Holistic Data Profiling: Simultaneous Discovery of Various Metadata

Jens Ehrlich[1] Mandy Roick[1] Lukas Schulze[1]
Jakob Zwiener[1] Thorsten Papenbrock[2] Felix Naumann[2]
Hasso-Plattner-Institute (HPI), Potsdam, Germany
[1] firstname.lastname@student.hpi.de [2] firstname.lastname@hpi.de

ABSTRACT

Data profiling is the discipline of examining an unknown dataset for its structure and statistical information. It is a preprocessing step in a wide range of applications, such as data integration, data cleansing, or query optimization. For this reason, many algorithms have been proposed for the discovery of different kinds of metadata. When analyzing a dataset, these profiling algorithms are often applied in sequence, but they do not support one another, for instance, by sharing I/O cost or pruning information.

We present the holistic algorithm MUDS, which jointly discovers the three most important metadata: inclusion dependencies, unique column combinations, and functional dependencies. By sharing I/O cost and data structures across the different discovery tasks, MUDS can clearly increase the efficiency of traditional sequential data profiling. The algorithm also introduces novel inter-task pruning rules that build upon different types of metadata, e.g., unique column combinations to infer functional dependencies. We evaluate MUDS in detail and compare it against the sequential execution of state-of-the-art algorithms. A comprehensive evaluation shows that our holistic algorithm outperforms the baseline by up to factor 48 on datasets with favorable pruning conditions.

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Database Applications; H.3.3 [**Information Systems**]: Information Search and Retrieval

General Terms

Algorithms, Performance, Theory

Keywords

data profiling, inclusion dependency, functional dependency, unique column combination

1. DEPENDENCY DISCOVERY

With the ever growing amount of digitally recorded information, the need to maintain, link, and query these information becomes increasingly hard to fulfill. For many applications, such as data mining, data linkage, query optimization, schema matching, or database reverse engineering, it is crucial to understand the data and, in particular, its structure and dependencies [13]. In biological research, for instance, scientists create large amounts of genome data that grow rapidly every year [2]. Originating from different genome sequencers, the data needs to be analyzed and linked to other datasets. This task requires knowledge of several structural properties of the data.

Usually, the reason why data becomes difficult to access is that metadata about the datasets' structure or their dependencies is missing. Therefore, various profiling algorithms have been proposed for the computationally intensive discovery of metadata, such as inclusion dependencies (INDs) [4, 8], unique column combinations (UCCs) [1, 9, 10, 16], and functional dependencies (FDs) [11, 14]. The algorithms SPIDER [4] for the discovery of INDs, DUCC [10] for UCCs and FUN [14] for FDs are among the most efficient algorithms in their respective problem domain. The idea of all these discovery algorithms is to reduce the tremendous search spaces with so called pruning rules: A pruning rule allows to infer the (in)validity of certain unchecked metadata candidates from already checked ones. Each algorithm, however, computes only one type of metadata. While this might be sufficient for some applications, most applications like data exploration or data integration require different types of metadata at the same time [13]. Therefore, current data profiling processes run several highly complex algorithms in a row. Considering that these algorithms and the metadata they discover have many commonalities, the sequential execution is a waste of time and resources.

Some existing profiling algorithms, such as HCA [1] or FUN [14], already leverage some knowledge about other types of metadata to reduce the discovery time (pruning). However, the combination of different profiling algorithms, i.e., a *holistic algorithm*, can utilize many more interleavings to increase its overall efficiency: First, it can facilitate new pruning rules using all collected information at once. In this way, fewer validity checks on the actual data are required. Second, it can share common costs like those required for I/O operations and iteration cycles. Third, it can combine similar data structures from different profiling tasks into one holistic data structure reducing overall memory consumption and initialization costs.

In this paper, we describe new inter-task pruning capabilities and analyze their impact on the algorithm's runtime. We also develop and evaluate a novel holistic algorithm called Muds, which jointly discovers unary INDs, minimal UCCs, and minimal FDs in one execution while facilitating all three opportunities for performance optimization mentioned above. If a dataset has favorable pruning conditions – which is true for most real-world datasets – Muds improves upon the sequential execution of profiling algorithms by up to a factor of 48. In our evaluation, we investigate dataset characteristics that lead to good and poor runtime behavior.

Contributions. We first analyze INDs, UCCs, and FDs for their commonalities and examine state-of-the-art profiling algorithms (Section 2). We then discuss different approaches for holistic data profiling and possible pruning rules across profiling tasks (Section 3). Next, we describe how different types of FDs can be discovered or pruned if minimal UCCs are already known (Section 4). Based on the discovery and pruning techniques, we present the novel algorithm Muds, which utilizes inter-task pruning rules (Section 5). Muds derives FDs directly from discovered minimal UCCs and facilitates a new depth-first traversal strategy that is based on minimality pruning and the knowledge about non-dependencies (unlike previous level-wise approaches for FDs that solely rely on minimality pruning). Finally, we compare Muds with the sequential execution of state-of-the-art algorithms and with a holistic adaption of the algorithms Spider and Fun (Section 6). Our evaluation shows that Muds usually not only considerably outperforms the baseline algorithms but also outperforms common FD discovery algorithms on datasets with more than 10 columns.

