PAPER 438: RESPONSE TO REVIEWS

Dear Referees and Meta Reviewer:

The paper has been substantially revised. We have addressed all the issues raised in the reviews and meta review.

For the convenience of Referees, changes to the paper are color-coded in blue.

We would like to thank all the referees for their thorough reading of our paper and for their helpful comments.

Response to the comments of Meta Review.

[Required Changes] *All revision items listed in Field 13 of the reviews. In particular:*

(a). R1: O1, O2, O3

(b). R2: O1, O2, O3, O4, O5, O6, O7

(c). R5: O1, O2, O3, O4

[A] Thanks! We have supplemented detailed explanations and new experiments. More specifically, we have addressed all the comments of Referees 1, 2 and 5.

[Optional Changes] *R5: O2, O3*

[A] We have also addressed O2 and O3 of R5.

Response to the comments of Referee 1.

[O1] *Since the proposed parallel algorithm is levelwise, a more detailed comparison to [57] might help. The paper mentions one difference, guaranteed scalability, but I wonder if there are others. For example, [57] mentions distributed primitives that can be used to build distributed versions of various existing algorithm. Are new primitives needed for REEs?*

[A] Thanks! We have added the following.

(1) We have expanded discussions about [57] to the related work of Section 1 (pp. 2). More specifically, (a) [57] does not study how to mine rules with constants, and it only discovers bi-variable rules, *e.g.,* $t_0.A = t_1.A$. Our proposed discovery algorithm is to find rules with multiple tuple variables, constants and ML predicates, which are not handled by [57]. (b) Yes, [57] proposed a set of primitives for discovery of FDs and DCs. However, the primitives are not very adaptive to multiple tuple variables, rules with constant values, and embedded ML models. For example, primitive genEVSet(t_i, t_j, P) only constructs evidences for bi-variables, *i.e.,* t_i and t_j in the predefined predicate space P, and its time complexity is $O(n^2)$, where n is the number of tuples in $\mathcal{D}$. While in theory it can be extended as genEVSet$(t_{i_1}, t_{i_2}, \ldots, t_{i_m}, P)$ to construct evidence set for m tuple variables, it takes $O(n^m)$ time and hence is not feasible in practice. (c) The primitives in [57] cannot be used for constant discovery.

In light of these, we adopt the levelwise approach, which is more adaptive to the parallel setting, and the optimization strategies in Section 5.3 further accelerate the discovery process.

(2) To better evaluate the primitives of [57], we have developed an algorithm REEFinder to discover bi-variable REEs, by extending the primitives of [57], and added new experiments to test REEFinder (Section 6, pp. 11). More specifically, REEFinder (a) extends the function generateEvidence of DCfinder [57] (Figure 2 in [57]) by modifying genEVSet to support ML predicates; (b) revises function setCover to support discovery of bi-variable REEs, denoted as setCover$^+$; (c) adds a new primitive mineConstant to mine frequent itemsets [19]; and (d) supports constant predicates of REEs by enumerating constant combinations. DCfinder used in our experiment has almost the same process as REEFinder except that it uses setCover of [57] instead of setCover$^+$, for discovering DCs.

As reported in Figures 7(k)-7(s) (pp. 12), we have added experiments to evaluate REEFinder, compared with other methods in Exp-2 of Section 6 (pp. 12-13). In particular, we compared it with DCfinder of [52] (in this revision we restored DCfinder to its original version by removing the support of ML and constant predicates that we added in the earlier version, and parallelized DCfinder for a fair comparison). As shown there, REEFinder takes longer than DCfinder since it discovers constant patterns and supports ML predicates; but both are much slower than PRMiner, our proposed algorithm, *e.g.,* PRMiner, REEFinder and DCfinder take 6502s, 24010s and 20315s over dataset Tax, respectively.

[O2] *It might be good to clarify if the guarantees mention in section 4 are impacted by the optimization strategies in section 5.3.*

[A] As suggested, we have clarified this as follows. (1) We have highlighted that Theorem 1 considers the sets Σ_s and Σ of REEs discovered by the same algorithm over samples and entire dataset, respectively (pp. 4-6). (2) We have added a remark to Section 5.3 (pp. 9) to show that Theorem 1 still holds under the optimization strategies. More specifically, Theorem 1 deduces the bound between Σ_s and Σ that are mined by the same discovery algorithm, regardless of what algorithm it is as long as it is based on the multi-round sampling method. The optimization strategies are applied to the same algorithm for discovering Σ_s and Σ, just to improve its efficiency. Hence the bound of Theorem 1 remains intact no matter whether the optimization strategies are in place or not.

[O3] *I didn't quite understand the value of some of the REEs reported in Exp-3. For example, what does (1) tell us about the semantics of the DBLP dataset?*

[A] Thanks! We have further clarified the rules (Section 6, pp. 13).

- Rule 1 is an ER rule to identify whether two tuples refer to the same paper in the DBLP dataset. It says that if two papers have similar titles and venues, and if they were published in the same year, then the two denote the same paper. It employs ML models to check the semantic similarity of titles and venues, beyond prior data quality rules such as CFDs and DCs. In practice, our industry collaborators use such ER rules to catch duplicates and merge tuples that refer to the same entity.
- Rule 2 says that for two authors, it is because their names are similar and their affiliations are the same that the ML model predicts the two to match. It makes an attempt to explain the predication of a black box ML model with logic conditions.
- Rule 3 says that for three airports t_0, t_1, t_3, if t_0 and t_2 have the same iso_region, continent and latitude_deg, and if t_0 and t_1 have the same municipality and latitude_deg, then t_1 and t_2 also have the same iso_region. This rule involves three tuple variables, which are not supported by other discovery algorithms, *e.g.,* [56].

- Rule 4 says that two airports have different municipalities if they are in different countries. It shows that our discovery method is able to find rules that distinguish entities/attributes. Such rules are needed for, *e.g.,* catching mismatched entities.

Compared with the previous rules such as DCs and CFDs, REEs discovered by the proposed algorithm are more expressive and can help us detect more inconsistencies, conflicts and mismatches.

Many thanks for your support and helpful suggestions!

Response to the comments of Referee 2.

[O1] *The intro alludes to a generic sampling framework for discovering rules across multiple tables, but in fact, the work is focused on REEs. This should be revised to make it clear the focus is on REE discovery.*

[A] As suggested, we have further clarified that we focus on REEs in this paper (pp. 1). This said, the proposed method can also be adopted for discovering FDs, CFDs, MDs and DCs, since REEs subsume these rules as special cases as shown in [21].

[O2] *How is recall (Σ, Σ_S) computed when we don't have Σ? (since it's prohibitively expensive to do so)*

[A] Thanks! We have clarified that Σ in all datasets (Table 4) was computed in our experiments (Section 6, pp. 11); thus all the reported recall values are exact. We have also remarked how to estimate recall when the datasets are extremely large (footnote, pp. 11).

More specifically, we discovered the set Σ of REEs from the entire datasets (Table 4) in our experiment. This was possible by employing our parallel algorithm with proposed optimization techniques. When the datasets are very large, *e.g.,* having billions of tuples, Σ might be very expensive to compute. In this case, the recall and its confidence intervals could be *estimated* by employing existing recall estimation methods, *e.g.,* Monte Carlo simulation [67].

[O3] *The seeding of the candidate consequent simplifies the discovery problem and search space. How feasible and onerous is it to ask users to provide each set of consequent attributes?*

[A] Thanks! We have clarified this and provided a real-life example about the selection of candidate RHS (Section 3.2, pp. 5). Based on our experience with industry collaborators, users or domain experts usually have developed a good understanding of their datasets and know their pain points. They only care about key attributes and are able to pick RHS by referencing the schema of datasets. For novice users, they can simply start with all predicates as RHS and then narrow down to what they need. Thus it is not very difficult for users to select the candidate consequences.

A real-life case of our collaborators is deduplication in the Organization dataset. The users want to find and merge all duplicate tuples in Organization. They decided to discover ER rules, *e.g.,* using t_0.id = t_1.id as RHS. Another requirement is to fill in missing values in the attribute Country of Organization; to this end they chose RHS predicates related to Country, *e.g.,* t_0.Country = t_1.Country.

Another real-life case is to detect the abnormality of the delivery cost in a few datasets at a large delivery corporation. They only concern about abnormal values in the PRICE column, which maliciously influence their Express business process. The discovery algorithm could focus on such attributes as RHS.

[O4] *The predicate correlation training appears to be expensive and there is little discussion on its overhead. Given that the* RMiner *is run k-times, once for each sample, there needs to be some discussion on how this training can be minimized across the samples.*

[A] As suggested, we have added discussions about the training overhead (Section 5.3, pp. 8, and Section 5.4, pp. 10). Moreover, we have verified that the cost is small in Figure 7(t) (Section 6, pp. 12).

(1) To see the training overhead, note that the predicate correlation computation mainly consists of three parts: (1) computing the reward; (2) training $\mathcal{M}_{\text{corr}}$; and (3) using $\mathcal{M}_{\text{corr}}$ for inference during discovery. In the experiment, the training cost is very small compared to the costs of reward computation and discovery, because $\mathcal{M}_{\text{Corr}}$ is designed as a lightweight model. Meanwhile, the procedure of reward computation is the same as calculating the supports of $\mathcal{P}_{\text{sel}}$; its results are saved and reused in the discovery step. Thus the only extra overhead is the training cost; it is quite small (pp. 10).

Indeed, $\mathcal{M}_{\text{Corr}}$ is implemented with DQN of 2 hidden layers. Denote the dimension of each layer as h; then $|\mathcal{M}_{\text{Corr}}|$ is in $O(h(h + |\mathcal{P}_0|))$. In practice, h is relatively small, *e.g.,* $h \leq 10^3$ [47]. As the number of epoch (*i.e.,* iteration) to train $\mathcal{M}_{\text{Corr}}$ is a constant value, the learning time of $\mathcal{M}_{\text{Corr}}$ is bounded by $O(Nh(h + |\mathcal{P}_0|))$.

(2) We have also elaborated the training across samples (Section 5.3, pp. 8). To be specific, in the multi-round sampling, the predicate correlation training is independent for different samples and thus we separately learn $\mathcal{M}_{\text{Corr}}$ pertaining to each sample. (a) Model $\mathcal{M}_{\text{Corr}}$ in the i-th round aims to learn the predicate correlation in the i-th sample, and it might not be accurate for samples of, *e.g.,* the j-th one for $i \neq j$. Thus, considering the small training overhead and strong correlation of $\mathcal{M}_{\text{Corr}}$ with the data distribution, we train an $\mathcal{M}_{\text{Corr}}$ in each round. (b) As discussed above, the only extra overhead is the training cost, which does not dominate the cost of the rule discovery in the multi-round sampling. Moreover, as shown in Figures 7(g)-7(j), k is set to be 1 to 3 by Theorem 1, and PRMiner is faster than PRMiner$_{\text{noml}}$, indicating that $\mathcal{M}_{\text{Corr}}$ indeed accelerates the discovery process with multi-round sampling.

[O5] *While discovering templates over the sample $\mathcal{D}_s$ saves enumeration of constants, a template may occur in $\mathcal{D}_s$ but not in $\mathcal{D}$, leading to insufficient support and confidence. Similarly, templates may exist in $\mathcal{D}$ and not in the sampled $\mathcal{D}_s$. How does the model handle the latter case?*

[A] Thanks! We have added discussion about this (Section 5.3, pp. 9). Yes, missing a small number of templates is inevitable due to the random mechanism. Nonetheless, we show that the missed cases are under control. (1) Our theoretical bound (Theorem 1) guarantees that our sampling method would not miss many cases. The more rounds are used, the less cases are missed. (2) As shown in the experiments, the recall retains a relatively large value, *e.g.,* 0.82 on average in Figures (a) and (c). Moreover, the discovery cost is substantially reduced (see Figures 7(g)-7(h)).

[O6] *The sampling evaluation is done over a relatively small k (up to 4), with datasets that are up to 170K, and only over 1 relation. Including more datasets with multi-relations (which is one of the* *benefits of* REEs*, and there is only* DBLP*), and over larger k would better demonstrate the scalability and accuracy of discovering accurate rules when the samples may not be representative of the entire dataset.*

[A] As suggested, we have (1) added a new dataset Realty with 12 relational tables (shown in Table 4 on pp. 11), and (2) evaluated our sampling algorithm over Airport, Inspection, Realty and Tax using larger k in Figures 7(c), 7(d), 7(e) and 7(f). We have also added discussions about the performance of the algorithm on the new dataset (Section 6, the accuracy of Exp-1, pp. 11), *e.g.*, recall of at least 0.91 over Realty when $k = 7$; this justifies the effectiveness of RandomWalkSampling.

We have also compared the discovery time with the state-of-the-art algorithms over Realty, as shown in Figures 7(n) and 7(q) (Section 6, pp. 12), and added discussions to Exp-2 (varying σ, δ and k on pp. 13). These verified that our discovery algorithm still performs well over datasets with multiple relations.

[O7] *Source code and data were not available, please indicate if this will be done so in the future.*

[A] Thanks! We have submitted the code and datasets after getting approval from our industry collaborators.

Many thanks for your support and comments!

Response to the comments of Referee 5.

[O1] *My main concern is the general presentation of the paper which is confusing, to my opinion (especially the repartition between Sections/Subsections/...).*

To be specific, in section 2 subsections "predicate", "REEs" are formal definitions without any introductory text, while subsection "Pattern format" contains formal definitions, illustrations and a subsubsection "pattern tableau".

In section 3.1, second paragraph begins with the first definition "We first define an order between rules. Given two rules...", then a second one "We use the following notions [3]. Given ..." and so on. You also use paragraph names to introduce definition ("Minimality", "Cover of rules", ...). This also occurs in some other places in the paper (e.g., Section "Template pattern discovery in Ds" with "Given a template predicate $p : t.A \oplus \_$, a valuation h is said to satisfy p, written as $h \models p$, if $h(t)$".

In my opinion, the paper can benefit from informal introduction in these sections/paragraphs, while the formal part of these definitions should be moved into the dedicated environment. Obviously, I understand that the space is limited, thus that my comment might be applied only to highlight the most important definitions.To give a few examples to illustrate O1 (which is closely related to O3): in section 2 subsections "predicate", "REEs" are formal definitions without any introductory text, while subsection "Pattern format" contains formal definitions, illustrations and a subsubsection "pattern tableau".

[A] Thanks! We have revised the presentation of the paper as suggested. (1) We have added introductory texts before definitions (Sections 2-3, pp. 3-4) and in template pattern (Section 5.3, pp. 8). (2) We have put formal notions in the definition environments (Sections 2-3, 5, pp. 3-4 and 6; please also see the response to **[O3]**).

[O2] *The QED square as a meaning and is not used properly here. It should end the proof of a theorem/lemma, not the theorem/lemma themselves and not it should not end an example.*

[A] As suggested, we have left out QED from theorems, lemmas and examples.

[O3] *Section 3, for example, deserves a few more Definition environment (e.g.: at the beginning of section 3.1, you "first define an order between rules" which deserves its own environment to highlight it in your text. Same goes for the notions borrowed from [3] (support,...).*

[A] As suggested, we have put important notions in the definition environments, including predicates, REEs, support, confidence, cover of rules, and order of rules (Sections 2-3, pp. 3-4). We have also put parallel scalability in the enironment (Section 5, pp. 6).

[O4] *I would have appreciated having the standard deviation bar or, even better, the boxplot of the results to have a better intuition of the results dispersion.*

[A] As suggested, we have revised Figure 7(t) (pp. 12) by adding the standard deviation bar for better intuition by running our methods 5 times. For the other figures, the boxplot and standard deviation bar that indicate the data dispersion might not be suitable in our experimental setting, because each algorithm outputs only a single result (*e.g.*, running time, recall, or precision) for each figure.

Many thanks for your support and comments!

Parallel Rule Discovery from Large Datasets by Sampling

Paper id: 438

ABSTRACT

Rule discovery from large datasets is often prohibitively costly. The problem becomes more staggering when the rules are collectively defined across multiple tables. To scale with large datasets, this paper proposes a multi-round random-walk sampling strategy for rule discovery. We consider entity enhancing rules (REEs) for collective entity resolution and conflict resolution, which may carry constant patterns and machine learning predicates. We sample large datasets with accuracy bounds α and β such that at least α% of rules discovered from samples are guaranteed to hold on the entire dataset (*i.e.,* precision), and at least β% of rules on the entire dataset can be mined from the samples (*i.e.,* recall). We also quantify the connection between support and confidence of the rules on samples and their counterparts on the entire dataset. To scale with the number of tuple variables in collective rules, we adopt deep Q-learning to select semantically relevant predicates. To improve the recall, we develop a tableau method to recover constant patterns from the dataset. We parallelize the algorithm such that it guarantees to reduce runtime when more processors are used. Using real-life and synthetic data, we empirically verify that the method speeds up REE discovery by 12.2 times with sample ratio 10% and recall 82%.

