a) there exist at most $O(\|\Sigma\|(|\Sigma|{+}1)|\mathcal{D}|^2)$ matches and validated ML predictions in Γ, (b) each MRL in Σ has at most k relation atoms for a constant k, and (c) for each predicate l, it is in PTIME to check whether l can be added to Γ since the number of all valuations of MRLs in $\mathcal{D}$ is bounded by $O(\|\Sigma\||\mathcal{D}|^k)$. □

We identify another tractable case of the problem. An MRL $\varphi = X \rightarrow l$ is *acyclic* if the hypergraph of X is acyclic. Here the hypergraph of X is defined by treating the attributes in X as vertices, and taking tuples in X as hyperedges.

Theorem 3: *The deep and collective ER problem is in* PTIME *with acyclic* MRLs. □

Proof sketch: We show that (†) given a dataset $\mathcal{D}$ and an acyclic MRL φ, it is in PTIME to check whether there exists a valuation of φ in $\mathcal{D}$ that does not satisfy φ by extending the proofs of [37], [21]. Then we compute the set Γ of matches of φ in $\mathcal{D}$ as follows: for each MRL $\varphi{=}X{\rightarrow}l$ in Σ, if there exists a valuation h of φ in $\mathcal{D}$ that does no satisfy φ (*i.e.,* $h \models X$ but $h \not\models l$), then add $h(l)$ to Γ. It takes PTIME by (†) above since there exist most $O((\|\Sigma\||\Sigma| + 1)|\mathcal{D}|^2)$ matches in Γ. □

Remark. (1) Deep ER is in PTIME for any fixed constant k of tuple variables, regardless of the specific value of k. When k is bounded, so is the number of tables involved. (2) The intractability of collective ER is introduced by the number of tuple variables. Collective ER (not deep) differs from deep ER in that (a) MRLs do not allow id predicates in their preconditions and hence, do not support recursive checking; but (b) it allows unbounded k and hence, an unbounded number of tables.

B. A Fixpoint Model

The intractability suggests that we develop a parallel model for deep and collective ER. The model supports data partitioned parallelism by using n workers (processors) $P_1, \ldots, P_n$ and a master P_0. Consider a dataset $\mathcal{D}$ of $\mathcal{R}$ and a set Σ of MRLs. We partition $\mathcal{D}$ and distribute the data across n workers.

Under the Bulk Synchronous Parallel (BSP) model [63], parallel deep and collective ER is conducted in supersteps, modeled as the following simultaneous fixpoint computation:

$$\Gamma_i^0 = \mathcal{A}(\Sigma, W_i) \quad (1)$$

$$\Gamma_i^{r+1} = \mathcal{A}_\Delta(\Sigma, W_i, \Delta\Gamma_i^r) \quad (2)$$

For $i \in [1, n]$, Γ_i^r is the set of local matches deduced at worker P_i in step r, W_i is the fragment of $\mathcal{D}$ allocated to worker P_i by the partition, and $\Delta\Gamma_i^r$ is the set of matches that are deduced at other workers and are passed to worker P_i in step r as messages. As will be seen in Section V, $\mathcal{A}$ is a sequential algorithm for deep and collective ER that operates on dataset W_i with MRLs in Σ, and $\mathcal{A}_\Delta$ is a sequential algorithm that incrementally deduce matches in response to *updates* $\Delta\Gamma_i^r$.

More specifically, the parallel deduction works as follows.

(1) In the first superstep, each worker P_i deduces its local matches Γ_i^0 by algorithm $\mathcal{A}$, by chasing its local data W_i with MRLs in Σ. As will be seen in Section V, this is a recursive process itself. All n workers execute algorithm $\mathcal{A}$ in parallel. At the end of the step, P_i sends Γ_i^0 to master P_0, which routes the new matches to relevant workers to $\Delta\Gamma_i^0$, *i.e.,* it sends $(t.\mathsf{id}, s.\mathsf{id})$ and $\mathcal{M}(t[\bar{A}], s[\bar{B}])$ to workers that also host t or s.

(2) Upon receiving $\Delta\Gamma_i^r$, worker P_i incrementally deduces its local matches Γ_i^{r+1} by executing $\mathcal{A}_\Delta$ on W_i, which treats $\Delta\Gamma_i^r$ as updates, and references matches in $\Gamma_i^r \cup \Delta\Gamma_i^r$ in the chase (see Section V). At the end of the step, P_i sends *newly deduced matches* in this step to master P_0, and P_0 routes the changes to relevant workers via messages as in step (1) above.

(3) Step (2) above iterates until no more changes can be made, *i.e.,* $\Delta\Gamma_i^r = \emptyset$. At this point, the process returns the union $\Gamma = \bigcup_{i\in[1,n]} \Gamma_i^r$. It is a fixpoint $\bigcup_{i\in[1,n]} \Gamma_i^{r+1} = \bigcup_{i\in[1,n]} \Gamma_i^r$.

Example 3: Continuing with Example 2, assume that the dataset $\mathcal{D}$ is partitioned into two fragments W_1 and W_2, where W_1=$\{t_1, t_2, t_3, t_4, t_9, t_{10}, t_{12}, t_{13}, t_{15}, t_{16}, t_{17}, t_{18}\}$ and $W_2 = \{t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{14}, t_{17}\}$ (see Example 5 for the partition). Fragments W_1 and W_2 are assigned to workers P_1 and P_2, respectively. Given a set Σ consisting of $\varphi_1 - \varphi_5$ presented in Example 2, the fixpoint model works as follows.

(1) In the first superstep, P_1 deduces local matches $\Gamma_1^0 = \{(t_2.\mathsf{id}, t_3.\mathsf{id}), (t_{12}.\mathsf{id}, t_{13}.\mathsf{id})\} \cup \Gamma_M^0$ by algorithm $\mathcal{A}$ using rules φ_1, φ_2, φ_3 and φ_5 to W_1, where $\Gamma_M^0 = \{\mathcal{M}_4(t_1.\mathsf{prf}, t_3.\mathsf{prf}), \mathcal{M}_4(t_1.\mathsf{prf}, t_4.\mathsf{prf}), \mathcal{M}_4(t_3.\mathsf{prf}, t_4.\mathsf{prf})\}$ is the set of validated ML predications; similarly P_2 deduces Γ_2^0=$\{(t_4.\mathsf{id}, t_5.\mathsf{id}), (t_9.\mathsf{id}, t_{10}.\mathsf{id})\}$ by applying φ_2 and φ_4 to W_2. Then P_1 and P_2 send Γ_1^0 and Γ_2^0 to P_0, respectively; P_0 generates $\Delta\Gamma_1^0$=$\{(t_9.\mathsf{id}, t_{10}.\mathsf{id})\}$ and sends it to P_1.

(2) Once receiving $\Delta\Gamma_1^0$ from master P_0, P_1 further identifies $(t_1.\mathsf{id}, t_3.\mathsf{id})$ by running algorithm $\mathcal{A}_\Delta$ on W_1 using matches in $\Gamma_1^0 \cup \Delta\Gamma_1^0$ and rule φ_4. Then the local matches of P_1 is updated to $\Gamma_1^1 = \{(t_2.\mathsf{id}, t_3.\mathsf{id}), (t_{12}.\mathsf{id}, t_{13}.\mathsf{id}), (t_1.\mathsf{id}, t_3.\mathsf{id})\} \cup \Gamma_M^0$.

(3) At this point, no new match can be exchanged between fragments. Then the process returns the union of Γ_1^1 and Γ_2^0, *i.e.,* $\Gamma = \Gamma_1^1 \cup \Gamma_2^0 = \{(t_1.\mathsf{id}, t_3.\mathsf{id}), (t_2.\mathsf{id}, t_3.\mathsf{id}), (t_4.\mathsf{id}, t_5.\mathsf{id}), (t_9.\mathsf{id}, t_{10}.\mathsf{id}), (t_{12}.\mathsf{id}, t_{13}.\mathsf{id})\} \cup \Gamma_M^0$ as the result. □

The fixpoint model warrants the correctness of collective ER.

Proposition 4: *Given any dataset $\mathcal{D}$ of schema $\mathcal{R}$ and any Σ of* MRLs *over $\mathcal{R}$, deep and collective ER converges at the set of matches deduced by Σ in $\mathcal{D}$ under the fixpoint model.* □

Proof sketch: We show this by induction on the number of supersteps, using the Church-Rosser property (Corollary 1). □

Remark. The fixpoint model parallelizes sequential algorithms $\mathcal{A}$ and $\mathcal{A}_\Delta$ along the same lines as the GRAPE model for graphs [33]. It differs from GRAPE as follows. GRAPE targets graph computations and exchanges messages between vertex neighbors along edges across different fragments. For relational data, in contrast, there exist no tuples that "connect" different fragments and there is *no* notion of "neighbors".

IV. Data Partitioning for Collective ER

In this section, we develop a data partitioning algorithm HyPart for deep and collective ER, to reduce both communication cost and computation cost. The algorithm extends Hypercube (HC) [6] and combines it with multiple query optimization (MQO) [44]. The objective is to use partitioning in place of blocking for collective rules across multiple relations.

Review. We start with a review of HC [6] and MQO [44].

Hypercube. Given a dataset $\mathcal{D}$ and a conjunctive query (CQ) Q that computes multiway natural joins in $\mathcal{D}$, HC partitions $\mathcal{D}$ into n fragments $W_1, \ldots, W_n$, such that Q can be answered locally, *i.e.,* $Q(\mathcal{D}) = \cup_{i\in[1,n]} Q(W_i)$. We use *distinct variables* $x_1.A_1, \ldots, x_l.A_l$ of Q to denote attributes appearing in the predicates of Q, *e.g.,* $x.A$=c or $x.A$=$y.B$, such that $x_i.A_i$= $x_j.A_j$ cannot be deduced from predicates in Q if $i \neq j$.

