Approximate Discovery of Functional Dependencies for Large Datasets

Tobias Bleifuß[1] Susanne Bülow[1] Johannes Frohnhofen[1]
Julian Risch[1] Georg Wiese[1] Sebastian Kruse[2]
Thorsten Papenbrock[2] Felix Naumann[2]
[1] firstname.lastname@student.hpi.de
[2] firstname.lastname@hpi.de
Hasso-Plattner-Institut, Prof.-Dr.-Helmert-Str. 2–3, 14482 Potsdam, Germany

ABSTRACT

Functional dependencies (FDs) are an important prerequisite for various data management tasks, such as schema normalization, query optimization, and data cleansing. However, automatic FD discovery entails an exponentially growing search and solution space, so that even today's fastest FD discovery algorithms are limited to small datasets only, due to long runtimes and high memory consumptions.

To overcome this situation, we propose an approximate discovery strategy that sacrifices possibly little result correctness in return for large performance improvements. In particular, we introduce Aid-Fd, an algorithm that approximately discovers FDs within runtimes up to orders of magnitude faster than state-of-the-art FD discovery algorithms. We evaluate and compare our performance results with a focus on scalability in runtime and memory, and with measures for completeness, correctness, and minimality.

1. FD DISCOVERY

Functional dependencies (FDs) are among the most important integrity constraints in relational databases. They are a prerequisite for various core data management tasks, such as schema normalization and query optimization [5,15].

Given a relational instance r of schema R, the FD $X \to A$ states that the values of an attribute set $X \subseteq R$ determine the value of a single attribute $A \in R$ in r. Formally, the FD $X \to A$ holds on r iff any two tuples agreeing in their values in X also agree in their values in A: $\forall t_1, t_2 \in r\colon t_1[X] = t_2[X] \implies t_1[A] = t_2[A]$. X is called the FD's left-hand side (LHS) and A is called right-hand side (RHS), respectively. A *minimal* FD is an FD $X \to A$ whose left-hand side X is minimal, which means that no FD $X' \to A$ with $X' \subset X$ holds. An FD is *non-trivial*, if its attributes on the LHS and RHS are disjoint. For instance, in Table 1, the values of C determine the values of A, i.e., $C \to A$ holds. Similarly, the values of A and B combined determine the values of C, yielding the FD $AB \to C$. Note that these FDs are the only minimal, non-trivial FDs that hold in this instance.

Table 1: Example dataset with $C \to A$ and $AB \to C$.

	A	B	C
t_1	1	1	1
t_2	1	2	2
t_3	1	3	1
t_4	2	2	3

FD discovery is the problem of finding all minimal, non-trivial functional dependencies that hold in a given relational instance. The major obstacle to this problem is its exponential complexity: The number of FD candidates is $\sum_{k=1}^{|R|} \binom{|R|}{k} \cdot (|R| - k)$, which makes enumerating the entire search space infeasible for large datasets [10]. Existing algorithms for FD discovery, therefore, apply various loss-free pruning rules to reduce the size of the search space. Still, these algorithms cannot process datasets of real-world size as shown by Papenbrock et al. [13]: Their evaluation of state-of-the-art FD discovery algorithms has demonstrated that all current, exact techniques fail to provide a result for datasets with more than 30 attributes and 250,000 tuples within reasonable time and memory limits.

To enable FD discovery on larger datasets, we suggest approximating the discovery. While the approximation might sacrifice completeness and correctness of the computed FD result sets, it can considerably reduce the usage of computation resources, in particular time and memory. Approximate FD result sets are still useful for most use-cases that utilize FDs. Especially interactive applications, such as data exploration, prefer a fast analysis over a perfectly correct result. Moreover, an approximated result is more valuable than not being able to calculate any result.

AID-FD. We propose the approximate FD discovery algorithm Aid-Fd (Approximate Iterative Discovery of FDs). Exact state-of-the-art FD discovery algorithms can process datasets with up to 250,000 rows and 30 columns; Aid-Fd, in contrast, processes the same datasets using only 2 % to 40 % of the time of the exact algorithms, while still discovering more than 99 % of all FDs. The approximation allows Aid-Fd to very efficiently handle much larger datasets, which no known algorithm is able to process within several hours.

Aid-Fd comprises three phases. At first, it loads the input relation into data structures to efficiently retrieve sets of tuples that share a value in a particular attribute.

CIKM '16 October 24–28, 2016, Indianapolis, IN, USA

DOI: http://dx.doi.org/10.1145/2983323.2983781

Secondly, Aid-Fd approximates the *negative cover*, which is a representation of the set of all non-FDs. For this purpose, our algorithm uses an incremental, focused sampling of tuple pairs to deduce non-FDs from them. The sampling stops, once a user-configured termination criterion is met, which can either be a fix time limit or a desired correctness.

In the third phase, Aid-Fd inverts the negative cover into the *positive cover*, which holds the alleged minimal, non-trivial FDs. We show that our inversion algorithm is up to 40 times faster compared to the best previous algorithms. Because our inversion is exact, it can also be used in some of the exact FD discovery algorithms, e.g., FDEP [6] and Dep-Miner [11], and improve their performance.

We exhaustively evaluate and compare the performance of Aid-Fd with regard to exact FD discovery algorithms, in terms of both efficiency and result quality. To make precise statements about the latter, we introduce three complementary quality measures, *completeness*, *correctness*, and *minimality*, each addressing a specific approximation dimension.

Structure. In Section 2 we discuss related work and compare our algorithm to other approaches for the exact and the approximate discovery of FDs. Section 3 defines the different dimensions of approximation and introduces measures for correctness, completeness, and minimality. We then describe our Aid-Fd algorithm in detail in Section 4, followed by an evaluation and a scalability discussion in Section 5.

2. RELATED WORK

The discovery of FDs is an extensively studied data profiling task. Abedjan et al. provide an overview of different profiling tasks including a classification of existing algorithms for FD discovery [1].

Exact discovery. *Column-based* FD discovery algorithms traverse all possible FD candidates through an attribute lattice. The most efficient column-based algorithm is either DFD [2] or TANE [7], depending on the dataset [13]. TANE traverses the candidate lattice bottom-up in a level-wise manner with an apriori-like candidate generation. DFD also uses the candidate lattice but traverses it with a depth-first random walk and additional top-down pruning. Both algorithms check candidates by partition intersections and refinement checks. The runtime of these and other column-based algorithms is dominated by the number of columns.

Row-based FD discovery algorithms compare tuples from the relation to build so-called difference and agree sets. From these sets, they derive all valid FDs. The derivation step differs for each algorithm. FDEP [6], the most efficient row-based algorithm [13], starts with a set of most general FDs and incrementally specializes them. The complexity of row-based algorithms depends on the number of tuple comparisons, and thus the number of rows.

