Discovery and Ranking of Functional Dependencies

Ziheng Wei
The University of Auckland
Email: zwei891@aucklanduni.ac.nz

Sebastian Link
The University of Auckland
Email: s.link@auckland.ac.nz

***Abstract*—Computing the functional dependencies that hold on a given data set is one of the most important problems in data profiling. Our research advances state-of-the-art in various ways. Utilizing new data structures and original techniques for the dynamic computation of stripped partitions, we devise a new hybridization strategy that outperforms the best algorithms in terms of efficiency, column-, and row-scalability. This is demonstrated on real-world benchmark data. We show that current outputs contain many redundant functional dependencies, but canonical covers greatly reduce output sizes. Smaller representations of outputs are easier to comprehend and use. We propose the number of redundant data values as a natural measure to rank the output of discovery algorithms. Our ranking assesses the relevance of functional dependencies for the given data set.**

TABLE I. SNIPPET OF NCVOTER TO ILLUSTRATE DATA REDUNDANCY

voter_	first_	last_	name_	gen	street_			zip_
id	name	name	suffix	der	address	city	state	code
131	joseph	cox		m	1108 highland ave	**new bern**	nc	28562
131	joseph	cox		m	9 casey rd	**new bern**	nc	28562
657	essie	warren		f	105 south st	lasker	nc	27845
725	lila	morris		f	500 w jefferson st	jackson	nc	27845
244	sallie	futrell		f	9802 us hwy 258	**murfreesboro**	nc	27855
247	herbert	futrell		m	9802 us hwy 258	**murfreesboro**	nc	27855
440	barbara	johnson		f	6155 kimesville rd	**liberty**	nc	27298
464	albert	johnson		m	6155 kimesville rd	**liberty**	nc	27298
265	w	johnson		m	11957 us hwy 158	**conway**	nc	27820
272	clyde	johnson		m	8944 us hwy 158	**conway**	nc	27820
26	louise	johnson		f	113 gentry st #20	**wilkesboro**	nc	28659
42	walter	johnson		m	169 otis brown dr	**wilkesboro**	nc	28659
604	christine	davenport		f	1710 matthews rd	**robersonville**	nc	27871
751	christine	hurst		f	106 w purvis st	**robersonville**	nc	27871

I. INTRODUCTION

Data profiling comprises the activities that determine meta data about a given data set [1]. In practice, data profiling is a scientific approach towards data preparation, a resource-intense task in data science projects. Applications include data cleaning, integration, repository design, quality, preparation for analytics, and query optimization [1]. A fundamental task in data profiling is the discovery of data dependencies that hold on the given data set. Since the 1980s many advances have been made. We will focus on functional dependencies (FDs). These have received most attention from academia and industry, due to their usefulness in many applications [2], [6], [9], [10], [12], [14], [16], [19], [20].

An FD $X \rightarrow Y$ with column sets X and Y expresses that the combination of values on the columns in X uniquely determines the value on each of the columns in Y. The discovery problem for FDs is to compute the set of FDs that are satisfied by a given relation. For example, the benchmark data set *ncvoter* with 1,000 rows and 19 columns exhibits 758 FDs with minimal left-hand sides (LHSs), and 3,754 total occurrences of attributes in those FDs. In general, the discovery problem is computationally challenging. There are relations over any given number of columns whose best representation of the satisfied FDs is of exponential size [13]. The decision variant is to decide for a given relation r and a given positive integer k if there is an FD $X \rightarrow A$ with $A \notin X$ and $|X| \leq k$ that is satisfied by r. The decision variant is NP-complete [5] and $W[2]$-complete in k [4]. Despite these fundamental barriers to generally efficient solutions, known algorithms can quickly solve many real-world instances. For example, row/column-based algorithms are efficient whenever the given data set has few columns/rows, respectively. However, real-world data, especially big data, have typically many rows and columns. The recent hybrid algorithm [16] combines row- and column-based approaches to address larger data sets. For example, it can find the 40,195 LHS-reduced FDs that the benchmark data set *diabetic* with 101,766 rows and 30 columns exhibits in 2,865 seconds using 2,253 MB of main memory. In the hybrid algorithm, a switch of strategies occurs whenever the current strategy is not working well, that is, if either too many FDs are invalidated or too few invalid FDs are found. However, a switch from the current strategy is never based on evidence that the other strategy will be successful. The lack of evidence leaves room to improve efficiency and scalability, which can make data sets with more rows, columns, and FDs accessible to FD discovery. Our first major contribution is a new hybrid algorithm with innovations in strategy and technology. Strategically, we switch from a column- to a row-based approach whenever it is likely that many FDs can be validated. Technically, this is made possible by a novel data structure and the first algorithm that computes stripped partitions dynamically. Experiments show that our algorithm leverages conservative main memory resources to outperform the state-of-the-art in run times, row- and column-scalability. For example, it discovers the 40,195 LHS-reduced FDs of *diabetic* within 848 seconds using 4,301 MB of main memory.

The aim of discovery algorithms is to represent the set of valid FDs efficiently. In previous work the representation is a left-reduced cover, minimizing the LHS X of FDs $X \rightarrow Y$. This has two shortcomings. Firstly, left-reduced covers may contain many redundant FDs. Non-redundant representations are smaller, and easier to process for computers and humans. Our second main contribution shows how quickly canonical covers can be computed and how much they reduce output sizes. They achieve an average of 50% savings on benchmark data. Ten of our data sets are smaller and achieve a reduction by 25%, while the remaining eleven data sets are larger and achieve a reduction by over 70%. For example, a canonical cover for *ncvoter* consists of only 185 FDs with 927 total attribute occurrences, reducing the left-reduced cover by approximately 4 times in size. The canonical cover can be computed in 0.023 seconds from the left-reduced cover.

Secondly, the output of FD discovery algorithms is not ranked. The more FDs are returned for a given data set, the more difficult it becomes for users to assess their relevance. As stated before [16], ultimately, a domain expert must assess whether an FD is meaningful for the application domain. Even though FDs that only hold accidentally on the data set are still useful for some applications, such as query optimization, it is still beneficial to automatically rank the *relevance* of the discovered FDs for the given data set. As our third major contribution we propose *data redundancy* as a natural measure of relevance. It is natural for at least two reasons. 1) FDs are a major source for data redundancy, having brought forward Boyce-Codd and Third Normal Form proposals [11], [17]. Consequently, the number of redundant data values caused by an FD indicates the relevance of this FD for normalization. 2) Data redundancy caused by an FD $X \to Y$ measures how many instances of the pattern "X-value determines Y-value" actually occur in the data set, again showing how relevant the pattern is. Applying our ranking to the FDs exhibited by real-world benchmark data, a quantitative and qualitative analysis illustrates that our measure can provide effective guidance for data stewards in assessing the relevance of discovered FDs. Finally, we report results for the two most common interpretations of missing values. Since values are missing frequently, such distinction is important for applications.

For illustration consider *ncvoter*, with a small snippet shown in Table I. Among many FDs, the full data set satisfies $\sigma_1 = \emptyset \to$ *state*, $\sigma_2 =$ *last_name, zip_code* $\to$ *city*, $\sigma_3 =$ *last_name, gender, zip_code* $\to$ *name_suffix*, and $\sigma_4 =$ *voter_id* $\to$ *state*. Recall that the occurrence of a data value is *redundant* for a set Σ of constraints [17] whenever every change of this value to a different value at this occurrence incurs a violation of some constraint in Σ. Hence, the value is fixed for this occurrence given Σ. For example, changing any occurrence of the *state* value 'nc' will result in the violation of σ_1. This FD expresses that the state value is constant in the data set. It is meaningful because the data set only considers voters from the state 'nc'. As a consequence, FD σ_1 causes 1,000 data value occurrences to be redundant. Similarly, FD σ_2 causes 182, FD σ_3 causes 61, and FD σ_4 causes 2 redundant occurrences in *ncvoter*. For instance, each of the bold occurrences in Table I are redundant due to the FD σ_2. While highly ranked FDs attract interest from data stewards, low ranked FDs do, too. For example, FD σ_4 has the likely key *voter_id* on its LHS, and the only violation of the key in the full data set is illustrated by the first two tuples in Table I. More insight unfolds when we exclude null markers from redundant occurrences. In this case, FD σ_3 causes only 2 redundant occurrences instead of the 61 when null markers are included. If nearly all redundant data values caused by an FD are null markers, then it is likely that the FD is not relevant for the data set.

Organization. We explain our contributions over related work in Section II, fix notation in Section III, present our discovery algorithm in Section IV, explain our experimental results for our discovery algorithms in Section V, and for our rankings in Section VI, respectively. We conclude and outline future work in Section VII. More details have been made available in a technical report [18].

II. Related Work

Revisiting a large body of previous work, we describe how we advance the FD discovery problem by i) new techniques that improve state-of-the-art in efficiency, row-, and column-scalability, and ii) the size and ranking of the output.

FD discovery. Since the 1980s, many FD discovery algorithms addressed data sets with a large number of *either* rows *or* columns. An important technique models the search space of FDs as an *attribute lattice* [9]. This is traversed level by level from smaller to larger sets of attributes. An attribute set is pruned if no attributes are functionally dependent on the set. Other column-based algorithms introduced different pruning and lattice traversal strategies [2], [14], [20]. Row-based algorithms use *agree sets*, determined by all pairs of distinct rows in the input. Based on maximal agree sets [10] or their set complements [19], the algorithms use hypergraph transversals to generate the output FDs. An FD-tree manages an FD set [6]. Examining all agree sets iteratively, an FD-tree is updated until it represents the output set. A column-based hybrid algorithm was introduced for the discovery of minimal keys [7]. Their algorithm traverses from the top and bottom of the attribute lattice simultaneously, essentially 'halving' the search space by faster pruning. Combining the row-based algorithm from [9] with the column-based algorithm from [6] was used to discover FDs [16]. The column-based part validates the FDs of an FD-tree [6] and switches to the row-based algorithm when too many FDs are invalidated. The row-based part generates FDs that do not hold on the input, and switches to the column-based part whenever too few of such invalid FDs are found. *Novelty.* Our article introduces extended FD-trees and a dynamic data manager (DDM) for stripped partitions. Our new hybrid strategy follows the column-based approach over extended FD trees, but uses the DDM as a row-based technique when many FDs are likely to be valid.

