EulerFD: An Efficient Double-Cycle Approximation of Functional Dependencies

Qiongqiong Lin*, Yunfan Gu*, Jingyan Sai*, Jinfei Liu†, Kui Ren*, Li Xiong‡
Tianzhen Wang§, Yanbei Pang§, Sheng Wang§, Feifei Li§
*Zhejiang University, {linqq, yunfangu, saijingyan, kuiren}@zju.edu.cn
†Zhejiang University, ZJU-Hangzhou Global Scientific and Technological Innovation Center, jinfeiliu@zju.edu.cn
‡Emory University, lxiong@emory.edu
§Alibaba Group, {tianzhen.wtz, yanbei.pyb, sh.wang, lifeifei}@alibaba-inc.com

***Abstract*—Functional dependencies (FDs) have been extensively employed in discovering inferential relationships in databases, which provide feasible approaches for many data mining tasks, such as data obfuscation, query optimization, and schema normalization. Since the explosive growth of data leads to a rapid increase of FDs on large datasets, existing algorithms that pay more attention to the exact FD discovery cannot extract FDs efficiently. To bridge this gap, we propose an Efficient double-cycle approximation of Functional Dependency (EulerFD) discovery algorithm, which ensures both efficiency and accuracy of FD discovery. EulerFD induces FDs from invalid ones as invalidating an FD only requires comparing and verifying some pairs of tuples (that violate the dependency) while validating an FD requires examining and verifying all tuples. Considering the abundant tuple pairs in large datasets, a novel sampling strategy is employed in EulerFD to quickly extract invalid FDs by revising the sampling range according to previous sampling results. Furthermore, EulerFD evaluates the stopping criteria in a double-cycle structure as feedback for further sampling. The sampling strategy and the double-cycle structure complement each other to achieve a more efficient sampling effect. Experimental results on real-world and synthetic datasets, especially the massive datasets from DMS of Alibaba Cloud, justify the design and verify the efficiency and effectiveness of the proposed EulerFD.**

I. Introduction

Functional dependencies (FDs) are one of the most fundamental concepts in relational databases. FDs refer to relationships among attributes in a relational instance, which states that values of some attributes can uniquely determine that of another attribute. Consequently, the definition of FDs can be explained by the well-known primary key and other unique keys of a database as they uniquely identify a tuple together with all its attributes. For example, Table I illustrates a patient dataset with nine tuples and five attributes, where attribute *Age* uniquely depends on attribute *Name*, and *Blood pressure* is uniquely determined by *Gender* and *Medicine* jointly.

FDs have wide applications in the fields of data cleaning [4, 21, 30, 31, 39], data integration [13, 23, 27], and query optimization [5, 15, 17, 19, 28]. In data cleaning, the problem of computing the number of repairs in inconsistent databases has been studied with the constraints of FDs [21]. Extended FDs that are modeled by combining patterns and integrity constraints can also be used for error detection [30]. When it comes to data integration, FDs are processed as keys and foreign keys to normalize relational databases into Boyce-Codd Normal Form (BCNF), eliminating duplicate values and making data constraints explicit [27]. Moreover, FDs are of great significance to the operation of sophisticated query optimization [5, 15, 28], and a type of query optimization called predicate move-around [19] achieves a better performance on evaluating SQL queries through FDs.

Jinfei Liu is the corresponding author.

TABLE I: Patient dataset.

	Name	Age	Blood pressure	Gender	Medicine
t_1	Kelly	60	High	Female	drugA
t_2	Jack	32	Low	Male	drugC
t_3	Nancy	28	Normal	Female	drugX
t_4	Lily	49	Low	Female	drugY
t_5	Ophelia	32	Normal	Female	drugX
t_6	Anna	49	Normal	Female	drugX
t_7	Esther	32	Low	Female	drugC
t_8	Richard	41	Normal	Male	drugY
t_9	Taylor	25	Low	Gender-queer	drugC

Applications on DMS. As one of the most fundamental functions of Alibaba Could (aka Aliyun), Data Management Service (DMS) is a database research and development service which supports unified management of multiple databases including relational databases (e.g., MySQL, SQL Server, and PolarDB), data warehouse related databases (e.g., AnalyticDB and ClickHouse), and NoSQL databases (e.g., MongoDB and Redis). DMS applies FDs for data obfuscation with the following steps: 1) industry experts of Alibaba manually label *sensitive attributes* (e.g., age and gender) that may leak private information based on domain knowledge, 2) DMS identifies FDs to find the unlabeled attributes that can uniquely determine the labeled sensitive attributes, and the newfound attributes are named *underlying sensitive attributes*, 3) DMS employs data obfuscation techniques (e.g., masking, encryption, and tokenization) to protect the sensitive attributes and the underlying sensitive attributes.

Motivation. Given such extensive applications of FDs in relational databases, devising efficient algorithms to discover FDs from relations has attracted widespread attention. Most existing works [1, 11, 14, 22, 24, 36, 37] that focus on discovering exact FDs can be classified into four categories: 1) lattice traversal algorithms, 2) difference- and agree-set algorithms, 3) dependency induction algorithms, and 4) hybrid

algorithms. Lattice traversal algorithms (e.g., Tane [14], Fun [24], FD_Mine [37], and Dfd [1]) formulate the search space as power set lattices over attributes and traverse the lattices to validate the FD candidates. Difference- and agree-set algorithms (e.g., Dep-Miner [22] and FastFDs [36]) calculate the sets of attributes with the same or different values in certain tuple pairs to derive FDs. Dependency induction algorithms (e.g., Fdep [11]) view FD discovery as an induction problem, comparing all tuples pairwise to find invalid FDs and then inverting them into valid FDs. Lattice traversal algorithms have been proven to scale well with the number of tuples while dependency induction algorithms scale well with the number of attributes, and difference- and agree-set algorithms scale moderately with both the number of tuples and the number of attributes [25]. The recent HyFD [26] combines the validation techniques in lattice traversal algorithms and the induction of FD candidates in dependency induction algorithms for a hybrid discovery algorithm. Despite decades of research, efficiently discovering exact FDs from relational databases remains a formidable challenge with the prohibitively high computational complexity of $\mathcal{O}(n^2(\frac{m}{2})^2 2^m)$ [20, 25], where n is the number of tuples and m is the number of attributes. These *exact discovery algorithms* are able to find the exact FDs in a reasonable time when n and m are relatively small, but quickly become impractical with the explosive growth of data due to the exponential time complexity. Thus, efficiently discovering FDs on large datasets is still a tall order.

To address this challenge, *approximate discovery algorithms* [3, 16] via sampling have been proposed to significantly improve the efficiency of FD discovery with a little sacrifice on accuracy. For the FD-based data obfuscation deployed on DMS, efficiency has a similar priority with accuracy as DMS is expected to respond in real-time to user requests for the dependent (and determined) attributes of their input attributes. However, existing approximate algorithms yield suboptimal runtimes and incomplete results, which is manifested in lower precision and recall measured by F_1 score (see Section V). For example, the representative approximate discovery algorithm AID-FD [3] avoids repeated sampling naively and neglects the fact that sampled tuples contribute differently to the results. To ensure both efficiency and accuracy, an appropriate sampling strategy that can quickly suggest the sampling range containing more invalid FDs for FD induction is highly desired.

Contribution. To this end, we propose a double-cycle algorithm for approximate discovery named EulerFD (Efficient Double-Cycle Approximation of Functional Dependencies) for large real-world datasets.

We adopt the strategy of inducing FDs from invalid ones as invalidating an FD only requires comparing and verifying some pairs of tuples (that violate the dependency) while validating an FD requires examining and verifying all tuples. To support efficient and accurate FD induction, EulerFD consists of four modules, including preprocessing, sampling, negative cover construction, and inversion, where the last three are integrated into a double-cycle structure. The strength of EulerFD is founded on the design of the sampling module and its cooperation with other modules. Considering the large number of tuple pairs in large datasets, we present a novel adaptive sampling algorithm to quickly select tuple pairs that are likely to contain more invalid FDs. In addition to the non-repeated sampling in [3], our sampling algorithm optimizes the sampling target selection by continuously revising the sampling range according to previous sampling results that reflect different contributions of tuple pairs. After sampling, invalid FDs are stored in tree structures in the negative cover construction module and are induced into valid FDs in the inversion module. We evaluate the stopping criteria of the negative cover construction module and the inversion module to serve as feedback for further sampling, hence forming a double-cycle structure of EulerFD. By dynamically adjusting the sampling strategy with two empirical thresholds, the combination of the sampling module and the double-cycle structure significantly improves the efficiency of the proposed EulerFD.

We evaluate and compare the performance of EulerFD with state-of-the-art FD discovery algorithms, and EulerFD is proved to be of high efficiency and accuracy on real-world and synthetic benchmark datasets. We briefly summarize our contributions as follows.

- We propose an efficient double-cycle approximation of functional dependency (EulerFD) algorithm that is capable of obtaining FDs efficiently and accurately from large datasets.
- We present a sampling strategy that can evaluate contributions of sampled tuples based on previous sampling. The sampling module and the double-cycle structure complement each other to achieve more effective and efficient sampling.
- Extensive experiments on real-world and synthetics datasets as well as DMS of Alibaba Cloud demonstrate the efficiency and effectiveness of EulerFD.

Organization. The rest of this paper is organized as follows. Section II presents the related work from the perspective of both exact and approximate discovery algorithms. Section III provides the preliminaries and then formalizes the problem of approximate discovery. The EulerFD algorithm as well as the detailed implementation is shown in Section IV. We report the experimental results and findings for performance evaluation in Section V. Finally, Section VI draws a conclusion and discusses future work.

