Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms

Thorsten Papenbrock[2] Jens Ehrlich[1] Jannik Marten[1]
Tommy Neubert[1] Jan-Peer Rudolph[1] Martin Schönberg[1]
Jakob Zwiener[1] Felix Naumann[2]

[1] firstname.lastname@student.hpi.uni-potsdam.de
[2] firstname.lastname@hpi.de
Hasso-Plattner-Institut, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany

ABSTRACT

Functional dependencies are important metadata used for schema normalization, data cleansing and many other tasks. The efficient discovery of functional dependencies in tables is a well-known challenge in database research and has seen several approaches. Because no comprehensive comparison between these algorithms exist at the time, it is hard to choose the best algorithm for a given dataset. In this experimental paper, we describe, evaluate, and compare the seven most cited and most important algorithms, all solving this same problem.

First, we classify the algorithms into three different categories, explaining their commonalities. We then describe all algorithms with their main ideas. The descriptions provide additional details where the original papers were ambiguous or incomplete. Our evaluation of careful re-implementations of all algorithms spans a broad test space including synthetic and real-world data. We show that all functional dependency algorithms optimize for certain data characteristics and provide hints on when to choose which algorithm. In summary, however, all current approaches scale surprisingly poorly, showing potential for future research.

1. FUNCTIONAL DEPENDENCIES

Functional dependencies (FDs) express relationships between attributes of a database relation. An FD $X \rightarrow A$ states that the values of attribute set X uniquely determine the values of attribute A. For instance, the *zipcode* attribute in an address relation usually determines the *city* attribute: *zipcode* $\rightarrow$ *city*. Due to this property, functional dependencies serve a wide range of data analysis, data integration, and data cleansing tasks. In particular, they are frequently used for schema normalization and database (re-)design.

Liu et al. provide an excellent survey of the literature, analyzing many algorithms for functional dependency discovery from an algorithmic and theoretical perspective [11]. Our evaluation paper, on the other hand, provides brief descriptions of the seven most cited and most important algorithms with a focus on their practical implementation and evaluation: TANE [7], FUN [14], FD_MINE [22], DFD [1], DEP-MINER [12], FASTFDS [20], and FDEP [6]. While all algorithms have the same goal, namely the *discovery of all minimal, non-trivial functional dependencies in a given dataset*, they differ in their approach. Existing evaluations and comparisons of subsets of the algorithms do not show well how the approaches differ in their runtime and memory behavior, and there has been no comparative study of this comprehensive set of algorithms in the past. Therefore, it is almost impossible for developers to find the right algorithm for their needs.

In this paper, we analyze and compare the FD algorithms in detail. We measure their execution time and memory consumption on synthetic and real-world datasets and explain the algorithms' behavior in different scenarios. Furthermore, we provide hints on which algorithm should be used for which datasets. Overall, we could confirm original experimental results, where available.

Contributions. We classify seven FD algorithms by their main concepts giving an overview on state-of-the-art developments. We revisit all algorithms and provide additional descriptions for their practical implementation where their original publications are sparse. We compare the algorithms on different datasets and evaluate both runtime and memory usage. From our experimental results, we derive concrete suggestions on when to use which algorithm. We also make all implementations, data, and the evaluation framework available online[1].

Structure. Section 2 gives an overview of FD discovery in general, discussing common concepts, alternative approaches, and our classification of discovery algorithms. Section 3 describes the implementation of the algorithms. Section 4 presents our evaluation results and Section 5 concludes the advantages and disadvantages of each discovery algorithm.

2. OVERVIEW OF FD ALGORITHMS

We first formally define functional dependencies following the notation of [7] and then classify the seven FD algorithms by their technical and logical commonalities to give an overview on state-of-the-art FD discovery approaches.

[1] `https://www.hpi.de/naumann/projects/repeatability/`

Proceedings of the VLDB Endowment, Vol. 8, No. 10
Copyright 2015 VLDB Endowment 2150-8097/15/06.

2.1 Preliminaries

Given a relational schema R and an instance r over R, a functional dependency (FD) $X \rightarrow A$ is a statement over a set of attributes $X \subseteq R$ and an attribute $A \in R$ denoting that all tuples in X uniquely determine the values in A [19]. More formally, let $t_i[A]$ be the value of tuple t_i in attribute A; the FD $X \rightarrow A$ holds 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 $t_1[A] = t_2[A]$. We call X the *left hand side* (lhs) of the FD, and A *right hand side* (rhs). A functional dependency $X \rightarrow A$ is *minimal* if no subset of X determines A, and it is *non-trivial* if $A \notin X$. To discover all functional dependencies in a dataset, 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.

Liu et al. have shown that the complexity of FD discovery is in $\mathcal{O}(n^2(\frac{m}{2})^2 2^m)$ where m is the number of attributes and n the number of records [11]. This complexity can be seen when modeling the search space of FDs as a *power set lattice* of all attribute combinations, i.e., a Hasse diagram. Each node contains a unique set of attributes and is connected to those nodes that contain either a direct super- or subset of attributes. Figure 1 in Section 3.1 shows an example lattice for three attributes A, B, and C. Each level i in this lattice contains all attribute sets of size i. Intuitively, each edge in the lattice represents one possible FD. As the lattice contains 2^m nodes and each node has $\frac{m}{2}$ edges on average, the number of FD candidates is $2^m \cdot \frac{m}{2}$. An FD candidate comprises $\frac{m}{2}$ attributes on average. To validate an FD candidate, a naive algorithm checks all records against all other records yielding a complexity of $n^2 \cdot \frac{m}{2}$ string comparisons. The resulting complexity $\mathcal{O}(n^2(\frac{m}{2})^2 2^m)$ is, of course, an upper bound, because the seven FD algorithms use aggressive candidate pruning and sophisticated validation methods.

Popular and efficient data structures for the validation of FD candidates are so-called *partitions*, sometimes also called *position list indexes* (PLI). First introduced by Cosmadakis et al., a partition denoted by π_X groups tuples into equivalence classes by their values of attribute set X [4]. Thereby, two tuples t_1 and t_2 of an attribute set X belong to the same equivalence class iff $\forall A \in X : t_1[A] = t_2[A]$. Consider, for instance, the tuples (Peter, Miller), (Thomas, Miller), (Peter, Moore), and (Peter, Miller) in a relation Person(Name, Family). Then, $\pi_{\{Name\}} = \{\{1,3,4\},\{2\}\}$ and $\pi_{\{Family\}} = \{\{1,2,4\},\{3\}\}$. A partition can, hence, efficiently be implemented as a set of record id sets. To calculate larger partitions, e.g., for $\pi_{\{Name,Family\}}$, one can simply *intersect* two smaller subset partitions by intersecting their equivalence classes. In this way we calculate $\pi_{\{Name,Family\}} = \pi_{\{Name\}} \cap \pi_{\{Family\}} = \{\{1,4\},\{2\},\{3\}\}$.

Once calculated, the partitions can be used to validate an FD candidate via *refinement* check: A partition π refines a partition π', if every equivalence class in π is a subset of some equivalence class of π'. This means that tuples with same values in π have also same values in π', which is the definition of a functional dependency. Therefore, the functional dependency $X \rightarrow A$ holds iff π_X refines $\pi_{\{A\}}$ [7]. The partition $\pi = \{\{1,2\},\{3\}\}$, for example, refines the partition $\pi' = \{\{1,2,3\}\}$, because both equivalence classes in π map into the same class in π'. One can finally compress partitions to so-called *stripped partitions* $\hat{\pi}$ by removing all equivalence classes with only a single entry, because these are irrelevant for the refinement checks.

2.2 Classification

To better understand the seven functional dependency discovery algorithms and their properties, we classify them into three categories. A detailed discussion of each algorithm follows in Section 3.

Lattice traversal algorithms: The algorithms TANE, FUN, FD_MINE, and DFD explicitly model the search space as a power set lattice of attribute combinations in order to traverse it. While TANE, FUN, and FD_MINE use a level-wise bottom-up traversal strategy that builds upon the *apriori-gen* candidate generation principle [2], DFD implements a depth-first random walk. Although the traversal strategies vary, all four algorithms successively generate new FD candidates and validate them individually using stripped partitions. To prune the search space, the algorithms derive the validity of not yet checked candidates from already discovered FDs and non-FDs. The pruning rules usually exploit the minimality criterion of FDs (see Section 2.1) and have a great overlap between the four approaches: all algorithms in this class use and extend the pruning rules of TANE.