2. PROFILING TASKS

For our holistic approach, we focus on three common and computationally intensive profiling tasks: The discovery of all unary inclusion dependencies, all unique column combinations, and all functional dependencies in a given dataset. This section defines the three tasks and explains one state-of-the-art algorithm for each of them. The chosen algorithms are among the most efficient algorithms for their specific task and exhibit favorable features to combine them into a holistic algorithm. At the end of this section, we compare the nature of the profiling tasks and their search space complexities.

2.1 Inclusion dependencies

An inclusion dependency (IND) $X \subseteq Y$ between attribute sets X and Y describes that the projection of Y contains all values of the projection of X, i.e., all values in X are also contained in Y. Attribute set X is called the *dependent* and Y is called the *referenced*. INDs with only one attribute in the sets X and Y are called *unary INDs*. Because only these unary INDs are of interest for the holistic discovery of other metadata types, we only consider them in our holistic algorithm. Without any loss of generality, we could discover n-ary INDs as well, but these would not contribute to the holistic discovery. We also artificially restrict the IND discovery to a single relation, because the two other metadata types UCCs and FDs are defined on only one relation. The search space for a relation with n attributes, hence, comprises $n \cdot (n-1)$ unary IND candidates.

Spider is the currently most efficient algorithm for the detection of unary INDs. The algorithm developed by Bauckmann et al. [4] consists of two phases: A sorting phase and a comparison phase. In the first phase, Spider sorts the values of each column, eliminates duplicate values, and stores the sorted values in separate lists (Tables 1.1 and 1.2). In the second phase, Spider initially assumes that all attributes are included in one another. Then, it iterates simultaneously over the sorted lists in order to invalidate IND candidates (Tables 1.3 and 1.4). For the invalidation, Spider selects the group of attributes that all contain the currently smallest value. In our example A and C both contain w. By set intersection, the algorithm then excludes INDs from the candidates: The attributes in this group can only be included in one another, because they exclusively contain a value. So, A can still depend on C, but A cannot depend on B, because B does not contain w. The algorithm continues with next smallest values until only valid INDs remain.

1.			2.			3.			4.		
A	B	C	A	B	C	A	B	C	A	B	C
w	z	x	w		w	→w		→w	w		w
w	x	x	x	x	x	x	→x	x	→x	→x	→x
x	z	w	y			y			y		
y	z	z		z	z		z	z		z	z

Table 1: Example execution of Spider

2.2 Unique column combinations

A set X of attributes is a unique column combination (UCC) if the projection of X does not contain duplicates; otherwise, it is a non-unique column combination (non-UCC). UCCs are also called key candidates, because the values of the projection of a UCC uniquely define all records. X is a *minimal* unique column combination (minimal UCC) if it is a UCC and no proper subset of X is a UCC. The number of possible candidates for UCCs in a relation R with n attributes is $2^n - 1$. We can visualize the search space of UCC candidates as a lattice of attribute sets, i.e., a Hasse diagram (Figure 1). Each node represents a set of attributes and each edge a superset/subset relationship.

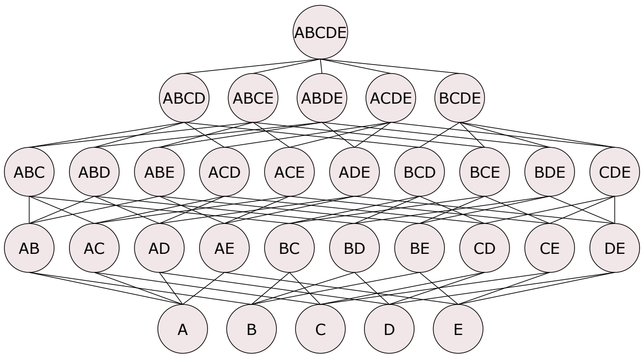

Figure 1: Attribute lattice for five columns

The Ducc algorithm is among the most efficient algorithms for detecting UCCs [10]. It applies a combination of depth-first and random walk strategies to traverse the lattice of UCC candidates. Thereby, Ducc uses the information about both discovered UCCs and non-UCCs for pruning.

The traversal starts from the bottom of the lattice. Every time the algorithm detects a non-UCC, it generates all direct supersets and picks one randomly as its next candidate; if the algorithm detects a UCC, the next candidate is a random direct subset. While traversing, Ducc prunes subsets of non-UCCs and supersets of UCCs from the search space, because subsets of non-UCCs are also non-UCCs, and supersets of UCCs cannot be minimal. It is possible that some column combinations remain unvisited after the random walk. The reason for such "holes" in the lattice is the combination of upwards and downwards pruning. Ducc identifies and fills these holes by comparing the found minimal UCCs with the complement of the found maximal non-UCCs.

During traversal, the algorithm has to check whether a column combination is a UCC or a non-UCC. This check is performed by constructing a position list index (PLI, also called stripped partition) for the column combination. A PLI is a list that contains sets of tuple ids [11]. These tuple ids belong to tuples of the column combination that contain the same value. If a PLI contains no id-set of size two or larger, the column combination contains no duplicate value and is, hence, unique. Because only id-sets of size two or larger are necessary for this test, all id-sets of size one can be removed, i.e., stripped from the PLI. So if the PLI is empty, all values are unique and the column combination is a UCC. To calculate the PLI for an unvisited lattice node AB, Ducc intersects the PLIs of the nodes A and B by pair-wise intersecting their id-sets.

