DFD: Efficient Functional Dependency Discovery

Ziawasch Abedjan
Hasso Plattner Institute
Potsdam, Germany
ziawasch.abedjan@hpi.uni-potsdam.de

Patrick Schulze
Hasso Plattner Institute
Potsdam, Germany
patrick.schulze@student.hpi.uni-potsdam.de

Felix Naumann
Hasso Plattner Institute
Potsdam, Germany
felix.naumann@hpi.uni-potsdam.de

ABSTRACT

The discovery of functional dependencies in a dataset is of great importance for database redesign, anomaly detection and data cleansing applications. However, as the nature of the problem is exponential in the number of attributes none of the existing approaches can be applied on large datasets. We present a new algorithm DFD for discovering all functional dependencies in a dataset following a depth-first traversal strategy of the attribute lattice that combines aggressive pruning and efficient result verification. Our approach is able to scale far beyond existing algorithms for up to 7.5 million tuples, and is up to three orders of magnitude faster than existing approaches on smaller datasets.

Categories and Subject Descriptors

H.2.0 [**Database Management**]: General

Keywords

functional dependency, algorithms, lattice traversal

1. INTRODUCTION

Functional Dependencies (FDs) are among the most relevant dependencies in relational databases. Despite their relevance for normalization of relations, FD constraints gained importance for data quality and data cleaning applications. In particular, they can resemble integrity constraints [5] or denial constraints [4], which can be used to detect and repair data violations.

The discovery of functional dependencies is a popular topic in the domain of data profiling [15]. Foundations regarding the general complexity of the problem, which is in $\Omega(2^m)$ for m columns, were proven by Mannila et al. [14] in 1992. Based on those principles, several major papers concerning FD discovery were published between 1999 and 2002. However, none of the approaches is applicable to realistic datasets with tens of attributes and millions of tuples. While further effort was put into the application area of FDs [4,7], by extending the FD concept to more specialized constraints, such as approximate FDs and conditional FDs, research on the efficiency of FD discovery approaches has been neglected.

Recent advances in the related domain of unique column combinations (composite keys) [10,17], however, showed that it is possible to achieve superior runtime behavior by aggressive pruning of the search space. We adapt the insights gained from that line of research and create a new efficient approach for the discovery of FDs in a relational table. Next, we formally define relevant concepts and the computational challenge of FD discovery and outline the contributions of this paper.

Functional Dependencies. Given a relational database schemata $R = \{C_1, C_2, \ldots C_n\}$ with a relational instance $r \subseteq dom(C_1) \times dom(C_2) \times \cdots \times dom(C_n)$, a functional dependency states that an attribute $C_k \in R$ is determined by some other attributes $X \subset R$. This can be formally defined as follows:

DEFINITION 1.1. ***Functional Dependency*** *(FD). Given a relational database schema R with a relational instance r. Let $C_k \in R$ and $X \subseteq R$, then $X \rightarrow C_k$ is a functional dependency over r iff*

$$\forall t_1, t_2 \in r : t_1[X] = t_2[X] \Rightarrow t_1[C_k] = t_2[C_k]$$

X is called the left-hand side (LHS) and C_k is the right-hand side (RHS) of the dependency.

Accordingly we define a non-functional dependency (*non*-FD) as follows:

DEFINITION 1.2. ***Non-Functional Dependency*** *(Non-FD) Let $C_k \in R$ and $X \subseteq R$, then $X \not\rightarrow C_k$ is a non-FD over r iff*

$$\exists t_1, t_2 \in r : t_1[X] = t_2[X] \wedge t_1[C_k] \neq t_2 C_k]$$

Naturally, for any $C_k \in X$ the trivial FD $X \rightarrow C_k$ holds and can be ignored during the discovery process. Furthermore, the set of all FDs can be generated based on the set of all minimal FDs, which are defined as follows:

DEFINITION 1.3. ***Minimal FD.*** *With $C_k \in R$ and $X, X' \subseteq R$ the FD $X \rightarrow C_k$ is minimal iff $\forall X' \subset X : X' \not\rightarrow C_k$.*

The maximality of a non-FD can be defined accordingly:

DEFINITION 1.4. ***Maximal non-FD.*** *With $C_k \in R$ and $X, X' \subseteq R$ the FD $X \not\rightarrow C_k$ is a maximal non-FD iff*

$$\forall X' \supset X : X' \rightarrow C_k \quad .$$

In general, the task of discovering all FDs can be reduced to the task of discovering all minimal FDs and all maximal non-dependencies. While for any FD $X \rightarrow C_k$, further non-minimal FDs exist by appending the LHS with further columns, for any non-FD the RHS is also non-dependent

CIKM'14, November 3–7, 2014, Shanghai, China.

http://dx.doi.org/10.1145/2661829.2661884.

from all subsets of the non-dependency's Lhs. In this paper, we show that similar to the key discovery problem [17], it is possible to leverage the complement relationship between the set of minimal Fds and the set of maximal non-dependencies.

Contributions. We make the following contributions:

1. With Dfd we present a new Fd discovery approach for very large datasets, significantly outperforming previous work.
2. Extensive experiments on multiple real world datasets compare Dfd to existing approaches and analyze its runtime behavior.

The next section discusses and compares related work. In Section 3 we present our approach Dfd. In Section 4 we evaluate our approach and conclude in Section 5.

2. RELATED WORK

Functional dependency discovery approaches can be classified into two main groups. The first group consists of algorithms that traverse the search space in a breadth-first, levelwise manner. The strategies of this group can be further divided into bottom-up, top-down, and hybrid approaches. The second class is represented by algorithms that traverse the search space depth-first.

The group of breadth-first levelwise algorithms consists of Tane [11], FUN [16], FDMine [19], and DepMiner [13]. They all use the Apriori algorithm for candidate generation [2]. The key characteristic of the breadth-first bottom-up approach is that it works levelwise, whereas each level k contains only Lhs candidates for Fds with a cardinality of k attributes. The results of each level are used to generate the left-hand side candidates of size $k+1$. Such breadth-first approaches work well on datasets that contain Fds on lower levels, where all minimal Fds can be detected early and all of their supersets can be pruned (see Def. 1.3). However, if some minimal Fds have high cardinalities, i.e., involve many LHS attributes, the algorithm has to check the candidates of all levels until the highest level with a minimal Fd is reached. Furthermore, if some columns are never part of a minimal Fd, the algorithms might need to generate all supersets until an Fd is discovered. While we use some techniques that have already been proposed by Tane, we present a new traversal approach that avoids the verification of unnecessary column combinations by flexibly traversing both bottom-up and top-down in a depth-first manner.