1 INTRODUCTION

Rule discovery has been a longstanding challenge for decades. To make practical use of rules such as functional dependencies (FDs [15]), conditional functional dependencies (CFDs [18]), denial constraints (DCs [5]) and matching dependencies (MDs [17]), we need to discover reliable rules from real-life datasets such that the rules can be frequently applied (*i.e.,* support) and frequently hold on the data (*i.e.,* confidence). While a number of discovery algorithms have been developed, the need for more scalable methods is evident. A recent study shows that it takes "days or longer" when mining FDs on a dataset with 100 attributes and 300K tuples [50].

The scalability problem of rule discovery becomes more staggering for rules beyond FDs. Recall that FDs $X \rightarrow Y$ are defined with *two relation atoms on a single table.* It has long been recognized that to accurately resolve conflicts (conflict resolution) and identify tuples that refer to the same tuple (entity resolution), one needs *collective* rules defined with multiple relation atoms to correlate information across tables [6, 22]. The cost of collective rule discovery is inherently exponential in the size of datasets since it requires to join multiple tables, no matter whether we use levelwise method [30] or depth-first search [68]. Worse yet, for rules that support constant patterns such as CFDs and DCs, the discovery cost inevitably increases since such patterns have to be enumerated.

To scale with large real-life datasets, a variety of sampling methods have been proposed [7–11, 14, 29, 31, 35, 41, 43, 45, 46, 51, 65]. The idea is to pick a representative sample $\mathcal{D}_s$ of a real-life dataset $\mathcal{D}$, and discover rules from smaller $\mathcal{D}_s$ instead of directly from large $\mathcal{D}$. However, these methods typically target rules that are defined on a single table with at most two relation atoms.

The sampling approach gives rise to several questions. Is there a sampling method that guarantees the accuracy of the rules Σ_s discovered from sample $\mathcal{D}_s$ *w.r.t.* the rules Σ mined from the entire dataset $\mathcal{D}$? That is, can we ensure that most rules in Σ_s hold on the dataset $\mathcal{D}$ and hence are also in Σ (precision), and moreover, most rules in Σ can be discovered from $\mathcal{D}_s$ and hence be covered by Σ_s (recall)? In addition, suppose that we discover Σ from $\mathcal{D}$ with thresholds σ and δ for support and confidence, respectively; then what thresholds and sampling size should we adopt for mining Σ_s from $\mathcal{D}_s$ to ensure a reasonable accuracy? How can we discover collective rules across multiple tables? If the rules support constant patterns like CFDs, how can we retrieve all such patterns when $\mathcal{D}_s$ inevitably drops constants from $\mathcal{D}$? When sampling alone does not suffice to scale with large datasets, *e.g.,* if the required samples are still large to meet an accuracy bound, is it possible to have a parallel algorithm that scales with the number of processors used?

Contributions & organization. This paper tackles these issues. As a testbed of our proposed method, we consider the class of Entity Enhancing Rules (REEs) [21, 22], which subsume CFDs, DCs and MDs as special cases, and are collectively defined across different tables with multiple relation atoms. As opposed to the previous rules, REEs specify rules for both entity resolution (ER) and conflict resolution (CR), and unify machine learning (ML) and rule-based methods by embedding ML models for ER, CR and similarity checking as predicates in logic rules. Like CFDs and DCs, REEs support constant patterns of semantically related data. REEs are the rules underlying Rock, an industrial system for data cleaning. In this paper, we propose a novel sampling framework with optimization strategies for REE discovery. This said, the techniques can also be used to discover CFDs, DCs and MDs, which are special cases of REEs.

(1) Generic rules (Section 2). We introduce a representation of REEs [21, 22] by means of a *tableau* that specifies constant patterns, which helps us efficiently retrieve such patterns. We also identify practical ML models that can be plugged into REEs as predicates.

(2) REE *discovery problem* (Section 3). We formulate the discovery problem for REEs with sampling. We also characterize the accuracy of the set Σ_s of REEs discovered from sample $\mathcal{D}_s$, relative to the set of Σ mined from the entire dataset $\mathcal{D}$, in both *precision* and *recall*.

(3) A sampling strategy (Section 4). We propose a multi-round random-walk sampling strategy to discover REEs independently in each sample. We prove an accuracy bound and deduce a sampling size for the sampling method. Under a given accuracy bound, we establish a connection between the thresholds of support and confidence for mining REEs on samples and the thresholds on the entire datasets, as a guidance for scaling samples and mining *valid* rules.

(4) Discovering REEs (Section 5). We present a parallel algorithm to discover REEs with sampling, denoted as PRMiner. We propose to train a reinforcement learning model and select semantically correlated predicates with the model to mine collective rules across multiple tables. We also show how we populate tableaux to efficiently retrieve constant patterns without enumerating numerous constant values. We prove that PRMiner is parallely scalable, *i.e.,* it guarantees to take less runtime when more processors are used.

(6) Experimental study (Section 6). Using real-life and synthetic data, we empirically verify the following. On average, (a) with sample ratio 10%, PRMiner speeds up rule discovery by 12.2 times with precision 90% and recall 82%, up to 93% and 85%, respectively; it improves random sampling by 13% in recall. (b) It scales well with the number n of processors, *e.g.,* it is 3.38 times faster when n varies from 4 to 20. (c) Employing ML correlation model in discovery improves the efficiency by 75%, with only 3% and 2% lower in precision and recall, respectively. (d) Constant pattern recovery further improves the recall by 3% on average, up to 5%.

Related work. We categorize the related work as follows.

Sampling methods. A variety of sampling methods have been explored for discovering rules. (1) *Random (uniform) sampling*. The method is commonly used for its simplicity. For association rule mining, the study has mostly focused on deriving size bounds for a single sample to obtain high-quality approximation, *e.g.,* [8, 46, 65] adopt random sampling to derive the sample size with Chernoff bounds and union bound, such that the frequencies of itemsets in the sample are close to the exact ones. The bounds are improved by using, *e.g.,* the central limit theorem [32, 40, 72] and hybrid Chernoff bounds [74]. For data quality rule, [43] uses uniform sampling to discover DCs using Chebyshev inequality such that for each rule, it estimates the number of its violations on the entire dataset based on the sample and uses this estimate to derive the appropriate parameters for mining rules from the sample. Unfortunately, [43] limits the discussion on bi-variable DCs on a single relation, and do not consider constant patterns. (2) *Focused sampling*. [35] and [7] utilize auxiliary structures, *e.g.,* agree sets or evidence sets, for efficient rule discovery. Due to the high construction complexity, they adopt focused sampling to sample tuple pairs for constructing the desired structures. (3) *Progressive sampling*. To improve the loose bound of single random sampling, [9, 11, 14, 29, 31, 45, 51] adopt multi-iteration sampling, such that an initial random sample is first drawn and is then revised iteratively until a stopping condition is satisfied. Unfortunately, they fail to bound the sample size. In contrast, [55] mines frequent itemsets with size guarantee using the theorem of Rademacher Averages. (4) *Stratified sampling*. The method is also adopted for mining association rules [41] on a single relation with size bound for estimating the support of itemsets, and answering aggregated queries [69] to minimize the sample size while satisfying a predefined error bound. (5) *Heuristic sampling* for mining association rules by *e.g.,* [10]. Although the work offers no theoretical guarantee, the strategies often achieve high accuracy.

This work differs from the previous work in the following. (a) To the best of our knowledge, this work proposes the first sampling strategy with accuracy bounds to discover data quality rules that carry (i) more than two relation atoms across multiple tables, (ii) patterns of semantically related constants, and (iii) ML predicates. (b) We provide a bound on the sample size while guaranteeing both precision and recall instead of error rates. (c) We establish the connection between support and confidence thresholds on the sample and on the entire dataset. (d) We make a first effort to employ multi-round sampling in data quality rule discovery, to boost up the accuracy guarantee compared with single-round sampling.

Rule discovery. Abundant research efforts have been made on discovery algorithms for data quality rules, which can be classified as follows. (1) *Levelwise search*. Lattice traversal are widely adopted to discover FDs, *e.g.,* TANE [30], FUN [49], FD_mine [70]. Various techniques are also developed to speed up the levelwise search, *e.g.,* agree sets are utilized in Depmine [44], sampling techniques are combined in HyFD [50] to prune non-FDs, DynFD [59] focuses on FD discovery in dynamic datasets, and SMFD [25] proposes a top-down framework to discover and validate FDs in a secured multi-party scenario. Levelwise methods are also used to discover CFDs and MDs, *e.g.,* CTANE [19] and tableau generation [26] for CFDs, and similarity thresholds [62] for MDs. (2) *Depth-first search*. Depth-first traversal is also applied in the lattice for FD discovery, *e.g.,* DFD [2] mines FDs via random walk, FastFDs [68] improves Depmine by using different sets. More recently, depth-first search is extended to CFDs and DCs, *e.g.,* FastCFDs [19] extends FastFDs to mine CFDs, while FastDC [13], Hydra [7], DCFinder [52] and ADCMiner [43] use evidence sets and min-cover algorithms to mine *bi-variable* DCs, although by definition, DCs support multiple relation atoms [13]. (3) *Hybrid approaches*. HyMD [58] combines levelwise and depth-first approaches to mine MDs with pre-computed structures. MDedup [34] selects useful MDs discovered by HyMD. (4) *Learning-based approaches*. State-of-the-art learning methods have been used for rule discovery, *e.g.,* inductive learning in [23] to discover FDs, structure learning (*i.e.,* graphical lasso) in AutoFD [73] to mine FDs, and the rule learning strategies in [33, 61] to find MDs with similarity functions. More ER solvers can be found in [12].

This work differs from the prior work in the following. (1) Our multi-round sampling strategy is very flexible and can also be combined with state-of-the-art rule discovery algorithms, *e.g.,* DCFinder [52] and HyMD [58]. (2) We propose a strategy to discover collective rules across multiple tables by employing a reinforcement learning model. (3) We propose a method for efficiently retrieving constant patterns, to supplement sampling strategies, which inevitably drop constants from the original datasets.

Parallel rule discovery. Several parallel algorithms are already in place for rule discovery. Massive parallelism is employed in [24, 38] to discover FDs. However, they do not take communication cost into consideration. [39] mines FDs in a distributed setting but it returns local FDs only. Communication cost is minimized in [56], which is extended from FastFD. A distributed framework is proposed in [57], which is capable of discovering both FDs and DCs, based on a set of primitives; however, the primitives do not support rules with constant patterns, ML predicates and multiple tuple variables; while in theory, the primitives can be extended to deal with these, it incurs an exponential cost in the number of tuple variables. SMFD [25] only focuses on how to enforce privacy constraints, although it discovers FDs based on the lattice structure in a distributed manner.

Different from the existing works, we develop a discovery algorithm that guarantees the parallel scalability, when both computational and communication costs are considered, and supports mining rules with constants, ML models and multiple tuples.

2 COLLECTIVE RULES WITH ML MODELS

We next present entity enhancing rules (REEs) defined in [21, 22].

Consider database schema $\mathcal{R} = (R_1, \ldots, R_m)$, where R_j is a schema $R(A_1 : \tau_1, \ldots, A_n : \tau_n)$, and each A_i is an attribute of

tid	oid	org_name	zipcode	org_address	city	country
t_1	o_1	Guangzhou No One School	510375	Liwan Guangzhou Guangdong	GZ	CN
t_2	o_2	Guangzhou No.1 Middle School	510000	Liwan District, GZ, Guangdong	GZ	CN
t_3	o_3	Indiana U., Department of Biology	47401	Indiana Ave, IN, USA		USA
t_4	o_4	Indiana Univ., Computer Science	47401	IN	Bloomington	US

Table 1: Example Organization (Org) **relation** D_1

tid	pid	oid	person_name	title	major	person_address	nationality
t_5	p_1	o_1	Qiang Zhang	Teacher	math	Liwan, Guangzhou, GD	CN
t_6	p_2	o_1	Qiang Zhang	Teacher	math	Guangzhou	CN
t_7	p_3	o_3	Matthew Hahn	Prof.	Bioinformatics	Bloomington, IN, USA	USA
t_8	p_4		M. Hahn	Prof.	Bio.		

Table 2: Example Person (Pers) **relation** D_2

type τ_i. We assume *w.l.o.g.* that each tuple t in D has an id attribute, which uniquely defines the entity that t represents. An instance $\mathcal{D}$ of $\mathcal{R}$ is a collection $(D_1, \ldots, D_m)$, where each D_i is a relation of R_i.

Predicates. Predicates are the atmoic formats of tuple relational calculus [3] and are used to represent correlations among attributes. They constitute the basic components of REE rule.

Definition 2.1: *We define* predicates *(i.e., atomic formulas) over a database schema* $\mathcal{R}$ *as follows:*

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

where $\oplus$ *is either* $=$ *or* $\neq$*. Following tuple relational calculus (see,* e.g., *[3]), (1)* $R(t)$ *is a* relation atom *over schema* $\mathcal{R}$*, 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.*

REEs. Employing predicates, we next define entity enhancing rules, which subsume most of existing data quality rules [21].

Definition 2.2: *An* entity enhancing rule *(*REE*)* φ *over a database schema* $\mathcal{R}$ *is a first-order logic formula and is defined as*

$$\varphi : X \to p_0,$$

where (1) X *is a conjunction of* predicates *over* $\mathcal{R}$*, and (2)* p_0 *is a predicate over* $\mathcal{R}$ *such that all tuples variables in* φ *are bounded in* X*. We refer to* X *as the* precondition *of* φ *and* p_0 *as the* consequence *of* φ*.*

Example 1: *Consider an organization database with two self-explained relation schemas:* Org (oid, org_name, zipcode, org_address, city, country) *in Table 1, and* Pers (pid, oid, persona_name, title, major, person_address, nationality*) in Table 2. Some example* REEs *over the database schema are given as follows.*

(1) φ_1 : $\text{Org}(t_a) \wedge \text{Org}(t_b) \wedge \mathcal{M}_{\text{Bert}}(t_a[\bar{A}], t_b.[\bar{A}]) \to t_a.\text{zipcode} = t_b.\text{zipcode}$*. Here we use the state-of-the-art ML model* $\mathcal{M}_{\text{Bert}}$ *[16] to check the semantic* similarity *of text attributes, where* $\bar{A}$ *denotes* (org_name, org_address) *in relation* Org*. Intuitively,* REE φ_1 *states that if two organizations have semantically similar names and addresses (checked by* $\mathcal{M}_{\text{Bert}}$*), then they have the same zipcode.*

(2) φ_2 : $\text{Pers}(t_a) \wedge \text{Pers}(t_b) \wedge \text{Org}(s_a) \wedge \text{Org}(s_b) \wedge t_a.\text{title} = t_b.\text{title} \wedge \mathcal{M}_{\text{Bert}}(t_a.\text{person\_name}, t_b.\text{person\_name}) \wedge \mathcal{M}_{\text{LP}}(t_a, s_a) \wedge \mathcal{M}_{\text{LP}}(t_b, s_b) \wedge s_a.\text{oid} = s_b.\text{oid} \to t_a.\text{pid} = t_b.\text{pid}$*. Here we use a* link prediction *model* $\mathcal{M}_{\text{LP}}(t, s)$ *[60, 64] to predict whether person* t *works in organization* s *(note that* Pers *might not record the fact "*t *works in* s*"). Intuitively, this* REE *collectively utilizes the information from both* Org *and* Pers *to identify two persons if they have similar names, same titles and works in same organizations. The* REE *is defined across three tables in terms of four tuple variables.*

(3) φ_3 : $\text{Org}(t_a) \wedge \text{Org}(t_b) \wedge X \to \mathcal{M}_{\text{addr}}(t_a.\text{org\_address}, t_b.\text{org\_address})$*, where* $\mathcal{M}_{\text{addr}}$ *is an ML model for checking the* closeness *of addresses, and* $X = \bigwedge_{A_s \in \mathcal{T}} t_a.A_s = t_b.A_s$ *and* $\mathcal{T}$ *denotes a designated set of attributes in* Org *(which are not shown in the simplified schema), including built-up area, located city, and neighborhood information. Here the conditions in* X *interpret the prediction of* $\mathcal{M}_{\text{addr}}(t_a.\text{org\_address}, t_b.\text{org\_address})$ *in logic. That is,* $\mathcal{M}_{\text{addr}}$ *predicts true because of the logic characteristics in* X*.*

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

Valuation h *satisfies* a predicate p, written as $h \models p$, if the following are satisfied: (1) If p is a relation atom $R(t)$, $t \oplus c$ 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 [3]. (2) If p is $\mathcal{M}(t[\bar{A}], s[\bar{B}])$, then $h \models p$ if $\mathcal{M}$ predicts true on $(h(t)[\bar{A}], h(s)[\bar{B}])$.