Hybrid FD discovery algorithms such as HyFD [14] combine row- and column-based strategies. Similar to our algorithm, they free the user from choosing the most efficient discovery strategy. Because the number of rows and columns in a dataset determines the fastest related algorithm, we compare Aid-Fd with always the most efficient competitor among TANE, DFD, and FDEP.

Approximate discovery. Because exact discovery algorithms do not scale to the size of many real-world datasets, *approximating* FD discovery has been proposed. Kivinen et al. provide a basic algorithm of inferring FDs with an arbitrary precision [9]. Furthermore, they present a theoretical analysis of the sample size and the consequent correctness estimation of the found FDs. This approach searches for valid FDs by the random generation of candidates. In case a generated candidate does not hold, it is used to refine the current result set of FDs. As the authors state themselves, this approach is inefficient in finding FDs with many attributes on the left-hand side due to the random generation of candidates. Because most discovered FDs, in fact, do have large left-hand sides, we do not follow their approach.

Another approximate FD discovery algorithm, CORDS, is intended to find FDs for query optimization [8]. It first generates candidates based on heuristics, such as the number of distinct values and data types. Then, the algorithm samples rows for each candidate until either a counter-example is found or the probability that the columns are dependent exceeds a certain threshold. Brown et al. propose another approach called B-Hunt also aiming at query optimization [4]. B-Hunt's candidate generation is similar to CORDS, but for candidate testing B-Hunt uses algebraic constraints, such as clustering or segmentation on a sample of the input relation. Both CORDS and B-Hunt are limited to finding FDs with a single column in the LHS in contrast to our algorithm, which aims at finding *all* FDs holding in a relation.

Partial discovery. In contrast to the approximate discovery of FDs, *partial* FD discovery algorithms are exact algorithms that allow a certain violation of the FD property, i.e., they discover FDs that are only partially fulfilled by the inspected data and the error is known. Algorithms for finding partial FDs, which are also called "soft FDs" [8] or "approximate dependencies" [7, 12], are often adaptations of exact algorithms and have even longer runtimes than exact algorithms, because allowing a certain error introduces an additional overhead and many pruning rules do not work with error tolerance. Therefore, the discovery of partial FDs is orthogonal work. Incorrect FDs reported by approximate discovery algorithms can, however, be partial FDs.

3. DIMENSIONS OF APPROXIMATION

An exact FD discovery algorithm returns *all* and *only* minimal, non-trivial FDs of a given relational instance. We call this set of FDs the *gold standard*, short *Gold*, and write $Gold^+$ to denote the set of all FDs that can be inferred from *Gold* using the Armstrong Axioms [3].

We call an FD discovery algorithm *approximate* if it cannot *guarantee* to find the gold standard for every possible input relation; it might miss some actual FDs or incorrectly report non-FDs as FDs. Notice that approximation is a feature of FD result sets but not of single FDs: Whether or not a non-FD has only few violations (we refer to this as partial FDs) is not relevant. Instead, we are interested in approximating the set of violation-free FDs.

In this section, we distinguish three independent dimensions that make up an exact FD result set: completeness (*does the result contain all true FDs?*), correctness (*does the result contain true FDs only?*), and minimality (*are the reported correct FDs also minimal?*). An approximate algorithm can relax one or more of these dimensions to speed up discovery. Depending on the FDs' use-case, different relaxations are tolerable. Table 2 shows example outputs for all possible combinations of the three relaxation dimensions and the example relation of Table 1.

Table 2: Three dimensions of approximating FDs.

Completeness	Correctness	Minimality	Example (wrt. Table 1)	Example Method
$= 1$	$= 1$	$= 1$	$AB \rightarrow C$, $C \rightarrow A$	exact discovery
		< 1	$AB \rightarrow C$, $C \rightarrow A$, $CB \rightarrow A$	test everything
	< 1	$= 1$	$AB \rightarrow C$, $C \rightarrow A$, $B \rightarrow A$	
		< 1	$AB \rightarrow C$, $C \rightarrow A$, $B \rightarrow A$, $CB \rightarrow A$	
< 1	$= 1$	$= 1$	$C \rightarrow A$	column sampling
		< 1	$AB \rightarrow C$, $BC \rightarrow A$	generate & test
	< 1	$= 1$	$B \rightarrow A$, $C \rightarrow A$, $A \rightarrow C$, $B \rightarrow C$	row sampling
		< 1	$BC \rightarrow A$, $C \rightarrow B$	random guessing

In the following, we discuss these three dimensions in detail. For each dimension, we propose a quality measure that evaluates the performance of an approximate FD discovery algorithm regarding this dimension. The measures adapt the widely used measures *precision* and *recall* to accommodate the fact that FDs can imply one another. Let *Out* be the result set of FDs returned by an approximate FD discovery algorithm. All given measures cover the range $[0, 1]$ with 1.0 being the best possible value.

Definition 1. Completeness *of a result is the share of discovered minimal, non-trivial FDs of a relation's gold standard FDs. A complete result set is equal to or a superset of the gold standard. Formally, completeness is defined as the fraction of FDs in Gold that are also contained in Out:*

$$\frac{|Out \cap Gold|}{|Gold|}$$

Potentially incomplete results can efficiently be obtained by, for example, executing an exact FD algorithm on a subset of the relation's attributes (*column sampling*): If only the attributes A and C of the example relation from Table 1 are considered, the correct FD $AB \rightarrow C$ cannot be discovered. Query optimization is one scenario for which incomplete FD sets are still very useful: A missing FD might cause some lost optimization potential, but query optimization works fine with only some available FDs.

Definition 2. Correctness *of a result is the share of discovered (not necessarily minimal) FDs of all discovered FDs for a given relational instance. We define correctness as the fraction of FDs in Out that hold in the given relation. Because non-minimal FDs in Out are also considered correct, we use the gold standard's transitive closure $Gold^+$:*

$$\frac{|Out \cap Gold^+|}{|Out|}$$

An FD result can become incorrect if we, for instance, execute an exact algorithm on a subset of tuples (*row sampling*). If only the single tuple t_2 in Table 1 is excluded from the discovery, then incorrect FDs, such as $B \rightarrow A$, are included in the result. Potentially incorrect FDs are for instance still valuable for query optimization where a few incorrect FDs might lead to incorrect cost estimations for plan alternatives, yet, the many correct FDs help improving the overall accuracy of the estimations.

Definition 3. Minimality *of a result is the degree of correct FDs in the result set that are also minimal. Formally,* minimality *is defined as the number of minimal FDs in Out normalized by the number of correct FDs in Out. Again, non-minimal FDs are also considered correct:*

$$\frac{|Out \cap Gold|}{|Out \cap Gold^+|}$$

When, for instance, generating candidate FDs from a set of rules or heuristics and then testing these candidates against a relation (*generate & test*), we cannot guarantee the FDs' minimality: This approach could generate the FD $BC \rightarrow A$, find that it holds in our example relation, and add it to the result set although it is non-minimal. Non-minimal FDs serve scenarios such as schema matching that search for corresponding FDs across datasets in order to identify matching schema elements; the FDs do not need to be minimal if only their attributes match.