Tane, FDMine, and FUN all rely on quite similar concepts. DepMiner follows a different approach than the classical breadth-first approaches: It calculates covers of so-called agree sets to determine minimal dependencies in the underlying dataset. DepMiner generates the candidates in a breadth-first levelwise manner, which still leads to an exponential runtime behavior depending on the number of columns. Furthermore, the calculation of agree sets has a quadratic time complexity depending on the number of tuples in the dataset. Despite the improved strategies to generate the agree sets by Lopes et al. [13], this preprocessing step still has a significant impact on the runtime, because nowadays datasets with millions of tuples are common. Our approach uses the partition refinement approach proposed in [11], which scales linearly in the number of tuples.

The second group of approaches are algorithms that search the powerset lattice of the attributes in a depth-first manner. For a given Rhs column, they start with an individual column as Lhs and append more columns until an Fd for the selected Rhs holds. Depth-first traversal algorithms have the advantage that they can identify the first dependency within n steps for n attributes. Although this approach suffers from the fact that the identified Fd is not necessarily minimal, it has the advantage that for any discovered non-Fd Lhs on the path to the discovered Fd all of its subsets can be pruned, because they also constitute non-Fd Lhss. This category is represented by FastFDs [18], a tuple-oriented algorithm. It features many similarities with DepMiner [13]. Candidates for Lhss are selected based on difference sets, which are similar to the agree sets of DepMiner. A difference set of two tuples contains all attributes of the relational schema in which the values of the tuples are distinct. Therefore, difference sets are the complements of agree sets. Similar to DepMiner [13] the computational complexity of this preprocessing step is quadratic in the number of tuples. Additionally, the number of possible covers grows exponentially with the number of attributes in the difference sets. Therefore, also the determination of minimal covers can be a costly processing step.

Flach et al. propose FDep, which uses machine learning techniques to infer functional dependencies based on the tuples of the underlying dataset [8]. Besides a naïve brute-force approach, the authors provide a top-down, bottom-up, and a bi-directional version of FDep. However, Wyss et al. have shown that FDep only performs well on small datasets [18]. FDep has the same drawbacks as FastFDs [18] and DepMiner [13], because it is based on pairwise tuple comparison.

In Section 4, we present a detailed evaluation that shows the effective pruning strategy of our algorithm Dfd in a comparative study with the most popular approaches Tane and FastFDs. After 2002 there were no papers that further improved the performance of Fd discovery. Instead, the research focus shifted from functional dependencies to approximate [9, 12] and conditional functional dependencies [3, 6].

3. THE DFD ALGORITHM

DFD is a novel algorithm for the Discovery of Functional Dependencies. It recombines components from known approaches for the discovery of Fds and the discovery of unique column combinations. First, we describe our search strategy for efficient identification of relevant column combinations and then we describe data structures that ensure fast pruning and fast verification of Fd candidates.

3.1 Leveraging Unique Column Combinations

For each relation, a powerset lattice can be constructed that represents all column combinations, as shown in Figure 1. To identify all existing Fds one has to traverse that lattice, pruning as many combinations as possible. Tane, for example, does this verification levelwise, which means for each $k = 2, 3, .., |R|$ it checks all combinations of size k at the same time beginning with $k = 2$. For each column combination it checks which $k-1$ subsets functionally determine the remaining column. Whenever an Fd is discovered, Tane ignores all possible Fds with a Lhs superset of the current Fd. For large k the generation of all possible candidates is a bottleneck. Especially for datasets where a small number of minimal Fds exist at relatively high levels

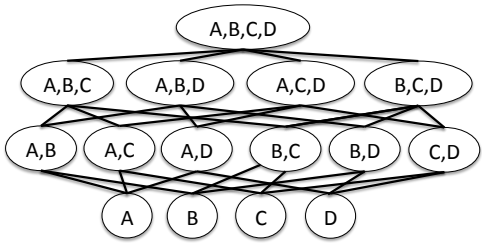


Figure 1: Powerset lattice for R=(A,B,C,D)

of the lattice, the breadth-first traversal of Tane results in a significant overhead for candidate generation.

We radically change the traversal of the lattice based on insights from the discovery of unique column combinations. Although, the problem of discovering all minimal unique column combinations is easier to solve than the problem of FD discovery, we show that some advances in that field can greatly contribute to the FD discovery problem. In particular, our approach is inspired by the algorithms [10] and [17].

In general both concepts, unique column combinations and FDs, are strongly related: functional dependencies can be seen as a generalization of the key concept for a relation. A unique column combination $K \subseteq R$ does not contain any duplicate tuples [1]. Minimal unique column combinations are valid LHSs of minimal FDs that functionally determine values in all remaining columns $R \setminus K$. Thus, there are at least as many minimal FDs in a dataset as there are minimal unique column combinations. In fact, unique discovery is a subproblem of determining all FDs of a dataset. One major difference between unique column combinations and FDs is that for discovering uniques we need to consider only column combinations as a whole. In contrast, FDs consist of LHS and a RHS column combination. Current algorithms manage this issue by considering all possible LHSs and RHSs when checking a column combination. For breadth-first approaches like Tane this is straight forward, because in any case only LHSs of a certain size are considered at each level. However, traversing the lattice in a depth-first manner requires to identify decidable paths. If one node provides multiple dependencies to check, the decision for the next traversal step might not be intuitively clear and result in multiple stacks of paths that have to be traced later.

To disentangle this situation, we decompose the lattice of all column combinations into multiple lattices, where each corresponds to one possible RHS of the relation. Having a relation $R = (A, B, C, D)$ for example, we create for the RHS A the lattice that contains the subsets of the powerset of $\{B, C, D\}$ to identify all LHSs that functionally determine the column A. For our example R we create four lattices for the four possible RHSs. In each lattice we can apply pruning for supersets of valid FD LHSs and subsets of non-FD LHSs. Iterating one by one over the RHS attributes, DFD is not only easier to understand, it also allows us to discard any redundant pruning data structures after each iteration step, as we show in the following.

3.2 Algorithm Workflow

Algorithm 1 illustrates the main loop of DFD. Its input is the set of columns and their partitions. We describe the structure of these partitions in Section 3.7. They directly represent the data of the single columns of the underlying dataset. In lines 1 to 4, DFD checks each attribute partition for uniqueness. Finding a unique attribute enables DFD to infer a set of minimal FDs without traversing the lattice. Each remaining attribute is then considered as a possible RHS and *findLHSs* is called to create the corresponding powerset lattice and identify possible LHSs.