II. Related Work

In this section, we discuss related work on both exact and approximate discovery algorithms as well as variants of the classical FD definition.

A. Exact Discovery Algorithms

Exact discovery algorithms can be generally categorized into four classes based on the employed techniques: lattice traversal algorithms, difference- and agree-set algorithms, dependency induction algorithms, and hybrid algorithms.

Lattice traversal algorithms. Lattice traversal algorithms [1, 14, 24, 37] formulate the search space as one or more

power set lattices over attributes of the given dataset, on which the traversal is carried out. Huhtala et al. [14] proposed a seminal method based on partitions of attributes for determining whether a dependency holds or not in their level-wise traversal algorithm. Novelli et al. [24] defined the concept of Free Set and introduced Fun, an easy-to-understand framework for mining FDs. The work in [37] presented another approach named FD_Mine, which leverages discovered equivalences from FD candidates to reduce the search space. In addition to these level-wise algorithms, Abedjan et al. [1] implemented a depth-first random walk traversal strategy in their algorithm Dfd. Despite varying technologies to prune the search space, lattice traversal algorithms generate a great number of FD candidates when the number of attributes is large, seriously impacting the column (attribute) scalability.

Difference- and agree-set algorithms. Lopes et al. [22] proposed Dep-Miner algorithm which calculates the sets of attributes with the same value in certain tuple pairs, and then these so-called agree-sets are maximized to derive minimal FDs. Wyss et al. [36] illustrated a depth-first algorithm named FastFDs. Unlike Dep-Miner, it maximizes difference-sets, the complementary set of agree-sets, to infer FDs. These algorithms compare all tuples in pairs to generate difference- or agree-sets, yielding a quadratic complexity in the number of tuples and leading to a defect in row (tuple) scalability.

Dependency induction algorithms. Flach et al. [11] viewed the discovery of FDs as an induction problem and proposed the dependency induction algorithm Fdep. Instead of generating and verifying numerous FD candidates, Fdep compares all tuples pairwise to find all invalid FDs and then inverts them to obtain valid FDs. This induction strategy enables Fdep to handle datasets with more attributes than the previous algorithms. However, the inversion operation and the pairwise comparisons of Fdep are highly time-consuming, and Fdep shows poor row (tuple) scalability.

Hybrid algorithms. Papenbrock et al. [26] presented a sampling-based discovery strategy in their algorithm HyFD, which alternately hybridizes sampling techniques with validation techniques, to scale the task of exact discovery to much larger datasets. HyFD induces FD candidates on the sampled tuples and then validates them on the entire dataset by lattice traversal technique, which leads to inefficiency caused by excessive FD candidates.

In this paper, to deal with large number of attributes in real-world datasets, we adopt the FD induction approach due to its good column scalability. Moreover, we propose a novel sampling strategy to reduce tuple pair comparisons and a double-cycle structure with the evaluation of the stopping criterion in the inversion module to mitigate the costly inversion of dependency induction algorithms.

B. Approximate Discovery Algorithms

Most of the existing works focused on the exact discovery, and even the best algorithms are only suitable for small real-world datasets [11, 14, 26]. Approximate discovery algorithms [3, 16] were proposed, aiming to obtain approximately correct and complete FD results by multiple sampling strategies. Kivinen et al. [16] used a random sampling strategy in their approximate algorithm and controlled the quality of the approximation through accuracy and confidence parameters, but found it inefficient when the dataset consists of many attributes. Bleifuß et al. [3] proposed an approximate algorithm AID-FD, which approximately discovers FDs based on tuple sampling and inversion. While it is orders of magnitude faster than state-of-the-art exact discovery algorithms, the accuracy of AID-FD is impacted by the sampling strategy which naively avoids repeated sampling without considering the tuple pair contributions. In addition, AID-FD stops sampling when the termination criterion is reached and does not have the ability to adjust or re-sample for optimal trade-off.

In this paper, to enhance the row scalability without sacrificing accuracy, EulerFD employs a novel sampling strategy that uses the contributions of sampled tuples from previous sampling to direct the follow-up sampling for higher sampling productivity. Furthermore, the sampling algorithm is dynamically adjusted by the evaluation of the stopping criterion in the negative cover module to further boost the efficiency and effectiveness of EulerFD.

C. Other Functional Dependency Definitions and Problem Settings

Many works also studied extensions or variants of the classical functional dependency definition. Kruse et al. [18] studied approximate FDs that are violated by a certain portion of tuple pairs due to data exceptions, ambiguities, and data errors, which is different from approximate discovery algorithms in this paper. Conditional FDs, which capture the data consistency by incorporating bindings of semantically related values, were studied in [2, 4, 8, 9]. Wei et al. [34, 35] discussed embedded FDs to address the schema design problem for data with missing values. [30, 31] presented pattern FDs, a class of integrity constraints that can model fine-grained data dependencies gleaned from partial attribute values. [10] proposed techniques for reasoning about graph FDs which extend FDs from relations to graphs. SMFD [12] formulated the FD discovery problem in the secure multi-party scenario against semi-honest adversaries and applied secure multi-party protocols over distributed partitions. FD discovery on noisy data [38] can also be regarded as the discovery of approximated FDs due to the violations. Denial constraints (DCs) [6, 29], which discover the relationships among defined predicates, are more expressive and general constraints than FDs. Discovery of DCs is much more time-consuming than that of FDs. In addition, discovery of DCs requires pre-defined predicates, thereby not appropriate for FDs. In this paper, we focus on the classical FD definition and the centralized FD discovery problem.

III. Preliminaries and Problem Statement

In this section, we first review the definition of functional dependency and related concepts which will be used in our

algorithm design. Then we present a problem statement for the approximate discovery of FDs.

Preliminaries. Given a relational instance r over schema R, FD $X \rightarrow A$ expresses that values of A uniquely depend on values of X, where attribute set $X \subseteq R$ and attribute $A \in R$. FD $X \rightarrow A$ holds (or is valid) if and only if all pairs of tuples that agree on X also share their values on A. Similarly, if there exists a pair of tuples on which FD $X \rightarrow A$ does not hold, then $X \not\rightarrow A$ is a non-FD. We call X the left hand side (LHS) and A the right hand side (RHS). Formal definitions are given as follows.

Definition 1: (**Functional Dependency (FD)**). Let $X \subseteq R$ and $A \in R$, functional dependency (FD) $X \rightarrow A$ holds in r *iff* $\forall t, u \in r,\ t[X] = u[X] \Rightarrow t[A] = u[A]$, where $t[X]$ is the value(s) of tuple t on X and $t[A]$ is the value of tuple t on A.

Definition 2: (**Non-FD**). Let $X \subseteq R$ and $A \in R$, $X \not\rightarrow A$ is a non-FD in r *iff* $\exists t, u \in r,\ t[X] = u[X] \wedge t[A] \neq u[A]$.

Example 1: We show examples for the above three definitions based on the data in Table I. To facilitate the demonstration, we denote attributes of the data with initials and omit the brackets and commas in LHSs. FD $AB \rightarrow M$ holds as all tuple pairs that agree on attribute set AB, i.e., t_2 and t_7, also agree on attribute M. FD $N \rightarrow B$ is valid because no tuple pairs agree on N, and values of B uniquely depend on values of N. $G \not\rightarrow M$ is a non-FD because tuples t_2 and t_8 agree on G with the shared value "Male" but disagree on M with different values "DrugC" and "DrugY".

Definition 3: (**Specialize and Generalize**). Let $X, Y \subseteq R$ and $A \in R$, if $Y \subset X$, then $X \rightarrow A$ specializes $Y \rightarrow A$ ($X \rightarrow A$ is a special FD of $Y \rightarrow A$), and $Y \rightarrow A$ generalizes $X \rightarrow A$ ($Y \rightarrow A$ is a general FD of $X \rightarrow A$). The definition of specialize and generalize applies to both FDs and non-FDs.

Example 2: Given the data in Table I, $NG \rightarrow M$ specializes $N \rightarrow M$ while $N \rightarrow M$ generalizes $NG \rightarrow M$ because $N \subset NG$. Similarly, $ABG \rightarrow N$ cannot specialize or generalize $AGM \rightarrow N$ as $ABG \not\subset AGM$ and $AGM \not\subset ABG$.

Definition 4: (**Non-trivial and Minimal FD**). Let $X \subseteq R$ and $A \in R$, $X \rightarrow A$ is non-trivial if $A \notin X$, and is minimal if $\forall Y \subset X,\ Y \rightarrow A$ does not hold.

Example 3: $AB \rightarrow M$ is a non-trivial and minimal FD since we have $M \notin AB$ and M cannot be determined by any subsets of AB. Besides, $NG \rightarrow M$ isn't a minimal FD because there exists a valid FD ($N \rightarrow M$) whose LHS is a subset of NG. $ABM \rightarrow M$ is a trivial FD because $M \in ABM$.

Lemma 1: Let $X \subseteq R$, $Y \subset X$, and $A \in R$, if $Y \rightarrow A$ holds in r, then $X \rightarrow A$ holds in r; if $X \rightarrow A$ does not hold in r, then $Y \rightarrow A$ also does not hold.

Lemma 1 indicates the relationships between generalizations and specializations of FDs/non-FDs and is essential for FD induction. We will utilize Lemma 1 to remove redundant non-FDs in Algorithm 2 and invalidate FD candidates in Algorithm 3.