Difference- and agree-set algorithms: The algorithms DEP-MINER and FASTFDS build upon so-called difference- and agree-sets to find all minimal functional dependencies. Instead of successively checking FD candidates, the two algorithms search for sets of attributes that agree on the values in certain tuple pairs. The search space is, hence, primarily defined by the cross product of all tuples. Intuitively, attribute sets that agree on certain tuple values can functionally determine only those attributes whose same tuples agree. Once the agree sets are calculated, both algorithms can derive all valid FDs from them: DEP-MINER first maximizes and then complements the agree sets in order to infer the FDs; FASTFDS first complements the agree sets into difference-sets and then maximizes these difference-sets to infer the FDs. For FD inference, DEP-MINER searches the difference sets level-wise while FASTFDS transforms them into a search tree that it traverses depth-first.

Dependency induction algorithms: Dependency induction algorithms like FDEP start with a set of most general dependencies, i.e., each attribute functionally determines all other attributes, and then successively specialize this set using observations made in the data. For the specialization part, FDEP compares all records pair-wise in order to find attribute sets with equal projections on the two inspected records. These attribute sets (and all their subsets) cannot functionally determine any of the other attributes. Hence, FDEP updates the current FDs as follows: It removes all FDs that the current tuple pair violates and adds all supersets of removed FDs that can still be assumed to be valid. When all records are compared, the remaining FDs must be valid, minimal, and complete. The tuple-wise comparisons are similar to the computation of difference- and agree-sets, but the reasoning part differs: FDEP specializes the set of dependencies with each observation, whereas DEP-MINER and FASTFDS first fully generate and maximize difference-sets before they infer dependencies. Furthermore, FDEP is the only algorithm that does not use stripped partitions.

2.3 Other Algorithms

For this evaluation paper, we have chosen the seven most cited and most important FD discovery algorithms. Due to the popularity of the topic, there are a few further approaches that the reader should be aware of.

In 1992, Mannila and Räihä published a survey on four early FD discovery algorithms [13]. The described techniques lay the foundations for most algorithms in our evaluation: They already model the search space as hypergraphs, use agree sets for the dependency checks and apply minimality pruning to reduce the search space. The algorithms are formally described and their complexities are analyzed, but no implementation or comparative evaluation is given.

In 1993, Schlimmer introduced another early algorithm, which uses a decision tree and a rich set of pruning rules to infer functional dependencies [16]. Although the paper itself did not evaluate the performance of the presented ideas, we find most pruning rules and similar search strategies in the algorithms of our evaluation, e.g., in TANE [7].

Apart from the exact and complete discovery algorithms surveyed here, there is a body of work on approximate solutions, such as [8] or [10]. We deliberately omit these approaches from our evaluation, because their results are incomparable: They can be faster than exact algorithms, but they trade off efficiency with correctness or completeness.

3. SEVEN FD DISCOVERY ALGORITHMS

We now describe the seven most important, previously published algorithms for exact functional dependency discovery. The algorithms were carefully re-implemented in Java based on their original publications and, in some cases, with the help of the original authors. Each algorithm has been coded by two experienced students with much interaction among all authors of this paper. All implementations follow the descriptions as closely as possible.

Where the original description and pseudo-code were not sufficient, we supplemented our implementations with the most probable solutions, which we describe in this section as well. In cases where the supplementations were not obvious, the authors were contacted to gather further information on the gaps and to help resolve specific questions. This collaboration was especially helpful to build correct implementations of TANE, FUN, DFD, and FDEP. We thank the authors for their helpful feedback.

Please note that the original algorithms were introduced in full scientific papers; in this experimental evaluation we can provide only high-level descriptions of each. We provide more details for those algorithmic parts that require more explanations than given in the respective original paper.

3.1 TANE

The TANE algorithm by Huhtala et al. [7] is based on three main concepts: *partition refinement* to check if a functional dependency holds, *apriori-gen* to ensure that all and only minimal functional dependencies are found, and *pruning rules* to dynamically reduce the search space.

Like all lattice traversal algorithms, TANE models the search space as Hasse diagram as described in Section 2.1. Figure 1 depicts one such Hasse diagram for the relation $R = \{A, B, C\}$. The lattice is partitioned into levels where level L_i contains all attribute combinations of size i. Instead of calculating the entire lattice in the beginning, TANE starts with Level 1 (attribute sets of size one) and then moves upwards level by level (bold lines in the example). In each level L_i, the algorithm tests all attribute combinations $X \in L_i$ for the functional dependency $X \setminus A \to A$ for all $A \in X$. If a test delivers a new functional dependency, TANE prunes all supersets of the discovered FD using a set of pruning rules. When moving upwards to the next level, the *apriori-gen* function [2] calculates only those attribute combinations from the previous level that have not been pruned already. Note that Figure 1 shows an example that we describe in more detail in the end of this section.

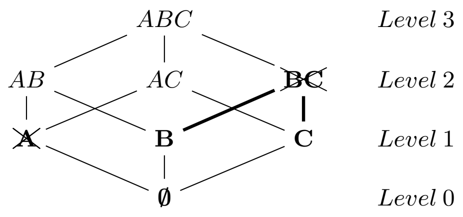

Figure 1: A pruned lattice used in TANE

TANE's search space pruning is based on the fact that for a complete result only minimal functional dependencies need be discovered. To prune efficiently, the algorithm stores a set of right-hand-side candidates $C^+(X)$ for each attribute combination X. The set $C^+(X) = \{A \in R \mid \forall B \in X : X \setminus \{A, B\} \to B \textit{ does not hold}\}$ contains all attributes that may still depend on set X. $C^+(X)$ is used in the following three pruning rules:

Minimality pruning: If a functional dependency $X \setminus A \to A$ holds, A and all $B \in C^+(X) \setminus X$ can be removed from $C^+(X)$. In other words, any FD $X \to B$ cannot be minimal, if X contains the FD $X \setminus A \to A$, because A makes X non-minimal. Note that this definition also includes $A = B$.

Right-hand-side pruning: If $C^+(X) = \emptyset$, the attribute combination X can be pruned from the lattice, because there are no more right-hand-side candidates for a minimal functional dependency.

Key pruning: If the attribute combination X is a key, it can be pruned from the lattice. A key X is an attribute combination that determines all other attributes $R \setminus X$ in a relation R. Hence, all supersets of X are super keys and by definition non-minimal.

To check all possible functional dependencies on a current level, TANE uses *stripped partitions* $\hat{\pi}$ as described in Section 2.1. An FD $X \to A$ is valid iff $\widehat{\pi_X}$ refines $\widehat{\pi_{\{A\}}}$, i.e., all equivalence classes in $\widehat{\pi_X}$ can entirely be mapped to equivalence classes in $\widehat{\pi_{\{A\}}}$. While traversing the attribute lattice from bottom to top, TANE successively calculates the stripped partitions for new attribute combinations from their subsets via partition intersection.

A refinement check for an FD candidate is an expensive operation. Therefore, TANE optimizes these checks using so-called *error measures.* The error measure $e(X)$ is the minimum fraction of tuples to remove from attribute combination X to be key. Each stripped partition holds such an error measure. It is calculated in the partition intersection operation that creates new stripped partitions. Having these error measures, TANE simply checks the validity of an FD candidate $X \to A$ by testing if $e(X) = e(X \cup A)$ holds. The special case $e(X) = 0$ shows that X is a key.

Our TANE implementation is based on the pseudo-code given in [7] and a reference implementation provided at [17].

When implementing Tane, we found that the proposed key pruning in [7] can lead to incorrect results: Once a key X has been discovered, the pruning procedure deletes the attribute combination X from the lattice (although its C^+ set is not empty); then, it splits $X \rightarrow R \setminus X$ into minimal FDs. To split the key X into its minimal FDs, the procedure needs to check all non-empty C^+ sets of attribute combinations within the same level. Due to previous key pruning, some of these sets might be missing so that certain minimal FDs are not generated. The following example illustrates this key pruning problem: Let $R = \{A, B, C\}$ be a relation with two keys A and BC. Hence, the minimal functional dependencies are $A \rightarrow B$, $A \rightarrow C$ and $BC \rightarrow A$. Now, we run Tane as shown in Figure 1. Tane checks only the elements in bold. The crossed elements have been deleted from the lattice by the key pruning strategies described above. In level L_1, Tane finds the key A. Since $B \in C^+(B)$ and $C \in C^+(C)$, Tane outputs $A \rightarrow B$ and $A \rightarrow C$ and deletes A from the lattice. In level L_2, Tane discovers the key BC, but $C^+(AB)$ and $C^+(AC)$ are not available, because A has already been removed. Therefore, the functional dependency $BC \rightarrow A$ is not added to the result.