Covers. FD sets can be represented by different notions of *covers* [11]. A *non-redundant* cover does not contain any FD that is implied by the remaining FDs in the cover. The results of previous algorithms are given by *left-reduced* covers. That is, for each column A the cover contains all valid FDs $X \to A$ with minimal LHS X. *Canonical* covers are non-redundant left-reduced covers with unique LHSs [11]. *Novelty.* We demonstrate on real-world benchmarks that canonical covers can greatly reduce output sizes with typically small overheads of running time.

Ranking FDs. Despite the savings by canonical covers, not all FDs are equally relevant for the given data set. We are unaware of any measures for ranking discovered FDs. Recently, genuine FDs were used to estimate which FDs are likely to hold on the 'true' completion of an incomplete data set by imputing null marker occurrences [3]. However, genuine FDs do not say anything about the relevance of the FDs for the underlying data set. Instead, we regard the number of redundant data value occurrences that an FD causes as the arguably most natural measure for relevance. The notion of a redundant data value occurrence [17] justifies schema normal forms for Boyce-Codd, Third, and Fourth normal forms [11], [17]. It has never been used to rank FDs. *Novelty.* We measure the relevance of an FD for a data set by the number of redundant data value occurrences it causes.

III. Preliminaries

We fix some notions and notation required for the exposition of our approach.

A relation schema is a finite, non-empty set R of attributes (or columns). With each attribute A, we associate a domain $dom(A)$ of values that can occur in A. We assume a total order on R, that is, $R = \{A_1, \cdots, A_n\}$. This allows us to use positive integers to identify columns. For $X = \{A_1, A_2, \cdots, A_n\}$, we write X as $A_1 A_2 \cdots A_n$ and XY as the set union $X \cup Y$. A *tuple* t over R, or *row*, maps each $A \in R$ to a value in $dom(A)$. Two tuples are equal if they have matching values on all the attributes, and distinct otherwise. For $X \subseteq R$ and a tuple t over R, $t(X)$ denotes the projection of t onto X. A *relation* is a finite set of tuples. The *active domain* of $A \in R$ for a given relation r is $adom_r(A) = \{t(A) \mid t \in r\}$.

A *functional dependency* (FD) over R has the form $X \rightarrow Y$ where $X, Y \subseteq R$. We call X the *left-hand-side* (LHS) and Y the *right-hand-side* (RHS) of the FD. A relation r *satisfies* the FD $X \rightarrow Y$ (or $X \rightarrow Y$ holds on r), denoted by $r \vDash X \rightarrow Y$, if for all $t, t' \in r$, $t(X) = t'(X)$ implies $t(Y) = t'(Y)$. If r does not satisfy $X \rightarrow Y$, we say r *violates* $X \rightarrow Y$. For fixed r, we say that the FDs satisfied by r are *valid*. We write $X \not\rightarrow Y$ if none of the FDs $X \rightarrow A$ for any $A \in Y$ holds on r. In consistency with previous work but by abuse of terminology, we also say in this case that $X \rightarrow Y$ is *invalid*. Non-FDs are invalid FDs where the RHS is the complement of the LHS, that is, $X \not\rightarrow R - X$. For an FD set $\Sigma \cup \{X \rightarrow Y\}$, relation r *satisfies* Σ, denoted by $r \vDash \Sigma$, if r satisfies all the FDs in Σ. We say Σ *implies* $X \rightarrow Y$, denoted by $\Sigma \vDash X \rightarrow Y$, if every relation that satisfies Σ also satisfies $X \rightarrow Y$. Σ' is a *cover* of Σ whenever Σ and Σ' imply the same set of FDs. The *FD discovery problem* is to compute for any given relation r a cover for the set of FDs that hold on r.

An FD set Σ is *left-reduced* if for every FD $X \rightarrow Y \in \Sigma$ and for all proper subsets $Z \subset X$ it is not true that $\Sigma \vDash Z \rightarrow Y$ holds. State-of-the-art algorithms such as [16] represent their output as left-reduced covers where every RHS consists of a single attribute. An FD set Σ is *non-redundant* if there is no FD $\sigma \in \Sigma$ that is already implied by the set $\Sigma - \{\sigma\}$ of remaining FDs, that is, if there is no FD $\sigma \in \Sigma$ such that $\Sigma - \{\sigma\} \vDash \sigma$. In this article we are specifically interested in computing left-reduced, non-redundant covers with unique LHSs. These are also known as *canonical covers* [11].

The X-equivalence class of tuple $t \in r$ is the set $[t]_X = \{s \in r \mid s(X) = t(X)\}$. The *stripped partition* of r over X is $\pi_X(r) = \{[t]_X \mid t \in r, |[t]_X| \geq 2\}$. We sometimes omit r and write π_X if r is fixed. We use $|\pi_X(r)|$ and $||\pi_X(r)||$ to denote the number of sets (aka the cardinality) in $\pi_X(r)$, and the total number of tuples (aka the size) in the sets of $\pi_X(r)$, respectively. In previous FD discovery algorithms [9], [16], stripped partitions either sample non-FDs or validate candidate FDs, but sampling and validation have never been combined. The reason is that no technique had been devised that can manage stripped partitions efficiently. They have huge memory requirements if the size of a relation or the number of valid FDs is large. We establish the first technique of dynamically computing stripped partitions. This overcomes memory limitations of static computations, using memory effectively whenever more FDs are likely to be valid.

IV. Discovery Algorithms

We introduce the *dynamic hybrid algorithm* (DHyFD). We describe how previous approaches can be improved, describe our extension of FD trees, and our dynamic data manager. These result in our new hybrid strategy which leads to DHyFD.

A. Revisiting previous algorithms

Column-based Algorithm. The column-based algorithm in [9] models the search space of FDs as an *attribute lattice*. The algorithm traverses the lattice from bottom to top. At each level, the LHS and RHS of an FD are attribute sets. Each pair of LHS and RHS is validated using the stripped partition of the LHS. If an FD does not hold, new LHSs on the next level for the same RHS attribute are generated. The stripped partition of the new LHSs are computed. This process is infeasible if there are too many columns. Generating LHSs by levels typically enumerates the entire lattice if valid FDs exist at different levels. Stripped partitions duplicate the original input aggressively, consuming any available memory for inputs with too many rows. Stripped partitions are efficient for FD validation. Validating $X \rightarrow Y$ usually requires a mapping from the X-values to their Y-values. Once duplicated X-values are found, unmatched $A \in Y$-values invalidate $X \rightarrow A$. This method is inefficient on larger inputs. For example, validating $X \rightarrow A$ and $XB \rightarrow A$ (assume $X \rightarrow A$ is invalid) creates mappings for X- and XB-values. Hence, many X-values are computed redundantly. Such redundancy causes inefficiency if the input contains too many rows and columns. If π_{XB} could be generated dynamically from π_X, then only B-values needed extraction. Computation would become more efficient. Hence, using previously computed stripped partitions can decrease the cost of FD validation effectively. Another challenge is to let stripped partitions consume only reasonable memory.

Row-based Algorithm. The *agree sets* of the row-based algorithm [6] consists of those attributes on which two tuples have matching values. That is, $ag(t, t') = \{A \in R \mid t(A) = t'(A)\}$. The *agree set* of r is the set of agree sets for all pairs of distinct tuples in r, that is, $ag(r) = \{ag_r(t, t') \mid t, t' \in r, t \neq t'\}$. Importantly, the agree set for each pair of distinct tuples implies the non-FD $ag(t, t') \rightarrow R - ag(t, t')$. Starting with the FD $\emptyset \rightarrow R$, the row-based algorithm processes the sets in $ag(r)$ iteratively, inducing new FDs that do not contradict any of the non-FDs implied by the agree sets processed so far. Details are in Section IV-C. Inducing FDs from non-FDs uses a tree-like data structure called FD-tree [6]. As in Figure 1, the LHS of an FD is represented by a path in the FD-tree where each node represents an attribute in the LHS with labels of RHS attributes. An FD-tree provides quick access to all FDs that do not contradict any of the non-FDs processed so far. A new observation is that the induction algorithm of [6] only handles singleton RHS of implied non-FDs. Instead of inducing new FDs based on the non-FD $X \not\rightarrow R - X$, the process iteratively induces new FDs based on the invalid FDs $X \not\rightarrow A$ for all $A \in R - X$. FD-trees are implemented as a linked data structure. Hence, path traversal is costly as links are maintained by heap memory. Our observation can thus significantly reduce time spent on FD induction as the number of non-FDs can be quadratic in the number of tuples.