Definition 5: (**Negative Cover and Positive Cover**). For all $X \subseteq R$ and $A \in R \backslash X$, the negative cover can be represented as $\{X \not\rightarrow A \mid \ X \not\rightarrow A$ is a non-FD in $r\}$, and the positive cover is $\{X \rightarrow A \mid X \rightarrow A$ is an FD in $r\}$.

Example 4: Given the data in Table I, FD set $\{N \rightarrow M, AB \rightarrow M\}$ is a proper subset of the positive cover, and a proper subset of the negative cover is $\{M \not\rightarrow A, BG \not\rightarrow N\}$.

In other words, the collection of all non-FDs in r is called the negative cover, abbreviated as *Ncover*. The collection of all valid FDs in r is called the positive cover, abbreviated as *Pcover*. For all $X \subseteq R$ and $A \in R \backslash X$, $X \rightarrow A$ must be either in Ncover or Pcover. Given relational instance *r*, we take the set of all non-trivial and minimal FDs (i.e., the output of exact discovery algorithms on *r*) as *target Pcover* to unify the output results. Although all FDs in the target Pcover together with their specialized ones exactly make up Pcover in a broad sense, the target Pcover suffices for FD discovery because the generalizations of a non-trivial and minimal FD are non-FDs and the specializations are FDs by logical inference.

Problem Statement. Given a relation r, the problem of the approximate discovery is to efficiently find non-trivial and minimal FDs in r with high accuracy.

It is far from trivial to discover non-trivial and minimal FDs efficiently on large datasets. The strategy of FD induction with an inversion operation is proved to be advantageous for runtime reduction [25]. However, pairwise comparisons and the costly inversion in exact FD induction algorithms limit this advantage. Approximate discovery algorithms with effective sampling suitable for large real-world datasets are highly desired.

IV. Algorithm

In this section, we propose an approximate discovery algorithm EulerFD. We describe the structure of EulerFD including the four modules and their cooperation in the two cycles. We first give an overview of EulerFD and then specify each module in detail.

A. Overview

As shown in Figure 1, EulerFD consists of four modules: preprocessing, sampling, negative cover construction, and inversion. We briefly discuss each module as follows.

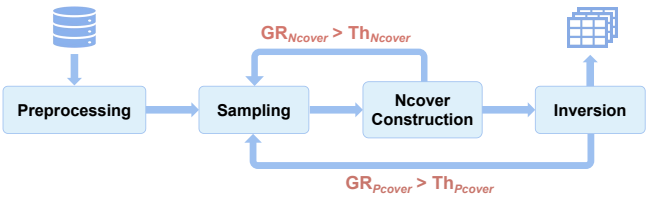


Fig. 1: Overview of EulerFD algorithm.

1) **Preprocessing Module.** To facilitate the massive tuple comparisons, we preprocess the input raw data into numerical labels that are organized in *partitions* (Definition 6), which significantly improves the efficiency as well as reduces the storage space. In addition, the partition groups tuples with shared values together, making the sampling module more

efficient by guaranteeing that each sample can obtain at least one non-FD.

2) **Sampling Module.** To overcome the defect that the dependency induction algorithm Fdep [11] cannot handle large datasets, EulerFD utilizes the tuple pairs sampled from the dataset rather than all tuples to examine and verify invalid FDs. Our sampling algorithm combines the multi-level feedback queue (MLFQ) [7] with the sliding window. The former gives suggestions on the sampling range while the latter operates sampling non-repeatedly in the suggested range with variable window sizes.

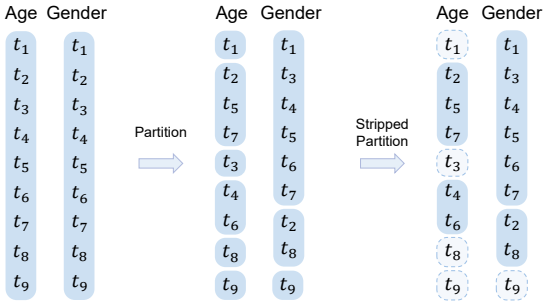
3) **Negative Cover Construction Module.** A set of non-FDs is obtained by EulerFD based on the results of sampling and converted into tree structures, Ncover, so that valid FDs can be induced accordingly later. When constructing Ncover, EulerFD records the growth rate of Ncover GR_{Ncover} as a criterion for deciding whether to continue sampling or enter the inversion module, forming the first cycle of EulerFD.

4) **Inversion Module.** The inversion module is responsible for converting Ncover into a set of valid FDs, Pcover. After inverting Ncover into Pcover, EulerFD records the growth rate of Pcover GR_{Pcover} as a criterion for deciding whether to continue sampling or terminate the entire discovery algorithm, forming the second cycle of EulerFD.

In summary, EulerFD utilizes a novel sampling strategy and a double-cycle structure complementing each other to achieve efficient and accurate FD discovery. Some general ideas we use have been presented in existing works, including partitions [14] and the strategy of inducing FDs [11] (i.e., the negative cover construction module and the inversion module). However, how to adjust and organize these modules to improve the performance of FD discovery algorithms is challenging. The idea of sampling has been widely used for approximate discovery of FDs but at the cost of accuracy. We propose a brand-new sampling strategy by combining the multilevel feedback queue (MLFQ) with the sliding window. In addition, these four modules are organized in a novel double-cycle structure in EulerFD by evaluating the growth rates of Nocver and Pcover and re-sample accordingly to improve the accuracy.

B. Preprocessing Module

Since data values in a dataset may be of various types (e.g., String, Decimal, and Character), frequent value comparisons make it challenging for FD induction algorithms to quickly obtain non-FDs from the raw data. During the initialization of EulerFD, various types of data are organized in a unified and compact format, *partitions*, making massive efficiency gains for non-FD acquisition in follow-up modules. We maintain a sequence of numerical labels for each attribute, where each label corresponds to a group of tuples with the same data value on that attribute. We then compare tuples based on their partition labels rather than the original attribute values afterwards. In other words, distinct numerical labels are assigned to different attribute value groups, because it is the equivalence of data values that matters for tuple comparisons, rather than the values themselves. The preprocessing module is based on partitions [14] and the definition is given as follows.



(a) Partition (b) Stripped partition

Fig. 2: Preprocessing of attribute Age and $Gender$.

Definition 6: (**Partition**). Let $A \in R$ and $t, u \in r$, we denote the equivalence class of t on A as $[t]_A = \{u \in r \mid t[A] = u[A]\}$ and the partition of r on A as $\Pi_A(r) = \{[t]_A \mid t \in r\}$.

Example 5: Given the data in Table I, the partition on attribute *Age* is $\Pi_{Age}(r) = \{\{t_1\}, \{t_2, t_5, t_7\}, \{t_3\}, \{t_4, t_6\}, \{t_8\}, \{t_9\}\}$ and on attribute *Gender* is $\Pi_{Gender}(r) = \{\{t_1, t_3, t_4, t_5, t_6, t_7\}, \{t_2, t_8\}, \{t_9\}\}$. For brevity, elements of the partition, i.e., equivalent classes, are also referred to as *clusters*. In Figure 2(a), EulerFD assigns label "1" to value "Female" for attribute *Gender* (the first cluster $\{t_1, t_3, t_4, t_5, t_6, t_7\}$), label "2" to value "Male" (the second cluster $\{t_2, t_8\}$), and label "3" to value "Gender-queer" (the third cluster $\{t_9\}$), and then replaces original values with these labels as shown in Table II. We note that the numerical labels of different attributes are independent and can be reused since the attribute values are compared independently given any two tuples.

TABLE II: Patient data after preprocessing.

	Name	Age	Blood pressure	Gender	Medicine
t_1	1	1	1	1	1
t_2	2	2	2	2	2
t_3	3	3	3	1	3
t_4	4	4	2	1	4
t_5	5	2	3	1	3
t_6	6	4	3	1	3
t_7	7	2	2	1	2
t_8	8	5	3	2	4
t_9	9	6	2	3	2

The partition is a fundamental data structure of EulerFD that shrinks the storage space and reduces the cost of comparisons between tuples, significantly improving the overall performance. Furthermore, the sampling on partitions avoids meaningless tuple comparisons that have no potential to generate non-FDs, because tuples from the same cluster always agree on at least one attribute which is contained in LHS of the generated non-FD. It can be observed that equivalent classes with only one tuple is inconsequential for FD discovery as it can neither validate an FD which requires checking the entire dataset, nor generate a non-FD which requires comparing two or more tuples. Taking this into consideration, the definition of *stripped partition* [14] is proposed as follows.

Definition 7: (**Stripped Partition**). Let $A \in R$, $t, u \in r$, and $[t]_A$ be the equivalence class of t on A, then the stripped partition of r on A is $\hat{\Pi}_A(r) = \{[t]_A \mid t \in r \wedge |[t]_A| > 1\}$.

Example 6: As shown in Figure 2(b), the stripped partition on attribute *Age* is $\hat{\Pi}_{Age}(r) = \{\{t_2, t_5, t_7\}, \{t_4, t_6\}\}$ and on attribute *Gender* is $\hat{\Pi}_{Gender}(r) = \{\{t_1, t_3, t_4, t_5, t_6, t_7\}, \{t_2, t_8\}\}$ because other clusters have only a single tuple.

The stripped partition compresses the partition by dismissing equivalent classes with only one tuple which do not to contribute to FD discovery. It further simplifies and unifies data to provide favorable support for EulerFD, especially for the following sampling module.

