Parallel Discrepancy Detection and Incremental Detection

Wenfei Fan[1,2],   Chao Tian[3],   Yanghao Wang[1],   Qiang Yin[3]

[1]University of Edinburgh   [2]Shenzhen Institute of Computing Sciences   [3]Alibaba Group

{wenfei@inf.,yanghao.wang@}ed.ac.uk,{tianchao.tc,qiang.yq}@alibaba-inc.com

ABSTRACT

This paper studies how to catch duplicates, mismatches and conflicts in the same process. We adopt a class of entity enhancing rules that embed machine learning predicates, unify entity resolution and conflict resolution, and are collectively defined across multiple relations. We detect discrepancies as violations of such rules. We establish the complexity of discrepancy detection and incremental detection problems with the rules; they are both NP-complete and W[1]-hard. To cope with the intractability and scale with large datasets, we develop parallel algorithms and parallel incremental algorithms for discrepancy detection. We show that both algorithms are parallelly scalable, *i.e.,* they guarantee to reduce runtime when more processors are used. Moreover, the parallel incremental algorithm is relatively bounded. The complexity bounds and algorithms carry over to denial constraints, a special case of the entity enhancing rules. Using real-life and synthetic datasets, we experimentally verify the effectiveness, scalability and efficiency of the algorithms.

1 INTRODUCTION

Entity resolution (ER) and conflict resolution (CR) have been long-standing challenges of data quality. ER is to identify tuples that refer to the same real-life entity. CR is to resolve semantic inconsistencies pertaining to an entity. There has been a host of work on the topics, notably rule-based methods, *e.g.,* matching dependencies (MDs [37]) for ER, conditional functional dependencies (CFDs [38]) and denial constraints (DCs [15]) for CR, as well as machine learning (ML) models for ER [13, 61] and CR [25].

However, several questions remain to be addressed. (1) It has been recognized that neither rule-based methods nor ML models consistently outperform the other in practice. Is it possible to integrate the two in a uniform framework and take advantage of both? (2) Prior work often treats ER and CR as separate tasks and develops different rules for each, *e.g.,* MDs for ER and CFDs for CR. However, semantic conflicts and mismatched entities often coexist, and ER and CR inherently intervene with each other [32, 44, 75]. Can we catch conflicts and identify entities in the same process? We refer to the integrated process as *discrepancy detection*. (3) The prior rules are often defined on a single relation. However, it is known that to accurately identify entities, one needs to correlate

information from multiple relations [22]. What rules should we use to *collectively* detect discrepancies (ER and CR) across relations?

Example 1: E-commerce companies need to identify different accounts that belong to the same person for *e.g.,* fraud detection. To do this, one might be tempted to compare the associated ID card numbers. However, the same ID card is allowed to register multiple accounts at, *e.g.,* Alipay [3], so that one can register for her elderly parents using her own ID. Hence we have to check additional attributes, especially diversified ones that cannot be easily imitated *e.g.,* product preference. ML methods are needed for assessing and linking such attributes, which often involve long textual descriptions.

On the other hand, when the locations of two cell phone numbers are different at certain time ω, the company employs a logic rule such that the two accounts registered with these two phone numbers must be distinguished, *i.e.,* they belong to different persons, if their last login time both refer to ω, regardless of whether they have the same ID. We can see that the e-commerce practice needs *both logic rules and ML models.* Moreover, since accounts and phones are maintained in different tables, the rule is "collective".

Another issue is the uncertain reliability of time recorded. For instance, the last login time may even be earlier than the creation time of some accounts. To identify and distinguish accounts, catching conflicts in such attributes is a must in order to reduce false positives. This highlights the need for CR in the process of ER. □

To tackle these issues, a class of *entity enhancing rules* [48] has recently been studied, denoted by REEs, which subsume MDs, CFDs and DCs as special cases. As opposed to the prior data quality rules, REEs embed ML classifiers as predicates, support both ER and CR, and are collectively defined across relations. As will be seen in Section 2, REEs are able to resolve the issues of Example 1 at Alipay.

Challenges. To make practical use of REEs, several questions have to be settled. Is it harder to detect discrepancies using collective rules with embedded ML predicates? Is there an algorithm that catches duplicates, mismatches and conflicts in the same process, and scales with large datasets? Can we incrementally catch discrepancies online in response to updates to the data, and guarantee to perform better than re-examining the entire dataset starting from scratch?

Contributions & organization. We answer these questions.

(1) Complexity (Section 3). We show that the expressive power of REEs does come at a price. With REEs, the discrepancy detection and incremental detection problems become NP-complete. We also show that both problems are W[1]-hard, *i.e.,* there exists no fixed-parameter tractable algorithm for them. As a byproduct, we show that these complexity results already hold for DCs.

In contrast, the detection problem is in polynomial time (PTIME) with CFDs. Moreover, it is known that detection with a set of CFDs can be easily done with a single SQL query. This is no longer doable for REEs, since the ML predicates in REEs have to be encoded as

user-defined functions (UFDs); worse yet, the corresponding SQL query may take exponential time due to their collective nature.

(2) Parallel detection (Section 4). The intractability of the problem suggests the need for parallel algorithms to scale with real-life datasets. However, to the best of our knowledge, few parallel discrepancy detection algorithms are in place [24, 39, 56, 65, 68, 72], and among available ones, none guarantees the parallel scalability, _i.e.,_ the more processors are used, the faster the algorithms run.

We develop parallel algorithm PREEDet that is able to catch duplicates, mismatches and conflicts with REEs, and guarantees the parallel scalability. Moreover, the parallel scalability is robust: it remains intact regardless of what ML models are embedded in REEs.

Since discrepancy detection is often conducted on critical information, _e.g.,_ user credit records, we develop exact detection algorithms rather than approximate methods or heuristics.

(3) Parallel incremental detection (Section 5). As another approach to coping with large datasets, we study incremental discrepancy detection. Real-life data is constantly updated. It is too costly to recheck discrepancies starting from scratch in response to frequent updates. These highlight the need for incremental methods. The rationale is that in the real world, updates $\Delta \mathcal{D}$ to a large dataset $\mathcal{D}$ are typically small. When $\Delta \mathcal{D}$ are small, the changes to the set of discrepancies are often small and are much less costly to find than to recheck the discrepancies, by making use of previous computation.

In light of the intractability of incremental detection, we develop parallel algorithm PIncDet to incrementally detect discrepancies, by incrementalizing and parallelizing the batch detection algorithm. In addition, we show that the incremental algorithm is parallelly scalable and moreover, is bounded relative [42] to the batch algorithm, _i.e.,_ it incurs only necessary cost for incrementalization.

(4) Experimental study (Section 6). Using real-life datasets and a benchmark, we empirically verify the following. (a) REEs are able to catch discrepancies that the prior methods fail to detect. On average our detection algorithms outperform rule-based and ML methods by 33% and 36%, and ER and CR alone by 31% and 41%, respectively, in accuracy. (b) PREEDet and PIncDet scale well with the size $|\mathcal{D}|$ of datasets and the complexity of REEs. Using $n = 20$ machines and 100 REEs, they take 1047s and 15s on datasets $\mathcal{D}$ with 150 million tuples and $|\Delta \mathcal{D}| = 0.1\%|\mathcal{D}|$, respectively. (c) Incremental PIncDet is effective. On average it outperforms PREEDet by 20.4 times when $|\Delta \mathcal{D}| = 1\%|\mathcal{D}|$, and is faster even when $|\Delta \mathcal{D}|$ is up to 45% of $|\mathcal{D}|$. (d) Both PREEDet and PIncDet are parallelly scalable; compared to the sequential algorithms, on average they are 3.2 to 12.2 (resp. 3.1 to 9.9) times faster when n varies from 4 to 20.

Related work. We categorize the related work as follows.

Discrepancy detection. There has been a large body of work on ER, classified as follows. (1) ML-based, notably deep learning [34, 61], active learning [13, 64] and unsupervised learning [27, 58, 81]. (2) Matching rules, _e.g.,_ uniqueness constraints [52], MDs [21, 37], and datalog-like rules [14, 76]. (3) Hybrid, _e.g.,_ ERBlox [16] employs MDs for blocking and ML models for classification. Collective ER is proposed in [22]. ER is generally believed to take quadratic-time.

There has also been a host of work on CR, based on (1) logic rules, _e.g.,_ CFDs [26, 38, 40, 51] and DCs [15, 55]; and (2) various ML

models [11, 31, 53, 60, 66, 71, 73]. Discrepancy detection is in PTIME (cf. [9]), PTIME [38] and coNP-complete [19] with functional dependencies (FDs), CFDs and equality-generating dependencies (EGDs), respectively. To the best of our knowledge, no prior work has settled the complexity of detecting discrepancies with DCs.

There has also been work on data repairing [15, 20, 45], to fix discrepancies. This paper focuses on discrepancy detection, not repairing. In practice, most of our clients want us to detect discrepancies, but do not allow any destructive updates to their data.

This work differs from the prior work in the following. (1) We make a first attempt to detect duplicates, mismatches and semantic inconsistencies _in the same process_ using a uniform set of rules. (2) REEs are the first _logic rules_ for collective ER and CR that carry embedded ML predicates. (3) With such rules, we show that the detection and incremental detection problems are NP-complete and W[1]-hard as opposed to CFDs, and reveal what leads to the intractability. We also settle the complexity of these problems for DCs.

Parallel detection. Several parallel algorithms are in place to catch discrepancies in relations [24, 39, 56, 72] and graphs [47, 49]. However, most of them target either CR or ER, not both. Parallel CFD-violation detection (CR) was studied for horizontally or vertically partitioned relations [39]. Parallel ER was studied under, _e.g.,_ MapReduce [29, 56, 57] and MPC [72]. Blocking strategies [35] have been revisited for parallel ER [24, 68], to divide data into "independent" blocks such that pairwise comparison is only needed within each block. However, none of these ensures the parallel scalability. Over graphs, parallel detection algorithms have been studied for ER [36] and CR [47, 49]. These algorithms make use of the locality of data quality rules on graphs and do not apply to relations.

For incremental CR, [46] studies incremental CFD validation over relational data, and parallelly scalable algorithms have been developed to incrementally capture numeric discrepancies in graphs [47]. For ER, FastPath [74] incrementally processes query records based on similarity and a search index created offline. Another incremental rule-based method for ER is proposed in [77], with certain correctness guarantees. Closer to our work is [72], which gives a batch ER algorithm that is parallelly scalable for communication load over two relations; it is unknown whether [72] is parallelly scalable when computational cost is taken into account.

Our methods differ from the previous ones in the following.

(1) Despite the intractability, we propose the first parallel method for detecting duplicates, mismatches and semantic conflicts simultaneously across multiple relations. Moreover, we provide the first such algorithm with provable _parallel scalability_, when both computational and communication costs are considered.

(2) We provide the first parallel incremental algorithms for both ER and CR. These are the first algorithms guarantee _both parallel scalability and relative boundedness_. To our knowledge, these incremental and batch algorithms are also the first parallel algorithms for MDs, CFDs and DCs with the performance guarantees.

(3) We extend the blocking strategies [24, 68] to a hybrid virtual blocking method that takes inequality comparison into account, beyond conventional equality checking of attributes. The new method works on multiple relations and ensures the parallel scalability.

	id	phn	ID_card	create_time	last_login	payment	level	preference	purchase_log
r_1:	a_1	17788071668	420102199003072817	2015-01-13	2020-05-29	2000	12	toys, diapers	Lego, Merries
r_2:	a_2	13057705421	420102199003072817	2010-07-25	2010-05-11	50	3	clothing, makeup	Prada, Chanel
r_3:	a_2	13682228882	420102199003072817	2009-03-07	2010-05-11	1100	5	video games, toys	Mattel, Nintendo
r_4:	a_3	17876106113	610102199308194132	2013-10-21	2019-12-30	3500	10	electronic products	Apple, Sony
r_5:	a_4	15657853565	310101196010016754	2008-04-18	2018-06-10	120	4	clothing, makeup	Gucci, Dior
r_6:	a_5	18999906745	230106197507113089	2016-09-06	2018-11-10	2800	8	furniture	IKEA, Muji

Figure 1: Example ACCOUNT relation D_1

	id	number	time_tracked	location
r_7:	m_1	13057705421	2010-05-11	Macau
r_8:	m_2	13682228882	2010-05-11	Xi'an

Figure 2: Example MOBILE relation D_2

2 RULES FOR DISCREPANCY DETECTION

We start with a review of entity enhancing rules (REEs) [48].

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 τ_i. A database $\mathcal{D}$ of $\mathcal{R}$ is $(D_1, \ldots, D_m)$, where D_i is a relation of R_i ($i \in [1, m]$) [9]. 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.

<u>Predicates</u>. *Predicates* over $\mathcal{R}$ are defined as follows:

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

where $\otimes$ is a comparison operator $=, \neq, <, \leq, >, \geq$. Following tuple relational calculus [9], (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 $t.A \otimes c$, c is a constant in the domain of attribute A in R. (4) In $t.A \otimes s.B$, $t.A$ and $s.B$ are *compatible*, i.e., t (resp. s) is a tuple of some relation R (resp. R'), and $A \in R$ and $B \in R'$ have the same type. Moreover, (5) $\mathcal{M}$ is an ML classifier, $t[\bar{A}]$ and $s[\bar{B}]$ are vectors of pairwise compatible attributes of t and s, respectively.