4. AID-FD

In the following, we give a detailed description of our Aid-Fd algorithm. The algorithm conducts a *focussed sampling of tuple pairs* to approximate the *negative cover*. In other words, we try to rule out as many FDs as possible without comparing all tuple pairs as done by exact row-based algorithms. In Table 2, Aid-Fd thus belongs to the category labeled as "row sampling". By inverting the approximate negative cover, the algorithm deduces an approximate set of minimal FDs for the given relational instance.

Aid-Fd is a 3-phase algorithm as illustrated in Figure 1. During the *Attribute Indexing* phase, the algorithm ingests the dataset and initializes the data structures needed in the later phases (Section 4.1). The *Negative Cover Creation* phase builds the negative cover using approximation techniques (Section 4.2). The *Negative Cover Inversion* phase inverts the negative cover in an exact manner, which yields an approximation to the set of minimal FDs (Section 4.4).

Notation. Throughout this section, $R(A, B, C)$ denotes a sample relation R with the attributes A, B, and C. Variables denoted by lower-case letters, such as *rhs* and *add*, store single attributes. Sets of attributes (subsets of R) are implemented as bitsets and denoted by a single upper-case letter (other than A, B, C), such as N or X. Sets of attribute sets are either implemented as hash sets or a prefix tree structure, which is introduced later. These sets of sets are denoted by calligraphic letters, such as $\mathcal{P}$ or $\mathcal{N}$.

4.1 Attribute indexing

The purpose of the attribute indexing is to load the input relation into memory-efficient data structures that provide fast access to all information relevant for the later phases. This allows discarding the actual data values afterwards.

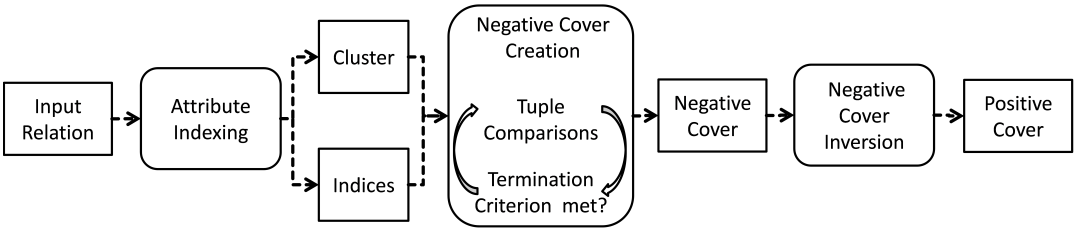

Figure 1: Overview of the AID-FD algorithm workflow.

First, AID-FD builds *clusters* for the given dataset. A cluster is an ordered set of tuples that share a value in a particular attribute. For instance, in Table 1, attribute A partitions the tuples into clusters $c_{A1} = \{t_1, t_2, t_3\}$ and $c_{A2} = \{t_4\}$. This set of clusters for attribute A is called the partition of A. The *clusters* table in Table 3 reflects this partitioning. It is an $|r| \times |R|$ table that points each cell in the relation to the corresponding cluster, e.g., $clusters[t_1, A]$ points to c_{A1} for the example dataset. Furthermore, AID-FD builds an $|r| \times |R|$ *indices* table, which is also shown in Table 3 for the example dataset. For each cell, this table stores the cell's *index* within its cluster. In other words, $indices[t, a]$ stores how many tuples in $clusters[t, a]$ have a smaller index than t. Note that the clusters are ordered so that tuples have fixed indices within their clusters.

Table 3: The *clusters* and *indices* tables and the actual cluster values for the example dataset.

clusters

	A	**B**	**C**
t_1	c_{A1}	c_{B1}	c_{C1}
t_2	c_{A1}	c_{B2}	c_{C2}
t_3	c_{A1}	c_{B3}	c_{C1}
t_4	c_{A2}	c_{B2}	c_{C3}

indices

	A	**B**	**C**
t_1	0	0	0
t_2	1	0	0
t_3	2	0	1
t_4	0	1	0

actual cluster values

A	$c_{A1} = \{t_1, t_2, t_3\}$, $c_{A2} = \{t_4\}$
B	$c_{B1} = \{t_1\}$, $c_{B2} = \{t_2, t_4\}$, $c_{B3} = \{t_3\}$
C	$c_{C1} = \{t_1, t_3\}$, $c_{C2} = \{t_2\}$, $c_{C3} = \{t_4\}$

For any tuple t, AID-FD uses these data structures to efficiently retrieve all clusters that t intersects with, i.e., the set of tuples that share at least one value with t and are therefore candidates for tuple comparisons. Furthermore, AID-FD can directly obtain the tuple indices w.r.t. those clusters. This allows to restrict the symmetric tuple comparison operation to those tuples t_i and t_j where $i < j$.

As an additional preprocessing step, AID-FD detects *constant* columns that contain only a single distinct value. The set of constant columns S is stored, because any minimal FD involving constant columns can be inferred in the *Negative Cover Inversion* phase without access to the other columns. For that reason, AID-FD can safely ignore constant columns during the following *Negative Cover Creation* phase.

4.2 Negative cover creation

This phase of AID-FD approximates the *negative cover* of a given relational instance r. The negative cover is the set of all non-FDs in r and can be found by comparing tuples pairwise [6]. So if R is the schema of r, the *agree set* $ag(t_1, t_2)$ of two tuples t_1 and t_2 from r is defined as the largest subset $X \subseteq R$, such that $t_1[X] = t_2[X]$ [11]. From the agree set, we can infer that $ag(t_1, t_2) \nrightarrow A$ is a non-FD for each $A \in R \setminus ag(t_1, t_2)$. In Table 1, for instance, $ag(t_1, t_2) = \{A\}$, from which follows that $A \nrightarrow B$ and $A \nrightarrow C$. Hence, we represent the negative cover as a set of agree sets.

The *tuple comparison* operation involves finding the agree set of two tuples t_1 and t_2 and adding the resulting non-FDs to the negative cover. It has the following properties:

(1) *Reflexivity:* Because $ag(t, t) = R$ for all tuples t, no non-FDs can be derived by comparing tuples to themselves.

(2) *Symmetry:* Because $ag(t_1, t_2) = ag(t_2, t_1)$ for all tuples t_1 and t_2, one comparison per tuple pair is sufficient.

(3) *Significance:* The empty agree set can yield only non-FDs $\emptyset \nrightarrow A$ with $A \in R$. This is equivalent to stating that no column is constant, which is true during negative cover creation anyway (see Section 4.1). Hence, we only compare tuples that share at least one value.