For a set X of predicates, we write $h \models X$ if $h \models p$ for *all* predicates p in X. For an REE φ, we write $h \models \varphi$ such that if $h \models X$, then $h \models p_0$. An instance $\mathcal{D}$ of $\mathcal{R}$ *satisfies* φ, denoted by $\mathcal{D} \models \varphi$, if $h \models \varphi$ *for all* valuations h of φ in $\mathcal{D}$. We say that $\mathcal{D}$ *satisfies* a set Σ of REEs, denoted by $\mathcal{D} \models \Sigma$, if for each REE $\varphi \in \Sigma$, $\mathcal{D} \models \varphi$.

Example 2: *Continuing with Example 1, assume that* $\mathcal{D}$ *consists of two relations* D_1 *and* D_2 *of schemas* Org *and* Pers*, shown in Tables 1 and 2, respectively. Consider valuation* h_2: $t_7 \mapsto t_a$, $t_8 \mapsto t_b$, $t_3 \mapsto s_a$ *and* $t_4 \mapsto s_b$*. It satisfies* REE φ_2*, since* p_3 *and* p_4 *have similar names, the same titles and their working organizations are predicted to be the same by the link prediction model* $\mathcal{M}_{\text{LP}}$*.*

As another example, consider the valuation h_1 *of* REE φ_1 *that has mappings:* $t_1 \mapsto t_a$ *and* $t_2 \mapsto t_b$*. It helps us fix conflicting values in* zipcode *of* t_1 *and* t_2*, which have similar names and addresses.*

ML predicates. REEs can embed ML classifiers of the following.

- NLP models. REEs can embed language models, such as NER [27, 71], for text classification and semantic matching.
- ER models. One can use entity resolution and link prediction classifiers for multi-attribute record matching, *e.g.*, ditto [42], DeepMatcher [48], which return Boolean values.
- CR models. We can also plug in classifiers for data fusion and error detection *e.g.*, HoloClean [54] and HoloDetect [28].

Pattern format. We call an REE expressed in $X \to p_0$ as its *regular format*. Equivalently, φ can be expressed in one-to-one corresponding *pattern format* $\varphi = (P \to Q, t_p)$ where P and Q are tuple attributes and operators used in X and p_0, respectively, and t_p is a pattern tuple, indicating how constant values and operators are applied in P and Q. To illustrate, consider a regular format REE:

$$\varphi : R(t) \wedge R'(s) \wedge t.A \neq s.B \wedge \mathcal{M}(t.C, s.D) \wedge t.E = c \wedge s.F \neq d \to t.G = e.$$

Its pattern format is expressed as: $\varphi = (P \to Q, t_p)$ where

$$P = [t.A, t.C, t.E, s.B, s.D, s.F, s.G, \oplus_1, \oplus_2], \; Q = [t.G],$$
$$t_p = (\oplus_1.\text{left}, \oplus_2.\text{left}, c, \oplus_1.\text{right}, \oplus_2.\text{right}, \bar{d}, \neq, \mathcal{M} \| e),$$

and $\oplus_i$.left (resp. $\oplus_i$.right) denotes the left (resp. right) operand of

$\oplus_i$. Note that the constant value of $t.E$ is directly assigned in t_p; similarly for $s.F$ and $t.G$ (a bar above indicates an inequality).

Note that we can easily *recover* a pattern format REE $\varphi = (P \to Q, t_p)$ to its regular format, *e.g.,* to recover X from P, (1) for each operator $\oplus_i$ in P whose assigned value in t_p is "=" (resp. "$\neq$"), we construct a predicate $p : \oplus_i.\text{left} = \oplus_i.\text{right}$ (resp. $p : \oplus_i.\text{left} \neq \oplus_i.\text{right}$) in X where $\oplus_i$.left and $\oplus_i$.right are the operands specified in t_p; (2) for each operator $\oplus_i$ in P whose assigned value in t_p is the ML model "$\mathcal{M}$", we construct a predicate $p : \mathcal{M}(\oplus_i.\text{left}, \oplus_i.\text{right})$ in X; and (3) for each attribute $t.A$ in P whose constant value c (resp. $\bar{d}$) is directly assigned in t_p, we construct a predicate $p : t.A = c$ (resp. $p : t.A \neq d$) in X. The recovery of p_0 from Q is similar.

Pattern tableau. When multiple REEs only differ in the constant values in constant predicates, they can be expressed concisely in one tableau REE in the form $\varphi_{\mathcal{T}} = (P \to Q, \mathcal{T}_p)$, where $\mathcal{T}_p$ is a *pattern tableau* consisting of a finite number of pattern tuples. For example, we can define a pattern tableau $\varphi_{\mathcal{T}} = (P \to Q, \mathcal{T}_p)$ using the same P and Q defined above, and $\mathcal{T}_p = \{t_p^1, \ldots, t_p^k\}$ where

$$t_p^i = (\oplus_1.\text{left}, \oplus_2.\text{left}, c_i, \oplus_1.\text{right}, \oplus_2.\text{right}, \bar{d}_i, \neq, \mathcal{M}||e_i).$$

The pattern tableau $\mathcal{T}_p$ can be classified into two categories: (1) *constant pattern*, if $\mathcal{T}_p$ contains at least one constant value, and (2) *variable pattern*, if $\mathcal{T}_p$ does not have any constant value. Later, we will use pattern tableau to retrieve constant patterns from the data.

3 RULE DISCOVERY WITH SAMPLING

In this section, we first present notions of rule discovery and formulate the accuracy of rules mined from samples (Section 3.1). We then state the rule discovery problem with sampling (Section 3.2).

3.1 Preliminary

In the literature of rule discovery, the validity of rules is usually measured by two measurements, namely, *support* and *confidence*.

Support. Support measures how frequently a rule can be applied. We adopt the support defined in [4] for REEs, which is among the first support notation for collective rules with *anti-monotonicity*.

To illustrate support, we first define an order on rules.

Definition 3.1: *Given two* REEs $\varphi : X \to p_0$ *and* $\varphi' : X' \to p_0$, *we say that* φ *has a* lower order *than* φ', *denoted by* $\varphi \preceq \varphi'$, *if* $X \subset X'$. *Intuitively,* φ *is less restrictive than* φ'.

We use the following notions [4]. Given a predicate p, we define an REE φ_p to verify whether two tuples satisfy p: $R(t) \wedge R'(s) \to p$, where t and s (of relation schema R and R', respectively) are the tuple variables used in p. Let H_p be the set of valuations of φ_p in $\mathcal{D}$. We define the *support set* of p on $\mathcal{D}$, denoted by spset$(p, \mathcal{D})$, as

$$\text{spset}(p, \mathcal{D}) = \{\langle h(t), h(s)\rangle \mid h \in H_p \wedge h \models \varphi_p\},$$

i.e., the set of tuple pairs satisfying p. Similarly, given a conjunction X of predicates, we define the support set of X as follows:

$$\text{spset}(X, \mathcal{D}) = \{\langle h(t), h(s)\rangle \mid \forall p \in X(\langle h(t), h(s)\rangle \in \text{spset}(p, \mathcal{D}))\},$$

i.e., the set of all tuple pairs satisfying *all* predicates in X.

Given $\varphi : X \to p_0$, assume that H is the set of all valuations of φ in $\mathcal{D}$, and t_0 and s_0 are the tuple variables used in p_0. Then the *support set* of φ, denoted by spset$(\varphi, \mathcal{D})$, is defined as

$$\text{spset}(\varphi, \mathcal{D}) = \{\langle h(t_0), h(s_0)\rangle \mid h \in H \wedge h \models X \wedge h \models \varphi\}.$$

Definition 3.2: *The* support *of* φ *to quantify its frequency is*

$$\text{supp}(\varphi, \mathcal{D}) = |\text{spset}(\varphi, \mathcal{D})|.$$

Similarly we define the notions of supp$(p, \mathcal{D})$ and supp$(X, \mathcal{D})$.

For an integer σ, an REE is σ-*frequent* on $\mathcal{D}$ if supp$(\varphi, \mathcal{D}) \geq \sigma$.

It is shown in [4] that the above notation of support satisfies the anti-monotonicity, *i.e.,* if $\varphi \preceq \varphi'$, then supp$(\varphi, \mathcal{D}) \geq$ supp$(\varphi', \mathcal{D})$.

Example 3: *Let* X *be* Org$(t) \wedge t$.zipcode = 510375 *and* p_0 *be* t.city = *GZ. Consider two* REEs $\varphi : X \to p_0$ *and* $\varphi' : X' \to p_0$, *where* $X' = X \wedge$ Pers$(s) \wedge t$.oid = s.oid. *Clearly,* $\varphi \preceq \varphi'$ *since* $X \subset X'$. *Then* supp$(\varphi, \mathcal{D}) \geq$ supp$(\varphi', \mathcal{D})$, *since* spset$(\varphi, \mathcal{D})$ = spset$(\varphi', \mathcal{D})$ = $\{t_1 \mapsto t\}$, i.e., *anti-monotonicity. Although* t_1 *can join with two tuples,* t_5 *and* t_6, *in* φ', *it does not lead to a larger support.*

Confidence. Confidence indicates how often an REE $\varphi : X \to p_0$ has been found to be true, given that X is satisfied.

Definition 3.3: *Given an* REE $\varphi : X \to p_0$, *the* confidence *of* φ *on* $\mathcal{D}$, *denoted by* conf$(\varphi, \mathcal{D})$, *is defined to be* conf$(\varphi, \mathcal{D}) = \frac{\text{supp}(X \wedge p_0, \mathcal{D})}{\text{supp}(X, \mathcal{D})}$.

For a threshold δ, an REE is δ-*confident* on $\mathcal{D}$ if conf$(\varphi, \mathcal{D}) \geq \delta$.

Minimality. An REE $\varphi : X \to p_0$ over $\mathcal{R}$ is said to be *trivial* if $p_0 \in X$. In the rest of this paper, we only consider non-trivial REEs.

An REE $\varphi : X \to p_0$ is *left-reduced* on $\mathcal{D}$ if φ is σ-frequent, δ-confident and moreover, there exists no REE φ' such that $\varphi' \preceq \varphi$ and φ' is σ-frequent and δ-confident. Intuitively, it means that no predicate in X can be removed, *i.e.,* the minimality of predicates.

A *minimal* REE φ on $\mathcal{D}$ is a non-trivial and left-reduced REE. Intuitively, there is no redundancy in a minimal REE.

Cover of rules. Consider a set Σ of minimal rules on $\mathcal{D}$.

We say that Σ *entails* another rule φ over $\mathcal{R}$ denoted by $\Sigma \models \varphi$, if for any instance $\mathcal{D}$ of $\mathcal{R}$, if $\mathcal{D} \models \Sigma$ then $\mathcal{D} \models \varphi$.

We say that Σ is *equivalent* to another set Σ' of rules, denoted by $\Sigma \equiv \Sigma'$, if $\Sigma \models \varphi'$ for all $\varphi' \in \Sigma'$ and vice versa.

We say that Σ is *minimal* if for all rules $\varphi \in \Sigma$, $\Sigma \not\equiv \Sigma \setminus \{\varphi\}$, *i.e.,* Σ includes no redundant rules.

Definition 3.4: *A* cover Σ_c *of* Σ *on* $\mathcal{D}$ *is a subset of* Σ *such that (a)* $\Sigma_c \equiv \Sigma$, *(b) all rules* φ *in* Σ_c *are* minimal, i.e., *non-trivial and left-reduced. (c)* Σ_c *is* minimal, i.e., Σ_c *contains no redundant rules [3].*

Accuracy. Let $\mathcal{D}_s$ be a sample picked from $\mathcal{D}$, and Σ_s and Σ be the set of minimal rules discovered by the same discovery algorithm on datasets $\mathcal{D}_s$ and $\mathcal{D}$, respectively. To quantify the effectiveness of the sampling strategy, we measure its performance by *precision* and *recall*. Here precision, denoted by precision(Σ, Σ_s), reports the percentage of rules in Σ_s that also hold on $\mathcal{D}$, and recall, denoted by recall(Σ, Σ_s), is defined to be the percentage of rules in Σ that can be discovered from $\mathcal{D}_s$ and thus, covered by Σ_s.

3.2 Problem Statement

Assume that Σ_s and Σ are the set of minimal rules mined by the same discovery algorithm on $\mathcal{D}_s$ and $\mathcal{D}$, respectively. Clearly, Σ_s and Σ may contain an excessive number of rules that are not very relevant to users' applications and interests. To reduce such rules, we adopt the following strategies. (1) We pick an application-dependent set of candidate consequences p_0, denoted

symbols	notations
$\mathcal{D}$, $\mathcal{D}_s$	dataset, and samples of the dataset
φ	REE $X \to p_0$
Σ, Σ_s	the sets of REEs discovered from $\mathcal{D}$ and $\mathcal{D}_s$, respectively
α, β	bounds for precision(Σ, Σ_s) and recall(Σ, Σ_s), respectively
σ, δ	thresholds for support and confidence, respectively
RHS	the set of candidate consequences
$\mathcal{P}_0$	the set of relevant predicates

Table 3: Notations

by RHS, which pertains to a particular application of users. (2) For each p_0 in RHS, we focus on discovering REEs φ: $X \to p_0$ such that $X \subseteq \mathcal{P}_0$ where $\mathcal{P}_0$ is a *subset* of semantically relevant predicates related to p_0. The *discovery problem with sampling* is as follows.

- *Input*: A database schema $\mathcal{R}$, a database $\mathcal{D}$ of $\mathcal{R}$, a consequence set RHS, the support threshold σ, the confidence threshold δ, the precision threshold α and the recall threshold β.
- *Output*: A cover Σ_s of REEs on a sample $\mathcal{D}_s$ picked from $\mathcal{D}$ such that (1) precision$(\Sigma, \Sigma_s) \geq \alpha$, recall$(\Sigma, \Sigma_s) \geq \beta$, and (2) for each REE $\varphi : X \to p_0$ in Σ_s, (a) $p_0 \in$ RHS; (b) $X \subseteq \mathcal{P}_0$, where $\mathcal{P}_0$ is a set of semantically relevant predicates related to p_0; and (c) φ is minimal and moreover, it is σ-frequent and δ-confident.

As verified by our industry collaborators, it is not very difficult for practitioners to select a few predicates in RHS since they often have a good understanding of their datasets and know their pain points. They only care about key attributes and are able to pick RHS by referencing the schema of datasets. For novice users, they can simply start with all predicates as RHS and then narrow down to what they need. A real-life case is deduplication in a particular dataset; in that case users simply choose the predicate t_0.id = t_1.id as the RHS predicate and only discover relevant ER rules.

The notations of the paper are summarized in Table 3.

4 SAMPLING WITH ACCURACY BOUNDS

In this section, we first propose the sampling method by adapting the random walk strategy (Section 4.1). We then establish the accuracy bounds (precision and recall) of the method (Section 4.2).

4.1 Sampling Strategy

Our sampling method is designed based on the random walk strategy [37], such that rules with multiple relational atoms can be discovered with theoretical guarantee. We first present a single-round sampling method and then extend it to multi-round sampling.

Single-round sampling. One might be tempted to uniformly sample tuples from $\mathcal{D}$. This is, however, impractical, because different predicates have different selectivity, *i.e.,* some predicates (*e.g.,* $t.A \neq s.B$) can be satisfied by a large number of tuples and thus, they have low selectivity. Hence, uniform sampling from $\mathcal{D}$ might cause bias towards low selectivity predicates while those rules defined with high selectivity predicates (*e.g.,* equality and ML predicates) are difficult to be discovered. In light of this, we transform the problem of sampling in $\mathcal{D}$ to sampling in a graph G, in which high selectivity predicates are explicitly considered.