2.3 Functional dependencies

Given a relation R, a functional dependency (FD) $X \rightarrow A$ between a set of attributes X and an attribute A exists if the values of X uniquely determine the values of A [6]. We call X the *left hand side* and A the *right hand side*. An FD is called trivial if $A \in X$. The FD is minimal if no proper subset of X determines A. A set of attributes X can determine multiple other attributes Y. In the following, we use the short notation $X \rightarrow Y$ to denote FDs with one or more right hand side attributes. In the lattice shown in Figure 1, every *edge* represents a potential FD. For example, the edge between ABC and $ABCD$ represents the FD candidate $ABC \rightarrow D$. To count all FD candidates for n attributes, we count the edges in each level k with $k = 1...n$. Starting with attribute sets of size one, the sets are extended by one attribute in each level until the set in the highest level contains all attributes. In each level k we find $\binom{n}{k}$ nodes. Each node in level k can be connected to $n-k$ nodes in the next level without generating duplicate connections. The number of FD candidates in a relation with n attributes is therefore $\sum_{k=1}^{n}\binom{n}{k}\cdot(n-k)$.

Among the many FD discovery algorithms, Fun is one of the fastest algorithms [14]. Fun discovers FDs in a relation R by traversing the attribute lattice level-wise with a bottom-up strategy. While exploring the attribute lattice, FUN generates PLIs for each traversed attribute set. The algorithm then derives the cardinality of attributes and attribute combinations from their PLIs. This information is used to efficiently detect FDs by *partition refinement* as described in Lemma 1 (from [14]). In a relation R and a relation instance r, let $|X|_r$ denote the cardinality of the projection of X over r.

Lemma 1. $\forall X \subseteq R, A \in R : X \rightarrow A \Leftrightarrow$ $|X|_r = |X \cup \{A\}|_r$

Fun classifies column combinations into *free sets* and *non-free sets*. A free set X contains only those attributes that are not functionally dependent on any other attribute in X. A non-free set contains at least one functionally dependent attribute. Definition 1 describes the set of free sets $\mathcal{FS}_r$:

Definition 1. *The set of free sets $\mathcal{FS}_r$ is defined as* $\forall X \subseteq R : X \in \mathcal{FS}_r \Leftrightarrow \nexists X' \subset X : |X'|_r = |X|_r$.

The Fun algorithm has a pruning advantage in comparison to other FD algorithms like Tane [11], because it omits certain PLI intersect operations and retrieves missing cardinality information from a node's child nodes. So instead of performing a PLI intersect, Fun infers the cardinality of a pruned non-free set with a recursive look-up in the non-free set's subsets.

2.4 Comparison of profiling tasks

The discovery of INDs inherently differs from the discovery of UCCs and FDs: For INDs, the attribute values are of interest whereas for FDs and UCCs only the *position* of equal values is relevant. Hence, UCC and FD algorithms use quite different data structures than IND algorithms; these data structures do not allow the reconstruction of values (e.g., PLIs), but improve the algorithms' efficiency. However, several relationships between FDs and UCCs can be used for pruning (see Sec. 3).

To determine the complexity of a holistic algorithm, we need to inspect the search spaces of the different sub-tasks: A relation with n attributes contains $n\cdot(n-1)$ unary IND candidates. IND discovery is, hence, in $O(n^2)$. In the same relation, the number of UCC candidates is $\sum_{k=1}^{n}\binom{n}{k}$, which places the search space of UCCs in $O(2^n)$. The number of FD candidates is $\sum_{k=1}^{n}\binom{n}{k}\cdot(n-k)$ so that FD discovery is in $O(n\cdot 2^n)$. The search space for FDs, therefore, clearly dominates the overall discovery cost. The exponential search spaces of UCCs and FDs in particular dominate the quadratic search space of unary INDs. Our evaluation in Sec. 6.4 shows that the time spent on IND discovery is indeed negligible in comparison to the time spent on UCC and FD discovery. For this reason, INDs can best be calculated as a byproduct in the starting phase of a holistic algorithm.

3. HOLISTIC APPROACH

As we described in Section 2.4, IND discovery differs greatly from the discovery of UCCs and FDs. Therefore, INDs are discovered while the input data is read and the values are still accessible. The discovery uses the Spider algorithm and mainly profits from sharing its I/O costs with the UCC and FD algorithms, i.e., the data is read only once and then used for the discovery of all three types of matadata. Spider additionally profits from the initial PLI-construction that is performed for both the UCC and FD discovery: At construction time, PLIs map values to positions so that Spider can retrieve duplicate-free value lists from this mapping, which are more efficient to sort.

If the input dataset contains two identical rows, i.e., duplicate records, then it cannot contain any UCC and, hence, most inter-task pruning rules would not apply. Therefore, we assume that duplicate records, which are forbidden in most database systems anyway, have been removed in a preprocessing step.

We now discuss three basic approaches for the holistic discovery of INDs, UCCs, and FDs.