Intuitively, $\mathcal{M}(t[\bar{A}], s[\bar{B}])$ can be any existing well-trained ML classifier for ER or CR, *e.g.,* [13, 61, 64], which returns true if $\mathcal{M}$ predicates that $t[\bar{A}]$ and $s[\bar{B}]$ are "associated" (*e.g.,* match), and false otherwise. We take $\mathcal{M}$ as a predicate, and refer to $\mathcal{M}$ as an *ML predicate*.

Rules. An REE φ over schema $\mathcal{R}$ is defined as

$$X \to e.$$

Here (1) X is a conjunction of predicates over $\mathcal{R}$, and (2) e is a predicate over $\mathcal{R}$ other than relation atoms. We refer to X and e as the *precondition* and *consequence* of φ, respectively.

Example 2: Consider database schema $\mathcal{R}_c$ with relation schemas ACCOUNT (id, phn, ID_card, create_time, last_login, payment, level, preference, purchase_log) and MOBILE (id, number, time_tracked, location). To identify and distinguish accounts as described in Example 1, we use the following two REEs.

(1) φ_1 : ACCOUNT$(t_1) \wedge$ ACCOUNT$(t_2) \wedge t_1$.ID_card=t_2.ID_card $\wedge$ $\mathcal{M}_r(t_1[\text{preference}], t_2[\text{preference}]) \to t_1$.id=$t_2$.id. ML model $\mathcal{M}_r$ checks whether two user portraits (preferences) are *close* enough.

(2) φ_2 : ACCOUNT$(t_1) \wedge$ ACCOUNT$(t_2) \wedge t_1$.last_login=t_2.last_login $\wedge$ MOBILE$(s_1) \wedge$ MOBILE$(s_2) \wedge t_1$.phn=s_1.number $\wedge t_2$.phn = s_2.number $\wedge t_1$.last_login = s_1.time_tracked $\wedge t_2$.last_login = s_2.time_tracked $\wedge s_1$.location $\neq s_2$.location $\to t_1$.id $\neq t_2$.id. This *collective* rule catches mismatched accounts across two relations.

In addition, we catch conflicts with REEs at Alipay.

(3) φ_3 : ACCOUNT$(t) \to t$.create_time $\leq t$.last_login. Clearly each account's last login time is later than its creation time.

(4) φ_4 : ACCOUNT$(t_1) \wedge$ ACCOUNT$(t_2) \wedge t_1$.create_time $\leq t_2$.create_time $\wedge t_1$.payment $\geq t_2$.payment $\to t_1$.level $\geq t_2$.level. That is, if one account is registered earlier and has a larger total amount of payment than another, then it should be at a higher level. This is how e-commerce platforms rate their users.

REEs can also use ML models for link prediction besides for similarity checking, and interpret ML prediction besides checking.

(5) φ_5 : ACCOUNT$(t_1) \wedge$ ACCOUNT$(t_2) \wedge \mathcal{M}_p(t_1[\text{purchase\_log}], t_2[\text{purchase\_log}]) \wedge \mathcal{M}_f(t_1, t_2) \to \mathcal{M}_r(t_1[\text{preference}], t_2[\text{preference}])$. Here $\mathcal{M}_r$ is the ML model used in φ_1, $\mathcal{M}_f$ is an ML model that predicts whether account owners of t_1 and t_2 are friends, using *all* attributes of t_1 and t_2, and $\mathcal{M}_p$ inspects the purchase history. This REE says that the owners of t_1 and t_2 have similar preferences if the two are friends and have similar purchase histories. It provides a possible interpretation of $\mathcal{M}_r$ on preference, in terms of a link prediction model $\mathcal{M}_f$ and another ML model $\mathcal{M}_p$ on purchase_log. □

REEs extend CFDs, DCs and MDs by embedding ML predicates. (1) CFDs [38] are REEs defined on a single relation, with equality $t.A = s.B$ and $t.A = c$. (2) DCs [15, 25] are REEs without ML predicates. (3) MDs [37] are REEs of the form $X \to e$, where X consists of two relation atoms $R_1(t_1)$ and $R_2(t_2)$, equality $x.A = y.B$ and similarity $t_1[\bar{A}_1] \approx t_2[\bar{A}_2]$ that can be carried out by ML predicates $\mathcal{M}(t_1[\bar{A}_1], t_2[\bar{A}_2])$, while e is t_1.id = t_2.id. Thus REEs can uniformly detect conflicts (CR), and catch duplicates and mismatches (ER).

Semantics. Consider a database $\mathcal{D}$ of schema $\mathcal{R}$. A *valuation* of tuple variables of an REE φ in $\mathcal{D}$, or simply *a valuation of φ*, is a mapping h that instantiates t in each relation atom $R(t)$ of φ with a tuple in the relation instance of schema 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 $R(t)$, $t.A \oplus c$ or $t.A \oplus s.B$, then $h \models p$ is interpreted as in tuple relational calculus following the standard semantics of first order logic. (2) If p is $\mathcal{M}(t[\bar{A}], s[\bar{B}])$, then $h \models p$ if the ML classifier $\mathcal{M}$ predicts true on $(h(t)[\bar{A}], h(s)[\bar{B}])$.

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 database $\mathcal{D}$ *satisfies* REE $\varphi = X \to e$ if for all valuations h of φ in $\mathcal{D}$, if $h \models X$ then $h \models e$.

A *violation* of φ in $\mathcal{D}$, also referred to as a *discrepancy*, is a valuation h of φ such that $h \models X$ but $h \not\models e$, *i.e.,* h witnesses that $\mathcal{D} \not\models \varphi$. We say that all the tuples involved in the discrepancy h are *potentially erroneous*, which can then be examined manually. For a set Σ of REEs, we denote by Vio$(\Sigma, \mathcal{D})$ the set of all violations of the REEs of Σ in $\mathcal{D}$, *i.e.,* $h \in$ Vio$(\Sigma, \mathcal{D})$ if h violates at least one REE in Σ.

Example 3: Continuing with Example 2, let database $\mathcal{D}_c = (D_1, D_2)$, where D_1 and D_2 are two relations of schemas ACCOUNT

Table 1: Notations

Notations	Definitions
$\mathcal{D} = (D_1, \ldots, D_n)$	database
$\mathcal{M}(t[\bar{A}], s[\bar{B}])$, φ, Σ	ML predicates, REE, a set of REEs
h	a valuation of an REE in a database
$\text{Vio}(\Sigma, \mathcal{D})$	violations of REEs Σ in database $\mathcal{D}$
$\Delta \mathcal{D}$	updates to database $\mathcal{D}$
$\Delta\text{Vio}(\Sigma, \mathcal{D}, \Delta\mathcal{D})$	changes to $\text{Vio}(\Sigma, \mathcal{D})$ by $\Delta\mathcal{D}$

and MOBILE, shown in Figures 1 and 2, respectively. Then there exists a violation in $\mathcal{D}_c$ of every REE from Example 2, *i.e.*, φ_1-φ_4 . For instance, the valuation h_1 of REE φ_1 that maps t_1 (resp. t_2) to r_1 (resp. r_3) violates φ_1, since r_1 and r_3 have the same ID card number and both like goods for kids but have different id attributes. Here the similarity of their preferences is determined by ML classifier $\mathcal{M}_r$. Similarly, one can find the violations of φ_2−φ_4.   □

The notations of the paper are summarized in Table 1.

3   COMPLEXITY BOUNDS

We next formalize the discrepancy detection problem and the incremental detection problem with REEs, and establish their complexity. It is known that the satisfiability and implication problems for REEs are NP-complete and Π_2^p-complete, respectively [48]. Taken together, these settle the complexity of important problems for REEs.

Detection. The *discrepancy detection problem* is as follows.

○ *Input*: A database schema $\mathcal{R}$, a set Σ of REEs over $\mathcal{R}$, and a database $\mathcal{D}$ of $\mathcal{R}$.
○ *Question*: Does $\text{Vio}(\Sigma, \mathcal{D}) \neq \emptyset$?

Intuitively, this is to study the complexity of discrepancy detection.

It is known that CR with CFDs is in PTIME [38], and ER takes quadratic-time on a single relation with simple rules. When it comes to detection with REEs, however, the problem becomes NP-complete. Here we assume *w.l.o.g.* that ML prediction (*i.e.,* testing with pre-trained $\mathcal{M}$) takes PTIME, as commonly found in practice.

Observe that the NP-completeness of the discrepancy detection problem concerns its *combined complexity*, *i.e.,* it takes both the dataset size $|\mathcal{D}|$ and the size $|\Sigma|$ of REEs as input; while for fixed size $|\Sigma|$, the problem is in PTIME. To better understand its intractability, we study its *parameterized complexity*, by treating $|\Sigma|$ as the parameter. A problem is fixed-parameter tractable (FPT) if it can be solved in $O(f(k) \cdot n^{O(1)})$ time, where k is the parameter, n is the input size and f is a computable function that depends only on k. As shown in the theorem below, the discrepancy detection problem for REEs is not FPT, assuming that FPT $\neq$ W[1]. That is, it is unlikely to find an efficient algorithm for discrepancy detection even when $|\Sigma|$ is small and $|\mathcal{D}|$ is large. As a result, we need to develop parallel algorithm to deal with large datasets $\mathcal{D}$.

Theorem 1: *The discrepancy detection problem is both* NP-*complete and* W[1]-*hard with parameter* $|\Sigma|$ *(1) for* REEs *and (2)* DCs.   □

Proof sketch: Discrepancy detection for both REEs and DCs is clearly in NP. For hardness, we give a reduction from the CLIQUE problem to discrepancy detection with DCs. The CLIQUE problem is to decide, given an undirected graph G and a natural number k, whether there is a k-clique in G. The CLIQUE problem is known to be both NP-complete and W[1]-complete with parameter k [33]. □

Incremental detection. Theorem 1 shows that discrepancy detection is costly in large datasets $\mathcal{D}$. Worse still, real-life data is constantly updated. This highlights the need for incremental discrepancy detection: we compute $\text{Vio}(\Sigma, \mathcal{D})$ once *offline*, and then incrementally compute $\text{Vio}(\Sigma, \mathcal{D} \oplus \Delta\mathcal{D})$ periodically *online* in response to updates $\Delta\mathcal{D}$ to $\mathcal{D}$, where $\mathcal{D} \oplus \Delta\mathcal{D}$ denotes $\mathcal{D}$ updated by $\Delta\mathcal{D}$.

When $\Delta\mathcal{D}$ are small, incremental detection is often more efficient than recomputing $\text{Vio}(\Sigma, \mathcal{D} \oplus \Delta\mathcal{D})$ starting from scratch, since the changes to $\text{Vio}(\Sigma, \mathcal{D})$ are often small as well in this case.

We consider *w.l.o.g.* $\Delta\mathcal{D}$ consisting of tuple insertions and deletions, which can simulate value modification. Denote by

$\Delta\text{Vio}^+(\Sigma, \mathcal{D}, \Delta\mathcal{D}) = \text{Vio}(\Sigma, \mathcal{D} \oplus \Delta\mathcal{D}) \setminus \text{Vio}(\Sigma, \mathcal{D})$,
$\Delta\text{Vio}^-(\Sigma, \mathcal{D}, \Delta\mathcal{D}) = \text{Vio}(\Sigma, \mathcal{D}) \setminus \text{Vio}(\Sigma, \mathcal{D} \oplus \Delta\mathcal{D})$,
$\Delta\text{Vio}(\Sigma, \mathcal{D}, \Delta\mathcal{D})\ = (\Delta\text{Vio}^+(\Sigma, \mathcal{D}, \Delta\mathcal{D}), \Delta\text{Vio}^-(\Sigma, \mathcal{D}, \Delta\mathcal{D}))$,

the set of new discrepancies inflicted by $\Delta\mathcal{D}$, the set of discrepancies removed by $\Delta\mathcal{D}$ and their combination, respectively. Then

$\text{Vio}(\Sigma, \mathcal{D} \oplus \Delta\mathcal{D}) = \text{Vio}(\Sigma, \mathcal{D}) \oplus \Delta\text{Vio}(\Sigma, \mathcal{D}, \Delta\mathcal{D})$.

The *incremental discrepancy detection problem* is stated as follows.

○ *Input*: Schema $\mathcal{R}$, REEs Σ and database $\mathcal{D}$ as in discrepancy detection, and moreover, $\text{Vio}(\Sigma, \mathcal{D})$ and updates $\Delta\mathcal{D}$ to $\mathcal{D}$.
○ *Question*: Does $\Delta\text{Vio}(\Sigma, \mathcal{D}, \Delta\mathcal{D}) \neq \emptyset$?

It is to settle the complexity of computing changes to $\text{Vio}(\Sigma, \mathcal{D})$.

The problem is also intractable, even when the sizes $|\mathcal{D}|$ and $|\Delta\mathcal{D}|$ of $\mathcal{D}$ and $\Delta\mathcal{D}$ are both predefined and fixed. Worse still, it is W[1]-hard when $|\Sigma|$ is taken as the parameter. As a byproduct, we show that the same holds for DCs on a single relation, for which the complexity is not yet settled to the best of our knowledge.