C. Sampling Module

Instead of obtaining non-FDs by comparing all tuples pairwise, which ensures that the discovered FDs are completely correct but with time complexity of $\mathcal{O}(n^2)$, we employ a sampling strategy for considerable efficiency improvement with a little sacrifice on accuracy. Simple random sampling in all clusters takes no account of the data characteristics, causing clusters that potentially contain more non-FDs to lose their deserved attention. Our sampling module captures the tuple pair contributions and adjusts the sampling strategy appropriately in line with their evaluation.

Intuition. To design a sufficient and efficient sampling algorithm on stripped partitions for approximate discovery, we should address the following two factors. (1) *Coverage*. The non-FDs derived from sampled tuple pairs are supposed to occupy the vast majority of the total non-FD set. Therefore, we perform multiple samples in all clusters until almost no more new non-FDs are generated to ensure the final FD results are as correct and complete as possible. (2) *Regularity*. A regular sampling method is necessary to guarantee that the tuple pairs from multiple samples of a cluster are not duplicated. EulerFD regards two tuples at regular intervals of a cluster as a pair and applies different intervals for multiple samples to avoid repeated sampling as much as possible.

Moreover, attributed to the regularity of our sampling method, the sampling range can be continuously revised to optimize the target selection in multiple samples and accelerate the acquisition of new non-FDs, which in turn improves the coverage. Specifically, the sampling results of a cluster reflect the contributions of tuple pairs sampled in it, based on which EulerFD revises the sampling range dynamically by giving priority to clusters that potentially contain more non-FDs. We use sampling capacity $capa$ to measure the contribution of the latest sampled tuple pairs in a cluster and define $capa$ as follows.

$$capa\ (of\ a\ cluster) = \frac{number\ of\ new\ non\text{-}FDs}{number\ of\ latest\ sampled\ tuple\ pairs}$$

The $capa$ values of clusters always change after each sample, thus EulerFD is capable of performing sampling in clusters with high $capa$ accordingly and terminating sampling appropriately for the Ncover construction module. However, some rare non-FDs that are only contained in clusters with low $capa$ cannot be left out. Our novel adoption of multilevel feedback queue (MLFQ) [7] tackles the problem by prioritizing clusters with high $capa$ while still giving deserved consideration to clusters with low $capa$.

Algorithm. For a sufficient and efficient sampling algorithm, we utilize the multilevel feedback queue (MLFQ) [7] among clusters to suggest the sampling range and the sliding window within a cluster to sample tuple pairs for non-FDs.

MLFQ is a CPU processor scheduling algorithm composed of multiple queues with different priorities and scheduling methods. Without any prior knowledge, the MLFQ scheduler learns about processes as they run, and moves processes between queues based on their observed behavior to compromise the turnaround time and response time [30, 32]. In our algorithm, clusters are approached as processes and $capa$ as their behavior. In view of the philosophy that MLFQ learns from history to predict the future, we observe the $capa$ values of clusters as they are sampled, and move clusters between queues accordingly to suggest the sampling range.

After the first sampling in all clusters, EulerFD initializes $capa$ for each cluster and assigns clusters to corresponding queues in MLFQ by their $capa$. Clusters with high $capa$ are assigned to high priority queues, which means that they are suggested as the sampling range earlier than those with low $capa$. Clusters with similar $capa$ may be assigned to the same queue and thus have the same priority. Therefore, the parameter settings, including the number of queues and the $capa$ range of each queue, are crucial for MLFQ, which will be detailedly discussed in our experiments. By sampling the cluster from the head of the highest priority queue that is not empty, EulerFD updates its $capa$ and reassigns it to the tail of a new queue, which enables continuous revision of the sampling range. The lowest priority queue runs in a round-robin fashion where EulerFD keeps sampling until no new non-FDs are generated in recent samples to ensure the coverage. Consequently, the adoption of MLFQ makes it possible to pay proper attention to both clusters with high $capa$ and low $capa$.

The sliding window is an auxiliary that slides in a cluster to sample tuple pairs at regular intervals, substantially reducing duplicate tuple pairs in multiple samples. EulerFD regards the two tuples at both ends of the window, i.e., the first and the last tuples in the window, as a pair and then compares their values of all attributes to obtain non-FDs. Then, we take the attributes where the tuple pair has same values as LHS and the attributes where the tuple pair has different values as RHSs to form non-FDs. Note that our sliding window is different from the traditional one, we only focus on the two tuples at the ends and leave the other tuples untouched. The window slides from the beginning to the end of a cluster to obtain all tuples pairs where the two tuples are at a regular interval. EulerFD maintains a window size for each cluster and changes it to distinct values in multiple samples to avoid repeated sampling.

Following with the above two conceptions, we show our complete sampling algorithm in Algorithm 1. Line 1 initializes

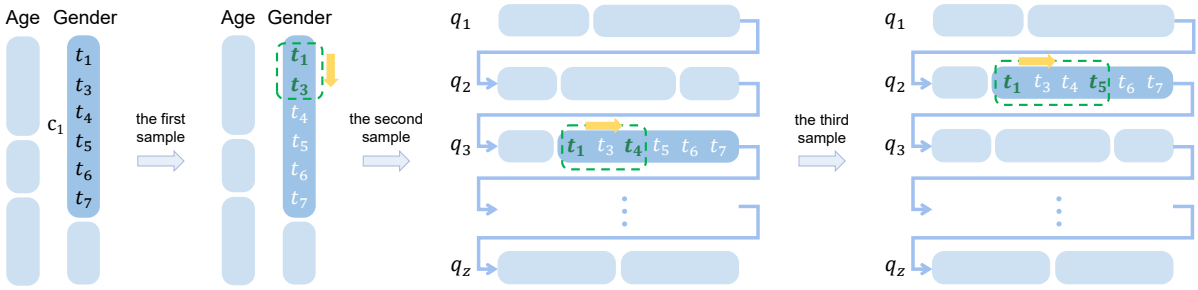


Fig. 3: Samples on cluster c_1.

an MLFQ Q and records the number of clusters in it as $currentClusterNum$. The first sampling on stripped partitions is implemented in Lines 2-4. For each cluster in $pars$, EulerFD samples it with an initial window size of 2 to obtain non-FDs and then assigns the cluster to the corresponding queue in Q by its $capa$. In Lines 5-10, we employ MLFQ to suggest the sampling range and continue sampling until there are no clusters in Q. Function $sample(cluster)$ is described in Lines 13-21. After comparing tuple pairs sampled by the sliding window, EulerFD reassigns clusters that potentially generate more non-FDs to Q by their $capa$ in the last sampling, and changes the cluster window size for further sampling.

Algorithm 1: Sampling.

```
   Input : stripped partitions pars
   Output: non-FD set nonFds
1  let the multilevel feedback queue Q = {q_1, ..., q_z} and the number
     of clusters in MLFQ currentClusterNum = 0 ;
2  for each cluster in pars do
3      cluster.window = 2 ;
4      nonFds = nonFds ∪ sample(cluster) ;
5  while currentClusterNum ≠ 0 do
6      for i = 1 to z do
7          if q_i is not empty then
8              cluster ← q_i.peek() ;
9              currentClusterNum −= 1 ;
10             nonFds = nonFds ∪ sample(cluster) ;
11 return nonFds ;
12
13 Function sample (cluster) :
14     window ← cluster.window ;
15     for i = 1 to cluster.length − window + 1 do
16         newNonFds = compare(cluster, i, i + window − 1) ;
17     if average capa > 0 in recent samples then
18         assign the cluster to the corresponding queue by capa ;
19         currentClusterNum += 1 ;
20     cluster.window += 1 ;
21     return newNonFds ;
```

We show a running example of Algorithm 1 in Figure 3. There are z queues in MLFQ, and the $capa$ range of q_2 is $[1, 10)$ and of q_3 is $[0.1, 1)$. In the first sampling of cluster c_1 $\{t_1, t_3, t_4, t_5, t_6, t_7\}$, t_1 and t_3, t_3 and t_4, t_4 and t_5, t_5 and t_6, and t_6 and t_7 are selected as tuple pairs. Taking tuple pair t_1 and t_3 as example, we compare all their attribute values and get four non-FDs $G \not\to N$, $G \not\to A$, $G \not\to B$, and $G \not\to M$. Given that four new non-FDs are obtained by comparing these tuple pairs (other non-FDs have been sampled previously), the $capa$ of c_1 is $4/5 = 0.8$ and we assign c_1 to q_3 in MLFQ. As the sampling algorithm runs, if q_1 and q_2 are empty, c_1 at the head of q_3 is suggested as the sampling range, and then tuple pairs t_1 and t_4, t_3 and t_5, t_4 and t_6, and t_5 and t_7 are compared with a window size of 3. Supposing this sampling generates five new non-FDs, the $capa$ value of c_1 is $5/4 = 1.25$ and then we reassign c_1 to q_2. When it comes to the third sampling on c_1, the tuple pairs are t_1 and t_5, t_3 and t_6, and t_4 and t_7. If no new non-FD is generated in this sampling, c_1 is assigned to q_z by $capa$ 0 and waits for continuous sampling until its average $capa$ of recent samples equals to 0.

D. Negative Cover Construction Module

Given the set of non-FDs obtained in the sampling module, EulerFD constructs Ncover that optimally stores non-FDs in a tree structure and serves as an effective pruning tool in the inversion module. The Ncover tree, on the one hand, minimizes the number of non-FDs to be stored since a non-FD is contained in its specialization (Lemma 1), and on the other hand, enables to find the specialization of a non-FD conveniently. Once Ncover is constructed, EulerFD monitors the growth rate of Ncover GR_{Ncover} and estimates whether to return to the sampling module or advance to the inversion module, forming the first cycle of EulerFD.