Specifically, we construct a graph $G = (V, E)$ for $\mathcal{D}$, where V is the set of vertices, each of which corresponds to a tuple in $\mathcal{D}$, and E is the set of edges, such that if $e = \langle v_1, v_2 \rangle$ is an edge in E, the tuple pair corresponding to $\langle v_1, v_2 \rangle$ satisfies at least one equality or ML predicate. For simplicity, we use the words "vertex" and "tuple" interchangeably. Given the maximum length len of a random walk, we can adopt the random walk algorithm [37] to sample random walks in G with length at most len. However, since $\mathcal{D}$ can be large, constructing G could be time-consuming. Motivated by this, we adopt an online sampling strategy to pick sample $\mathcal{D}_s$ from $\mathcal{D}$.

```
Algorithm RandomWalkSampling
Input: The relational instances $\mathcal{D}$, the maximum length of a random walk.
       len, a sampling ratio $r$, and a termination probability $\epsilon$
Output: Sample $\mathcal{D}_s$.
1. Construct PLI and $\mathcal{L}_{\mathsf{ML}}$ for $\mathcal{D}$;
2. Array degree := GenDegree($\mathcal{D}$, PLI, $\mathcal{L}_{\mathsf{ML}}$); $\mathcal{D}_s = \emptyset$;
3. while $|\mathcal{D}_s| < |\mathcal{D}| \cdot r$ do
4.    Draw an initial sample $t$ in $\mathcal{D}$ according to its degree [t];
5.    Simulate a random walk with length at most len from $t$, i.e., terminate
       with probability $\epsilon$ or move to a neighbor with probability $(1 - \epsilon)$;
6.    Add all vertices in the random walk to $\mathcal{D}_s$;
7. return $\mathcal{D}_s$;
```

Figure 1: Algorithm RandomWalkSampling

The algorithm for online random walk sampling is shown in Figure 1. We first construct position list indexes (PLI, line 1) [52], which groups tuples by attribute values so that we can easily retrieve tuple pairs that satisfy a certain predicate and thus, are connected by an edge in G. For ML predicates p, we adopt the blocking algorithm of [22] to save the satisfied tuple pairs in an auxiliary structure $\mathcal{L}_{\mathsf{ML}}$, such that $\langle t, s \rangle \models p$ for $t \in \mathcal{L}_{\mathsf{ML}}$ and $s \in \mathcal{L}_{\mathsf{ML}}[t]$. Based on PLI and $\mathcal{L}_{\mathsf{ML}}$, we generate an array degree, where degree$[t]$ is the degree of t in G. We then iteratively sample tuples from $\mathcal{D}$ via random walk (line 3-6). We first draw an initial vertex t based on degree$[t]$ (line 4), such that t with higher degree$[t]$ is sampled with a higher probability. Then, we iteratively sample more tuples by simulating a random walk with length at most len, starting from t: at each step, the random walk either terminates with probability ϵ, or moves to a neighbor of the current vertex with $(1 - \epsilon)$ probability. Here we utilize PLI again to determine the neighbors of a given vertex. All vertices in the walk are included in the sample $\mathcal{D}_s$; The entire process iterates until $|\mathcal{D}_s|$ exceeds the maximum sample size.

Example 4: *Consider applying our random walk method to Tables 1 and 2. Suppose that t_4 is first sampled. Then t_3 is the only tuple to sample next, since it satisfies at least one predicate with t_1, e.g., $\langle t_1, t_3 \rangle \models p$ when p is t.zipcode = s.zipcode. In contrast, if the conventional random walk method is adopted, it may end up with $\mathcal{D}_s = \{t_1, t_4\}$, on which no equality or ML predicates hold; as a consequence, we cannot find useful rules such as* $\mathcal{M}_{\mathsf{Bert}}(t_0.\text{org\_name}, t_1.\text{org\_name}) \to t_0.\text{zipcode} = t_1.\text{zipcode}$.

Multi-round sampling. Single-round sampling suffers from the drawback of poor recall, *i.e.,* a rule that holds globally on $\mathcal{D}$ might not be discovered from the small sample $\mathcal{D}_s$. Therefore, we improve the recall by adopting a multi-round sampling strategy.

Specifically, we run RandomWalkSampling k times, and obtain k samples, namely $\mathcal{D}_s^1, \ldots, \mathcal{D}_s^k$. For each sample $\mathcal{D}_s^i$, we perform rule discovery to mine a set Σ_s^i of minimal REEs, and finally, we have $\Sigma_s = \bigcup_{i=1}^k \Sigma_s^i$. As will be seen, in Section 4.2, multi-round sampling improves the recall of a fixed size sample from 78% to 89%.

4.2 Theoretical Analysis

We next show that our multi-round random sampling method guarantees accuracy bounds for precision and recall. Recall that we consider the precision and recall computed by Σ_s and Σ that are discovered by the same discovery algorithm (Section 3.1).

Theorem 1: *Given a maximum length* len *of a random walk, a sampling ratio r, a precision bound α, a recall bound β, a support bound σ, a confidence bound δ, and an error parameter ϵ, a number Z of predicates*[1]*, the multi-round sampling method draws a set $\mathcal{D}_s$ of $\frac{1}{\epsilon^2 r\sigma}\log\left(\max\left\{\frac{2^{3\text{len}Z}}{(1-\alpha)^3}, \frac{2^{2\text{len}Z}}{(1-\beta)^2}\right\}\right)$ samples*[2]*, such that*

(1) precision$(\Sigma, \Sigma_s) \geq \alpha$ *and* recall$(\Sigma, \Sigma_s) \geq \beta$*, where Σ and Σ_s are the set of minimal rules on $\mathcal{D}$ and $\mathcal{D}_s$, respectively; and*

(2) such REEs *in Σ_s can be discovered by setting the support bound on $\mathcal{D}_s$ as $(1-\epsilon)r^2\sigma$ and confidence bound as $\frac{1-\epsilon}{1+\epsilon}\delta$.*

Proof sketch. The proof consists of three parts, for verifying the number of samples in $\mathcal{D}_s$, the support bound on $\mathcal{D}_s$ and the confidence bound on $\mathcal{D}_s$. Here we assume that the samples in $\mathcal{D}_s$ have the maximum size $|\mathcal{D}|r$. The proof is a little involved and is deferred to the full version due to the space constraint.

(1) Number of samples. Denote the number of samples in $\mathcal{D}_s$ by k. Denote $\mathbf{Pr}[X \leq (1-\epsilon)\mathbb{E}[X]]$ and $\mathbf{Pr}[X \geq (1+\epsilon)\mathbb{E}[X]]$ by p_h and q_h, respectively, for the probability of random variable X of each sample deviating from its expected value $\mathbb{E}[X]$. We deduce Chernoff bounds for p_h and q_h as functions of k to represent the probability of the support for all combinations of len relational atoms in $\mathcal{D}_s$ deviating from the support on $\mathcal{D}$. We then use recall and precision to bound the probability of not discovering a σ-frequent rule from $\mathcal{D}_s$, and the probability of discovering a non-σ-frequent rule from $\mathcal{D}_s$. We deduce k from the inequality involving the Chernoff bounds and the upper bound on the number of such rules, and obtain $k \geq \frac{1}{\epsilon^2 r\sigma}\log\left(\max\left\{\frac{2^{3\text{len}Z}}{(1-\alpha)^3}, \frac{2^{2\text{len}Z}}{(1-\beta)^2}\right\}\right)$.

(2) Support bound. We deduce the support bound on $\mathcal{D}_s$ as the minimal support within the error parameter ϵ in the sampling domain, by using ϵ, sampling ratio r and the support bound σ.

(3) Confidence bound. We deduce the confidence bound on $\mathcal{D}_s$ in terms of the error parameter ϵ and the confidence bound δ, with the probability that the preconditions of a rule meet the minimal support σ being at least the minimum of $1 - p_h$ and $1 - q_h$, which can be determined after we find the lower bound of k. □

Example 5: *Consider sampling from a database $\mathcal{D}$ of relation schemas Org and Pers (Example 1) by setting* len*, r and Z as 4, 0.1 and 8, respectively. To ensure that* precision$(\Sigma, \Sigma_s) \geq 0.9$ *and* recall$(\Sigma, \Sigma_s) \geq 0.7$*, we draw 4 samples, with support 10^4, confidence 0.9 and error rate 0.1 determined by Theorem 1. We can discover* REEs *such as φ_2 (see Example 1) from the samples when the support bound and the confidence bound on $\mathcal{D}_s$ are $(1-0.1) \cdot 0.1^2 \cdot 10^4 = 90$ and $\frac{1-0.1}{1+0.1} \cdot 0.9 = 0.74$, respectively.*

Remark. The bounds above differ from the previous results for sampling in the following. (1) Taking sampling ratio as input, we provide the first bound on the number of samples with a multi-round sampling strategy, while guaranteeing both precision and recall as opposed to an error rate. (2) The number of samples is independent of the size of the dataset $\mathcal{D}$. Apart from the given thresholds and error rate, the bound is determined by knowing only the sampling ratio and the maximum length of a random walk. (3) The Chernoff bounds have been extended to rules with relation atoms across multiple tables, and the lower bound of k is independent of the number of multiple tuple pairs that satisfy each rule in Σ_s. (4) We establish the first connections between support/confidence on samples $\mathcal{D}_s$ and their counterparts on the entire dataset $\mathcal{D}$.

5 PARALLEL RULE DISCOVERY

When sampling alone does not suffice to scale with big datasets, we develop a parallel rule discovery algorithm with performance guarantees, namely, *parallel scalability*. Below we first review the parallel scalability (Section 5.1) and present a sequential discovery algorithm RMiner (Section 5.2). We then show how we handle multiple tuple variables in collective rules and how we retrieve constant patterns (Section 5.3). Finally, we provide a parallel algorithm PRMiner that is parallelly scalable relative to RMiner (Section 5.4).

We run the discovery algorithms k times on a set $\{\mathcal{D}_s^1, \ldots, \mathcal{D}_s^k\}$ of samples that are extracted from dataset $\mathcal{D}$ by following the multi-round random walk sampling strategy (Section 4); samples in $\mathcal{D}_s$ have a size bounded by sampling ratio r and guarantee the bounds on precision and recall as stated in Theorem 1, subject to error ratio ϵ. For simplicity, we still use $\mathcal{D}_s$ to denote one sample.

5.1 Parallel Scalability

We revisit the widely adopted notion of parallel scalability [36]. Assume that $\mathcal{A}$ is a sequential algorithm which, given a dataset $\mathcal{D}_s$, consequences RHS, and thresholds σ and δ for support and confidence, respectively, computes a cover Σ_c of minimal σ-frequent and δ-confident REEs on $\mathcal{D}_s$ such that precision$(\Sigma, \Sigma_s) \geq \alpha$, recall$(\Sigma, \Sigma_s) \geq \beta$, and the REEs are relevant to consequences RHS. Denote its worst running time as $t(|\mathcal{D}_s|, |\text{RHS}|, \sigma, \delta)$.

Definition 5.1: *We say that a parallel* REE *discovery algorithm $\mathcal{A}_p$ is parallelly scalable relative to $\mathcal{A}$ if its running time by using n processors can be expressed as:*

$$T(|\mathcal{D}_s|, |\text{RHS}|, \sigma, \delta,) = \widetilde{O}\left(\frac{t(|\mathcal{D}_s|, |\text{RHS}|, \sigma, \delta)}{n}\right),$$

where the notation $\widetilde{O}()$ hides $\log(n)$ *factors.*

Intuitively, parallel scalability guarantees "linear" speedup of $\mathcal{A}_p$ relative to the yardstick algorithm $\mathcal{A}$. That is, the more processors are used, the faster $\mathcal{A}_p$ is. Hence $\mathcal{A}_p$ scales well with large databases by adding processors, and makes REE discovery feasible in practice.

5.2 Sequential Algorithm

We start with a sequential rule discovery algorithm, referred to as RMiner, on the sample $\mathcal{D}_s$. As shown in Figure 2, RMiner is a *levelwise search* algorithm. Given a consequence predicate p_0 in RHS and the set $\mathcal{P}_0$ of its correlated predicates, we maintain two predicate sets for discovering new REEs $\varphi : X \rightarrow p_0$ with $X \subseteq \mathcal{P}_0$:

- $\mathcal{P}_{\text{sel}}$, the set of predicates selected to constitute X; and
- $\mathcal{P}_{\text{re}}$, the set of remaining predicates in $\mathcal{P}_0$.

Initially, $\mathcal{P}_{\text{sel}}$ is empty and $\mathcal{P}_{\text{re}}$ is $\mathcal{P}_0$ (line 4). RMiner then traverses the search space level by level by maintaining a queue Q (line 8), where at the i-th level, it discovers $\varphi : X \rightarrow p_0$ with $|X| = i$. It

[1]Z could be computed by state-of-the-art algorithms, *e.g.,* [52].
[2]We add a small constant value in the denominator to prevent it from being zero.

```
Algorithm RMiner
Input: $\mathcal{D}_s$, RHS, $\sigma$ and $\delta$.
Output: A cover $\Sigma_s$ of minimal REEs such that for each $\varphi : X \rightarrow p_0$ in $\Sigma$,
    (1) $p_0 \in$ RHS; (2) $X \subseteq \mathcal{P}_0$, where $\mathcal{P}_0$ is a set of predicates correlated to $p_0$.
1.  $\Sigma := \emptyset$;
2.  Build auxiliary structures, e.g., position list indexes (PLI) [52];
3.  for each $p_0 \in$ RHS do
4.      $\mathcal{P}_{\mathsf{sel}} := \emptyset$; $\mathcal{P}_{\mathsf{re}} := \mathcal{P}_0$;
5.      $\Sigma :=$ Expand($\mathcal{D}_s, \mathcal{P}_{\mathsf{sel}}, \mathcal{P}_{\mathsf{re}}, p_0, \delta, \sigma, \Sigma$);
6.  $\Sigma_s :=$ computeCover($\Sigma$);
7.  return $\Sigma_s$;

Procedure Expand
Input: $\mathcal{D}_s, \mathcal{P}_{\mathsf{sel}}, \mathcal{P}_{\mathsf{re}}, p_0, \delta, \sigma$ and the current set $\Sigma$ of minimal REEs.
Output: An updated set $\Sigma$ of minimal REEs.
8.  $Q :=$ an empty queue; $Q$.add($\langle \mathcal{P}_{\mathsf{sel}}, \mathcal{P}_{\mathsf{re}} \rangle$);
9.  while $Q \neq \emptyset$ do
10.     $\langle \mathcal{P}_{\mathsf{sel}}, \mathcal{P}_{\mathsf{re}} \rangle := Q$.pop(); $\varphi := \mathcal{P}_{\mathsf{sel}} \rightarrow p_0$;
11.     if $\varphi$ is minimal then
12.         $\Sigma := \Sigma \cup \{\varphi\}$;
13.         continue; // do not further expand
14.     if supp($\varphi$) $\geq \sigma$ then // Anti-monotonicity
15.         for each $p \in \mathcal{P}_{\mathsf{re}}$ do // Add predicates from $\mathcal{P}_{\mathsf{re}}$ to $\mathcal{P}_{\mathsf{sel}}$
16.             $Q$.add($\langle \mathcal{P}_{\mathsf{sel}} \cup \{p\}, \mathcal{P}_{\mathsf{re}} \setminus \{p\} \rangle$);
17. return $\Sigma$;
```

Figure 2: Algorithm RMiner

iteratively moves predicates from $\mathcal{P}_{\mathsf{re}}$ to $\mathcal{P}_{\mathsf{sel}}$ (lines 15-16) until one of the following conditions is satisfied: (1) $\mathcal{P}_{\mathsf{re}}$ is exhaustive; or (2) $\varphi : \mathcal{P}_{\mathsf{sel}} \rightarrow p_0$ is a minimal REE (lines 11-13), since in this case, adding more predicates will not make supp($\varphi, \mathcal{D}$) larger, while it increases the order of φ. If $\varphi : \mathcal{P}_{\mathsf{sel}} \rightarrow p_0$ is still not a minimal REE, we expand it; before expansion, anti-monotonicity is applied to check whether we can terminate the expansion early (line 14). Finally, the cover of discovered rules is computed and returned (line 6).

Algorithm RMiner employs two optimization strategies commonly used in rule discovery. (a) When multiple p_0's in RHS share similar correlated predicates $\mathcal{P}_0$, it processes these p_0's together (not shown). (b) It pre-computes auxiliary structures (line 2) such as PLI [52], to efficiently compute supports and confidences when verifying whether the mined REEs are above the required thresholds.

5.3 Optimization Strategies

To speed up rule discovery, we propose two optimization strategies, namely, dynamic predicate expansion and constant pattern recovery. The former allows us to discover collective rules across multiple tables efficiently, and the latter complements the sampling strategy, which inevitably drops constants from the dataset.

Dynamical predicate expansion. RMiner might examine all predicate combinations (lines 15-16), which is clearly inefficient. To reduce the enumeration cost, we propose to employ ML models in rule expansion so that we only focus rule discovery among semantically correlated predicates. In the following, we first train an ML model for capturing predicate correlation, and then show how we embed the proposed model in rule discovery.

Learning predicate correlation. Recall that we maintain a set $\mathcal{P}_{\mathsf{sel}}$ of selected predicates, and we iteratively add predicate p to $\mathcal{P}_{\mathsf{sel}}$ and check whether $\mathcal{P}_{\mathsf{sel}} \cup \{p\} \rightarrow p_0$ is a minimal REE. Instead of trying all possible p, we add p to $\mathcal{P}_{\mathsf{sel}}$ only if p and $\mathcal{P}_{\mathsf{sel}}$ are *correlated.* This correlation, denoted by $\mathcal{M}_{\mathsf{Corr}}(\mathcal{P}_{\mathsf{sel}}, p)$, is learned via *reinforcement learning (RL),* and is trained based on the support supp($\mathcal{P}_{\mathsf{sel}} \cup \{p\}$).

