A Hybrid Approach to Functional Dependency Discovery

Thorsten Papenbrock
Hasso Plattner Institute (HPI)
14482 Potsdam, Germany
thorsten.papenbrock@hpi.de

Felix Naumann
Hasso Plattner Institute (HPI)
14482 Potsdam, Germany
felix.naumann@hpi.de

ABSTRACT

Functional dependencies are structural metadata that can be used for schema normalization, data integration, data cleansing, and many other data management tasks. Despite their importance, the functional dependencies of a specific dataset are usually unknown and almost impossible to discover manually. For this reason, database research has proposed various algorithms for functional dependency discovery. None, however, are able to process datasets of typical real-world size, e.g., datasets with more than 50 attributes and a million records.

We present a hybrid discovery algorithm called HyFD, which combines fast approximation techniques with efficient validation techniques in order to find *all minimal functional dependencies* in a given dataset. While operating on compact data structures, HyFD not only outperforms all existing approaches, it also scales to much larger datasets.

1. FUNCTIONAL DEPENDENCIES

A functional dependency (FD) written as $X \rightarrow A$ expresses that all pairs of records with same values in attribute combination X must also have same values in attribute A [6]. The values in A *functionally depend* on the values in X. Consequently, keys in relational datasets express functional dependencies, because they uniquely determine all other attributes. Functional dependencies also arise naturally from real-world facts that a dataset describes. In address datasets, for instance, a person's *firstname* might determine the *gender* attribute, the *zipcode* might determine *city*, and *birthdate* should determine *age*.

The most important use for functional dependencies is schema normalization [5]. Normalization processes systematically decompose relations with their functional dependencies to reduce data redundancy. But functional dependencies also support further data management tasks, such as query optimization [22], data integration [17], data cleansing [3], and data translation [4]. Although we find many such use cases, functional dependencies are almost never specified as concrete metadata. One reason is that they depend not only on a schema but also on a concrete relational instance. For example, *child* $\rightarrow$ *teacher* might hold for kindergarten children, but it does not hold for high-school children. Consequently, functional dependencies also change over time when data is extended, altered, or merged with other datasets. Therefore, discovery algorithms are needed that reveal all functional dependencies of a given dataset.

Due to the importance of functional dependencies, many discovery algorithms have already been proposed. Unfortunately, none of them is able to process datasets of real-world size, i.e., datasets with more than 50 columns and a million rows, as a recent study showed [20]. Because the need for normalization, cleansing, and query optimization increases with growing dataset sizes, larger datasets are those for which functional dependencies are most urgently needed. The reason why current algorithms fail on larger datasets is that they optimize for either many records *or* many attributes. This is a problem, because the discovery of functional dependencies is by nature quadratic in the number of records n and exponential in the number attributes m. More specifically, it is in $\mathcal{O}(n^2(\frac{m}{2})^2 2^m)$ as shown by Liu et al. [15]. Therefore, any truly scalable algorithm must be able to cope with both large schemata *and* many rows.

To approach such datasets, we propose a novel hybrid algorithm called HyFD, which combines row- and column-efficient discovery techniques: In a first phase, HyFD extracts a small subset of records from the input data and calculates the FDs of this non-random sample. Because only a subset of records is used in this phase, it performs particularly column-efficient. The result is a set of FDs that are either valid or almost valid with respect to the complete input dataset. In a second phase, HyFD validates the discovered FDs on the entire dataset and refines such FDs that do not yet hold. This phase is row-efficient, because it uses the previously discovered FDs to effectively prune the search space. If the validation becomes inefficient, HyFD is able to switch back into the first phase and continue there with all results discovered so far. This alternating, two-phased discovery strategy clearly outperforms all existing algorithms in terms of runtime and scalability, while still discovering *all minimal FDs*. In detail, our contributions are the following:

(1) *FD discovery.* We introduce HyFD, a hybrid FD discovery algorithm that is faster and able to handle much larger datasets than state-of-the-art algorithms.

(2) *Focused sampling.* We present sampling techniques that leverage the advantages of dependency induction algorithms while, at the same time, requiring far fewer comparisons.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915203

(3) *Direct validation.* We contribute an efficient validation technique that leverages the advantages of lattice traversal algorithms with minimal memory consumption.

(4) *Robust scaling.* We propose a best-effort strategy that dynamically limits the size of resulting FDs if these would otherwise exhaust the available memory capacities.

(5) *Exhaustive evaluation.* We evaluate our algorithm on more than 20 real-world datasets and compare it to seven state-of-the-art FD discovery algorithms.

In the following, Section 2 discusses related work. Then, Section 3 provides the theoretical foundations for our discovery strategy and Section 4 an overview on our algorithm HyFD. Sections 5, 6, 7, and 8 describe the different components of HyFD in more detail. Section 10 evaluates our algorithm and compares it against seven algorithms from related work. We then conclude in Section 11.

2. RELATED WORK

The evaluating work of [20] compared the seven most popular algorithms for functional dependency discovery and demonstrated their individual strengths and weaknesses. Some effort has also been spent on the discovery of *approximate* [12] and *conditional* [3,7] functional dependencies, but those approaches are orthogonal to our research: We aim to discover *all minimal* functional dependencies without any restrictions or relaxations. Parallel and distributed dependency discovery systems, such as [10] and [14], form another orthogonal branch of research. They rely on massive parallelization rather than efficient pruning to cope with the discovery problem. We focus on more sophisticated search techniques and show that these can still be parallelized accordingly. In the following, we briefly summarize current state-of-the-art in non-distributed FD discovery.

Lattice traversal algorithms: The algorithms Tane [12], Fun [18], Fd_Mine [25], and Dfd [1] conceptually arrange all possible FD candidates in a powerset lattice of attribute combinations and then traverse this lattice. The first three algorithms search through the candidate lattice level-wise bottom-up using the *apriori-gen* candidate generation [2], whereas Dfd applies a depth-first random walk. Lattice traversal algorithms in general make intensive use of pruning rules and their candidate validation is based on stripped partitions (also called position list indices). They have been shown to perform well on long datasets, i.e., datasets with many records, but due to their candidate-driven search strategy, they scale poorly with the number of columns in the input dataset. In this paper, we adopt the pruning rules and the position list index data structure from these algorithms for the validation of functional dependencies.

Difference- and agree-set algorithms: The algorithms Dep-Miner [16] and FastFDs [24] analyze a dataset for sets of attributes that agree on the values in certain tuple pairs. These so-called agree-sets are transformed into difference-sets from which all valid FDs can be derived. This discovery strategy scales better with the number of attributes than lattice traversal strategies, because FD candidates are generated only from concrete observations rather than being generated systematically. The required maximization and minimization of agree- and difference-sets respectively, however, reduces this advantage significantly. With regard to the number of records, Dep-Miner and FastFDs scale much worse than the previous algorithms, because they need to compare all pairs of records. Our approach also compares records pair-wise, but we choose these comparisons carefully.

Dependency induction algorithms: The Fdep [9] algorithm also compares all records pair-wise to find all *invalid* functional dependencies. This set is called *negative cover* and is stored in a prefix tree. In contrast to Dep-Miner and FastFDs, Fdep translates this negative cover into the set of valid functional dependencies, i.e., the *positive cover*, not by forming complements but by successive specialization: The positive cover initially assumes that each attribute functionally determines all other attributes; these functional dependencies are then refined with every single non-FD in the negative cover. Apart from the fact that the pair-wise comparisons do not scale with the number of records in the input dataset, this discovery strategy has proven to scale well with the number of attributes. For this reason, we follow a similar approach during the induction of functional dependency candidates. However, we compress records before comparison, store the negative cover in a more efficient data structure, and optimize the specialization process.

The evaluation section of this paper provides a comparison of our approach with *all* mentioned related work.

3. HYBRID FD DISCOVERY

We begin this section by formally defining *functional dependencies* (FDs). We then discuss sampling-based FD discovery and our hybrid discovery approach.

Preliminaries. Our definition of a functional dependency follows [23]: A functional dependency (FD) written as $X \rightarrow A$ is a statement over a relational schema R, where $X \subseteq R$ and $A \in R$. The FD is valid for an instance r of R, iff for all pairs of tuples $t_1, t_2 \in r$ the following is true: if $t_1[B] = t_2[B]$ for all $B \in X$, then $t_1[A] = t_2[A]$. In other words, the values in X functionally determine the values in A. We call the determinant X the FD's *left hand side* (Lhs), and the dependent A the FD's *right hand side* (Rhs). Moreover, an FD $X \rightarrow A$ is a *generalization* of another FD $Y \rightarrow A$ if $X \subset Y$ and it is a *specialization* if $X \supset Y$. The FD $X \rightarrow A$ is *non-trivial* if $A \notin X$ and it is *minimal* if no B exists such that $X \backslash B \rightarrow A$ is a valid FD, i.e., if no valid generalization exists. To discover all FDs in a given relational instance r, it suffices to discover all minimal, non-trivial FDs, because all Lhs-subsets are non-dependencies and all Lhs-supersets are dependencies by logical inference.