The pseudo-code at [17] bypasses the key pruning issue by simply outputting key attribute combinations as keys and not as minimal FDs. In order to produce the same results as other algorithms, this is not a solution for us. Therefore, we modified the key pruning step as follows: Instead of deleting X from the lattice as described in the paper, our implementation keeps X in the lattice, but marks it as invalid. Invalid lattice nodes are not tested for functional dependencies, but they are used to generate further invalid nodes unless their C^+ sets are not empty. In this way, C^+ sets are retained as long as necessary. The key pruning still causes Tane to skip many FD checks and, most importantly, saves expensive partition intersections, because invalid nodes do not require stripped partitions.

3.2 FUN

Similar to Tane, the Fun algorithm by Novelli and Cicchetti traverses the attribute lattice level-wise bottom-up and applies partition refinement techniques to find functional dependencies [14,15]. However, Fun explores a smaller portion of the search space through a more restrictive candidate generation and a lazy look-up of *cardinality* values. The cardinality of an attribute combination X is the number of distinct values in the projection of X. Like the error measure in Tane, the cardinality values can be used to optimize the validation of FD candidates.

Instead of C^+ sets, Fun uses *free sets* and *non-free sets* to prune FD candidates that yield only non-minimal FDs. Free sets are sets of attributes that contain no element that is functionally dependent on a subset of the remaining set. In other words, no FD exists among the attributes of a free set. The set of free sets $\mathcal{FS}$ is defined as follows [14]:

Definition 1. Let $X \subseteq R$ be a set of attributes in relation R, r be a relation instance of R and $|X|_r$ be the cardinality of the projection of r over X, i.e., the number of distinct values in X. Then, $X \in \mathcal{FS}_r \Leftrightarrow \nexists X' \subset X : |X'|_r = |X|_r$.

All attribute sets that are not in $\mathcal{FS}_r$ are called non-free sets. Free and non-free sets implement the minimality pruning, which is already known from Tane. The right-hand-side and key pruning has also been adapted from Tane: Only free sets that are non-unique column combinations, i.e., non-keys, are considered in the candidate generation. The generation itself also uses the *apriori-gen* function.

In some cases, however, the Fun algorithm is able to deduce the cardinality of attribute combinations from its subsets. This is possible if a set X is known to be a non-free set. Then, one of the attributes of set X must be functionally dependent on one of the direct subsets of X. This implies that one of the direct subsets of X has the same cardinality as the set X. This rule can be formalized as follows:

$$\forall X \notin \mathcal{FS}_r, \forall X' \subset X : X' \in \mathcal{FS}_r \Rightarrow |X|_r = Max(|X'|_r) \tag{1}$$

The deduction of cardinality values allows Fun to prune attribute combinations more aggressively than Tane: All non-free sets can directly be pruned from the lattice, because if the cardinality of a superset is needed later, the algorithm can infer this cardinality from the superset's subsets. Tane, on the other hand, needs to process such candidate sets further until their C^+ sets become empty. Hence, the cardinality deduction yields the main performance advantage of Fun. The following example illustrated in Figure 2 explains this pruning and deduction process in more detail.

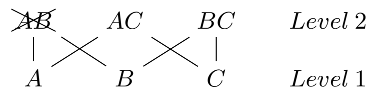

Figure 2: Pruned example lattice for FUN

Consider a dataset $R = \{A, B, C\}$ and the minimal functional dependencies $A \rightarrow B$ and $BC \rightarrow A$. On level L_1, Fun discovers the functional dependency $A \rightarrow B$ and prunes the lattice node AB for level L_2, because it is a non-free set due to the discovered FD. For the next level L_2, Fun's candidate generation omits the node ABC for level L_3, because its subset AB is already pruned and it is no candidate for the left-hand-side of an FD. To find the functional dependency $BC \rightarrow A$, Fun still needs to compare the cardinality of BC and ABC. As the node ABC is pruned, it must be a non-free set and its cardinality can be deduced from its direct subsets following Equation 1.

The cardinality look-up in [15] is implemented by a method called *fastCount()*. According to Equation 1, this method calculates the cardinality of a non-free set as the maximum cardinality of its direct subsets. In reality, one encounters datasets for which this lookup fails, because the direct subsets are sometimes also pruned so that their cardinality information has not been calculated. Therefore, the look-up needs to be implemented recursively as shown in [14]. In this way, the fastCount()-method collects all free set subsets in order to find the largest subset. To enable the recursive look-up, Fun must keep the cardinality values of all previous levels in memory.

3.3 FD_Mine

The FD_Mine algorithm has been published by Yao at al. in two versions [21,22], which exhibit different structures but implement the same theoretical concepts. Like Tane and Fun, FD_Mine traverses the attribute lattice level-wise bottom-up using stripped partitions and partition intersections to discover functional dependencies. It also builds upon TANE's pruning rules. In contrast to TANE and FUN, FD_MINE uses an additional pruning rule that is based on equivalence classes of attribute sets. Two attribute sets are considered *equivalent* with respect to their implied partition ($\Leftrightarrow_\pi$) iff they functionally depend on each other:

$$\forall X, Y \subseteq R : (X \Leftrightarrow_\pi Y) \Leftrightarrow X \rightarrow Y \wedge Y \rightarrow X \tag{2}$$

Whenever a level in the attribute lattice has been validated, FD_MINE scans this level and the discovered FDs for equivalent FDs. If equivalent attribute sets are found, FD_MINE can prune all but one of each group from the lattice, because their functional dependencies can be reconstructed according to the following properties [21, 22]:

$$\forall W, X, Y, Z \subseteq R : \ (X \Leftrightarrow_\pi Y) \wedge XW \rightarrow Z \Rightarrow YW \rightarrow Z \tag{3}$$

$$\forall W, X, Y, Z \subseteq R : \ (X \Leftrightarrow_\pi Y) \wedge WZ \rightarrow X \Rightarrow WZ \rightarrow Y \tag{4}$$

When implementing FD_MINE according to its publications, we discovered three important issues that we discuss in the following:

Non-minimal results: FD_MINE as described in [21, 22] may output non-minimal functional dependencies due to the novel pruning rule. Using Properties 3 and 4, the pruning rule guarantees that the derived functional dependencies hold, but it does not assert their minimality. We show this with a small example:
Assume that X is equivalent to Y and the FD $XW \rightarrow Z$, which fulfills the premise of Property 3, exists. Further assume that an attribute A in Y is functionally dependent on a subset W' of W, which means $\exists W' \subseteq W, \exists A \in Y : W' \rightarrow A$. In this case, Property 3 still implies the functional dependency $YW \rightarrow Z$, but a subset of YW, namely $YW \setminus A$, already functionally determines Z. So the deduced functional dependency is not minimal. We observed many of these cases in our experiments.

Algorithmically, FD_MINE generates the final output (and its non-minimal FDs) from the discovered functional dependencies and the set of found equivalences. Because the code that performs this reasoning operation is not shown in the paper, we must assume that the FDs are not minimized. In our implementation, we did not add a minimization method, because the minimization of FDs is equally complex than the discovery of FDs, i.e., it corresponds to a top-down FD discovery algorithm.

Improvable key pruning: TANE and FUN implement the key pruning one level earlier than FD_MINE. This is performance-wise a disadvantage, but could be intended by the authors. Therefore, we implemented FD_MINE with the key pruning described in [21, 22].

Candidate generation: The proposed *apriori-gen* candidate generation produces incorrect results: The shown method does not set the closure of the functional dependencies to the correct value if more than two children generate the same candidate and the test for existence of all children is omitted. We fixed this in our implementation by setting the correct closure and testing all children for existence as the TANE algorithm does.

3.4 DFD

The DFD algorithm by Abedjan et al. is the most recent algorithm on FD discovery [1]. Like the three previously discussed algorithms, it also models the search space as a lattice of attribute combinations. The search strategy, however, differs significantly, because DFD traverses the attribute lattice in a depth first random walk.

The random walk requires DFD to model the search space, which is a *single* lattice of attribute combinations in TANE, as *multiple* lattices, where each lattice describes all possible right hand sides for one left hand side. When processing these lattices one after another, the random walk can use what the authors call decidable paths. This means that at any node in the lattice, DFD knows if the next node to visit must be smaller or larger.