Algorithm 1: The main loop of DFD.

```
   Data: All attributes A ∈ R, relational instance r
   Result: the set of minimal non-trivial functional
           dependencies F_r
1  foreach A ∈ R do
2      if A is unique then
3          R ← R \ {A};
4          add (A → R) to F_r;
5  foreach RHS ∈ R do
6      F_r ← F_r ∪ {K → RHS' | K ∈ findLHSs(RHS, r)}
7  return F_r
```

Algorithm 2 shows how DFD determines all minimal FDs for the current RHS attribute by classifying all possible LHSs. In line 1, DFD picks randomly a seed as an initial LHS candidate. Initially the individual columns except the current RHS candidate constitute the seeds. Subsequently, DFD starts to traverse the search space beginning with the chosen LHS and classifies each visited combination/node according to the following categories:

- dependency
- minimal dependency
- candidate minimal dependency
- non-dependency
- maximal non-dependency
- candidate maximal non-dependency

Dependency and *non-dependency* represent LHS nodes that are neither minimal nor maximal, respectively. A *candidate* minimal dependency/maximal non-dependency is a column combination that still can be the LHS of a minimal FD or maximal non-FD. In case the current column combination was already traversed at an earlier stage of the process, DFD examines only those column combinations again that are classified as candidates (line 6). In lines 8 and 11, DFD checks whether the category of a candidate can be changed or not. It could be that after revisiting a candidate minimal dependency all of the subsets of that node have been classified as non-dependency, making the candidate a minimal dependency. That is why DFD maintains the visited candidate nodes in a stack trace. The trace allows DFD to backtrack its way through the lattice, revisiting nodes that can be eventually classified at a later stage.

In case the node was not yet visited, DFD examines if the node is a proper superset/subset of a previously discovered dependency/non-dependency and updates its category accordingly. Otherwise, it has to perform the costly partition calculation in order to identify which kind of candidate the combination represents (line 17). If the partition calculation results into a dependency, the node is classified as a candidate minimal dependency, otherwise as a candidate maximal non-dependency.

After categorizing the current node, DFD chooses the next node that is visited during the lattice traversal in line 18.

Algorithm 2: *findLHSs*()

```
   Data: RHS attribute A, relational instance r
   Result: the set of minimal non-trivial Lhss
1  seeds ← R \ {A};
2  while !isEmpty(seeds) do
3      node ← pickSeed();
4      repeat
5          if visited(node) then
6              if isCandidate(node) then
7                  if isDependency(node) then
8                      if isMinimal(node) then
9                          add (node) to minDeps;
10                 else
11                     if isMaximal(node) then
12                         add (node) to maxNonDeps;
13                 updateDependencyType(node);
14             else
15                 inferCategory(node);
16                 if category(node) is null then
17                     computePartitions(node, A);
18             node ← pickNextNode(node);
19     until node is null ;
20     seeds ← generateNextSeeds();
21 return minDeps
```

Algorithm 3: *pickNextNode()*

```
   Data: node
   Result: nextNode
1  if isCandidateMinimalDep(node) then
2      S ← uncheckedSubsets(node);
3      P ← prunedSets(node);
4      S ← S \ P;
5      if S is empty then
6          add node to minDeps;
7      else
8          nextNode ← random(S);
9          push node to trace;
10         return nextNode;
11 else if isCandidateMaximalNonDep(node) then
12     S ← uncheckedSupersets(node);
13     P ← prunedSupersets(node);
14     S ← S \ P;
15     if S is empty then
16         add node to maxNonDeps;
17     else
18         nextNode ← random(S);
19         push node to trace;
20         return nextNode;
21 else
22     return nextNode from trace;
23
```

The general idea of choosing the next node is to move downwards in the lattice if the current column combination represents a dependency Lhs and upwards otherwise. The detailed process determining the next node is shown in Section 3.3. When all reachable nodes from the current trace have been visited, Dfd picks the next seed from the list of seeds that were initially calculated. Whenever the list of seeds is empty, the algorithm calls *generateNextSeeds()* to generate possibly remaining candidates.

3.3 Traversal Step

Algorithm 3 shows how Dfd picks the next node based on its stack-trace and the currently considered column combination. The first important step is to determine if the current combination is a candidate for a minimal dependency or maximal non-dependency. If not, the algorithm instantly backtracks by choosing the latest available node on the trace.

If the current node is still a candidate for a minimal dependency, Dfd needs to determine if it is actually minimal. Therefore, in line 2 Dfd retrieves all unchecked subsets of the current node and removes all subsets that can be pruned according to Def. 1.3. If the set of unchecked subsets is empty, then the current node is a Lhs for a valid minimal Fd (lines 5 and 6). This is because there are no subsets left that still can be a Lhs for a valid Fd with fewer attributes than the current node.

However, if the set of unchecked subsets is not empty, Dfd picks an unchecked subset as the next node in line 8, and adds the previous node to the trace.

In case of the current node being a candidate for a maximal non-dependency, Dfd retrieves the set of unchecked supersets and removes the prunable supersets according to Def. 1.4. Those nodes are potential Lhss for maximal non-dependencies that contain more attributes than the current node.

Example 3.1. *Given the schema $R = \{A, B, C, D, E\}$, assume the current right-hand side attribute is E and the current examined node is $\{A, C\}$. DFD categorized $\{A, C\}$ as a candidate for a maximal non-dependency. To determine whether the left-hand side is actually a valid maximal non-dependency for the current right-hand side, DFD needs to make sure that there are no supersets of $\{A, C\}$ that are also non-dependencies. The unchecked supersets of the current node are $\{A, B, C\}$ and $\{A, C, D\}$.*

Assume $\{C, D\} \rightarrow E$ is a valid functional dependency that was already found by Dfd*. In this case, the right-hand side $\{A, C, D\}$ can be excluded from the set of potential next nodes, because it already can be classified as a dependency.*

3.4 Identifying unclassified nodes

The lattice traversal terminates whenever there is no reachable column combination left. As previously mentioned, the aggressive pruning approach can lead to unreachable but not finally categorized candidate nodes. To cope with that problem, we again make use of insights from unique discovery algorithms. This time we leverage the complement operation that was introduced for the Gordian algorithm [17]. Sismanis et al. show that having the set of all maximal non-unique column combinations, it is possible to generate all minimal unique column combinations through a complementation approach. Heise et al. prove the reversibility of that operation and its feature to identify missing combinations if one of the sets is not complete [10]. Thus, to prove the correctness of Dfd we need to show that the correctness proof given by Heise et al. can be generalized to Fds. The only characteristics relevant for the proof are the minimality of uniques and maximality of non-uniques. For a Rhs column A, the set of