Given a CQ query Q, HC works in three steps. (1) It first organizes n workers $P_i (i\in[1, n])$ into an l-dimensional hypercube $\mathcal{H}$=$[n_1]\times\ldots\times[n_l]$, where l is the number of distinct variables in Q, n=$n_1\cdot\ldots\cdot n_l$, and each n_i is determined via Lagrangean multipliers [6]. (2) It then independently chooses l hash functions h_i $(i\in[1, l])$, one for each distinct attribute $x_i.A_i$. (3) For each tuple $t_\mathcal{D}$ of relation schema R_r in $\mathcal{D}$ and each tuple variable t_Q of R_r in Q, it sends $t_\mathcal{D}$ to worker P_i identified by both attributes in t_Q and hash functions; specifically, let $x_{i_1}.A_{i_1}, \ldots, x_{i_k}.A_{i_k}$ be all distinct attributes in t_Q; then $t_\mathcal{D}$ is sent to the worker located at $(p_1, \ldots, p_l)$, where p_{i_j}=$h_{i_j}(t_\mathcal{D}.A_{i_j})(j\in[1, k])$; for an attribute $x_T.A_T$ that is not in t_Q, we set p_T=$*$, indicating that the tuple is sent to all workers with $p_T\in[1, n_T]$. Denote by $t_\mathcal{D}^{t_Q}$ the tuple generated from $t_\mathcal{D}$ by replacing $t_\mathcal{D}.A_{i_j}$ with $h_{i_j}(t_\mathcal{D}.A_{i_j})$ for each $j\in[1, k]$.

Remark. There are connections between CQ and MRLs.

(1) For an ML classifier $\mathcal{M}(t[\bar{A}], s[\bar{B}])$ in MRLs, we can treat $t[\bar{A}]$ and $s[\bar{B}]$ as distinct variables since $\mathcal{M}(t[\bar{A}], s[\bar{B}])$ can only be computed by comparing all pairs of tuples. Similarly, we also treat id attributes as distinct variables. Here we slightly extend the definition of distinct variables of [6], since MRLs contain ML predicates that specify associations between attributes, which are not considered in the traditional Hypercube.

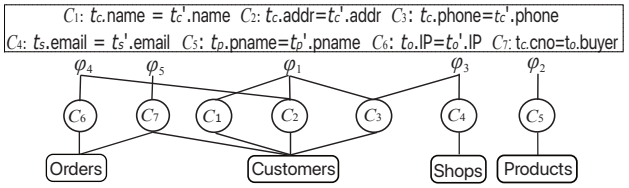

Fig. 1. An example of MQO query plan

(2) Given a set Σ of MRLs and a dataset $\mathcal{D}$, one might want to partition $\mathcal{D}$ by applying HC to $\mathcal{D}$ for each MRL. However, this requires multiple accesses to dataset $\mathcal{D}$, one for each MRL, and incurs heavy cost when $\mathcal{D}$ is large.

It is actually intractable to minimize the hash function computation. We formulate the problem as follows. Given an MRL $\varphi{=}X{\rightarrow}l$, assume that X involves tuple variables $t_1, \ldots, t_L$. Given $\mathcal{D}$, we denote the set of generated tuples via HC for φ as $\mathcal{E}_\varphi{=}\cup_{j\in[1,L]}\mathcal{P}_\varphi(t_j,\mathcal{D})$, where $\mathcal{P}_\varphi(t_j,\mathcal{D})$ consists of all generated tuples $t_{\mathcal{D}}^{t_j}$ (see above) for each tuple $t_{\mathcal{D}}$ in $\mathcal{D}$, which has the same relation schema as t_j. Denote by $\mathcal{H}(\Sigma,\mathcal{D}) = \cup_{\varphi\in\Sigma}\mathcal{E}_\varphi$ the set of all generated tuples to process $\mathcal{D}$ using Σ.

The *minimum hashing function problem* (MHFP) is:

- Input: A dataset $\mathcal{D}$, a set Σ of MRLs and a threshold k.
- Question: Is there HC for Σ such that $|\mathcal{H}(\Sigma,\mathcal{D})|{\leq}k$?

Theorem 5: *MHFP is* NP*-complete.* □

Proof sketch: The problem is in NP, since we can guess a hash function assignment for each rule φ, and check whether $|\mathcal{H}(\Sigma,\mathcal{D})|{=}|\cup_{\varphi\in\Sigma}\mathcal{E}_\varphi|{\leq}k$. The lower bound is verified by reduction from the subgraph isomorphism problem (cf. [36], which is to decide, given two undirected graphs G_1 and G_2, whether G_2 has a subgraph that is isomorphic to G_1. □

MQO. To reduce the hash function computation, we adopt the MQO technique (see *e.g.,* [44]). Given multiple CQ queries $Q_1, \ldots, Q_k$ on a dataset $\mathcal{D}$, MQO is to generate a query plan and compute all answers $Q_1(\mathcal{D}), \ldots, Q_k(\mathcal{D})$ such that the intermediate results can be shared as much as possible; here a query plan is a DAG (Direct Acyclic Graph) consisting of data access and natural joins operators that are used to answer the CQ queries. Intuitively, MQO decomposes $Q_1, \ldots, Q_k$ into smaller subqueries, finds common subqueries, and constructs a query plan QP for these queries (see Fig. 1 for an example).

Combining HC *and* MQO. We adopt MQO to improve the performance of HC as follows. We first construct a query plan QP for MRLs in Σ using the MQO technique. We then assign hash functions based on QP such that different rules with common subqueries share the same hash functions.

This is, however, nontrivial. (1) We need a strategy to assign the hash functions such that their computations can be reused as much as possible. (2) For different rules, a tuple with the same hash functions may not be sent to the same worker, since (a) workers are organized as a hypercube, and (b) the same hash functions in different rules may correspond to different dimensions of the hypercube and hence different positions (*i.e.,* workers). This incurs redundant communication cost.

To tackle these, we introduce three orderings.

(1) We impose an order O_r on the rules, and apply HC to the rules in Σ following O_r. Intuitively, the rules ranked higher share more relation atoms and hence more hash functions.

Algorithm Partition
Input: A dataset $\mathcal{D}$ and a set Σ of MRLs.
Output: A set of fragments $W_1, \ldots, W_n$.
1. QP := QPforMQO($Q_1, \ldots, Q_n$);
2. StackQuery := SortQuery($Q_1, \ldots, Q_n$);
3. **while** StackQuery $\neq \emptyset$ **do**
4. pop the top query Q_c off the stack StackQuery;
5. QP := AssignHash(QP, Q_c);
6. $W_1 := \emptyset; \ldots; W_n := \emptyset$;
7. **for each** query tuple t_Q in QP **do**
8. $(\Delta W_1, \ldots, \Delta W_n)$:= HyperCube($\mathcal{D}, t_Q$);
9. $W_1 := W_1 \cup \Delta W_1; \ldots; W_n := W_n \cup \Delta W_n$;
10. **return** $(W_1, \ldots, W_n)$.

Fig. 2. Algorithm HyPart

(2) We also use an order O_p on the predicates of rules, and assign hash functions to the distinct variables following O_p. Intuitively, the higher the predicates are ranked, the more rules share hash functions for their distinct variables.

(3) We use another order O_h on hash functions, and sort distinct variables following this order. Then a tuple with same functions for different rules can be sent to the same worker. Here O_h can be an arbitrary order. It is used to ensure that the same order is conformed by different rules.

Example 4: Consider three MRL rules $\varphi_1{=}R(t_1){\wedge}S(t_2){\wedge}$ $t_1.B{=}t_2.A{\wedge}t_2.B{=}t_1.A{\rightarrow}t_1.\mathsf{id}{=}t_2.\mathsf{id}$, $\varphi_2 = R(t_3){\wedge}T(t_4){\wedge}$ $t_3.B{=}t_4.A{\wedge}t_4.B{=}t_3.A{\rightarrow}t_3.\mathsf{id}{=}t_4.\mathsf{id}$ and $\varphi_3{=}T(t_5){\wedge}P(t_6){\wedge}$ $t_5.B{=}t_6.A{\wedge}t_6.B{=}t_5.A{\rightarrow}t_5.\mathsf{id}{=}t_6.\mathsf{id}$. We assign hash functions to maximize sharing as follows. We can use only six hash functions (*e.g.,* h_1, h_2, h_3, h_4, h_5 and h_6) to process all of φ_1, φ_2 and φ_3 by sharing hash functions, although the rules have 12 distinct variables. More specifically, (1) we assign h_1, h_2 and h_3 to $R.A$, $R.B$ and $R.\mathsf{id}$, respectively, in both φ_1 and φ_2, since these variables appear in both φ_1 and φ_2 (here we simply use the relation schemas R, S and T to represent the tuple variables). (2) For $S.A$ and $S.B$ in φ_1 (resp. $T.A$ and $T.B$ in φ_2), since they equal to $R.B$ and $R.A$, respectively, they share hash functions with $R.B$ and $R.A$, *i.e.,* h_2 and h_1 are assigned to $S.A$ and $S.B$ in φ_1 (resp. $T.A$ and $T.B$ in φ_2), respectively. (3) For $T.A$ and $T.B$ in φ_3, they can reuse these function since they have been assigned hash functions in φ_2. (4) For the remaining variables (*i.e.,* $S.\mathsf{id}$ in φ_1, $T.\mathsf{id}$ in φ_2 and $P.\mathsf{id}$ in φ_3), we assign h_4, h_5 and h_6 to $S.\mathsf{id}$, $T.\mathsf{id}$ and $P.\mathsf{id}$, respectively.