As a result, at most $\frac{|r|(|r|-1)}{2}$ tuple comparisons have to be made to obtain an exact negative cover. Still, the number of tuple comparisons of an exact algorithm is in $O(|r|^2)$. AID-FD circumvents this quadratic complexity by iteratively picking and comparing only promising tuple pairs, thereby incrementally creating the negative cover $\mathcal{N}$. As shown in Algorithm 1, this process is perpetuated until some termination criterion is met (Line 3). Possible criteria are discussed in Section 4.3. In each iteration, the algorithm runs over all tuples, and for each current tuple t it chooses some other tuples to compare t with (Lines 4–6).

The function *make_ith_checks(t,i)* of Algorithm 1 describes how the other tuples for tuple t and iteration i are chosen. It iterates over all clusters that intersect with tuple t, which is one per attribute (Line 10). Then, a pseudo-random permutation $prp()$ is calculated to pick t' from the previous tuples in the current cluster, which is afterwards compared to t (Lines 11–15). This proceeding ensures that AID-FD compares only tuples that have at least one value in common and do not yield empty agree sets. Moreover, duplicate comparisons are avoided because t' has a smaller cluster index than t. Concretely, $prp(index, i)$ returns the i^{th} number of a pseudo-random permutation of all numbers n with $0 \leq n < index$ by using a large prime number *prime*:

$$prp(index, i) = (i \cdot prime) \bmod index$$

While picking tuples in their plain order would favor tuples with small indices, the pseudo-random permutation ensures that no tuple pair is more likely chosen for comparison. Using a random permutation rather than a random number generator ensures that no other tuple is picked twice from

Algorithm 1: Build negative cover.

Data: A relation schema R and a relation instance r
Result: A negative cover $\mathcal{N}$

```
1  N = ∅
2  i ← 1
3  while Termination criterion not met do
4      for t ∈ r do
5          make_ith_checks(t, i)
6      i ← i + 1
7  return N
8
9  Function make_ith_checks(t, i)
10     for a ∈ R do
11         cluster ← clusters[t, a]
12         index ← indices[t, a]
13         if i ≤ index then
14             other_cluster_index ← prp(index, i)
15             t' ← cluster[other_cluster_index]
16             N ← N ∪ ag(t, t')
```

the same cluster. This also ensures that all previous tuples have already been compared to the current tuple on previous iterations exactly if $i > index$. If this is the case, the tuple check for this cluster can be skipped (Line 13).

Finally, Aid-Fd compares the two chosen tuples and adds the result to the negative cover (Line 16), which is represented by the set of agree sets. Agree sets are represented as *bitsets*, i.e., bit vectors with one bit per attribute, where the i-th bit is set if and only if the corresponding agree set contains the i-th attribute. This negative cover representation is more space-efficient than storing non-FDs in a prefix tree as proposed in [6].

Note that if no termination criterion is defined, the result of the described algorithm is guaranteed to converge to the exact negative cover, because eventually all relevant tuple comparisons are executed.

Example 1. *The following table illustrates the make_ith_checks() function of Algorithm 1 for $t = t_3$ and $i = 1$. On attribute A, t_3 is in cluster $c_{A1} = \{t_1, t_2, t_3\}$ at index $= 2$, so either t_1 or t_2 is chosen as t'. In this example, let $prp(index, i) = prp(2, 1) = 1$, so t_2 is picked as t'. On attribute B, index $= 0 < i$, so no comparisons are executed. On attribute C, t_3 is in cluster $c_{C1} = \{t_1, t_3\}$ at index $= 1$, so t_1 is the only tuple that can be picked for comparison. In consequence, t_3 is compared to t_1 and t_2 during iteration 1.*

	A	B	C
t_1	1	1	**1**
t_2	**1**	2	2
t_3	**1**	**3**	**1**
t_4	2	2	3

4.3 Termination criteria

As stated above, the creation of the negative cover is an iterative process and each iteration performs a number of tuple comparisons. After a certain number of iterations, Aid-Fd would retrieve the exact set of FDs that hold for the input relation. However, assuming that the later iterations contribute only little or even no new non-FDs to the negative cover, our algorithm skip those, thereby not necessarily yielding the exact result, but an approximate result in noticeably shorter time. In this section, we discuss two termination criteria, an *effort-driven* and a *quality-driven* one, for the negative cover creation.

Fixed time. This termination criterion stops the creation of the negative cover, when a user-defined amount of time has elapsed. The actual number of iterations executed within this time interval of course varies with the dataset, because the time needed for each iteration depends on the number of rows and columns of the input relation. Note that the fixed time applies only to the time needed for the creation of the negative cover; the total execution time of the algorithm still exceeds this time due to the subsequent negative cover inversion (see Section 4.4). The inversion time is hard to predict, because it does not depend on the input relation's dimensions, but on the number and nature of the FDs in the output. Thus, Aid-Fd as a whole is not designed to terminate its execution within this given time limit.

Fixed negative cover growth. Observing the negative cover size after each iteration shows that, at first, the negative cover grows rapidly, then, its growth decreases continuously and, finally, the size of the negative cover converges against its real size. This seems to support our assumption that the later iterations in the negative cover creation do not provide much new information. However, because the final size of the negative cover is unknown during its creation, the algorithm cannot know at runtime if the negative cover size has converged. Yet, it is possible to regard the *growth* of the negative cover size after k iterations as

$$growth_k = \frac{|\mathcal{N}_k|}{|\mathcal{N}_{k-1}|} - 1 \tag{1}$$

where $|\mathcal{N}_k|$ is the size of the negative cover after the kth iteration. Figure 2 shows the development of the negative cover growth together with the result's completeness and correctness for the *ncvoter* dataset, which we introduce later. The plot indicates that the growth decreases with each iteration while completeness and correctness grow correspondingly.

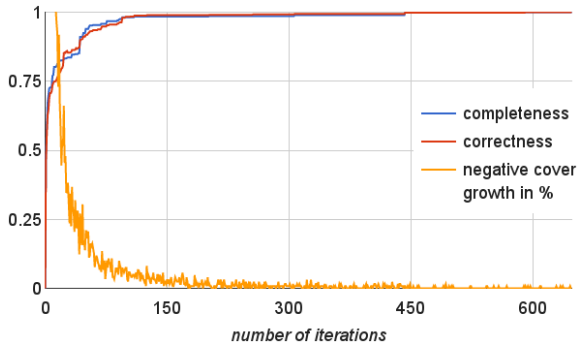

Figure 2: Negative cover growth, correctness, and completeness by iteration for ncvoter (Sec. 5.1).

We observe that the closer the algorithm comes to the final size of the negative cover, the smaller the growth of the negative cover size becomes. For this reason, the growth rate of the negative cover serves as a good termination criterion (see Section 5.6). So we define a threshold for the growth and the algorithm stops iterating when the growth of the

negative cover falls below this threshold. The smaller the growth threshold is set, the higher the quality of the result becomes and the longer Aid-Fd runs.