The traversal of a lattice starts by choosing one node of a set of seed nodes. DFD then classifies this node as a *dependency* or *non-dependency*, also checking for minimality/-maximality properties. If the node is a dependency, DFD prunes all super-sets using the same pruning rules as TANE and continues with a random, unvisited child node; if the node is a non-dependency, DFD prunes all sub-sets by also classifying them as non-dependencies and continues with a random, unvisited parent node. When DFD cannot find an unvisited node in the appropriate direction, it back-tracks to a previous node in the path or, if the entire path has been processed, to a new seed node to continue from there.

Due to the super- and sub-set pruning, islands of unclassified candidate nodes with no seed connection can arise in the lattice. DFD finds these islands in a post-processing step by complementing the discovered maximum non-dependencies and checking them against the discovered minimal dependencies. All nodes in the complement set must be valid dependencies, i.e., they must be known minimal FDs or supersets of them; if they are not, they are unclassified candidate nodes and serve as new seeds.

Like all lattice-based algorithms, DFD also uses partition refinement techniques to check dependencies. Once calculated, stripped partitions are kept in memory for later use. In contrast to previous algorithms, DFD dynamically deallocates stripped partitions when memory is exhausted. For this purpose, DFD calculates usage counts for the stripped partitions and removes less often used partitions when memory is needed. Some partitions, hence, need to be calculated more than once.

3.5 Dep-Miner

The DEP-MINER algorithm by Lopes et al. infers all minimal functional dependencies from sets of attributes that have same values in certain tuples [12]. These sets are called *agree sets* and their inverse *difference sets*. On an abstract level, DEP-MINER can be divided into five phases as shown in Figure 3: In Phase 1, DEP-MINER computes the stripped partition $\widehat{\pi_A}$ for each attribute in a relational instance r. The $\widehat{\pi_A}$ are then used in Phase 2 to build the agree sets $ag(r)$. Phase 3 transforms the agree sets into maximal sets, i.e., sets of attributes that have no superset with same values in two records of r. In Phase 4, DEP-MINER inverts the agree sets into complement sets. From the complement sets, the algorithm then calculates all minimal FDs in Phase 5. Thereby it uses a level-wise search on top of the complement sets. We describe Phases 2 to 5 in more detail.

Phase 2: An agree set $ag(t_i, t_j)$ is defined pairwise between tuples t_i and t_j of r: $ag(t_i, t_j) = \{A \in R \mid t_i[A] = t_j[A]\}$ where $t_i[A]$ denotes the value of tuple t_i in attribute A. The authors introduce two different procedures to calculate the set of all agree sets $ag(r)$ in r from the set of all stripped

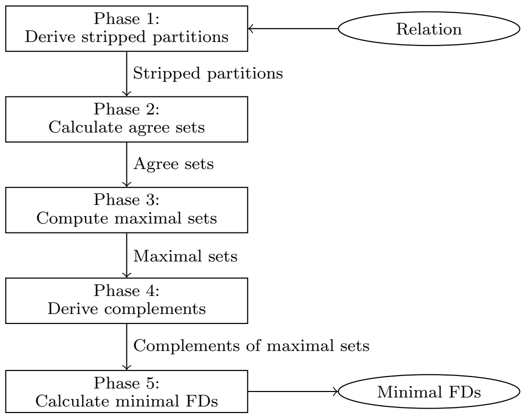

Figure 3: Phases of Dep-Miner

partitions $\hat{\pi}$: The first procedure generates all pairs of tuples that can form agree sets to compute the agree sets on them; the second procedure uses an identifier set $ec(t)$ to compute the agree sets. For our implementation, we chose the second procedure, because the authors of [12] deemed it the more efficient approach if the partitions are large or numerous. The identifier set $ec(t)$ of this procedure is defined as follows:

Definition 2. Let R be a relational schema, t a tuple identifier and $\widehat{\pi_{A,i}}$ is the ith equivalence class of $\widehat{\pi_A}$. Then, $ec(t) := \{(A,i) \mid A \in R \wedge t \in \widehat{\pi_{A,i}}\}$.

In other words, $ec(t)$ describes the relationship between a tuple t and all partitions containing t. The agree sets can now be calculated by intersecting identifier sets $ag(t_i,t_j) = ac(t_i) \cap ac(t_j)$. Consider the following example as an illustration for this operation: Assume $\widehat{\pi_A} = \{\widehat{\pi_{A,0}}, \widehat{\pi_{A,1}}\}$ and $\widehat{\pi_B} = \{\widehat{\pi_{B,0}}, \widehat{\pi_{B,1}}\}$ with $\widehat{\pi_{A,0}} = \{1,2\}$, $\widehat{\pi_{A,1}} = \{3,4\}$, $\widehat{\pi_{B,0}} = \{1,4\}$ and $\widehat{\pi_{B,1}} = \{2,3\}$. Then, the identifier set for tuple 1 is $ec(1) = \{(A,0),(B,0)\}$ and for tuple 2 is $ec(2) = \{(A,0),(B,1)\}$. For $ag(1,2)$ we calculate the intersection of $ec(1)$ and $ec(2)$, which is $\{(A,0)\}$. Removing the index, we find $ag(1,2) = \{A\}$.

Because tuples can be contained in multiple stripped partitions, Dep-Miner might need to compare a specific pair of tuples (t_i,t_j) multiple times. To reduce the number of redundant comparisons, $\hat{\pi}$ is first transformed into its maximal representation $MC = Max_{\subseteq}\{c \in \hat{\pi} | \hat{\pi} \in \hat{r}\}$, where $\hat{r}$ is the set of all stripped partitions of r. The following example shows how MC avoids redundant comparisons: Let $\hat{\pi}_A = \{\{1,2\}\}$ and $\hat{\pi}_B = \{\{1,2,3\}\}$. Without computing MC, the algorithm would need to execute the comparisons $(1,2)$ for $\hat{\pi}_A$ and $(1,2)$, $(1,3)$, and $(2,3)$ for $\hat{\pi}_B$. By computing $MC = \{\{1,2,3\}\}$ the necessary comparisons are reduced to $(1,2),(1,3),(2,3)$ so that $(1,2)$ is compared only once. The original paper [12] omitted the description on how to actually compute MC. Our solution for this task first builds an index on top of the stripped partitions in order to efficiently retrieve and compare all sets containing a certain tuple; then it successively tests the sets of records for sub- and superset relationships.

Phase 3: From the agree sets, Dep-Miner calculates for each attributes A the maximal sets $max(dep(r),A)$ where $dep(r)$ denotes the set of all FDs in relation r. For a specific relational instance r, a maximal set contains all free sets that *(i)* do not include the attribute A and *(ii)* have no subsets in the maximal set. Hence, all agree sets in $max(dep(r),A)$ barely do not functionally determine A:

$$max(dep(r),A) := \{X \subseteq R \,|\, X \nrightarrow A \wedge \forall Y \subseteq R, X \subset Y, Y \rightarrow A\}$$

Fortunately, maximum sets are easy to compute using the following equation:

$$max(dep(r),A) = Max_{\subseteq}\{X \in ag(r) \mid A \notin X, X \neq \emptyset\} \quad (5)$$

Phase 4: Maximal sets describe maximal non-FDs. To derive minimal FDs, Dep-Miner needs to compute the complement $cmax(dep(r),A)$ of the maximal sets $max(dep(r),A)$ for each $A \in R$. This can be done by calculating $R \backslash X$ for all $X \in max(dep(r),A)$.

Phase 5: In the last phase, Dep-Miner generates the left hand sides for all minimal FDs with right hand side A from $cmax(dep(r),A)$ for each attribute $A \in R$. To this end, the algorithm searches level-wise through the complement sets in $cmax(dep(r),A)$: The first level is initialized with all unary attribute sets $\{B\} \in cmax(dep(r),A)$. Then the algorithm moves upwards level by level to infer the minimal FDs. On level L_i, a functional dependency $X \rightarrow A$ for a specific $X \in L_i$ holds, if $\forall Y \in cmax(dep(r),A) : X \cap Y \neq \emptyset$ is true. Before generating the next level, Dep-Miner removes all attribute sets X from L_i that yielded valid left hand sides to ensure that only minimal functional dependencies are found. Then, the generation of level L_{i+1} from level L_i uses an adapted version of the *apriori-gen* algorithm [3]. Dep-Miner terminates when the generation of the next level results in no candidates for each $A \in R$.