We reduce the communication cost by using O_h. Define O_h as $(h_1, h_2, h_3, h_4, h_5, h_6)$, and sort distinct variables in φ_1 (resp. φ_2 and φ_3) as $(R.A, R.B, R.\mathsf{id}, S.\mathsf{id})$, (resp. $(R.A,$ $R.B, R.\mathsf{id}, T.\mathsf{id})$ and $(T.B, T.A, P.\mathsf{id}, T.\mathsf{id})$); note that all of φ_1, φ_2 and φ_3 have four distinct variables. Then for all of φ_1, φ_2 and φ_3, tuples in R (resp. T) to be checked are sent to the same place, *i.e.,* workers at positions $(h_1(R.A), h_2(R.B),$ $h_3(R.\mathsf{id}), *)$ (resp. $(h_1(T.B), h_2(T.A), *, h_5(T.\mathsf{id}))$). □

Algorithm. Putting these together, we present algorithm HyPart in Fig. 2. Given a dataset $\mathcal{D}$ and a set Σ of MRLs, HyPart partitions $\mathcal{D}$ into $W_1, \ldots, W_n$ as follows. It first computes a "query plan" QP for rules in Σ via procedure QPforMQO (line 1). Next, it defines order O_r on rules in Σ via

function SortQuery (line 2), and assigns hash functions to each rule via function AssignHash following orders O_p, O_h and O_r (lines 3-5). After that, HyPart partitions $\mathcal{D}$ into fragments $W_1, \ldots, W_n$ based on the assigned hash functions with HC (lines 7-9). It returns the fragments as the partition (line 10).

Procedure SortQuery. SortQuery defines an order O_r on the rules in Σ as follows. Given a rule φ, denote by $\mathcal{N}_\varphi$ the set of rules in Σ that share common predicates with φ in QP. It sorts rules φ in Σ in the descending order of $|\mathcal{N}_\varphi|$ (*i.e.,* the number of rules in $\mathcal{N}_\varphi$; denoted by S_φ), which makes the order O_r.

Procedure AssignHash. Given a query plan QP and a rule φ, AssignHash assigns hash functions to distinct variables of φ (see HC above), and sorts these variables following the order O_h on hash functions as described above. As remarked earlier, we cannot assign hash functions randomly, since the order on hash functions affects the dimensions of hypercube, which in turn affects the number of hash functions shared among rules.

At first, we assign hash functions to distinct variables of φ. (1) We first impose an order O_p on the predicates in φ. More specifically, given a predicate l_p of φ in QP, denote by $\mathcal{N}_{l_p}$ the set of rules having l_p as a predicate; the score of a predicate l_p is defined as $S_{l_p} = |N_{l_p}|$, which makes the order O_p. (2) We then assign hash functions to distinct variables in φ following both O_p and O_h, and also assign these hash functions to other rules that share predicates with φ, such that rules with the same predicates share the same hash functions.

Next we sort the distinct variables based on the order O_h on hash functions. Assume that hash functions $h_1, \ldots, h_m$ are assigned to distinct variables $t_1.A_1, \ldots, t_m.A_m$, respectively. If the order O_h on hash functions is $(h_1, \ldots, h_m)$, then the order of these distinct variables is $(t_1.A_1, \ldots, t_m.A_m)$.

Example 5: Consider dataset $\mathcal{D}$ in Example 1 and the set Σ of φ_1–φ_5 in Example 2. A fragment of query plan QP of Σ is shown in Fig. 1. Algorithm HyPart partitions $\mathcal{D}$ as follows. (1) HyPart first assigns hash functions to rules in Σ. Since each rule has at most 24 attributes (*i.e.,* the number of attributes in φ_4), we need 24 hash functions, sorted as $(h_1, \ldots, h_{24})$; denote this order by O_h. It assigns these functions as follows.

At first, HyPart sorts MRLs in Σ based on score function S_φ, and obtains O_r=$(\varphi_1, \varphi_3, \varphi_4, \varphi_2, \varphi_5)$. This is because φ_1 shares predicates (*i.e.,* t_c.addr=t'_c.addr and t_c.phone = t'_c.phone) with φ_3 and φ_4, *i.e.,* S_{φ_1}=2; φ_3 and φ_4 only share predicates with φ_1, *i.e.,* S_{φ_3}=1 and S_{φ_4}=1 (see Fig. 1); and φ_2 and φ_5 do not share predicates with others, *i.e.,* S_{φ_2}=S_{φ_5} = 0.

HyPart then assigns hash functions to variables of the MRLs in Σ following O_r. Here we only show how to handle φ_1; the other rules are handled similarly. Rule φ_1 has five distinct variables: t_c.name, t_c.phone, t_c.addr, t_c.id and t'_c.id. We assign hash functions as follows. (a) HyPart first determines the order O_p of predicates in φ_1: p_3, p_1, p_2, p_4, where $p_1 = (t_c.\text{phone} = t'_c.\text{phone})$, $p_2 = (t_c.\text{addr} = t'_c.\text{addr})$, $p_3 = (t_c.\text{id} = t'_c.\text{id})$ and $p_4 = (t_c.\text{name} = t'_c.\text{name})$ are all predicates in φ_1. (b) Based on O_p, HyPart then sorts the variables in φ_1 as: (t_c.id, t'_c.id, t_c.phone, t_c.addr, t_c.name). (c) Then distinct variables t_c.id, t'_c.id, t_c.phone, t_c.addr and t_c.name are assigned hash functions h_1, h_2, h_3, h_4 and h_5, *i.e.,* the first five hash functions in O_h, respectively; meanwhile, we also assign the hash functions to other rules that share predicates with φ_1, *e.g.,* we assign h_3 (*i.e.,* the hash function for t_c.phone) to t_c.phone in φ_3.

(2) After all rules are assigned hash functions, HyPart partitions $\mathcal{D}$ using HC. Here we only show the assignment of the tuple t_1 in Table I; the other tuples are assigned similarly.

Since φ_1, φ_3, φ_4 and φ_5 have a relation atom in Customer, we distribute t_1 once for each of these rules. Take φ_1 as an example; it sends t_1 to worker P_i located at $(h_1(t_1.\text{id}), *, h_2(t_1.\text{phone}), h_3(t_1.\text{addr}), h_4(t_1.\text{name}))$, where $*$ means that t_1 is sent to all workers sharing the same first four dimensions (see [6]). We will see that since φ_1 and φ_4 use both attributes t_1.name and t_1.addr, the computations of $h_1(t_1.\text{name})$ and $h_3(t_1.\text{addr})$ are reused when HyPart handles t_1 for φ_4. □

Analysis. Recall that with HC we can compute the query result $Q(\mathcal{D})$ locally, *i.e.,* $Q(\mathcal{D}) = \cup_{i\in[1,n]} Q(W_i)$. We show that HyPart partitions $\mathcal{D}$ such that $\mathcal{D} \models \Sigma$ can be checked locally, *i.e.,* $\mathcal{D} \models \Sigma$ if and only if $W_i \models \Sigma$ $(i \in [1, n])$.

Lemma 6: *Given a dataset $\mathcal{D}$ and a set Σ of* MRLs*,* HyPart *partitions $\mathcal{D}$ such that $\mathcal{D} \models \Sigma$ can be verified locally.* □

Proof sketch: This is to show that $\mathcal{D} \not\models \Sigma$ if and only if there exists a fragment W_i $(i \in [1, n])$ such that $W_i \not\models \Sigma$. It suffices to prove that if there exist an MRL $\varphi = X \rightarrow l$ in Σ and a valuation h of φ in $\mathcal{D}$ such that $h \models X$ but $h \not\models l$, then h is also a match of φ in some fragment W_i. To this end, we verify by contradiction that if h is not a valuation in any fragment, then there exist two attributes $t.A$ and $s.B$ in φ such that $t.A = s.B$ is a precondition in φ (*i.e.,* they are assigned the same hash function f); however, $h(t).A \neq h(s).B$ since they are sent to different workers using the same hash function, which contradicts the fact that $h \models X$. Note that for id predicates $t_1.\text{id} = t_2.\text{id}$ and ML predicates $\mathcal{M}(t_1[\bar{A}_1], t_2[\bar{A}_2])$ in X, there exists at least one worker containing both $h(t_1)$ and $h(t_2)$ by the property of Hypercube [6], since we treat t_i.id and $t_i[\bar{A}_i]$ $(i \in [1, 2])$ as distinct variables as remarked earlier. □

Remarks. (1) As will be seen in Section V, MQO not only reduces hash function computation, but also speeds up the computation of local matches (*i.e.,* to validate $\mathcal{D} \models \Sigma$ locally). (2) To balance the workload among workers, we adopt the virtual block and skewness reduction techniques of [32], [18]. More specifically, when partitioning $\mathcal{D}$ into n fragments, we first construct n^2 virtual blocks to form the hypercube, and then partition $\mathcal{D}$ into n^2 virtual blocks as described as above; after that we split the "heavy" blocks (*i.e.,* blocks with much more tuples than the others) into multiple small blocks by evenly partitioning an instance in $\mathcal{D}$ [18]. We evenly distribute these virtual blocks to n fragments using the algorithm of [7] for the minimum makespan problem, to balance the workload.

V. Parallel Entity Resolution

We next develop a parallel algorithm DMatch to implement the fixpoint model (Section III). We first present Match, a sequential algorithm (Section V-A). We then deduce parallelly scalable DMatch by parallelizing Match (Section V-B).

Input: a dataset $\mathcal{D}$ and a set Σ of MRLs.
Output: Set Γ of local matches of $\mathcal{D}$ using Σ.
1. $\Gamma := \{(t.\mathsf{id}, t.\mathsf{id}) \mid t \in \mathcal{D}\}$; $\mathcal{H} := \emptyset$;
2. $(\Gamma, \mathcal{H}) \leftarrow \mathsf{Deduce}(\mathcal{D}, \Sigma)$;
3. $\Delta\Gamma := \Gamma$;
4. **while** $\Delta\Gamma \neq \emptyset$ **do**
5. $(\Delta\Gamma, \Delta\mathcal{H}) := \mathsf{IncDeduce}(\mathcal{D}, \Sigma, \Gamma, \mathcal{H}, \Delta\Gamma)$;
6. $\Gamma := \Gamma \cup \Delta\Gamma$; $\mathcal{H} := \mathcal{H} \cup \Delta\mathcal{H}$;
7. return Γ;

Fig. 3. Algorithm Match

A. *Sequential Algorithm* Match

Algorithm Match computes the set Γ of matches using the query plan QP constructed in Section IV. To speed up the process, it reuses intermediate results when checking valuations of Σ in $\mathcal{D}$. However, this is nontrivial. (a) When $\mathcal{D}$ is large, it is too costly to store all intermediate results generated by QP. (b) Unlike answering multiple CQ queries [44], the computation of Γ is recursive. As a consequence, unsatisfied id or ML predicates may become valid during the computation, and hence we have to revisit some valuations that involve these changed predicates. Moreover, (c) id predicates are transitive, *i.e.,* if $t_1.\mathsf{id} = t_2.\mathsf{id}$ and $t_2.\mathsf{id} = t_3.\mathsf{id}$, then $t_1.\mathsf{id} = t_3.\mathsf{id}$. It is necessary to efficiently deduce the transitivity.