Ideally, if supp($\mathcal{P}_{\mathsf{sel}} \cup \{p\}$) $\geq \sigma$, $\mathcal{M}_{\mathsf{Corr}}(\mathcal{P}_{\mathsf{sel}}, p)$ returns true; otherwise if supp($\mathcal{P}_{\mathsf{sel}} \cup \{p\}$) $< \sigma$, $\mathcal{M}_{\mathsf{Corr}}(\mathcal{P}_{\mathsf{sel}}, p)$ returns false and we do not add p to $\mathcal{P}_{\mathsf{sel}}$. Here we use the support supp($\mathcal{P}_{\mathsf{sel}} \cup \{p\}$) as the correlation indicator since if supp($\mathcal{P}_{\mathsf{sel}} \cup \{p\}$) $< \sigma$, $\mathcal{P}_{\mathsf{sel}} \cup \{p\} \rightarrow p_0$ and all rules expanded from it cannot be σ-frequent and thus, it is useless to try p in the predicate enumeration.

One way to obtain $\mathcal{M}_{\mathsf{Corr}}$ is to adopt a regression model, *e.g., feedforward neural network (FNN),* to predict the support of $\mathcal{P}_{\mathsf{sel}} \cup \{p\}$. However, it is nontrivial to train $\mathcal{M}_{\mathsf{Corr}}$ in this way due to the exponential number of predicate combinations; it is also impractical to compute a large number of support to train $\mathcal{M}_{\mathsf{Corr}}$ well.

In light of this, we adopt Deep Q-learning (DQN) [47] to learn $\mathcal{M}_{\mathsf{Corr}}(\mathcal{P}_{\mathsf{sel}}, p)$; we treat the currently selected $\mathcal{P}_{\mathsf{sel}}$ as state s, the next predicate p to be added as action a, and supp($\mathcal{P}_{\mathsf{sel}} \cup \{p\}$) $- \sigma$ as reward r in DQN. Intuitively, if $\mathcal{P}_{\mathsf{sel}}$ and p are correlated, the reward is positive, and a larger reward is more desirable. Otherwise, it is negative. Initially, the first state, denoted by s_1, is an empty $\mathcal{P}_{\mathsf{sel}}$. At the i-th state s_i, DQN determines the next action p to be applied, adds p to $\mathcal{P}_{\mathsf{sel}}$, and transforms s_i to s_{i+1}, denoted by $s_i \rightarrow_p s_{i+1}$.

To determine the next action (e.g., from s_i to s_{i+1}), DQN utilizes two networks: a Q-network and a target network. (1) The Q-network is implemented as a feedforward network with two hidden layers. It takes a state s_i and an action a as inputs, and outputs an estimated value $\hat{Q}$, as the estimated reward of taking action a. It is learned and updated in each state, *e.g.,* from s_i to s_{i+1}. The larger $\hat{Q}$ is, the more likely a is applied. (2) For the target network, its parameters are only updated by the parameters of Q-network in the last state, *e.g.,* s_i. Specifically, it is obtained by cloning Q-network in every state to generate the Q-learning targets for the next update. The Q-network gradually learns its parameters with increasing size of $\mathcal{P}_{\mathsf{sel}}$. Given a $\mathcal{P}^1_{\mathsf{sel}}$, it expands it with a certain number ΔL of predicates, one at a time. After that, we obtain a sequence s_{seq} of actions and states, say $\mathcal{P}^1_{\mathsf{sel}}, p_1, \mathcal{P}^2_{\mathsf{sel}}, p_2, \ldots, \mathcal{P}^{\Delta L-1}_{\mathsf{sel}}, p_{\Delta L-1}, \mathcal{P}^{\Delta L}_{\mathsf{sel}}$, where $\mathcal{P}^i_{\mathsf{sel}}$ is a set of predicates of size i; then the value Q^* is:

$$Q^*(s_{\mathsf{seq}}, p) = \mathbb{E}_{s'_{\mathsf{seq}} \sim \xi}[r + \gamma \max_{p'} Q^*(s'_{\mathsf{seq}}, p') | s_{\mathsf{seq}}, p].$$

where ξ is the environment [47], *i.e.,* the rule discovery function, and γ is a discount ratio. In DQN, the approximate value $\hat{Q}$ is learned along with the output of the target network, such that $\hat{Q} \approx Q^*$. We transform $\mathcal{P}_{\mathsf{sel}}$ into a $|\mathcal{P}_0|$-dimensional bit vector $\mathbf{v}_{\mathsf{ps}}$ since the Q-network takes $\mathcal{P}_{\mathsf{sel}}$ and p as inputs, and outputs a $|\mathcal{P}_0|$-dimensional dense vector, where $\mathbf{v}_{\mathsf{ps}}[p] = 1$ if $p \in \mathcal{P}_{\mathsf{sel}}$, and $\mathbf{v}_{\mathsf{ps}}[p] = 0$ otherwise. Then Corr($\mathcal{P}_{\mathsf{sel}}, p$) is computed as

$$\mathsf{Corr}(\mathcal{P}_{\mathsf{sel}}, p) = \mathsf{Sigmoid}(\mathsf{Qnetwork}(\mathbf{v}_{\mathsf{ps}})[p])$$

The learning method is the same as DQN, and its loss function is:

$$\mathcal{L}(\theta_i) = \mathbb{E}_{s_{\mathsf{seq}}, a \sim \rho(\cdot); s'_{\mathsf{seq}} \sim \xi}[(r + \gamma \max_{a'} \hat{Q}(s'_{\mathsf{seq}}, a'; \theta_{i-1})) - \hat{Q}(s_{\mathsf{seq}}, a; \theta_i)]$$

where $\rho(s_{\mathsf{seq}}, a)$ is the behavior distribution [47], and θ_i denotes parameters of Q-network in the i-th step. Different from DQN, which simply chooses actions from a fixed action set, we select actions, *i.e.,* predicates that do not belong to the current state($\mathcal{P}_{\mathsf{sel}}$). Hence for a given $\mathcal{P}_{\mathsf{sel}}$, when obtaining its corresponding sequence of actions and states, the action set is constantly shrinking.

```
Algorithm LearnRL
Input: $\mathcal{D}_s, \mathcal{P}_0, \sigma, \delta, N$.
Output: $\mathcal{M}_{\text{Corr}}$.
1. Initialize the DQN model $\mathcal{M}$;
2. num := 1;
3. while num $\leq N$ do
4.     Sample $p_0 \in \mathcal{P}_0$;
5.     Train $\mathcal{M}_{\text{Corr}}$ by randomly selecting one predicate from $\mathcal{P}_0$ as $\mathcal{P}_{\text{sel}}$
         at a time, and expanding $\mathcal{P}_{\text{sel}}$ until either supp($\mathcal{P}_{\text{sel}}, \mathcal{D}_s$) $< \sigma$ or
         conf($\mathcal{P}_{\text{sel}} \to p_0, \mathcal{D}_s$) $> \delta$ (controlled by DQN);
6.     Store their supports and confidence in Mem;
7.     num := num + 1;
8. return $\mathcal{M}_{\text{Corr}}$;
```

Figure 3: Algorithm LearnRL

Model training. As shown in Figure 3, we train model $\mathcal{M}_{\text{Corr}}$ on samples $\mathcal{D}_s$, denoted by LearnRL. It takes as input thresholds σ, δ for support and confidence, a set $\mathcal{P}_0$ of predicates, and a number N of sequences for training $\mathcal{M}_{\text{Corr}}$. Here P_0 collects predicates from $\mathcal{D}$; we use N to strike a balance between the accuracy of $\mathcal{M}_{\text{Corr}}$ and the training time (by default, $N = 60$; see Section 6).

LearnRL first initializes a DQN model $\mathcal{M}_{\text{Corr}}$ that consists of one Q network and target network (line 1). It then trains $\mathcal{M}_{\text{Corr}}$ with N sequences (lines 3–7). For each sequence, we sample a predicate p_0 from $\mathcal{P}_0$ (line 3), and iteratively expand $\mathcal{P}_{\text{sel}}$ (line 5) with a single predicate from $\mathcal{P}_0$ each time, until either the support of $\mathcal{P}_{\text{sel}}$ is below δ or the confidence of $\mathcal{P}_{\text{sel}} \to p_0$ reaches δ. We treat the support and confidence as rewards and feed them to $\mathcal{M}_{\text{Corr}}$ for further training.

After $\mathcal{M}_{\text{Corr}}$ is learned, we make use of it to select predicates, by replacing lines 14-16 of algorithm RMiner with the following:

```
14. if supp($\varphi$) $\geq \sigma$ then
15.     for each $p \in \mathcal{P}_{\text{re}}$ do // Add predicates from $\mathcal{P}_{\text{re}}$ to $\mathcal{P}_{\text{sel}}$
16.         if $\mathcal{M}_{\text{Corr}}(\mathcal{P}_{\text{sel}}, p)$ = True then
17.             $Q$.add($\langle \mathcal{P}_{\text{sel}} \cup \{p\}, \mathcal{P}_{\text{re}} \setminus \{p\} \rangle$);
```

That is, we use $\mathcal{M}_{\text{Corr}}$ to prune irrelevant predicates.

Example 6: *For Tables 1 and 2, the set P_0 collects predicates such as t_0.org_name $= t_1$.org_name and t_0.city $= t_1$.city. After $\mathcal{M}_{\text{Corr}}$ is trained, we use it to prune meaningless $\mathcal{P}_{\text{sel}}$. For instance, $\mathcal{M}_{\text{Corr}}(\mathcal{P}_{\text{sel}}, p)$ = False for $\mathcal{P}_{\text{sel}} = \{t_0$.zipcode $= t_1$.zipcode$\}$ and $p : t_0$.oid $= t_1$.oid; hence $\mathcal{P}_{\text{sel}}$ should not be expanded with p.*

We adopt $\mathcal{M}_{\text{Corr}}$ to prune meaningless predicate combinations during discovery. As will be seen in Section 6, $\mathcal{M}_{\text{Corr}}$ substantially reduces the search space and accelerates the discovery process by using DQN. This allows us to discover collective rules that are defined with multiple tuple variables and across different tables.

Remark. We remark the following about predicate correlation. (1) Its training procedure does not dominate the discovery cost. The computation of predicate correlation mainly includes reward computation and $\mathcal{M}_{\text{Corr}}$ training. The former is the same as calculating the supports of $\mathcal{P}_{\text{sel}}$; its results are saved and reused in the discovery step (see Section 5.4). The only extra overhead is the training cost. In practice, the training cost is quite small, as will seen shortly and be verified in Section 6. (2) Predicate correlation training depends on datasets; model $\mathcal{M}_{\text{Corr}}$ in the i-th round might not be accurate for samples of, *e.g.*, the j-th one for $i \neq j$. Thus, given the small training overhead and strong correlation of $\mathcal{M}_{\text{Corr}}$ with data distribution, we train an $\mathcal{M}_{\text{Corr}}$ in each round, *i.e.*, we separately learn $\mathcal{M}_{\text{Corr}}$ pertaining to a sample in each round.

```
Algorithm CRecover
Input: A template pattern $\varphi_{\text{tem}}$ and the relational instances $\mathcal{D}$.
Output: A set $\Sigma$ of REEs recovered from $\varphi_{\text{tem}}$.
1. Recover non-constant predicates $\mathcal{P}_v$ and consequence $p_0$ from $\varphi_{\text{tem}}$;
2. $\mathcal{P}_c = \emptyset$;
3. for each $p \in \mathcal{P}_v$ do
4.     for each $\langle t, s \rangle$ s.t. $h\langle t, s \rangle \models p$ do
5.         Fill the wildcards in $\varphi_{\text{tem}}$ with the constant values of $t$ and $s$;
6.         Add the recovered constant predicates to $\mathcal{P}_c$;
7. Resume rule discovery and obtain $\Sigma = \{\varphi : X \to p_0 \mid X \subseteq \mathcal{P}_v \cup \mathcal{P}_c\}$;
8. return $\Sigma$;
```

Figure 4: Algorithm CRecover

Constant pattern. As remarked earlier, sampling inevitably drops constants from $\mathcal{D}$. If we only mine REEs on $\mathcal{D}_s$, we might miss some REEs that hold on $\mathcal{D}$ but not on $\mathcal{D}_s$ due to the absence of some constants from $\mathcal{D}_s$. To compensate the missing constants, we propose a constant pattern recovery strategy. Instead of directly mining constant patterns in $\mathcal{D}_s$, we mine *template patterns* in $\mathcal{D}_s$, which are constant patterns whose constant values are unassigned. Then, we fill in the unassigned value in the template patterns with concrete constants, to recover constant patterns in $\mathcal{D}$.

Template pattern discovery in $\mathcal{D}_s$. We start with the notion of *template patterns*, denoted by $\varphi_{\text{tem}} : (P \to Q, t_{\text{tem}})$, which are defined analogously to a pattern format REE, except that each constant in the template pattern t_{tem} is replaced with a wildcard '_'. Intuitively, a wildcard can match an arbitrary constant. For example, for the constant pattern at the end of Section 2, we can define a template pattern $\varphi_{\text{tem}} : (P \to Q, t_{\text{tem}})$, where P, Q remain the same, and

$$t_{\text{tem}} = (\oplus_1.\text{left}, \oplus_2.\text{left}, \_, \oplus_1.\text{right}, \oplus_2.\text{right}, \bar{\_}, \neq, \mathcal{M}||\_).$$

Then instead of directly mining constant patterns in $\mathcal{D}_s$, we mine template patterns in $\mathcal{D}_s$, by revising RMiner as follows. (1) Instead of enumerating multiple constant predicates $t.A \oplus c_1, \ldots, t.A \oplus c_l$ for the same attribute $t.A$, we only identify a *template predicate* $t.A \oplus \_$. (2) We then use template predicates to check which data satisfies them. Given a template predicate $p : t.A \oplus \_$, a valuation h is said to *satisfy* p, written as $h \models p$, if $h(t).A$ exists, regardless of its value; similarly for $h \models \varphi_{\text{tem}}$. (3) Instead of checking the supports and confidences in REEs verification, we say that a template pattern φ_{tem} is *valid* if there exists at least one valuation h such that $h \models \varphi_{\text{tem}}$. Only valid template patterns will be used later for constant pattern recovery. (4) To reduce the complexity of template predicate enumeration, we pre-process $\mathcal{D}_s$ by mining free itemsets [19] from it. For each itemset mined, *e.g.*, $\{t.A = c_1, t.B = c_2\}$, we construct a corresponding *template itemset*, *e.g.*, $\{t.A = \_, t.B = \_\}$. Then, instead of enumerating all combinations of template predicates, we only enumerate template itemsets. This strategy effectively reduces the useless constant combinations that do not exist in $\mathcal{D}_s$, speeding up the template pattern discovery process.

Constant pattern recovery in $\mathcal{D}$. Given $\varphi_{\text{tem}} : (P \to Q, t_{\text{tem}})$ discovered from $\mathcal{D}_s$, we recover the constant patterns that hold on $\mathcal{D}$, by instantiating the wildcards in φ_{tem} with concrete constants.

To do this, a straightforward way is to retrieve *all* constant values in $\mathcal{D}$, fill them into φ_{tem}, and check whether the resulting rule is σ-frequent and δ-confident. Clearly, this strategy is costly since most tuple pairs in $\mathcal{D}$ may not satisfy non-constant predicates (*e.g.*,

$t.A \oplus s.B$) defined in φ_{tem}, and thus, it is unnecessary to try those constants. Besides, directly filling the wildcards with constants might not yield minimal REEs and thus, requires us to resume the discovery procedure for removing redundant predicates.

We propose a method, denoted as CRecover, to recover constant patterns based on a given template pattern φ_{tem}, as shown in Figure 4. We first recover the set of all non-constant predicates (line 1), denoted by $\mathcal{P}_v$, based on φ_{tem} (see Section 2 for how to recover predicates from pattern format REEs). Then, we find the tuple pairs $\langle t, s\rangle$ in $\mathcal{D}$ that satisfy at least one predicate in $\mathcal{P}_v$, instantiate the wildcards in φ_{tem} with the constant values of t and s, and obtain a set of constant predicates, denoted by $\mathcal{P}_c$ (lines 3-6). That is, for each $t.A \oplus \_$ in φ_{tem}, we create a constant predicate $t.A \oplus c$ in $\mathcal{P}_c$, where c is the A-attribute value of t; similarly for s. Finally, we resume the rule discovery process based on $\mathcal{P}_v$ and $\mathcal{P}_c$, *i.e.,* using RMiner, to find REEs $\varphi : X \rightarrow p_0$, where $X \subseteq \mathcal{P}_v \cup \mathcal{P}_c$ and p_0 is the RHS specified in φ_{tem} (line 7). Different from complete rule discovery, CRecover focuses discovery on the constants of φ_{tem}, without enumerating all constants and predicate combinations.