3.6 FastFDs

The FastFDs algorithm was introduced by Wyss et al. in [20] as an improvement of Dep-Miner. Therefore, it also builds upon agree sets to derive functional dependencies. After calculating these agree sets, FastFDs follows a different strategy to derive minimal functional dependencies: Because the maximization of agree sets in Phase 3 of Dep-Miner is an expensive operation, FastFDs instead calculates all *difference sets* as $\mathcal{D}_r := \{R \backslash X | X \in ag(r)\}$ directly on the agree sets $ag(r)$. In Phase 4, the algorithm then calculates the difference sets of r modulo A as $\mathcal{D}_r^A := \{D - \{A\} | D \in \mathcal{D}_r \wedge A \in D\}$. The $\mathcal{D}_r^A$ sets are FastFDs' equivalent to complement sets $cmax(dep(r),A)$ used in Dep-Miner and also serve the derivation of minimal functional dependencies. With the $\mathcal{D}_r^A$ sets, FastFDs can reduce the problem of finding all minimal FDs to the problem of finding all *minimal covers* over $\mathcal{D}_r^A$. A minimal cover is defined as follows:

Definition 3. Let $\mathcal{P}(R)$ be the power set of the relation R and $\mathcal{X} \subseteq \mathcal{P}(R)$. The attribute set $X \subseteq R$ is called a *cover* for $\mathcal{X}$ iff $\forall Y \in \mathcal{X}, Y \cap X \neq \emptyset$. Furthermore, X is a *minimal cover* for $\mathcal{X}$ iff X covers $\mathcal{X}$ and $\nexists Z \subset X$ such that Z is a cover for $\mathcal{X}$.

To retrieve FDs from minimal covers, FastFDs uses the following Lemma 3.1:

Lemma 3.1. *Let $X \subseteq R$ and $A \notin X$. Then $X \rightarrow A$ iff X covers $\mathcal{D}_r^A$.*

To find the minimal covers for all $\mathcal{D}_r^A$, FastFDs constructs a search tree for each possible right hand side $A \in R$. Figure 4 shows an example of such a search tree. Each node of the tree stores both the difference sets that are not already covered and the current attribute ordering $>_{curr}$. The attribute ordering orders all attributes contained in $\mathcal{D}_r^A$ by the number of difference sets that they cover. Attributes that cover the same number of difference sets are ordered lexicographically. For the calculation of minimal covers, FastFDs traverses the nodes of the tree using a depth-first search strategy. A greedy heuristic that always chooses the highest ordered attribute in $>_{curr}$ thereby decides which attribute and therefore which branch of the tree should be investigated next. If the search arrives at a node for which $>_{curr} = \emptyset$ but that still holds uncovered difference sets, the set of chosen attributes X within this branch is not a cover for $\mathcal{D}_r^A$ and, hence, $X \rightarrow A$ does not hold; on the other hand, if FastFDs reaches a node that contains no more difference sets to cover, the set of attributes X within this branch is a cover for $\mathcal{D}_r^A$. Because of Lemma 3.1, the functional dependency $X \rightarrow A$ then holds. However, FastFDs still needs to ensure that this dependency is minimal by investigating its left hand side. The algorithm has, finally, discovered all minimal functional dependencies when the search tree has been traversed for each attribute $A \in R$.

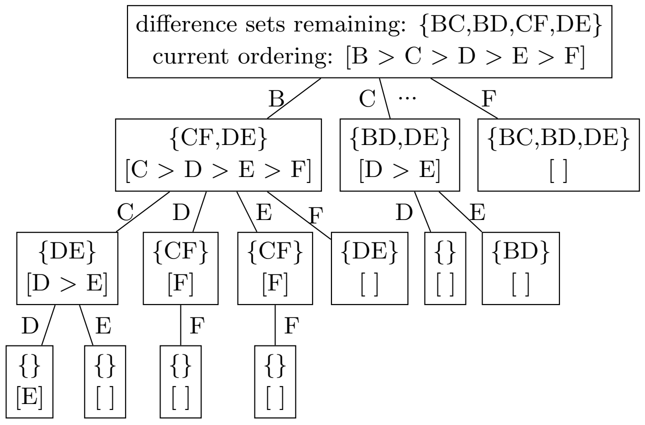

Figure 4: Example searchtree for attribute A

3.7 FDEP

In contrast to previously described algorithms, Fdep by Flach and Savnik [6] follows an approach that is neither based on candidate generation nor on attribute set analysis. Instead, the algorithm successively specializes a set of minimal FDs (or maximal non-FDs) by pair-wise comparing all tuples in a given relation. The authors propose three variants of Fdep: top-down (specializing minimal FDs), bidirectional (specializing both FDs and non-FDs) and bottom-up (specializing maximal non-FDs). Our implementation is based on the bottom-up variant, because it exhibited the best performance in [6]. We referred to the reference implementation at [5] for algorithmic parts that the pseudo-code in [6] does not describe.

The proposed bottom-up Fdep consists of two steps: negative cover construction and negative cover inversion. The *negative cover* is a set of all non-FDs that have been found in a relational instance r. When calculated, its inverse, the *positive cover*, is the set of all minimal FDs. In the following, we first describe the *FD-tree* data structure that efficiently stores negative or positive covers. Then, we describe the two steps of Fdep.

FD-Tree: An *FD-tree* is an extended prefix tree that stores functional dependencies $X \rightarrow A$. The root node represents the empty set and each node in the tree represents an attribute. Each path in the tree describes a left hand side of a dependency X. All nodes of such a path are labeled with the dependent attribute A. The last node of each path denoting a dependency holds an extra flag indicating that A is a leaf and the path X for the FD $X \rightarrow A$ ends here. Using a depth-first search, Fdep can easily look up specializations (larger X) and generalizations (shorter X) of FDs $X \rightarrow A$ in the *FD-tree*.

Negative cover construction: Fdep's first step builds the negative cover, which contains all dependencies that cannot hold in a relational instance r. The negative cover is initialized as an empty *FD-tree*. Then, Fdep compares all pairs of tuples $t_i, t_j \in r$. Each comparison extracts those attribute sets X that have equal values in t_i and t_j. Hence, all dependencies $X \rightarrow A$ with $A \in R \backslash X$ are not valid. Fdep simply adds all these non-FDs to the negative cover.
In the end, the negative cover contains all invalid FDs and most of these non-FDs are not maximal. The next step does not require the negative cover to contain only maximal FDs, but Flach and Savnik propose to filter all generalizations of an FD when inserting a specialization for performance reasons. The reference implementation performs the filtering of generalizations as a post-processing after the negative cover construction. Because the post-filtering has shown a better performance in our implementation, we use this strategy for the evaluation in Section 4.

Negative cover inversion: Fdep's second step calculates the positive cover from the negative cover. The positive cover is the inverse of the negative cover calculated as follows: First, Fdep initializes a new *FD-tree* containing the most general dependencies $\emptyset \rightarrow A$ for all $A \in R$. Then, the algorithm incrementally specializes this *FD-tree* with the non-FDs of the negative cover: For each non-FD $X \rightarrow A$ obtained via recursive depth-first search in the negative cover, Fdep generates all direct specializations $XB \rightarrow A$ with $B \in R \setminus X$ and $B \neq A$, which can be valid FDs. Then the algorithm removes all generalizations of $XB \rightarrow A$ from the positive cover (these are violated by $X \rightarrow A$) and adds $XB \rightarrow A$ to the positive cover. After transferring all non-FD specializations into the positive cover, it contains all and only all minimal non-trivial FDs of r.

4. EVALUATION

In this section we analyze and compare the seven previously described algorithms. We first describe our experimental setup. Then we evaluate the scalability of the FD discovery algorithms regarding the number of columns and the number of rows in the input dataset. Afterwards, we measure the algorithms' execution times on various real-world datasets. Finally, we analyze memory consumption on selected datasets and derive algorithm recommendations for specific situations. Our experiments largely confirm the results on the individual algorithms, but no previous work performed such a comprehensive evaluation and usually compares the performance to only the TANE algorithm.

4.1 Experimental setup

Metanome: We implemented all algorithms for our *Metanome* data profiling framework (`www.metanome.de`), which defines standard interfaces for different kinds of data profiling algorithms. All common tasks, such as input parsing, results formatting, performance measuring, or algorithm parametrization, are standardized by the framework and decoupled from the algorithm. In this way, we can guarantee a uniform test environment for all seven implementations. Our implementations, the Metanome tool, additional documentation and all datasets used in the experiments are available online on our repeatability page[2].