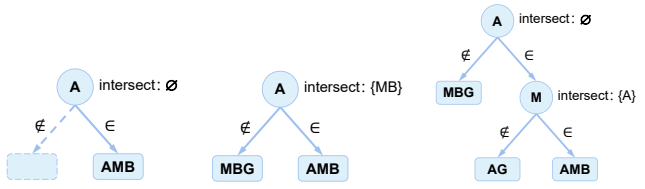
Algorithm 2: Negative cover construction.

```
   Input : non-FD set nonFds
   Output: negative cover Ncover
1  sort nonFds and their LHSs ;
2  for each nonFd in nonFds do
3      if Ncover.findSpecialization(nonFd) then
4          continue ;
5      add nonFd to Ncover ;
6  compute the growth rate of Ncover GR_Ncover ;
7  if GR_Ncover > Th_Ncover then
8      return to the sampling module ;
9  else
10     go to the inversion module ;
```

The detailed algorithm is shown in Algorithm 2. In Line 1, we sort non-FDs in decreasing order of LHS length and sort their LHSs in ascending order of attribute frequency to support specialization checks, which is explained later in the building of extended binary trees. In Lines 2-5, we construct Ncover with sorted non-FDs and minimize Ncover by discarding

general non-FDs. We compute the growth rate of Ncover as GR_{Ncover} in Line 6 and compare it with the empirical threshold Th_{Ncover} (discussed in Section V-F) in Lines 7-10. If GR_{Ncover} is greater than Th_{Ncover}, we recognize that the expansion of Ncover is promising and return to the sampling module for further sampling, otherwise, we proceed to the inversion module.

Our algorithm is based on an extended binary tree (proposed in [3]) to expedite the specialization and generalization checks that are frequent and expensive in the Ncover construction module and the inversion module. Compared with the typical FD-tree [11], the binary tree consumes less memory while quickly searching for specialization and generalizations of FDs/non-FDs. For each attribute as RHS, a binary tree is build. The binary tree stores LHSs in leaf nodes and distinguishes their paths based on whether attributes of internal nodes are included in them. Specifically, LHSs including the attribute of a internal node are stored in the right subtree of the internal node while LHSs excluding the attribute are stored in the left subtree. To optimize specialization and generalization checks of FDs/non-FDs with long LHSs, internal nodes record intersections of all their descendants, so that we can finish the unnecessary search in advance if an intersection is not included in the LHS being checked. Thus, sorting non-FDs in decreasing order of LHS length contributes to reducing modifications in binary tree building, while sorting LHSs in ascending order of attribute frequency contributes to locating FDs/non-FDs rapidly.



(a) Add $AMB \not\to N$ (b) Add $MBG \not\to N$ and $BG \not\to N$ (c) Add $AG \not\to N$

Fig. 4: Ncover construction for RHS N with $ABM \not\to N$, $BG \not\to N$, $BGM \not\to N$, and $AG \not\to N$.

We show a running example in Figure 4. Taking attribute $Name$ as RHS, the sampling module obtains four non-FDs $ABM \not\to N$, $BG \not\to N$, $BGM \not\to N$, and $AG \not\to N$, which are derived from tuple pairs t_2 and t_7, t_4 and t_7, t_5 and t_6, and t_5 and t_7, respectively. After sorting, we have $AMB \not\to N$, $MBG \not\to N$, $BG \not\to N$, and $AG \not\to N$. Non-FD $AMB \not\to N$ is added to the right child of internal node A (selected in the order of LHS attributes) in Figure 4(a), and $MBG \not\to N$ is added to the left child of internal node A in Figure 4(b) with intersection $\{MB\}$. We discard non-FD $BG \not\to N$ because it is specialized by $MBG \not\to N$. In Figure 4(c), we add non-FD $AG \not\to N$, which is not specialized by others, to Ncover and modify the binary tree by inserting a new internal node M with leaf nodes AG and AMB and intersection $\{A\}$. The intersection $\{MB\}$ recorded by internal node A is then changed to $\{\emptyset\}$ because LHSs in the descendants of A have no intersection.

The growth rate GR_{Ncover} indicates the probability of Ncover expansion in subsequent sampling. That is, the higher GR_{Ncover} is, the more potential sampling module has to continue obtaining new non-FDs. Due to the fact that the inversion module is time-consuming in EulerFD, we prefer to keep sampling and constructing Ncover until GR_{Ncover} is not greater than empirical threshold Th_{Ncover}, i.e., the current samples are sufficient to construct a relatively complete Ncover. Consequently, GR_{Ncover} is a critical evaluation of the stopping criterion that constitutes the first cycle in our approximate discovery algorithm.

E. Inversion Module

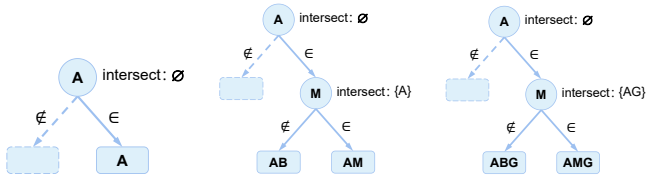
The constructed Ncover is then inverted to Pcover that consists of non-trivial and minimal FDs in the inversion module. Specifically, EulerFD employs Ncover as a pruning tool and removes the invalid FDs, which are derived from the generalizations of non-FDs in Ncover (Lemma 1), from FD candidates in Pcover to yield the final FD results. Based on the binary tree as well, Pcover supports the extensive generalization checks and creates minimal FD candidates from invalid FDs. Similarly, the growth rate of Pcover GR_{Pcover} is monitored by EulerFD to estimate whether to return to the sampling module or terminate the entire algorithm, forming the second cycle of EulerFD to improve the result integrity.

Algorithm 3: Inversion.

```
   Input : negative cover Ncover and attribute set R
   Output: positive cover Pcover
1  for each rhs in R do
2      add {∅} → rhs to Pcover ;
3  for each nonFd in Ncover do
4      invert(nonFd) ;
5  compute the growth rate of Pcover GR_Pcover ;
6  if GR_Pcover > Th_Pcover then
7      return to the sampling module ;
8  else
9      return Pcover ;
10
11 Function invert (nonFd) :
12     while Pcover.findGeneralization(nonFd) do
13         remove the generalization general from Pcover ;
14         for each attr in R do
15             if attr ∈ general.lhs ∪ general.rhs then
16                 continue ;
17             create candidate as
                 {general.lhs ∪ attr} → general.rhs ;
18             if Pcover.findGeneralization(candidate) then
19                 continue ;
20             add candidate to Pcover ;
```

The detailed algorithm for inversion is shown in Algorithm 3. In Lines 1-2, we initialize $Pcover$ and add the most general FD candidate $\emptyset \to A$ of each attribute A (as RHS) to it. In Lines 3-4, we invert $Ncover$ to $Pcover$ by pruning and creating FD candidates with each $nonFd$. Lines 5-9 compute the growth rate of $Pcover$ as GR_{Pcover} and compare it with the empirical threshold Th_{Pcover} (discussed in Section V-F). If GR_{Pcover} is greater than Th_{Pcover}, we return to

the sampling module and extract more non-FDs to search for the undiscovered FDs, otherwise, we terminate the FD discovery algorithm and present $Pcover$ as the final result. Function $invert(nonFd)$ is described in Lines 11-20. For each generalization $general$ of the given $nonFd$, we remove it from $Pcover$ in Line 13, create new FD candidates by adding another attribute to its LHS in Lines 15-17, and then check their generalizations to keep the FD candidates minimal in Lines 18-20.



(a) Invert $MBG \not\rightarrow N$ (b) Invert $AG \not\rightarrow N$ (c) Invert $AMB \not\rightarrow N$

Fig. 5: Inversion for RHS N with $MBG \not\rightarrow N$, $AG \not\rightarrow N$, and $AMB \not\rightarrow N$.

Figure 5 shows the process of inversion on the running example with attribute $Name$ as RHS. Given $Ncover$ in Section IV-D which consists of $MBG \not\rightarrow N$, $AG \not\rightarrow N$, and $AMB \not\rightarrow N$, we traverse the binary tree in depth-first order for inversion. By Lemma 1, the most general FD candidate $\emptyset \rightarrow N$ is invalid because it generalizes non-FD $MBG \not\rightarrow N$. In Figure 5(a), We remove the invalid FD $\emptyset \rightarrow N$ from $Pcover$ and create FD candidate $A \rightarrow N$ by adding A to LHS. In Figure 5(b), $A \rightarrow N$ generalizes non-FD $AG \not\rightarrow N$ and we replace it with FD candidates $AB \rightarrow N$ and $AM \rightarrow N$. Non-FD $AMB \not\rightarrow N$ is then processed in Figure 5(c). Since $AMB \not\rightarrow N$ has two generalizations $AB \rightarrow N$ and $AM \rightarrow N$ in $Pcover$, the two invalid FDs are removed and changed to $ABG \rightarrow N$ and $AMG \rightarrow N$, respectively.

The growth rate GR_{Pcover} indicates the result integrity and we continue sampling and constructing Ncover to search for the undiscovered FDs if GR_{Pcover} exceeds our expectations. However, the heavy generalization checks make the inversion module the most time-consuming in EulerFD. Thus, an appropriate Th_{Pcover} that we discuss in Section V-F is significant for the second cycle to compromise the efficiency and result integrity of our algorithm.

V. Experiments

In this section, we present experimental studies validating the efficiency and effectiveness of our proposed algorithms.