There might be a small number of missing templates from $\mathcal{D}_s$. Nonetheless, Theorem 1 ensures that our sampling method would not miss too many cases. The more rounds are used, the less are missed. As will be seen in Section 6, despite the missing cases, the recall remains high, and the discovery cost is substantially reduced.

Example 7: *Consider a template pattern* φ_{tem} : $\text{Org}(t_a) \wedge \text{Org}(t_b) \wedge \text{Pers}(t_c) \wedge \mathcal{M}_{\text{Bert}}(t_a.\text{org\_name}, t_b.\text{org\_name}) \wedge t_a.\text{oid} = t_c.\text{pid} \wedge t_c.\text{person\_address} = \_ \rightarrow t_a.\text{zipcode} = \_$ *from Tables 1 and 2. We first fetch valuations satisfying non-constant predicates in* LHS*, denoted as* (t_x, t_y, t_z)*. Then we find out valid values of attribute, e.g.,* $t_z.\text{person\_address} =$ *Guangzhou and* $t_x.\text{zipcode} =$ *510375.*

Remark. The optimization strategies do not impact the theoretical guarantee of Theorem 1. Indeed, Theorem 1 deduces the bound between Σ_s and Σ that are mined by the same discovery algorithm, regardless of what algorithm it is as long as it is based on the multi-round sampling method. Predicate correlation and constant pattern recovery are applied to the same algorithm for discovering Σ_s and Σ, just to improve its efficiency. Hence the bound of Theorem 1 remains intact under the optimization strategies.

5.4 Parallel Algorithm

We next develop a parallel algorithm, denoted by PRMiner, for rule discovery, and show that it is parallelly scalable relative to RMiner.

Setting. PRMiner runs with one coordinator S_c and n workers $P_1, \ldots, P_n$ under the Bulk Synchronous Parallel (BSP) model [66], where the coordinator is responsible for generating and distributing work units, and the workers parallelly discover rules in a levelwise manner. The overall computation is divided into supersteps, where each superstep corresponds to one level in the levelwise search.

Overview. As shown in Figure 5, algorithm PRMiner works by employing the pre-trained ML model $\mathcal{M}_{\text{Corr}}$, as follows. The coordinator S_c maintains a set Σ of REEs (line 1). It generates a set of work units and distributes them evenly to n workers (see below; lines 2-5). It also distributes $\mathcal{M}_{\text{Corr}}$ to all workers (line 6). Upon receiving the work units, each worker fetches related data from S_c and builds the corresponding auxiliary structures (lines 7-8); then all workers perform rule discovery in parallel by running RMiner locally at each worker (lines 9-11. At the end of each superstep, the coordinator S_c collects the newly discovered rules from each worker (line 14). Moreover, workloads are adjusted and balanced when needed (see below; line 12-13). Finally, the cover of the set of discovered rules is computed in parallel (line 16) and is returned (line 17).

Workload assignment. Given RHS, S_c evenly partitions it into n parts: $\text{RHS}_1, \ldots, \text{RHS}_n$. Based on RHS_j, S_c constructs a set $\mathcal{W}_j$ of work units for worker P_j. Each work unit w in $\mathcal{W}_j$ is a triple $\langle \mathcal{P}_{\text{sel}}, \mathcal{P}_{\text{re}}, p_0\rangle$, where $\mathcal{P}_{\text{sel}}$ is the set of selected predicates, $\mathcal{P}_{\text{re}}$ is the set of remaining predicates and p_0 is a consequence in RHS_j. Initially, $\mathcal{P}_{\text{sel}}$ is empty and $\mathcal{P}_{\text{re}}$ is $\mathcal{P}_0$. After receiving $\mathcal{W}_j$, each P_j fetches the corresponding data, say $\mathcal{D}_{\mathcal{W}_j}$, from $\mathcal{D}_s$, consisting of tuples and auxiliary structures that satisfy at least one predicate in $\mathcal{P}_0$ or p_0 (line 7); all workers then run RMiner on $\mathcal{D}_{\mathcal{W}_j}$ in parallel. By doing so, tuples that satisfy multiple predicates are fetched only once by each worker, reducing the communication cost.

Workload balancing. At each superstep, we split the workload $\mathcal{W}_j$ of the heaviest worker P_j and assign half of it to an idle worker P_x that has finished its work, in the following two cases.

- If there are more than one work unit in $\mathcal{W}_j$, P_j sends half of $\mathcal{W}_j$ along with the corresponding auxiliary structures to P_x.
- When only one heavy work unit $w = \langle \mathcal{P}_{\text{sel}}, \mathcal{P}_{\text{re}}, p_0\rangle$ remains in P_j, we partition its data $\mathcal{D}_{\mathcal{W}_j}$. Recall that RMiner expands $\mathcal{P}_{\text{sel}}$ with a new $p \in \mathcal{P}_{\text{re}}$ and validates rule $\mathcal{P}_{\text{sel}} \rightarrow p_0$ at each level. We reduce the validation cost since it dominates. First, we select $p' \in \mathcal{P}_{\text{sel}}$ that has the largest support, where p' is $t.A \oplus s.B$ or $\mathcal{M}(t.A, s.B)$. Then we partition attributes $t.A$ and $s.B$ into h parts, and divide the data into $\lceil \frac{|\mathcal{D}_{t.A}|}{h} \rceil \times \lceil \frac{|\mathcal{D}_{s.B}|}{h} \rceil$ partitions. We send half of these from P_j to P_x along with auxiliary structures of $\mathcal{P}_{\text{sel}} \setminus \{p'\}$ and p_0. Finally, P_x computes the supports and confidences, and sends them back to P_j; P_j integrates the results and returns valid rules at current superstep.

Learning predicate correlation in parallel. We also train the predicate correlation model $\mathcal{M}_{\text{Corr}}$ in parallel. More specifically, similar to algorithm LearnRL (Figure 3), coordinator S_c maintains N sequences, such that N initial $\mathcal{P}_{\text{sel}}$'s are randomly sampled and expanded until their supports are below σ or they make valid rules. Different from the sequential version, we distribute the time-consuming support and confidence computation evenly to all workers. Recall that each sequence contains a set $\mathcal{P}_{\text{sel}}$ and a newly added predicate p. For each valid $\mathcal{P}_{\text{sel}}$ in each sequence such that $\mathcal{P}_{\text{sel}} \rightarrow p_0$ is a candidate rule, S_c employs the current $\mathcal{M}_{\text{Corr}}$ to pick a predicate p with the maximum predicted reward. Then, S_c distributes N such sequences to n workers evenly, such that the supports and confidences of $\mathcal{P}_{\text{sel}} \wedge p \rightarrow p_0$ are computed in parallel. After all workers finish their computations, they send the actual rewards back to S_c, based on which S_c continues to trains $\mathcal{M}_{\text{Corr}}$. When N sequence expansions are finalized, the model $\mathcal{M}_{\text{Corr}}$ is learned.

Example 8: *Consider* n=3 *and* N=3*. Assume that coordinator* S_c *randomly selects* N *valid* $\mathcal{P}_{\text{sel}}$*'s, e.g.,* $\mathcal{P}_{\text{sel}_1} = \{p_1, p_2, p_6\}$, $\mathcal{P}_{\text{sel}_2} = \{p_2, p_3, p_4\}$ *and* $\mathcal{P}_{\text{sel}_3} = \{p_4, p_6, p_7\}$ *in level 3. It examines predicates* p_5, p_1*, and* p_2 *separately with the maximum predicted rewards w.r.t.* $\mathcal{P}_{\text{sel}_1}$, $\mathcal{P}_{\text{sel}_2}$*, and* $\mathcal{P}_{\text{sel}_3}$ *by current* $\mathcal{M}_{\text{Corr}}$*. It then*

Algorithm PRMiner

```
Input: $\mathcal{D}_s$, RHS, $\sigma$, $\gamma$, $\Delta L$, and pre-trained ML model $\mathcal{M}_{\text{Corr}}$.
Output: A cover $\Sigma_s$ of minimal REEs.
 /* executed at coordinator $S_c$ */
1. $i := 0$;  $\Sigma_i := \emptyset$;
2. for each $p_0 \in$ RHS do
3.     Construct a work unit $w = \langle \mathcal{P}_{\text{sel}}, \mathcal{P}_{\text{re}}, p_0 \rangle$, where $\mathcal{P}_{\text{sel}} = \emptyset$ and $\mathcal{P}_{\text{re}} = \mathcal{P}_0$;
4. Evenly divide RHS into $n$ partitions, namely RHS$_1, \ldots,$ RHS$_n$;
5. Assign workload $\mathcal{W}_j = \{w = \langle \mathcal{P}_{\text{sel}}, \mathcal{P}_{\text{re}}, p_0 \rangle \mid p_0 \in$ RHS$_j\}$ to worker $P_j$;
6. $S_c$ distributes $\langle \mathcal{M}_{\text{Corr}}$, Mem$\rangle$ to $n$ workers;
 /* Fetch data for n workers */
7. for each worker $P_j$ do
8.     Fetch $\mathcal{D}_{\mathcal{W}_j} = \{t \in \mathcal{D} \mid \exists s \in \mathcal{D}, p \in \mathcal{P}_0$ s.t. $h\langle t,s \rangle \models p$ or $h\langle t,s \rangle \models p_0$,
           where $p_0 \in$ RHS$_j\}$ and build the corresponding auxiliary structures;
 /* executed at coordinator $S_c$ in levels */
9. while there exists unfinished work do   /* superstep $i$ */
       /* run on n workers in parallel*/
10.    for each $P_j$ with non-empty workload $W_j$ do
11.        $P_j$ runs RMiner from the $i$-th to $(i+1)$-th level in parallel;
12.    for each $P_x$ that has finished the assigned workload do
13.        Balance workload between $P_j$ and $P_x$ ($P_j$ is the heaviest worker);
14.    Upon receiving new REEs from workers, $S_c$ updates $\Sigma_i$ to $\Sigma_{i+1}$;
15.    $i := i + 1$;
16. $\Sigma_s :=$ the cover of $\Sigma_i$; // computed in parallel;
17. return $\Sigma_s$;
```

Figure 5: Algorithm PRMiner

distributes $\langle \mathcal{P}_{\text{sel}_1}, p_5 \rangle$, $\langle \mathcal{P}_{\text{sel}_2}, p_1 \rangle$ and $\langle \mathcal{P}_{\text{sel}_3}, p_2 \rangle$ to three workers to compute support and confidence in parallel. The coordinator receives rewards from the workers and trains $\mathcal{M}_{\text{Corr}}$ incrementally. The process iterates until N many $\mathcal{P}_{\text{sel}}$'s have all been expanded.

Constant pattern recovery in parallel. We show PCRecover for parallel constant pattern recovery in Figure 6. We simply divide Σ_{tem} into n partitions, and distribute them to workers (line 2-4). Then all workers run algorithm CRecover in parallel and output rules containing constant predicates (line 6-7). PCRecover solves the skewness as follows. Assume that P_j has a workload much heavier than the others, including the following two cases.

- If there are more than one template patterns in P_j, we evenly divide and re-distribute them between P_j and one idle worker P_x.
- If there exists only one template pattern, we adopt the same workload balancing strategy as in PRMiner, because we can directly execute the discovery algorithm CRecover in line 7.

Parallel cover computation. Implication checking for REEs can be reduced to its counterpart for GFDs, for which a parallelly scalable algorithm is already in place [20]. Our algorithm for computing the cover of discovered REEs is developed along the same lines as [20]. The algorithm retains the parallel scalability.

Parallel scalability. The parallel scalability is shown as follows.

Theorem 2: *Algorithm* PRMiner *(resp.* PCRecover*) is parallelly scalable relative to the sequential algorithm* RMiner *(resp.* CRecover*).*

Proof. For PRMiner, observe that the worst-case time complexity of RMiner is $t(|\mathcal{D}|, |\text{RHS}|, \sigma, \delta, \alpha, \beta) = O(\sum_{\varphi \in C(\mathcal{P}_0) \times \text{RHS}} |\mathcal{D}|^{|\varphi|})$, where $C(\mathcal{P}_0)$ is the power set of $\mathcal{P}_0$ and $|\varphi|$ denotes the number of predicates in φ. In PRMiner, coordinator S_c conducts workload assignment in $O(|\text{RHS}|)$ time. The cost at each worker is dominated by the following: (a) fetch its corresponding data in time at most $O(|\mathcal{D}|)$; (b) transmit the mined rules to the coordinator in at most

Algorithm PCRecover

```
Input: A set $\Sigma_{\text{tem}}$ of template patterns and the relational instances $\mathcal{D}$.
Output: A set $\Sigma$ of REEs recovered from $\Sigma_{\text{tem}}$.
 /* executed at coordinator $S_c$ */
1. $i := 0, \Sigma_i := \emptyset$
2. for each $\varphi_{\text{tem}} \in \Sigma_{\text{tem}}$ do
3.     Construct a work unit $w = \langle \varphi_{\text{tem}}, \mathcal{S} \rangle$ ($\mathcal{S}$ is an index related with $\varphi_{\text{tem}}$);
4. Evenly divide workload and assign $\mathcal{W}_j$ to worker $P_j$;
 /* run on n workers in parallel, in supersteps */
5. while there exists unfinished work do   /* superstep $i$ */
6.     for each $P_j$ with non-empty workload $W_j$ do
7.         Run CRecover in parallel;
8.     for each $P_x$ that has finished the assigned workload do
9.         Balance workload between $P_j$ and $P_x$ ($P_j$ is the heaviest worker);
10.    Upon receiving new REEs from workers, $S_c$ updates $\Sigma_i$ to $\Sigma_{i+1}$;
11.    $i := i + 1$;
12. return $\Sigma_i$;
```

Figure 6: Algorithm PCRecover

$O(|\mathcal{D}|)$ time; (c) balance its workload, such that at most $O(|\mathcal{D}|)$ data is sent to idle workers; and (d) locally conduct discovery in $\frac{t(|\mathcal{D}|, |\text{RHS}|, \sigma, \delta, \alpha, \beta)}{n}$ time, since the workload is evenly distributed in (d). Taken together, PRMiner takes at most $\frac{t(|\mathcal{D}|, |\text{RHS}|, \sigma, \delta, \alpha, \beta)}{n}$ time, and is thus parallelly scalable relative to RMiner.

In PCRecover, coordinator S_c splits template patterns in $O(|\Sigma_{\text{tem}}|)$ time, which is much smaller than the cost of CRecover, denoted by *cost*(CRecover). Each worker (1) balances its workload with at most $O(|\mathcal{D}|)$ communication cost; and (2) performs constant recovery in $\frac{O(cost(\text{CRecover}))}{n}$ time since workload is balanced. Thus, PCRecover is parallelly scalable relative to CRecover. □

The cost of training. The learning time of $\mathcal{M}_{\text{Corr}}$ is also much smaller than $O(\frac{t(|\mathcal{D}|, |\text{RHS}|, \sigma, \delta, \alpha, \beta)}{n})$ time. Indeed, $\mathcal{M}_{\text{Corr}}$ is implemented with DQN of two hidden layers. Denote the dimension of each layer as h; then $|\mathcal{M}_{\text{Corr}}|$ is in $O(h(h + |\mathcal{P}_0|))$. In practice, h is relatively small, *e.g.*, $h \leq 10^3$ [47]. As the number of epoch (*i.e.*, iteration) to train $\mathcal{M}_{\text{Corr}}$ is a constant value, the learning time of $\mathcal{M}_{\text{Corr}}$ is bounded by $O(Nh(h + |\mathcal{P}_0|))$. Moreover, the most time-consuming computation of supports and confidences is conducted in parallel. As will be seen in Section 6, the cost is relatively small.

6 EXPERIMENTAL STUDY

We experimentally evaluated (1) the efficiency and accuracy of the proposed sampling strategy, constant pattern recovery and correlated predicate learning, (2) the scalability of PRMiner, and (3) the effectiveness of REE discovery in real-life and synthetic datasets.

Experimental setting. We start with the experimental setting.

Datasets. We used seven datasets, including six real-life datasets and a synthetic one, as summarized in Table 4. Airport, Hospital, Inspection and NCVoter are commonly used in the existing studies [43, 52]. DBLP is an academic dataset with multiple relations. Realty is a property dataset with 12 relations.

We also used a synthetic dataset Tax, modified from the tax data (1M) [7, 13], by duplicating each original tuple 10 times and modifying the attributes using a program of [18].

ML models. REEs use three ML models as predicates: ditto [42] for ER, and Bert [53] and edit distance (ED) for textual attributes. For predicate correlation model $\mathcal{M}_{\text{Corr}}$, we use 2 hidden layers and set their dimension to 50. We used Adam optimizer with a batch-size of 256; the learning rate is 0.01. We trained our model with 100 epochs on all datasets. By default, $N = 60$ unless stated otherwise.