Hybridization. The strategy of the sampling-focused hybrid algorithm [16], see the left of Figure 2, has three components

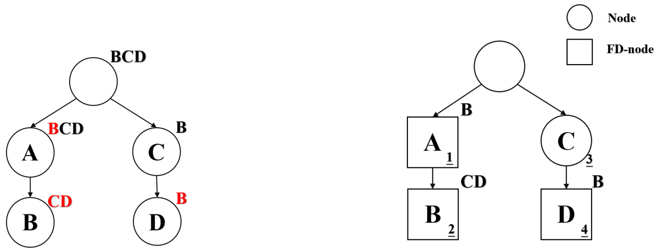

Fig. 1. The FD-tree (left) and extended FD-tree (right) for FDs $A \rightarrow B$, $AB \rightarrow CD$, and $CD \rightarrow B$

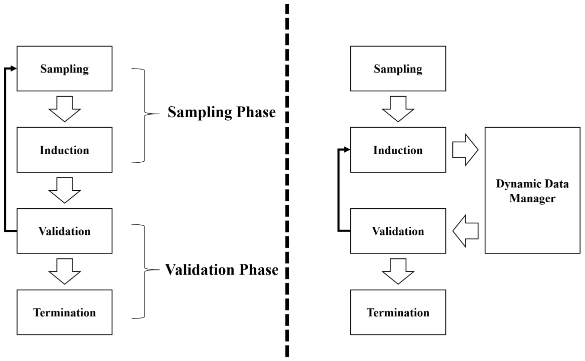

Fig. 2. Sampling-validation hybridization strategy

and two phases. On input r over R, the *sample component* computes a stripped partition for each attribute in R. Then the *sorted neighborhood pair selection method* [8] extracts non-FDs by sampling agree sets of the stripped partitions. The agree sets are sampled from tuple pairs of the same equivalence class in the stripped partition. Here the underlying sorting algorithm of the method determines the neighborhood of the tuples. The *validation component* validates candidate FDs, and uses invalid FDs to update an FD-tree. The *induction phase* uses either the non-FDs from the sample component or the invalid FDs from the validation component to induce new FDs. The algorithm [16] starts with the *sampling phase*, and then applies invalid FDs to update the FD-tree. Once too few new samples (non-FDs) are generated, the algorithm switches to the *validation phase*. Likewise, it switches back to the sampling phase if too many FDs are invalidated.

The hybrid algorithm implements the row-based algorithm as induction component. No advantage is taken of the column-based approach in which stripped partitions reduce redundant computations of values on the LHS of FDs. However, adapting the validation method from the column-based algorithm proves challenging. Here, a larger stripped partition is computed by joining the stripped partitions of two adjacent *prefix blocks* [9]. For example, π_{XAB} is computed by intersecting π_{XA} and π_{XB}. Hence, all invalid LHSs need to be known before the stripped partitions on a higher level can be computed. Due to the randomness in FD induction some of the invalid LHSs are eliminated by invalid FDs. This means each level of the FD-tree contains only some but not all of the invalid FDs. Hence, FD induction can make the computation of stripped partitions obsolete. The major challenge in hybridizing column- and row-based algorithms is to dynamically compute stripped partitions while avoiding excessive memory consumption. It is always better not to use invalid FDs from the validation component if more general non-FDs can be found. For example, the invalid FD $X \not\rightarrow Y$ over R could be inefficient in two ways. Firstly, there may be a non-FD $X \not\rightarrow R - X$ such that $Y \subseteq R - X$. Secondly, $X \not\rightarrow Y$ may induce new FDs that could still be invalidated by a non-FD $X' \not\rightarrow R - X'$ where $X \subseteq X'$.

B. An Overview of DHyFD

We introduce the *dynamic hybrid algorithm* for FD discovery (DHyFD), as shown on the right of Figure 2. Firstly, DHyFD performs *synergized FD inductions* on an *extended FD-tree*. In comparison, the extended FD-tree is more efficient for searching FDs than the classical FD-tree [6], and the new induction method dramatically eliminates redundant traversals. Secondly, DHyFD introduces a *dynamic data manager* (DDM) to help with FD validation and non-FD sampling. Balancing the main memory use by stripped partitions with the cost of validation, DDM dynamically refines stripped partitions so that FD candidates can be validated efficiently. While validating FDs, new non-FDs are extracted. As the refined partition contains more equivalence classes with fewer tuples, the extraction of non-FDs based on tuple pairs from the same class is more efficient. The non-FDs are finally applied to the extended FD-tree to derive new FDs.

Before going into details, we outline DHyFD as illustrated in Figure 3. Given a relation r over schema R, DDM pre-computes stripped partitions for each singleton attribute of R. The partitions are refined later dynamically. Before any iteration, a set of initial non-FDs (aka *initial samples*) are sampled from the pre-computed stripped partitions. This step extracts a wide range of non-FDs to perform *synergized induction* on the extended FD-tree, providing a good first approximation of the final FD-tree. DHyFD then starts validating the extended FD-tree level by level. The current level is called the validation level (vl). DHyFD validates the FDs on the current vl by stripped partitions from DDM. During validation, a set of non-FDs is generated. Afterwards, *synergized induction* uses these non-FDs to derive new FDs. After induction, DHyFD decides if DDM can perform better by refining stripped partitions. The decision applies a novel measure called *efficiency-inefficiency ratio*. We refer to the validation level vl at which the latest refinement occurred as the controlled level (cl). In essence, the refined stripped partitions are based on the FDs associated with nodes at the controlled level. At the end of an iteration, DHyFD either computes the nodes at the next validation level or terminates if the extended FD-tree does not require further traversal. Next, we provide the details of each component.

C. Extended FD-Trees

FD-trees [6] facilitate FD induction. We enhance FD induction by extended FD-trees that help derive candidate FDs and validate them faster.

Let Σ be a set of FDs over R. Assuming attributes of R are integers, an *extended FD-tree* has the following properties: (1) A unique *root node* represents the empty LHS; (2) Each node represents an attribute in R except the root node; (3) Each node only has children of larger attributes; (4) For each FD $A_1 \dots A_n \rightarrow Y \in \Sigma$, there is a path representing $A_1 \dots A_n$ where A_n is called an FD-node; (5) Each FD-node

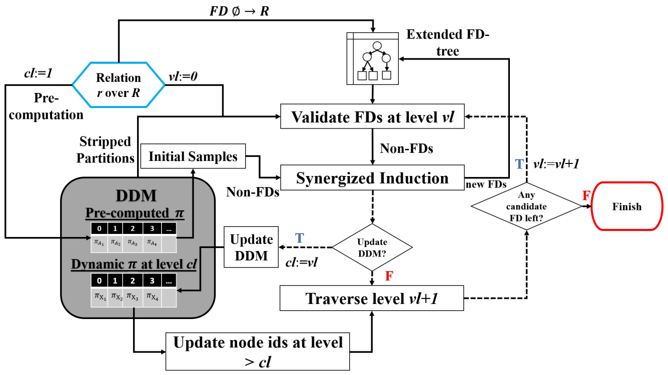

Fig. 3. Overview of DHyFD

is associated with a non-empty attribute set as the RHS of an FD; (6) Each node is assigned a positive integer *id*; (7) The default id of a node $A \in R$ is A; and (8) Given a DDM of relation r over R and a node with id i where $i > |R|$, the $(i - |R|)$-th stripped partition in the DDM is $\pi_{X'}$ where $X' \subseteq X$ and X is the path from the root to the node.

Example 1: The right of Figure 1 shows an extended FD-tree of the FDs $A \rightarrow B$, $AB \rightarrow CD$, $CD \rightarrow B$. The integer ids are shown at the bottom right. The extended FD-tree has fewer RHS labels than the FD-tree on the left of Figure 1.

The novelty of extended FD-trees is a new type of node, called FD-node, which stores RHS attributes of FDs. In contrast, nodes of an FD-tree store RHS attributes not only for the FDs represented by themselves but also for FDs represented by their descendants. In the left of Figure 1 the root node and the nodes at level 1 all have B as a RHS attribute even though only node A represents FD $A \rightarrow B$. This overhead in labeling is often inefficient. Thus, an FD-tree requires typically more maintenance of RHS attributes than extended FD-trees. Excessive labeling of RHS attributes does not accelerate FD search. For example, on the left of Figure 1, if we search for $X \rightarrow B$, checking nodes for FDs with attribute B on the RHS does not prune the search space: indeed, the root node and the nodes at level 1 all have B as a RHS attribute.

A core idea of DHyFD is to smartly apply stripped partitions for the validation of FD candidates that are associated with FD-nodes on the current validation level vl. Smartly means that the stripped partitions were dynamically computed with respect to the controlled level $cl \leq vl$, that is, from the paths of length cl in the extended FD-tree. A key observation is that, even if $cl < vl$, the stripped partition $\pi_{X'}$ of the controlled level cl can still be used to validate the node's underlying FD $X \rightarrow Y$ once $\pi_{X'}$ has been further refined to π_X (see Algorithm 5 for details). However, this refinement can only happen when X' is a subset of X. An array of the stripped partitions with respect to the controlled level cl is maintained by the DDM. In addition, the DDM assigns integer ids to nodes in the extended FD-tree that can index this array. For this to happen, the node id must be *consistent*, that is, the attribute set X' of the stripped partition $\pi_{X'}$ must be a subset of the path that leads to the FD-node representing $X \rightarrow Y$ on the current validation level vl. Hence, a major task of DDMs is to assign consistent ids to the nodes of the extended FD-tree. During FD validation, DHyFD must know all the nodes at vl such that the nodes on the next level can be traversed. However, after validating current FD candidates, the FD induction process may introduce new nodes by inserting a completely new path or extending an existing one.

Example 2: Let FD $AC \rightarrow E$ be the only path in an extended FD-tree over $R = \{A, B, C, D, E\}$. If non-FD $AC \not\rightarrow BDE$ is applied to the tree, $ABC \rightarrow E$ and $ACD \rightarrow E$ are induced. To add $ACD \rightarrow E$, RHS E of FD-node C is removed and a child FD-node D is appended to node C. In the end, node C is no longer an FD-node and its child FD-node D stores E as RHS. To add $ABC \rightarrow E$, a new path ABC is created from node A since the only existing path is ACD. Suppose the validation level is 2 before non-FD $AC \not\rightarrow BDE$ is implied. One can only retrieve FD-node C from level 2 of the tree. After induction, node B from the new path ABC is added to level 2. Without knowing the new node B, it is impossible to validate $ABC \rightarrow E$ by only exploring the children of node C at level 2.