3.1 FDs first

By discovering minimal FDs first, we can derive all minimal UCCs from the discovered FDs [15]. The UCC inference follows Lemma 2. Under the assumption that each row in R is distinct, every column combination that functionally determines all other attributes of the relation R is a key in R and, hence, a UCC:

LEMMA 2. *$\forall U \subseteq R : U \rightarrow R \setminus U \Rightarrow U$ is a UCC*

Thus, all UCCs can be inferred from the set of minimal FDs. Without describing an actual algorithm for UCC inference, it is clear that the inference and minimization of UCCs introduces an additional overhead. As several FD discovery algorithms (e.g., FUN and TANE) exist that already find all minimal UCCs while discovering FDs, we do not pursue this FDs-first approach for a holistic algorithm. Instead, we focus on approaches that improve the overall runtime by avoiding this additional overhead and by leverageing pruning.

3.2 FDs and UCCs simultaneously

To discover FDs and UCCs simultaneously, we analyzed several FD discovery algorithms and evaluated their extensibility towards UCC discovery. As already described in Sections 2.3 and 2.4, FUN uses an attribute lattice for pruning that is very similar to DUCC's way of calculating minimal UCCs. Furthermore, FUN traverses all unpruned free sets. The following Lemma 3 shows that all minimal UCCs are "free sets" in FUN's sense. Therefore, FUN must traverse the minimal UCCs for FD discovery, enabling the discovery of minimal UCCs with little impact on the overall runtime. In a relation instance r, $\mathfrak{U}_r$ is the set of all minimal UCCs in r.

LEMMA 3. $\forall U \in \mathfrak{U}_r : U \in \mathcal{FS}_r$

PROOF. Let $X \in \mathfrak{U}_r$ be a minimal UCC and let $X \notin \mathcal{FS}_r$ be a non-free set. According to Definition 1, $X' \subset X$ with $|X'|_r = |X|_r$ exists. X' has the same distinct count as X. Therefore, X' is a UCC. This contradicts the initial assumption that X is a *minimal* UCC and, therefore, no subset of the minimal UCC X can be a UCC. □

In the original version of FUN, minimal UCCs are detected and used for key-pruning, which is a common pruning rule for FD discovery: The supersets of UCCs can be pruned, because they cannot be the left hand side of a minimal FD. This pruning rule is applied in FD discovery algorithms like TANE and FUN. With small adaptions, it is possible to store the minimal UCCs and return them when the algorithm terminates. This does not impair the runtime of FUN, because no further checks are necessary. We implemented this algorithm and call it HOLISTIC FUN.

3.3 UCCs first

If a holistic algorithm first discovers all minimal UCCs, it can leverage this information for the discovery of minimal FDs, due to the key-pruning rule. As we explain in Section 4.1, many left hand sides of minimal FDs are subsets of minimal UCCs. This observation can be used to discover and minimize relevant FDs faster. It can further be used for pruning, because applying this rule reduces the number of traversed column combinations in comparison to level-wise FD discovery algorithms. Several rules can derive the invalidity of certain FDs from the known minimal UCCs. We present these rules in more detail in Section 4 and use them in the implementation of our algorithm MUDS in Section 5.

4. DISCOVERING FDS BASED ON UCCS

In this section, we present pruning and inference rules that allow for a fast FD discovery based on known UCCs. These rules lay the foundation for our MUDS algorithm described in Section 5. We show in Section 6 that an algorithm using these rules is usually faster than current state-of-the-art FD discovery algorithms. We first describe the UCC-based pruning rules for FDs. The following three subsections then match the three sub-algorithms of MUDS' FD discovery: *minimize FDs*, *calculate $R \setminus Z$*, and *shadowed FDs*.

Figure 2 categorizes the attributes of a relation R into the set $Z := \bigcup_{U \in \mathfrak{U}} U$, which is the union of all minimal UCCs, and the set $R \setminus Z$, which contains all columns that are not contained in any minimal UCC. In the following, we describe the two situations, shown in Figure 2, where the non-existence of functional dependencies can be inferred from the set of UCCs.

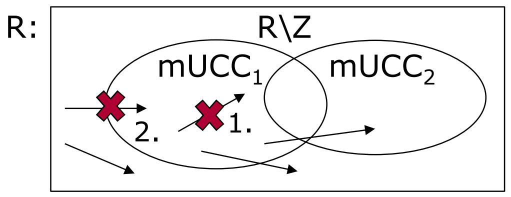

Figure 2: Possible FDs in R with two minimal UCCs

1. FDs fully contained in a minimal UCC: An FD cannot exist if it is fully contained in a minimal UCC (both left and right hand side are subsets of the same minimal UCC). The right hand side of an FD carries only information that could be inferred from the left hand side. Therefore, if a functional dependency existed that is contained in a minimal UCC, the right hand side can be removed from the minimal UCC without changing the uniqueness property. This contradicts the minimality of the UCC.

2. FDs with a lhs in R\Z and rhs in Z: Consider an FD $X \rightarrow A$ with a left hand side $X \subseteq R \setminus Z$ and a right hand side $A \in U$ with $U \in \mathfrak{U}$. A is determined by X; hence, A can be substituted by X in U. This yields at least one unique $U_{subs} = X \cup U \setminus A$. Thus, a minimal unique U_{min} must exist that is subset of U_{subs}. U_{min} must still contain one or more attributes of X (otherwise U could not have been minimal, as A could have been omitted). It follows that a part of X is contained in a minimal UCC, which contradicts the proposition that X is a subset of $R \setminus Z$. Thus, no such FD may exist.