We address these by using the following data structures.

(1) Instead of storing all intermediate results, we dynamically maintain a set of inverted indices for predicates p associated with nodes n_p in QP. (a) Consider $t.A{=}s.B$, where t (resp. s) is a tuple variable for instance D_1 (resp. D_2). For each common value d in both attribute A of D_1 and attribute B of D_2, we create an inverted index from d to tuples t' in D_1 (resp. D_2) with $t'.A{=}d$ (resp. $t'.B{=}d$); similarly for $t.A{=}c$. (b) For $\mathcal{M}(t[\bar{A}], s[\bar{B}])$, an inverted index is constructed from n_p to tuples t and s with $\mathcal{M}(t[\bar{A}], s[\bar{B}]){=}\mathsf{true}$. (c) For $t.\mathsf{id}{=}s.\mathsf{id}$, an inverted index is maintained from n_p to tuples t and s such that $t.\mathsf{id}{=}s.\mathsf{id}$ is not in Γ but is verified during the computation.

(2) To avoid checking the same valuations repeatably, we maintain a set $\mathcal{H}$ of dependencies among id and ML predicates. where a dependency is in the form of $l_1 \wedge l_2 \wedge \ldots \wedge l_n \rightarrow l$, and l and l_i $(i \in [1, n])$ are either id predicates or ML predicates. Intuitively, a dependency encodes that whenever all predicates l_1, l_2, …, and l_n are valid, predicate l has to be enforced. We do not consider predicates $t.A = s.B$ and $t.A = c$ since their validations do not change during the recursive process.

We maintain $\mathcal{H}$ of a bounded size. (a) We use a predefined constant K to bound the number of dependencies in $\mathcal{H}$; here K is determined by the available memory; and (b) whenever a predicate l is validated, we remove all dependencies $l_1 \wedge l_2 \wedge \ldots \wedge l_n{\rightarrow}l$ from $\mathcal{H}$, which will no longer be checked later on.

(3) We define an equivalence relation $\mathcal{E}_{\mathsf{id}}$ of matches in Γ to cope with the transitivity of id predicates. That is, for each tuple t in $\mathcal{D}$ we define an equivalence class $[t.\mathsf{id}]_{\mathcal{E}_{\mathsf{id}}}$, containing all tuples s such that $t.\mathsf{id} = s.\mathsf{id}$ can be deduced from Γ.

Algorithm. Using these structures, we develop Match in Fig. 3. Given a dataset $\mathcal{D}$ and a set Σ of MRLs, Match iteratively deduces a set Γ of matches from $\mathcal{D}$ using Σ. It first initializes Γ with pairs $(t.\mathsf{id}, t.\mathsf{id})$ for all tuples t in $\mathcal{D}$ (line 1).

Input: a dataset $\mathcal{D}$, set Σ of MRLs, set Γ of local matches, a set $\mathcal{H}$ of dependencies and a set $\Delta\Gamma$ of updates to Γ.
Output: A set $\Delta\Gamma'$ (resp. $\Delta\mathcal{H}$) of updates to $\Gamma \cup \Delta\Gamma$ (resp. $\mathcal{H}$).
1. $\mathcal{E}_{\mathsf{id}} := \mathsf{ConstructEq}(\Gamma)$; $\Delta\Gamma' := \emptyset$; $\Delta\mathcal{H} = \emptyset$;
2. **for each** $l_1 \wedge \ldots \wedge l_n \rightarrow l$ in $\mathcal{H}$ s.t. $l_i \in \mathcal{E}_{\mathsf{id}}$ for all $i \in [1, n]$ **do**
3. add l to $\Delta\Gamma'$; remove all $l'_1 \wedge \ldots \wedge l'_m \rightarrow l$ from $\mathcal{H}$;
4. **for each** $(t_1, t_2) \in \mathcal{D} \times \mathcal{D}$ s.t. $l_t = (t_1.\mathsf{id} = t_2.\mathsf{id}) \notin \mathcal{E}_{\mathsf{id}}$ **do**
5. **if** $\exists\ h$ and $\varphi = X \rightarrow l$ s.t. $h(l) = l_t \wedge h(X) \cap \Delta\Gamma \neq \emptyset$ **then**
6. add l to $\Delta\Gamma'$;
7. **else** extend the set $\Delta\mathcal{H}$ of dependencies;
8. **return** $(\Delta\Gamma', \Delta\mathcal{H})$.

Fig. 4. Algorithm IncDeduce

It then builds the auxiliary structures mentioned above, and deduces matches in Γ and dependencies in $\mathcal{H}$ in one round by using procedure Deduce (line 2, see below). After this, it iteratively and incrementally extends Γ using rules in Σ and new matches in Γ via procedure IncDeduce (lines 4-6). It terminates when no new match can be deduced (line 4).

Procedure Deduce. Deduce first constructs inverted indices for predicates $t.A{=}c$ and $t.A = s.B$. Then for each $\varphi{=}X{\rightarrow}l$ in Σ and each valuation h of φ in $\mathcal{D}$, it checks whether $h{\models}l$, *i.e.,* $h(l)$ can be validated, where $h(l)$ is either $(h(t_1).\mathsf{id}, h(t_2).\mathsf{id})$ or $\mathcal{M}(h(t_1)[\bar{A}], h(t_2)[\bar{B}])$. If so, it adds $h(l)$ to Γ; otherwise, it adds dependency $l_1 \wedge l_2 \wedge \ldots \wedge l_n{\rightarrow}l$ to $\mathcal{H}$ if there is still space in $\mathcal{H}$, and builds inverted indices for l_1, …, l_n and l.

Procedure IncDeduce. As shown in Fig. 4, IncDeduce incrementally extends Γ by using an update-driven strategy. It first constructs the equivalence relation $\mathcal{E}_{\mathsf{id}}$ (line 1). It then deduces new matches using $\mathcal{H}$ and $\mathcal{E}_{\mathsf{id}}$ (lines 2-3), *i.e.,* for a dependency $l_1{\wedge}l_2{\wedge}\ldots{\wedge}l_n{\rightarrow}l$ in $\mathcal{H}$, if each l_i $(i{\in}[1, n])$ has been validated in $\mathcal{E}_{\mathsf{id}}$, then it adds l to a set $\Delta\Gamma'$ of updates, and changes $\mathcal{H}$ accordingly. After that it extends $\Delta\Gamma'$ by inspecting only valuations that involve new matches in $\Delta\Gamma$ (updated-driven; lines 4-7). For each unverified id or ML predicate $l_t{\notin}\mathcal{E}_{\mathsf{id}}$, it checks whether there exist a rule $\varphi{=}X{\rightarrow}l$ and a valuation h of φ in $\mathcal{D}$ such that $h{\models}X$, $h(l){=}l_t$ and $h(X)$ involves at least one match in $\Delta\Gamma$ (*i.e.,* $h(X){\cap}\Delta\Gamma{\neq}\emptyset$); if so, it adds l_t to $\Delta\Gamma'$; otherwise, it updates the set $\Delta\mathcal{H}$ of new dependencies; this can be done using inverted indices constructed for id and ML predicates.

IncDeduce iterates until no new matches are found.

Analysis. The correctness follows from Theorem 1. Its runtime, denoted by $T_{\mathsf{Match}}(\mathcal{D}, \Sigma)$, is in $O(\|\Sigma\|^2(|\Sigma| + 1)|\mathcal{D}|^{|\Sigma|+2})$. Indeed, (1) there exist at most $O(\|\Sigma\|(|\Sigma|{+}1)|\mathcal{D}|^2)$ many matches; and (2) Match deduces matches by checking the existence of valuations of Σ in $\mathcal{D}$, which is in $O(\|\Sigma\||\mathcal{D}|^{|\Sigma|})$.

Remark. We can also extend IncDeduce to handle updates $\Delta\mathcal{D}$. Given $\Delta\mathcal{D}$, it inspects valuations of MRLs that involve changes, and recursively propagates changes to other matches. We can verify that it only checks affected areas. With the update strategy we can also incrementally compute partitions.

B. *Parallel Algorithm* **DMatch**

Before presenting parallel algorithm DMatch, we first review the notion of *parallel scalability* [47], which is widely used to characterize the effectiveness of parallel algorithms.

Parallel scalability. A parallel algorithm $\mathcal{A}_p$ for deep and collective ER is *parallelly scalable relative to* the sequential

Match if it is in $O(\frac{t_{\mathsf{Match}}(\mathcal{D},\Sigma)}{n})$ time, where $t_{\mathsf{Match}}(\mathcal{D},\Sigma)$ is the runtime of Match, and n is the number of processors used.

Intuitively, a parallelly scalable $\mathcal{A}_p$ "linearly" reduces the cost when n increases. Hence a parallelly scalable algorithm is able to scale with large $\mathcal{D}$ by adding processors as needed.

The main result of this section is the following.

Theorem 7: *There exists a parallel algorithm* DMatch *that is parallelly scalable relative to* Match. □

As a proof of Theorem 7, we next develop DMatch, and show that DMatch is parallelly scalable relative to Match.

Algorithm. Given fragments $W_1, \ldots, W_n$ partitioned by algorithm HyPart (Section IV), DMatch first uses an algorithm $\mathcal{A}$ to compute local matches in W_i in parallel; it then iteratively runs an algorithm $\mathcal{A}_\Delta$ to incrementally deduce matches in response to new matches from other workers (see Section III-B).