Theorem 2: *The incremental discrepancy detection problem (1) is* NP-*complete for* REEs *and* DCs *and remains* NP-*hard even when both* $|\mathcal{D}|$ *and* $|\Delta\mathcal{D}|$ *are constants; and (2) it is* W[1]-*hard for* REEs *and* DCs *by taking* $|\Sigma|$ *as the parameter.*   □

Proof sketch: (1) The problem is clearly in NP; we verify its NP-hardness by reduction from the 3-COLORABILITY problem [50], which asks whether a given graph is 3-colorable. The reduction uses $\mathcal{D}=\emptyset$ and $|\Delta\mathcal{D}|=12$. (2) The W[1]-hardness is verified by reduction from the discrepancy detection problem for DCs, by Theorem 1. □

4   PARALLEL DISCREPANCY DETECTION

In light of the intractability (Theorem 1), to scale with large datasets we provide an algorithm with parallel scalability. We first review the notion (Section 4.1), and give a sequential algorithm (Section 4.2). We then develop a parallel algorithm with the property (Section 4.3).

4.1   Parallel Scalability Revisited

To characterize the effectiveness of parallel algorithms for discrepancy detection, we revisit the notion of *parallel scalability* that was introduced in [59] and has been widely used in practice.

Consider a sequential (single-machine) algorithm $\mathcal{A}$ that, given a database $\mathcal{D}$ and a set Σ of REEs, computes $\text{Vio}(\Sigma, \mathcal{D})$. Denote its worst-case runtime as $t(|\mathcal{D}|, |\Sigma|)$. We say that a parallel algorithm $\mathcal{A}_p$ for discrepancy detection is *parallelly scalable relative to* $\mathcal{A}$ if its running time by using n processors can be expressed as:

$$T(|\mathcal{D}|, |\Sigma|, n) = \tilde{O}\left(\frac{t(|\mathcal{D}|, |\Sigma|)}{n}\right),$$

where the notation $\tilde{O}$ hides $\log(n)$ factors (see, *e.g.,* [78]).

Intuitively, parallel scalability guarantees speedup of $\mathcal{A}_p$ *relative to* a "yardstick" sequential $\mathcal{A}$. Such an algorithm $\mathcal{A}_p$ is able to "linearly" reduce the sequential cost of $\mathcal{A}$ when more processors are used. That is, the more processors are used, the faster $\mathcal{A}_p$ is. Hence $\mathcal{A}_p$ can scale with large $\mathcal{D}$ by increasing n when needed.

Similarly, we say that a parallel algorithm $\mathcal{A}_p$ for incremental detection is *parallelly scalable relative to a sequential incremental algorithm* $\mathcal{A}$ if its cost with n processors can be expressed as:

$$T(|\mathcal{D}|, |\Delta\mathcal{D}|, |\Sigma|, n) = \tilde{O}\big(\frac{t(|\mathcal{D}|, |\Delta\mathcal{D}|, |\Sigma|)}{n}\big),$$

where $t(|\mathcal{D}|, |\Delta\mathcal{D}|, |\Sigma|)$ is the worst-case running time of $\mathcal{A}$.

4.2 Sequential Algorithm

We start with a sequential algorithm for discrepancy detection with a single REE φ, denoted as REEDet. For a set Σ of REEs, we repeatedly run REEDet with each rule in Σ, also denoted by REEDet.

Given a database $\mathcal{D}$ of schema $\mathcal{R}$ and an REE $\varphi = X \rightarrow e$ over $\mathcal{R}$, REEDet finds the violation set $\mathrm{Vio}(\{\varphi\}, \mathcal{D})$ in two steps.

(1) REEDet first generates a *detection query* Q_φ defined as:
$$Q_\varphi = \sigma_{P_X \wedge \hat{e}}(R_1 \times \cdots \times R_m),$$
where (a) P_X is the conjunction of all predicates in X except those relation atoms, and $\hat{e}$ is the negation of the predicate e; and (b) $R_1, \ldots, R_m$ are relation atoms that appear in X, and each R_i corresponds to exactly one tuple variable from φ.

(2) It next evaluates $Q_\varphi(\mathcal{D})$ by translating Q_φ into an SQL statement, which is then executed by a DBMS. Here the ML predicates are implemented via user-defined functions, which can incorporate the logic of existing validity checking methods for ML classifiers [58].

It is easy to see the correctness of REEDet, *i.e.*, a valuation h can be derived from the answer $Q_\varphi(\mathcal{D})$ if and only if $h \in \mathrm{Vio}(\{\varphi\}, \mathcal{D})$. Its runtime is dominated by evaluating Q_φ over $\mathcal{D}$. Although DBMS exploits tricks to speed up query evaluation and ML predicates are PTIME binary operations after ML models are trained, it still takes $O(|\mathcal{D}|^{|Q_\varphi|})$ time in the worst case to compute $Q_\varphi(\mathcal{D})$, since Q_φ is a conjunctive query [62]. When given a set Σ of REEs, the total cost of REEDet is $O(\sum_{\varphi \in \Sigma} |\mathcal{D}|^{|Q_\varphi|}) \leq O(|\mathcal{D}|^{|\Sigma|})$ in the worst case.

Note that REEDet is able to plug in any ML classifier $\mathcal{M}$ for ER or CR without affecting the complexity bound, as long as pre-trained $\mathcal{M}$ runs in PTIME as commonly found in practice.

4.3 Parallel Algorithm

We develop a parallel algorithm PREEDet for discrepancy detection and show that it is parallelly scalable relative to REEDet. Given a database $\mathcal{D}$ and a set Σ of REEs, it computes $\mathrm{Vio}(\Sigma, \mathcal{D})$ in parallel.

Challenges. One naturally wants to have a blocking strategy [17, 24] that divides $\mathcal{D}$ into independent blocks *w.r.t.* the rules, such that discrepancy detection can be carried out on each block in parallel, without interaction between blocks. However, it is challenging to make a blocking strategy parallelly scalable for REEs.

(1) Multiple relations. Most existing blocking strategies are designed for a single relation [24] and consider equality predicates only [17, 18]. These do not apply to REEs since REEs are collectively defined across multiple relations and moreover, carry not only equality predicates, but also inequality and ML predicates.

Input: A database $\mathcal{D}$, processors $S_1, \ldots, S_n$, and a set Σ of REEs.
Output: The set $\mathrm{Vio}(\Sigma, \mathcal{D})$ of all violations.
1. $\mathcal{W} := \emptyset$; $\mathcal{W}^L := \emptyset$;
2. **for each** REE $\varphi \in \Sigma$ **do**   /*executed at coordinator S_c */
3. $\mathcal{W} := \mathcal{W} \cup \mathrm{HPartition}(\varphi, \mathcal{D})$;
4. retrieve a set $\mathcal{W}^H$ of heavy blocks from $\mathcal{W}$;
5. distribute $\mathcal{W}$ evenly across n processors in parallel;
6. **for each** $w \in \mathcal{W}^H$ **do**   /*reduce skewness*/
7. $W' := \mathrm{WDivide}(w, \mathcal{D}_w)$; $\mathcal{W}^L := \mathcal{W}^L \cup W'$;
8. distribute $\mathcal{W}^L$ evenly to n processors and refine the partition;
9. shuffle tuples in $\mathcal{D}$ based on the partition of blocks;
10. **for each** $w \in (\mathcal{W} \backslash \mathcal{W}^H) \cup \mathcal{W}^L$ **do** /*run on n processors in parallel*/
11.   filter local data according to w to get $\mathcal{D}_w$;
12. $\mathrm{Vio}(w) := \mathrm{REEDet}(\varphi_w, \mathcal{D}_w)$;
13. **return** $\bigcup_w \mathrm{Vio}(w)$ as $\mathrm{Vio}(\Sigma, \mathcal{D})$;

Procedure WDivide
Input: A heavy block $w = (\bar{c}, \varphi)$ and its dataset $\mathcal{D}_w = (D_w^1, \ldots, D_w^p)$.
Output: A set $\mathcal{W}'$ of blocks.
1. identify dividable relations $D_w^1, \ldots, D_w^r$ from $\mathcal{D}_w$;
2. $\mathrm{BM} := \max(\sqrt[r]{|D_w^1| \times \cdots \times |D_w^r|/n^2}, 1)$;
3. **for each** dividable relation D_w^i **do**
4. $D_w^i.\mathrm{div} := \lfloor |D_w^i|/\mathrm{BM} \rfloor$; $D_w^i.\mathrm{extn} := 1/D_w^i.\mathrm{div}$;
5.   **if** $|D_w^i|$ mod $\mathrm{BM} = 0$ **then** $D_w^i.\mathrm{extn} := 0$;
6. map tuples in dividable relations to $\prod_{i \in [1,r]} D_w^i.\mathrm{div}$ partitions *s.t.* each partition holds at most $(1 + D_w^i.\mathrm{extn}) \times \mathrm{BM}$ tuples from D_w^i;
7. extend partitions with all tuples in undividable relations;
8. **return** the set $\mathcal{W}'$ of blocks derived from the resulting partitions;

Figure 3: Parallel algorithm PREEDet

(2) Multiple rules. To handle a set Σ of REEs, a brute-force approach is to repeatedly partition $\mathcal{D}$ into blocks, one for each REE in Σ. This would require partitioning $\mathcal{D}$ and physically moving its data multiple times, incurring redundant communication and computation.

(3) Load balancing. The computational costs within some blocks, *i.e.*, running REEDet, may be significantly larger than others due to heavy hitters [18]. These lead to skewness in parallel processing and slow down the process, which hampers the parallel scalability.

Algorithm overview. The parallel algorithm PREEDet works in four stages to tackle the challenges above, as shown in Figure 3.

(1) PREEDet first computes a set $\mathcal{W}$ of "virtual" blocks by a range-based partition scheme (lines 1-5). Each block corresponds to a work unit, and different work units will work in parallel.

(2) PREEDet then eliminates "heavy" blocks to balance the workload and reduce skewness. It decomposes each heavy block w into a set of new blocks and re-distributes them (lines 6-8).

(3) PREEDet distributes $\mathcal{D}$ to n processors according to the partition of blocks (line 9). By the virtual nature of blocks, it sends each tuple to a processor *at most once* when shuffling the raw data of $\mathcal{D}$, even if it is needed by multiple blocks (see details below).

(4) After this, PREEDet executes REEDet (Section 4.2) on the work units deduced from virtual blocks, in parallel at all processors (lines 10-12). Since the blocks are independent, it simply returns the union of results at different processors as $\mathrm{Vio}(\Sigma, \mathcal{D})$ (line 13).

Intuitively, PREEDet disjointly partitions the workload of detection into independent blocks and work units. Therefore, no cross-block violations exist and only the raw data in $\mathcal{D}$ is shuffled.

We next show how to build up blocks with the range-based partitioning strategy, and how to break heavy blocks for load balancing.

Block construction. PREEDet divides the overall computation of $\text{Vio}(\Sigma, \mathcal{D})$ into small tasks, referred to as *work units* for parallel processing. Each work unit includes a subset $\mathcal{D}_w$ of $\mathcal{D}$ and an REE rule $\varphi_w \in \Sigma$; it is to execute REEDet on $\mathcal{D}_w$ with φ_w. Denote by $\bar{c}$ the condition by which $\mathcal{D}_w$ is selected from the database $\mathcal{D}$, which is actually an assembled cell of ranges (see below). We refer to each pair $w = (\bar{c}, \varphi_w)$ as a virtual *block w.r.t.* REE φ_w. There is a one-to-one mapping between the blocks and work units, since the subset $\mathcal{D}_w$ in each unit can be derived by applying its condition $\bar{c}$. The blocks are *virtual* because they denote the conditions for finding $\mathcal{D}_w$ for work units rather than storing the raw data.

Range-based partition. For each REE $\varphi \in \Sigma$, PREEDet builds a set of virtual blocks *w.r.t.* φ, by utilizing a range-based partition procedure HPartition (lines 2-3). It extends HyperCube algorithm [17, 18] proposed for parallelly answering conjunctive queries in *one communication round*. Besides equality checking considered in HyperCube, we also inspect *inequality and ML predicates* in the REEs.

For an attribute A in a relation D, we define a partition function $f_A(x) : x \rightarrow [1, k_A]$ that maps tuples in D into k_A fragments based on the values of their A-attribute. Here k_A is an integer, referred as the *share* of function f_A. To handle both equality and inequality checking, we employ range-based partition functions defined as

$$f_A(x) = \max(\lceil \frac{x.A - \min(D[A])}{|\text{dom}(D[A])|} * k_A \rceil, 1),$$

where $\min(D[A])$ is the minimum A-value in D and $|\text{dom}(D[A])|$ denotes the size of the active domain of A-values in D. That is, $f_A(x)$ divides the domain of A-values in D into k_A segments evenly, and each integer in $[1, k_A]$ refers to a range. Given an REE φ, denote by $\{A_1, \ldots, A_{m'}\}$ the set of attributes that appear in the predicates of φ. Let $k_{A_1}, \ldots, k_{A_{m'}}$ be the shares of partition functions $f_{A_1}, \ldots, f_{A_{m'}}$, respectively. Then a subset $\mathcal{D}_w$ of $\mathcal{D}$ can be identified by a tuple $(p_1, \ldots, p_{m'}) \in [k_{A_1}] \times \cdots \times [k_{A_{m'}}]$, where tuples in $\mathcal{D}_w$ are such selected that their A_i-attribute values are covered by the range to which integer p_i corresponds in function f_{A_i} ($i \in [1, m']$). We denote by $\bar{c}$ the *assembled cell*, *i.e.*, the combination of ranges deduced from $(p_1, \ldots, p_{m'})$, which is also the condition in virtual blocks.