Datasets: We evaluate all algorithms on a synthetic and 16 real-world datasets (see Table 1 in Section 4.4 for an overview). Most of the datasets have also been used in previous evaluations on FD discovery. They cover a wide range of topics and feature different structural properties: The *fd-reduced-30* dataset is the only synthetic dataset in our experiments. It has been generated with the *dbtesma* data generator, which can be found on our repeatability page. The *plista* [9] dataset on web log data and the *flight*[3] dataset on flight route data are excerpts from data streams. Both datasets comprise comparatively many columns and any number of rows. The *uniprot*[4] dataset is the Universal Protein Resource database containing protein sequences with many attributes and the *ncvoter*[5] dataset is the North Carolina's Voter Registration database listing public voter statistics. All further datasets originate from the UCI machine learning repository [18] and have rather small numbers of rows and columns. The datasets *abalone*, *horse*, *balance-scale*, *breast-cancer*, *echocardiogram*, and *hepatitis* all contain anonymous, clinical data about patients and diseases; *bridges* contains data about bridges in Pittsburgh, *iris* contains information about iris plants, *chess* contains end game chess situations, and *nursery* contains background information about nursery school applications; the *letter* dataset comprises information about the English alphabet and the *adult* dataset aggregates census data.

Null values: Because most experiments use real-world data, `null` values ($\perp$) occur. Given two tuples t_1 and t_2 both with a `null` value in attribute A, i.e., $t_1[A] = \perp$ and $t_2[A] = \perp$, then two semantics are possible: $t_1[A] = t_2[A]$ evaluates to `false` or to `true`. Because the two semantics lead to different functional dependencies, our implementations of the algorithms support both settings. For our evaluation, however, we chose the `null` equals `null` semantics: First, we believe it to be more intuitive, because a completely empty column, for instance, should not functionally determine all other columns; second, `null` equals `null` corresponds to the SQL semantics; third, this semantic was also chosen by the authors of related work so that a comparable evaluation must be based on `null` equals `null`. Performance-wise, however, the `null` equals `null` semantic causes longer runtimes than its counterpart, because FDs become larger on average requiring more attributes to determine one another.

Constraints: The execution time of an algorithm includes the time needed to load and parse the input data. It also includes the time needed to parse the output into a uniform format, but it does not include the time for writing results to disk. In this way, experiments with very large result sets can simply count the FDs instead of keeping them in memory without influencing the actual runtime of the algorithm. For precision, we repeat each experiment four times and only report on the shortest execution time in our evaluation; these times show the best possible performance for each algorithm. Each experiment also limits the maximum execution time to four hours and the maximum memory consumption to 100 GB. The former is enforced by external termination and the latter by the size of the JVM.

Hardware: All experiments are executed on a Dell PowerEdge R620 running CentOS 6.4. The test machine contains two Intel Xeon E5-2650 (2.00 GHz, Octa-Core) processors, 128 GB DDR3-1600 RAM and a 4 TB raid5 storage. Note that all algorithms are single threaded and use only one of the available processor cores. As Java environment, we used OpenJDK 64-Bit Server VM 1.7.0_25.

4.2 Row scalability experiments

In our first experiment, we evaluate the algorithms' scalability with the number of rows in the input dataset. We use the *ncvoter* dataset with all 19 columns and the *uniprot* dataset with its first 30 columns (more columns would make some algorithms inapplicable as we show in Section 4.3). The runtime measurements of this experiment are depicted in Figure 5. FD_MINE does not show up in the two charts, because the first two measurement points already exceed the main memory limit of 100 GB. This is due to the fact that FD_MINE reports a huge number of non-minimal functional dependencies, e.g., around 1.1 million functional dependencies on the *ncvoter* dataset where only 758 minimal functional dependencies exist.

The runtimes of TANE, FUN, and DFD scale almost linearly with the number of rows. This is because the three lattice traversal algorithms validate FD candidates using position list indexes (PLIs) and the costs for PLI intersections grow linearly with the number of rows. Thereby, FUN performs worse than TANE due to its recursive cardinality lookups, which constitute an overhead on these datasets. DFD, on the contrary, scales better than TANE with the number of rows, because it executes far fewer PLI intersects; the overhead for random walk and hole-filling still causes DFD to perform worse than TANE for small numbers of rows. TANE and FUN eventually run into the memory limit of 100 GB, because they have to maintain a huge amount PLIs that outgrows the memory capacity.

The runtimes of DEP-MINER, FASTFDS, and FDEP scale quadratically with the number of rows, because the three algorithms compare all records pair-wise. We also see that the runtimes of DEP-MINER and FASTFDS differ greatly for small numbers of rows, because processing difference sets is more efficient than processing agree sets if the agree sets are created from only a few rows. For large numbers of rows, however, the performance difference becomes negligible. Still, FDEP performs better than both DEP-MINER and FASTFDS.

[2] `https://hpi.de/naumann/projects/repeatability`

[3] `www.transtats.bts.gov`

[4] `www.uniprot.org`

[5] `www.ncsbe.gov/ncsbe/data-statistics`

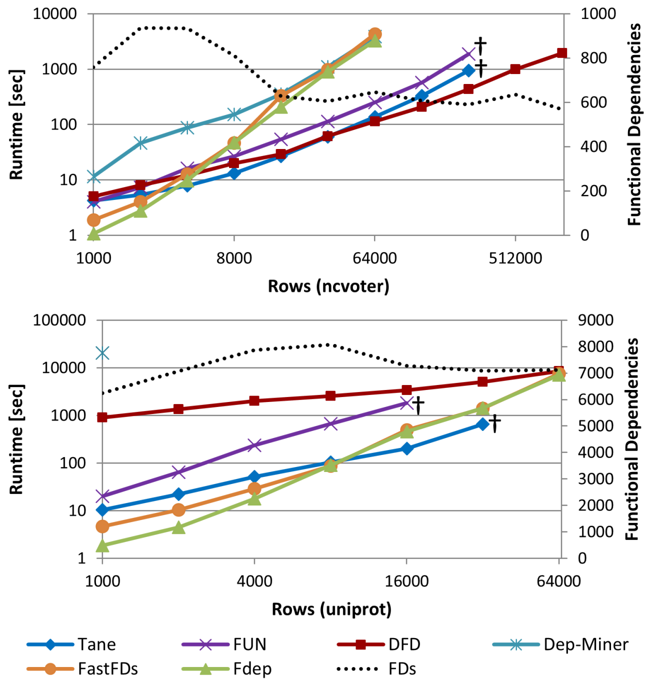

Figure 5: Row scalability on *ncvoter* and *uniprot*

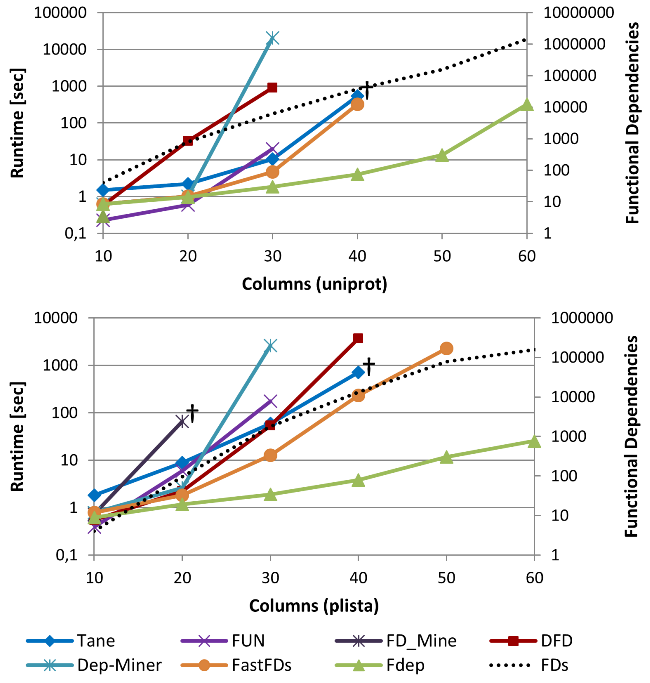

Figure 6: Column scalability on *uniprot* and *plista*

In summary, lattice traversal algorithms perform best on datasets with many rows; difference- and agree-set algorithms as well as dependency induction algorithms perform best on datasets with few rows. The different across the two datasets are due to the different numbers of columns.