The discovery of all minimal, non-trivial FDs can best be modeled as a graph search problem: The graph is a powerset lattice of attribute combinations and an edge between the nodes X and XA represents the FD candidate $X \rightarrow A$. Figure 1 depicts an example lattice and its FDs and non-FDs. In all such lattices, FDs are located in the upper part of the lattice; non-FDs are located at the bottom. A virtual border separates the FDs and non-FDs. All minimal FDs, which we aim to discover, reside on this virtual border line.

Sampling-based FD discovery. For a relational instance r, a sample r' of r contains only a subset of records $r' \subset r$. Because r' is smaller than r, discovering all minimal FDs on r' is expected to be cheaper than discovering all minimal FDs on r (with any FD discovery algorithm). The resulting r'-FDs can, then, be valid or invalid in r, but they exhibit three properties that are important for our hybrid algorithm:

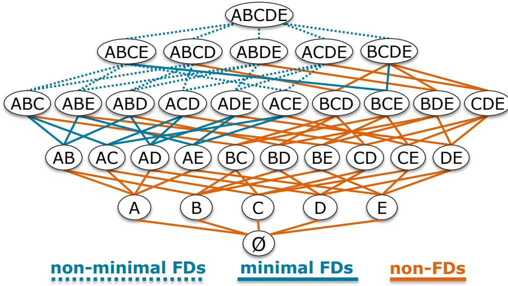

Figure 1: FD discovery in a power set lattice.

(1) *Completeness:* The set of r'-FDs implies the set of r-FDs, i.e., we find an $X' \to A$ in r' for each valid $X \to A$ in r with $X' \subseteq X$. Hence, all $X \to A$ are also valid in r' and the sampling result is complete. To prove this, assume $X \to A$ is valid in r but invalid in r'. Then r' must invalidate $X \to A$ with two records that do not exist in r. So it is $r' \not\subset r$, which contradicts $r' \subset r$.

(2) *Minimality:* If a minimal r'-FD is *valid* on the entire instance r, then the FD must also be minimal in r. This means that the sampling cannot produce non-minimal or incomplete results. In other words, a functional dependency cannot be valid in r but invalid in r'. This property is easily proven: If $X \to A$ is invalid in r', then r' contains two records with same X values but different A values. Because $r' \subset r$, the same records must also exist in r. Therefore, $X \to A$ must be invalid in r as well.

(3) *Proximity:* If a minimal r'-FD is *invalid* on the entire instance r, then the r'-FD is still *close* to specializations that are valid in r. In other words, r'-FDs are always located closer to the virtual border, which holds the true r-FDs, than the FDs at the bottom of the lattice. Therefore, any sampling-based FD discovery algorithm approximates the real FDs. The distance between r'-FDs and r-FDs still depends on the sampling algorithm and the entire data.

In summary, a sampling-based FD discovery algorithm calculates a set of r'-FDs that are either r-FDs or possibly close generalizations of r-FDs. In terms of Figure 1, the result of the sampling is a subset of solid lines.

The hybrid approach. Recently, the authors of [20] made the observation that current FD discovery algorithms either scale well with the number of records (e.g., DFD) or they scale well with the number of attributes (e.g., FDEP). None of the algorithms, however, addresses both dimensions equally well. Therefore, we propose a hybrid algorithm that combines column-efficient FD induction techniques with row-efficient FD search techniques in two phases.

In Phase 1, the algorithm uses column-efficient FD induction techniques. Because these are sensitive to the number of rows, we process only a small sample of the input. The idea is to produce with low effort a set of FD candidates that are according to property (3) *proximity* close to the real FDs. To achieve this, we propose focused sampling techniques that let the algorithm select samples with a possibly large impact on the result's precision. Due to sampling properties (1) *completeness* and (2) *minimality*, these techniques cannot produce non-minimal or incomplete results.

In Phase 2, the algorithm uses row-efficient FD search techniques to validate the FD candidates given by Phase 1. Because the FD candidates and their specializations represent only a small subset of the search space, the number of columns in the input dataset has a much smaller impact on the row-efficient FD search techniques. Furthermore, the FD candidates should be valid FDs or close to valid specializations due to sampling property (3) *proximity*. The task of the second phase is, hence, to check all FD candidates and to find valid specializations if a candidate is invalid.

Although the two phases match perfectly, finding an appropriate, dataset-independent criterion for when to switch from Phase 1 into Phase 2 is difficult. If we switch too early into Phase 2, the FD candidates approximate the real FDs only poorly and the search space becomes large; if we remain too long in Phase 1, we might end up analyzing the entire dataset with only column-efficient FD induction techniques, which is very expensive on many rows. For this reason, we propose to switch between the two phases back and forth whenever the currently running strategy becomes inefficient.

For Phase 1, we track the *sampling efficiency*, which is defined as the number of new observations per comparison. If this efficiency falls below an optimistic threshold, the algorithm switches into Phase 2. In Phase 2, we then track the *validation efficiency*, which is the number of discovered valid FDs per validation. Again, if this efficiency drops below a given threshold, the validation process can be considered inefficient and we switch back into Phase 1. In this case, the previous sampling threshold was too optimistic, so the algorithm dynamically increases it.

When switching back and forth between the two phases, the algorithm can share insights between the different strategies: The validation phase obviously profits from the FD candidates produced by the sampling phase; the sampling phase, in turn, profits from the validation phase, because the validation hints on interesting tuples that already invalidated some FD candidates. The hybrid FD discovery terminates when Phase 2 finally validated all FD candidates. We typically observe three to eight switches from Phase 2 back into Phase 1 until the algorithm finds the complete set of minimal functional dependencies. This result is correct, complete, and minimal, because Phase 1 is complete and minimal, as we have shown, and Phase 2 finally releases a correct, complete, and minimal result as shown by [12].

4. THE HYFD ALGORITHM

We implemented the hybrid FD discovery approach as the HyFD algorithm. Figure 2 gives an overview of HyFD showing its components and the control flow between them. In the following, we briefly introduce each component and their tasks in the FD discovery process. Each component is later explained in detail in their respective sections. Note that the `Sampler` and the `Inductor` component together implement Phase 1 and the `Validator` component implements Phase 2.

(1) `Preprocessor`. To discover functional dependencies, we must know the positions of same values for each attribute, because same values in an FD's LHS can make it invalid if the according RHS values differ. The values itself, however, must not be known. Therefore, HyFD's `Preprocessor` component first transforms the records of a given dataset into compact *position list indexes* (PLI). For performance reasons, the component also pre-calculates the inverse of this index, which is later used in the validation step. Be-

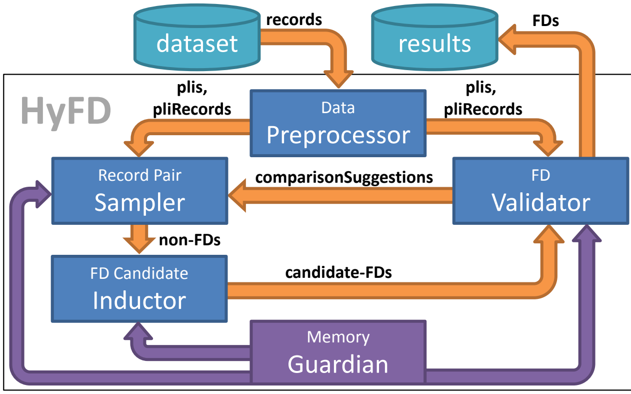

Figure 2: Overview of HyFD and its components.

cause HyFD uses sampling to combine row- with column-efficient discovery techniques, it still needs to access the input dataset's records. For this purpose, the `Preprocessor` compresses the records using the Plis as dictionaries.

(2) `Sampler`. The `Sampler` component implements the first part of a column-efficient FD induction technique: It starts the FD discovery by checking the compressed records for FD-violations. An FD-violation is a pair of two records that match in one or more attribute values. From such record pairs, the algorithm infers that the matching attributes cannot functionally determine any of the non-matching attributes. Hence, they indicate non-valid FDs or short *non-FDs*. The schema $R(A, B, C)$, for instance, could hold the two records $r_1(1, 2, 3)$ and $r_2(1, 4, 5)$. Because the A-values match and the B- and C-values differ, $A \not\to B$ and $A \not\to C$ are two non-FDs in R. Finding all these non-FDs requires to systematically match all records pair-wise. Because this quadratic complexity does not scale in practice, the `Sampler` carefully selects only a subset of record pairs, namely those that indicate possibly many FD-violations. For the selection of record pairs, the component uses a deterministic, focused sampling technique that we call *cluster windowing*.

(3) `Inductor`. The `Inductor` component implements the second part of the column-efficient FD induction technique: From the `Sampler` component, it receives a rich set of non-FDs that must be converted into *FD-candidates*. An FD-candidate is an FD that is minimal and valid with respect to the chosen sample – whether a candidate is actually valid on the entire dataset is determined in Phase 2. The conversion algorithm is similar to the Fdep algorithm [9]: We first assume that the empty set functionally determines all attributes; then, we successively specialize this assumption with every known non-FD. Recall the example schema $R(A, B, C)$ and its known non-FD $A \not\to B$. Initially, we define our result to be $\emptyset \to ABC$, which is a short notation for the FDs $\emptyset \to A$, $\emptyset \to B$, and $\emptyset \to C$. Because $A \not\to B$, the FD $\emptyset \to B$, which is a generalization of our known non-FD, must be invalid as well. Therefore, we remove it and add all valid, minimal, non-trivial specializations. Because this is only $C \to B$, our new result set is $\emptyset \to AC$ and $C \to B$. To execute the specialization process efficiently, the `Inductor` component maintains the valid FDs in a prefix tree that allows for fast generalization look-ups. If the `Inductor` is called again, it can continue specializing the FDs that it already knows, so it does not start with an empty prefix tree.