With range-based partition functions, we are able to prune *invalid* assembled cells (*i.e.*, conditions) and stop further construction of their corresponding virtual blocks and work units, which do not contribute to the output of PREEDet. We say that an assembled cell $\bar{c}$ is *valid w.r.t.* an REE $\varphi = X \rightarrow e$, if the dataset $\mathcal{D}_w$ identified via $\bar{c}$ contains possible tuples that can form valuations satisfying X and the negation of e. It is called *invalid* otherwise. For instance, if $\bar{c}$ has the range $[1, 10]$ (resp. $[30, 50]$) for attribute A (resp. B) of relation D_1 (resp. D_2) and if there exists $t.A > s.B$ in X such that t (resp. s) is a tuple variable of the schema of D_1 (resp. D_2), then $\bar{c}$ is not valid *w.r.t.* φ.

Plugging in ML-based blocking. The range-based partition scheme can incorporate existing blocking strategies B_M developed for ML classifiers M, *e.g.*, those in [12, 23, 54]. That is, for each assembled cell $\bar{c}$ that includes attributes in an ML predicate $M(t[\bar{A}], s[\bar{B}])$, we revise their ranges by dropping the parts that do not comply with B_M. For example, FisherDisjunctive [54], an ML classier for entity resolution, deduces blocking schemes in the form of disjunctions

of terms (*i.e.*, blocking predicates). We validate such disjunctions using the ranges in $\bar{c}$ and remove unsatisfied subranges. The ranges tailored for ML blocking help us reduce invalid assembled cells. When no ML-based blocking scheme is available, the original assembled cells are used for further pairwise checking.

Capitalizing on the range-based partition scheme, procedure HPartition computes the set $\mathcal{W}_\varphi$ of virtual blocks *w.r.t.* each REE φ, at a designated processor S_c (coordinator in Figure 3). It generates a partition function f_{A_i} for each attribute A_i in φ and sets the share k_{A_i} to be $\lfloor \sqrt[m']{n^2} \rfloor$ by default, where m' is the number of attributes in φ. When $\lfloor \sqrt[m']{n^2} \rfloor = 1$, HPartition arbitrarily selects as many attributes as possible from φ and increases their shares to 2, while guaranteeing that the product of the shares does not exceed n^2. After revising the ranges with ML-based blocking scheme, HPartition checks whether each assembled cell $\bar{c}$ built with ranges is valid *w.r.t.* φ, and returns the pairings of valid ones and REE φ as blocks.

<u>*Remark*</u>. The range-based blocking scheme can naturally partition numeric attribute values and dates. For other attribute types, *e.g.*, string and category, we apply range-based blocking after enforcing a lexicographical order [67] on the values. For attribute types to which this is not applicable, one can adopt hash partitioning [63].

Example 4: Recall REEs $\varphi_1 - \varphi_4$ from Example 2 and database $\mathcal{D}_c$ from Example 3. Assume that we have $n = 8$ processors. We show how HPartition generates virtual blocks with range-based partition.

Take REE φ_4 as an example, which carries two tuple variables t_1 and t_2 of the same schema ACCOUNT. As such, HPartition collects six attributes from φ_4, and distinguishes different occurrences of the same attributes for t_1 and t_2. It then decides partition functions by dividing the domain of attribute values in relation D_1 into two ranges, *i.e.*, all the shares of functions are 2. For instance, the domain of payment is partitioned into 50 to 1775 and 1776 to 3500. By definition, $8^2 = 64$ candidate assembled cells are built with the ranges above. However, among them only 27 are valid *w.r.t.* φ_4, since an assembled cell $\bar{c}$ is invalid if it $h(t_1).\text{create\_time} > h(t_2).\text{create\_time}$ or $h(t_1).\text{payment} < h(t_2).\text{payment}$ or $h(t_1).\text{level} \geq h(t_2).\text{level}$ for each valuation h computed in the work unit deduced from $\bar{c}$. As an example, no valid assembled cell can be built when ranges $[50, 1775]$ and $[1776, 3500]$ are enforced on payment of t_1 and t_2, respectively. Procedure HPartition drops invalid ones and only creates 27 virtual blocks *w.r.t.* φ_4. Along the same lines, the blocks *w.r.t.* $\varphi_1 - \varphi_3$ are constructed using valid assembled cells. □

Reducing skewness. After all the virtual blocks are computed, algorithm PREEDet identifies "heavy" blocks whose corresponding work units may introduce skewness. Let $D_w^1, \ldots, D_w^p$ denote the relations in the dataset $\mathcal{D}_w$ derived from the assembled cell $\bar{c}$ of a virtual block $w = (\bar{c}, \varphi)$, sorted in the descending order of their sizes. Each one has a corresponding relation atom in φ. We say that w is *heavy* if $\prod_i |D_w^i| > (|\mathcal{D}|/n)^{|Q_\varphi|}$, where Q_φ is the detection query of φ (see Section 4.2). Here $\prod_i |D_w^i|$ represents the worst-case cost when evaluating the work unit deduced from w. To improve load balancing, PREEDet uses procedure WDivide to split heavy blocks, also illustrated in Figure 3. In fact, when dividing the evaluation, *i.e.*, the Cartesian product of relations in $\mathcal{D}_w$ among n processors, the optimal parallel computational cost is $\prod_i |D_w^i|/n$, *i.e.*, the computa-

tion is evenly partitioned. Moreover, by the inequality of arithmetic and geometric means [69], the optimal communication cost for such partitioning is achieved when every processor receives the same amount of tuples from each relation. In light of this, procedure WDivide ensures that the numbers of tuples received by each partition from different relations are *as close as possible*.

More specifically, WDivide first extracts *dividable* relations $D_w^1, \ldots, D_w^r$ from the input dataset $\mathcal{D}_w$ such that they have the top-r largest sizes, $r \leq 5$ and $|D_w^i|/\text{BM} > 1$ for $i \in [1, r]$ (line 1), where BM refers to a benchmark amount of $\sqrt[r]{|D_w^1| \times \cdots \times |D_w^r|/n^2}$ or 1 (line 2). Here r is bounded by 5 since most data quality rules involve at most three relations [43, 55, 66]. It next computes a division factor $D_w^i.\text{div}$ and an extension value $D_w^i.\text{extn}$ for each dividable D_w^i based on its size and BM (lines 3-5). They represent the number of fragments that D_w^i should be divided into and the percentage of an additional BM many tuples from D_w^i that need to be allocated to each fragment, respectively. Then it maps the tuples to partitions such that every partition contains at most $(1 + D_w^i.\text{extn}) \times \text{BM}$ tuples from each dividable D_w^i (line 6). This is conducted by sequentially scanning the sorted tuples of each dividable relation, and establishing the mapping from fragments to r-dimensional tuples, similar to the practice of the range-based partition scheme. It finally extends partitions by replicating tuples from the remaining *undividable* relations and returns the virtual blocks deduced from the partitions (lines 7-8). The blocks are then evenly distributed among the processors (line 8 in PREEDet).

The problem of finding partitions for Cartesian product with minimum computational and communication costs can be reduced to the integer programming problem, which is NP-complete [50]. Nevertheless, the simple PTIME skewness reduction strategies adopted by algorithm PREEDet have the following property.

Lemma 3: PREEDet *splits heavy blocks with computational cost of $c_1 \cdot \text{CP}_{\text{opt}}$, and the communication cost for handling dividable relations is $c_2 \cdot \text{CM}_{\text{opt}}$, where c_1 and c_2 are constants, CP_{opt} and CM_{opt} are the optimal cost for computation and communication, respectively.* □

Partition refinement. In the presence of existing cost models for answering queries Q_φ, PREEDet refines the partition of blocks to make the load better balanced (line 8). It assigns a *load* $C(Q_\varphi, D_w)$ to each block $w = (\bar{c}, \varphi_w)$, in which C denotes a given cost model, *e.g.*, the estimation of join result sizes [70]. The load indicates the cost for evaluating Q_φ (*i.e.*, applying REE φ) over D_w. Then PREEDet redistributes the blocks *w.r.t.* φ via a greedy strategy for solving the *minimum makespan problem* [10]. That is, each time it greedily reallocates an unvisited block to a processor that has the minimum total load of the blocks *w.r.t.* φ, until all blocks are inspected. In fact, it is a 2-approximation algorithm for finding optimal distribution.

Data shuffling. After all the heavy blocks are decomposed and the block partition is refined in regards to certain input cost models, algorithm PREEDet uniformly shuffles the raw data of database $\mathcal{D}$, *i.e.*, each processor S_i fetches its data guided by the assembled cells in the virtual blocks it receives (line 9 of PREEDet). In this way, the same data will be merged and transmitted to S_i *only once* even if it is within the ranges of different assembled cells, reducing the total communication cost when processing multiple REEs.

Example 5: Continuing with Example 4, PREEDet finds that there are heavy virtual blocks *w.r.t.* REE φ_4. For instance, due to the skewed distribution of attribute values in relation D_1, tuples r_2, r_3 and r_5 are contained in the same dataset $\mathcal{D}_w$ deduced from a virtual block $w = (\bar{c}, \varphi_4)$. It hence splits the Cartesian product of these tuples into nine different partitions guided by the required number of tuples. Here each partition could have at most two tuples and all relations are dividable. The virtual blocks of such resulting partitions are also evenly allocated among the processors.

Algorithm PREEDet transfers the raw data according to the selection condition in the virtual blocks generated; here tuples r_1 and r_4 (which finally form a violation of φ_4) will be sent to a processor S_i holding the block with range [1776, 3500] for attribute payment of both tuple variables t_1 and t_2. They are sent to S_i only once even if they are required by blocks *w.r.t.* other REEs. □

We now show the parallel scalability of algorithm PREEDet.

Theorem 4: PREEDet *is parallelly scalable relative to* REEDet. □

Proof sketch: We show that PREEDet is parallelly scalable relative to REEDet for a single REE φ in Σ, even if φ contains ML predicates. Then the theorem follows because (a) the computational cost of PREEDet for multiple REEs is bounded by the sum of the cost on each REE; and (b) the communication cost is bounded by $O(|\mathcal{D}|)$, and thus is subsumed by computational cost when $n \ll |\mathcal{D}|$, since the cost of sequential REEDet with φ is $O(|\mathcal{D}|^{|Q_\varphi|})$. □

Remark. As indicated in the proof, the parallel scalability is robust to different ML models. Moreover, one can use PREEDet to detect discrepancies with CFDs, MDs and DCs, with parallel scalability.

5 PARALLEL INCREMENTAL DETECTION

Theorem 2 tells us that incremental discrepancy detection is also intractable. Hence we next develop a parallelly scalable incremental algorithm PIncDet that computes the changes $\Delta\text{Vio}(\Sigma, \mathcal{D}, \Delta\mathcal{D})$.

Analogous to Section 4, we start with a sequential incremental algorithm IncDet (Section 5.1). We then present parallelly scalable PIncDet and show that it is bounded relative to IncDet (Section 5.2).

5.1 Sequential Incremental Algorithm

Observe that the set $\text{Vio}(\Sigma, \mathcal{D})$ of violations has a monotone property *w.r.t.* $\mathcal{D}$, *i.e.*, $\text{Vio}(\Sigma, \mathcal{D}_1) \subseteq \text{Vio}(\Sigma, \mathcal{D}_1 \cup \mathcal{D}_2)$. In light of this, by treating $\mathcal{D}$ as $\mathcal{D}_1$ and updates $\Delta\mathcal{D}$ as $\mathcal{D}_2$, we can compute $\Delta\text{Vio}(\Sigma, \mathcal{D}, \Delta\mathcal{D})$ by accumulating the small sets of violations contributed by tuples in $\Delta\mathcal{D}$. Here we say that a violation $h \in \text{Vio}(\Sigma, \mathcal{D}_1 \cup \mathcal{D}_2)$ is *contributed by $\mathcal{D}_2$* if and only if h includes at least one tuple from $\mathcal{D}_2$. We denote by $\text{CrtVio}(\Sigma, \mathcal{D}_1, \mathcal{D}_2)$ the set of violations in $\text{Vio}(\Sigma, \mathcal{D}_1 \cup \mathcal{D}_2)$ that are contributed by $\mathcal{D}_2$.

Let $\Delta\mathcal{D}^+$ and $\Delta\mathcal{D}^-$ be the set of tuple insertions and deletions in $\Delta\mathcal{D}$, respectively. The lemma below shows that the newly introduced discrepancies in $\Delta\text{Vio}^+(\Sigma, \mathcal{D}, \Delta\mathcal{D})$ and the removed ones in $\Delta\text{Vio}^-(\Sigma, \mathcal{D}, \Delta\mathcal{D})$ are contributed by $\Delta\mathcal{D}^+$ and $\Delta\mathcal{D}^-$, respectively.