In the following, we present rules and operations that enable the discovery of FDs using the two pruning rules described above. In Section 4.1, we first describe how to discover FDs that have left and right hand sides in overlapping minimal UCCs.

4.1 FDs between minimal UCCs

A UCC functionally determines all other columns of the same relation. Thus, FDs can be inferred from a discovered UCC. Not all of these FDs are minimal. To find the minimal FDs, the substitution pruning rule can be applied:

Substitution rule: For every FD that has an attribute of a minimal UCC as right hand side, we can infer a new UCC, by substituting that attribute in the UCC with the left hand side of the FD. If the inferred UCC does not exist, the corresponding FD cannot hold and must not be checked. This insight is used to validate FD candidates by checking for corresponding UCCs. For instance, given a relation $R = \{A, B, C, D, E, F\}$, the minimal UCCs ABC and DEF, and the FD candidate $BC \rightarrow D$ that we need to check. The UCC $BCEF$ follows by substituting D with BC. Now, a subset of $BCEF$ that is also a proper superset of EF must be a minimal UCC. As this minimal UCC does not exist, we know that the FD $BC \rightarrow D$ cannot exist.

It follows that valid FDs between minimal UCCs must fulfill the following condition: The left hand side and the right hand side of valid FD must be subsets of different and intersecting minimal UCCs. We use this insight in the Muds algorithm, using an operation that we call *connector lookup*. We describe this operation in detail in Section 5.1.

4.2 FDs with right hand sides in R \ Z

Tane and similar algorithms for FD discovery traverse the attribute lattice bottom-up. In the traversal, these algorithms utilize pruning rules based on the minimality of dependencies. If a left hand side is known to yield only non-minimal FDs (e.g., it already contains an FD), this left hand side is pruned from the lattice and the traversal is continued with a reduced candidate set (upwards pruning). We now propose a traversal strategy that operates similarly to the random walk strategy in the UCC discovery algorithm Ducc. For this traversal, pruning of subsets (downwards pruning) is necessary. This pruning is based on a property of FDs: If $X \rightarrow A$ does not hold, A cannot be functionally dependent on any subset of X.

Lemma 4. $\forall X \subseteq R, \forall A \in R, \forall X' \subseteq X :$
$X \nrightarrow A \Rightarrow X' \nrightarrow A$

Lemma 4 can be used to prune downwards. To facilitate this pruning, we traverse a *sub-lattice* for each possible right hand side. A sub-lattice is a lattice created for a specific right hand side attribute, which is omitted from the lattice. The nodes in the sub-lattice contain only the different left hand side candidates. All sub-lattices for an exemplary relation with columns A, B, C, D are shown in Figure 3. In the sub-lattices, the above mentioned pruning rule applies, because of the fixed right hand side.

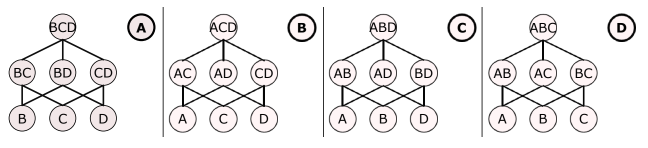

Figure 3: Sub-lattices for the right hand side columns A, B, C, and D

The example in Figure 3 shows that some column combinations are represented in multiple lattices. The column combination CD, for example, is contained in the first and in the second lattice. Such column combinations are only fully pruned, i.e., we do not need to calculate their PLIs, if the column combination is pruned in all sub-lattices. In the best case, entire sub-lattices can be pruned.

Section 5.2 presents an algorithm that leverages the sub-lattice pruning. The algorithm also assures that only columns of $R \setminus Z$ are used as right hand sides of FD candidates.

4.3 Shadowed FDs

Section 4.1 describes how FDs can be deduced from minimal UCCs. Unfortunately, not all minimal FDs can be found in this way. If a minimal UCC U and a column combination $X \subseteq U$ exist and the FD $X \rightarrow A$ holds, a column A may be *shadowed* in U: A can be part of a left hand side with only other columns of U. As our algorithm would not consider this left hand side, we call the left hand side and the resulting FD *shadowed*. To still find all minimal FDs, a post-processing step is needed that explicitly checks all left hand sides containing attributes from different minimal UCCs or even $R \setminus Z$. We explain the algorithm in Section 5.3 but first give an example of how shadowed attributes arise:

Let BCD, CDE, and AD be the only minimal UCCs in a dataset for the relation $R = \{A, B, C, D, E\}$. Suppose that the minimal UCCs directly contain the minimal FDs, i.e., $BCD \rightarrow AE$, $CDE \rightarrow AB$, and $AD \rightarrow BCE$. Now, assume that there is an additional minimal functional dependency $AC \rightarrow B$. The inference rule from Section 4.1 cannot find this FD, because the left hand side AC cannot be deduced from the minimal UCCs BCD, CDE, or AD. Thus, $AC \rightarrow B$ is never checked. The existence of the minimal FDs $BCD \rightarrow A$ indicates that A is shadowed in every subset of BCD. C is shadowed analogously in every subset of AD by the minimal FD $AD \rightarrow C$. Therefore, every attribute that is determined by a subset of BCD or AD may also be determined by a subset of $ABCD$ or ACD containing A and C.