(4) `Validator`. The `Validator` component implements a row-efficient FD search technique: It takes the candidate-FDs from the `Inductor` and validates them against the entire dataset, which is given as a set of Plis from the `Preprocessor`. When modeling the FD search space as a powerset lattice, the given candidate-FDs approximate the final FDs from below, i.e., a candidate-FD is either a valid FD or a generalization of a valid FD. Therefore, the `Validator` checks the candidate-FDs level-wise bottom-up: Whenever the algorithm finds an invalid FD, it exchanges this FD with all its minimal, non-trivial specializations using common pruning rules for lattice traversal algorithms [12]. If previous calculations yielded a good approximation of the valid FDs, only few FD candidates need to be specialized; otherwise, the number of invalid FDs increases rapidly from level to level and the `Validator` switches back to `Sampler`. The FD validations themselves build upon direct refinement checks and avoid the costly hierarchical Pli intersections that are typical in all current lattice traversal algorithms. In the end, the `Validator` outputs all minimal, non-trivial FDs for the given input dataset.

(5) `Guardian`. FD result sets can grow exponentially with the number of attributes in the input relation. For this reason, discovering complete result sets can sooner or later exhaust any memory-limit, regardless of how compact intermediate data structures, such as Plis or results, are stored. Therefore, a robust algorithm must prune the results in some reasonable way, if memory threatens to be exhausted. This is the task of HyFD's `Guardian` component: Whenever the prefix tree, which contains the valid FDs, grows, the `Guardian` checks the current memory consumption and prunes the FD tree, if necessary. The idea is to give up FDs with largest left-hand-sides, because these FDs mostly hold accidentally in a given instance but not semantically in the according schema. Overall, however, the `Guardian` is an optional component in the HyFD algorithm and does not contribute in the discovery process itself. Our overarching goal remains to find the *complete* set of minimal FDs.

5. PREPROCESSING

The `Preprocessor` is responsible for transforming the input data into two compact data structures: *plis* and *pliRecords*. The first data structure *plis* is an array of *position list indexes* (Pli). In the literature, these Plis are also known as *stripped partitions* [8, 12]. A Pli, denoted by π_X, groups tuples into equivalence classes by their values of attribute set X. Thereby, two tuples t_1 and t_2 of an attribute set X belong to the same equivalence class if $\forall A \in X : t_1[A] = t_2[A]$. These equivalence classes are also called *clusters*, because they cluster records by same values. For compression, a Pli does not store clusters with only a single entry, because tuples that do not occur in any cluster of π_X can be inferred to be unique in X. Consider, for example, the relation Class(Teacher, Subject) and its tuples (Brown, Math), (Walker, Math), (Brown, English), (Miller, English), and (Brown, Math). Then, $\pi_{\{Teacher\}} = \{\{1, 3, 5\}\}$, $\pi_{\{Subject\}} = \{\{1, 2, 5\}, \{3, 4\}\}$, and $\pi_{\{Teacher, Subject\}} = \{\{1, 5\}\}$. Such Plis can efficiently be implemented as sets of record ID sets, which we wrap in Pli objects.

To check a functional dependency $X \to A$ using only Plis, we can test if every cluster in π_X is a subset of some cluster of π_A. If this holds true, then all tuples with same values in

X have also same values in A, which is the definition of an FD. This check is called *refinement* (see Section 8) and was first introduced in [12].

Algorithm 1 shows the `Preprocessor` component and the two data structures it produces: The already discussed *plis* and a PLI-compressed representation of all records, which we call *pliRecords*. For their creation, the algorithm first determines the number of input records *numRecs* and the number of attributes *numAttrs* (Lines 1 and 2). Then, it builds the *plis* array – one π for each attribute. This is done by hashing each value to a list of record IDs and then simply collecting these lists in a PLI object (Line 4). When created, the `Preprocessor` sorts the array of PLIs in descending order by the number of clusters (including clusters of size one, whose number is implicitly known). This sorting improves the FD-candidate validations of the `Validator` component, which we discuss in Section 8.

Algorithm 1: Data Preprocessing

Data: *records*
Result: *plis*, *invertedPlis*, *pliRecords*

1. *numRecs* ← |*records*|;
2. *numAttrs* ← |*records*[0]|;
3. **array** *plis* **size** *numAttrs* **as** PLI;
4. *plis* ← *buildPlis*(*records*);
5. *plis* ← *sort*(*plis*, DESCENDING);
6. **array** *pliRecords* **size** *numRecs* × *numAttrs* **as** Integer;
7. *pliRecords* ← *createRecords*(*invertedPlis*);
8. **return** *plis*, *invertedPlis*, *pliRecords*;

With the *plis*, the `Preprocessor` finally creates dictionary compressed representations of all records, the *pliRecords* (Lines 6 and 7). A compressed record is an array of cluster IDs where each field denotes the record's cluster in attribute $A \in [0, numAttrs[$. We extract these representations from the *plis* that already map cluster IDs to record IDs for each attribute. The PLI-compressed records are needed in the sampling phase to find FD-violations and in the validation phase to find LHS- and RHS-cluster IDs for certain records.

6. SAMPLING

The idea of the `Sampler` component is to analyze a dataset, which is represented by the *pliRecords*, for FD-violations, i.e., non-FDs that can later be converted into FDs. To derive FD-violations, the component compares records pair-wise. These pair-wise record comparisons are robust against the number of columns, but comparing all pairs of records scales quadratically with their number. Therefore, the `Sampler` uses only a subset, i.e., a sample of record pairs for the non-FD calculations. The record pairs in this subset should be chosen carefully, because some pairs are more likely to reveal FD-violations than others. In the following, we first discuss how non-FDs are identified; then, we present a deterministic focused sampling technique, which extracts a non-random subset of promising record pairs for the non-FD discovery; lastly, we propose an implementation of our sampling technique.

Retrieving non-FDs. A functional dependency $X \rightarrow A$ can be invalidated with two records that have matching X and differing A values. Therefore, the non-FD search is based on pair-wise record comparisons: If two records match in their values for attribute set Y and differ in their values for attribute set Z, then they invalidate all $X \rightarrow A$ with $X \subseteq Y$ and $A \in Z$. The corresponding FD-violation $Y \not\rightarrow Z$ can be efficiently stored in bitsets that hold a 1 for each matching attribute of Y and a 0 for each differing attribute Z. To calculate these bitsets, we use the *match*()-function, which compares two PLI-compressed records element-wise. Because the records are given as Integer arrays (and not as, for instance, String arrays), this function is cheap in contrast to the validation and specialization functions used by other components of HYFD.

Sometimes, the sampling discovers the same FD-violations with different record pairs. For this reason, the bitsets are stored in a set called *nonFds*, which automatically eliminates duplicate observations. For the same task, related algorithms, such as FDEP [9], proposed prefix-trees, but these data structures consume much more memory and do not yield a better performance. Reconsidering Figure 1, we can easily see that the number of non-FDs is much larger than the number of minimal FDs, so storing the non-FDs in a memory efficient data structure is crucial.

Focused sampling. FD-violations are retrieved from record pairs, and while certain record pairs indicate important FD-violations, the same two records may not offer any new insights when compared with other records. So an important aspect of focused sampling is that we sample *record pairs* and not *records*. Thereby, only record pairs that match in at least one attribute can reveal FD-violations; comparing records with no overlap should be avoided. A focused sampling algorithm can easily assure this by comparing only those records that co-occur in at least one PLI-cluster. But due to columns that contain only few distinct values, most record pairs co-occur in some cluster. Therefore, more sophisticated pair selection techniques are needed.

The problem of finding promising comparison candidates is a well known problem in duplicate detection research. A popular solution for this problem is the *sorted neighborhood* pair selection algorithm [11]. The idea is to first sort the data by some domain-dependent key that sorts similar records close to one another; then, the algorithm compares all records to their w closest neighbors, where w is called *window*. Because our problem of finding violating record pairs is similar to finding matching record pairs, we use the same idea for our focused sampling algorithm.

At first, we sort similar records, i.e., records that co-occur in certain PLI-clusters, close to one-another. We do this for all clusters in all PLIs with different sorting keys each. Then, we slide a window over the clusters and compare all record pairs within this window. Because some PLIs produce better sortations than others in the sense that they reveal more FD-violations than others, the algorithm shall automatically prefer more efficient sortations over less efficient ones. This can be done with a progressive selection technique, which is also known from duplicate detection [21]: The algorithm first compares all records to their direct neighbors and counts the results; afterwards, the result counts are ranked and the sortation with the most results is chosen to run a slightly larger window ($w + 1$). The algorithm stops continuing best sortations, when all sortations have become inefficient. In this way, the algorithm automatically chooses most profitable comparisons. When adapting the same strategy for our FD-violation search, we can save many comparisons: Because efficient sortations anticipate most informative comparisons, less efficient sortations become quickly inefficient.