Lemma 5: *(1)* $\Delta\text{Vio}^+(\Sigma, \mathcal{D}, \Delta\mathcal{D}) = \text{CrtVio}(\Sigma, \mathcal{D} \setminus \Delta\mathcal{D}^-, \Delta\mathcal{D}^+)$; *and (2)* $\Delta\text{Vio}^-(\Sigma, \mathcal{D}, \Delta\mathcal{D}) = \text{CrtVio}(\Sigma, \mathcal{D}, \Delta\mathcal{D}^-)$. □

In fact, we can compute $\text{CrtVio}(\Sigma, \mathcal{D}_1, \mathcal{D}_2)$ by invoking algorithm REEDet of Section 4.2 on the composite databases of $\mathcal{D}_1$

and $\mathcal{D}_2$. Given two databases $\mathcal{D}_1$ and $\mathcal{D}_2$ of the same schema $\mathcal{R}$ that pertains to the relation atoms $R_1, \ldots, R_m$ in Σ, a database $\tilde{\mathcal{D}} = (\tilde{D}_1, \ldots, \tilde{D}_m)$ is called *a composite database* of $\mathcal{D}_1$ and $\mathcal{D}_2$ if either $\tilde{D}_i \in \mathcal{D}_1$ or $\tilde{D}_i \in \mathcal{D}_2$ for each $i \in [1, m]$. Denote by $\mathrm{Cmp}(\mathcal{D}, \mathcal{D}')$ the set of all composite databases of $\mathcal{D}$ and $\mathcal{D}'$. Let $\mathbb{D} = \mathrm{Cmp}(\mathcal{D}_1, \mathcal{D}_2)$. Then one can easily verify the following.

$$\mathrm{CrtVio}(\Sigma, \mathcal{D}_1, \mathcal{D}_2) = \bigcup \{\mathrm{Vio}(\Sigma, \tilde{\mathcal{D}}) \mid \tilde{\mathcal{D}} \in \mathbb{D}, \tilde{\mathcal{D}} \cap \mathcal{D}_2 \neq \emptyset\}. \quad (1)$$

Based on Lemma 5 and Equation (1), we now present IncDet.

Algorithm IncDet. Given a database $\mathcal{D}$, the previous set $\mathrm{Vio}(\Sigma, \mathcal{D})$ of violations and updates $\Delta\mathcal{D}$ to $\mathcal{D}$, the sequential IncDet computes updates $\Delta\mathrm{Vio}(\Sigma, \mathcal{D}, \Delta\mathcal{D})$ to $\mathrm{Vio}(\Sigma, \mathcal{D})$ as follows. (1) It first computes the set $\mathrm{Cmp}(\mathcal{D} \setminus \Delta\mathcal{D}^-, \Delta\mathcal{D}^+)$ of composite databases. For each such $\mathcal{D}' \in \mathrm{Cmp}(\mathcal{D} \setminus \Delta\mathcal{D}^-, \Delta\mathcal{D}^+)$ that $\mathcal{D}' \cap \Delta\mathcal{D}^+ \neq \emptyset$, it applies REEDet to get $\mathrm{Vio}(\Sigma, \mathcal{D}')$. After this, it assembles the results $\mathrm{Vio}(\Sigma, \mathcal{D}')$ as $\Delta\mathrm{Vio}^+(\Sigma, \mathcal{D}, \Delta\mathcal{D})$. (2) The set $\Delta\mathrm{Vio}^-(\Sigma, \mathcal{D}, \Delta\mathcal{D})$ of removed discrepancies is computed in a similar way by inspecting composite databases in $\mathrm{Cmp}(\mathcal{D}, \Delta\mathcal{D}^-)$ that have tuples from $\Delta\mathcal{D}^-$.

Analysis. The worst-case runtime of algorithm IncDet is the sum of the cost incurred by REEDet on all composite databases that include tuples in $\Delta\mathcal{D} = (\Delta\mathcal{D}^+, \Delta\mathcal{D}^-)$. It is in $O(\sum_{\varphi \in \Sigma} \sum_{\mathcal{D}' \in \mathbb{D}'} |\mathcal{D}'|^{|Q_\varphi|})$, where Q_φ is the detection query of REE φ and $\mathbb{D}'$ consists of all composite databases deduced from $\Delta\mathcal{D}^+$ and $\Delta\mathcal{D}^-$, *i.e.,* those in $\mathrm{Cmp}(\mathcal{D} \setminus \Delta\mathcal{D}^-, \Delta\mathcal{D}^+) \cup \mathrm{Cmp}(\mathcal{D}, \Delta\mathcal{D}^-)$ having tuples in $\Delta\mathcal{D}$.

5.2 Parallel Incremental Algorithm

We next propose the parallel algorithm PIncDet for incremental discrepancy detection and verify its performance guarantees.

Making use of Lemma 5, the main idea of PIncDet is by reducing the computation of $\Delta\mathrm{Vio}(\Sigma, \mathcal{D}, \Delta\mathcal{D})$ to deriving violations contributed by tuple insertions $\Delta\mathcal{D}^+$ and deletions $\Delta\mathcal{D}^-$ in the given input updates $\Delta\mathcal{D}$. However, to improve the degree of parallelism of the computation, PIncDet demands an effective strategy to split the computation of violations contributed by updated tuples. Indeed, the constraints imposed by the relative boundedness of PIncDet make it more intriguing to handle skewed computations.

In light of these, we develop (a) a specific blocking approach *w.r.t.* composite databases, and (b) a corresponding strategy to eliminate skewed computation in the parallel incremental algorithm PIncDet.

Blocking revisited. One might be tempted to adopt the blocking strategy of batch algorithm PREEDet (Section 4.3). However, it handles the original database $\mathcal{D}$ for computing the entire set of violations. In contrast, PIncDet needs to find the violations contributed by the updated tuples and extract such tuples from composite databases. Therefore, we revise the range-based partition scheme to accommodate the special effects of updates $\Delta\mathcal{D}$.

Recall from Section 4.3 that each block *w.r.t.* an REE φ_w is a pair $w = (\bar{c}, \varphi_w)$, where $\bar{c}$ is an assembled cell of ranges, *i.e.,* the condition for selecting an appropriate dataset $\mathcal{D}_w$ from $\mathcal{D}$ for the corresponding work unit of w. We extend it to a triple $w = (\bar{c}', \bar{o}, \varphi_w)$, referred to as an *augmented block w.r.t.* REE φ_w. Here $\bar{c}'$ is still an assembled cell of ranges, $\bar{o}$ is a list of symbols of ins or del such that there is a mapping between these symbols and the relations in a subset of $\mathcal{D}$ to which φ_w is being applied. Here the length of list $\bar{o}$ is smaller than or equal to the number of relation atoms in φ_w.

Figure 4: Parallel incremental algorithm PIncDet

Note that augmented blocks are also "virtual" since they store no raw data. Moreover, each augmented block $w = (\bar{c}', \bar{o}, \varphi_w)$ also corresponds to one work unit that consists of dataset $\mathcal{D}_w$ and REE φ_w, where $\mathcal{D}_w$ is extracted from $\mathcal{D}$ and $\Delta\mathcal{D}$. More specifically, if the list $\bar{o}$ contains symbols ins (resp. del), then $\mathcal{D}_w$ is built with (a) inserted tuples (resp. deleted tuples) from those relations having symbols ins (resp. del); and (b) old tuples from the rest of relations that do not appear in $\Delta\mathcal{D}$ and fall in the ranges (*i.e.,* selection condition) in $\bar{c}'$.

Note that we only reserve those augmented blocks $(\bar{c}', \bar{o}, \varphi_w)$ such that the assembled cell $\bar{c}'$ is valid *w.r.t.* the REE φ_w (see Section 4.3). The others have no impact on the output of incremental detection for the same reason as that given in Section 4.3.

Intuitively, augmented blocks extend blocks with simple symbols indicating where to choose updated tuples for the datasets $\mathcal{D}_w$ in their work units, and the remaining parts in $\mathcal{D}_w$ are still filtered out in a range-based manner. It can be verified that each such $\mathcal{D}_w$ involves updated tuples in $\Delta\mathcal{D}$ and is a subset of the composite databases in $\mathrm{Cmp}(\mathcal{D} \setminus \Delta\mathcal{D}^-, \Delta\mathcal{D}^+)$ or $\mathrm{Cmp}(\mathcal{D}, \Delta\mathcal{D}^-)$. That is, the blocking strategy guides us to construct composite databases.

Skewness reduction. An augmented block $w = (\bar{c}', \bar{o}, \varphi)$ is *heavy* if $\prod_{i=1}^{p} |D_w^i| > |\Delta\mathcal{D}|^2 (|\mathcal{D}|/n)^{|Q_\varphi|-2}$, where $D_w^1, \ldots, D_w^p$ are the relations in the dataset $\mathcal{D}_w$ of the work unit built from w, each corresponding to a relation atom in φ. To reduce skewed computation, we divide heavy augmented blocks and their work units along the same lines as procedure WDivide of algorithm PREEDet (Section 4.3). As a result, Lemma 3 still holds for PIncDet.

Algorithm PIncDet. Putting these together, we present the main driver of incremental PIncDet in Figure 4, which works with n processors to deduce the set $\Delta\mathrm{Vio}(\Sigma, \mathcal{D}, \Delta\mathcal{D})$ of changes. It first splits updates $\Delta\mathcal{D}$ into tuple insertions $\Delta\mathcal{D}^+$ and deletions $\Delta\mathcal{D}^-$ (line 1). It then uses a parallel procedure PCrtDet to compute the violations contributed by $\Delta\mathcal{D}^+$ and $\Delta\mathcal{D}^-$ (line 2). Finally, it collects all such violations and returns them as $\Delta\mathrm{Vio}(\Sigma, \mathcal{D}, \Delta\mathcal{D})$ (line 3).

Procedure PCrtDet. As shown in Figure 4, given two databases $\mathcal{D}_1$ and $\mathcal{D}_2$ and a set Σ of REEs, PCrtDet computes the violations $\mathrm{CrtVio}(\Sigma, \mathcal{D}_1, \mathcal{D}_2)$ contributed by $\mathcal{D}_2$ in parallel. Note that when called by algorithm PIncDet, the input $\mathcal{D}_2$ refers to either inserted

tuples or deleted tuples in $\Delta\mathcal{D}$. Hence PCrtDet first generates a set $\mathcal{W}$ of augmented blocks *w.r.t.* the REEs in Σ (lines 2-3), through procedure CrsPar that is executed at a designated coordinator S_c (not shown). CrsPar joins each valid assembled cell $\bar{c}'$ *w.r.t.* REE φ with different lists $\bar{o}$ of symbols ins or del to build augmented blocks $(\bar{c}', \bar{o}, \varphi)$. Here the valid assembled cells are obtained directly from the result of the batch run of PREEDet as byproducts.

Procedure PCrtDet then decomposes heavy augmented blocks into smaller ones as discussed above (line 4). It then distributes these augmented blocks among n processors, revises the resulting partition with the more accurate cost models and shuffles the raw data accordingly (line 5). Again, each tuple is sent to a processor *at most once* as in PREEDet. PCrtDet next invokes sequential algorithm REEDet on the work units of all augmented blocks in parallel at n processors (lines 6-8), and returns the union of these results (line 9).

Example 6: Consider REE φ_4 and database $\mathcal{D}_c$ of Example 4. Let updates $\Delta\mathcal{D}_c$ include deletion of r_4 and insertion of a new tuple r_6' to D_1, whose create_time (resp. payment and level) value is 2018-12-13 (resp. 70, 5). Given $\Delta\mathcal{D}_c$, PIncDet constructs augmented blocks *w.r.t.* φ_4 to build composite databases. It combines all the ranges derived in Example 4 with symbol ins (resp. del) that represents r_6' (resp. r_4) to build 16 augmented blocks, and each block includes a valid assembled cell of 3 ranges and a list $\bar{o}$ with a single ins or del. Note that no heavy augmented block *w.r.t.* φ_4 exists, since each corresponding dataset has at most 4 tuples from D_1 and $\Delta\mathcal{D}_c$. Thus PIncDet just allocates the 16 augmented blocks evenly among the processors. It also shuffles the tuples in D_1 and $\Delta\mathcal{D}_c$ accordingly.

After invoking REEDet on each corresponding work unit of the augmented blocks, PIncDet concludes that r_1 and r_4 (resp. r_5 and r_6') form a violation to be removed (resp. introduced). □

We have the following for the parallel procedure PCrtDet.