4.3 Column scalability experiments

Next, we evaluate the algorithms' scalability with the number of columns. For this experiment, we use the *plista* and *uniprot* datasets, because both comprise many columns. The columns are added by their order in the datasets' schema and the number of rows is fixed to 1000. Figure 6 shows the results of the experiment.

The measurements show that all algorithms scale exponentially with the number of columns: For lattice traversal algorithms the search space of candidate sets grows exponentially, for difference- and agree-set algorithms the set of attribute sets grows exponentially, and for dependency induction algorithms the intermediate results grow exponentially. All these effects are a consequence of the exponentially growing result set of minimal FDs.

FDEP can handle high numbers of columns by far best, because the number of columns influences only the costs of transforming the negative cover into the positive cover. Thereby, the transformation of non-FDs into FDs is a cheap operation compared to PLI intersections, which are exponentially often performed in lattice traversal algorithms.

We also see that FUN, FD_MINE, and DFD are all better than TANE on small numbers of columns due to their special pruning strategies. However, these strategies do not pay off on larger numbers of columns, because of individual reasons: FUN's cardinality look-up is a recursive tree traversal that scales exponentially with the size of the search space; it is hence a huge disadvantage when used on many columns. FD_MINE's reasoning on equivalent attribute sets requires the algorithm to iterate intermediate results that grow exponentially; therefore, this pruning rule introduces a high overhead when the number of columns grows. DFD prunes the search space upwards and downwards, which generates islands of unvisited candidate nodes; as the size of these islands grows exponentially with the number of columns and filling candidate islands is the algorithm's most expensive part, DFD becomes very slow for high column numbers.

Overall, dependency induction algorithms perform best on datasets with many columns; lattice traversal algorithms, on the other hand, can be faster if the number of columns is low; difference- and agree-set algorithms lie performance-wise in between.

4.4 Experiments on different datasets

To see how the seven algorithms perform on different inputs, this section evaluates them on 17 datasets. Most of the datasets in this experiment have also been used in the experiments of the algorithms' individual publications. By using same datasets, we were able to compare our results to previous measurements and confirm them. Note that very long datasets have been shortened to 1,000 rows, which still constitutes a challenge for most algorithms.

Table 1 lists the different datasets and the algorithms' execution times needed for finding all minimal FDs. The table also lists additional information for each dataset[6]. We now comment on the performance of the individual algorithms.

Tane: The TANE algorithm follows a strict bottom-up level-wise search strategy and only implements reliable pruning rules. This makes the algorithm slower than most other

[6]The *FDs*-column counts only minimal FDs and, following most related work, uses the empty set notation $\emptyset \to A$ for attributes A that contain only a single unique value and are thus determined by each other attribute.

Dataset	Columns [#]	Rows [#]	Size [KB]	FDs [#]	Tane [7]	Fun [14]	Fd_Mine [21]	Dfd [1]	Dep-Miner [12]	FastFDs [20]	Fdep [6]
iris	5	150	5	4	1.1	**0.1**	0.2	0.2	0.2	0.2	**0.1**
balance-scale	5	625	7	1	1.2	**0.1**	0.2	0.3	0.3	0.3	0.2
chess	7	28,056	519	1	2.9	1.1	3.8	**1.0**	174.6	164.2	125.5
abalone	9	4,177	187	137	2.1	**0.6**	1.8	1.1	3.0	2.9	3.8
nursery	9	12,960	1,024	1	4.1	1.8	7.1	**0.9**	121.2	118.9	46.8
breast-cancer	11	699	20	46	2.3	0.6	2.2	0.8	1.1	1.1	**0.5**
bridges	13	108	6	142	2.2	0.6	4.2	0.9	0.5	0.6	**0.2**
echocardiogram	13	132	6	538	1.6	0.4	69.9	1.2	0.5	0.5	**0.2**
adult	14	48,842	3,528	78	67.4	111.6	531.5	**5.9**	6039.2	6033.8	860.2
letter	17	20,000	695	61	260.0	529.0	7204.8	**6.0**	1090.0	1015.5	291.3
ncvoter	19	1,000	151	758	4.3	4.0	ML	5.1	11.4	1.9	**1.1**
hepatitis	20	155	8	8,250	12.2	175.9	ML	326.7	5576.5	9.5	**0.8**
horse	27	368	25	128,726	457.0	TL	ML	TL	TL	385.8	**7.2**
fd-reduced-30	30	250,000	69,581	89,571	**41.1**	77.7	ML	TL	377.2	382.4	TL
plista	63	1,000	568	178,152	ML	ML	ML	TL	TL	TL	**26.9**
flight	109	1,000	575	982,631	ML	ML	ML	TL	TL	TL	**216.5**
uniprot	223	1,000	2,439	unknown	ML	ML	ML	TL	TL	TL	ML

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

Table 1: Runtimes in seconds for several real-world datasets

algorithms on tiny datasets with fewer than 10 columns or fewer than 1000 rows. Because it avoids fancy but costly pruning methods, Tane can process most larger datasets in the experiment. Tane performs especially well on the *fd-reduced-30* dataset, because almost all minimal FDs in this dataset can be found on the third lattice level, which greatly limits the search space for the level-wise traversal. However, if the search space becomes too large, i.e., more than 40 columns, Tane exceeds the memory limit.

Fun: The Fun algorithm is clearly faster than Tane on very small datasets, because of its recursive cardinality look-up. As in previous experiments, this look-up becomes a disadvantage on datasets with many columns: the search space for child cardinalities grows exponentially with each additional lattice level; the partition intersections that the look-ups should prevent, however, become ever cheaper on higher lattice levels, because the partitions become smaller.

Fd_Mine: The Fd_Mine algorithm finds many non-minimal FDs, which is a disadvantage for two reasons: First, the result set quickly outgrows memory capacity. Second, the reasoning that is performed on these non-minimal FDs becomes increasingly time consuming outweighing its gain. Therefore, Fd_Mine performs worst in our experiments.

DFD: The Dfd algorithm is extremely powerful when the combination of up-wards and down-wards pruning removes many candidates without producing too many islands of unchecked nodes. In these cases, Dfd can be orders of magnitude faster than all other algorithms; otherwise, the algorithm "starves" in finding unchecked FD candidates. As we showed earlier, the overhead of finding and classifying island nodes scales exponentially with the number of columns. Hence, Dfd performs well for datasets with only few columns (< 20), but becomes inapplicable for datasets with many columns (> 30). Since Dfd is a non-deterministic algorithm, its random walk can find a lattice path with optimal pruning, but it can also choose a path that creates many islands. The runtimes of Dfd, however, do not vary more than those of other algorithms, because the number of island nodes is almost constant throughout repeated runs. The experiments also show that the random walk in combination with the PLI aging prevents Dfd from memory limits.

Dep-Miner: The Dep-Miner algorithm performs well on datasets with few rows. But it is also sensitive to the number of columns, as the maximization of agree sets can become very expensive. This is apparent for the *hepatitis* dataset.

FastFDs: The FastFDs algorithm improves Dep-Miner by minimizing difference sets instead of maximizing agree sets. On most datasets, this optimization has no effect, but if it has an effect like on the *hepatitis* and *horse* dataset, then the improvement is significant.

Fdep: The Fdep algorithm is the overall best algorithm in this experiment, because almost all datasets have only few rows ($\leq 1,000$). On datasets with more rows, i.e., 250,000 rows in *fd-reduced-30*, Fdep runs into time limits, because it tries to compare all records pair-wise. But Fdep scales so well with the number of columns that it can also process more than 100 columns in a few minutes. It still exceeds the memory limit on *uniprot*'s 223 columns, because the resulting set of FDs simply does not fit into 100 GB main memory any more.

4.5 Memory Experiments

To decide which algorithm should be used on which dataset, one must not only consider the algorithms' runtimes but also their memory consumptions. Therefore, we now evaluate the memory footprint of each algorithms on three selected datasets. Our experiment limits the Java Virtual Machine to certain amounts of memory and tests the seven algorithms on each dataset. The memory limits have been chosen from typical real-world memory limits. In Table 2, we report on whether the memory was sufficient or not and what runtimes were achieved. For each algorithm, we list only the execution time for the first memory limit that was sufficient. These times are sometimes a bit higher than previously measured runtimes, because having less memory triggers the garbage collector more frequently.