Considering only the growth of the negative cover from one iteration to the next, as Equation 1 does, is sensitive to single iterations that increase the negative cover only little; even though the growth may be zero in one iteration, it is still possible and not unlikely to find a new non-FD a few iterations later. Therefore, the Aid-Fd algorithm observes the growth within a sliding window of size w that stores the growth for the last w iterations. With this technique, we can capture general trends of negative cover growth much better. The size of this window defines sensitivity of the algorithm: It reacts quickly to changes using a window of size 1 and it reacts much slower using a window of size 10. Note that the negative cover might still grow, even though all correct FDs in the dataset can already be derived from it, because new non-FDs might be implied by previously found non-FDs; a growth of the negative cover, hence, does not necessarily entail a change in the final result. So using a very large window can result in unnecessary long runtimes.

4.4 Negative cover inversion

To obtain the valid FDs, Aid-Fd inverts the negative cover similarly to [6]: Each non-FD is used to identify invalid FD candidates and to turn them into valid specializations, i.e., their left-hand sides are augmented. We formalize this problem as follows. The result of the negative cover creation is a set $\mathcal{N}$ of agree sets, each of which is represented by a bitset and corresponds to a set of non-FDs. Algorithm 2 uses $\mathcal{N}$ to calculate the corresponding positive cover: It iterates over all right-hand sides $rhs \in R$ and constructs a set of left-hand sides $\mathcal{P}$, such that for every left-hand side $L \in \mathcal{P}$ the FD $L \to rhs$ is minimal, non-trivial and not ruled out by the negative cover. The latter means that there exists no agree set $N \in \mathcal{N}$ such that $L \subseteq N$ and $rhs \notin N$. Otherwise $N \nrightarrow rhs$ (and therefore $L \nrightarrow rhs$) would hold.

Algorithm 2 first handles constant columns: For every constant column $const \in S$ an FD $\emptyset \to const$ is added to the output set Ω (Line 1). This covers all minimal FDs that contain a constant column, because constant columns cannot be part of the LHS of a minimal FD. Therefore, these attributes can be ignored subsequently and are removed from the set of attributes R in Line 2. The purpose of sorting the negative cover $\mathcal{N}$ in Line 3 is discussed later.

Line 4 iterates over all remaining, non-constant columns $rhs \in R$. For each rhs, Aid-Fd calculates the set of minimal, non-trivial FDs that do not contradict $\mathcal{N}$ and determine the column rhs:

(1) The most general FDs are assumed to be valid: The current rhs is thought to be determined by each non-constant column $attr \neq rhs$. For every such column, a singleton $\{attr\}$ is added to the set of left-hand sides $\mathcal{P}$ in Line 5.

(2) The algorithm iterates over all agree sets N in the negative cover $\mathcal{N}$ (Line 6). If N does not contain rhs, N is used to specialize the current set of left-hand sides $\mathcal{P}$ by calling the function `handleNonFD` (Lines 7 & 8).

(3) After handling the complete negative cover for this rhs, Line 9 adds one FD $L \to rhs$ for every remaining left-hand side L in $\mathcal{P}$ to Ω.

The function `handleNonFD` accepts three arguments: The bitset N, representing a non-FD $N \nrightarrow rhs$, the set of left-hand sides $\mathcal{P}$, and a set of attributes X, which contains all non-constant attributes except for rhs. These are all attributes that can be added to left-hand sides L for this rhs, as L would not be minimal if it contained constant columns, and trivial if it contained rhs. In Line 13, all subsets of N are retrieved from $\mathcal{P}$. These are the left-hand sides, which are invalidated by the non-FD $N \nrightarrow rhs$ and are therefore removed from the set of left-hand sides $\mathcal{P}$ (Line 14).

Next, Aid-Fd adds new left-hand sides, which are not invalidated by N, but that were not minimal until now, because they are supersets of one of the removed sets $L \in \mathcal{S}$. The algorithm creates supersets $L \cup \{add\}$ for every removed left-hand side L in the set of removed left-hand sides $\mathcal{S}$ and every column $add \in (X \setminus N)$. It is necessary to add at least one attribute $add \notin N$ to ensure $L \cup \{add\}$ is no subset of N, which would invalidate $L \cup \{add\} \to rhs$. Furthermore, it is necessary to add at most one attribute to keep minimality. These supersets are added to the set of left-hand sides $\mathcal{P}$ only if there is no subset of these supersets in $\mathcal{P}$ (Line 17), as they would not be minimal otherwise.

When every non-FD has been handled for one rhs, the FDs represented by $\mathcal{P}$ are non-trivial, because the algorithm ensures that the current rhs is never added on the left-hand side: The initial set of left-hand sides $\mathcal{P}$ excludes rhs (Line 5) and rhs is also excluded from the set X, which is passed to `handleNonFD`. Therefore, rhs cannot be added to any left-hand side. All generated FDs are also correct, meaning that they do not contradict any non-FD: Each iteration ensures that no L in $\mathcal{P}$ is invalidated by the current non-FD by removing all subsets of it. The systematic extension process of sets in $\mathcal{P}$ guarantees that any newly added set is not a subset of any non-FD that we have seen so far. It follows that when the algorithm handled all non-FDs, no set in $\mathcal{P}$ can contradict a non-FD in $\mathcal{N}$.

Throughout each iteration of the loop in Line 4 the following property holds:

Algorithm 2: Phase 2: Negative cover inversion.

```
   Data: negative FD cover N, set of attributes R, set of
         constant columns S
   Result: the set of minimal, non-trivial FDs Ω
1  Ω ← {∅ → const | const ∈ S}
2  R ← R \ S
3  N ← sort(N)
4  for rhs ∈ R do
5      P ← {{attr} | attr ∈ R, attr ≠ rhs}
6      for N ∈ N do
7          if rhs ∉ N then
8              handleNonFD (N, P, R \ {rhs})
9      Ω ← Ω ∪ {L → rhs | L ∈ P}
10 return Ω
11
12 Function handleNonFD(N, P, X)
13     S ← {P ∈ P | P ⊆ N}
14     P ← P \ S
15     for L ∈ S do
16         for add ∈ (X \ N) do
17             if ∀P ∈ P: P ⊈ (L ∪ {add}) then
18                 P ← P ∪ {L ∪ {add}}
```

PROPERTY 1. *If $X \rightarrow rhs$ is a valid, non-trivial FD, $\mathcal{P}$ always contains a subset L of X.*

Property 1 obviously holds at the initialization of $\mathcal{P}$ in Line 5. If at some point the last subset of X is removed in Line 14, then a new subset of X would be added in Line 18, as at least one superset must not be contradictory to N, otherwise $X \rightarrow rhs$ would not be valid. This property immediately leads to completeness in the sense that all valid FDs can be derived from the output. The FDs are also minimal, because we start with the most general FDs and add new left-hand sides only if there are no subsets of them in $\mathcal{P}$. If that is the case, none of the subsets is a valid left-hand side for this rhs and the newly added left-hand side is therefore still minimal.