Lemma 6: *Let* $T(|\Sigma|, |\mathcal{D}_1|, |\mathcal{D}_2|)$ *and* $t(|\Sigma|, |\mathcal{D}_1|, |\mathcal{D}_2|)$ *be the worst-case cost to compute the set* $\mathsf{CrtVio}(\Sigma, \mathcal{D}_1, \mathcal{D}_2)$ *by parallel procedure* PCrtDet *and sequential algorithm* IncDet, *respectively. Then* $T(|\Sigma|, |\mathcal{D}_1|, |\mathcal{D}_2|) = \tilde{O}(t(|\Sigma|, |\mathcal{D}_1|, |\mathcal{D}_2|)/n)$. □

Relative boundedness. We now show the main properties of algorithm PIncDet. To evaluate the effectiveness of incremental algorithms, a notion of relative boundedness was proposed in [42]. Here we adapt it to incremental detection. Denote by $(\mathcal{D}, \Sigma)_\mathcal{A}$ the data accessed by a batch algorithm $\mathcal{A}$ for computing violations $\mathsf{Vio}(\Sigma, \mathcal{D})$. For updates $\Delta\mathcal{D}$, the affected area $\mathsf{AFF}(\mathcal{D}, \Delta\mathcal{D}, \Sigma)_\mathcal{A}$ is measured as the difference between $(\mathcal{D}, \Sigma)_\mathcal{A}$ and $(\mathcal{D} \oplus \Delta\mathcal{D}, \Sigma)_\mathcal{A}$, *i.e.*, the difference in the data inspected by $\mathcal{A}$ when computing $\mathsf{Vio}(\Sigma, \mathcal{D})$ and $\mathsf{Vio}(\Sigma, \mathcal{D} \oplus \Delta\mathcal{D})$. We say that an incremental discrepancy detection algorithm $\mathcal{A}_\Delta$ is *bounded relative to* $\mathcal{A}$ if its cost for computing $\Delta\mathsf{Vio}(\Sigma, \mathcal{D}, \Delta\mathcal{D})$ can be expressed by a polynomial of $|\Sigma|$, $|\Delta\mathcal{D}|$ and $|\mathsf{AFF}(\mathcal{D}, \Delta\mathcal{D}, \Sigma)_\mathcal{A}|$. Intuitively, $|\mathsf{AFF}(\mathcal{D}, \Delta\mathcal{D}, \Sigma)_\mathcal{A}|$ is the essential cost needed for "incrementalizing" algorithm $\mathcal{A}$.

Theorem 7: *The incremental algorithm* PIncDet *is (1) parallelly scalable relative to* IncDet; *and (2) bounded relative to* PREEDet. □

Proof sketch: The parallel scalability of PIncDet follows from Lemmas 5 and 6. We show that it is bounded relative to PREEDet by proving the following: (a) all tuples collected in PIncDet for construing work units are covered by $\mathsf{AFF}(\mathcal{D}, \Delta\mathcal{D}, \Sigma)_{\mathsf{PREEDet}}$, re-

gardless of what ML predicates used, and (b) the sizes of the tuples inspected for building augmented blocks and load balancing are proportional to that of the tuples finally collected. □

Remark. From the proof one can see that both parallel scalability and relative boundedness are robust to various ML predicates. Moreover, PIncDet is also the first parallel incremental algorithms for discrepancy detection with CFDs, MDs and DCs with these properties.

6 EXPERIMENTAL STUDY

Using real-life and synthetic data, we conducted four sets of experiments to evaluate (1) the accuracy, (2) efficiency, (3) (parallel) scalability of our (incremental) discrepancy detection algorithms PREEDet and PIncDet, and (4) a case study with real-life data.

Experimental Settings. We start with the setting.

Datasets. We used three real-life datasets. (a) TFACC, a real-life dataset that integrates the test data from Ministry of Transport [4] and data from National Public Transport Access Nodes [1]. It covers test records of vehicles in the UK from 2005 to 2017. TFACC has 19 tables, 113 attributes and over $480M$ tuples, about 21.8GB of raw data in total. (b) Movie, a dataset that combines the metadata of real-life movies and artists in [2] and the databases of movies in [28]. It consists of 9 tables and 73 attributes, having over $97M$ tuples and 5.8GB of raw data in total. In particular, $2M$ tuples in Movie had already been labeled with matches and non-matches [28]. (c) House, a real-life dataset that integrates the private sector "empty homes" in London [7], properties owned by Greater London Authority [5], and the meta information of different areas in London [6]. House contains 7 tables, 166 attributes and $12K$ tuples in total.

We also generated synthetic datasets based on TPCH [8], which is a standard benchmark. TPCH creates data using TPCH dbgen [8]. The synthetic data has 8 relations and 61 attributes.

Updates. The updates $\Delta\mathcal{D}$ were randomly generated, controlled by the size $|\Delta\mathcal{D}|$ and the ratio γ of tuple insertions to deletions in $\Delta\mathcal{D}$. Based on the release dates in Movie dataset, we also introduced real periodic updates to Movie, by inserting new movies and deleting oldest movies *w.r.t.* γ. Unless stated otherwise, we set $\gamma = 1$, *i.e.*, the size of dataset remains unchanged after incurring the updates.

ML models. We applied ML predicates attributes of long textual values, on which conventional logic predicates do not work very well. We adopted a pre-trained BERT [30] model with 12 layers and $110M$ parameters, to initiate an embedding vector for each list of textual values. The ML predicates check whether the similarity of two embedding vectors reaches a given threshold.

REE *rules*. We discovered 40, 50, 100 and 20 REEs for TFACC, Movie, TPCH and House, respectively. Our REE mining algorithm (a) separates a dataset into sampling data D_s and testing data D_t, to discover REEs from D_s and validate the rules with D_t; (b) picks sample data D_s by clustering and taking representative tuples with minimal regret [79], (c) iteratively applies newly discovered REEs to D_s for "denoising", (d) incrementally discovers REEs from the cleaned sample data, and (e) selects appropriate ML predicates. The discovery method for REE is a nontrivial extension of parallel GFD discovery algorithm [41]. We defer its full treatment to another paper.

Ground truth and noises. To evaluate the accuracy of discrepancy detection, we injected noise into *random* data cells when we are short of ground truth. For TFACC and TPCH, assuming that the original datasets are correct, we updated their attribute values and added duplicate tuples, controlled by a percentage $\alpha\%$ of the number of changes to the total number of meaningful attributes values, and a ratio $\beta\%$ of duplicates to the whole set of tuples, respectively.

The labeled results in Movie were treated as ground truth for entity resolution without adding duplicates, while erroneous values were *randomly* injected for conflict detection. We also manually examined House and identified erroneous values such as postcode misspelling. We manually labeled erroneous cells in the entire dataset and took them as the ground truth for House.

Evaluation. We measured the accuracy in terms of Precision, Recall, and F-Measure defined as $2 \cdot \text{Precision} \cdot \text{Recall}/(\text{Precision} + \text{Recall})$. Precision (resp. Recall) is the ratio of detected *true* erroneous tuples to all the potential erroneous tuples in the violations of the REEs (resp. all the tuples that include erroneous values deduced from the ground truth). Here true erroneous tuples are those that violate REEs and must contain verified erroneous values *w.r.t.* the ground truth. The accuracy of the baselines is computed in a similar way in terms of the potential erroneous tuples found by different methods, *i.e.,* those tuples that involve the detected erroneous values.

Baselines. Apart from PREEDet (Section 4.3) and PIncDet (Section 5.2), we implemented the following, all in Python. (1) Five variants of PREEDet, including (a) PCRDet that catches conflict values only; (b) PERDet that detects duplicates and mismatched entities only; (c) PDet_{nml} without using ML predicates; (d) PDet_{nh}, a parallel detection algorithm without the skewness reduction given in Section 4.3; and (e) PDet_{mul}, a parallel algorithm that applies the blocking strategy *w.r.t.* each single REE one by one. (2) Two variants PIncDet_{nh} and PIncDet_{mul} of PIncDet, which revise PIncDet analogous to how PDet_{nh} and PDet_{mul} revise PREEDet, respectively. We compared with PCRDet, PERDet and PDet_{nml} for accuracy, and with PDet_{nh}, PDet_{mul}, PIncDet_{nh} and PIncDet_{mul} for efficiency.

We also compared with six baselines, including (3) DeepER [34] (denoted as DER), a state-of-the-art ER method based on the latest deep learning model; (4) HoloClean [66] (denoted as Holo), a data repairing system based on probabilistic inference. We took its discrepancy detection component, which implements a rule-based discrepancy detection approach with DCs; (5) HoloDetect [53] (denoted as HDet), a CR system that predicts the correct values of certain attributes via ML strategies; (6) SCODED [80], a discrepancy detection system that checks the violations of statistical constraints (SCs); (7) UniClean [44] (denoted as UClean), a framework that unifies MDs and CFDs for data repairing; we considered only its discrepancy detection module; and (8) ER-N [75], an ER framework that enhances ER with matching and negative rules.

We have also tested (9) DeepMatcher [61], another ER method based on the latest deep learning; and (10) Raha [60], an ML-based system that can automatically configure an approach for CR. The results are consistent (not shown due to the lack of space).

For DER, we sampled 10% fraction from each original dataset without noises as the training data and used their default parameters to initialize their model training. For HDet, we built the training set

Table 2: Accuracy (Exp-1: FM: F-Measure, P: Precision, R: Recall)

Dataset	TFACC			Movie			TPCH			House		
	FM	P	R	FM	P	R	FM	P	R	FM	P	R
PREEDet	0.83	0.84	0.83	0.89	0.91	0.87	0.83	0.85	0.81	0.82	0.91	0.75
PCRDet	0.55	0.84	0.41	0.58	0.85	0.44	0.52	0.79	0.39	0.82	0.91	0.75
PERDet	0.64	0.72	0.58	0.7	0.83	0.61	0.61	0.69	0.54	-	-	-
PDet_{nml}	0.67	0.73	0.62	0.73	0.81	0.67	0.74	0.78	0.71	0.76	0.9	0.66
DER	0.65	0.69	0.61	0.62	0.71	0.55	0.64	0.71	0.58	-	-	-
Holo	0.65	0.73	0.59	0.71	0.83	0.62	0.66	0.81	0.56	0.77	0.88	0.68
HDet	0.68	0.73	0.64	0.62	0.59	0.65	0.55	0.68	0.46	0.61	0.64	0.58
SCODED	0.66	0.61	0.72	0.54	0.52	0.57	0.46	0.57	0.38	0.53	0.43	0.69
UClean	0.72	0.84	0.63	0.74	0.79	0.69	0.72	0.79	0.66	0.72	0.86	0.62
ER-N	0.61	0.66	0.56	0.66	0.76	0.58	0.59	0.68	0.52	-	-	-

with each dataset in twofold, in which we set a 2% fraction as training data with ground truth, and another 8% fraction as the sampling set for active learning. For each rule-based baseline, we generated a set of constraints such that it includes our REEs expressed in the baseline's rule language whenever possible. More specifically, we built (a) DCs for Holo; (b) MDs and CFDs for UClean; and (c) matching rules written in MDs, and manually crafted negative rules for ER-N. In addition, we constructed (d) SCs for SCODED, in which for each REE $X \rightarrow e$ defined on a single relation, we added one dependence SC $A_X \not\perp A_e$ [80]. We also manually designed independence SCs based on domain knowledge of the database.

Configuration. We deployed the algorithms on a cluster of up to 20 machines, each with 32GB DDR4 RAM and 2.80GHz Intel i5-8400 CPU. We used PostgreSQL 10 as the underlying DBMS when needed by the algorithms. All the experiments were repeated 5 times. The average is reported here.

Experimental results. We next report our findings.

Exp-1: Accuracy. Fixing $\alpha\% = 0.6\%$ and $\beta\% = 5\%$, we first tested the accuracy of PREEDet for discrepancy detection. We compared PREEDet with various competitors on datasets TFACC, Movie, House and TPCH, using all the REEs discovered. The results are summarized in Table 2. Observe the following.

(1) PREEDet has the highest accuracy. It outperforms DER, Holo, HDet, SCODED, UClean and ER-N by 34%, 21%, 38%, 56%, 16%, and 37% on average, respectively. This is because PREEDet combines rule-based and ML-based methods, while the others are based on either ML models (DER, HDet, SCODED) or logic rules alone (Holo, UClean, ER-N). This verifies the effectiveness of unifying logic rules and ML predicates in one framework together. Since there exist no duplicate tuples in the manually checked House, the ER methods PERDet, DER and ER-N are not applicable to House.

(2) PREEDet also outperforms its variants PCRDet, PERDet and PDet_{nml} by 47%, 31% and 19%, respectively. This is because (i) REE unifies CR and ER together; and (ii) ML predicates improve the accuracy when dealing with long textual attributes.

(3) The accuracy of PIncDet (not shown) is the same as that of PREEDet, since the two share the same detection method.