A. Experiment Setup

We compare the proposed EulerFD with state-of-the-art exact discovery algorithms and approximate discovery algorithms. As described in Section II-A, lattice traversal algorithms scale well with the number of tuples while dependency induction algorithms scale well with the number of attributes, and difference- and agree-set algorithms scale moderately with both the number of tuples and the number of attributes [25]. Because Tane outperforms other lattice traversal algorithms on large datasets [25], we take Tane, Fdep, and HyFD as the representative exact discovery algorithms. For the approximate algorithms, we choose AID-FD as the representative because the approximate discovery algorithm proposed by Kivinen et al. [16] becomes inefficient when the number of attributes is large. We implemented the following algorithms in Java and ran experiments on a machine with an Intel Core i7 3.0GHz CPU and 32GB memory.

- Tane [14]: lattice traversal algorithm.
- Fdep [11]: dependency induction algorithm
- HyFD [26]: hybrid algorithm.
- AID-FD [3]: approximate discovery algorithm.
- EulerFD: approximate discovery algorithm proposed in this paper.

We used both real-world and synthetic datasets in our experiments. In addition to the 17 datasets used in the seminal work on FD discovery [25, 26], we added two datasets $weather$ and $lineitem$ with more than 260000 and 6000000 tuples, respectively, to evaluate algorithms on large datasets. An overview of these 19 datasets with their numbers of tuples, attributes, and non-trivial minimal FDs is shown in Table III. Furthermore, we deployed EulerFD on Data Management Service (DMS) of Alibaba Cloud, processing more than 500000 real-world datasets per week with the number of columns varying from 2 to 312. Experiments on DMS were conducted on a server with an Intel XEON E7 3.0GHz CPU with 512GB memory.

B. Overall Performance

We experimentally study the efficiency and accuracy of the proposed EulerFD. Table III shows the time costs and F_1 scores of Tane, Fdep, HyFD, AID-FD, and EulerFD on 19 datasets with the fastest runtimes in bold. Each experiment was repeated 100 times for the average performance and we set a time limit (TL) of 4 hours and a memory limit (ML) of 32GB. The empirical thresholds Th_{Ncover} and Th_{Pcover} of EulerFD are set to 0.01 and 0.01, and MLFQ parameters are the same as the 6 queues in Section V-E. AID-FD adopts the same threshold 0.01 in its Ncover creation [3]. The parameter settings are followed by all subsequent experiments.

Efficiency. As shown in Table III, EulerFD is able to process all 19 datasets and performs best in terms of time cost on most datasets, especially on large datasets that are difficult for exact discovery algorithms to deal with.

Specifically, EulerFD shows advantages of efficiency on 14 out of 19 datasets (accounting for a percentage of 74%) while maintaining high accuracy. EulerFD shows no advantage only on those datasets with a small number of tuples, attributes, or FDs for which exact algorithms perform well. On large datasets, especially on $linitem$, $weather$, $fd\text{-}reduced$-30, and $uniprot$, the advantage of EulerFD is prominent, which is attributed to the sampling strategy that adjust the sampling range according to previous sampling results and the double-cycle structure that coordinates all modules properly.

Accuracy. We measure the accuracy of approximate discovery algorithms with F_1 score [33]. As shown in Table III, EulerFD achieves F_1 scores of 1.000 on 12 datasets with

TABLE III: Experimental results on real-world and synthetic $fd\text{-}reduced$-30 datasets.

Dataset	Dataset Information				Time[s]					AID-FD Details		EulerFD Details	
	Rows[#]	Cols[#]	Size[KB]	FDs[#]	Tane[14]	Fdep[11]	HyFD[26]	AID-FD[3]	EulerFD	FDs[#]	F_1 Score	FDs[#]	F_1 Score
iris	150	5	5	4	0.061	**0.028**	0.084	0.070	0.065	4	1.000	4	1.000
balance-scale	625	5	7	1	0.078	0.062	0.081	0.078	**0.061**	1	1.000	1	1.000
chess	28056	7	547	1	0.550	41.683	0.206	0.252	**0.203**	1	1.000	1	1.000
abalone	4177	9	192	137	0.191	1.397	0.164	0.129	**0.123**	138	0.989	137	1.000
nursery	12960	9	1048	1	0.780	14.062	0.208	0.236	**0.199**	1	1.000	1	1.000
breast-cancer	699	11	21	46	0.209	0.110	0.107	0.108	**0.091**	48	0.957	46	1.000
bridges	108	13	7	142	0.131	**0.028**	0.095	0.080	0.078	142	1.000	142	1.000
echocardiogram	132	13	7	527	0.096	**0.029**	0.097	0.081	0.078	526	0.997	526	0.997
adult	32561	15	3560	78	36.401	255.013	**0.546**	2.603	0.828	78	1.000	78	1.000
lineitem	6001215	16	946395	3879	ML	ML	318.513	1019.001	**216.063**	4050	0.953	3861	0.996
letter	20000	17	716	61	1149.095	94.137	**1.232**	4.665	3.391	63	0.935	60	0.975
weather	262920	18	17350	918	ML	ML	25.095	95.258	**9.564**	1289	0.772	916	0.991
ncvoter	1000	19	152	758	0.834	0.391	0.167	0.172	**0.126**	757	0.995	757	0.995
hepatitis	155	20	8	8250	3.699	0.138	0.231	0.170	**0.133**	8250	1.000	8250	1.000
horse	300	28	22	139725	10020.026	3.467	1.491	0.718	**0.629**	139721	1.000	139725	1.000
fd-reduced-30	250000	30	69581	89571	21.658	TL	27.568	13.775	**6.677**	89571	1.000	89571	1.000
plista	1001	63	576	178152	ML	13.373	3.767	1.694	**1.548**	170651	0.919	178129	0.999
flight	1000	109	569	982631	ML	114.687	7.218	2.299	**1.918**	982558	0.999	982592	1.000
uniprot	1000	223	2440	unknown	ML	ML	TL	ML	**4529.932**	-	-	146319	-

The fastest runtimes are shown in bold. **TL**: time limit of 4 hours exceeded **ML**: memory limit of 32 GB exceeded

deviations of less than 0.025, while AID-FD achieves 1.000 on 9 datasets with more erratic deviations. By analysing the FD results on the datasets with F_1 scores < 0.999, we find an overlap between the incorrect FDs discovered by AID-FD and EulerFD. That is, some rare non-FDs can only be found on a few tuples and thus are easily missed in sampling, resulting in the same error. However, due to the double-cycle structure which obtains more non-FDs by optimizing the sampling target selection, EulerFD outperforms AID-FD on all 18 datasets (excluding $uniprot$ which lacks benchmarks) with higher accuracy or the same accuracy and higher efficiency.

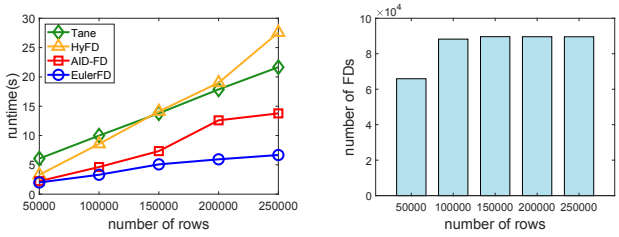
C. Row Scalability Analysis

In this subsection, we vary the number of rows (tuples) on datasets $fd\text{-}reduced$-30 and $lineitem$ to study the row scalability of EulerFD. Figures 6 and 7 show the runtimes of Tane, HyFD, AID-FD, and EulerFD as well as the numbers of FDs on $fd\text{-}reduecd$-30 and $lineitem$ with rows varying from 50000 to 250000 and 8000 to 4096000, respectively. The results of Fdep is not presented because it runs into the time limit and memory limit on the two datasets. Since the accuracy of EulerFD and AID-FD is high, we focus on the efficiency.

The runtimes of EulerFD scale with the number of FDs and increase almost linearly with row expansion. EulerFD shows the best row scalability and is more than twice as fast as AID-FD on $fd\text{-}reduced$-30 (more than 6 times on $lineitem$), because our sampling strategy uses the contributions of sampling tuples from previous sampling and dynamically revises the sampling range to serve the row scalability.

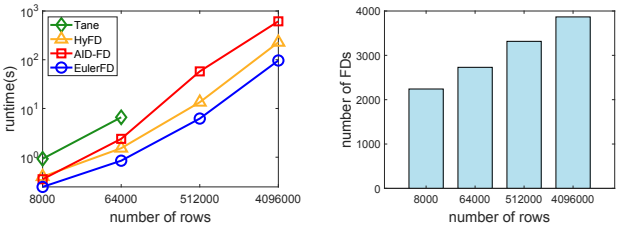
D. Column Scalability Analysis

We evaluate the column scalability by varying the number of columns (attributes) on datasets $plista$ and $uniprot$. Figure 8 and Figure 9 show the runtimes of Fdep, HyFD, AID-FD, and EulerFD as well as the numbers of FDs on $plista$ and



(a) Runtimes on $fd\text{-}reduced$-30 (b) FDs on $fd\text{-}reduced$-30

Fig. 6: Row scalability on $fd\text{-}reduced$-30.

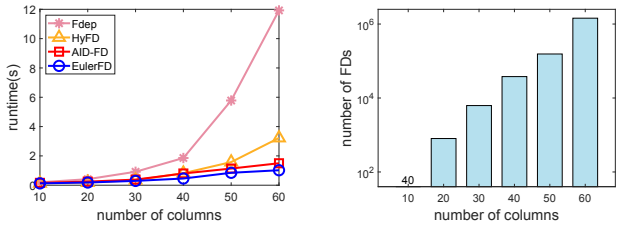


(a) Runtimes on $lineitem$ (b) FDs on $lineitem$

Fig. 7: Row scalability on $lineitem$.