(1) Partial evaluation $\mathcal{A}$. Algorithm $\mathcal{A}$ is a simple extension of the sequential procedure Deduce; it is run at all workers in parallel. At the end of $\mathcal{A}$, each worker P_i sends deduced matches $(t.\mathsf{id}, s.\mathsf{id})$ and $\mathcal{M}(t[\bar{A}], s[\bar{B}])$ to master P_0 such that t or s also resides at other workers (in other fragments).

(2) Incremental computation $\mathcal{A}_\Delta$. Algorithm $\mathcal{A}_\Delta$ is an extension of the sequential procedure IncDeduce with the following: (a) when receiving messages from other workers, master P_0 first takes union of all received matches; then P_0 routes those matches to workers P_i that contain a tuple in these matches; (b) these matches are treated as update $\Delta\Gamma$ to Γ; and (c) when $\mathcal{A}_\Delta$ is done with a round of computation, it sends newly-deduced local matches to other workers just like in $\mathcal{A}$ above.

Remark. Algorithms $\mathcal{A}$ and $\mathcal{A}_\Delta$ can be implemented by extending Deduce and IncDeduce because (a) fragments are generated via HyPart; hence matches can be deduced using only local data, and it suffices to exchange newly deduced ids or ML predicates (Lemma 6); and moreover, (b) IncDeduce follows an update-driven strategy to incrementally compute Γ.

Example 6: Continuing with Examples 3 and 5, assume that dataset $\mathcal{D}$ in Example 1 is partitioned via HyPart into 2 fragments W_1 and W_2. Then parallel ER is conducted as follows.

(1) Algorithm $\mathcal{A}$ first conducts local matching and constructs the set $\mathcal{H}$ of dependencies, at all workers in parallel. As shown in Fig. 5(1), (a) it identifies $t_2.\mathsf{id}{=}t_3.\mathsf{id}$ and $t_{12}.\mathsf{id}{=}t_{13}.\mathsf{id}$ in W_1 and $t_9.\mathsf{id}{=}t_{10}.\mathsf{id}$ in W_2, as marked by dashed circles. (b) For $t_1.\mathsf{id}{=}t_3.\mathsf{id}$, it finds that it relies on $t_9.\mathsf{id}{=}t_{10}.\mathsf{id}$ and $t_{12}.\mathsf{id}{=}t_{13}.\mathsf{id}$; since $t_9.\mathsf{id}{=}t_{10}.\mathsf{id}$ is not yet identified in W_1, it cannot deduce $t_1.\mathsf{id}{=}t_3.\mathsf{id}$. So it adds dependency $t_9.\mathsf{id}{=}t_{10}.\mathsf{id} \rightarrow t_1.\mathsf{id}{=}t_3.\mathsf{id}$ to $\mathcal{H}$ (marked by an arrow in Fig. 5(1)).

When $\mathcal{A}$ terminates, match $t_9.\mathsf{id} = t_{10}.\mathsf{id}$ is sent from W_2 to W_1. No match is sent from W_1 to W_2 since the matches found in W_1 do not involve tuples residing in W_2.

(2) After receiving $t_9.\mathsf{id}{=}t_{10}.\mathsf{id}$, incremental algorithm $\mathcal{A}_\Delta$ continues to deduce other matches in W_1. Using dependency $t_9.\mathsf{id}{=}t_{10}.\mathsf{id}{\rightarrow}t_1.\mathsf{id}{=}t_3.\mathsf{id}$ in $\mathcal{H}$, it can identify $t_1.\mathsf{id}{=}t_3.\mathsf{id}$, and merge the equivalence classes containing t_1 and t_2 (*i.e.,* $\{t_1\}$ and $\{t_2, t_3\}$), from which we can deduce $t_1.\mathsf{id}{=}t_2.\mathsf{id}$. Meanwhile, it removes $t_9.\mathsf{id}{=}t_{10}.\mathsf{id}{\rightarrow}(t_1.\mathsf{id}{=}t_3.\mathsf{id})$ from $\mathcal{H}$. After that no more match can be deduced in W_1, and $\mathcal{A}_\Delta$ terminates.

(3) At this point, no message is exchanged, and the process terminates for the same reason as given in Example 3. □

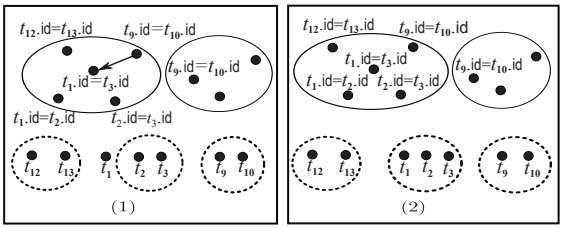

Fig. 5. The constructed hypergraphs G_1 and G_2

Correctness. Algorithm DMatch is correct.

Proposition 8: *Given a set Σ of* MRLs *and a partition $W_1, \ldots, W_n$ of a dataset $\mathcal{D}$ via* HyPart*, algorithm* DMatch *correctly conducts deep and collective ER.* □

Proof sketch: We show that (1) by induction on chase steps, each match deduced by the chase can be deduced by $\mathcal{A}$ and $\mathcal{A}_\Delta$; and (2) by induction on supersteps of $\mathcal{A}$ and $\mathcal{A}_\Delta$, each match deduced by $\mathcal{A}$ and $\mathcal{A}_\Delta$ can be deduced by the chase. □

Parallel scalability. Intuitively, DMatch is parallelly scalable because (a) checking $\mathcal{D} \models \Sigma$ can be done locally by partitioning $\mathcal{D}$ via algorithm HyPart (Lemma 6); and (b) only new deduced matches are transmitted among fragments.

Proof sketch of Theorem 7. Since the sequential algorithm Match is in $O(\|\Sigma\|^2(|\Sigma|{+}1)|\mathcal{D}|^{|\Sigma|+2})$ time (see Section V-A), we show that DMatch is in $O(\frac{\|\Sigma\|^2(|\Sigma|+1)|\mathcal{D}|^{|\Sigma|+2}}{n})$ time. Indeed, (1) $\mathcal{A}$ runs only once, and takes $O(\frac{\|\Sigma\|^2(|\Sigma|+1)(|\mathcal{D}|)^{|\Sigma|+2}}{n})$ time, since $\mathcal{D}$ is evenly partitioned using HC and skewness reduction [32], [18] (Section IV); (2) the total runtime of $\mathcal{A}_\Delta$ is also bounded by $O(\frac{\|\Sigma\|^2(|\Sigma|+1)|\mathcal{D}|^{|\Sigma|+2}}{n})$, since (a) $\mathcal{A}_\Delta$ adopts an update-driven approach, (b) there exist at most $O(\|\Sigma\|(|\Sigma|+1)|\mathcal{D}|^2)$ many updates, and (c) for each update, $\mathcal{A}_\Delta$ runs in $O(\frac{\|\Sigma\||\mathcal{D}|^{|\Sigma|}}{n})$ time; and (3) the communication cost is bounded by $O(\|\Sigma\|(|\Sigma|{+}1)|\mathcal{D}|^2)$, since only newly deduced matches are exchanged among workers. Hence DMatch is in $O(\|\Sigma\|(|\Sigma|{+}1)|\mathcal{D}|^2\frac{\|\Sigma\||\mathcal{D}|^{|\Sigma|}}{n}){\leq}O(\frac{\|\Sigma\|^2(|\Sigma|+1)|\mathcal{D}|^{|\Sigma|+2}}{n})$ time. □

VI. Experimental Study

Using real-life and synthetic data, we conducted three sets of experiments to evaluate the (1) accuracy, (2) efficiency, (3) (parallel) scalability of algorithm DMatch. We also conducted a case study to exemplify effective MRLs from real-life data.

Experimental setup. We start with the experimental setting.

Datasets. We used five real-life datasets. (a) IMDB, a movie dataset with 1.5+M tuples [24]; (b) DBLP, a bibliographic dataset with 5k tuples and 4 attributes [2]. (c) Movie, movies and directors [24] with 5 tables, 22 attributes and 1+M tuples. (d) Songs, a dataset of musics and artists with 2+M tuples and 8 attributes [24]. (e) TFACC, a dataset of Ministry of Transport [1] with 19 tables, 113 attributes and 480+M tuples.

We also generated synthetic datasets TPCH using TPCH-dbgen [4], with 30M tuples, 8 relations and 61 attributes.

Ground truth and noises. IMDB, DBLP, Movie and Songs have 270942, 2225, 14989 and 90581 tuple pairs labeled as matches and non-matches, which were treated as ground truth. For TFACC and TPCH without labeled data, we assumed that they are correct (*i.e.,* no duplicate exists in TFACC and TPCH [32]), and duplicated tuples randomly, controlled by the number Dup of duplicates. The unit of Dup is million tuples.

Measurements. The accuracy was measured in F-Measure = $2 \times \text{Precision} \times \text{Recall}/(\text{Precision} + \text{Recall})$. Precision (resp. Recall) is the ratio of deduced *true* matches included in the ground truth to all deduced matches (resp. the ground truth).

ML classifiers. We adopted two existing bi-variable ML models as predicates in MRLs: supervised ER model DeepER [25] and semantic similarity detection model fasttext [19].

MRLs. We discovered 10, 10, 10, 10, 30 and 75 MRLs from IMDB, DBLP, Movie, Songs, TFACC and TPCH, respectively, by extending the algorithm of [23] for discovering denial constraints (denoted by DCs); DCs have the form of $\forall t_1, \ldots, t_m \neg(P_1 \wedge \ldots \wedge P_n)$, stating that for each valuation of tuple variables $t_1, \ldots, t_m$, one of predicates $P_1, \ldots, P_n$ does not hold. Given a database $\mathcal{D}$, the algorithm of [23] discovers DCs as follows: (1) it first builds a predicate space $\mathcal{P}$ and prepares an evidence set for $\mathcal{D}$, which consists of a set $E_{(t,s)}$ of predicates in $\mathcal{P}$ for every tuple pair (t, s) such that (t, s) satisfies all predicates in $E_{(t,s)}$; (2) it then generates DCs by computing minimal set covers of the evident set; and (3) it finally checks the support/confidence of DCs, and returns DCs that have support/confidence above predefined bounds.