For each minimal FD $Y \rightarrow W$, we must find all shadowed attributes S. The shadowed attributes are added to the left hand side of the functional dependency. This results in the FD $Y \cup S \rightarrow W$, which holds, but is not necessarily minimal. The FD must be minimized and we provide pseudo-code for a minimization algorithm in Section 5.3.

5. THE MUDS ALGORITHM

In this section, we develop a holistic profiling algorithm called Muds, which jointly discovers unary INDs, minimal UCCs, and minimal FDs facilitating pruning across the different profiling tasks. The acronym Muds is a composition of the algorithms key features: "Mu" for <u>M</u>inimizing <u>U</u>CCs, "d" for the <u>D</u>epth-first approach that is used to find FDs with right hand side in $R \setminus Z$, and "s" for <u>s</u>hadowed FDs.

The algorithm uses the following execution strategy: While reading the input dataset, Muds directly applies the Spider algorithm described in Section 2.1 to calculate INDs (one shared I/O operation). Since this algorithm already requires to read and sort all records, Muds also builds the PLIs in this step. Afterwards, Muds runs the Ducc algorithm, described in Section 2.2, using the previously calculated PLIs to identify all minimal UCCs. Finally, it executes a novel FD discovery algorithm that is based on the concepts explained in Section 4. The FD discovery takes advantage of the known minimal UCCs (inter-task pruning) and the already calculated PLIs (shared data structures).

The FD discovering part of Muds consists of three phases: In the first phase, Muds discovers the FDs among overlapping minimal UCCs (Section 5.1). In the second phase, the algorithm finds FDs with right hand sides in the set of columns that are non-minimal UCCs ($R \setminus Z$) (Section 5.2). In the third phase, Muds discovers and minimizes the remaining FDs that have a shadowed left hand side (Section 5.3). Because the FD validations in the Muds algorithm perform many superset look-ups to find minimal UCCs for left hand side column combinations, we introduce a prefix tree of UCCs that ensures fast look-up times (Section 5.4).

5.1 FDs in connected minimal UCCs

Muds is a "Unique Column Combinations First" algorithm. So it first discovers all minimal UCCs to use them for the FD discovery. So when starting the FD discovery, all minimal UCCs are already known. Now, the first part of Muds' FD discovery analyzes only those FDs whose left and right hand side columns are included in minimal UCCs. For an FD $X \to Y$, we call the complete set of functionally determined attributes Y the *closure* of X. Suppose U is a minimal UCC in the relation R; then, the closure of U is R, because U functionally determines all other attributes. It is possible that some of the attributes in the closure of U are non-minimally determined, which means that they are also determined by a subset of U. To minimize the left hand sides of the non-minimal FDs, we check whether the direct subsets of U functionally determine attributes in the closure. In the following, we present an algorithm that minimizes the left hand sides, starting from the minimal UCCs in a top-down manner.

Recursive minimization. In order to minimize the left hand sides of the FDs deduced from minimal UCCs, we start from the UCCs themselves and recursively analyze subsets of the UCCs. Algorithm 1 shows the pseudocode for the recursive minimization. minimizeFDs takes two input parameters: the set $\mathfrak{U}$ of all minimal UCCs discovered in the previous step and the union Z of all attributes that appear in at least one minimal UCC. The algorithm iterates over all minimal UCCs $U \in \mathfrak{U}$, generates FD candidates from them, and then creates minimization tasks for the generated FDs (lines 2-3). We use tasks and a queue of tasks to avoid recursive method calls, which uses heap memory instead of stack memory.

For every task, we know the valid right hand sides from the parent closure, but the minimality of these FDs is not yet known. To determine the minimal right hand sides, minimizeFDs iterates over the left hand side's direct subsets and checks which FDs are also valid in the subsets $U' \subset U$ (line 8). To this end, the algorithm determines the subset's connector $C := U \setminus U'$ and performs the connector look-up (lines 9-10). We describe this connector look-up below. Then, minimizeFDs tests the candidate FDs using partition refinement (line 11). If it finds an FD, minimizeFDs removes the corresponding attribute from the closure of the current left hand side, because the left hand side cannot be minimal (line 12). Finally, we create new tasks for all direct subsets with the valid right hand side (line 15). The valid right hand sides contain all attributes that are still functionally determined by the current subset. After checking the FDs in all subsets, the closure of the current left hand side contains only those right hand sides for which the current left hand side is minimal. The minimizeFDs function then outputs these minimal FDs (line 16).