Algorithm 2: Record Pair Sampling

```
Data: plis, pliRecords, comparisonSuggestions
Result: nonFds
1  if efficiencyQueue = ∅ then
2  |  for pli ∈ plis do
3  |  |  for cluster ∈ pli do
4  |  |  |  cluster ← sort(cluster, ATTR_LEFT_RIGHT);
5  |  nonFds ← ∅;
6  |  efficiencyThreshold ← 0.01;
7  |  efficiencyQueue ← new PriorityQueue;
8  |  for attr ∈ [0, numAttributes[ do
9  |  |  efficiency ← new Efficiency;
10 |  |  efficiency.attribute ← attr;
11 |  |  efficiency.window ← 2;
12 |  |  efficiency.comps ← 0;
13 |  |  efficiency.results ← 0;
14 |  |  runWindow(efficiency, plis[attr], nonFds);
15 |  |  efficiencyQueue.append(efficiency);
16 else
17 |  efficiencyThreshold ← efficiencyThreshold / 2;
18 |  for sug ∈ comparisonSuggestions do
19 |  |  nonFds ← nonFds ∪ match(sug[0], sug[1]);
20 while true do
21 |  bestEff ← efficiencyQueue.peek();
22 |  if bestEff.eval() < efficiencyThreshold then
23 |  |  break;
24 |  bestEff.window ← bestEff.window + 1;
25 |  runWindow(bestEff, plis[bestEff.attribute], nonFds);
26 return newFDsIn(nonFds);
   function runWindow(efficiency, pli, nonFds)
27 prevNumNonFds ← |nonFds|;
28 for cluster ∈ pli do
29 |  for i ∈ [0, |cluster| − efficiency.window [ do
30 |  |  pivot ← pliRecords[cluster[i]];
31 |  |  partner ← pliRecords[cluster[i + window − 1]];
32 |  |  nonFds ← nonFds ∪ match(pivot, partner);
33 |  |  efficiency.comps ← efficiency.comps + 1;
34 newResults ← |nonFds| − prevNumNonFds;
35 efficiency.results ← efficiency.results + newResults;
```

Finally, the focused sampling must decide on when the comparisons of records in a certain sortation, i.e., for a certain PLI, become inefficient. We propose to start with a rather strict definition of efficiency, because HYFD will return into the sampling phase anyway, if the number of identified FD-violations was too low. So an efficiency threshold could be 0.01, which is one new FD-violation within 100 comparisons – in fact, Section 10 shows that this threshold performs well on *all* dataset sizes. To relax this threshold in subsequent iterations, we double the number of comparisons whenever the algorithm returns to the sampling phase.

The sampling algorithm. Algorithm 2 implements the focused sampling strategy introduced above. It requires the *plis* and *pliRecords* from the `Preprocessor` and the *comparisonSuggestions* from the `Validator`. Figure 3 illustrates the algorithm.

The priority queue *efficiencyQueue* is a local data structure that ranks the PLIs by their sampling efficiency. If the *efficiencyQueue* is empty (Line 1), this is the first time the `Sampler` is called. In this case, we need to sort all clusters by some cluster-dependent sorting key (Lines 2 to 4). As shown in Figure 3.1, we sort the records in each cluster of attribute A_i's PLI by their cluster number in attribute A_{i-1} and, if

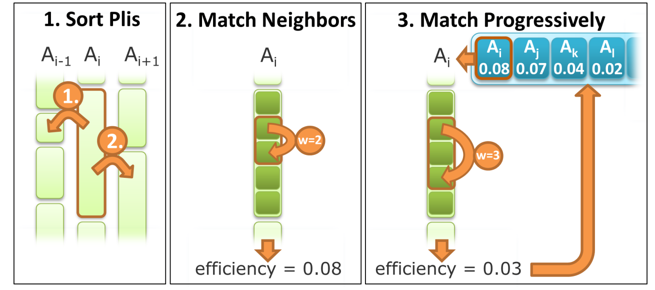

Figure 3: Focused sampling: Sorting of Pli clusters (1); record matching to direct neighbors (2); progressive record matching (3).

numbers are equal or unknown, by the cluster number in A_{i+1}. The intuition here is that attribute A_{i-1} has more clusters than A_i, due to the sorting of *plis* in the `Preprocessor`, which makes it a promising key; some unique values in A_{i-1}, on the other hand, do not have a cluster number, so the sorting also checks the PLI of attribute A_{i+1} that has larger clusters than A_i. However, the important point in choosing sorting keys is not which $A_{i+/-x}$ to take but to take different sorting keys for each PLI. In this way, the neighborhood of one record differs in each of its PLI clusters.

When the sorting is done, the algorithm initializes the *efficiencyQueue* with first *efficiency* measurements. The efficiency of an attribute's PLI is an object that stores the PLI's sampling performance: It holds the attribute identifier, the last window size, the number of comparisons within this window, and the number of results, i.e., FD-violations first revealed with these comparisons. An efficiency object can calculate its efficiency by dividing the number of results by the number of comparisons. For instance, 8 new FD-violations in 100 comparisons yield an efficiency of 0.08. To initialize the efficiency object of each attribute, the `Sampler` runs a window of size two over the attribute's PLI clusters (Line 14) using the *runWindow*()-function shown in Lines 27 to 35. Figure 3.2 illustrates how this function compares all direct neighbors in the clusters with window size two.

If the `Sampler` is not called for the first time, the PLI clusters are already sorted and the last efficiency measurements are also present. We must, however, relax the efficiency threshold (Line 17) and execute the suggested comparisons (Lines 18 and 19). The suggested comparisons are records pairs that violated at least one FD candidate in Phase 2 of the HYFD algorithm so that they probably also violate some more FDs. With the suggested comparisons, Phase 1 incorporates knowledge from Phase 2 to focus the sampling.

No matter whether this is the first or a subsequent call of the `Sampler`, the algorithm finally starts a progressive search for more FD-violations (Lines 20 to 25): It selects the efficiency object *bestEff* with the highest efficiency in the *efficiencyQueue* (Line 21) and executes the next window size on its PLI (Line 25). This updates the efficiency of *bestEff* so that it might get re-ranked in the priority queue. Figure 3.3 illustrates one such progressive selection step for a best attribute A_i with efficiency 0.08 and next window size three: After matching all records within this window, the efficiency drops to 0.03, which makes A_j the new best attribute.

The `Sampler` algorithm continues running ever larger windows over the PLIs until all efficiencies have fallen below the

current *efficiencyThreshold* (Line 22). At this point, the row-efficient discovery technique has apparently become inefficient and the algorithm decides to proceed with a column-efficient discovery technique.

7. INDUCTION

The `Inductor` component concludes the column-efficient discovery phase and leads over into the row-efficient discovery phase. Its task is to convert the *nonFds* given by the `Sampler` component into corresponding minimal FD-candidates *fds*. These FD-candidates are stored in a data structure called *FDTree*, which is a prefix-tree optimized for functional dependencies. Figure 4 shows three such FDTrees with example FDs. First introduced by Flach and Savnik in [9], an FDTree maps the Lhs of FDs to nodes in the tree and the Rhs of these FDs to bitsets, which are attached to the nodes. A Rhs attribute in the bitsets is marked if it is at the end of an FD's Lhs path, i.e., if the current path of nodes describes the entire Lhs to which the Rhs belongs to.

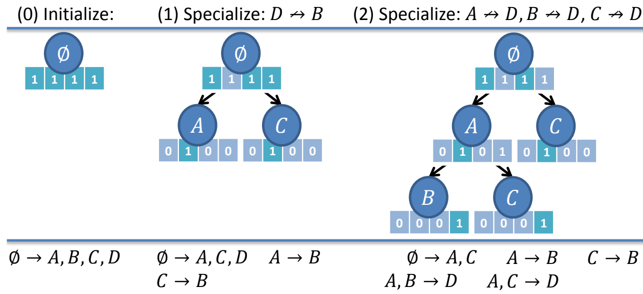

Figure 4: Specializing the FDTree with non-FDs.

Algorithm 3 shows the conversion process in detail. The `Inductor` first sorts the *nonFds* in descending order by their cardinality, i.e., the number of set bits (Line 1). The sorting of FD-violations is important, because it lets HyFD convert non-FDs with long Lhss into FD-candidates first and non-FDs with ever shorter Lhss gradually later achieving much stronger pruning in the beginning. In this way, the prefix-tree of candidate-FDs *fds* grows much slower, significantly reducing the costs for early generalization look-ups.

When the `Inductor` is called for the first time, the FDTree *fds* has not been created yet and is initialized with a schema R's most general FDs $\emptyset \rightarrow R$, where the attributes in R are represented as integers (Line 4); otherwise, the algorithm continues with the previously calculated *fds*. The task is to specialize the *fds* with every bitset in *nonFds*: Each bitset describes the Lhs of several non-FDs (Line 5) and each zero-bit in these bitsets describes a Rhs of a non-FD (Lines 6 and 7). Once retrieved from the bitsets, each non-FD is used to specialize the FDTree *fds* (Line 8).