We can extend the algorithm to mine MRLs, since MRLs can be rewritten as $\neg(X \wedge \neg l)$ by De Morgan's law, in the form of DCs. However, we need to address the following: (1) MRLs support multiple tuple variables, while the algorithm of [23] only discovers bi-variable DCs. To cope with this, we built a lattice for tuple variables, and followed the lattice to construct tuple variables in MRLs [59]; and (2) MRLs contain ML predicates; to support ML predicates, (a) we first collected candidate ML models, *e.g.,* DeepER [25] for ER and fasttext [19] for semantic similarity checking; and (b) we uniformly picked equality and ML predicates when constructing the evidence set; *i.e.,* for any two tuples t and s, we treated ML predicates $\mathcal{M}(t[\bar{A}], s[\bar{B}])$ in the same way as $t.A{=}s.B$, and added them to the evident set for tuple pair (t, s) if $\mathcal{M}(t[\bar{A}], s[\bar{B}])$ returns true. The rest is the same as the algorithm of [23] for DCs.

Baselines. We implemented the following algorithms, all in C++. (1) DMatch (Section V); (2) its variant $\text{DMatch}_{\text{noMQO}}$ without MQO technique; (3) DMatch_{C} and (4) DMatch_{D}, two variants of DMatch that conduct collective ER and deep ER only, respectively; specifically, DMatch_{C} uses only MRLs without id predicates in preconditions; and DMatch_{D} uses only MRLs with at most 4 tuple variables, since real-life data quality rules usually use at most 3 tuple variables [32].

We also compared with 8 baselines: (5) ERBlox [12] that exploits MDs rules and machine learning for ER; (6) DeepMatcher [43] that applies deep learning to ER; (7) DeepER [25], an ML method that uses LSH for blocking and adopts LSTM to detect duplication; (8) Ditto [48], an ER system that adopts pre-trained language models to capture the context of entities; (9) JedAI [53], a rule-based tool that focuses on non-learning and structure-agnostic ER; (10) Dedoop [45], a parallel MD-based deduplication method in Hadoop; (11) SparkER [35] that implements both schema-agnostic and Blast meta-blocking approaches [60] in Spark; and (12) DisDedup [22] that is similar to Dedoop but minimizes the maximum workload across all workers in Spark environment.

	IMDB		ACM-DBLP		Movie		Songs	
	F	T(sec.)	F	T	F	T	F	T
DeepMa.	0.71	-	0.92	-	**0.99**	-	0.82	-
JedAI	**0.97**	-	0.88	-	0.44	-	0.38	-
ERBlox	0.91	-	0.66	-	0.37	-	0.25	-
DeepER	0.71	-	0.64	-	0.47	-	0.66	-
Ditto	0.79	6741.2	**0.98**	10.36	0.66	648.83	**0.99**	670.9
DisDedup	0.67	534	0.82	12.3	0.9	2485	0.16	45.6
Dedoop	0.53	$*$	0.19	$*$	0.65	$*$	0.65	$*$
SparkER	0.66	393	0.77	22.7	0.03	**46.132**	0.09	17.9
DMatch	**0.97**	**387**	0.96	**3.48**	**0.99**	271	0.98	**3.68**

TABLE V
ACCURACY

For ML models such as DeepMatcher and DeepER, datasets are split into training data and testing data in the ratio 2:1. We configured JedAI with the recommended parameters of [53], and DisDedup with the same configuration of [3]. Blocking and weight average matching are used in Dedoop [45].

Environment. We conducted the experiments on a HPC cluster of up to 32 machines, each powered with 2.40GHz Intel Xeon Gold CPU, 4TB Intel P4600 SSD and 128GB memory. These machines are connected by 100 Mbps links. By default we used n=32 machines and all discovered MRLs unless stated otherwise. Experiments with Dedoop were conducted on other machines, due to hardware constraints of the cluster; we only report the accuracy of Dedoop. Each experiment was repeated for 5 times, and the average is reported. We only show the results on some datasets; results on the others are consistent.

Experimental results. We report our findings as follows.

Exp-1: Accuracy. We first tested the accuracy of the methods.

(1) On datasets with ground truth (*i.e.,* IMDB, ACM-DBLP, Movie and Songs), on average DMatch outperforms SparkER, Dedoop, DisDedup, DeepER, ERBlox and JedAI by 58.75%, 47%, 33.74%, 35.5%, 42.75% and 30.75%, respectively, as shown in Table V. It is 26% and 33% better than DeepMatcher on IMDB and Ditto on Movie, respectively. This is because DMatch combines rule-based and ML-based methods, while others are based on either ML models (DeepER, DisDedup, DeepMatcher, Ditto) or logic rules (Dedoop, JedAI) alone.

(2) Fixing Dup=0.5, Figures 6(a)-(b) report results on TPCH and TFACC, respectively. As shown there, the F-measure of DMatch is above 0.86. On average, it beats Dedoop, DisDedup and SparkER by 39.9%, 57.24% and 64.6%, respectively. This is because some duplicates can only be detected recursively, not by baselines; we only report the results of distributed baselines, since the others cannot terminate in 4 hours.

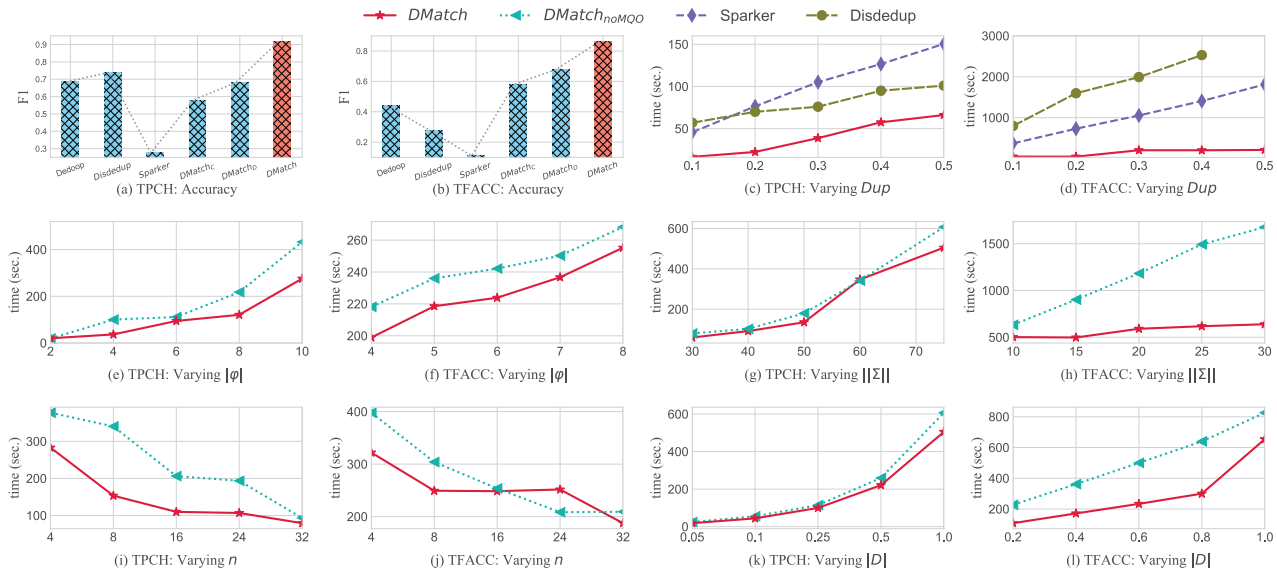

Fig. 6. Performance evaluation

(3) Deep and collective ER outperforms both deep ER and collective ER in accuracy. As shown in Figure 6(a), the F-measure of DMatch is 0.92, which is 33% and 23% higher than DMatch$_C$ and DMatch$_D$, respectively, on TPCH; the results on TFACC are consistent (see Figure 6(b)). This justifies the need for deep and collective ER. Note that while Dedoop and Disdedup perform better than DMatch$_C$ and DMatch$_D$ (Figure 6(a)), they may still miss duplicates that can only be detected recursively. This justifies the need for DMatch that unifies deep ER and collective ER, which outperforms Dedoop and Disdedup (see Figure 6(a)). DMatch$_C$ and DMatch$_D$ may miss matches since DMatch$_C$ cannot detect ER recursively, and DMatch$_D$ restricts the number of relation atoms.

(4) Varying Dup from 0.1 to 0.5, we report its impact on the accuracy of DMatch. As shown in Table VI, (1) on TPCH the accuracy of DMatch slightly decreases with larger Dup, when tuples are duplicated randomly; and (2) the discovered MRLs can identify almost all added duplicates, *e.g.,* when Dup=0.5, the accuracy of DMatch reaches 0.8669 on TPCH.

(5) Fixing Dup $=$ 0.5, we also tested ER on a universal relation by denormalizing datasets; here we only reported the results on TPCH; the results on other datasets are consistent. On TPCH, we combined tables via foreign keys; *e.g.,* we joined tables Parts and Supplier using foreign keys PartKey and SuppKey of table ParSupp; denote by TPCH$_d$ the joined dataset. Over TPCH$_d$, the accuracy of SparkER and DisDedup is 28% and 83%, respectively. We found that (a) denormalizing the tables incurs heavy cost, *e.g.,* it takes 1517.44 seconds and 134 GB of memory to denormalize TPCH$_d$ that has 30M tuples; (b) DMatch is consistently more accurate than all the baselines (the accuracy of DMatch on TPCH is above 0.86); as remarked earlier, some duplicates can only be detected via recursions, and it is hard to decide the number of joins needed for denormalizing in order to catch such duplicates. (c) As an example to justify the need for recursion, in TPCH there exist duplicate orders that are handled by the same clerk in the same day but are placed by two customers who have the same name but different nationalities. To find that the two orders are actually the same, (1) we first identified countries "Argenztina" and "Argwentisna" in table nation; (b) based on this result we identified the two customers in the next round of recursion; and (c) we then concluded that the two orders are the same in the 3rd round of deep ER (recursion), since they buy the same products, and are placed by the same customer and handled by the same clerk. This process needs 3 levels of recursion.