Managing attribute sets. The most expensive operation in the process of inverting the negative cover is to look for subsets in the set of left-hand sides $\mathcal{P}$. We propose a binary tree to support this operation. Its leaves store the actual attribute sets and its internal nodes divide these sets into those containing a certain attribute and those that do not. Figure 3 exemplifies such a tree. Its root node divides on attribute a, so all leaves on the left subtree do not contain a while all leaves on the right side do. Thus, in a subset query for a set that does not contain a, only the children on the left need to be searched further. Additionally, every internal node stores the intersection set of all its successors. If an internal node with an intersection set, which is no subset of the query set, is encountered, then there is no need to look at any of its children. If the example tree of Figure 3 is queried for a subset of $\{a, c, e\}$ all children of the internal node c could be skipped, because the intersection $\{a, b\}$ is no subset of $\{a, c, e\}$.

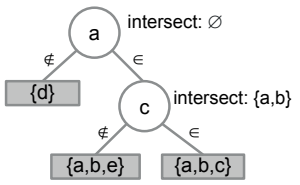

Figure 3: Binary tree for faster subset lookup. This instance stores the sets $\{a, b, c\}$, $\{a, b, e\}$ and $\{d\}$.

Further, it is interesting whether the order of attributes in the tree matters. Indeed, we observed that splitting the attributes in the order of ascending frequency in the negative cover yields faster runtimes than a random order, as shown in the evaluation (Section 5.7). This improvement is achieved, because splitting first on rare attributes allows skipping large subtrees (the right side) when searching for sets not containing these attributes. As these attributes are rare in the negative cover, most of the subset queries in Line 13 do not contain them.

As mentioned earlier, Algorithm 2 sorts the negative cover $\mathcal{N}$ in Line 3. Specifically, we sort the agree sets by their size in descending order. The sorting aims to keep the tree small for as long as possible and to reduce the number of modifications, which strongly impact the algorithm's performance. Section 5.7 evaluates the effect of the sorting.

Due to the fact that the positive cover is created successively for every rhs, the algorithm can skip any non-FD $N \in \mathcal{N}$ with $rhs \notin N$ and for which $\mathcal{N}$ also contains a superset $N' \supset N$ with $rhs \notin N'$. Every left-hand side that would have been marked invalid by N is also marked invalid by the superset N'. As the non-FDs are sorted in a way that a subset of a non-FD never appears after its supersets, a check of whether a superset has been seen yet is sufficient. For this check we use the same data structure as for storing the positive cover. This modification also has a positive impact on performance as it reduces the number of subset lookups on the positive cover.

In contrast to [6], our cover inversion procedure stores the left-hand sides of only one right-hand side at a time. Thus, and with the bitset representation of the negative cover, our approach consumes less main memory. Our experiments also show that the proposed inversion process scales better with the increasing number of columns.

5. EVALUATION

In the following section, we evaluate both AID-FD's performance and the quality of its FD result sets. At first, we compare the runtimes of our algorithm on different levels of completeness and correctness to the best known exact algorithms, TANE, DFD, and FDEP. Then, we investigate AID-FD's scaling behavior to understand how far it can be adopted to large datasets. Finally, we evaluate individual algorithm components in more detail. Overall, we find that AID-FD is a viable replacement for exact algorithms for the FD discovery in medium-sized and large datasets whenever approximate results are sufficient.

5.1 Setup and datasets

All our experiments have been conducted on a machine with a 3.50 GHz Quad Core i5-4690 processor, 8 GB of main memory, 440 GB SATA SSD, Ubuntu 14.04.3 64-bit, and Java 8.0 (64-bit). We used a variety of datasets from different domains, with different sizes w.r.t. both columns and rows, and with widely varying numbers of FDs, as summarized in Table 4. Even though our largest dataset, *ncvoter*, has "only" about 1 million rows, we note that none of the existing FD discovery approaches has been able to scale to that volume, as we show in the next section. To make our experimental results reproducible, AID-FD and all compared algorithms and datasets are available online[1]. Note that all our measurements cover the entire runtime including I/O time and time needed for creating intermediate data structures such as indexes.

Table 4: Datasets and their characteristics.

dataset	rows	columns	$\|Gold\|$
ncvoter_64k	64,000	19	646
uniprot	256,000	30	4,561
letter	20,000	17	61
plista_1k	1,000	63	178,152
fd-reduced-30	250,000	30	89,571
flight_500k	500,000	20	290
ncvoter	1,024,001	19	568

5.2 Comparison to exact algorithms

Figure 4 compares the runtimes of the fastest known exact algorithms to runtimes of AID-FD for different degrees

[1]https://hpi.de/naumann/projects/repeatability/data-profiling/fd.html

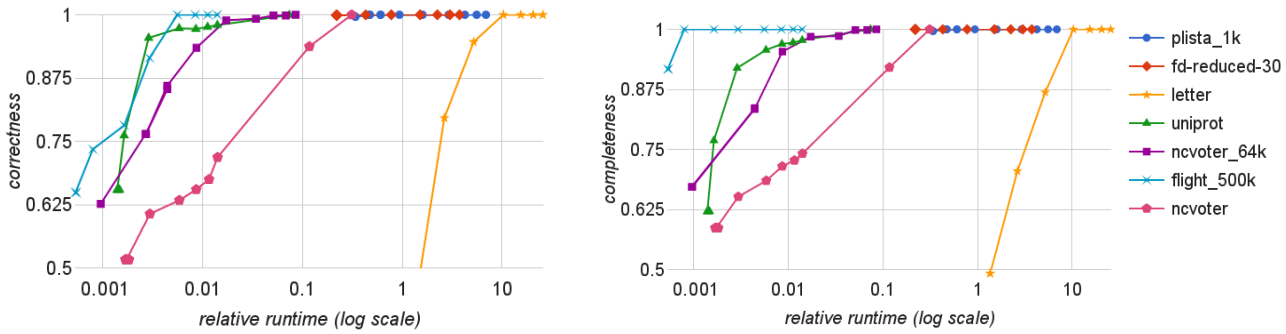

Figure 4: Aid-Fd's runtime compared to fastest exact algorithms, and observed correctness and completeness.

of correctness and completeness. For each data set, we compared the runtime against the fastest out of DFD, TANE, and FDEP. The fastest exact algorithms for the different datasets are FDEP for *ncvoter_64k* (1177.0s) and *plista_1k* (15.1s), DFD for *letter* (3.9s), and TANE for *fd-reduced-30* (26.7s). For *uniprot*, *flight_500k*, and *ncvoter*, no exact algorithm could calculate a result within two hours, so we took two hours as a reference runtime for them.