Algorithm 1 assigns consistent ids while adding a new FD path. The algorithm checks if the new FD requires new nodes at the end of some path (step 5-8). If it does, the new nodes are assigned the ids from their ancestors at the controlled level (step 11 - 14). The new nodes are added to the validation level (step 15) for correct traversal of the FD-tree.

Algorithm 1 Add FD

```
 1: Input:  An $FD\ X \rightarrow Y$ over $R$, the root node $root$ of an FD-tree,
    controlled level $cl$, the set $vl\_nodes$ of all nodes at validation level $vl$
 2: Output:  a new $FD$-tree with $FD\ X \rightarrow Y$
 3: $current = root$
 4: $i = 1, n = |X|$
 5: while $i \leq n$ do
 6:     if $current$ has child $c$ of $A_i \in X$ then
 7:         $current = c, i = i + 1$
 8:     else break
 9: while $i \leq n$ do
10:     Create a new node $c$ of $A_i \in X$ as the child of $current$
11:     if $i > cl$ then
12:         Assign id of $current$ as id of $c$
13:     else
14:         Assign the order of $A_i$ in $R$ as id of $current$
15:     if $i = vl$ then $vl\_nodes = vl\_nodes \cup \{c\}$
16:     $current = c, i = i + 1$
17: Let $rhs(current)$ be the RHS of $current$
18: $rhs(current) = rhs(current) \cup Y$
19: Return  $root$
```

D. Synergized Induction

Given a non-FD $X \not\rightarrow Y$ over R, classical FD induction [6] updates a given FD set over R using $X \not\rightarrow A$ for all $A \in Y$. Hence, if $|Y| > 1$, multiple traversals of an FD-tree are caused. This can trigger overheads because FD-trees are linked-based data structures and stored in heap memory. We introduce *synergized FD induction* that processes several RHS attributes at once to minimize traversals. Given a non-FD $X \not\rightarrow Y$, no FD $X' \rightarrow Y'$ can be valid when $X' \subseteq X$ and $Y' \subseteq Y$ hold. Synergized FD induction augments $X' \rightarrow Y'$ to create all non-trivial candidates for valid FDs.

Example 3: Let $AC \rightarrow E$ and $AC \rightarrow BE$ be FDs. If $AC \not\rightarrow BDE$ is a non-FD on a given relation, the two FDs cannot be valid. $ABC \rightarrow E$ and $ACD \rightarrow E$ are all non-trivial candidates of valid FDs that result from augmenting

$AC \to E$. $ACD \to BE$, $ABC \to E$, and $ACE \to B$ are all non-trivial candidates of valid FDs that result from augmenting $AC \to BE$.

Algorithm 2 performs a synergized induction to update an extended FD-tree given any invalid FDs. It works as follows. Any FD $X' \to Y'$ where $X' \subseteq X$ and $Y' \subseteq Y$ cannot be satisfied by r given the non-FD $X \not\to Y$. In steps 20-24, the algorithm only traverses the paths which are subsets of X. During a traversal, a node's invalid RHS Y' is removed if the node is an FD-node and Y' intersects with Y (step 5-10). Although $X' \to Y'$ cannot form a valid FD in r, there are two ways to add another attribute $A \in R$ to X' such that $X'A$ is the LHS of some candidate FD. (1) We take A outside the union XY' (step 12). Then $X'A \to Y'$ is non-trivial and has a LHS that is not a subset of X. (2) We take A from Y' (step 16) and then $X'A \to Y' - \{A\}$ is also non-trivial and has a LHS that is not a subset of X.

Algorithm 2 Synergized Induction

```
 1: Input: A relation schema $R$, an invalid FD $X \not\to Y$, the root node $root$ of an FD-tree
 2: function induct($X, Y$)
 3:     induct_recursive($X = \{A_1, \ldots, A_n\}, Y, root$)
 4: function induct_recursive($X = \{A_i, \ldots, A_n\}, Y, current$)
 5:     if $current$ is an FD-node then
 6:         Let $X'$ be the path leading to $current$
 7:         Let $rhs(current)$ be the RHS of $current$
 8:         $removed = rhs(current) \cap Y$
 9:         Remove FD $X' \to removed$
10:         $rhs(current) = rhs(current) - Y$
11:         if $removed \neq \emptyset$ then
12:             for each $A' \in R - (X \cup removed)$ do
13:                 $Y' \subseteq removed$ is the minimal RHS of $X'A'$
14:                 Add FD $X'A' \to Y'$ if $|Y'| > 0$
15:             if $|removed| > 1$ then
16:                 for each $A' \in removed$ do
17:                     $removed' = removed - \{A'\}$
18:                     $Y' \subseteq removed'$ is the minimal RHS of $X'A'$
19:                     Add FD $X'A' \to Y'$ if $|Y'| > 0$
20:     for each $j \in [i, n]$ do
21:         if $A_j > \max\{A' \in R \mid current$ has a child of $A'\}$ then
22:             Return
23:         if there is a child $c$ of $current$ with $A_j$ then
24:             induct_recursive($\{A_j, \ldots, A_n\}, Y, c$)
```

E. Dynamic Data Manager

Our *dynamic data manager* (DDM) uses extended FD-trees to compute stripped partitions dynamically and efficiently. DDM pre-computes stripped partitions for all single attributes of a given relation schema. These are required during FD induction when completely new FD paths are added. Moreover, DDM maintains an array of dynamic stripped partitions. These are based on the paths ending at the current controlled level. Hence, they are refined as the controlled level increases. The ids of nodes in an extended FD-tree index stripped partitions, which are either pre-computed or dynamic.

Example 4: Figure 4 shows an extended FD-tree over $R = \{A, B, C, D, E, F\}$, with validation and controlled level 3. The DDM contains π_{ABD}, π_{ACD}, and π_{ACE}, indexed by $1, 2, 3$, respectively. Since node F resides at level 4, its id is that of its parent node E at the validation level. If the value of an id exceeds $|R|$, the id indexes a stripped partition in the DDM. Here, node E's id (9) corresponds to π_{ACE} as $9 - |R| = 3$. The id (7) of node B is inconsistent because it corresponds to π_{ABD} as $7 - |R| = 1$. So, if FD $ABC \to E$ is added to the tree now, node C becomes an FD-node with id 3 (default) instead of 7.

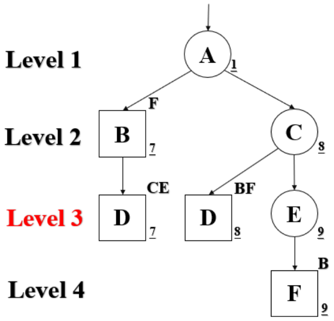

Fig. 4. Example of id assignments

The main task of a DDM is to update dynamic stripped partitions and assign nodes with consistent ids in an FD-tree. Given the nodes at the controlled level, DDM uses the underlying paths to compute new stripped partitions. Algorithm 3 updates a DDM from controlled level i to j. For each node at level j, the algorithm finds the path to the node, and refines the node's stripped partition in array $\mathcal{A}$ using the new attributes in the node's path. The refined partition is appended to the new array $\mathcal{A}'$ (step 10). The new id of the node is the node's position in $\mathcal{A}'$ plus $|R|$ (step 13). Then, the new id is copied to the node's descendants, ensuring consistency in the extended FD-tree. Any new node that is introduced at the controlled level is not processed by Algorithm 3. Hence, no corresponding stripped partition exists for such a node. Here, the order of its attribute is used as the id of this node (default id).

Algorithm 3 Update DDM

```
 1: Input: A relation $r$ over relation schema $R$, an array $\mathcal{A}$ of stripped partitions from level $i$, the set $L$ of nodes from level $j > i$
 2: Output: A new array of stripped partitions at level $j$
 3: Let $\mathcal{A}'$ be an array of size $|L|$
 4: $i = 1$
 5: for each node $n \in L$ do
 6:     Let $X$ be the path leading to $n$
 7:     Let $\pi_{X'} = \pi_A$ where $A$ is the attribute of $n$
 8:     if $n.id > |R|$ then
 9:         $\pi_{X'} = \mathcal{A}[n.id - |R|]$
10:         Let $\mathcal{A}'[i] = \pi_{X'}$
11:         for each $B \in X - X'$ do
12:             $\mathcal{A}'[i] = refine(r, \mathcal{A}'[i], B)$
13:         $n.id = i + |R|$
14:         $i = i + 1$
15:         Copy id of node $n$ to its descendants
16: Return $\mathcal{A}'$
```

F. Validation and Refinement

For computing stripped partitions efficiently, we use a *domain independent indexing scheme* (DIIS). This compresses an input relation into a two-dimensional array. Given relation r over R, a DIIS for $A \in R$ is a bijective mapping of the active domain $adom_r(A)$ to $\{1, \ldots, |adom_r(A)|\}$. This is easy to compute, convenient for generating stripped partitions, and validating FDs. Note that the active domain of A with respect to r is just the finite set $\{t(A) \mid t \in r\}$.

Validation of an FD $X \to Y$ returns a set of non-FDs that cover all RHS attributes in Y that are not functionally dependent on the given LHS X. Classical FD validation, see Section IV-C, typically requires multiple runs to find valid mappings between LHS and RHS values [9], [16]. These mappings are generated incrementally from the static stripped partitions of the singleton attributes. Consequently, LHS values are typically computed redundantly. We now discuss how DDMs improve classical FD validation. With the aim to avoid redundant computation, we propose Algorithm 4 to validate FDs using dynamic stripped partitions.