Dup	0.1	0.2	0.3	0.4	0.5
TPCH	0.8874	0.8845	0.8760	0.8705	0.8669
TFACC	0.8542	0.8502	0.8565	0.8501	0.8606

TABLE VI
ACCURACY OF DMatch ON TPCH AND TFACC

Exp-2: Efficiency. We evaluated the efficiency of DMatch.

Partitioning. We tested the partitioning time (*i.e.,* the running time of HyPart) and the ER time (*i.e.,* the time of DMatch) by varying n from 4 to 32. We found the following: (1) the ER time dominates overall cost. When n=4, $|\varphi|$=8 and $\|\Sigma\|$=10, on TPCH it takes 18.19s to partition, and 254.73s to conduct ER. (2) When n is varied from 4 to 32, the partitioning time decreases (from 18.19s to 11.49s), and it accounts for at most 15.32% of the ER time. Thus in the following we only report the ER time, *i.e.,* the running time of DMatch.

Varying Dup. Fixing $n = 16$, we varied Dup from 0.1 to 0.5, and tested the impact of the number of duplicates on the efficiency of DMatch. As reported in Figures 6(c)-(d) on TPCH and TFACC, respectively, (1) all algorithms take longer when Dup gets larger, as expected. (2) On TPCH, DMatch is 2.6 and 2.3 times faster than SparkER and DisDedup, respectively. The other baselines cannot terminate in 4 hours (Table V); as mentioned above, we did not report the running time of Dedoop, and marked it by a star $*$. These verify that while deep and collective ER correlates data from multiple relations and is recursively conducted, it is even faster than the baselines by partitioning data with HyPart and reducing the cost with

MQO, while substantially improving the accuracy. Note that DisDedup ran out of memory on TFACC when Dup $= 0.5$.

Varying $|\varphi|$. We also tested the impact of the average number $|\varphi|$ of predicates in each MRL φ. Fixing $n = 16$ and the number $\|\Sigma\|$ of MRLs as 10, we varied $|\varphi|$ from 2 to 10 over TPCH (resp. from 4 to 8 over TFACC). As shown in Figures 6(e)-6(f) on TPCH and TFACC, respectively, (1) DMatch takes longer with larger MRLs, as expected. (2) DMatch is practical for real-life MRLs. When $|\varphi|$=8 it takes 216s on TFACC with 8+M tuples, which is the subset of TFACC covered by the used rules. (3) On average DMatch beats $\mathsf{DMatch}_{\mathsf{noMQO}}$ by 35.9%. This validates the effectiveness of the MQO technique, since the more predicates MRLs contain, the more intermediate results these rules may share. This verifies the effectiveness of our optimization techniques.

Varying $\|\Sigma\|$. We next evaluated the impact of the number $\|\Sigma\|$ of MRLs in Σ. Fixing n=16, we varied $\|\Sigma\|$ from 30 to 75 for TPCH (resp. from 10 to 30 for TFACC) to evaluate the efficiency of DMatch and $\mathsf{DMatch}_{\mathsf{noMQO}}$. As shown in Figure 6(g)-6(h), (1) DMatch takes longer when given more rules, as expected. (2) DMatch is faster than $\mathsf{DMatch}_{\mathsf{noMQO}}$. When $\|\Sigma\|$=75, DMatch outperforms $\mathsf{DMatch}_{\mathsf{noMQO}}$ by 20% on TFACC. This is because DMatch applies MQO and allows different rules to share intermediate results.

Exp-3: Scalability. We further evaluated the scalability of DMatch and $\mathsf{DMatch}_{\mathsf{noMQO}}$ using TPCH and TFACC.

Varying n. Fixing $\|\Sigma\|$=75 for TPCH (resp. $\|\Sigma\| = 30$ for TFACC), we varied the number n of workers from 4 to 32. As shown in Figures 6(i)-6(j), (a) DMatch and $\mathsf{DMatch}_{\mathsf{noMQO}}$ scale well. When n varies from 4 to 32, DMatch (resp. $\mathsf{DMatch}_{\mathsf{noMQO}}$) is 3.56 (resp. 4.03) times faster on TPCH. (b) DMatch works well on large datasets. When $n = 32$, DMatch takes 187s to process TFACC with 8+M tuples. (c) On average, DMatch outperforms $\mathsf{DMatch}_{\mathsf{noMQO}}$ by 68% and 8% on TPCH and TFACC, respectively, by using MQO.

Synthetic data. We tested the scalability of DMatch on TPCH by varying the scale factor from 0.05 to 1 and fixing $n = 16$. As shown in Figure 6(k), (1) DMatch takes longer on larger datasets; it takes 505s on TPCH when the scale factor is 1, *i.e.,* the entire TPCH; while $\mathsf{DMatch}_{\mathsf{noMQO}}$ takes more than 607s. (2) On average, DMatch outperforms $\mathsf{DMatch}_{\mathsf{noMQO}}$ by 58.5%. The results on TFACC are consistent (Figure 6(l)).

Exp-4: Case Study. Below we give 4 example MRLs that were discovered from TPCH, DBLP and the data of our industry partners. All these rules are defined across 2-3 tables and carry 4-8 relation atoms; and each carries both ML and id predicates. These are beyond rules used in previous work for ER.

(1) φ_a: $\mathsf{Parts}(t_p) \wedge \mathsf{Parts}(t'_p) \wedge \mathsf{Partsupp}(t_{ps}) \wedge \mathsf{Partsupp}(t'_{ps}) \wedge \mathsf{Supplier}(t_s) \wedge \mathsf{Supplier}(t'_s) \wedge t_p.\mathsf{partkey} = t_{ps}.\mathsf{partkey} \wedge t_{ps}.\mathsf{suppkey} = t_s.\mathsf{suppkey} \wedge t'_{ps}.\mathsf{partkey} = t'_p.\mathsf{partkey} \wedge t'_s.\mathsf{suppkey} = t'_{ps}.\mathsf{suppkey} \wedge t_s.\mathsf{id} = t'_s.\mathsf{id} \wedge t_{ps}.\mathsf{supplycost} = t'_{ps}.\mathsf{supplycost} \wedge \mathcal{M}(t_p.\mathsf{desc}, t'_p.\mathsf{desc}) \rightarrow t_p.\mathsf{id} = t'_p.\mathsf{id}$. The rule identifies two parts if they share the same supplier and supply cost, and bear similar descriptions (by ML). It plugs in an ML predicate for checking similarity of long text data.

(2) φ_b : $\mathsf{Orders}(t_o) \wedge \mathsf{Orders}(t'_o) \wedge \mathsf{Cust}(t_c) \wedge \mathsf{Cust}(t'_c) \wedge \mathsf{Lineitem}(t_l) \wedge \mathsf{Lineitem}(t'_l) \wedge t_o.\mathsf{custkey} = t_c.\mathsf{custkey} \wedge t_o.\mathsf{orderkey} = t_l.\mathsf{orderkey} \wedge t'_o.\mathsf{custkey} = t'_c.\mathsf{custkey} \wedge t'_o.\mathsf{orderkey} = t'_l.\mathsf{orderkey} \wedge t_o.\mathsf{totalprice} = t'_o.\mathsf{totalprice} \wedge t_o.\mathsf{orderdate} = t'_o.\mathsf{orderdate} \wedge t_c.\mathsf{id}{=}t'_c.\mathsf{id} \wedge t_l.\mathsf{partkey} = t'_l.\mathsf{partkey} \wedge \mathcal{M}(t_o.\mathsf{clerkname}, t'_o.\mathsf{clerkname}) \rightarrow t_o.\mathsf{id} = t'_o.\mathsf{id}$. It catches two orders as duplicates if they have the same totalprice, orderdate, clerk, customers and partkey of items.

(3) φ_c : $\mathsf{Article\_Author}(t_1) \wedge \mathsf{Article\_Author}(t_2) \wedge \mathsf{Article}(t_3) \wedge \mathsf{Article}(t_4) \wedge \mathsf{Author}(t_5) \wedge \mathsf{Author}(t_6) \wedge t_1.\mathsf{article\_id}{=}t_3.\mathsf{article\_id} \wedge t_2.\mathsf{article\_id} = t_4.\mathsf{article\_id} \wedge t_1.\mathsf{author\_id} = t_5.\mathsf{author\_id} \wedge t_2.\mathsf{author\_id} = t_6.\mathsf{author\_id} \wedge \mathcal{M}_l(t_5, t_6) \wedge t_3.\mathsf{title}{=}t_4.\mathsf{title} \wedge t_3.\mathsf{booktitle}{=}t_4.\mathsf{booktitle} \wedge t_3.\mathsf{year}{=}t_4.\mathsf{year} \wedge t_3.\mathsf{issue}{=}t_4.\mathsf{issue} \wedge \mathcal{M}_a(t_3.\mathsf{abstract}, t_4.\mathsf{abstract}) \rightarrow t_3.\mathsf{id}{=}t_4.\mathsf{id}$. It identifies two papers if they have the same title, booktitle, year and issue, and similar abstractions, and if they have a common author. We can verify that all the preconditions in φ_c are necessary.

(4) φ_d : $\mathsf{Comment}(t_1) \wedge \mathsf{Comment}(t_2) \wedge \mathsf{User}(t_3) \wedge \mathsf{User}(t_4) \wedge t_1.\mathsf{item\_id}{=}t_2.\mathsf{item\_id} \wedge t_1.\mathsf{user\_id}{=}t_3.\mathsf{user\_id} \wedge t_2.\mathsf{user\_id} = t_4.\mathsf{user\_id} \wedge t_1.\mathsf{score} = 0 \wedge t_2.\mathsf{score} = 0 \wedge t_1.\mathsf{review} = t_2.\mathsf{review} \wedge t_1.\mathsf{prod\_name} =$ "Disney" $\wedge \mathcal{M}(t_3.\mathsf{user\_name}, t_4.\mathsf{user\_name}) \wedge \mathcal{M}(t_3.\mathsf{address}, t_4.\mathsf{address}) \rightarrow \mathcal{M}_l(t_3, t_4)$. It explains why an ML model $\mathcal{M}_l$ identifies two users: $\mathcal{M}_l$ identifies t_3 and t_4 as the same person if they have similar names and similar addresses (determined via $\mathcal{M}$) and give an unusual low rating 0 and same review to a popular place like Disney.