Algorithm 4: *generateNextSeeds*()

```
   Data: minimal dependencies for the current RHS
         minDeps
   maximal non-dependencies for the current RHS
   maxNonDeps
   Result: new seeds
1  seeds ← {}
2  newSeeds ← {}
3  foreach maxNonDep ∈ maxNonDeps do
4      maxNonDep' ← complement of maxNonDep;
5      if seeds is empty then
6          emptyColumns ← Bitset(| maxNonDep |);
7          foreach setBitIndex ∈ maxNonDep' do
8              add emptyColumns.setCopy(setBitIndex) to
               seeds;
9      else
10         foreach dep ∈ seeds do
11             foreach setBitIndex ∈ maxNonDep' do
12                 add dep.setCopy(setBitIndex) to
                   newSeeds;
13         minimizedNewDeps ← minimize newSeeds;
14         seeds ← {};
15         foreach newSeed ∈ minimizedNewDeps do
16             add newSeed to seeds;
17         newSeeds ← {};
18 seeds ← seeds \ minDeps;
19 return seeds;
```

all Lhss for minimal Fds that determine A and the set of all maximal column combinations that constitute a non-Fd for A have the same characteristics as the set of all minimal unique and maximal non-unique column combinations, respectively.

Algorithm 4 describes how to identify the untouched nodes of the Lhs lattice graph. Whenever the lattice traversal for the current right-hand side column terminates in Algorithm 2, Dfd has to determine whether all minimal functional dependencies for the current Rhs have been found. Algorithm 4 is able to detect missing Lhss for the current Rhs. Furthermore, it provides new seeds that enable Dfd to classify missing nodes.

Algorithm 4 allocates two data structures, *seeds* and *newSeeds* that maintain found seeds and the new set of seeds for the current iteration. In line 3, Dfd iterates over the whole set of maximal non-dependencies. For each of those non-Fd Lhs combinations, Dfd determines the complement. Because we use bitsets to store the set of columns in a column combination, this is a very cheap operation.

In the first iteration of the outer loop, *seeds* in line 5 is still empty. In that case, Dfd creates a column combination bitset called *emptyColumns*, which has the same size as the number of columns provided by the underlying dataset without the Rhs column. All of its bits have been set to zero, which means that the combination contains no columns yet.

Subsequently, Dfd adds a new potential Lhs to the set of *seeds* for each set bit in the complement of the current maximal non-dependency combination. Dfd finally removes the already determined Lhs of minimal Fd from the set of new seeds in *newSeeds*. The remaining set contains unclassified seed nodes that can be used for the next traversal iteration. In general those nodes are very close to the actual Lhs of minimal Fds and maximal non-dependencies so that the remaining search space is effectively narrowed down.

Example 3.2 illustrates why the approach of Sismanis et al. [17] for detecting missing keys from maximal non-keys, also works for minimal dependencies and maximal non-dependencies.

Example 3.2. *Given the schema $R = \{A, B, C, D, E, F\}$, assume the current right-hand side is F and the lattice traversal terminated.* Dfd *was not able to determine any minimal dependency, but it identified $\{A, B, C\} \not\rightarrow F$ as a maximal non-dependency.*

Using that result, lines 5 to 9 of Algorithm 4 generate the seeds D and E based on $\{A, B, C\}' = \{\{D\}, \{E\}\}$ as candidates for dependencies. This is because both of those attributes are not part of the maximal non-dependency, which implies that they have to be part of a minimal dependency: if they were not a component of a minimal dependency, they would have to be a part of the present maximal non-dependency Lhss. *However, in that case $\{A, B, C\} \not\rightarrow F$ would not be a maximal non-dependency anymore. Therefore, D and E can be used as seeds for the missing minimal dependency.*

If we now extend the example with a second maximal non-dependency $\{A, C, D\} \not\rightarrow F$, we can explain what happens in lines 9 to 17 of Algorithm 4. We know already that $seeds = \{D, E\}$ from the previous step. Dfd *now simply merges the dependencies of the first step with the complements of the remaining maximal non-dependency in the same manner as in the first part of the algorithm. Because $\{A, C, D\}' = \{B, E\}$,* Dfd *generates all cross combinations $\{\{B, D\}, \{D, E\}, \{B, E\}, \{E\}\}$ as new dependency candidates. After the minimization step in line 13, only $\{\{B, D\}, \{E\}\}$ remain as seeds for* Lhss.

3.5 Managing LHS of dependencies and non-dependencies

The pruning of supersets of dependency Lhs and subsets of non-dependency Lhs are a crucial factor for the efficiency of Dfd, because it enables Dfd to classify column combinations without calculating their partition, based only on the available dependencies and non-dependencies. Therefore, we need to provide a data structure that enables us to quickly identify classified supersets and subsets.

Similar to [10], we use two hashmaps to manage subset/ superset relationships. For the case of dependency Lhss, we use a hash map whose keys are represented by the individual column indices. Those columns map to sets of dependency Lhss that contain the respective column. Whenever Dfd discovers a dependency with a Lhs K, that dependency is added to the value sets of all columns that are contained in K. Since it is redundant to store supersets of Lhs of valid Fds, we minimize the sets of dependencies afterwards by removing previously stored supersets of K. The same is valid for non-Fds, albeit it is necessary to remove redundant subsets of non-Fd Lhs after adding a new combination.

However, as shown in [14], the search space of the complete set of minimal Fds and consequently for maximal non-Fds is exponential, and the set of stored dependencies and non-dependencies increases very fast during the traversal of the powerset lattice. That is why we rebalance those data

structures after adding new combinations by creating sublists for column pairs. The threshold for rebalancing the dependency and non-dependency sets can be adapted specifically to certain data sources. In general, the minimization of dependency sets or respectively the maximization of non-dependency sets in combination with the rebalancing allows DFD to reduce the lookup times. Whenever DFD performs a check whether a column combination can be pruned, it only needs to compare the current column combination to the associated combinations that have been stored for the columns of the current combination in the respective hashmap. In our experiments, we chose to set the maximum value set size to 100,000, which triggers the rebalancing process for some of the tested datasets.

3.6 Dependency validation and partition calculation

Similar to Tane, we use stripped partitions for DFD to validate dependencies [11]. Stripped partitions have also been referred to as position list indices [10].