Algorithm 4 Validation

```
 1: Input:  Relation $r$ over schema $R$, FD $X \to Y$, stripped partition $\pi_{X'}$
    where $X' \subseteq X$
 2: Output:  Non-FDs that invalidate $X \to Y'$ where $Y' \subseteq Y$
 3: $non\_fds = \emptyset$
 4: $valid\_rhs = Y$
 5: for each $S \in \pi_{X'}$ do
 6:     Let $\pi = \{S\}$
 7:     for each $A \in X - X'$ do
 8:         $\pi = refine(r, \pi, A)$
 9:     for each $\{t_0, \ldots, t_n\} \in \pi$ do
10:         for each $i \in [1, n]$ do
11:             $invalid\_rhs = \{A \in valid\_rhs \mid t_i(A) \neq t_0(A)\}$
12:             $valid\_rhs = valid\_rhs - invalid\_rhs$
13:             if $invalid\_rhs \neq \emptyset$ then
14:                 $Z = ag(t_0, t_i)$
15:                 $non\_fds = non\_fds \cup \{Z \not\to R - Z\}$
16:             if $valid\_rhs = \emptyset$ then
17:                 Return  $non\_fds$
18: Return  $non\_fds$
```

Given relation r over R and an FD $X \to Y$, a DDM may hold the stripped partition $\pi_{X'}$ instead of π_X where $X' \subseteq X$. In general, it is inefficient to generate π_X from $\pi_{X'}$ when validating $X \to Y$. Indeed, the computation of π_X must scan every tuple in $\pi_{X'}$. This wastes resources if $X \to Y$ is not satisfied by r. In fact, the overhead is substantial if there are many tuples in $\pi_{X'}$. Hence, Algorithm 4 only *refines* one set in a stripped partition at a time, using Algorithm 5. It can thus terminate quickly if a given FD is not satisfied. Algorithm 4 reduces redundant computations of X-values since $\pi_{X'}$ is known. Only $(X - X')$-values are processed (step 7-8) if there are valid RHSs. That is, in steps 16 and 17 the algorithm returns a set of non-FDs if there is no valid RHS for X.

Algorithm 5 refines stripped partitions. It stores the new equivalence classes in an array. Our data compression scheme eases the allocation of tuples to their new classes in the array, since the index of each class corresponds to some domain value. The algorithm refines every class in the input partition one attribute at a time. When a tuple is allocated to an empty set, we store the set's position by retrieving the tuple's projected value on the current attribute (step 8). Recording these positions saves the search for non-empty sets.

Algorithm 5 Refinement

```
 1: Input:  A relation $r$ over relation schema $R$, a subset $\pi'$ of the stripped
    partition $\pi_X$, an attribute $A \in R$
 2: Output:  A subset of $\pi_{XA}$
 3: Let $sets\_array = \{\emptyset, \ldots, \emptyset\}$ where $|sets\_array| = |r|$
 4: $result = \emptyset, ids = \emptyset$
 5: for each $S \in \pi$ do
 6:     for each $i \in S$ do
 7:         if $sets\_array[r[i][A]]$ is empty then
 8:             $ids = ids \cup \{r[i][A]\}$
 9:         $sets\_array[r[i][A]] = sets\_array[r[i][A]] \cup \{i\}$
10:     for each $id \in ids$ do
11:         if $|sets\_array[id]| \geq 2$ then
12:             $result = result \cup sets\_array[id]$
13:         $sets\_array[id] = \emptyset$
14:     $ids = \emptyset$
15: Return  $result$
```

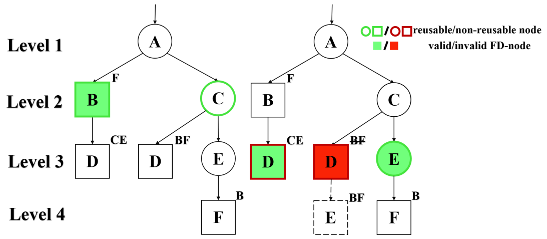

Fig. 5. An efficiency-inefficiency ratio calculation

G. When to Update Stripped Partitions

For enabling a DDM to decide if the stripped partitions need updating, we define the *efficiency* and *inefficiency* of a validation level. At each level, the total number of FDs is the sum of the RHS sizes over the nodes at the current level (also see line 13 of Algorithm 6), before FD induction takes place (line 20). The number of valid FDs is counted in the same way (line 13) but after FD induction. The *efficiency* of the validation level is the ratio of valid FDs over all FDs (including invalid FDs) at the given level. Only actually valid FDs require a scan of all the tuples in a stripped partition. If efficiency is low, more nodes in higher levels may represent invalid FDs. Generating stripped partitions for more invalid FDs is inefficient. The *inefficiency* of a validation level is the proportion of *reusable nodes* over all the FDs that reside in higher levels. Here, a node is *reusable* if it is not a leaf. If inefficiency is high, most FDs in higher levels cannot share stripped partitions. Hence, it is more efficient to directly validate these FDs when their FD node is reached. In summary, updates of stripped partitions are more beneficial when the efficiency at the current validation level is high and the inefficiency is low. Hence, we define the efficiency-inefficiency ratio at the current validation level as the ratio of its efficiency over its inefficiency. Experiments that determine the actual ratio used by DHyFD are in Section V.

Example 5: Figure 5 shows calculations of the efficiency-inefficiency ratio. After processing level 2 (left tree), node B represents the valid FD $B \to F$. The efficiency of level 2 (left tree) is $1/1$ since there is only one FD-node at level 2. Both nodes B and C are reusable. They lead to 5 FDs: $ABD \to C$, $ABD \to E$, $ACD \to B$, $ACD \to F$ and $ACEF \to B$. So, the inefficiency is $2/5$ and the ratio is 2.5. After processing level 3 (right tree), the FD $ABD \to CE$ is valid but the node D in path ABD is not reusable. The other FD $ACD \to BF$ at level 3 (right tree) is not valid. Hence, the efficiency is $1/2$. By an induction on invalid FD $ACD \not\to BF$, a new path $ACDE$ is constructed. The reusable nodes at level 3 (node D and E) lead to 3 FDs: $ACDE \to B$, $ACDE \to F$, and $ACEF \to B$. So, the inefficiency is $2/3$ and the ratio is 0.75.

H. DHyFD algorithm

Algorithm 6 implements DHyFD. It starts by initializing the DDM and extended FD-tree (lines 3-4). The controlled and validation levels are tracked by the variables cl and vl, respectively. DHyFD performs the sorted neighborhood pair selection sampling only once at the beginning to extract a diverse selection of non-FDs (line 5). Re-sampling would only cause computational overheads. For example, sampling with an input of $1,000$ tuples already compares more than 1 million tuple pairs according to Section V. In lines 14-18, the DDM of DHyFD finds a stripped partition (lines 15-16), and validates the corresponding FDs level by level with Algorithm 4 (line 18). Subsequently, any identified violations of FDs are used to update the extended FD-tree with Algorithm 2 (line 20). Lines 21-25 calculate the efficiency-inefficiency ratio to determine if the DDM should update the stripped partitions (line 27). The iterations continue until no candidate FD is left (line 11).

Note that sorting non-FDs (in steps 7 and 19) helps eliminate redundant inductions faster than using non-redundant non-FDs, as demonstrated in Section V. This is explained as follows. Let T be an FD-tree. Suppose an update is processed by a non-FD $X \not\rightarrow Y$. Now consider another update by a non-FD $X' \not\rightarrow A$ where $X' \subset X$ and $A' \in Y$. Here, $X' \not\rightarrow A$ is redundant with respect to $X \not\rightarrow A$. Hence, no new FDs will be induced if $X \not\rightarrow A$ is applied first. In addition, if $X' \not\rightarrow A$ is applied first, then some of the new FDs can still be eliminated by $X \not\rightarrow A$, which causes redundant inductions.

Algorithm 6

```
1:  Input:  A relation $r$ over relation schema $R$
2:  Output:  The left-reduced cover of the FDs satisfied by $r$
3:  Initialize DDM $M$ with the stripped partitions of all $A \in R$
4:  Let $tree$ be an extended FD-tree for the single FD $\emptyset \rightarrow R$
5:  $violations$ is the set of non-FDs extracted by sorted neighborhood pair
    selection sampling
6:  $violations = violations \cup validate(root, \{r\})$
7:  Sort the non-FDs in descending order by the sizes of their LHSs
8:  for each $X \not\rightarrow R - X \in violations$ do $tree.induct(X, R - X)$    ▷
    Algorithm 2
9:  Let $candidates$ be the set of nodes at level 1 of $tree$
10: Let $cl = 1, vl = 1, num\_fds = 0$
11: while $candidates \neq \emptyset$ do
12:     $violations = \emptyset$
13:     $total = \sum_{n \in candidates} |rhs(n)|$
14:     for each $node \in candidates$ do
15:         if $node.id \leq |R|$ then
16:             $node.id = arg_A \min\{||\pi_A|| \mid A \in R\}$
17:         Let $\pi$ be the stripped partition assigned to $node$ by $M$
18:         $violations = violations \cup validate(node, \pi)$    ▷ Algorithm 4
19:     Sort $violations$ in descending order
20:     for all $X \not\rightarrow R - X \in violations$ do $tree.induct(X, R - X)$    ▷
    Algorithm 2
21:     $reusables = \{n \in candidates \mid n$ is not a leaf$\}$
22:     $num\_new\_fds = \sum_{n \in candidates} |rhs(n)|$
23:     $num\_fds = num\_fds + num\_new\_fds$
24:     efficiency $= num\_new\_fds / total$
25:     inefficiency $= |reusables| / (|tree| - num\_fds)$
26:     if $vl > 1$ and efficiency/inefficiency $> 3.0$ then
27:         $cl = vl$, Update $M$ with $reusables$    ▷ Algorithm 3
28:     $vl = vl + 1$
29:     Let $candidates$ be the set of nodes of $tree$ at level $vl$
30: Return  $\{FDs\ in\ tree\}$
```