Summary We find the following. (a) By supporting deep and collective ER, DMatch is 38% more accurate on average than all competitors over all datasets. (b) DMatch is 23% and 38% more accurate than ML-based and rule-based methods on real-life datasets, and it beats deep ER and collective ER by 21% and 32%, respectively. (c) DMatch is feasible on large dataset; it takes 505s on TPCH with 8 tables and 30M tuples, using 16 workers. On average it is 4.35 times faster than all the baselines on TPCH. (d) DMatch is parallelly scalable; when the number of workers increases from 4 to 32, it is on average 3.56 times faster. (e) The MQO technique improves the performance of DMatch by 43.4% on average.

VII. Conclusion

We have studied deep and collective ER with MRLs that may embed ML predicates, to improve the accuracy of entity resolution. We have settled its complexity, modeled it as a fixpoint computation, and provided optimization techniques to accelerate the computation. Our experimental study has shown that deep and collective ER is promising in practice.

One topic for future work is to extend MRLs to soft rules that return the probability of ER. Another topic is to develop incremental algorithm for deep and collective ER.

Acknowledgment

Deng and Lu are supported in part by National Key R&D Program of China (2021ZD0113903) and SKLSDE-2021ZX-11. Fan is supported in part by ERC 652976 and Royal Society Wolfson Research Merit Award WRM/R1/180014. Ping Lu is the corresponding author of the paper.

REFERENCES

[1] Mot tests and results, 2020. *https://data.gov.uk/dataset/e3939ef8-30c7-4ca8-9c7c-ad9475cc9b2f/anonymised-mot-tests-and-results.*

[2] Benchmark datasets for entity resolution, 2021. https://dbs.uni-leipzig.de/de/research/projects/.

[3] Sparklyclearn. https://github.com/david-siqi-liu/sparklyclean, 2021.

[4] TPCH. http://www.tpc.org/tpch/, 2021.

[5] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[6] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[7] G. Aggarwal, R. Motwani, and A. Zhu. The load rebalancing problem. *J. Algorithms*, 60(1):42–59, 2006.

[8] Y. Altowim and S. Mehrotra. Parallel progressive approach to entity resolution using MapReduce. In *ICDE*, pages 909–920, 2017.

[9] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. In *SIGMOD*, page 783–794, 2010.

[10] A. Arasu, C. Ré, and D. Suciu. Large-scale deduplication with constraints using Dedupalog. In *ICDE*, pages 952–963, 2009.

[11] Z. Bahmani and L. E. Bertossi. Enforcing relational matching dependencies with Datalog for entity resolution. In *FLAIRS*, 2017.

[12] Z. Bahmani, L. E. Bertossi, and N. Vasiloglou. ERBlox: Combining matching dependencies with machine learning for entity resolution. *IJAR*, 83:118–141, 2017.

[13] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *PODS*, pages 273–284, 2013.

[14] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *PODS*, pages 212–223, 2014.

[15] I. Beltagy, K. Erk, and R. J. Mooney. Probabilistic soft logic for semantic textual similarity. In *ACL*, pages 1210–1219, 2014.

[16] L. E. Bertossi, S. Kolahi, and L. V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. *Theory Comput. Syst.*, 52(3):441–482, 2013.

[17] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *TKDD*, 1(1), 2007.

[18] B. Bhattarai, H. Liu, and H. H. Huang. CECI: compact embedding cluster index for scalable subgraph matching. In *SIGMOD*, 2019.

[19] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *TACL*, 5:135–146, 2017.

[20] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, page 77–90, 1977.

[21] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *ICDT*, pages 56–70, 1997.

[22] X. Chu, I. F. Ilyas, and P. Koutris. Distributed data deduplication. *PVLDB*, 9(11):864–875, 2016.

[23] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.

[24] S. Das, A. Doan, P. S. G. C., C. Gokhale, P. Konda, Y. Govind, and D. Paulsen. The Magellan data repository. https://sites.google.com/site/anhaidgroup/projects/data.

[25] M. Ebraheem, S. Thirumuruganathan, S. R. Joty, M. Ouzzani, and N. Tang. Distributed representations of tuples for entity resolution. *PVLDB*, 11(11):1454–1467, 2018.

[26] V. Efthymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, and T. Palpanas. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Inf. Syst.*, 65:137–157, 2017.

[27] V. Efthymiou, G. Papadakis, K. Stefanidis, and V. Christophides. MinoanER: Schema-agnostic, non-iterative, massively parallel resolution of Web entities. In *EDBT*, pages 373–384, 2019.

[28] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1):1–16, 2007.

[29] W. Fan. Dependencies revisited for improving data quality. In *PODS*, page 159–170, 2008.

[30] W. Fan, H. Gao, X. Jia, J. Li, and S. Ma. Dynamic constraints for record matching. *VLDB J.*, 20(4):495–520, 2011.

[31] W. Fan, P. Lu, and C. Tian. Unifying logic rules and machine learning for entity enhancing. *Science China Information Sciences*, 2020.

[32] W. Fan, C. Tian, Y. Wang, and Q. Yin. Parallel discrepancy detection and incremental detection. *PVLDB*, 14(8):1351–1364, 2021.

[33] W. Fan, W. Yu, J. Xu, J. Zhou, X. Luo, Q. Yin, P. Lu, Y. Cao, and R. Xu. Parallelizing sequential graph computations. *TODS*, 2018.

[34] J. R. Finkel, T. Grenager, and C. D. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *ACL*, pages 363–370, 2005.

[35] L. Gagliardelli, G. Simonini, D. Beneventano, and S. Bergamaschi. SparkER: Scaling Entity Resolution in Spark. In *EDBT*, 2019.

[36] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[37] M. Grohe, T. Schwentick, and L. Segoufin. When is the evaluation of conjunctive queries tractable? In *STOC*, pages 657–666, 2001.

[38] S. Guo, X. L. Dong, D. Srivastava, and R. Zajac. Record linkage with uniqueness constraints and erroneous values. *PVLDB*, 2010.

[39] M. A. Hernndez and S. J. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, page 127–138, 1995.

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

[41] A. Jurek, J. Hong, Y. Chi, and W. Liu. A novel ensemble learning approach to unsupervised record linkage. *Inf. Syst.*, 71:40–54, 2017.

[42] A. Jurek and D. P. It pays to be certain: Unsupervised record linkage via ambiguity minimization. In *PAKDD*, pages 177–190, 2018.

[43] J. Kasai, K. Qian, S. Gurajada, Y. Li, and L. Popa. Low-resource deep entity resolution with transfer and active learning. In *ACL*, 2019.

[44] T. Kathuria and S. Sudarshan. Efficient and provable multi-query optimization. In *PODS*, pages 53–67, 2017.

[45] L. Kolb, A. Thor, and E. Rahm. Dedoop: Efficient deduplication with hadoop. *PVLDB*, pages 1878–1881, 2012.

[46] l. Koumarelas, T. Papenbrock, and F. Naumann. MDedup: Duplicate detection with matching dependencies. *PVLDB*, page 712–725, 2020.

[47] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theor. Comput. Sci.*, 71(1):95–132, 1990.

[48] Y. Li, J. Li, Y. Suhara, A. Doan, and W. Tan. Deep entity matching with pre-trained language models. *PVLDB*, 14(1):50–60, 2020.

[49] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD*, pages 169–178, 2000.

[50] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep learning for entity matching: A design space exploration. In *SIGMOD*, page 19–34, 2018.

[51] G. Papadakis, D. Skoutas, E. Thanos, and T. Palpanas. Blocking and filtering techniques for entity resolution: A survey. *ACM Comput. Surv.*, 53(2), 2020.

[52] G. Papadakis, J. Svirsky, A. Gal, and T. Palpanas. Comparative analysis of approximate blocking techniques for entity resolution. *PVLDB*, 2016.

[53] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, and M. Koubarakis. The return of JedAI: End-to-end entity resolution for structured and semi-structured data. *PVLDB*, pages 1950–1953, 2018.

[54] N. Peinelt, D. Nguyen, and M. Liakata. tBERT: Topic Models and BERT Joining Forces for Semantic Similarity Detection. In *ACL*, 2020.

[55] K. Qian, L. Popa, and P. Sen. Active learning for large-scale entity resolution. In *CIKM*, page 1379–1388, 2017.

[56] V. Rastogi, N. N. Dalvi, and M. N. Garofalakis. Large-scale collective entity matching. *PVLDB*, 4(4):208–218, 2011.

[57] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[58] F. Sadri and J. D. Ullman. The interaction between functional dependencies and template dependencies. In *SIGMOD*, 1980.

[59] P. Schirmer, T. Papenbrock, I. K. Koumarelas, and F. Naumann. Efficient discovery of matching dependencies. *TODS*, 45(3):13:1–13:33, 2020.

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

[61] G. Simonini, L. Gagliardelli, S. Bergamaschi, and H. V. Jagadish. Scaling entity resolution: A loosely schema-aware approach. *Inf. Syst.*, 83:145–165, 2019.

[62] Y. Tao. Massively parallel entity matching with linear classification in low dimensional space. In *ICDT*, 2018.

[63] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[64] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *NIPS*, 2017.

[65] S. E. Whang and H. Garcia-Molina. Joint entity resolution on multiple datasets. *VLDB J.*, 22(6):773–795, 2013.

[66] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD*, 2009.

[67] D. Zhang, L. Guo, X. He, J. Shao, S. Wu, and H. T. Shen. A graph-theoretic fusion framework for unsupervised entity resolution. In *ICDE*.

[68] P. Zhang, S. Bin, and G. Sun. Electronic word-of-mouth marketing in e-commerce based on online product reviews. *IJUNESST*, 8(8), 2015.