Figure 4 exemplarily shows the specialization of the initial FDTree for the non-FD $D \nrightarrow B$ in (1): First, the *specialize*-function recursively collects the invalid FD and all its generalizations from the *fds* (Line 10), because these must be invalid as well. In our example, the only invalid FD in the tree is $\emptyset \rightarrow B$. HyFD then successively removes these non-FDs from the FDTree *fds* (Line 12). Once removed, the non-FDs are specialized, which means that the algorithm *extends* the Lhs of each non-FD to generate still valid specializations (Line 17). In our example, these are $A \rightarrow B$ and $C \rightarrow B$. Before adding these specializations, the `Inductor` assures that the new candidate-FD are minimal by searching the *fds* for generalizations of the candidate-FDs (Line 18). Figure 4 also shows the result when inducing three more non-FDs into the FDTree. After specializing the *fds* with all *nonFds*, the prefix-tree holds the entire set of valid, minimal FDs with respect to these given non-FDs [9].

Algorithm 3: Functional Dependency Induction

```
Data: nonFds
Result: fds
1  nonFds ← sort(nonFds, CARDINALITY_DESCENDING);
2  if fds = null then
3      fds ← new FDTree;
4      fds.add(∅ → {0, 1, ..., numAttributes});
5  for lhs ∈ nonFds do
6      rhss ← lhs.clone().flip();
7      for rhs ∈ rhss do
8          specialize(fds, lhs, rhs);
9  return fds;
   function specialize(fds, lhs, rhs)
10 invalidLhss ← fds.getFdAndGenerals(lhs, rhs);
11 for invalidLhs ∈ invalidLhss do
12     fds.remove(invalidLhs, rhs);
13     for attr ∈ [0, numAttributes[ do
14         if invalidLhs.get(attr) ∨
15            rhs = attr then
16             continue;
17         newLhs ← invalidLhs ∪ attr;
18         if fds.findFdOrGeneral(newLhs, rhs) then
19             continue;
20         fds.add(newLhs, rhs);
```

8. VALIDATION

The `Validator` component takes the previously calculated FDTree *fds* and validates the contained FD-candidates against the entire input dataset, which is represented by the *plis* and the *invertedPlis*. For this validation process, the component uses a row-efficient lattice traversal strategy. We first discuss the lattice traversal; then, we introduce our direct candidate validation technique; and finally, we present the specialization method of invalid FD-candidates. The `Validator` component is shown in detail in Algorithm 4.

Traversal. Usually, lattice traversal algorithms need to traverse a huge candidate lattice, because FDs can be everywhere (see Figure 1 in Section 3). Due to the previous, sampling-based discovery, HyFD already starts the lattice traversal with a set of promising FD-candidates *fds* that are organized in an FDTree. Because this FDTree maps directly to the FD search space, i.e., the candidate lattice, HyFD can use it to systematically check all necessary FD candidates: Beginning from the root of the tree, the `Validator` component traverses the candidate set breath-first level by level.

When the `Validator` component is called for the first time (Line 1), it initializes the *currentLevelNumber* to zero (Line 2); otherwise, it continues the traversal from where it stopped before. During the traversal, the set *currentLevel* holds all FDTree nodes of the current level. So before entering the level-wise traversal in Line 5, the `Validator` initializes the *currentLevel* using the *getLevel*()-function (Line 3). This function recursively collects all nodes with depth *currentLevelNumber* from the prefix-tree *fds*.

Algorithm 4: Functional Dependency Validation

```
   Data: fds, plis, pliRecords
   Result: fds, comparisonSuggestions
1  if currentLevel = null then
2      currentLevelNumber ← 0;
3  currentLevel ← fds.getLevel(currentLevelNumber);
4  comparisonSuggestions ← ∅;
5  while currentLevel ≠ ∅ do
       /* Validate all FDs on the current level          */
6      invalidFds ← ∅;
7      numValidFds ← 0;
8      for node ∈ currentLevel do
9          lhs ← node.getLhs();
10         rhss ← node.getRhss();
11         validRhss ← refines(lhs, rhss, plis, pliRecords,
                                 comparisonSuggestions);
12         numValidFds ← numValidFds + |validRhss|;
13         invalidRhss ← rhss.andNot(validRhss);
14         node.setFds(validRhss);
15         for invalidRhs ∈ invalidRhss do
16             invalidFds ← invalidFds ∪ (lhs, invalidRhs);
       /* Add all children to the next level             */
17     nextLevel ← ∅;
18     for node ∈ currentLevel do
19         for child ∈ node.getChildren() do
20             nextLevel ← nextLevel ∪ child;
       /* Specialize all invalid FDs                     */
21     for invalidFd ∈ invalidFds do
22         lhs, rhs ← invalidFd;
23         for attr ∈ [0, numAttributes[ do
24             if lhs.get(attr) ∨ rhs = attr ∨
25                fds.findFdOrGeneral(lhs, attr) ∨
26                fds.findFd(attr, rhs) then
27                 continue;
28             newLhs ← lhs ∪ attr;
29             if fds.findFdOrGeneral(newLhs, rhs) then
30                 continue;
31             child ← fds.addAndGetIfNew(newLhs, rhs);
32             if child ≠ null then
33                 nextLevel ← nextLevel ∪ child;
34     currentLevel ← nextLevel;
35     currentLevelNumber ← currentLevelNumber + 1;
       /* Judge efficiency of validation process         */
36     if |invalidFds| > 0.01 * numValidFds then
37         return fds, comparisonSuggestions;
38 return fds, ∅;
```

On each level (Line 5), the algorithm first validates all FD-candidates removing those from the FDTree that are invalid (Lines 6 to 16); then, the algorithm collects all child-nodes of the current level to form the next level (Lines 17 to 20); finally, it specializes the invalid FDs of the current level which generates new, minimal FD-candidates for the next level (Lines 21 to 33). The level-wise traversal stops, if the validation process becomes inefficient (Lines 36 and 37). Here, this means that more than 1% of the FD-candidates of the current level were invalid and the search space started growing rapidly. HyFD then returns into the sampling phase. We use 1% as a static threshold for efficiency of this phase, but our experiments in Section 10.5 show that any small percentage performs well here due to the observed high growth rate of invalid FD-candidates. The validation terminates when the next level is empty (Line 5) and all FDs in the FDTree *fds* are valid. This also ends the entire HyFD algorithm.

Validation. Each node in an FDTree can harbor multiple FDs with the same Lhs and different Rhss (see Figure 4 in Section 7): The Lhs attributes are described by a node's path in the tree and the Rhs attributes that form FDs with the current Lhs are marked. The `Validator` component validates all FD-candidates of a node simultaneously using the *refines*()-function (Line 11). This function checks which Rhs attributes are *refined* by the current Lhs using the *plis* and *pliRecords*. The refined Rhs attributes indicate valid FDs, while all other Rhs attributes indicate invalid FDs.

Figure 5 illustrates how the *refines*()-function works: Let $X \to Y$ be the set of FD-candidates that is to be validated. At first, the function selects the *pli* of the first Lhs attribute X_0. Due to the sorting of *plis* in the `Preprocessor` component, this is the Pli with the most and, hence, the smallest clusters of all Lhs attributes. For each cluster in X_0's Pli, the algorithm iterates all record IDs r_i in this cluster and retrieves the according compressed records from the *pliRecords*. A compressed record contains all cluster IDs in which a record is contained. Hence, the algorithm can create one array containing the Lhs cluster IDs of X and one array containing the Rhs cluster IDs of Y. The Lhs array, then, describes the cluster of r_i regarding attribute combination X. To check which Rhs Pli these Lhs clusters refine, we map the Lhs clusters to the corresponding array of Rhs clusters. We fill this map while iterating the record IDs of a cluster. If an array of Lhs clusters already exists in this map, the array of Rhs clusters must match the existing one. All non-matching Rhs clusters indicate refinement-violations and, hence, invalid Rhs attributes. The algorithm immediately stops checking such Rhs attributes so that only valid Rhs attributes survive until the end.

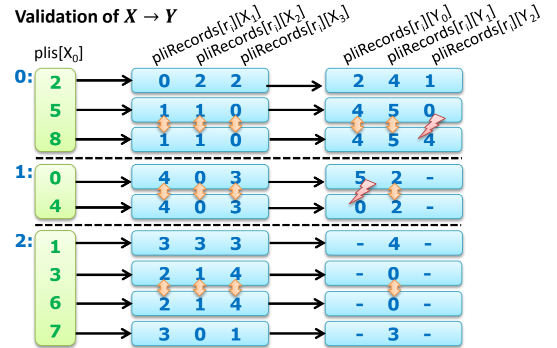

Figure 5: Directly validating FD-candidates $X \to Y$.

In comparison to other Pli-based algorithms, such as Tane, HyFD's validation technique avoids the costly hierarchical Pli intersections. By mapping the Lhs clusters to Rhs clusters, the checks are independent of other checks and do not require intermediate Plis. The direct validation is important, because the `Validator`'s starting FD candidates are – due to the sampling-based induction part – on much higher lattice levels and successively intersecting lower level Plis would undo this advantage. Furthermore, HyFD can terminate refinement checks very early if all Rhs attributes are invalid, because the results of the intersections, i.e., the intersected Plis are not needed for later intersections. Not storing intermediate Plis also has the advantage of demanding much less memory – most Pli-based algorithms fail at processing larger datasets, for exactly this reason [20].