Tane: Tane requires 8 GB of main memory for the *adult* dataset and about 32 GB for the *letter* dataset. This is a remarkable amount of memory for datasets that are only a few megabytes small. The reason is the level-wise generation of the attribute lattice: Each next level can become a factor larger than the previous one and the algorithm always tries

		256MB	4GB	8GB	16GB	32GB	100GB
Adult	Tane	ML	ML	74			
	Fun	ML	ML	ML	112		
	Fd_Mine	ML	ML	ML	ML	ML	532
	Dfd	ML	6				
	Dep-Miner	6103					
	FastFDs	6097					
	Fdep	861					
Letter	Tane	ML	ML	ML	ML	274	
	Fun	ML	ML	ML	ML	534	
	Fd_Mine	ML	ML	ML	ML	ML	7205
	Dfd	ML	6				
	Dep-Miner	1090					
	FastFDs	1015					
	Fdep	293					
Horse	Tane	ML	ML	ML	ML	491	
	Fun	ML	ML	ML	ML	TL	TL
	Fd_Mine	ML	ML	ML	ML	ML	ML
	Dfd	TL	TL	TL	TL	TL	TL
	Dep-Miner	TL	TL	TL	TL	TL	TL
	FastFDs	411					
	Fdep	8					

TL: time limit of 4 hours exceeded
ML: memory limit exceeded

Table 2: Memory experiment (runtimes in seconds)

to pre-build all PLIs for the next level. To prevent memory overflows, PLIs could be written to disk when memory is exhausted. In this way, Tane could turn memory limits into longer execution times.

Fun: In contrast to Tane, Fun needs to keep all cardinality counts from already finished lattice levels in memory, because the recursive cardinality look-ups might require them later on. So in the worst case, namely if most PLIs are actually needed, the memory consumption for Fun becomes a bit higher than the memory consumption of Tane.

Fd_Mine: Fd_Mine would have a similar memory consumption than Tane if it would not produce so many non-minimal results. These huge result sets eat up all memory.

DFD: Dfd has a better memory performance than the other lattice traversal algorithms, because it prunes much more aggressively and, hence, creates much fewer PLIs. It also monitors its memory consumption to free least recently used PLIs from its internal PLI store. This basically trades the memory limit on the *horse* dataset for a time limit.

Dep-Miner, FastFDs and Fdep: The three algorithms Dep-Miner, FastFDs and Fdep have much lower memory requirements than lattice-based algorithms, because they operate directly on the data and store intermediate results in memory efficient tree structures. Fdep's *FD-tree* is especially memory efficient, because its size directly scales with the size of the result set.

4.6 Extrapolation of experimental results

Our experiments have shown that all algorithms have specific advantages and disadvantages: Lattice traversal algorithms scale well with the number of rows, but their performance decreases for a large number of columns; difference- and agree-set algorithms as well as dependency induction algorithms scale well with an increasing number of columns, but have performance issues with many rows. For these insights, each experiment evaluated the algorithms on only a small cross section of parameters. In the following, we extrapolate previous measurements in order to predict the fastest algorithm for any input dataset.

For this extrapolation, we assume that main memory is arbitrary large. This gives us the best performance for each algorithm. If the memory is not sufficient, the algorithms require memory management techniques that either write intermediate data structures (partially) to disk or optimistically delete and later rebuild them if necessary. These techniques would shift the performance to the disadvantage of lattice based algorithm, because they hit memory limits much earlier than the other algorithms. We do not analyze this dimension here, because no FD algorithm besides Dfd has actually tried memory management techniques and extending the algorithms is not in the focus of this paper.

For any combination of column and row counts, we want to give a prediction for the fastest algorithm. From our experiments, we already know the fastest algorithms for some of these combinations, e.g., the bold runtimes in Table 1. The scalability experiments have further shown the best algorithms for longer ranges of row or column numbers. Figure 7 places all these observations into a matrix. There are points for Tane, Fun, Dfd, and Fdep. All other algorithms never performed best. We already see that all points from Fun lie in the very lower left corner of the chart and are superimposed by points from Fdep. Since Fun performs only sporadically best and only for such a small parameter setting, we ignore these points in our extrapolation.

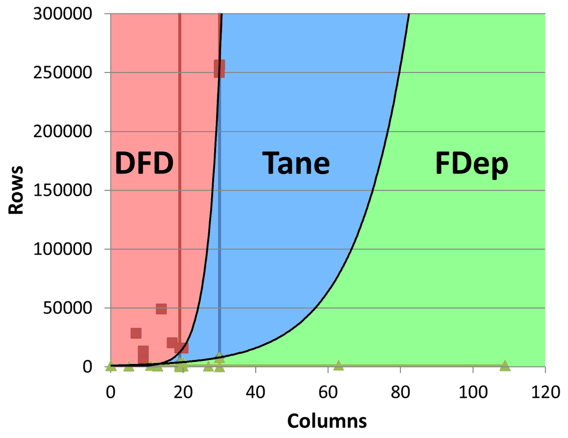

Figure 7: Fastest algorithm with respect to column and row counts when memory is arbitrary large.

With the measurement points of the best algorithms set, we now select those points for which Dfd and Tane perform equally well; then we calculate a regression through these points. Afterwards, we do the same for Tane and Fdep. These two regression lines border areas in which one algorithm is expected to perform best.

Note that the border line between Tane and Fdep is calculated with only small datasets. It is therefore less precise than the line between Dfd and Tane. We tried to add some more measurements at 40 and 60 columns using the *plista* and *uniprot* datasets, but Tane always exceeded our memory limit of 100 GB (it actually exceeded 128 GB). The exact border lines may vary slightly anyways depending on the distribution of FDs in the input dataset, because this also influences the algorithms' performance as shown for the *fd-reduced* dataset in Section 4.4.

When using the chart in Figure 7 as a decision matrix to find the fastest algorithm, one must consider that an algorithm might be the fastest algorithm for a specific input, but it may still be infeasible to actually run this algorithm on that input. For instance, FDEP is the best algorithm for a dataset with 100 columns and 300,000 rows, but it will still run for days or longer; TANE is the best algorithm for 40 columns and 100,000 rows, but one probably needs 256 GB main memory or more.

5. CONCLUSION

In this paper, we discussed the seven most important FD-discovery algorithms in detail and showed their individual strengths and weaknesses. We provided relevant technical details when missing in the algorithms' original publications.

Our evaluation constitutes the first qualitative comparison of all FD-discovery algorithms and analyzes both execution times and memory consumptions. With the extrapolation of our evaluation results, we provide the reader with a tool to choose the fastest algorithm for any given dataset.

In summary, our experiments have shown that FD discovery is still an open research field: None of the state-of-the-art algorithms in our experiments scales to datasets with hundreds of columns or millions of rows. Given a dataset with 100 columns and 1 million rows, which is a reasonable size for a table, DFD, DEP-MINER, FASTFDS, and FDEP will starve in runtime, whereas TANE, FUN, and FD_MINE will use up any available memory. This observation indicates potential for future research.

Furthermore, we discovered very large numbers of FDs in many datasets. For a user to make sense of such large result sets, further techniques are needed, such as ranking FDs by interestingness or visualizing attributes and their FDs. This interpretation of dependencies is a further important topic for future research for all data profiling approaches.

6. 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] 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.

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

[5] FDEP home page. `www.cs.bris.ac.uk/~flach/fdep`. Accessed: 2015-02-26.

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

[7] 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.

[8] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 647–658, 2004.

[9] 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.

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

[11] 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.

[12] 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.

[13] H. Mannila and K.-J. Räihä. On the complexity of inferring functional dependencies. *Discrete Appl. Math.*, 40:237–243, 1992.

[14] 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.

[15] N. Novelli and R. Cicchetti. Functional and embedded dependency inference: a data mining point of view. *Information Systems*, 26(7):477–506, 2001.

[16] J. C. Schlimmer. Efficiently inducing determinations: A complete and systematic search algorithm that uses optimal pruning. In *In Proceedings of the International Conference on Machine Learning (ICML)*, pages 284–290, 1993.

[17] Tane home page. `www.cs.helsinki.fi/research/fdk/datamining/tane/`. Accessed: 2015-02-26.

[18] UCI machine learning repository. `http://archive.ics.uci.edu/ml`. Accessed: 2015-02-26.

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

[20] 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.

[21] H. Yao and H. J. Hamilton. Mining functional dependencies from data. *Data Mining and Knowledge Discovery*, 16(2):197–219, 2008.

[22] 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.