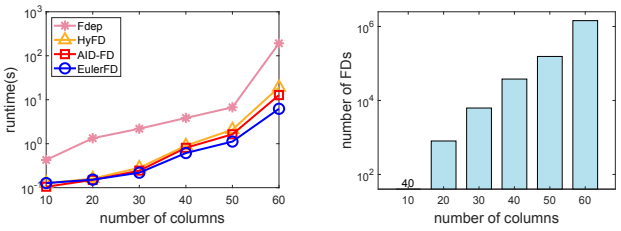
$uniprot$ with columns varying from 0 to 60. Similarly, we do not present the experimental results of Tane because it runs into the memory limit on the two datasets, and focus on the runtimes due to the high F_1 scores of EulerFD and AID-FD.

As shown in Figure 8 and Figure 9, EulerFD performs better than other algorithms with the shortest runtimes. The reason is that EulerFD adopts the approach of FD induction and inherits the good column scalability. Furthermore, EulerFD employs the proposed sampling strategy to mitigate the costly inversion of dependency induction algorithms, which strikingly contributes to its efficiency. It can be observed that a large number of FDs seriously impacts the performance of algorithms, and EulerFD outperforms other algorithms in column scalability even when there are numerous FDs.



(a) Runtimes on *plista* (b) FDs on *plista*

Fig. 8: Column scalability on *plista*.



(a) Runtimes on *uniprot* (b) FDs on *uniprot*

Fig. 9: Column scalability on *uniprot*.

E. MLFQ Parameter Evaluation

In this subsection, we analyze the parameters in MLFQ and investigate how the parameters affect the efficiency and accuracy of EulerFD. We alter the number of queues from 1 to 7, and set the *capa* ranges for different # of queues as in Table IV. Considering that the *capa* value of a cluster decreases rapidly (exponentially) during the multiple samples and the values of most clusters are not greater than 10 after the second sampling, the range of the highest priority queue is set to $[10, +\infty)$ and others are exponentially divided.

TABLE IV: Different MLFQ parameters.

# of queues	The *capa* ranges of queues (q_z to q_1)
1	$[0, +\infty)$
2	$[0, 10)$, $[10, +\infty)$
3	$[0, 1)$, $[1, 10)$, $[10, +\infty)$
4	$[0, 0.1)$, $[0.1, 1)$, $[1, 10)$, $[10, +\infty)$
5	$[0, 0.01)$, $[0.01, 0.1)$, $[0.1, 1)$, $[1, 10)$, $[10, +\infty)$
6	$[0, 0.001)$, $[0.001, 0.01)$, $[0.01, 0.1)$, $[0.1, 1)$, $[1, 10)$, $[10, +\infty)$
7	$[0, 0.0001)$, $[0.0001, 0.001)$, $[0.001, 0.01)$, $[0.01, 0.1)$, $[0.1, 1)$, $[1, 10)$, $[10, +\infty)$

Figure 10 presents the effects of different MLFQ parameters on datasets *adult*, *letter*, *plista*, and *flight*. F_1 scores increase with the increasing number of queues, while the runtimes first decrease and then slightly increase. The results indicate that the adoption of MLFQ improves the accuracy and efficiency of EulerFD, but too many queues lead to over-sampling and long runtimes. EulerFD mostly achieves the shortest runtimes with 6 queues and the highest F_1 scores with 6-7 queues. While we fix the *capa* ranges in this work, the dynamic adjustment of the *capa* ranges at runtime to achieve optimal performance is worth investigating for future work.

F. Threshold Evaluation

In this subsection, we experimentally study the influence of the two thresholds Th_{Ncover} and Th_{Pcover} on the efficiency and accuracy of approximate discovery algorithms. Given the growth rate defined as the percentage of additions in the original Ncover/Pcover, Figure 11 shows the experimental results of different thresholds 0.1, 0.01, 0.001, and 0 on four representative datasets: *flight* (with large number of attributes), *fd-reduced*-30 (with large number of tuples), *horse* (with large number of FDs), and *ncvoter* (moderate characteristics), respectively.

As shown in Figure 11, Th_{Ncover} of 0.01 is an elbow point for AID-FD to achieve a good trade-off between efficiency and accuracy. The threshold values beyond that (0.001 and 0) yields negligible accuracy improvement at the significant expense of the runtime on most datasets. For EulerFD, we set Th_{Ncover} to 0.01 when experimenting on Th_{Pcover} and vice versa. EulerFD achieves the best performance with Th_{Ncover} of 0.01 and Th_{Pcover} of 0.01 on all datasets. What's more, EulerFD outperforms AID-FD with all thresholds in terms of accuracy and efficiency. The results show that the smaller GR_{Ncover} is, the more samples EulerFD obtains, indicating higher sampling rate and accuracy. Thus, when GR_{Ncover} is smaller than 0.01, Ncover is relatively complete with almost all samples that contain non-FDs. The results verify that the number of attributes and the number of tuples have very little impact on the choice of thresholds. Intuitively, the number of FDs may have an impact on the choice of the threshold. However, since the number and distribution of FDs are unknown in advance, we choose $Th_{Ncover} = 0.01$ for AID-FD and $Th_{Ncover} = Th_{Pcover} = 0.01$ for EulerFD as universal thresholds on all datasets.

G. Performance on DMS of Alibaba Cloud

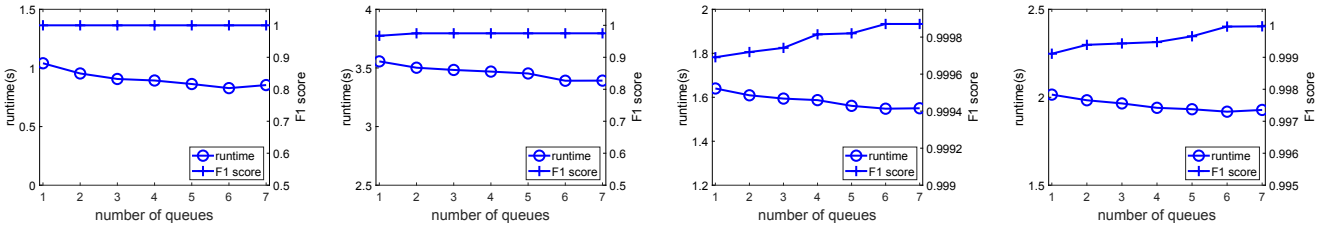
In this subsection, we report the performance of EulerFD, which has been already been deployed on DMS (https://www.aliyun.com/product/dms) of Alibaba Cloud. Since the exact discovery algorithms cannot deal with datasets with more than 223 columns (as shown in Table III), the accuracy evaluated based on benchmarks using exact discovery algorithms is not reported on large datasets.

During a go-live week from 09/12/2022 to 09/18/2022, EulerFD processed 500578 real-world datasets on DMS with columns varying from 2 to 312, and the total number of rows (columns) reaches 17386915 (433797). The execution time adds up to 1081.626 seconds and the average time cost is only 33.955 milliseconds. We compare the efficiency of EulerFD with AID-FD by τ_e as

$$\tau_e = \frac{\sum_{i=1}^{n} e_i(EulerFD)\sqrt{R_i \cdot C_i}}{\sum_{i=1}^{n} e_i(AID\text{-}FD)\sqrt{R_i \cdot C_i}},$$

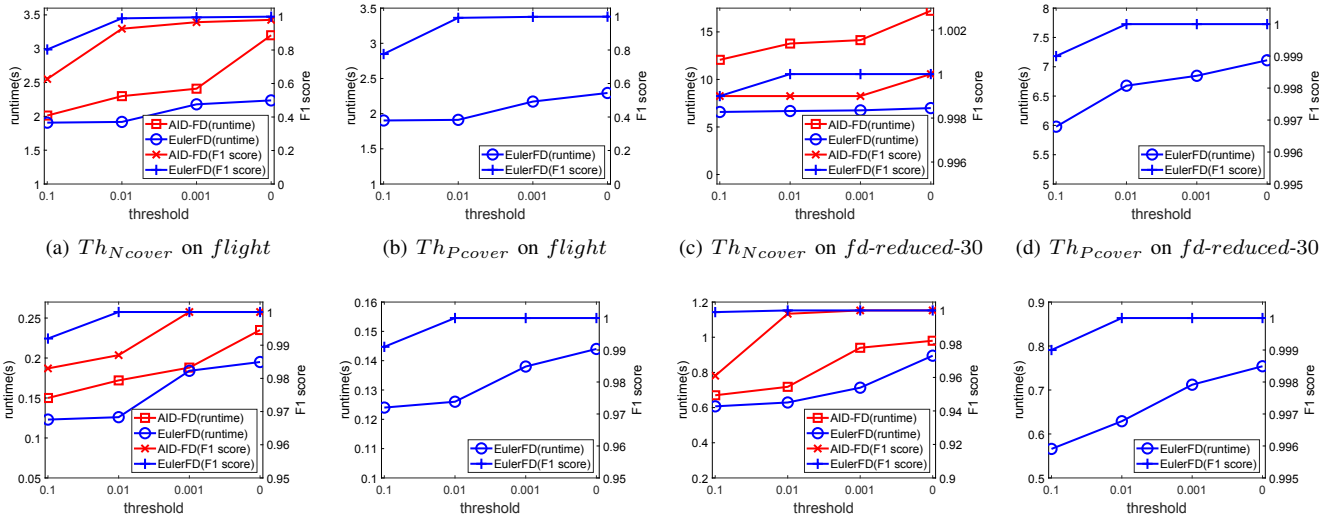
and compare the accuracy by τ_a as

$$\tau_a = \frac{\sum_{i=1}^{n} a_i(EulerFD)\sqrt{R_i \cdot C_i}}{\sum_{i=1}^{n} a_i(AID\text{-}FD)\sqrt{R_i \cdot C_i}},$$



(a) *adult* (b) *letter* (c) *plista* (d) *flight*

Fig. 10: Runtimes and F_1 scores with different MLFQ parameters on *adult*, *letter*, *plista*, and *flight*.