Specialization. The validation of FD candidates identifies all invalid FDs and collects them in the set *invalidFds*. The specialization part of Algorithm 4, then, extends these invalid FDs in order to generate new FD candidates for the next higher level: For each invalid FD represented by lhs and rhs (Line 21), the algorithm checks for all attributes $attr$ (Line 23) if they specialize the invalid FD into a new minimal, non-trivial FD candidate $lhs \cup attr \rightarrow rhs$. To assure minimality and non-triviality of the new candidate, the algorithm tests the following:

(1) *Non-triviality*: $attr \notin lhs$ and $attr \neq rhs$ (Line 24)

(2) *Minimality 1*: $lhs \not\rightarrow attr$ (Line 25)

(3) *Minimality 2*: $lhs \cup attr \not\rightarrow rhs$ (Lines 24 and 29)

For the minimality checks, the `Validator` algorithm recursively searches for generalizations in the FDTree *fds*. This is possible, because all generalizations in the FDTree have already been validated and must, therefore, be correct. The generalization look-ups also include the new FD candidate itself, because if this is already present in the tree, it does not need to be added again. The minimality checks logically correspond to candidate pruning rules, as used by lattice traversal algorithms, such as TANE, FUN, and DFD.

If a minimal, non-trivial specialization has been found, the algorithm adds it to the FDTree *fds* (Line 31). The adding of a new FD into the FDTree might create a new node in the graph. To handle these new nodes on the next level, the algorithm must add them to *nextLevel*. When the specialization has finished with all invalid FDs, the `Validator` moves to the next level. If the next level is empty, all FD-candidates have been validated and *fds* contains all minimal, non-trivial functional dependencies of the input dataset.

9. MEMORY GUARDIAN

The memory `Guardian` is an optional component in HYFD and enables a best-effort strategy for FD discovery for very large inputs. Its task is to observe the memory consumption and to free resources if HYFD is about to reach the memory limit. Observing memory consumption is a standard task in any programming language. So the question is, what resources the `Guardian` can free if the memory is exhausted.

The PLI data structures grow linearly with the input dataset's size and are relatively small. The number of FD-violations found in the sampling step grows exponentially with the number of attributes, but it takes quite some attributes to exhaust the memory with these compact bitsets. The data structure that grows by far the fastest is the FDTree *fds*, which is constantly specialized by the `Inductor` and `Validator` components. Hence, this is the data structure the `Guardian` must prune.

Obviously, shrinking the *fds* is only possible by giving up some results, i.e., giving up completeness of the algorithm. In our implementation of the `Guardian`, we decided to successively reduce the maximum LHS size of our results; we provide three reasons: First, FDs with a long LHS usually occur accidentally, meaning that they hold for a particular instance but not for the relation in general. Second, FDs with long LHSs are less useful in most use cases, e.g., they become worse key/foreign-key candidates when used for normalization and they are less likely to match a query when used for query optimization. Third, FDs with long LHSs consume more memory, because they are physically larger, and preferentially removing them retains more FDs in total.

To restrict the maximum size of the FDs' LHSs, we need to add some additional logic into the FDTree: It must hold the maximum LHS size as a variable, which the `Guardian` component can control; whenever this variable is decremented, the FDTree recursively removes all FDs with larger LHSs and sets their memory resources free. The FDTree also refuses to add any new FD with a larger LHS. In this way, the result pruning works without changing any of the other four components. However, note that the `Guardian` component prunes only such results whose size would otherwise exceed the memory capacity, which means that the component in general does not take action.

10. EVALUATION

FD discovery has shown to be quadratic in the number of records n and exponential in the number of attributes m [15]. This also holds for HYFD: Phase 1 is in $\mathcal{O}(mn^2 + m^2 2^m)$, because in the worst case n^2 records are compared with comparison costs m and for each of the m possible RHS attributes, 2^{m-1} LHS attribute combinations must be refined $m-1$ times in the negative cover. Phase 2 is also in $\mathcal{O}(mn^2 + m^2 2^m)$ as shown in [15], because our complexity is the same as for TANE. Note that each phase can (potentially) discover all minimal FDs without the other. The following experiments, however, show that HYFD is able to process significantly larger datasets than state-of-the-art FD discovery algorithms in less runtime. At first, we introduce our experimental setup. Then, we evaluate the scalability of HYFD with both a dataset's number of rows and columns. Afterwards, we show that HYFD performs well on different datasets. In all these experiments, we compare HYFD to seven state-of-the-art FD discovery algorithms. We, finally, analyze some characteristics of HYFD in more detail and discuss the results of the discovery process.

10.1 Experimental setup

Metanome. HYFD and all algorithms from related work have been implemented for the *Metanome* data profiling framework (www.metanome.de), which defines standard interfaces for different kinds of profiling algorithms. Metanome also provided the various implementations of the state of the art. Common tasks, such as input parsing, result formatting, and performance measurement are standardized by the framework and decoupled from the algorithms [19].

Hardware. We run all our experiments on a Dell PowerEdge R620 with two Intel Xeon E5-2650 2.00 GHz CPUs and 128 GB RAM. The server runs on CentOS 6.4 and uses OpenJDK 64-Bit Server VM 1.7.0_25 as Java environment.

Null Semantics. Real-world data often contains `null` values. So a schema $R(A, B)$ could hold the two records $r_1 = (\perp, 1)$ and $r_2 = (\perp, 2)$. Depending on whether we choose the semantics `null` = `null` or the semantics `null` $\neq$ `null`, the functional dependency $A \rightarrow B$ is `false` or `true` respectively. Hence, the `null` semantics changes the results of the FD discovery. Our algorithm HYFD supports both settings, which means that the semantics can be switched in the `Preprocessor` (PLI-construction) and in the `Sampler` (*match*()-function) with a parameter. For the experiments, however, we use `null` = `null`, because this is how related work treats `null` values [20].

Datasets. We evaluate HYFD on various synthetic and real-world datasets. Table 1 in Section 10.4 and Table 2 in

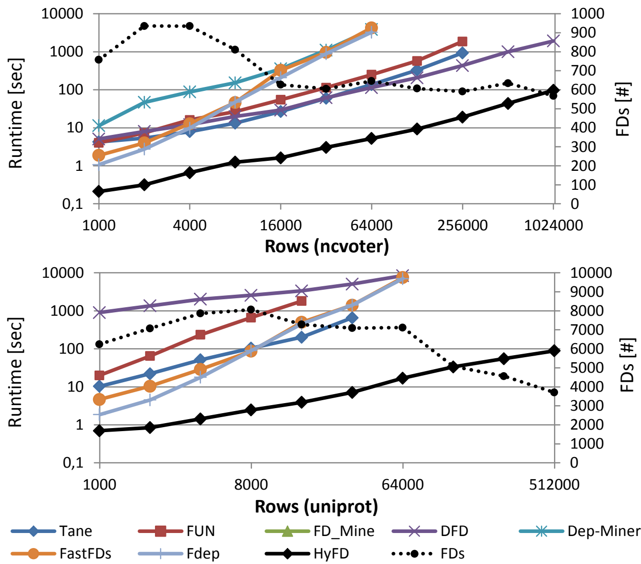

Figure 6: Row scalability on *ncvoter* and *uniprot*.

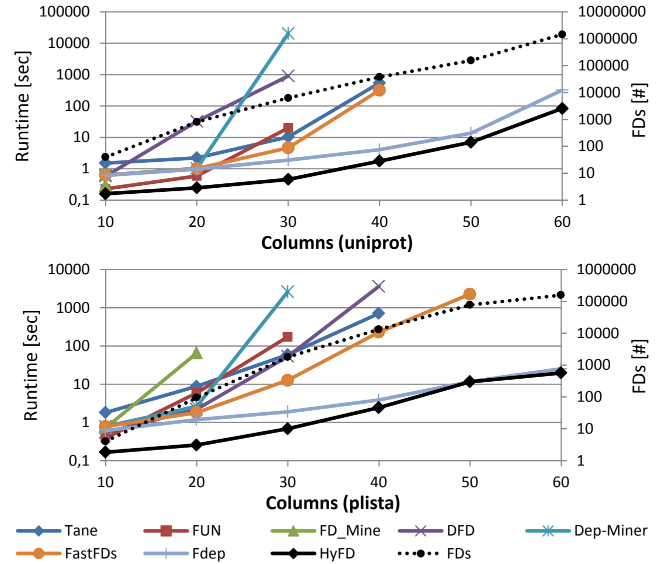

Figure 7: Column scalability on *uniprot* and *plista*.

Section 10.5 give an overview of these datasets. The data shown in Table 1 was already used in [20]. We also use the *plista* [13] dataset containing web log data, the *uniprot*[1] dataset storing protein sequences, and the *ncvoter*[2] dataset listing public voter statistics. The datasets listed in Table 2 have never been analyzed for FDs before, because they are much larger than the datasets of Table 1 and most of them cannot be processed with any of the related seven FD discovery algorithms within reasonable time (<1 month) and memory (<100 GB): The *CD* dataset contains CD-product data, the synthetic *TPC-H* dataset models business data, the *PDB* dataset stores protein sequence data, and the *SAP_R3* dataset holds data of a real SAP R3 ERP system.