DEFINITION 3.1. ***Partition of an attribute set.*** *Given a relational database schema $R = \{C_1, \dots C_n\}$ with a database instance $r \subseteq dom(C_1) \times \cdots \times dom(C_n)$, let $X \subseteq R$. Then the partition π associated with X is defined as $\pi_X = \{[a]_X \mid a \in r\}$.*

A stripped partition $\widehat{\pi_X}$ for a column combination X contains a list of equivalence classes where each class contains the row/position ids of one specific value projection in X and all classes with only one element have been removed. Furthermore, $||\widehat{\pi_X}||$ is the total number of duplicate value combinations and $|\widehat{\pi_X}|$ the number of equivalence classes.

DEFINITION 3.2. ***Validity check of FDs.*** *Given a relational database schema $R = \{C_1, \dots C_n\}$ with a database instance $r \subseteq dom(C_1) \times \cdots \times dom(C_n)$, let $X, Y \subseteq R$ and let $|\pi|$ be the number of equivalence classes in π.*
Then $X \to Y \Leftrightarrow |\pi_X| = \left|\pi_{X \cup \{Y\}}\right|$.

The calculation of $\pi_{X \cup \{Y\}}$ from π_X and π_Y is called partition intersection. This means, in order to determine whether a LHS X is valid for a right-hand side C_k in the form of $X \to C_k$, DFD needs to calculate not only the stripped partition of the LHS, but also the respective partition of the LHS intersected with the partitions of the RHS.

Computing the intersection of two stripped partitions scales linearly in the number of tuples of a relation. We use probe tables to intersect partitions of two attribute sets $\widehat{\pi_X}$ and $\widehat{\pi_Y}$, as also applied for Tane and DUCC. A probe table is a data structure that maps each tuple index $t \in r$ of the first input partition to its corresponding equivalence class index $c \in \widehat{\pi_X}$. In a second path on the tuples in $\widehat{\pi_Y}$, a second map is created that maps pairs of class indexes to sets of tuple ids. Each tuple id from $\widehat{\pi_Y}$ is probed against the previously created map. Whenever the lookup succeeds a map-entry is created with the classes $c \in \widehat{\pi_X}$ and $c' \in \widehat{\pi_X}$ as the key and the tuple id that matched both classes as value. If the key (c, c') already exists the new tuple is just appended to the list of values of (c, c'). In a final step, we strip the second map from all class pairs that map only to one tuple id.

Because we use the same concept as Tane for determining the validness of FDs, DFD can easily be modified to discover approximate FDs. Approximate or also called partial FDs are dependencies where some tuples violate the dependency. For example data corruption might be a reason why specific column combinations are not discovered as dependencies. Therefore it might be reasonable to consider a dependency also as valid, when it holds only for 99.99% of tuples.

3.7 Partition removal

In general, when calculating composed partitions, most of the time DFD re-uses previously computed partitions over and over again. Nevertheless, the number of partitions might still grow exponentially and result in high memory consumption.

Therefore, we provide a method to deallocate partitions that are not needed any longer. Since we cannot be absolutely sure which partitions are not necessary anymore, we keep track of the recently used partitions and the usage counts. Whenever the number of partitions exceeds a certain threshold we determine the median usage count of the currently allocated partitions. We then delete all non-atomic partitions that have a usage count below the median value, giving least recently used partitions an advantage.

The difficulty is to determine the threshold for the start of that process. The reason is that the structure of datasets, such as the distinctiveness count of the columns, the column and the row count itself, and the number of FDs lead to a different number of partitions that we need to calculate. One possibility is to trigger the partition removal operation whenever a static threshold is exceeded. However, that is not suitable since the number of partitions between different datasets for a constant number of rows and columns differs by orders of magnitude. It can be more practical to connect the partition removal process to the currently allocated memory by the partitions. In our experimental section, we show the effect of static partition removal on some dataset. We label the DFD version with partition removal as DFD-Mem.

4. EXPERIMENTS

To show the effective pruning and overall efficiency of our approach we performed multiple experiments. We analyzed the effect of different parameters, such as number of columns and number of rows, on the runtime behavior of DFD. We provide a detailed comparison of DFD with the most popular FD mining algorithms Tane and FastFDs, which represent different lines of FD discovery strategy.

4.1 Experimental Setup

For the experiments we used a Dell PowerEdge R620 server:

- 4 x Intel Xeon Processor E5-2650 @ 2.00GHz, 20MB Cache (32 CPU cores)
- 128GB DDR3-RAM with 1600 MHz
- CentOS 6.4, OpenJDK 64-Bit Server VM – 1.7.025

If not stated otherwise, all tests are under a maximum heap size of 64GB and a maximal running time of 3 hours.

Name	Type	#Columns	#Rows
Uniprot	real-world	223	539,165
NCVoter	real-world	94	7,503,575
lineitem	synthetic	16	6,001,215

Table 1: Datasets

Table 1 shows our datasets, selected based on the type, the column, and row count. The Uniprot dataset is the public Universal Protein Resource database(UniProt, `www.uniprot.org`) that contains protein sequences and functional information. NCVoter is a collection of North Carolina's Voter Registration statistics[1]. Lineitem is one relation of the TPC-H database benchmark, emulating the common structure of a table of shopping transactions. For different experiments we created modified versions of the real-world datasets by truncating them after the first 20 columns or the respective number of rows.

4.2 DFD versus Tane and FastFDs

In this section we analyze the effectiveness of the pruning strategies used by Tane, FastFDs, and DFD. We investigate the impact of the number of columns, the number of rows. Subsequently, we inspect the memory requirements of each approach.

4.2.1 *Impact of number of columns*

As mentioned in Section 3.5, functional dependency discovery is a problem with an exponential time complexity depending on the number of columns. The complete search space for a dataset with n columns has a cardinality of 2^n-1. Adding one column to a dataset doubles the number of nodes to examine. Therefore, a brute force approach that does not use pruning strategies is infeasible.

Figure 2 shows the scaling for the number of columns on the NCVoter and Uniprot datasets. We had to restrict the number of rows in order to have at least some datapoints for FastFDs and Tane. On both datasets, DFD outperforms existing work significantly. While being up to three orders of magnitude faster than FastFDs, DFD was also able to process 10 more columns than FastFDs. Because of the calculation of the difference sets, FastFDs has high initial costs even for only two columns. Although Tane initially was only slightly slower than DFD it ran into memory problems from the 23rd column on, hitting the 64 GB memory restriction. We can observe a similar behavior for the NCVoter dataset. However, this time FastFDs was able to process two columns more than DFD, because DFD hit the memory limit from the 32nd column on. In general we can observe a plain correlation of DFD's runtime and the result size, (number of dependencies) illustrated via the grey bars. This supports our claim that DFD significantly prunes the search space through the random walk strategy. To further clarify our point we compared the number of generated partitions through DFD and Tane in Figure 3. Here we see clearly that DFD performs by magnitudes fewer FD verifications than Tane.