The execution of the negative cover creation of Aid-Fd was interrupted after different time periods between 100ms and 30min to compare the result quality after different runtimes. Figure 4 shows that even after a short amount of time, the algorithm reaches completeness and correctness values close to 1.0. That is, Aid-Fd can produce correct results in only a fraction of the time needed by exact algorithms. Furthermore, Aid-Fd sustains its performance on datasets that exact algorithms cannot handle due to memory or runtime limits: None of FDEP, DFD, and TANE are able to handle the *uniprot*, *flight_500k*, and *ncvoter* datasets on the test machine within a time limit of 2 hours. To be able to state relative runtimes of Aid-Fd, we use the 2 hour time limit as reference runtime on both datasets. Only for the *letter* dataset Aid-Fd's performance looks less promising compared to DFD's performance, because DFD can process datasets with up to 20 columns very fast with its random-walk approach [13]. For these datasets, DFD can outperform row-based approaches, even if they are approximate such as Aid-Fd; although the absolute time differences are not devastating. Thus, Aid-Fd is better suited for medium and large sized datasets to replace TANE and FDEP.

5.3 Scaling the number of rows

We evaluate the row scalability of Aid-Fd by measuring runtimes with gradually increasing number of rows of the datasets *ncvoter* and *uniprot*. Both datasets have a fixed number of columns: 19 columns for *ncvoter* and 30 columns for *uniprot*. We use a negative cover growth threshold of 0 without a window (window size = 1) to allow a quick reaction of the algorithm to changes in the negative cover growth. Figure 5 visualizes that the algorithm scales well with the increasing number of rows. This is due to the linear complexity of the negative cover creation phase, which dominates the overall runtime of Aid-Fd.

Algorithm 1 performs at most $\#iterations \times |r| \times |R|$ comparisons. Even though the number of iterations can differ for each data point due to the chosen termination criteria, the experiment shows that it stays stable with increasing

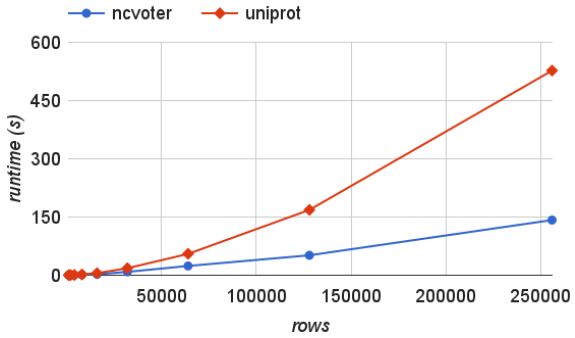

Figure 5: Row scalability on the datasets *ncvoter* with 19 columns and *uniprot* with 30 columns.

number of rows. The second phase, which is the inversion of the negative cover, depends on the size of the negative cover but not on the number of rows. Because the numbers of FDs of the datasets remain in the same magnitudes while increasing the number of rows (between 597 and 936 for *ncvoter* and between 3684 and 8069 for *uniprot*), the second phase has only little influence on row scalability. The quality of the results is hence very good with an average of 99.4% correctness and 99.2% completeness.

5.4 Scaling the number of columns

We now evaluate the column scalability of Aid-Fd by measuring the algorithm's runtime with gradually increasing number of columns of the datasets *plista_1k* and *uniprot*. The datasets now have a fixed number of 1001 rows each. Again, we use a negative cover growth threshold of 0 with-

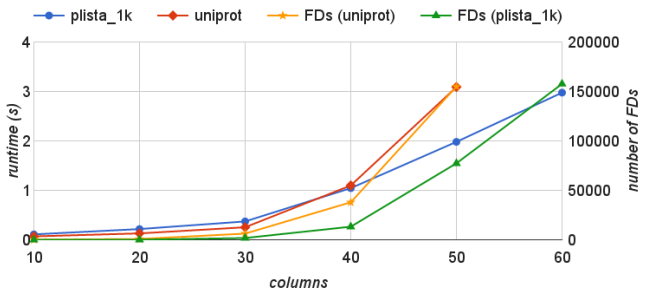

Figure 6: Column scalability on *plista_1k* and *uniprot* each with 1001 rows.

out a window (window size = 1) as a termination criterion. Note that *uniprot* with 60 columns could not be processed due to the limited main memory and high number of FDs in the result. Figure 6 shows that the runtimes grow more than linearly with an increasing number of columns. This is because the output size, i.e., the number of FDs grows similarly fast and the algorithm's runtime is output bound. The quality of the results has an average of 93.4% correctness and 99.7% completeness, including a single outlier for *plista_1k* with 10 columns (correctness = 0.3).

5.5 Memory consumption

Especially column-based algorithms, such as TANE, run into memory problems on relatively small datasets. Papenbrock et al. [13] have shown that TANE even exceeds a memory limit of 100 GB on the *uniprot_223c*, *flight_1k*, and *plista_1k* datasets. In contrast to these lattice-based algorithms, Aid-Fd uses much less memory, similar to the FDEP algorithm. More concretely, Aid-Fd never consumes more than 1.3 GB of main memory on any dataset in our test scenarios, rendering it an eligible choice in low-memory environments, such as personal computers.

Our algorithm's memory consumption is dominated by the number of FDs in the result set: Its memory consumption is highest on the *uniprot* dataset (with more than one million FDs), whereas lowest on the *letter* dataset (61 FDs). Other data structures, such as the indices and clusters tables, have little impact on the memory consumption. This observation explains why Aid-Fd's memory usage scales well with the increasing number of tuples, which has only limited influence on the size of the solution space. On the contrary, an increasing number of columns can increase the solution space exponentially and thus explains why the FD set of *uniprot_223c* does not fit into 100 GB of main memory, rendering it infeasible even for Aid-Fd. This is the reason for using the smaller *uniprot* dataset with 30 instead of 223 columns in the evaluation.

5.6 Evaluation of termination criteria

To evaluate the trade-off between result quality and time savings, we conducted experiments using the two different termination criteria presented in Section 4.3. The termination criterion using a *fixed amount of time* is already being employed in in the comparison of Aid-Fd with exact algorithms (Section 5.2). Figure 4 shows that completeness and correctness converge towards 1.0 with increasing timeout but the different datasets require vastly different timeouts to yield acceptable results. Therefore, this termination criterion should be chosen only if Aid-Fd is used in an interactive scenario where a user is steering the profiling process.