10.2 Varying the number of rows

Our first experiment measures the runtime of HyFD on different row numbers. The experiment uses the *ncvoter* dataset with 19 columns and the *uniprot* dataset with 30 columns. The results, which also include the runtimes of the other seven FD discovery algorithms, are shown in Figure 6. A series of measurements stops if either the memory consumption exceeded 128 GB or the runtime exceeded 10,000 seconds. The dotted line shows the number of FDs in the input using the second y-axis: This number first increases, because more tuples invalidate more FDs so that more larger FDs arise; then it decreases, because even the larger FDs get invalidated and no further minimal specializations exist.

With our HyFD algorithm, we could process the 19 column version of the *ncvoter* dataset in 97 seconds and the 30 column version of the *uniprot* dataset in 89 seconds for the largest row size. This makes HyFD more than 20 times faster on *ncvoter* and more than 416 times faster on *uniprot* than the best state-of-the-art algorithm respectively. The reason why HyFD performs so much better than current lattice traversal algorithms is the fact that the number of FD-candidates that need to be validated against the many rows is greatly reduced by the `Sampler` component.

[1]www.uniprot.org

[2]www.ncsbe.gov/ncsbe/data-statistics

10.3 Varying the number of columns

In our second experiment, we measure HyFD's runtime on different column numbers using the *uniprot* dataset and the *plista* dataset with 1,000 records each. Again, we plot the measurements of HyFD with the measurements of the other FD discovery algorithms and cut the runtimes at 10,000 seconds. Figure 7 shows the result of this experiment.

We first notice that HyFD's runtime rather scales with the number of FDs, i.e., with the result size than with the number of columns. This is a desirable behavior, because the increasing effort is compensated by an also increasing gain. We further see that HyFD again outperforms all existing algorithms. The improvement factor is, however, smaller in this experiment, because the two datasets are with 1,000 rows so small that comparing all pairs of records, as Fdep does, is feasible and probably the best way to proceed. HyFD is still slightly faster than Fdep, because it does not compare all record pairs; the overhead of creating Plis is compensated by then being able to compare Pli compressed records rather than String-represented records.

10.4 Varying the datasets

To show that HyFD is not sensitive to any dataset peculiarity, the next experiment evaluates the algorithm on many different datasets. For this experiment, we set a time limit (TL) of 4 hours and a memory limit (ML) of 100 GB. Table 1 summarizes the runtimes of the different algorithms.

The measurements show that HyFD was able to process all datasets and that it usually performed best. There are only two runtimes, namely those for the *fd-reduced-30* and for the *uniprot* dataset, that are in need of explanation: First, the *fd-reduced-30* dataset is a generated dataset that exclusively contains random values. Due to these random values, all FDs are accidental and do not have any semantic meaning. Also, all FDs are of same size, i.e., 99% of the 89,571 minimal FDs reside on lattice level three and none of them above this level. Thus, bottom-up lattice traversal algorithms, such as Tane and Fun, and algorithms that have bottom-up characteristics, such as Dep-Miner and Fast-

Dataset	Cols [#]	Rows [#]	Size [KB]	FDs [#]	TANE [12]	FUN [18]	FD_MINE [25]	DFD [1]	DEP-MINER [16]	FASTFDS [24]	FDEP [9]	HYFD
iris	5	150	5	4	1.1	**0.1**	0.2	0.2	0.2	0.2	**0.1**	**0.1**
balance-scale	5	625	7	1	1.2	**0.1**	0.2	0.3	0.3	0.3	0.2	**0.1**
chess	7	28,056	519	1	2.9	1.1	3.8	1.0	174.6	164.2	125.5	**0.2**
abalone	9	4,177	187	137	2.1	0.6	1.8	1.1	3.0	2.9	3.8	**0.2**
nursery	9	12,960	1,024	1	4.1	1.8	7.1	0.9	121.2	118.9	46.8	**0.5**
breast-cancer	11	699	20	46	2.3	0.6	2.2	0.8	1.1	1.1	0.5	**0.2**
bridges	13	108	6	142	2.2	0.6	4.2	0.9	0.5	0.6	0.2	**0.1**
echocardiogram	13	132	6	527	1.6	0.4	69.9	1.2	0.5	0.5	0.2	**0.1**
adult	14	48,842	3,528	78	67.4	111.6	531.5	5.9	6039.2	6033.8	860.2	**1.1**
letter	17	20,000	695	61	260.0	529.0	7204.8	6.0	1090.0	1015.5	291.3	**3.4**
ncvoter	19	1,000	151	758	4.3	4.0	ML	5.1	11.4	1.9	1.1	**0.4**
hepatitis	20	155	8	8,250	12.2	175.9	ML	326.7	5576.5	9.5	0.8	**0.6**
horse	27	368	25	128,727	457.0	TL	ML	TL	TL	385.8	7.2	**7.1**
fd-reduced-30	30	250,000	69,581	89,571	**41.1**	77.7	ML	TL	377.2	382.4	TL	513.0
plista	63	1,000	568	178,152	ML	ML	ML	TL	TL	TL	26.9	**21.8**
flight	109	1,000	575	982,631	ML	ML	ML	TL	TL	TL	216.5	**53.4**
uniprot	223	1,000	2,439	>2,437,556	ML	ML	ML	TL	TL	TL	ML	>**5254.7**

Results larger than 1,000 FDs are only counted — **TL**: time limit of 4 hours exceeded — **ML**: memory limit of 100 GB exceeded

Table 1: Runtimes in seconds for several real-world datasets (extended from [20])

FDs, perform very well on such an unusual dataset. The runtime of HyFD, which is about 9 minutes, is an adequate runtime for any dataset with 30 columns and 250,000 rows.

The *uniprot* dataset is another extreme, but real-world dataset: Because it comprises 223 columns, the total number of minimal FDs in this dataset is much larger than 100 million. This is, as Figure 7 shows, due to the fact that the number of FDs in this dataset grows exponentially with the number of columns. For this reason, we limited HyFD's result size to 4 GB and let the algorithm's `Guardian` component assure that the result does not become larger. In this way, HyFD discovered all minimal FDs with a LHS of up to four attributes; all FDs on lattice level five and above have been successively pruned, because they would exceed the 4 GB memory limit. So HyFD discovered the first 2.5 million FDs in about 1.5 hours. One can compute more FDs on *uniprot* with HyFD using more memory, but the entire result set is – at the time – impossible to store.

The datasets in Table 1 brought all state-of-the-art algorithms to their limits, but they are still quite small in comparison to most real-world datasets. Therefore, we also evaluated HyFD on much larger datasets. This experiment reports only HyFD's runtimes, because no other algorithm can process the datasets within reasonable time and memory limits. Table 2 lists the results for the single-threaded implementation of HyFD (left column) that we also used in the previous experiments and a multi-threaded implementation (right column), which we explain below.

The measurements show that HyFD's runtime depends on the number of FDs, which is fine, because the increased effort pays off in more results. Intuitively, the more FDs are to be validated, the longer the discovery takes. But

Dataset	Cols [#]	Rows [#]	Size [MB]	FDs [#]	HyFD [s/m/h/d]	
TPC-H.lineitem	16	6 m	1,051	4 k	**39 m**	4 m
PDB.POLY_SEQ	13	17 m	1,256	68	**4 m**	3 m
PDB.ATOM_SITE	31	27 m	5,042	10 k	**12 h**	64 m
SAP_R3.ZBC00DT	35	3 m	783	211	**4 m**	2 m
SAP_R3.ILOA	48	45 m	8,731	16 k	**35 h**	8 h
SAP_R3.CE4HI01	65	2 m	649	2 k	**17 m**	10 m
NCVoter.statewide	71	1 m	561	5 m	**10 d**	31 h
CD.cd	107	10 k	5	36 k	**5 s**	3 s

Table 2: Single- and multi-threaded runtimes on larger real-world datasets.

the *CD* dataset shows that the runtime also depends on the number of rows, i.e., the FD-candidate validations are much less expensive if only a few values need to be checked. If both the number of rows and columns becomes large, which is when they exceed 50 columns and 10 million rows, HyFD might run multiple days. This is due to the exponential complexity of the FD-discovery problem. However, HyFD was able to process all such datasets and because no other algorithm is able to achieve this, obtaining a complete result within some days is the first actual solution to the problem.

Multiple threads. We introduced and tested a single-threaded implementation of HyFD to compare its runtime with the single-threaded state-of-the-art algorithms. HyFD can, however, easily be parallelized, because the comparisons in the `Sampler` component are like the validations in the `Validator` component independent of one another. We implemented these simple parallelizations and the runtimes reduced to the measurements shown in the right column of Table 2 running 32 parallel threads. Compared to the parallel FD discovery algorithm PARADE [10], HyFD is 8x (*POLY_SEQ*), 38x (*lineitem*), 89x (*CE4HI01*), and 1178x (*cd*) faster due to its novel, hybrid search strategy – for the other datasets, we stopped PARADE after two weeks.