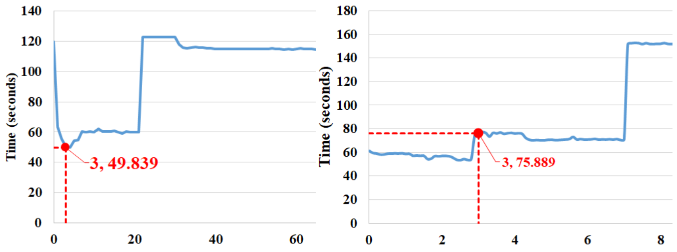

Fig. 6. Time of FD discovery on $weather$ (left) and $uniprot\_512kr\_30c$ (right) with different efficiency-inefficiency ratios

V. Experiments

We analyze our new FD discovery algorithm over real-world benchmarks, including run time, memory use, row- and column-scalability for different interpretations of missing values. We also analyze the savings in output sizes made by canonical over left-reduced covers.

We implemented DHyFD in Visual C++. For comparison we also implemented state-of-the-art algorithms (TANE [9], FDEP [6], HyFD [16]). These algorithms present benchmark performances on data sets with large numbers of rows, or columns, or both [15], [16]. We ran our experiments on an Intel Xeon 3.6 GHz, 256GB RAM, Windows 10 Dell workstation. We used real-world data from the UCI machine learning data repository[1] and previous research [16]. The data sets are available for download[2].

A. Parameter Tuning

DHyFD generates stripped partitions dynamically. They are refined at validation levels with high efficiency and low inefficiency. However, what actual efficiency-inefficiency ratio minimizes the running time of DHyFD? The left of Figure 6 shows the time used to discover FDs on the $weather$ data set for different ratios. It contains 18 columns and more than $260,000$ rows. DHyFD performed best when the ratio was around 3. For that ratio it discovered the left-reduced cover of 68 FDs within 50 seconds. While the best ratio depends on the data, DHyFD performed well on each data set with ratio 3. As example, the right of Figure 6 shows that the best ratio on $uniprot$ with $512,001$ rows and 30 columns is 2.5. Nevertheless, the performance at ratio 3 is satisfying.

B. Performance on Real-world Data

We conduct experiments that discover FDs for a range of real-world data using DHyFD and other algorithms. In Section IV-C we described how FDEP can be improved over its original proposal with classical FD induction over classical FD-trees [6]. Accordingly, we have two implementations FDEP1 and FDEP2. Both implement the new synergized induction on extended FD-trees, but FDEP2 sorts all of the non-FDs of a relation the same way FDEP does, while FDEP1 computes a non-redundant cover of non-FDs before induction. We can thus provide fair and comprehensive performance reports on these algorithms. The row- and column-based algorithms cannot terminate on all of the data sets within

[1] https://archive.ics.uci.edu/ml/

[2] http://bit.do/erhUF

TABLE II. RUNNING TIME (SECONDS) UNDER $null = null$ SEMANTICS, AND MEMORY USAGE (IN MB)

Data set	#R	#C	#FD	TANE	FDEP	FDEP1	FDEP2	HyFD	DHyFD	Old best	HyFD	DHyFD
iris	150	5	4	0.001	0.002	0.002	0.002	**0.0001**	**0.0001**	0.1	0.67	0.64
balance	625	5	1	0.002	0.031	0.04	0.024	0.001	**0.0001**	0.1	0.7	0.69
chess	28056	7	1	0.154	50.192	94.13	47.942	**0.017**	**0.017**	0.2	12	12
abalone	4177	9	137	0.029	0.785	2.794	1.191	0.03	**0.017**	0.2	3	3
nursery	12960	9	1	0.241	23.415	26.205	13.684	0.011	**0.01**	0.5	7	5
breast	699	11	46	0.044	0.127	0.09	0.048	0.02	**0.009**	0.2	1	1
bridges	108	13	142	0.03	0.011	0.007	0.005	0.004	**0.003**	0.1	0.7	0.73
echo	132	13	527	0.01	0.007	0.009	0.006	0.003	**0.002**	0.1	0.69	0.76
adult	48842	14	78	22.491	311.365	278.591	129.174	0.279	**0.215**	1.1	14	14
letter	20000	17	61	208.67	73.718	130.414	47.4	6.96	**2.035**	3.4	33	29
ncvoter	1000	19	758	0.444	0.384	0.551	0.216	0.046	**0.029**	0.4	3	3
hepatitis	155	20	8250	9.851	0.532	0.158	**0.153**	0.174	0.189	0.6	9	14
horse	368	29	128727	130.527	4.985	4.607	3.334	4.728	**2.595**	7.1	123	268
plista	1000	63	178152	TL	35.985	17.945	**13.894**	19.203	15.403	21.7	389	2048
flight	1000	109	982631	TL	16.134	21.28	**9.04**	37.064	9.934	53.4	841	2048
fd-reduce	250000	30	89571	**8.084**	TL	TL	TL	201.005	158.94	41.1	170	181
weather	262920	18	918	TL	TL	TL	TL	332.734	**49.839**	N/A	140	1024
diabetic_30c	101766	30	40195	TL	TL	TL	TL	2864.84	**847.582**	N/A	2253	4301
PDBX	17305799	13	68	TL	TL	TL	TL	**95.893**	100.906	240	6348.8	6451.2
lineitem	6001215	16	3984	TL	TL	TL	TL	1352.87	**1047.44**	2340	2662.4	27648
uniprot_512kr_30c	512000	30	3703	TL	TL	TL	TL	184.573	**75.442**	N/A	3481.6	4608

competitive time. Hence, we set the time limit (TL) of the experiments to 1 hour. Note that HyFD also implements our synergized FD induction. For our experiments, we show the number of rows (#R), columns (#C), FDs (#FD) in a left-reduced cover, incomplete rows (#IR), incomplete columns (#IC), missing values (#⊥), and the running time in seconds.

Results. Table II summarizes the performance of the FD discovery algorithms on the real-world data sets. In addition, we include memory usage of HyFD and DHyFD. Overall, there are considerable improvements over the previously best known times. Expectedly, the hybrid algorithms outperform the row- and column-based algorithms on data with sufficiently many rows and columns. Note that $fd\_reduced$ is the only synthetic data set and an exception. DHyFD performs better than HyFD and FDEP2 on small data sets with less than $10,000$ rows or 50 columns. However, FDEP2 performs extremely well on data with few rows and many columns, such as $plista$ and $flight$. The performance of FDEP2 is mainly influenced by the non-redundant cover of non-FDs. For data with few rows (e.g. 1000) and few columns, FDEP2 will perform worse than the hybrid algorithms because FDEP2 computes all the non-FDs but most of them only create overhead (a redundant non-FD is only used to traverse an FD-tree but cannot induce new FDs). If the data set has more columns (e.g. $plista$, $flight$), then the number of useless non-FDs is reduced. Accordingly, the inductions of FDEP2 become more effective. Nevertheless, FDEP2 performs always better than FDEP1, which means the computation of a non-redundant cover of non-FDs does not yield good performance in practice. Comparing FDEP2 with FDEP demonstrates the advantages of synergized induction and extended FD-trees over classical induction and FD-trees. Note that in the case of *abalone* the classical method performs better. This is mainly because excessive labeling in an FD-tree sometimes does help prune a search space. However, such situation is rare and less significant. In general, excessive labeling creates huge overheads, for example, FDEP takes almost 200 more seconds than FDEP2 to perform the same FD induction over *adult*. For these reasons, we will only discuss FDEP2 in what follows, and simply refer to it as FDEP from here on. TANE only performs well on $fd\_reduced$ because this data is particularly suitable for TANE. TANE traverses the attribute lattice from bottom to top and all the LHSs of the FDs discovered in $fd\_reduced$ only have 3 attributes. As a result, the FDs with short LHSs will be discovered quickly. As our implementation of HyFD uses synergized induction and performs better than the best known bounds, it is further evidence for the performance gains that synergized induction facilitates.

DHyFD gains performance over HyFD by leveraging more memory whenever new FD discoveries are likely. We report the use of memory by the various algorithms in the technical report [18]. In brief, TANE uses huge memory even on small data like $horse$ but does not gain performance. Meanwhile, DHyFD only uses more memory when it is rational, as measured by our efficiency-inefficiency ratio. On $PDBX$, only an extremely small number of FDs is satisfied by a large number of tuples, so non-FDs that derive true FDs can be sampled easily. DHyFD shows similar efficient time and memory use as HyFD, which proves that the efficiency-inefficiency ratio also suggests correctly to DHyFD that more memory usage will not improve running time further. On other data, DHyFD outperforms HyFD by better use of memory for FD validation and non-FD extraction. In fact, DHyFD beats HyFD in many cases, such as data sets like $weather$, $lineitem$ and $uniprot\_512kr\_30c$ that contain only a small number of FDs that are randomly spread over the entire FD-tree, and data sets like $diabetic$ that are highly dimensional and contain a large number of FDs. Lastly, compared to TANE, DHyFD uses much less memory.