Baselines. We implemented the following, all in Java: (1) PRMiner, including RandomWalkSampling and PCRecover; (2-5) four variants of PRMiner: $\text{PRMiner}_{\text{RS}}$ that replaces RandomWalkSampling with RandomSampling; $\text{PRMiner}_{\text{full}}$ to discover REEs directly from $\mathcal{D}$; $\text{PRMiner}_{\text{noml}}$ without using dynamic predicate expansion; and $\text{PRMiner}_{\text{noCR}}$ without constant pattern recovery; (6) DCfinder [52], a state-of-the-art algorithm that mines all bi-variable DCs that hold on the dataset; we tested DCfinder due to its superiority over other DC discovery methods, as shown in [52]; we parallelize DCfinder for a fair comparison; and (7) REEFinder, a revision of DCfinder to mine bi-variable REEs in parallel by extending the primitives in [57] and adding new primitives to support constant and ML predicates; REEFinder has almost the same discovery procedure as DCfinder, except its use of the new/extended primitives for constant and ML predicates.

We compared with the four variants of PRMiner to test the effectiveness of random walk, dynamic predicate expansion and constant tableaux, respectively, and with DCfinder for efficiency although DCfinder only mines bi-variable DCs, which are a special case of REEs. For large datasets, *e.g.,* DBLP, NCVoter and Tax, we included 4 predicates in RHS, which is typical in an application.

We conducted experiments on a cluster of up to 21 virtual machines (one for the coordinator), each powered by 32GB RAM and 2 processors with 3.10 GHz. We ran the experiments 3 times, and report the average here. We do not include the time of loading datasets and constructing auxiliary structure, *i.e.,* PLI for all algorithms. Unless stated explicitly, we set $\sigma = 10^{-4}|\mathcal{D}_s|^2$, $\delta = 0.9$, $\alpha = 0.8$, $\beta = 0.9$, and $r = 0.1$. We deduce the number k of samples and thresholds for support and confidence on samples following Theorem 1.

Experimental results. We next report our findings.

Exp-1: Sampling. The first set of experiments evaluated the usefulness of the sampling strategy, by running PRMiner and its four variants on the entire dataset and on samples $\mathcal{D}_S$. We report the accuracy (precision and recall) of our sampling strategy to verify Theorem 1. We also tested the efficiency of (1) the support threshold σ, (2) the confidence threshold δ, (3) the precision threshold α, (4) the recall threshold β, and (5) the number k of samples. The exact values of recall are computed in our experiments by also discovering the set Σ of REEs from the entire datasets in parallel.[3] For the lack of space, we only show the results on some of the datasets; the results on the other datasets are consistent.

Accuracy. Varying sampling ratio r from 0.1 to 0.4 (Figures 7(a) and 7(b)) and the number k of samples from 1 to 8 (Figures 7(c), 7(d) and 7(e)), we first tested the accuracy of the methods on Airports and Inspection. We find the following. (a) On average, PRMiner has precision 90% and recall 81% respectively, when $r = 0.1$ and $k = 3$ on Inspection and Airport. This experimentally verifies Theorem 1. (b) Constant pattern recovery improves the recall by 3% on Airport, as indicated by PRMiner vs. $\text{PRMiner}_{\text{noCR}}$. This

[3]When the datasets are very large, *e.g.,* having billions of tuples, recall could be estimated by following existing methods, *e.g.,* Monte Carlo simulation of [67].

Name	Type	#tuples	#attributes	#relations
Airport [43, 52]	real-life	55,113	18	1
Hospital [7, 13, 43, 52]	real-life	114,919	15	1
Inspection [43, 52, 54]	real-life	170,000	19	1
NCVoter [43, 52]	real-life	1,681,617	12	1
DBLP [63]	real-life	1,799,559	18	3
Realty	real-life	642,257	110	12
Tax [7, 13, 18, 43, 52]	synthetic	10,000,000	15	1

Table 4: Dataset statistic

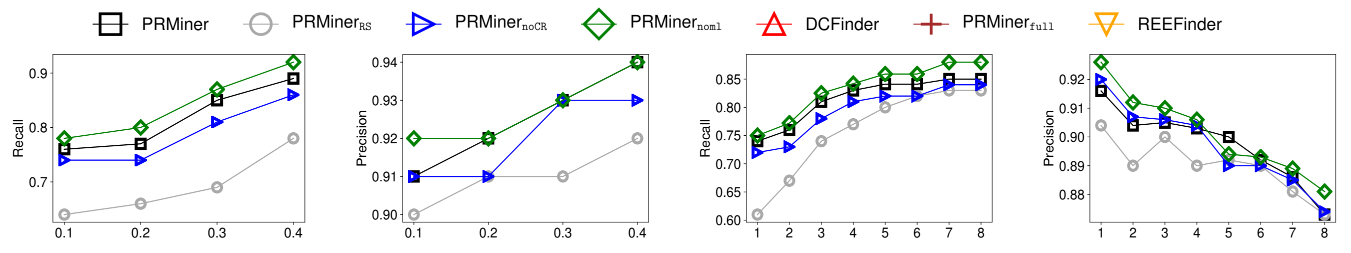
justifies the need for PCRecover. (c) The use of $\mathcal{M}_{\text{Corr}}$ speeds up rule discovery (see below) without much reduction in accuracy. The precision and recall of PRMiner are only 1% and 2% lower than those of $\text{PRMiner}_{\text{noml}}$ when $r = 0.2$ and $k = 2$, respectively. (d) On average PRMiner outperforms $\text{PRMiner}_{\text{RS}}$ in precision and recall by 2.3% and 13%, respectively; this justifies the effectiveness of RandomWalkSampling. (e) When r increases, both precision and recall improve, as expected, since $\mathcal{D}_s$ retains more data of $\mathcal{D}$. (f) The recall also improves with larger k for the same reasons. However, the precision might fluctuate slightly with larger k (note that the y-axis of Figure 7(d) has increment 0.05). This is because there are more rules pertaining to $\mathcal{D}_s$ but not to $\mathcal{D}$. This said, the overall accuracy (precision and accuracy) is improved when k gets larger.

Varying k. We varied k from 1 to 8. As shown in Figure 7(e) over multi-table Realty, recall of PRMiner increases when k gets larger, *e.g.,* 0.913 when $k = 7$. As shown in Figure 7(f) over synthetic data, (a) as k increases, the discovery time of PRMiner also increases linearly, as expected. (b) PRMiner with the multi-round sampling is still faster than the discovery in the entire dataset, *e.g.,* when k is as large as 8. This indicates that multi-round sampling supports larger value of k and is able to achieve both effectiveness and efficiency.

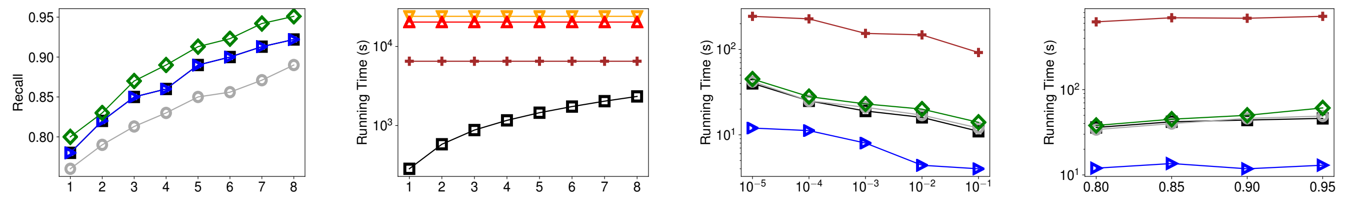
Varying σ. We then evaluated PRMiner by varying the support threshold σ from $10^{-5}|\mathcal{D}_s|^2$ to $10^{-1}|\mathcal{D}_s|^2$. As shown in Figures 7(g) on Hospital, (a) it takes PRMiner much less time to run on the sample $\mathcal{D}_s$ than on the entire dataset $\mathcal{D}$, *e.g.,* it takes PRMiner 25s on $\mathcal{D}_s$, as opposed to 228s by $\text{PRMiner}_{\text{full}}$ on $\mathcal{D}$ ($\sigma = 10^{-4}|\mathcal{D}_s|^2$). In general, PRMiner is much faster than $\text{PRMiner}_{\text{full}}$, *e.g.,* 9.3 times faster when $\sigma = 0.01|\mathcal{D}_s|^2$. This verifies sampling effectively speeds up the discovery process. (b) When σ is smaller, it takes PRMiner longer since it needs to examine more candidates, *e.g.,* PRMiner is 4.1 times faster when σ varies from $10^{-5}|\mathcal{D}_s|$ to $10^{-1}|\mathcal{D}_s|$. (c) $\text{PRMiner}_{\text{noCR}}$ is faster than PRMiner since it does not recover constant patterns, with the price of lower accuracy as shown above. $\text{PRMiner}_{\text{noml}}$ is slower than PRMiner since it does not use $\mathcal{M}_{\text{Corr}}$ to prune. The effectiveness of $\mathcal{M}_{\text{Corr}}$ is more evident on larger datasets (see Exp-2).

Varying δ. Varying confidence δ from 0.8 to 0.95, we report the results in Figure 7(h) on Inspection. As shown there, PRMiner becomes slightly faster given a smaller δ, *e.g.,* PRMiner is 1.28 times faster when $\delta = 0.8$ than when $\delta = 0.95$. This is because with larger δ, more minimal REEs have to be checked; hence longer time. Also from Figure 7(h), sampling substantially reduces the runtime.

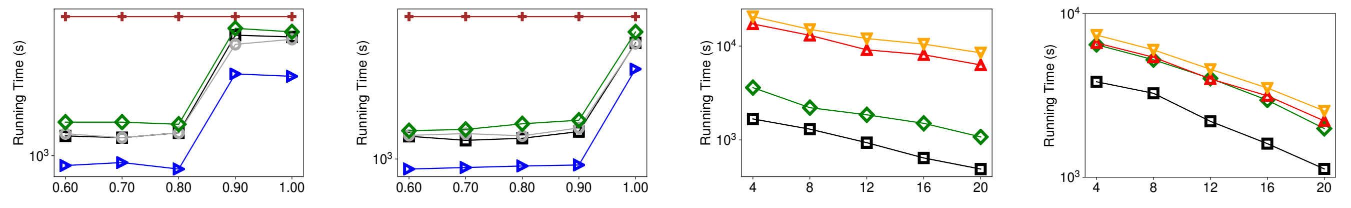
Varying α. We varied the precision threshold α from 0.60 to 1.0. As shown in Figures 7(i) on NCVoter, (a) to get higher precision, we have to sample more data from $\mathcal{D}$, and thus take longer execution time. (b) Our sampling method is effective in typical settings of α, *e.g.,* when $\alpha = 0.7$, we reduce the data size to $|\mathcal{D}_s| = 0.1|\mathcal{D}|$ with



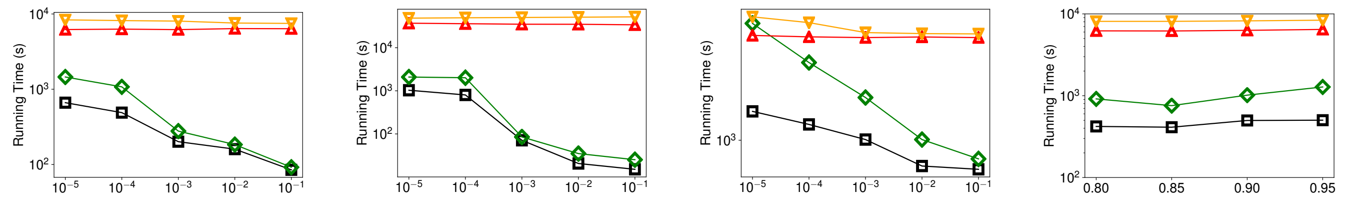
(a) Airport: Varying r (accuracy) (b) Inspection: Varying r (accuracy) (c) Airport: Varying k (accuracy) (d) Inspection: Varying k (accuracy)



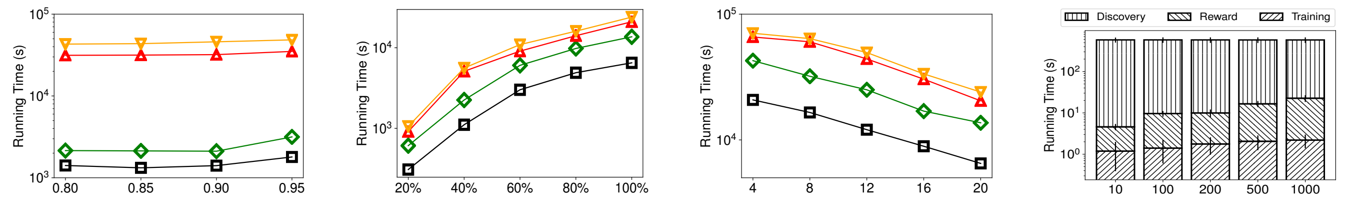
(e) Realty: Varying k (accuracy) (f) Synthetic: Varying k (time) (g) Hospital: Varying σ (time) (h) Inspection: Varying δ (time)



(i) NCVoter: Varying α (time) (j) NCVoter: Varying β (time) (k) NCVoter: Varying n (time) (l) DBLP: Varying n (time)



(m) (NCVoter): Varying σ (time) (n) Realty: Varying σ (time) (o) DBLP: Varying σ (time) (p) NCVoter: Varying δ (time)



(q) Realty: Varying δ (time) (r) Synthetic: Varying $\mathcal{D}_s$ (time) (s) Synthetic: Varying n (time) (t) Inspection: Varying N

Figure 7: Performance evaluation

only 2 round, achieving 1.89 speedup. The jump from 0.8 to 0.9 is because more rounds are needed to retain the recall by Theorem 1.

Varying β. We varied the recall threshold β from 0.6 to 1.0. The results in Figure 7(j) on NCVoter are consistent with the counterpart for α. A higher recall comes with the price of a longer execution time. Our sampling method shows convincing reduction on the sample size and the execution time, *e.g.*, when $\beta = 0.6$, it takes PRMiner 942s on $\mathcal{D}_s$, as opposed to 1875s by PRMiner$_{\text{full}}$ on $\mathcal{D}$.

Exp-2: Scalability We next studied the scalability of PRMiner on (large) entire dataset $\mathcal{D}$, by varying (1) the number n of machines, (2) the support threshold σ, and (3) the size of synthetic data. Unless stated explicitly, we set $n = 20$, $\sigma = 10^{-4}|\mathcal{D}|$, $\delta = 0.9$ by default. The results with varying α and β are consistent with their counterparts reported in Exp-1 and are hence not shown.

Varying n. We first varied the number n of machines from 4 to 20. As shown in Figures 7(k) and 7(l) on NCVoter and DBLP, respectively, (a) PRMiner scales well with the increase of machines: it is 3.42 times faster when n varies from 4 to 20. (b) PRMiner is feasible in practice. It takes 489s on NCVoter when $n = 20$, as opposed to 6292s by DCfinder and 8470s by REEFinder. This is because PRMiner utilizes σ for early termination, while the other two compute the evidence set as one part of the discovery methods, which is independent of σ and δ, and dominates their rule-discovery costs. (c) REEFinder is slower than DCfinder since it mines bi-variable REEs with constants and ML predicates, beyond bi-variable DCs targeted by DCfinder. (d) PRMiner is 1.91 times faster than $\text{PRMiner}_{\text{noml}}$ on average, up to 2.09 times. This verifies that our dynamic predicate expansion effectively reduces the execution time.

Varying σ. Varying the support threshold σ from $10^{-5}|\mathcal{D}|^2$ to $10^{-1}|\mathcal{D}|^2$, we report the results in Figures 7(m), 7(o) and 7(n) on NCVoter, DBLP and Realty, respectively. PRMiner and $\text{PRMiner}_{\text{noml}}$ take longer when σ is smaller, as expected, which is consistent with the results in Figure 7(g); DCfinder and REEFinder are not very sensitive to σ as explained above. This said, PRMiner is faster than $\text{PRMiner}_{\text{noml}}$, REEFinder and DCfinder under all values of σ, *e.g.,* PRMiner is 2.19, 12.59 and 9.36 times faster than them on NCVoter on average, respectively, when $\sigma = 10^{-5}|\mathcal{D}|^2$. It also performs the best over Realty, *e.g.,* 35.89 and 50.44 times faster than DCfinder and REEFinder, respectively; this verifies that our discovery algorithm is robust on datasets with multiple relational tables. In particular, the runtime of PRMiner increases more slightly than $\text{PRMiner}_{\text{noml}}$ when σ decreases, since it checks much less REEs than $\text{PRMiner}_{\text{noml}}$ due to its optimization strategies.

Varying δ. We varied the confidence threshold δ from 0.8 to 0.95. As shown in Figures 7(p) and 7(q), PRMiner is slightly faster in most cases when δ decreases, while the others are not very sensitive. Nonetheless, PRMiner consistently beats all its competitors, *e.g.,* it is on average 1.59 and 22.09 (resp. 2.15 and 13.81) times faster than $\text{PRMiner}_{\text{noml}}$ and DCfinder on Realty (resp. NCVoter), respectively. Again this verifies that PRMiner performs well over multiple tables.