Connector look-up. Muds generates the right hand side candidates using an operation that we call *connector look-up*. The connector look-up ensures that the left and right hand side are subsets of different minimal UCCs and that the minimal UCCs overlap. We already discussed the princi-

Algorithm 1 Calculate FDs from minimal UCCs

```
Require: minimal UCCs $\mathfrak{U}$, union of all minimal UCCs $Z$
 1: function minimizeFDs($\mathfrak{U}$, $Z$)
 2:     for $U \in \mathfrak{U}$ do
 3:         tasks.add(⟨lhs←$U$, rhs←$Z \setminus U$, mUcc←$U$⟩)
 4:     FDs ← new Map<ColumnComb,ColumnComb>()
 5:     while !tasks.isEmpty() do
 6:         task ← tasks.remove()
 7:         currentRhs ← tasks.rhs
 8:         for lhsSubset ∈ task.lhs.getDirectSubsets() do
 9:             connector ← task.mUcc$\setminus$lhsSubset
10:             potentialRhs ← connectorLookup(connector$\setminus$
                        getImpossibleColumns(lhsSubset))
11:             validRhs ← checkFDs(task.lhs, potentialRhs)
12:             currentRhs ← currentRhs $\setminus$ validRhs
13:             if validRhs.isEmpty then
14:                 continue
15:             tasks.add(⟨lhs←lhsSubset, rhs←validRhs,
                        mUcc←task.mUcc⟩)
16:         FDs[lhs] ← FDs[lhs] ∪ currentRhs
17:     return FDs
```

ples used for this operation in Section 4.1. We now describe the connector look-up in an example shown in Table 2. To perform the connector look-up for a column combination, Muds splits the minimal UCCs into a potential left hand side and a connector. In our example, we split the minimal UCC AFG into the potential left hand side A and the connector FG. Then, we use the connector to perform a look-up on the minimal UCCs: All minimal UCCs that are supersets of the connector, are candidates for the right hand side. Table 2 lists all minimal UCCs of our example in the *mUCCs* column. In the *matched* column, Table 2 depicts the retrieved UCCs. The subset of the matching UCCs that is not the connector is printed in bold font. The result of the look-up is the union of these columns; these columns serve as right hand sides for the next FD candidates.

mUCCs	matched	
$A\underline{FG}$	**A**FG	
$BD\underline{FG}$	**BD**FG	**union: ABCDE**
DEF	-	
$CE\underline{FG}$	**CE**FG	

Table 2: Connector look-up with connector FG

After the connector look-up, the algorithm removes all left hand side columns of the new FD candidate from the right hand side columns, because trivial FDs do not need to be checked. So in the example, we would remove A from the union $ABCDE$. Additionally, the algorithm checks whether the union of the left and right hand side is a subset of a minimal UCC; such dependencies cannot exist, because minimal UCCs cannot contain FDs in themselves.

In the following, we describe the minimization of FDs for the example minimal UCC AFG. Figure 4 depicts the traversed graph. Note that for conciseness, we describe the recursion in only one branch. At first, Muds initializes the closure of the minimal UCC AFG to $R \setminus \{AFG\}$, which in this case is $BCDE$. Then, the algorithm splits AFG into its direct subsets, i.e., left hand sides for new FD candidates, and the connector (the connector is shown in round brackets). At node AF, MUDS performs the connector look-up, which yields the potential right hand sides BD. By checking the FDs $AF \rightarrow B$ and $AF \rightarrow D$ using partition refinement (see Section 2.3), the algorithm finds that both FDs still hold. Therefore, it can remove the FDs' right hand sides from the superset closure in node AFG, because these cannot be minimal at AFG. Then, MUDS computes the direct subsets of AF and again performs the connector look-up. For the column A, the look-up yields BD, whose dependencies are then tested. In the test, the algorithm finds only $A \rightarrow B$ to be valid, which invalidates the minimality of the parent FD $AF \rightarrow B$. After completing an entire level, MUDS outputs the previous level's remaining right hand sides as valid minimal FDs. The algorithm terminates when no potential right hand sides remain or the list of direct subsets is empty. Upon termination, the algorithm has discovered and minimized all FDs among connected minimal UCCs.

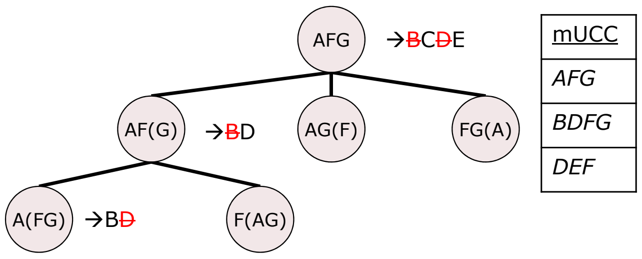

Figure 4: Example for recursive FD minimization

5.2 Graph traversing for R \ Z

This section focusses on those FDs whose right hand side is in $R \setminus Z$. The set $R \setminus Z$ contains all columns that are not element of any minimal UCC. Hence, the previous step did not find these FDs. In contrast to existing FD discovery algorithms, we can restrict possible right hand sides to columns that belong to $R \setminus Z$, because the previous step has already found all minimal FDs with right hand side in Z. Therefore, not all columns in the relation need to be analyzed as potential right hand sides.

As discussed in Section 4.2, MUDS uses one sub-lattice for each potential right hand side to profit from downwards pruning. The algorithm traverses each sub-lattice using a random walk strategy that is based on the traversal strategy of the UCC discovery algorithm DUCC, which we already introduced in Section 2.2: Suppose that we search for FDs with right hand side A. MUDS first constructs the lattice of left hand side candidates for A using the attributes $R \setminus A$. Then the random walk starts at a random seed node on the first level of this lattice. At each node, we check if the column combination in that node determines A.