Figure 7 shows the results for the criterion of a *fixed negative cover growth* without a window (window of size 1) for the datasets *plista_1k*, *ncvoter_64k*, and *uniprot*. For the negative cover growth threshold, we use the values 0.1, 0.01, 0.001, and 0.0. Using the threshold 0.1, the average completeness and correctness are 0.68 while using about 4% of the execution time of the fastest exact algorithm on average. Decreasing the threshold significantly increases the correctness and completeness of the result. A threshold of 0.0 yields an average correctness and completeness of 0.996 and 0.995 while using 10% of the runtime of the fastest exact algorithm on average. The relative execution time (compared to the fastest exact approach) for the threshold 0.0 varies from 0.02

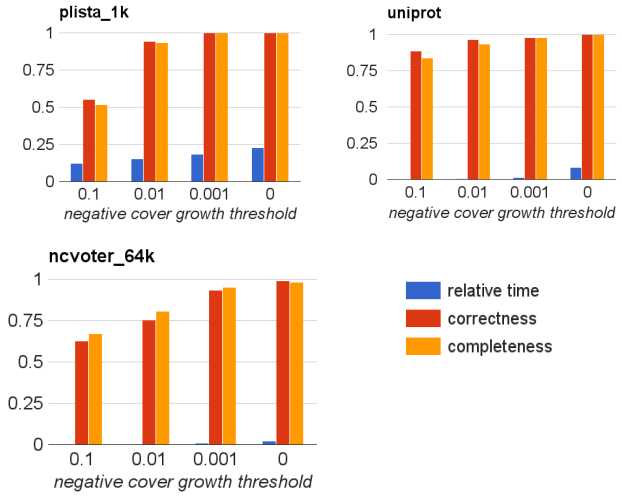

Figure 7: Correctness, completeness and runtime for different datasets using different negative cover growth thresholds (window size = 1). Runtime is given relative to exact runtimes (see Section 5.2).

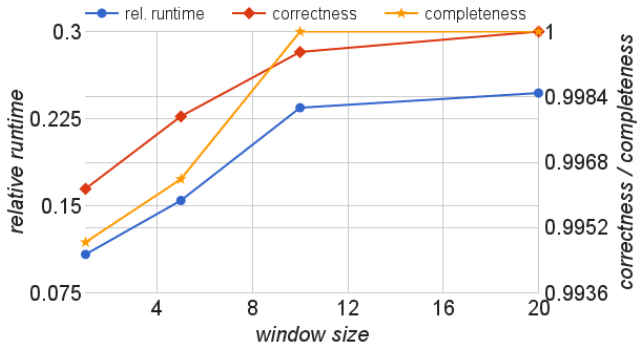

Figure 8: Average relative runtime, correctness, and completeness for different window sizes (negative cover growth threshold = 0).

for the *ncvoter_64k* dataset to 0.4 for the *plista_1k* dataset. We observe the greatest time savings and best quality measures for datasets with both many rows and columns, e.g., *ncvoter_64k* and *uniprot*. Smaller datasets, such as *plista_1k*, can be already processed very fast by exact algorithms and thus our approach cannot achieve high runtime savings.

Additionally, we evaluate the same termination criterion using different window sizes with a negative cover growth threshold of 0.0. Figure 8 shows the average results for relative runtime, correctness, and completeness for the three datasets used above. The result quality (correctness / completeness) increases from about 0.995 / 0.996 (window size = 1) to 1.0 / 1.0 (window size = 20) while the relative runtime rises from 10% to about 25%. The experiments show that a larger window yields a higher quality and even allows to achieve the exact result when the window size is large enough. However, the execution time increases strongly with a growing window size and must be carefully chosen as a tradeoff between quality and runtime.

5.7 Cover inversion

Figure 9 shows the effect of different optimizations described in Section 4.4. For this evaluation, we use datasets that have many columns, because the cover inversion takes much longer than the cover creation on these datasets. On datasets with only very few columns (less than 20), the cover inversion usually takes less than one second.

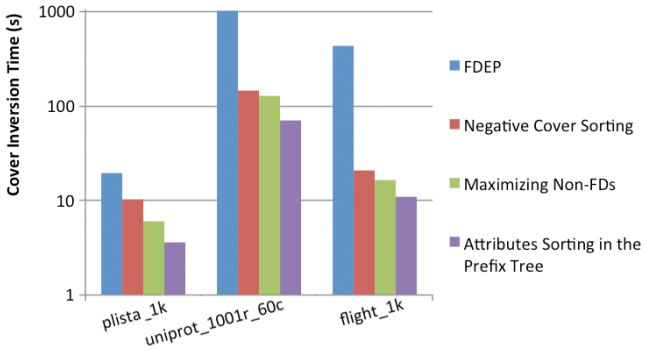

Figure 9: The effect of different optimizations on the cover inversion process (log-scale).

To allow for a fair comparison to the un-optimized cover inversion that is known from the FDEP algorithm [6], we first calculate the complete negative cover for all three datasets and then compare only the time spent on inverting the negative cover. Even if we enable only the first optimization (sorting the negative cover), our cover inversion is on average about 20 times faster than FDEP's. Maximizing the non-FDs, then, reduces the time by another 12–41%. Finally, the sorting of attributes in the tree structure reduces the runtime by additional 33–46%. This leaves us with a total improvement of a factor of 5.5 on *plista_1k*, 14.7 on *uniprot_1001r_60c* (adapted *uniprot* dataset with 1001 rows and 60 columns), and 39.5 on *flight_1k*.

6. CONCLUSIONS

We presented Aid-Fd, an approximate algorithm for the discovery of functional dependencies, which is centered around an iterative, focused sampling, memory-efficient data structures, and a highly efficient negative cover inversion. To compare the quality of Aid-Fd's results, we introduced measures for correctness, completeness, and minimality of FD result sets. In fact, Aid-Fd is only approximate w.r.t. the former two while guaranteeing minimality. In our experiments, we showed that, in comparison to exact algorithms, Aid-Fd is much more efficient on almost all datasets and makes FD discovery feasible on datasets so large that exact approaches cannot handle them. Still, Aid-Fd is highly effective in terms of completeness and correctness measures. For future work, our findings shall be transferred to other data profiling tasks, such as the discovery of order dependencies or inclusion dependencies.

Acknowledgements. This research was partially funded by the German Research Society (grant no. FOR 1306).

7. REFERENCES

[1] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: a survey. *VLDB Journal*, 24(4):557–581, 2015.

[2] 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.

[3] W. W. Armstrong. Dependency structures of data base relationships. In *IFIP Congress*, volume 74, pages 580–583. North-Holland Publishing, 1974.

[4] P. G. Brown and P. J. Hass. BHUNT: Automatic discovery of fuzzy algebraic constraints in relational data. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 668–679, 2003.

[5] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems (TODS)*, 15(2):162–207, 1990.

[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. 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] J. Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. *Theoretical Computer Science*, 149(1):129–149, 1995.

[10] 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, 2010.

[11] 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.

[12] S. Lopes, J.-M. Petit, and L. Lakhal. Functional and approximate dependency mining: database and FCA points of view. *Journal of Experimental & Theoretical Artificial Intelligence*, 14(2-3):93–114, 2002.

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

[14] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 821–833, 2016.

[15] J.-M. Petit, F. Toumani, J.-F. Boulicaut, and J. Kouloumdjian. Towards the reverse engineering of denormalized relational databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 218–227, 1996.