Figure 4 shows the development of the memory requirements of Tane, FastFDs, and DFD depending on the number of columns in the dataset. On both datasets, Tane is the first algorithm that exceeds the default heap size of the Java Virtual Machine. In both cases, Tane's memory requirement increases dramatically with the number of columns. DFD manages to process eight more columns than Tane on the NCVoter and 11 more on the Uniprot dataset. Unlike Tane, DFD did not fail because of the memory limit on the second benchmark.

Since FastFDs is a tuple oriented approach, its memory requirement is strongly dependent on the value distributions of the dataset. For Uniprot, the distribution favored FastFDs, which managed to process two columns more than DFD.

[1] `http://www.ncsbe.gov/ncsbe/data-statistics.`

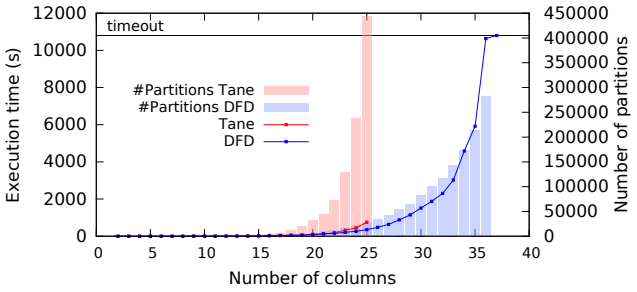
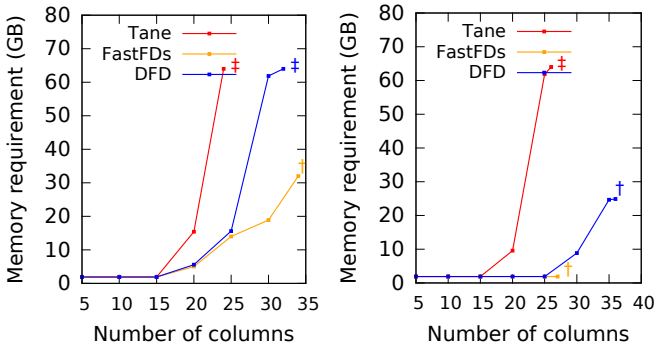


Figure 3: Execution time and number of partitions for Tane and DFD n the first 100,000 rows of the Uniprot dataset.



(a) NCVoter (b) Uniprot

Figure 4: Memory requirements depending on the column count for Tane, FastFDs, and DFD on NCVoter and Uniprot. († - Time Limit ‡ - Memory Limit)

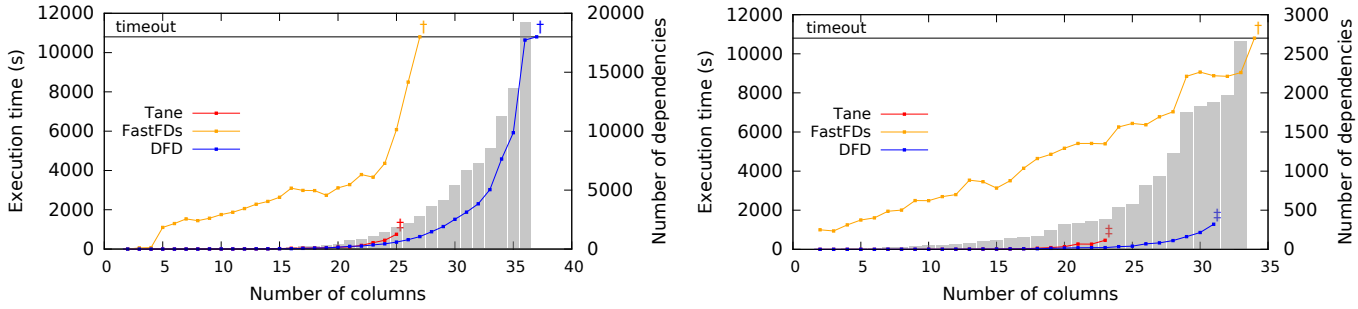
Since the goal for the design of DFD was a scalable algorithm for the discovery of FDs, we also compared the runtimes of Tane, FastFDs, and DFD on the complete datasets. Figure 5 shows how Tane, FastFDs, and DFD performed on the whole datasets of NCVoter and Uniprot. As expected, FastFDs was not able to process the datasets and already failed at the calculation of the difference sets.

On the NCVoter dataset, Tane and DFD perform equally fast for up to 11 columns. However, because of the memory limit, DFD is able to process two columns more than Tane. DFD fails at a column count of 14, because of the given time constraint. A similar result can be observed regarding the Uniprot dataset. Up to 12 columns, Tane and DFD have a similar runtime. Afterwards, Tane's runtime increases faster than DFD and hits the memory limit early for 22 columns. DFD manages to determine the set of minimal FDs for up to 31 columns.

4.2.2 *Impact of number of rows*

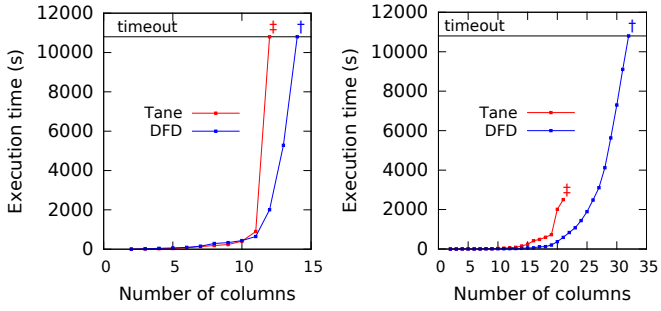
Mannila et al. state that Tane's time complexity is only linearly dependent on the number of rows [11]. This intuition holds for the partition verifications of Tane and DFD. However, FastFD's preprocessing step, the generation of all difference sets has a quadratic time complexity [18].