(4) PREEDet consistently beats the baselines in capturing all types of discrepancies. On average PREEDet improves the F-Measure by 32% and 42% in catching real and injected discrepancies, respectively. This validates the effectiveness of REEs in real-life practice.
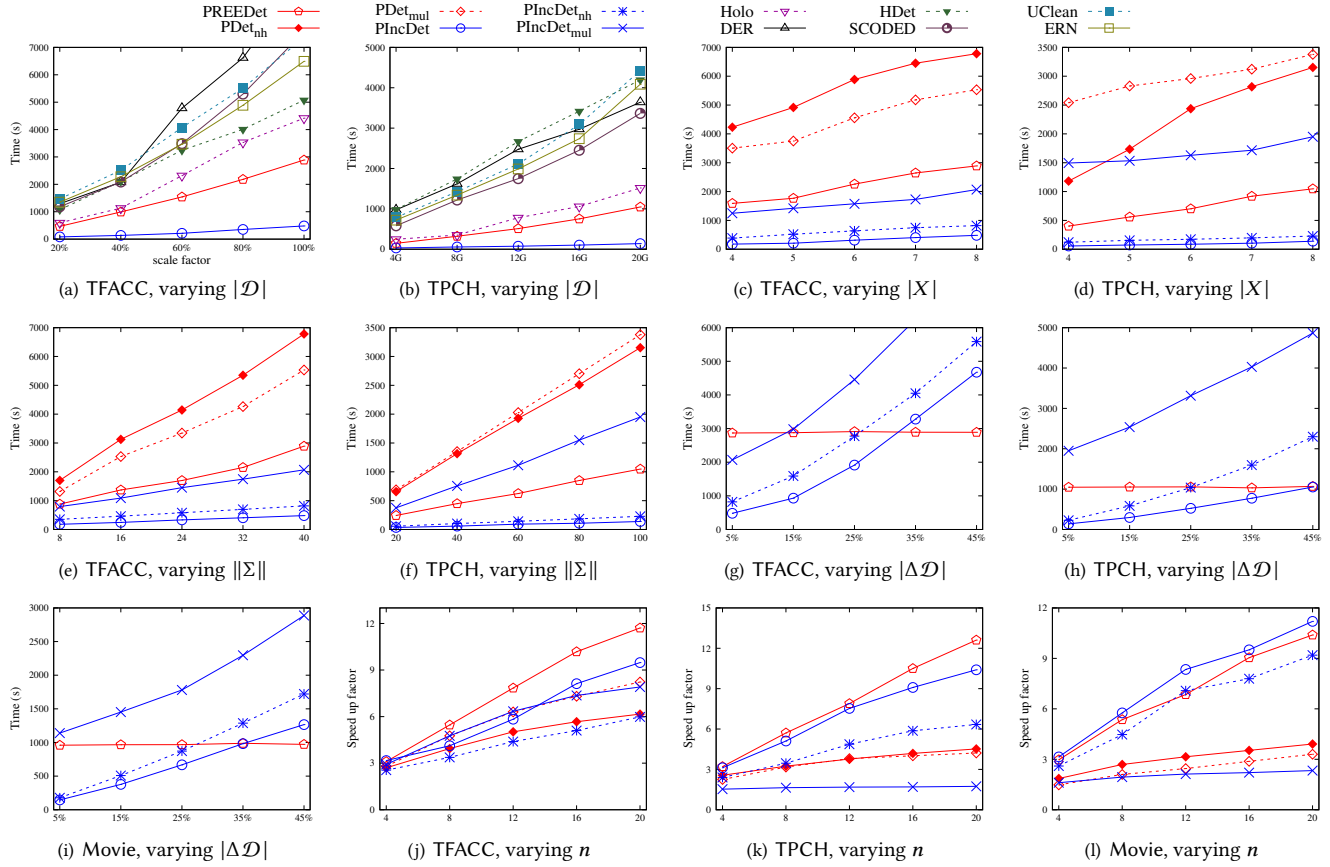
Figure 5: Performance evaluation

(5) When we additionally plug the ML models for ER (resp. CR) trained by DER (resp. HDet) in the REEs, on average, the F-Measure of PREEDet is 0.9, which is 41%, 29%, 46%, 64%, 24% and 45% better than DER, Holo, HDet, SCODED, UClean and ER-N, respectively. These justify the need to embed ML predicates in REEs.

Exp-2: Efficiency. We tested PREEDet and PIncDet in efficiency, by varying (1) the size $|\mathcal{D}|$ of datasets, (2) the number $|X|$ of predicates in the precondition of a single REE, *i.e.,* the complexity of REEs, (3) the number $\|\Sigma\|$ of rules used, and (4) the size $|\Delta\mathcal{D}|$ of updates. We used TFACC and TPCH; the results on Movie and House are consistent (not shown except for varying $|\Delta\mathcal{D}|$ over Movie). We used the same $\alpha\%$ and $\beta\%$ as in Exp-1 and $n = 20$ machines. For PIncDet, we set $|\Delta\mathcal{D}| = 5\%|\mathcal{D}|$ by default.

(1) Varying $|\mathcal{D}|$. We first evaluated the impact of $|\mathcal{D}|$ on both batch PREEDet and incremental PIncDet. Using all the REEs discovered, we varied $|\mathcal{D}|$ from 4.4 GB to 21.8 GB for TFACC, *i.e.,* the scale factor is from 20% to 100%, and 4 GB to 20 GB for TPCH. From the results reported in Figures 5(a) and 5(b), we find the following.

(a) All algorithms take longer when $|\mathcal{D}|$ gets larger. PREEDet consistently outperforms DER, Holo, HDet, SCODED, UClean and ER-N, by 3.8, 1.4, 3.6, 2.9, 3.5 and 3.3 times on average, respectively. In addition, PIncDet is less sensitive than PREEDet to the increase of $|\mathcal{D}|$, due to its incremental nature (see more in Exp-2 (4)).

(b) PREEDet beats its variants PDet_{nh} and PDet_{mul} by 2.3 times and 1.8 times on average, respectively (not shown). This is because PREEDet reduces (i) skewness to balance workload and (ii) communication among multiple rules (see Section 4). For the same reason, PIncDet beats the variants PIncDet_{nh} and PIncDet_{mul} by 2 times and 3.9 times, respectively (not shown).

(2) Varying $|X|$. We next tested the impact of the complexity of REEs by varying the number $|X|$ of predicates in the preconditions, in which the number of relation atoms is at most 4. The results are reported in Figures 5(c) and 5(d), which tell us the following.

(a) All algorithms take longer when $|X|$ gets larger. This is consistent with their complexity since we translate discrepancy detection tasks into conjunctive queries (see Section 4.2). We found that the number of relation atoms is the dominant factor for the increased cost. For instance, one additional relation atom in the preconditions of REEs increases the running time of PREEDet by 21.9% on average.

(b) Algorithms PREEDet and PIncDet are less sensitive to $|X|$ than PDet_{nh} and PIncDet_{nh}. This is because (i) more blocks are generated when $|X|$ increases and *heavy blocks* are more likely to occur; and (ii) PREEDet and PIncDet effectively eliminate heavy blocks to reduce skewness, while PDet_{nh} and PIncDet_{nh} do not.

(3) Varying $\|\Sigma\|$. We next varied the number $\|\Sigma\|$ of rules used. As shown in Figures 5(e) and 5(f), all algorithms take longer when

$\|\Sigma\|$ increases. Among them, PDet_{mul} and $\text{PIncDet}_{\text{mul}}$ are most sensitive to $\|\Sigma\|$, since their data shuffling goes up substantially when $\|\Sigma\|$ increases. On average, PDet_{mul} (resp. $\text{PIncDet}_{\text{mul}}$) ships 9.6 (resp. 7.8) times more data than PREEDet (resp. PIncDet) when $\|\Sigma\|$ varies from 20 and 100 on TPCH. Both PREEDet and PIncDet reduce communication by transmitting common data only once (Sections 4 and 5), especially when $\|\Sigma\|$ is large, *i.e.,* more attributes are shared by the REEs in Σ. The results on TFACC are consistent.

(4) Varying $|\Delta\mathcal{D}|$. We also evaluated the impact of the size $|\Delta\mathcal{D}|$ of updates. Fixing $|\mathcal{D}|$ and REEs as in Exp-1, we compared PIncDet with its variants and batch PREEDet by varying $|\Delta\mathcal{D}|$. As shown in Figures 5(g)–5(i), the results tell us the following.

(a) Algorithm PIncDet constantly beats its batch counterpart. On average PIncDet beats PREEDet by 3.7 times when $|\Delta\mathcal{D}|$ varies from 5% to 45% of $|\mathcal{D}|$, and by 9.7 times when $|\Delta\mathcal{D}| = 5\%|\mathcal{D}|$. PIncDet is faster than PREEDet even when $|\Delta\mathcal{D}|$ is up to 25% of TFACC, 45% of TPCH, and 35% of Movie, respectively. This verifies the effectiveness of relatively bounded incremental processing (Theorem 7). It also shows that the results on periodic updates over Movie are consistent with those on randomly generated ones.

(b) PIncDet outperforms $\text{PIncDet}_{\text{nh}}$ and $\text{PIncDet}_{\text{mul}}$ on average by 2.1 and 4.6 times, respectively. This again verifies the effectiveness of our optimization techniques (see also Exp-2 (1)).

Exp-3: Parallel scalability. We next evaluated the parallel scalability of PREEDet, PIncDet and their variants, by varying the number n of machines used from 4 to 20. We used the same REEs as in Exp-1 over TFACC, TPCH and Movie. We find the following.

As shown in Figures 5(j)-5(l) (in which y-axis shows the speedup compared with the single-machine computation), (1) PREEDet and PIncDet scale well with n. These methods are on average 3.2 to 12.2 (resp. 3.1 to 9.9) times faster than the single-machine computation when n is varied from 4 to 20. (2) All algorithms take less time when n increases, as expected. (3) PREEDet is 2.1 and 2.5 times faster than PDet_{nh} and PDet_{mul}, respectively, and PIncDet is 1.7 and 2.9 times faster than $\text{PIncDet}_{\text{nh}}$ and $\text{PIncDet}_{\text{mul}}$, respectively.

These experimentally verify Theorems 4 and 7, and further demonstrate the effectiveness of our optimization strategies.

Exp-4: Case study. We also found that REEs are able to detect the discrepancies that are beyond the capability of other methods.

(1) In TFACC, tables result (testid, vehicleid, test_result, model) and item(testid, rfr_type_code, location) record the details of vehicle tests and failed items, respectively. Here test_result can be either 'P' (pass) or 'F' (fail), while rfr_type_code, *i.e.,* the type of reason for failure, is 'F' (fail) or 'P' (failing item repaired within one hour).

One REE for TFACC is $\varphi_a = \text{result}(t_1) \wedge \text{item}(t_2) \wedge t_1.\text{testid} = t_2.\text{testid} \wedge t_2.\text{tft\_type\_code} = P \rightarrow t_1.\text{test\_result} \neq P$, which ensures that no passed test has failing items. It catches a discrepancy that the test of id 792454969 passed in 2016 but had a failing item.

(2) In Movie, imdb(id, title, year, plot, country) and omdb (id, title, release_year, plot) include movies from different data sources, and person(id, name, birthYear, profession, film) contains the details of artists and their films. The REEs below were used for Movie.

(i) $\varphi_b = \text{person}(t_1) \wedge \text{person}(t_2) \wedge \text{imdb}(t_3) \wedge \text{imdb}(t_4) \wedge t_1.\text{film} = t_3.\text{id} \wedge t_2.\text{film} = t_4.\text{id} \wedge t_3.\text{country} = \text{China} \wedge t_4.\text{country} = \text{Norway} \rightarrow t_1.\text{id} \neq t_2.\text{id}$. That is, no one had participated in both Chinese and Norwegian films. It finds a mismatch, *i.e.,* two producers named "James Wang" should not match due to their different films.

(ii) $\varphi_c = \text{person}(t_1) \wedge \text{imdb}(t_2) \wedge \text{omdb}(t_3) \wedge t_1.\text{film} = t_2.\text{id} \wedge \mathcal{M}(t_2, t_3) \rightarrow t_1.\text{birthYear} < t_3.\text{release\_year}$. Here $\mathcal{M}$ is an ML model trained in DER to predict whether movies t_2 from IMDB and t_3 in OMDB match. The REE says that the birth year of a person must be earlier than the release year of his movie. It catches "Spencer Gray", who was born in 2000, but stared in "Days of Our Lives" released in 1965. Note that the year attribute in relation imdb is ambiguous, which may also denote the starting year of photography and can be earlier than the birth years of its actors. Hence we "reinforce" the ML model $\mathcal{M}$ and discrepancy detection by additionally comparing release_year and birthYear. This REE shows how we can leverage existing ML models and further improve them with logic predicates. It cannot be expressed as CFDs or DCs.

Summary. We find the followings. (1) It is effective to detect discrepancies by embedding ML predicates in logic rules and unifying ER and CR. On average, PREEDet outperforms rule-based and ML methods by 33% and 36%, and ER and CR alone by 31% and 41%, respectively, in accuracy. (2) PREEDet and PIncDet scale well with the complexity of REEs and the size $|\mathcal{D}|$ of datasets. Using 20 machines, they take 1047s and 139s on TPCH with 150 million tuples and $|\Delta\mathcal{D}| = 5\%|\mathcal{D}|$, respectively. PREEDet is also more efficient than existing methods for discrepancy detection, at least 3.2 times faster than the competitors on TPCH. (3) Incremental PIncDet is effective. It takes 15s when $\mathcal{D}$ has 150 million tuples and $|\Delta\mathcal{D}| = 0.1\%|\mathcal{D}|$ (online changes to real-life large datasets are small, typically below 0.1%). On average, it outperforms batch algorithm PREEDet by 20.4 times when $|\Delta\mathcal{D}| = 1\%|\mathcal{D}|$, and is still faster even when $|\Delta\mathcal{D}|$ is up to 45% of $|\mathcal{D}|$. (4) PREEDet and PIncDet are parallelly scalable; compared to the single-machine computation, on average they are 3.2 to 12.2 (resp. 3.1 to 9.9) times faster when n varies from 4 to 20. (5) Our optimization strategies, including workload balancing and uniform data shuffling, improve the performance by 2.1 times for PREEDet and 3.6 times for PIncDet on average.