Using large Tax synthetic data (15 attributes and 10M tuples), we tested the impact of the size $|\mathcal{D}|$ and the number n of machines.

Varying $|\mathcal{D}|$ (synthetic). Varying the scaling factor of $\mathcal{D}$ from 20% to 100%, *i.e.,* we changed the tuples per relation from 2 million to 10 million, we computed the corresponding sample $\mathcal{D}_s$ on each $\mathcal{D}$. As shown in Figure 7(r), all algorithms take longer, as expected. However, PRMiner still outperforms $\text{PRMiner}_{\text{noml}}$, REEFinder and DCfinder. It takes 308s when $\mathcal{D}$ has 2M tuples, as opposed to 610s, 1059s and 912s by the other three, respectively.

Varying n (synthetic). Fixing the data size $|\mathcal{D}|$ as 10M, we varied the number n of machines from 4 to 20. The results are reported in Figure 7(s) and are consistent with those in Figures 7(k) and 7(l). PRMiner is 3.38 times faster when n varies from 4 to 20.

Learning cost. Varying N from 10 to 1000, we report the learning cost in Figure 7(t). The cost consists of two parts, *i.e.,* the cost of computing rewards of N sequences in $\mathcal{D}_s$, and the cost of training $\mathcal{M}_{\text{Corr}}$. As shown there, (1) the learning cost increases as N increases as expected, while the overall time is relatively stable. (2) The training is very fast, e.g., 2.19s when $N = 1000$. (3) The learning cost is much smaller than the discovery time.

Exp-3: Effectiveness. We manually examined REEs discovered by PRMiner from samples of DBLP and Airport. Below are examples.

(1) ψ_1: $\text{Paper}_{\text{DBLP}}(t_0) \land \text{Paper}_{\text{DBLP}}(t_1) \land \mathcal{M}_{\text{ED}}(t_0.\text{title}, t_1.\text{title}) \land \mathcal{M}_{\text{Bert}}(t_0.\text{venue}, t_1.\text{venue}) \land t_0.\text{year} = t_1.\text{year} \rightarrow t_0.\text{id} = t_1.\text{id}$. This REE is an ER rule. It says that if two papers have similar titles and similar venues, and if they were published in the same year, then the two denote the same paper. It employs ML models, *e.g.,* Bert, to check the semantic similarity of titles and venues, which are not supported by prior data quality rules such as CFDs and DCs.

(2) ψ_2: $\text{Author}_{\text{DBLP}}(t_0) \land \text{Author}_{\text{DBLP}}(t_1) \land t_0.\text{affiliation} = t_1.\text{affiliation} \land \mathcal{M}_{\text{Bert}}(t_0.\text{name}, t_1.\text{name}) \rightarrow \mathcal{M}_{\text{ditto}}(t_0.\bar{A}, t_1.\bar{A})$, where $\bar{A}$ is the set of all attributes in relation Author of DBLP. It says that for two authors, it is because their names are similar and their affiliations are the same that the ML model predicts the two to match. It makes an attempt to explain the predication of a black box ML model $\mathcal{M}_{\text{ditto}}$ in terms of logic characteristics.

(3) ψ_3: $\text{Airport}(t_0) \land \text{Airport}(t_1) \land \text{Airport}(t_2) \land t_0.\text{continent} = t_2.\text{continent} \land t_0.\text{latitude\_deg} = t_1.\text{latitude\_deg} \land t_0.\text{iso\_region} = t_2.\text{iso\_region} \land t_0.\text{latitude\_deg} = t_2.\text{latitude\_deg} \land t_0.\text{municipality} = t_1\text{municipality} \rightarrow t_1.\text{ios\_region} = t_2.\text{ios\_region}$. This rule involves three tuple variables. It indicates that if two tuples both share a few attributes with the third one, they may have the same ios_region value. In detail, for three airports t_0, t_1, t_2, if t_0 and t_2 have the same iso_region, continent and latitude_deg, and if t_0 and t_1 have the same municipality and latitude_deg, then t_1 and t_2 have the same iso_region. It shows that PRMiner is able to discover rules with multiple tuple variables from sample $\mathcal{D}_S$. No prior algorithms discover rules with more than two tuple variables.

(4) ψ_4: $\text{Airport}(t_0) \land \text{Airport}(t_1) \land t_0.\text{iso\_country} \neq t_1.\text{iso\_country} \rightarrow t_0.\text{municipality} \neq t_1.\text{municipality}$. It says that two airports have different municipalities if their countries are different. It shows that PRMiner is able to find rules that distinguish entities/attributes. Such rules are needed for, *e.g.,* catching mismatched entities.

Summary. We find the following. (1) when $r = 0.1$ and $k = 2$, PRMiner on samples is 12.2 times faster than $\text{PRMiner}_{\text{full}}$ on the entire datasets, on average over all the datasets tested. In particular, over DBLP that has 3 relations, 18 attributes and 1.8M tuples, PRMiner with sampling takes 406s, as opposed to 2960s and 3148s by $\text{PRMiner}_{\text{full}}$ and DCfinder, respectively, when $n = 20$. (2) With the sample ratio 10%, the precision and recall of PRMiner are 90% and 82% (up to 93% and 85%), respectively, which are 2% and 13% higher than random sampling. (3) PRMiner is parallelly scalable: it is 3.38 times faster when the number n of machines varies from 4 to 20. (4) The proposed optimization methods are effective. Employing dynamic predicate expansion, PRMiner is 1.9 times faster than $\text{PRMiner}_{\text{noml}}$ on average, up to 2.1 times. Constant recovery improves the recall of $\text{PRMiner}_{\text{noCR}}$ by 3%. (5) PRMiner is capable of finding useful REEs from real-life data.

7 CONCLUSION

We have studied parallel rule discovery with sampling. The novelty of this work consists of the following: (1) a multi-round sampling strategy with accuracy guarantees; (2) a rule discovery algorithm with the parallel scalability; (3) an ML-based predicate expansion strategy to efficiently discover collective rules with multiple relation atoms; and (4) a method to efficiently retrieve constant patterns. Our experimental study has verified that the method is promising.

One topic for future work is to further tighten the accuracy bounds for rule discovery with sampling. Another topic is to conduct data repairing and rule discovery simultaneously in the same process, to enrich each other and reduce the impact of noisy data.

REFERENCES

[1] 2021. Full version. *https://github.com/yyssl88/rulediscoverybysampling/blob/main/paper.pdf.*

[2] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. 2014. DFD: Efficient functional dependency discovery. In *CIKM*. 949–958.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[4] Anonymous. 2021. Discoverying top-k interesting rules.

[5] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. 1999. Consistent Query Answers in Inconsistent Databases. In *PODS*.

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

[7] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. 2017. Efficient Denial Constraint Discovery with Hydra. *PVLDB* 11, 3 (2017), 311–323.

[8] Venkatesan T. Chakaravarthy, Vinayaka Pandit, and Yogish Sabharwal. 2009. Analysis of sampling techniques for association rule mining. In *ICDT*. 276–283.

[9] B. Chandra and Shalini Bhaskar. 2011. A new approach for generating efficient sample from market basket data. *Expert Syst. Appl.* 38, 3 (2011), 1321–1325.

[10] Bin Chen, Peter J. Haas, and Peter Scheuermann. 2002. A new two-phase sampling based algorithm for discovering association rules. In *SIGKDD*.

[11] Chyouhwa Chen, Shi-Jinn Horng, and Chin-Pin Huang. 2011. Locality sensitive hashing for sampling-based algorithms in association rule mining. *Expert Syst. Appl.* 38, 10 (2011), 12388–12397.

[12] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2021. An Overview of End-to-End Entity Resolution for Big Data. *ACM Comput. Surv.* 53, 6 (2021), 127:1–127:42.

[13] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Discovering Denial Constraints. *PVLDB* (2013).

[14] Kun-Ta Chuang, Ming-Syan Chen, and Wen-Chieh Yang. 2005. Progressive Sampling for Association Rules Based on Sampling Error Estimation. In *PAKDD*.

[15] E. F. Codd. 1972. Relational Completeness of Data Base Sublanguages. *Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987* (1972).

[16] 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.

[17] Wenfei Fan, Hong Gao, Xibei Jia, Jianzhong Li, and Shuai Ma. 2011. Dynamic constraints for record matching. *VLDB J.* 20, 4 (2011), 495–520.

[18] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional functional dependencies for capturing data inconsistencies. *TODS* 33, 2 (2008), 6:1–6:48.

[19] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. 2011. Discovering conditional functional dependencies. *TKDE* 23, 5 (2011), 683–698.

[20] Wenfei Fan, Xueli Liu, and Yingjie Cao. 2018. Parallel Reasoning of Graph Functional Dependencies. In *ICDE*. 593–604.

[21] Wenfei Fan, Ping Lu, and Chao Tian. 2020. Unifying Logic Rules and Machine Learning for Entity Enhancing. *Science China Information Sciences* (2020).

[22] Wenfei Fan, Chao Tian, Yanghao Wang, and Qiang Yin. 2021. Discrepancy Detection and Incremental Detection. *PVLDB* (2021).

[23] Peter A Flach and Iztok Savnik. 1999. Database dependency discovery: A machine learning approach. *AI communications* 12, 3 (1999), 139–160.

[24] Eve Garnaud, Nicolas Hanusse, Sofian Maabout, and Noël Novelli. 2014. Parallel mining of dependencies. In *HPCS*. IEEE, 491–498.

[25] Chang Ge, Ihab F. Ilyas, and Florian Kerschbaum. 2019. Secure Multi-Party Functional Dependency Discovery. *PVLDB* 13, 2 (2019), 184–196.

[26] Lukasz Golab, Howard Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. 2008. On generating near-optimal tableaux for conditional functional dependencies. *VLDB* (2008).

[27] Han He. 2020. *HanLP: Han Language Processing*. https://github.com/hankcs/HanLP

[28] Alireza Heidari, Joshua McGrath, Ihab F. Ilyas, and Theodoros Rekatsinas. 2019. HoloDetect: Few-Shot Learning for Error Detection. In *SIGMOD*.

[29] Xuegang Hu and Haitao Yu. 2006. The Research of Sampling for Mining Frequent Itemsets. In *Rough Sets and Knowledge Technology (RSKT)*.

[30] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Comput. J.* (1999).

[31] Wontae Hwang and Dongseung Kim. 2006. Improved Association Rule Mining by Modified Trimming. In *Computer and Information Technology (CIT)*.

[32] Caiyan Jia and Ruqian Lu. 2005. Sampling Ensembles for Frequent Patterns. In *Fuzzy Systems and Knowledge Discovery (FSKD)*.

[33] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2010. Evaluation of entity resolution approaches on real-world match problems. *PVLDB* 3, 1-2 (2010), 484–493.

[34] loannis Koumarelas, Thorsten Papenbrock, and Felix Naumann. 2020. MDedup: Duplicate detection with matching dependencies. *PVLDB* 13, 5 (2020), 712–725.

[35] Sebastian Kruse and Felix Naumann. 2018. Efficient discovery of approximate dependencies. *VLDB* (2018).

[36] Clyde P Kruskal, Larry Rudolph, and Marc Snir. 1990. A complexity theory of efficient parallel algorithms. *TCS* (1990).

[37] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2019. Wander Join and XDB: Online Aggregation via Random Walks. *TODS* 44, 1 (2019), 2:1–2:41.

[38] Weibang Li, Zhanhuai Li, Qun Chen, Tao Jiang, and Hailong Liu. 2015. Discovering functional dependencies in vertically distributed big data. In *WISE*. 199–207.

[39] Weibang Li, Zhanhuai Li, Qun Chen, Tao Jiang, and Zhilei Yin. 2016. Discovering approximate functional dependencies from distributed big data. In *APWeb*. 289–301.

[40] Yanrong Li and Raj P. Gopalan. 2004. Effective Sampling for Mining Association Rules. In *Advances in Artificial Intelligence*.

[41] Yanrong Li and Raj P. Gopalan. 2005. Stratified Sampling for Association Rules Mining. In *Artificial Intelligence Applications and Innovations (IFIP)*.

[42] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep Entity Matching with Pre-Trained Language Models. *arXiv preprint arXiv:2004.00584* (2020).

[43] Ester Livshits, Alireza Heidari, Ihab F. Ilyas, and Benny Kimelfeld. 2020. Approximate Denial Constraints. *PVLDB* 13, 10 (2020), 1682–1695.

[44] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. 2000. Efficient discovery of functional dependencies and Armstrong relations. In *EDBT*. Springer, 350–364.

[45] Basel A. Mahafzah, Amer F. Al-Badarneh, and Mohammed Z. Zakaria. 2009. A new sampling technique for association rule mining. *J. Inf. Sci.* 35, 3 (2009), 358–376.

[46] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. 1994. Efficient Algorithms for Discovering Association Rules. In *Knowledge Discovery in Databases*.

[47] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nat.* (2015).

[48] 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*.

[49] Noel Novelli and Rosine Cicchetti. 2001. Fun: An efficient algorithm for mining functional and embedded dependencies. In *ICDT*. Springer, 189–203.

[50] Thorsten Papenbrock and Felix Naumann. 2016. A Hybrid Approach to Functional Dependency Discovery. In *SIGMOD*.

[51] Srinivasan Parthasarathy. 2002. Efficient Progressive Sampling for Association Rules. In *International Conference on Data Mining (ICDM)*. 354–361.

[52] Eduardo H. M. Pena, Eduardo Cunha de Almeida, and Felix Naumann. 2019. Discovery of Approximate (and Exact) Denial Constraints. *PVLDB* 13, 3 (2019), 266–278.

[53] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *CoRR* abs/1908.10084 (2019). arXiv:1908.10084

[54] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *PVLDB* (2017).

[55] Matteo Riondato and Eli Upfal. 2015. Mining Frequent Itemsets through Progressive Sampling with Rademacher Averages. In *SIGKDD*.

[56] Hemant Saxena, Lukasz Golab, and Ihab F Ilyas. 2019. Distributed discovery of functional dependencies. In *ICDE*. IEEE, 1590–1593.

[57] Hemant Saxena, Lukasz Golab, and Ihab F Ilyas. 2019. Distributed implementations of dependency discovery algorithms. *PVLDB* 12, 11 (2019), 1624–1636.

[58] Philipp Schirmer, Thorsten Papenbrock, Ioannis Koumarelas, and Felix Naumann. 2020. Efficient Discovery of Matching Dependencies. *TODS* 45, 3 (2020), 1–33.

[59] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. 2019. DynFD: Functional Dependency Discovery in Dynamic Datasets. In *EDBT*.

[60] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *ESWC*.

[61] Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed K. Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. 2017. Synthesizing Entity Matching Rules by Examples. *PVLDB* 11, 2 (2017), 189–202.

[62] Shaoxu Song and Lei Chen. 2009. Discovering matching dependencies. In *CIKM*.

[63] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. ArnetMiner: Extraction and Mining of Academic Social Networks. In *KDD'08*. 990–998.

[64] Benjamin Taskar, Ming Fai Wong, Pieter Abbeel, and Daphne Koller. 2003. Link Prediction in Relational Data. In *NIPS*. 659–666.

[65] Hannu Toivonen. 1996. Sampling Large Databases for Association Rules. In *VLDB*. Morgan Kaufmann, 134–145.

[66] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111.

[67] William Webber. 2012. Approximate Recall Confidence Intervals. *CoRR* abs/1202.2880 (2012).

[68] Catharine M. Wyss, Chris Giannella, and Edward L. Robertson. 2001. FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies

from Relation Instances - Extended Abstract. In *DaWak*.

[69] Ying Yan, Liang Jeff Chen, and Zheng Zhang. 2014. Error-bounded Sampling for Analytics on Big Sparse Data. *PVLDB* (2014).

[70] H Yao, H Hamilton, and C Butz. 2002. Fd_mine: Discovering functional dependencies in a database using equivalences. In *IEEE ICDM*. 1–15.

[71] Juntao Yu, Bernd Bohnet, and Massimo Poesio. 2020. Named Entity Recognition as Dependency Parsing. In *ACL*.

[72] Chengqi Zhang, Shichao Zhang, and Geoffrey I. Webb. 2003. Identifying Approximate Itemsets of Interest in Large Databases. *Appl. Intell.* 18, 1 (2003), 91–104.

[73] Yunjia Zhang, Zhihan Guo, and Theodoros Rekatsinas. 2020. A Statistical Perspective on Discovering Functional Dependencies in Noisy Data. In *SIGMOD*. 861–876.

[74] Yanchang Zhao, Chengqi Zhang, and Shichao Zhang. 2006. Efficient Frequent Itemsets Mining by Sampling. In *Advances in Intelligent (IT)*.