10.5 In-depth experiments

Memory consumption. Many FD discovery algorithms demand a lot of main memory to store intermediate data structures. The following experiment contrasts the memory consumption of HyFD with its three most efficient competitors TANE, DFD, and FDEP on different datasets (the memory consumption of FUN and FD_MINE is worse than TANE's; DEP-MINER and FASTFDS are similar to FDEP [20]). To measure the memory consumption, we limited the available memory successively to 1 MB, 2 MB, ..., 10 MB, 15 MB, ..., 100 MB, 110 MB, ..., 300 MB, 350 MB, ..., 1 GB, 2 GB, ..., 10 GB, 15 GB, ..., 100 GB and stopped increasing the memory when an algorithm finished without memory issues. Table 3 lists the results. Note that the memory consumption is given for complete results and HyFD can produce smaller results on less memory using the `Guardian` component. Because DFD takes more than 4 hours, which is our time limit, to process *horse*, *plista*, and *flight*, we could not measure the algorithm's memory consumption on these datasets.

Dataset	TANE	DFD	FDEP	HYFD
hepatitis	400 MB	300 MB	9 MB	**5 MB**
adult	5 GB	300 MB	100 MB	**10 MB**
letter	30 GB	400 MB	90 MB	**25 MB**
horse	25 GB	-	100 MB	**65 MB**
plista	> 100 GB	-	800 MB	**110 MB**
flight	> 100 GB	-	900 MB	**200 MB**

Table 3: Memory consumption

Due to the excessive construction of PLIS, TANE of course consumes the most memory. DFD manages the PLIS in a PLI-store using a least-recently-used strategy to discard PLIS when memory is exhausted, but the minimum number of required PLIS is still very large. Also, DFD becomes very slow on low memory. FDEP has a relatively small memory footprint, because it does not use PLIS at all. HYFD uses the same data structures as TANE and FDEP and some additional data structures, such as the comparison suggestions, but it still has the overall smallest memory consumption: In contrast to TANE, HYFD generates much fewer candidates and requires only the single-column PLIS for its direct validation technique; in contrast to FDEP, it stores the non-FDs in bitsets rather than index lists and uses the PLIS instead of the original data for the record comparisons.

Efficiency threshold. HYFD requires a parameter that determines when Phase 1 or Phase 2 become inefficient: It stops the record matching in the `Sampler` component if less than x percent matches delivered new FD-violations and it stops the FD-candidate validations in the `Validator` component if more than x percent candidates have shown to be invalid. In the explanation of the algorithm and in all previous experiments, we set this parameter to 1% regardless of the datasets being analyzed. The following experiment evaluates different parameter settings on the *ncvoter_statewide* dataset with ten thousand records.

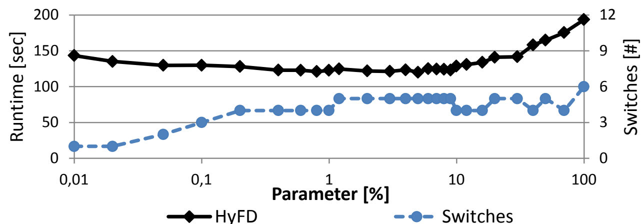

Figure 8: Effect of HyFD's only parameter on 10 thousand records of the *ncvoter_statewide* dataset.

The first line in Figure 8 plots HYFD's runtime for parameter values between 0.01% and 100%. It shows that HYFD's performance is not very sensitive to the efficiency threshold parameter. In fact, the performance is almost the same for any value between 0.1% and 10%. This is because the efficiency of either phase falls suddenly and fast so that all low efficiency values are met quickly: The progressive sampling identifies most matches very early and the validation generates many new, largely also invalid FD-candidates for every candidate tested as invalid.

However, if we set the parameter higher than 10%, then HYFD starts validating some lattice levels with too many invalid FD-candidates, which affects the performance negatively; if we, on the other hand, set the value lower than 0.1%, HYFD invests too much time on sampling than actually needed, which means that it keeps matching records although all results have already been found. We observed the same effects an different dataset, so we propose 1% as the default efficiency threshold for HYFD.

The second line in Table 8 depicts the number of switches from Phase 2 back into Phase 1 that HYFD made with the different parameter settings. We observe that four to five phase-switches are necessary on *ncvoter_statewide* and doing fewer or more switches is disadvantageous for the performance. Note that HYFD did these switches on different lattice-levels depending on the parameter setting, i.e., with low thresholds it switches earlier; with high thresholds later.

10.6 Result analysis

The number of FDs that HYFD can discover is very large. In fact, the size of the discovered metadata can easily exceed the size of the original dataset (see the *uniprot* dataset in Section 10.4). A reasonable question is, hence, whether complete results, i.e., *all* minimal FDs, are actually needed. Schema normalization, for instance, requires only a small subset of FDs to transform a current schema into a new schema with smaller memory footprint. Data integration also requires only a subset of all FDs, namely those that overlap with a second schema. In short, most use-cases for FDs indeed require only a subset of all results.

However, one must inspect all functional dependencies to identify these subsets: Schema normalization, for instance, is based on closure calculation and data integration is based on dependency mapping, both requiring complete FD result sets to find the optimal solutions. Furthermore, in query optimization, a subset of FDs that optimizes a given query workload by 10% is very good at first sight, but if a different subset of FDs could have saved 20% of the query load, one would have missed some high optimization potential. For these reasons and because we cannot know which other use cases HYFD will have to serve, we discover all functional dependencies – or at least as many as possible.

11. CONCLUSION & FUTURE WORK

In this paper, we proposed HYFD, a hybrid FD discovery algorithm that discovers all minimal, non-trivial functional dependencies in relational datasets. Because HYFD combines row- and column-efficient discovery techniques, it is able to process datasets that are both long and wide. This makes HYFD the first algorithm that can process datasets of relevant real-world size, i.e., datasets with more than 50 attributes and a million records. On smaller datasets, which some other FD discovery algorithms can already process, HYFD offers the smallest memory footprints and the fastest runtimes; in many cases, our algorithm is orders of magnitude faster than the best state-of-the-art algorithm. Because the number of FDs grows exponentially with the number of attributes, we also proposed a component that dynamically prunes the result set, if the available memory is exhausted.

A task for future work is the development of use-case-specific algorithms that leverage FD result sets for schema normalization, query optimization, data integration, data cleansing, and many other tasks. In addition, knowledge of the use-case might help develop specific semantic pruning rules to further speed-up detection. The only reasonable semantic pruning we found was removing FDs with largest left-hand-sides, because these are most prone to being accidental, and we only apply it when absolutely necessary.

Acknowledgements. We thank Tobias Bleifuß for the idea of compressing non-FDs as bitsets and the authors of [20] and [10] for providing code and data for our comparative evaluation.

12. REFERENCES

[1] Z. Abedjan, P. Schulze, and F. Naumann. DFD: Efficient functional dependency discovery. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 949–958, 2014.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 487–499, 1994.

[3] P. Bohannon, W. Fan, and F. Geerts. Conditional functional dependencies for data cleaning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 746–755, 2007.

[4] C. R. Carlson, A. K. Arora, and M. M. Carlson. The application of functional dependency theory to relational databases. *Computer Journal*, 25(1):68–73, 1982.

[5] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[6] E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971.

[7] G. Cormode, L. Golab, K. Flip, A. McGregor, D. Srivastava, and X. Zhang. Estimating the confidence of conditional functional dependencies. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 469–482, 2009.

[8] S. S. Cosmadakis, P. C. Kanellakis, and N. Spyratos. Partition semantics for relations. *Journal of Computer and System Sciences*, 33(2):203–233, 1986.

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

[10] E. Garnaud, N. Hanusse, S. Maabout, and N. Novelli. Parallel mining of dependencies. In *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*, pages 491–498, 2014.

[11] M. A. Hernández and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[12] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[13] B. Kille, F. Hopfgartner, T. Brodt, and T. Heintz. The plista dataset. In *Proceedings of the International Workshop and Challenge on News Recommender Systems*, 2013.

[14] W. Li, Z. Li, Q. Chen, T. Jiang, and H. Liu. Discovering functional dependencies in vertically distributed big data. *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*, pages 199–207, 2015.

[15] J. Liu, J. Li, C. Liu, and Y. Chen. Discover dependencies from data – a review. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(2):251–264, 2012.

[16] S. Lopes, J.-M. Petit, and L. Lakhal. Efficient discovery of functional dependencies and Armstrong relations. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 350–364, 2000.

[17] R. J. Miller, M. A. Hernandez, L. M. Haas, L.-L. Yan, H. Ho, R. Fagin, and L. Popa. The Clio project: Managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

[18] N. Novelli and R. Cicchetti. FUN: An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 189–203, 2001.

[19] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data profiling with metanome. *Proceedings of the VLDB Endowment*, 8(12):1860–1871, 2015.

[20] 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. *Proceedings of the VLDB Endowment*, 8(10):1082–1093, 2015.

[21] T. Papenbrock, A. Heise, and F. Naumann. Progressive duplicate detection. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 27(5):1316–1329, 2015.

[22] G. N. Paulley. Exploiting functional dependence in query optimization. Technical report, University of Waterloo, 2000.

[23] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990.

[24] C. Wyss, C. Giannella, and E. Robertson. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In *Proceedings of the International Conference of Data Warehousing and Knowledge Discovery (DaWaK)*, pages 101–110, 2001.

[25] H. Yao, H. J. Hamilton, and C. J. Butz. FD_Mine: discovering functional dependencies in a database using equivalences. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 729–732, 2002.