7 CONCLUSION

We have proposed a method to collectively detect duplication, mismatches and conflicts by unifying rules and ML models. We have settled the complexity of (incremental) discrepancy detection with REEs and DCs. We have provided (a) a parallelly scalable algorithm to detect discrepancies offline, and (b) a parallelly scalable and relatively bounded incremental algorithm to catch discrepancies online; these are also the first such algorithms for CFDs, DCs and MDs. We have empirically verified that the method is promising.

One topic for future work is to identify what ML models are effective on attributes of different types. Another topic is to discover interesting REEs from possibly dirty data with denoising.

REFERENCES

[1] 2014. NaPTAN. *http://data.gov.uk/dataset/naptan*.
[2] 2019. IMDB. *https://www.imdb.com/interfaces/*.
[3] 2020. Alipay. *https://www.alipay.com*.
[4] 2020. MOT tests and results. *https://data.gov.uk/dataset/e3939ef8-30c7-4ca8-9c7c-ad9475cc9b2f/anonymised-mot-tests-and-results*.
[5] 2021. GLA Group Land Assets. *https://data.london.gov.uk/dataset/gla-group-land-assets*.
[6] 2021. London Borough Profiles and Atlas. *https://ckan.publishing.service.gov.uk/dataset/london-borough-profiles-and-atlas*.
[7] 2021. London Empty Homes Audit. *https://ckan.publishing.service.gov.uk/dataset/london-empty-homes-audit*.
[8] 2021. TPC-H. *http://www.tpc.org/tpch/*.
[9] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.
[10] Gagan Aggarwal, Rajeev Motwani, and An Zhu. 2006. The load rebalancing problem. *J. Algorithms* 60, 1 (2006), 42–59.
[11] João Paulo Aires and Felipe Meneguzzi. 2017. Norm Conflict Identification Using Deep Learning. In *AAMAS Workshops*. 194–207.
[12] Rohit Ananthakrishna, Surajit Chaudhuri, and Venkatesh Ganti. 2002. Eliminating Fuzzy Duplicates in Data Warehouses. In *VLDB*. 586–597.
[13] Arvind Arasu, Michaela Götz, and Raghav Kaushik. 2010. On active learning of record matching packages. In *SIGMOD*. 783–794.
[14] Arvind Arasu, Christopher Ré, and Dan Suciu. 2009. Large-Scale Deduplication with Constraints Using Dedupalog. In *ICDE*. 952–963.
[15] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. 1999. Consistent Query Answers in Inconsistent Databases. In *PODS*. 68–79.
[16] Zeinab Bahmani, Leopoldo E. Bertossi, and Nikolaos Vasiloglou. 2017. ERBlox: Combining matching dependencies with machine learning for entity resolution. *Int. J. Approx. Reasoning* 83 (2017), 118–141.
[17] Paul Beame, Paraschos Koutris, and Dan Suciu. 2013. Communication steps for parallel query processing. In *PODS*. 273–284.
[18] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in parallel query processing. In *PODS*. 212–223.
[19] Catriel Beeri and Moshe Y. Vardi. 1981. *On the complexity of testing implications of data dependencies*. Technical Report. The Hebrew University of Jeruslem.
[20] Leopoldo Bertossi. 2011. *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers.
[21] Leopoldo E. Bertossi, Solmaz Kolahi, and Laks V. S. Lakshmanan. 2013. Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. *Theory Comput. Syst.* 52, 3 (2013), 441–482.
[22] Indrajit Bhattacharya and Lise Getoor. 2007. Collective entity resolution in relational data. *ACM Trans. Knowl. Discov. Data* 1, 1 (2007), 5.
[23] Mikhail Bilenko, Beena Kamath, and Raymond J Mooney. 2006. Adaptive Blocking: Learning to Scale Up Record Linkage. In *ICDM*. 87–96.
[24] Xu Chu, Ihab F. Ilyas, and Paraschos Koutris. 2016. Distributed Data Deduplication. *PVLDB* 9, 11 (2016), 864–875.
[25] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *ICDE*. 458–469.
[26] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. 2007. Improving Data Quality: Consistency and Accuracy. In *VLDB*. 315–326.
[27] Sanjib Das, Paul Suganthan G. C., AnHai Doan, Jeffrey F. Naughton, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, Vijay Raghavendra, and Youngchoon Park. 2017. Falcon: Scaling Up Hands-Off Crowdsourced Entity Matching to Build Cloud Services. In *SIGMOD*. 1431–1446.
[28] Sanjib Das, AnHai Doan, Paul Suganthan G. C., Chaitanya Gokhale, Pradap Konda, Yash Govind, and Derek Paulsen. 2020. The Magellan Data Repository. https://sites.google.com/site/anhaidgroup/projects/data.
[29] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. 2014. ClusterJoin: A Similarity Joins Framework using Map-Reduce. *PVLDB* 7, 12 (2014), 1059–1070.
[30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. 4171–4186.
[31] Mohamad Dolatshah, Mathew Teoh, Jiannan Wang, and Jian Pei. 2018. Cleaning Crowdsourced Labels Using Oracles For Statistical Classification. *PVLDB* 12, 4 (2018), 376–389.
[32] Xin Dong, Alon Halevy, and Jayant Madhavan. 2005. Reference Reconciliation in Complex Information Spaces. In *SIGMOD*. 85–96.
[33] Rod G. Downey and Michael R. Fellows. 1995. Fixed-Parameter Tractability and Completeness I: Basic Results. *SIAM J. Comput.* 24, 4 (1995), 873–921.
[34] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq R. Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed Representations of Tuples for Entity Resolution. *PVLDB* 11, 11 (2018), 1454–1467.
[35] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate Record Detection: A Survey. *IEEE Trans. Knowl. Data Eng.* 19, 1 (2007), 1–16.
[36] Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. 2015. Keys for Graphs. *PVLDB* 8, 12 (2015), 1590–1601.

[37] Wenfei Fan, Hong Gao, Xibei Jia, Jianzhong Li, and Shuai Ma. 2011. Dynamic constraints for record matching. *VLDB J.* 20, 4 (2011), 495–520.
[38] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.* 33, 2 (2008), 6:1–6:48.
[39] Wenfei Fan, Floris Geerts, Shuai Ma, and Heiko Müller. 2010. Detecting inconsistencies in distributed data. In *ICDE*. 64–75.
[40] Wenfei Fan, Floris Geerts, Nan Tang, and Wenyuan Yu. 2014. Conflict resolution with data currency and consistency. *J. Data and Information Quality* 5, 1-2 (2014), 6:1–6:37.
[41] Wenfei Fan, Chunming Hu, Xueli Liu, and Ping Lu. 2020. Discovering Graph Functional Dependencies. *ACM Trans. Database Syst.* 45, 3 (2020), 15:1–15:42.
[42] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental Graph Computations: Doable and Undoable. In *SIGMOD*. 155–169.
[43] Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. 2009. Reasoning about Record Matching Rules. *PVLDB* 2, 1 (2009), 407–418.
[44] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. 2011. Interaction between record matching and data repairing. In *SIGMOD*. 469–480.
[45] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. 2012. Towards certain fixes with editing rules and master data. *VLDB J.* 21, 2 (2012), 213–238.
[46] Wenfei Fan, Jianzhong Li, Nan Tang, and Wenyuan Yu. 2014. Incremental Detection of Inconsistencies in Distributed Data. *IEEE Trans. Knowl. Data Eng.* 26, 6 (2014), 1367–1383.
[47] Wenfei Fan, Xueli Liu, Ping Lu, and Chao Tian. 2020. Catching Numeric Inconsistencies in Graphs. *ACM Trans. Database Syst.* 45, 2 (2020), 1–47.
[48] Wenfei Fan, Ping Lu, and Chao Tian. 2020. Unifying Logic Rules and Machine Learning for Entity Enhancing. *Sci. China Inf. Sci.* 63, 7 (2020).
[49] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Functional Dependencies for Graphs. In *SIGMOD*. 1843–1857.
[50] Michael Garey and David Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.
[51] Lukasz Golab, Howard Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. 2008. On generating near-optimal tableaux for conditional functional dependencies. *PVLDB* 1, 1 (2008), 376–390.
[52] Songtao Guo, Xin Luna Dong, Divesh Srivastava, and Remi Zajac. 2010. Record Linkage with Uniqueness Constraints and Erroneous Values. *PVLDB* 3, 1 (2010), 417–428.
[53] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. 2019. HoloDetect: Few-Shot Learning for Error Detection. In *SIGMOD*. 829–846.
[54] Mayank Kejriwal and Daniel P. Miranker. 2013. An Unsupervised Algorithm for Learning Blocking Schemes. In *ICDM*. 340–349.
[55] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2015. BigDansing: A System for Big Data Cleansing. In *SIGMOD*. 1215–1230.
[56] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Dedoop: Efficient Deduplication with Hadoop. *PVLDB* 5, 12 (2012), 1878–1881.
[57] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Load Balancing for MapReduce-based Entity Resolution. In *ICDE*. 618–629.
[58] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2010. Evaluation of entity resolution approaches on real-world match problems. *PVLDB* 3, 1 (2010), 484–493.
[59] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. 1990. A Complexity Theory of Efficient Parallel Algorithms. *Theor. Comput. Sci.* 71, 1 (1990), 95–132.
[60] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2019. Raha: A Configuration-Free Error Detection System. In *SIGMOD*. 865–882.
[61] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD*. 19–34.
[62] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms. In *PODS*. 37–48.
[63] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2020. Blocking and Filtering Techniques for Entity Resolution: A Survey. *ACM Comput. Surv.* 53, 2 (2020), 31:1–31:42.
[64] Kun Qian, Lucian Popa, and Prithviraj Sen. 2017. Active Learning for Large-Scale Entity Resolution. In *CIKM*. 1379–1388.
[65] Vibhor Rastogi, Nilesh N. Dalvi, and Minos N. Garofalakis. 2011. Large-Scale Collective Entity Matching. *PVLDB* 4, 4 (2011), 208–218.
[66] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *PVLDB* 10, 11 (2017), 1190–1201.
[67] Bernd SW Schröder. 2003. Ordered sets. *Springer* 29 (2003), 30.
[68] Giovanni Simonini, Sonia Bergamaschi, and H. V. Jagadish. 2016. BLAST: a Loosely Schema-aware Meta-blocking Approach for Entity Resolution. *PVLDB* 9, 12 (2016), 1173–1184.
[69] J Michael Steele. 2004. *The Cauchy-Schwarz master class: an introduction to the art of mathematical inequalities*. Cambridge University Press.

[70] Arun N. Swami and K. Bernhard Schiefer. 1994. On the Estimation of Join Result Sizes. In *EDBT*. 287–300.

[71] Katia P. Sycara. 1993. Machine learning for intelligent support of conflict resolution. *Decision Support Systems* 10, 2 (1993), 121–136.

[72] Yufei Tao. 2018. Massively Parallel Entity Matching with Linear Classification in Low Dimensional Space. In *ICDT*. 20:1–20:19.

[73] Larysa Visengeriyeva and Ziawasch Abedjan. 2018. Metadata-driven error detection. In *SSDBM*. 1:1–1:12.

[74] Michael J Welch, Aamod Sane, and Chris Drome. 2012. Fast and accurate incremental entity resolution relative to an entity knowledge base. In *CIKM*. 2667–2670.

[75] Steven Euijong Whang, Omar Benjelloun, and Hector Garcia-Molina. 2009. Generic entity resolution with negative rules. *VLDB J.* 18, 6 (2009), 1261–1277.

[76] Steven Euijong Whang and Hector Garcia-Molina. 2013. Joint entity resolution

[77] on multiple datasets. *VLDB J.* 22, 6 (2013), 773–795.

[77] Steven Euijong Whang and Hector Garcia-Molina. 2014. Incremental entity resolution on rules and data. *VLDB J.* 23, 1 (2014), 77–102.

[78] David P. Woodruff and Qin Zhang. 2013. When Distributed Computation Is Communication Expensive. In *DISC*. 16–30.

[79] Min Xie, Raymond Chi-Wing Wong, and Ashwin Lall. 2019. Strongly Truthful Interactive Regret Minimization. In *SIGMOD*. 281–298.

[80] Jing Nathan Yan, Oliver Schulte, MoHan Zhang, Jiannan Wang, and Reynold Cheng. 2020. SCODED: Statistical Constraint Oriented Data Error Detection. In *SGIMOD*. 845–860.

[81] Dongxiang Zhang, Long Guo, Xiangnan He, Jie Shao, Sai Wu, and Heng Tao Shen. 2018. A Graph-Theoretic Fusion Framework for Unsupervised Entity Resolution. In *ICDE*. 713–724.