Figure 6a shows that for a very small amount of rows, all approaches are suitable. FastFDs is not able to process the dataset for 250,000 rows. Note that we also had to re-



(a) Uniprot (b) NCVoter

Figure 2: Execution time for Tane, FastFDs, and DFD on the first 100,000 rows of the NCVoter and Uniprot dataset. († - Time Limit ‡ - Memory Limit)



(a) NCVoter (b) Uniprot

Figure 5: Execution time for Tane, FastFDs, and DFD on the complete NCVoter and Uniprot dataset. († - Time Limit ‡ - Memory Limit)

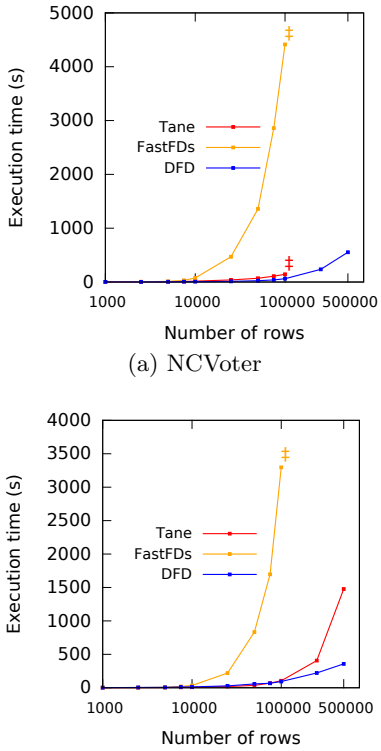
strict the number of columns on both datasets to the first 20 columns, in order to produce any numbers. Here, the limiting factor was not the timeout as seen in the measurements in Section 4.2.1, but the heap space. Profiling showed that the generation of the search tree for the calculation of the difference set's minimal covers is responsible for the memory consumption.

Although DFD is only a slightly faster than Tane for up to 100,000 rows, it is able to process 250,000 rows within the given memory limits, which Tane cannot. Even the maximal dataset of 500,000 rows is not an issue for DFD. As expected, DFD scales linearly on the number of rows.

The results of the Uniprot dataset in Figure 6b show a similar behavior for all approaches. Again, FastFDs fails to process 250,000 rows, because it exceeds the memory limit when calculating the minimal covers. DFD is by multiple factors faster than Tane for 250,000 rows and nearly an order of magnitude faster for 500,000 rows. The reason is that Tane needs to calculate more than 11 times the number of partitions compared to DFD.

Figure 7 displays the growth of the memory requirement dependent on the number of rows in the dataset. We used the same datasets as in Section 4.2.2.

Unlike in the previous experiment where we tested the memory growth depending on the number of columns, FastFDs performs worse than both Tane and DFD. The reason is the preprocessing step, which calculates the difference sets based on the tuples of the dataset. Since the number of possible



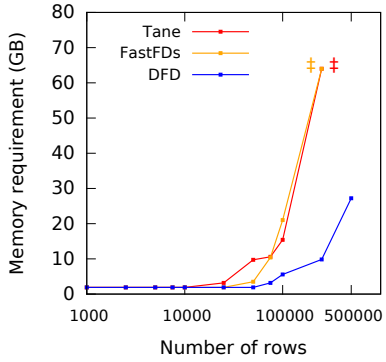
(a) NCVoter

(b) Uniprot

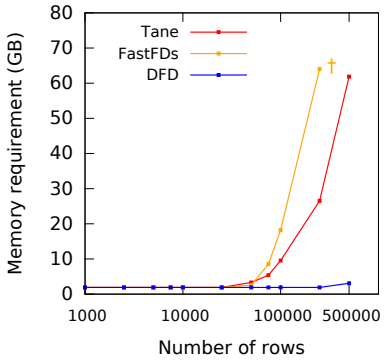
Figure 6: Execution time for Tane, FastFDs, and DFD on the first 20 columns of the NCVoter and Uniprot dataset. († - Time Limit ‡ - Memory Limit)

difference sets increases quadratically with the number of rows, the number of difference sets that need to be covered in the later stage of FastFDs increases in the same manner. However, this forces FastFDs to cover a much larger number of paths in the search tree that determines the minimal covers that eventually result in the minimal dependencies. Profiling of FastFDs showed that storing that search tree led to the violation of the given memory limit.

Tane performed slightly better than FastFDs for the NCVoter dataset. However, both algorithms failed when processing 250,000 rows. On the Uniprot dataset Tane was able to



(a) NCVoter



(b) uniprot

Figure 7: Memory requirements depending on the row count for Tane, FastFDs, and DFD on NCVoter and Uniprot. (‡ - Memory Limit)

calculate the complete result for the largest dataset, albeit almost exhausting the given memory pool.

In contrast, DFD easily processes both datasets, regardless of the number of rows. DFD requires around 30% of the memory that Tane needs in order to process NCVoter with 100,000 rows. The advantage is even more noticeable for 500,000 tuples on Uniprot. DFD solely requires 5% of Tane's memory pool. That is because of DFD's aggressive pruning strategy, which significantly reduces the partition computation.

Table 2 shows a comparison of the runtimes on the lineitem dataset. We generated the lineitem datasets with different scale factors (SF): 0.1, 0.3, and 1.0.

FastFDs was only able to calculate a results in the given memory and time constraints for 2 and 3 columns in the dataset with a scale factor of 0.1. Even for those two data points, DFD was two orders of magnitude faster. This shows that FastFDs scales exceptionally poor with an increasing number of rows. Profiling showed that it was not even able to reach the minimal cover calculation step, because it was still occupied with computing the difference sets. DFD and Tane perform much better. Both algorithms are able to determine the complete result for the whole 0.1 dataset. For a scale factor of 0.3, Tane has to terminate early, because of the memory limit for column counts beyond 13. In contrast, DFD processes the dataset for the whole range of columns. At a scale-factor of 1.0 Tane already fails because of memory issues at a column count of 11. DFD is able to process the dataset for up to 14 columns. However, unlike Tane, it fails later due to the time limit.

For some data points, especially for $SF = 0.1$, Tane performs better than DFD. The reason is that for the lineitem dataset, Tane actually often calculates fewer partitions than DFD, e.g., only 9,417 partitions for the complete 0.1 dataset while DFD created 19,224 partitions. As lineitem is a generated dataset, the cardinality of the vast amount of the dependencies is very similar and small, being at maximum size 5 and equally distributed on only a few levels of the powerset lattice. In such a scenario a level-wise breadth-first approach is more efficient than a depth-first approach if the dataset is small enough.