(a) Th_{Ncover} on *flight* (b) Th_{Pcover} on *flight* (c) Th_{Ncover} on *fd-reduced*-30 (d) Th_{Pcover} on *fd-reduced*-30

(e) Th_{Ncover} on *ncvoter* (f) Th_{Pcover} on *ncvoter* (g) Th_{Ncover} on *horse* (h) Th_{Pcover} on *horse*

Fig. 11: Runtimes and F_1 scores with different Th_{Ncover} and Th_{Pcover} on *flight*, *fd-reduced*-30, *ncvoter*, and *horse*.

where n is the number of datasets, $e_i/a_i(EulerFD)$ is the runtime/F_1 score of EulerFD on the i^{th} dataset that indicates the efficiency and accuracy, and so is $e_i/a_i(AID\text{-}FD)$. R_i and C_i represent the number of rows and columns of the i^{th} dataset, respectively. Considering the influence of dataset sizes, τ_e and τ_a reflect the algorithm performance well. Table V shows that the accuracy and efficiency of EulerFD are higher than that of AID-FD on all datasets. We also observe that the accuracy of EulerFD and AID-FD slightly decrease with the increasing number of tuples and attributes, which is caused by the sampling in the approximate discovery of FDs as expected.

TABLE V: Performance on DMS with various sizes (τ_e/τ_a).

rows \ columns	1∼10	11∼50	51∼100	100+
1∼10	0.875 / 1.000	0.701 / 1.000	0.512 / 1.000	0.404 / 1.001
11∼100	0.775 / 1.000	0.633 / 1.005	0.534 / 1.019	0.496 / 1.031
101∼1000	0.625 / 1.000	0.643 / 1.029	0.420 / 1.030	0.322 / 1.037
1001∼10000	0.423 / 1.000	0.501 / 1.051	0.314 / –	0.259 / –
10001∼100000	0.343 / 1.001	0.405 / –	0.387 / –	0.304 / –
100000+	0.389 / –	0.232 / –	0.189 / –	0.123 / –

VI. Conclusion and Future Work

In this paper, we proposed EulerFD, an efficient double-cycle approximation of the functional dependency discovery algorithm, to tackle the FD discovery problem on large datasets. Instead of validating each FD candidate by examining and verifying all tuples, we induced FDs from non-FDs which can be quickly obtained by comparing and verifying some pairs of tuples. To overcome the challenge of ambitious pair-wise comparisons, we proposed a novel sampling strategy that constantly revises the sampling range according to previous sampling results. EulerFD evaluated the stopping criteria in the double-cycle structure and adjusted the sampling strategy dynamically, which significantly improved the efficiency. Experimental results on real-world and synthetic datasets demonstrated that the proposed EulerFD is efficient and effective. Dynamic revision of sampling ranges is challenging and worth investigating for future work as a good dynamic revision strategy may significantly improve performance.

VII. Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments. This work was supported in part by NSFC grants (62102352), the National Key R&D Program of China (2021YFB3101100, 2022YFB310340), NSF grants (CNS-2124104, CNS-2125530), and NIH grants (R01ES033241, R01LM013712, UL1TR002378).

References

[1] Z. Abedjan, P. Schulze, and F. Naumann. DFD: efficient functional dependency discovery. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*, pages 949–958. ACM, 2014.

[2] G. Beskales, I. F. Ilyas, L. Golab, and A. Galiullin. Sampling from repairs of conditional functional dependency violations. *VLDB J.*, 23(1):103–128, 2014.

[3] T. Bleifuß, S. Bülow, J. Frohnhofen, J. Risch, G. Wiese, S. Kruse, T. Papenbrock, and F. Naumann. Approximate discovery of functional dependencies for large datasets. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*, pages 1803–1812. ACM, 2016.

[4] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 746–755. IEEE Computer Society, 2007.

[5] P. Brown and P. J. Haas. BHUNT: automatic discovery of fuzzy algebraic constraints in relational data. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 668–679. Morgan Kaufmann, 2003.

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

[7] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), page 335–344, New York, NY, USA, 1962. Association for Computing Machinery.

[8] G. Cormode, L. Golab, F. Korn, A. McGregor, D. Srivastava, and X. Zhang. Estimating the confidence of conditional functional dependencies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 469–482. ACM, 2009.

[9] W. Fan, F. Geerts, L. V. S. Lakshmanan, and M. Xiong. Discovering conditional functional dependencies. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 1231–1234. IEEE Computer Society, 2009.

[10] W. Fan, X. Liu, and Y. Cao. Parallel reasoning of graph functional dependencies. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 593–604. IEEE Computer Society, 2018.

[11] P. A. Flach and I. Savnik. Database dependency discovery: A machine learning approach. *AI Commun.*, 12(3):139–160, 1999.

[12] C. Ge, I. F. Ilyas, and F. Kerschbaum. Secure multi-party functional dependency discovery. *Proc. VLDB Endow.*, 13(2):184–196, 2019.

[13] M. Haddad, J. Stevovic, A. Chiasera, Y. Velegrakis, and M. Hacid. Access control for data integration in presence of data dependencies. In *Database Systems for Advanced Applications - 19th International Conference, DASFAA 2014, Bali, Indonesia, April 21-24, 2014. Proceedings, Part II*, volume 8422 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2014.

[14] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: an efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

[15] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 647–658. ACM, 2004.

[16] J. Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. *Theor. Comput. Sci.*, 149(1):129–149, 1995.

[17] J. Kossmann, T. Papenbrock, and F. Naumann. Data dependencies for query optimization: a survey. *VLDB J.*, 31(1):1–22, 2022.

[18] S. Kruse and F. Naumann. Efficient discovery of approximate dependencies. *Proc. VLDB Endow.*, 11(7):759–772, 2018.

[19] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 96–107. Morgan Kaufmann, 1994.

[20] J. Liu, J. Li, C. Liu, and Y. Chen. Discover dependencies from data - A review. *IEEE Trans. Knowl. Data Eng.*, 24(2):251–264, 2012.

[21] E. Livshits, B. Kimelfeld, and J. Wijsen. Counting subset repairs with functional dependencies. *J. Comput. Syst. Sci.*, 117:154–164, 2021.

[22] S. Lopes, J. Petit, and L. Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000, Proceedings*, volume 1777 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2000.

[23] R. J. Miller, M. A. Hernández, L. M. Haas, L. Yan, C. T. H. Ho, R. Fagin, and L. Popa. The clio project: Managing heterogeneity. *SIGMOD Rec.*, 30(1):78–83, 2001.

[24] N. Novelli and R. Cicchetti. FUN: an efficient algorithm for mining functional and embedded dependencies. In *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*, volume 1973 of *Lecture Notes in Computer Science*,

pages 189–203. Springer, 2001.

[25] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proc. VLDB Endow.*, 8(10):1082–1093, 2015.

[26] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 821–833. ACM, 2016.

[27] T. Papenbrock and F. Naumann. Data-driven schema normalization. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 342–353. OpenProceedings.org, 2017.

[28] G. N. Paulley. *Exploiting functional dependence in query optimization*. Citeseer, 2001.

[29] E. H. M. Pena, E. C. de Almeida, and F. Naumann. Discovery of approximate (and exact) denial constraints. *Proc. VLDB Endow.*, 13(3):266–278, 2019.

[30] A. A. Qahtan, N. Tang, M. Ouzzani, Y. Cao, and M. Stonebraker. ANMAT: automatic knowledge discovery and error detection through pattern functional dependencies. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1977–1980. ACM, 2019.

[31] A. A. Qahtan, N. Tang, M. Ouzzani, Y. Cao, and M. Stonebraker. Pattern functional dependencies for data cleaning. *Proc. VLDB Endow.*, 13(5):684–697, 2020.

[32] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts*. John Wiley & Sons, 2006.

[33] C. J. van Rijsbergen. *Information Retrieval*. Butterworth, 1979.

[34] Z. Wei, S. Hartmann, and S. Link. Discovery algorithms for embedded functional dependencies. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 833–843. ACM, 2020.

[35] Z. Wei, S. Hartmann, and S. Link. Algorithms for the discovery of embedded functional dependencies. *VLDB J.*, 30(6):1069–1093, 2021.

[36] C. M. Wyss, C. Giannella, and E. L. Robertson. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract. In *Data Warehousing and Knowledge Discovery, Third International Conference, DaWaK 2001, Munich, Germany, September 5-7, 2001, Proceedings*, volume 2114 of *Lecture Notes in Computer Science*, pages 101–110. Springer, 2001.

[37] H. Yao, H. J. Hamilton, and C. J. Butz. Fd_mine: Discovering functional dependencies in a database using equivalences. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan*, pages 729–732. IEEE Computer Society, 2002.

[38] Y. Zhang, Z. Guo, and T. Rekatsinas. A statistical perspective on discovering functional dependencies in noisy data. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 861–876. ACM, 2020.

[39] Z. Zheng, M. Alipour, Z. Qu, I. Currie, F. Chiang, L. Golab, and J. Szlichta. Fastofd: Contextual data cleaning with ontology functional dependencies. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 694–697. OpenProceedings.org, 2018.