Additional experiments show how DHyFD leverages the tradeoff between performance and memory. We quantify on some benchmarks the performance gain and additional memory use by DHyFD over HyFD. The *performance increase rate* (PIR) is the difference in run time of HyFD and DHyFD over that of HyFD, and the *memory increase rate* (MIR) is the difference in memory use of DHyFD and HyFD over that of DHyFD on a data set. Figure 7 illustrates how the numbers of rows and columns in a data set affect the memory use by HyFD

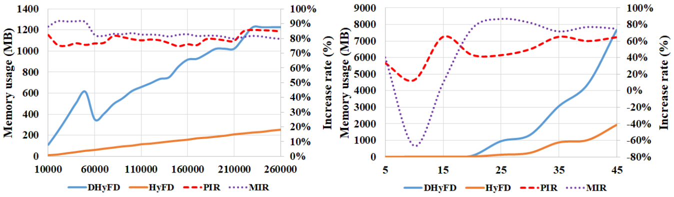

Fig. 7. Memory used in FD discovery on *weather* fragments with varying numbers of rows (left) and on *diabetic* fragments with different numbers of columns

and DHyFD. All round, additional memory use results in solid performance gains. With more rows or columns, the PIRs and MIRs are typically getting closer. Future work may show how memory use can maximize performance, or for which efficiency-inefficiency ratio memory use is most effective.

Null semantics. As in practice, many benchmarks contain missing values. Different interpretations of missing values cause differences in discovery algorithms and performance. We report results on the most common semantics that treats missing values just like any other value ($null = null$). Results on the semantics where each missing value is treated as a unique value ($null \neq null$) are reported in the technical report [18]. In brief, $null \neq null$ tends to exhibit more FDs and, hence, longer runtime, especially on larger data sets. The performance of the algorithms is similar to $null = null$. However, FDEP is fastest on some smaller data sets for $null \neq null$: bridges, hepatitis and horse. For weather, diabetic and PDBX, the ranking is the same: DHyFD is again the fastest by far on the former two, while HyFD is marginally faster on PDBX. On uniprot_512kr_30c, HyFD is marginally faster than DHyFD under $null \neq null$.

In summary, DHyFD improves state-of-the-art. It performs well on data with more columns and rows, making effective use of conservatively more main memory to discover FDs more quickly. Specialized algorithms outperform hybrids on data sets for which they are designed. Our optimization of FDEP is effective on data sets with few rows and many columns.

C. Scalability

We explore the row- and column-scalability of TANE, FDEP, and both hybrid algorithms. Qualitative and quantitative experiments show how the row and column numbers impact on the performance of these algorithms. The qualitative experiments measure the performance of all algorithms on fragments of *weather* and *diabetic_30c* with varying numbers of rows and columns. The quantitative experiments show which algorithm performs best on which fragment.

Quantitative Experiments. Each mark in Figure 8 represents a data fragment where the numbers of rows and columns are the values at its horizontal and vertical axes, respectively. The color of a mark denotes the fastest algorithm. FDEP wins consistently on the left of both charts. As columns increase, FDEP gains more advantage. For example, in the right of Figure 8, FDEP performs worse on data with $10,000$ rows and 20 columns, but better on data with $10,000$ rows and 30 columns. So, FDEP scales well on columns but poorly on rows. DHyFD wins when more rows and columns are present. There are only few fragments where HyFD performs better than DHyFD, and the differences are small in these cases. This is mainly due to the random behavior of the sampling method in HyFD. TANE performs poorly as it is targeted at data with FDs that have smaller LHSs, which happens only rarely on real-world data.

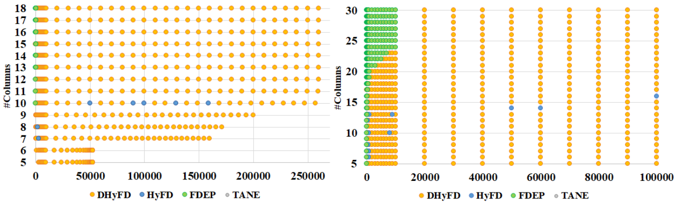

Fig. 8. Best performers on *weather* and *diabetic_30c*

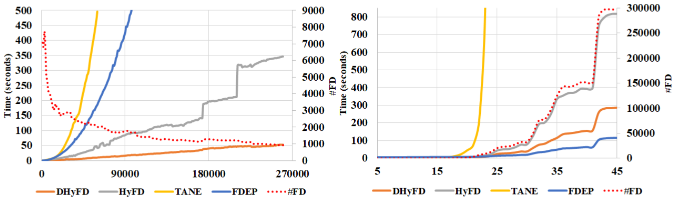

Fig. 9. Row scalability of *weather* (left) and column scalability on *diabetic_10000r* (right)

Qualitative Experiments. Figure 9 shows how much one algorithm performs better than the others. The left of Figure 9 shows the row scalability of the benchmark algorithms and DHyFD. We ran FD discovery on *weather* by selecting $1,000$ - $260,000$ rows with increments of $1,000$ rows. The time of TANE and FDEP dramatically increases for many rows. When a data set exceeds approximately $10,000$ rows, TANE and FDEP are not feasible. HyFD suffered a significant performance loss at $211,000$ rows. Instead, DHyFD shows smooth row scalability. The right of Figure 9 shows the column scalability on *diabetic*. For comparing the performance of all algorithms we only selected $10,000$ rows. TANE performs well when there are less than 15 columns. If there are more than 41 columns, the time of HyFD increases significantly. As the second vertical axis shows on the right of Figure 9, this change is caused by the doubling of valid FDs. That means, HyFD requires much more time to validate FDs. In contrast, DHyFD handles the situation more smoothly. This demonstrates a huge improvement achieved by the new validation method and the DDM. Lastly, FDEP and DHyFD perform similarly on data sets with few rows. With more columns the performance of FDEP improves over that of DHyFD. Indeed, FDEP saves substantial FD validation time when more valid FDs are exhibited.

D. Covers of FD Profiles

Previous work has represented the output of FD discovery algorithms as left-reduced covers. Table III contains the results of applying standard algorithms for the computation of canonical covers from left-reduced ones to the benchmark data [11]. The table shows the number of FDs in a left-reduced cover (|L-r|), the total number of attributes in a left-reduced cover (||L-r||), the number of FDs in a canonical cover (|Can|), the total number of attributes in a canonical cover (||Can||), the percentage of the ratios |Can|/|L-r| (%Size) and ||Can||/||L-r||

TABLE III. PROPERTIES OF LEFT-REDUCED & CANONICAL COVERS

Data set	|L-r|	||L-r||	|Can|	||Can||	%S	%C	Time
iris	4	16	4	16	100	100	0
balance	1	5	1	5	100	100	0
chess	1	7	1	7	100	100	0
abalone	137	715	41	217	30	30	0.001
nursery	1	9	1	9	100	100	0
breast	46	214	39	184	85	86	0
bridges	142	669	65	337	46	50	0.002
echo	527	2322	93	392	18	17	0.012
adult	78	495	42	267	54	54	0.001
letter	61	786	61	786	100	100	0
ncvoter 1001r_19c	758	3754	185	927	24	25	0.023
hepatitis	8250	54821	2204	14718	27	27	0.927
horse	128727	1045762	34053	267385	26	26	81.85
fd-reduce	89571	358238	1550	6203	2	2	79.46
plista	178152	1397038	22680	166963	13	12	276.35
flight	982631	6106725	83496	520623	8	9	19996
weather	918	7219	514	4061	56	56	0.015
diabetic	40195	464871	32689	378546	81	81	9.14
PDBX	68	157	19	58	28	37	0
lineitem	3984	24927	679	4241	17	17	0.6
uniprot 512kr_30c	3703	23530	1677	11179	45	48	0.104

TABLE IV. DATA REDUNDANCY IN NUMBERS AND PERCENTAGES

data set	#values	#red	%red	#red+0	%red+0
abalone	37,593	67	0.18		
adult	683,788	75718	11.07		
balance	3,125	0	0		
chess	196,392	0	0		
fd_reduced	7,500,000	2,500,000	33.33		
iris	750	31	4.13		
letter	340,000	6,809	2		
lineitem	96,019,440	11,407,131	11.88		
nursery	116,640	0	0		
breast	7,689	706	9.18	706	9.18
bridges	1,404	388	28.13	395	28.13
china	4,732,560	1,971,104	41.65	2,022,994	42.75
diabetic	3,052,980	420,607	13.78	474,460	15.54
echo	1,716	375	21.85	416	24.24
flight	109,000	48,297	44.31	100,233	91.96
hepatitis	3,100	1,588	51.23	1,629	52.55
horse	10,304	3,703	35.94	4,854	47.11
ncvoter	19,000	2,886	15.19	3,659	19.26
plista	63,000	27,024	42.9	50,047	79.44
uniprot	15,360,030	1,288,502	8.39	2,556,639	16.64
pdbx	224,975,387	131,743,942	58.56	132,441,479	58.87

(%Card), and the time in seconds to compute a canonical from the left-reduced cover (Time). On average, the canonical covers have about 50% savings in both the numbers of FDs and total numbers of attributes, about 25% savings on smaller data sets (the first ten), and about 70% savings on bigger data sets (the remaining eleven). This makes the outputs of FD discovery algorithms not just clearer, because redundancy is avoided, but also easier to comprehend and process. Our results also demonstrate potential for future improvements of discovery algorithms. The gap between left-reduced and canonical covers shows that current algorithms do not prune many redundant FDs. However, efficient pruning based on the transitivity rule of FDs ($X \to Y$ and $Y \to Z$ imply $X \to Z$) is challenging.

VI. RANKING FDS ACCORDING TO THEIR RELEVANCE

We provide a quantitative and qualitative analysis of applying our relevance measure of FDs to the canonical covers of our benchmark data.