4.3 Partition Removal

In Section 3.7, we introduced a partition removal concept that enables DFD to remove partitions from the heap if a certain amount of created partitions is exceeded. Figure 2b shows results where DFD performed worse than FastFDs because the memory limit was already exceeded for 32 columns. Since DFD stored 40,732 partitions to calculate the complete set of minimal FDs for 31 columns, we set the maximum partition threshold to 10,000 for our experiment.

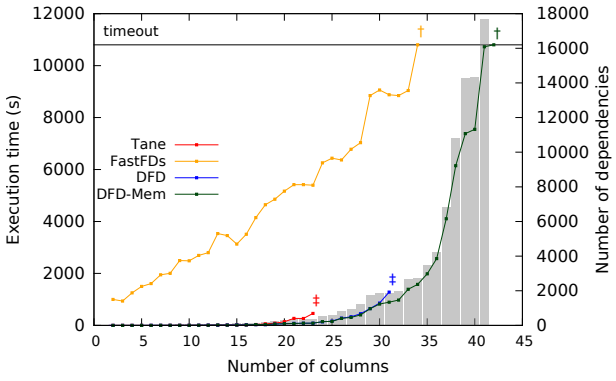


Figure 8: Execution time for Tane, FastFDs, DFD, and DFD-Mem on the first 100,000 rows of the NCVoter dataset. († - Time Limit ‡ - Memory Limit)

Figure 8 shows how the memory aware DFD, DFD-Mem performs compared to the other three algorithms. The threshold of 10,000 partitions is exceeded the first time for a column count of 26. Up to 30 columns DFD and DFD-Mem are equally as fast. However, at 31 columns the negative impact of the memory limit is noticeable for the runtime of DFD. That is because DFD spends a lot of time garbage collecting whereas DFD-Mem does not exceed the memory limit at all. As expected, DFD-Mem actually performs only slightly worse than DFD. Considering that we reduced the amount of stored partitions to less than 25%, the partition removal heuristic seems to be promising. Table 3 illustrates that the heuristic for choosing which partitions can be removed, works quite well. In fact, DFD-Mem rarely needs to recalculate partitions.

5. CONCLUSION

In this paper, we presented DFD – a new algorithm for discovering functional dependencies. DFD benefits from aggressive pruning through a random-walk depth-first traver-

	600,122 rows (SF=0.1)			1,800,366 rows (SF=0.3)			6,001,215 rows (SF=1.0)		
#Columns	Tane	FastFDs	DFD	Tane	FastFDs	DFD	Tane	FastFDs	DFD
2	**1.3**	612.2	1.7	**3.6**	TL	4.6	**11.9**	TL	15.1
3	**2.9**	667.9	2.8	**7.4**	TL	8.2	**25.8**	TL	33.2
4	**4.7**	TL	5.0	**13.8**	TL	18.8	**51.4**	TL	59.3
5	**8.1**	TL	6.6	26.1	TL	**23.4**	84.2	TL	**80.0**
6	13.3	TL	**12.0**	46.8	TL	**44.2**	156.3	TL	**141.9**
7	25.1	TL	**17.9**	86.0	TL	**63.8**	266.3	TL	**215.4**
8	42.6	TL	**28.8**	138.3	TL	**85.2**	437.7	TL	**290.1**
9	88.2	TL	**47.0**	218.4	TL	**132.3**	948.8	TL	**416.3**
10	144.5	TL	**64.8**	385.7	TL	**194.2**	2054.0	TL	**682.9**
11	165.3	TL	**114.6**	532.1	TL	**322.5**	ML	TL	**1059.8**
12	**216.0**	TL	262.4	800.7	TL	**749.9**	ML	TL	**2693.9**
13	**341.1**	TL	479.8	**1041.6**	TL	1390.2	ML	TL	**4584.8**
14	**549.7**	TL	869.8	ML	TL	**2635.5**	ML	TL	**7868.0**
15	**848.4**	TL	1412.0	ML	TL	**4218.3**	ML	TL	TL
16	**969.6**	TL	1894.1	ML	TL	**5672.4**	ML	TL	TL

Table 2: Execution time (seconds) for Tane, FastFDs, and DFD on three different lineitem datasets

#columns	25	26	27	28	29	30	31
#DFD	8,994	13,967	16,166	21,787	31,163	35,071	40,732
#DFD-Mem	8,994	14,066	16,211	22,082	31,825	35,882	41,390
#recalculated	0	99	45	295	662	811	658

Table 3: Calculated partitions of DFD and DFD-Mem for the first 100,000 rows of the NCVoter dataset.

sal and an efficient result verification that enables it to ignore many false positives. We presented extensive experiments that illustrate the superiority of DFD over the popular algorithms Tane and FastFDs on large real world datasets, outperforming them most of the time by orders of magnitude. In general, a random walk approach for choosing LHS candidates works well. However, we can imagine that incorporating some heuristics might further improve the runtime, despite considering the number of distinct values of a column combination as a heuristic had no influence.

6. REFERENCES

[1] Z. Abedjan and F. Naumann. Advancing the discovery of unique column combinations. In *CIKM*, pages 1565–1570, 2011.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

[3] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, pages 746–755, 2007.

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

[5] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, pages 458–469, 2013.

[6] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *TKDE*, 23(5):683–698, 2011.

[7] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *SIGMOD*, pages 469–480, 2011.

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

[9] L. Golab, H. Karloff, F. Korn, and D. Srivastava. Data Auditor: Exploring data quality and semantics using pattern tableaux. *PVLDB*, 3(1-2):1641–1644, 2010.

[10] A. Heise, J.-A. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. *PVLDB*, 7(4):301–312, 2013.

[11] 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.

[12] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658, 2004.

[13] S. Lopes, J.-M. Petit, and L. Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *EDBT*, volume 1777, pages 350–364, 2000.

[14] H. Mannila and K.-J. Räihä. On the complexity of inferring functional dependencies. *Discrete Applied Mathematics*, 40(2):237–243, 1992.

[15] F. Naumann. Data profiling revisited. *SIGMOD Rec.*, 42(4):40–49, 2014.

[16] N. Novelli and R. Cicchetti. FUN: An efficient algorithm for mining functional and embedded dependencies. In *ICDT*, volume 1973, pages 189–203, 2001.

[17] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. GORDIAN: Efficient and scalable discovery of composite keys. In *VLDB*, pages 691–702, 2006.

[18] C. Wyss, C. Giannella, and E. Robertson. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *DaWaK, 2001*, pages 101–110, 2001.

[19] H. Yao, H. J. Hamilton, and C. J. Butz. FD_Mine: Discovering functional dependencies in a database using equivalences. In *ICDM*, pages 729–732, 2002.