A. *Quantitative analysis*

Table IV lists for each data set the number #values of data occurrences, the number #red of those that are redundant excluding null, their percentage %red, the number #red+0 of those that are redundant including null, and their percentage %red+0. As stressed before, we let the data speak for itself and do not judge the FDs in the covers on their meaningfulness.

The table provides the first insight ever on the data redundancy exhibited by the benchmarks, and clearly shows the significance of the measure by sheer volume. Furthermore, the impact of nulls can be large.

Figure 10 analyzes for our bigger incomplete data sets how many FDs cause how much data redundancy, and the time in seconds to compute all redundant occurrences given the data set and canonical cover. Each x-value is a given number of redundant occurrences an FD can exhibit to be listed under that x-value (and it must exhibit more than the previous x-value). The x-values always represent 0, 2.5%, 5%, 10%, 15%, 20%, 40%, 60%, 80%, and 100% of the maximum number of redundant occurrences caused by any FD exhibited in the data set. The charts show how the relevance of FDs ranks the output of FD discovery algorithms. Many FDs are ranked within a low percentile of redundant data values (more than 0 and less than 5% of the maximum). Data stewards may first focus on the other FDs, including those that are ranked higher, and those that do not cause any data redundancy because the latter may indicate keys. FDs in the low percentile need to be looked at carefully by domain experts: they could indicate the presence of dirty data or represent FDs that hold only accidentally. For such FDs it is also useful to analyze how many of the redundant data values are null marker occurrences.

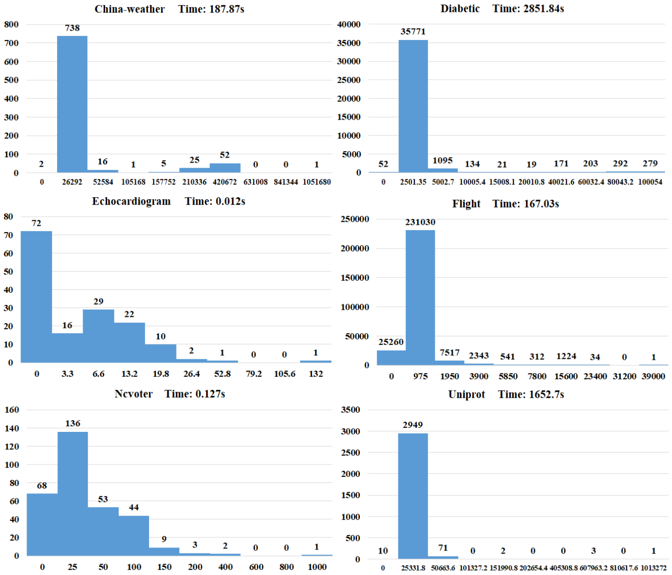

Fig. 10. Number of FDs in canonical covers (y-axis) that cause not more than the given number of redundant occurrences (x-axis), plus time (s) taken to compute all redundant occurrences in the data set using the canonical cover

B. *Some qualitative analysis*

Let us consider *ncvoter* for some qualitative illustrations. The introduction listed some FDs with many, very

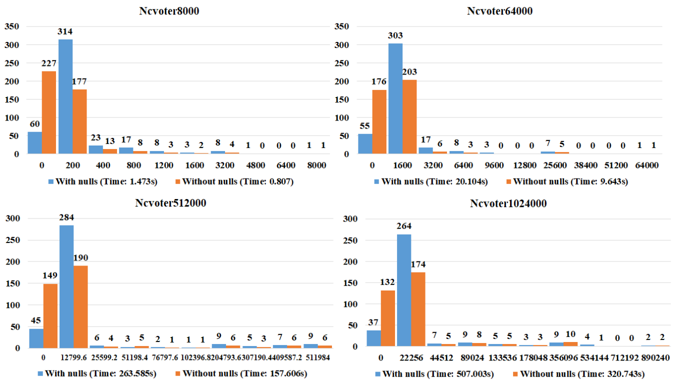

Fig. 11. Comparison of FD numbers for given numbers of redundancies with (blue) and without any nulls on LHSs and RHS (orange) across increasing fragments of *ncvoter*, plus times to determine them

few, and no redundant occurrences. In particular, σ_3 is an example FD that may hold only accidentally because most redundant occurrences are nulls (59 out of 61). In contrast, σ_4 only has 2 redundant occurrences because of some dirty data. Another example is *first_name, last_name* → *name_prefix, name_suffix, gender* which triggers 60 redundant occurrences, but 40 of those are null. The remaining 20 are all caused by *first_name, last_name* → *gender*, which appears to be a reasonable constraint. Our rankings provide data stewards with different ways to analyze the relevance of FDs for applications. One view is to fix a column of interest, and see which minimal LHSs cause how many redundant occurrences in that column. For example, some minimal LHSs that functionally determine *city* in *ncvoter* and cause some redundant occurrences in *city* are as follows.

minimal LHSs for *city*	#red	#red-0
last_name, zip_code	158	158
middle_name, zip_code	231	114
street_address	81	81
first_name, zip_code	71	71
age, gender, zip_code, full_phone_num	173	16
voter_id	2	2
last_name, age, full_phone_num, register_date	4	0
first_name, last_name, full_phone_num, download_month	2	0

Here, #red lists the number of any redundant occurrences while #red-0 lists the number of redundant occurrences that do not involve any nulls on neither LHS attributes nor *city*. This places stronger relevance on the FDs with boldly marked LHSs, which also appear to represent more reasonable FDs. Indeed, redundancies caused by FDs that do not involve any nulls on LHS and RHS attributes are strong evidence of the FD pattern, and larger numbers of such redundancies are testimony to the stronger relevance of the FD. Figure 11 compares the numbers of FDs that cause up to a given number of redundancies with (blue) and without (organge) nulls. Over four different fragments of *ncvoter* with 8k, 16k, 512k, and 1024k tuples, it is interesting to see how these numbers remain stable, and how many FDs with small redundancies are shifted to FDs without redundancies when nulls are excluded from occurrences on LHS and RHS attributes.

VII. Conclusion and Future Work

DHyFD is a new algorithm for the discovery of FDs. Using a novel hybridization strategy and the dynamic computation of stripped partitions, DHyFD outperforms the state-of-the-art for runtime, row-, and column-scalability, by effective use of more memory. Hence, DHyFD can handle larger inputs. Canonical covers significantly decrease the outputs of FD discovery algorithms by an average 50%. FDs can be ranked by the number of redundant data values they cause. The ranking guides data stewards towards FDs of higher relevance. Cover computation and FD ranking open up new research inquiries.

References

[1] Z. Abedjan, L. Golab, and F. Naumann, "Profiling relational data: a survey," *VLDB J.*, vol. 24, no. 4, pp. 557–581, 2015.

[2] Z. Abedjan, P. Schulze, and F. Naumann, "Dfd: Efficient functional dependency discovery," in *CIKM*. ACM, 2014, pp. 949–958.

[3] L. Berti-Équille, H. Harmouch, F. Naumann, N. Novelli, and S. Thirumuruganathan, "Discovery of genuine functional dependencies from relational data with missing values," *PVLDB*, vol. 11, no. 8, pp. 880–892, 2018.

[4] T. Bläsius, T. Friedrich, and M. Schirneck, "The parameterized complexity of dependency detection in relational databases," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 63, 2017.

[5] S. Davies and S. Russell, "NP-completeness of searches for smallest possible feature sets," in *AAAI*, 1994.

[6] P. A. Flach and I. Savnik, "Database dependency discovery: a machine learning approach," *AI communications*, vol. 12, no. 3, pp. 139–160, 1999.

[7] C. Giannella and C. Wyss, "Finding minimal keys in a relation instance," http://www.cs.indiana.edu/∼cgiannel/keys.ps, 1999.

[8] M. A. Hernández and S. J. Stolfo, "Real-world data is dirty: Data cleansing and the merge/purge problem," *Data mining and knowledge discovery*, vol. 2, no. 1, pp. 9–37, 1998.

[9] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen, "Tane: An efficient algorithm for discovering functional and approximate dependencies," *The computer journal*, vol. 42, no. 2, pp. 100–111, 1999.

[10] S. Lopes, J.-M. Petit, and L. Lakhal, "Efficient discovery of functional dependencies and Armstrong relations," in *EDBT*. Springer, 2000, pp. 350–364.

[11] D. Maier, *The Theory of Relational Databases*. Computer Science Press, 1983.

[12] H. Mannila and K. Räihä, "Dependency inference," in *VLDB*, 1987, pp. 155–158.

[13] ——, "On the complexity of inferring functional dependencies," *Discrete Applied Mathematics*, vol. 40, no. 2, pp. 237–243, 1992.

[14] N. Novelli and R. Cicchetti, "Fun: An efficient algorithm for mining functional and embedded dependencies," in *ICDT*. Springer, 2001, pp. 189–203.

[15] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann, "Functional dependency discovery: An experimental evaluation of seven algorithms," *PVLDB*, vol. 8, no. 10, pp. 1082–1093, 2015.

[16] T. Papenbrock and F. Naumann, "A hybrid approach to functional dependency discovery," in *SIGMOD*. ACM, 2016, pp. 821–833.

[17] M. W. Vincent, "Semantic foundations of 4NF in relational database design," *Acta Inf.*, vol. 36, no. 3, pp. 173–213, 1999.

[18] Z. Wei and S. Link, "Discovery and ranking of functional dependencies," The University of Auckland, New Zealand, Technical Report CDMTCS-531, 2019.

[19] C. Wyss, C. Giannella, and E. Robertson, "Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies," in *DaWaK*. Springer, 2001, pp. 101–110.

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