Suppose we are at the node that contains the column combination X. If X determines A, we continue the traversal by choosing a random subset of X, because the discovered FD $X \rightarrow A$ might not be minimal. All supersets of X also determine A, but are minimal. Therefore, we prune all supersets of X. If the column combination X does not determine A, the next traversal step chooses a random superset of X. All subsets of X can then be pruned, because they cannot determine A (Lemma 4). As noted in Section 2.2, the lattice might contain holes of unvisited nodes, which have to be discovered in the end. For this task, MUDS uses the same approach as DUCC [10].

The mayor difference of this traversal strategy to DUCC's traversal strategy lies in the check that decides which nodes are traversed next: DUCC checks the uniqueness of the node's column combination by inspecting the cardinality of the node's PLI; MUDS checks whether the node's column combination functionally determines the sub-lattice's right hand side A by testing for partition refinement as described in Lemma 1 in Section 2.3.

Before MUDS starts the graph traversal for $R \setminus Z$, it has already determined several minimal FDs (see Section 5.1). These known FDs are used for additional pruning in the lattice: The combination of a left hand side with its right hand side can never be the left hand side of an already known minimal FD (see Section 2.3). The random walk on a sub-lattice terminates when all candidates are checked or pruned. Then, MUDS continues with the remaining sub-lattices until all FDs with right hand sides in $R \setminus Z$ are found.

5.3 Shadowed FD discovery

The two previously described sub-algorithms of MUDS' FD discovery part do not yet find all FDs. This is due to the fact that some FDs are shadowed (see Section 4.3). In the following, we present an algorithm that finds and minimizes these shadowed FDs by first extending and then minimizing the left hand sides of already discovered FDs.

Shadowed FD discovery: Algorithm 2 shows the discovery of the (not necessarily minimal) shadowed FDs. At first, the algorithm calculates the potentially shadowed columns. The columns are shadowed, because they occur in a right hand side of an FD and, thus, are not discovered when the algorithm deduces the FDs from UCCs in a previous step.

To find the missing FDs, the algorithm iterates over all previously discovered FDs in line 2 of Algorithm 2. For each FD, the algorithm iterates over the subsets in line 3 and creates a connector in line 4 that is used for a lookup of potentially shadowed columns in line 5. With these shadowed columns, the algorithm creates a new left hand side from the union of the current left hand side and the shadowed columns in line 6. The FD $newLhs \rightarrow fd.rhs$ obviously holds, because $fd.lhs \rightarrow fd.rhs$ holds and $newLhs = fd.lhs \cup shadowedRhs$. However, the FD $newLhs \rightarrow fd.rhs$ is not minimal and FDs that were previously not discovered might be deduced by minimizing the FD $newLhs \rightarrow fd.rhs$. $newLhs$ can contain many columns and be larger than UCCs. Thus, the amount of columns can be reduced prior to the minimization of the FD. This is a powerful pruning step, which is shown in line 7-8 of Algorithm 2. The

Algorithm 2 Discover shadowed FDs

```
Require: map(lhs,rhs) of minimal $FDs$, minimal UCCs $\mathfrak{U}$
1: procedure DISCOVERSHADOWEDFDS($FDs$, $\mathfrak{U}$)
2:     for $fd \in FDs$ do
3:         for $subset \in fd$.lhs.getAllSubsets() do
4:             $connector \leftarrow fd$.lhs $\setminus$ $subset$
5:             $shadowedRhs \leftarrow FDs[connector]$
6:             $newLhs \leftarrow fd$.lhs $\cup$ $shadowedRhs$
7:             for $reduceLhs \in$ removeUCCs($newLhs$, $\mathfrak{U}$) do
8:                 $shadowedTasks$.add($reduceLhs$, $fd$.rhs)
9:             minimizeFDs($shadowedTasks$)
```

Algorithm 3 Remove UCCs from the left hand side

```
Require: column combination to minimize $lhs$, minimal UCCs $\mathfrak{U}$
 1: function REMOVEUCCS($lhs$, $\mathfrak{U}$)
 2:     $reducedLhs \leftarrow \{\}$
 3:     $tasks$.add($\langle$pos$\leftarrow$0, remCol$\leftarrow\{\}\rangle$)
 4:     $subsetUniques \leftarrow \mathfrak{U}$.getSubsetsOf($lhs$)
 5:     while !$tasks$.isEmpty() do
 6:         $task \leftarrow tasks$.remove()
 7:         if $task$.pos $\geq subsetUniques$.size() then
 8:             $reducedLhs$.add($lhs \setminus task$.remCol)
 9:             continue
10:         $unique \leftarrow subsetUniques[task$.pos$]$
11:         if ($task$.remCol $\cap$ $unique$).isEmpty() then
12:             $tasks$.add($\langle$pos$\leftarrow task$.pos + 1,
                                 remCol$\leftarrow task$.remCol$\rangle$)
13:             continue
14:         for $column \in unique$.columns do
15:             $tasks$.add($\langle$pos$\leftarrow task$.pos + 1,
                          remCol$\leftarrow task$.remCol $\cup$ column$\rangle$)
16:     return $reducedLhs$
```

method *removeUCCs()* compares the FD to the previously discovered UCCs. If *newLhs* contains at least a single UCC, a part of the UCC is removed and multiple *reducedLhs* are returned that do not contain any UCC. This is possible because no left hand side that contains a UCC can yield a minimal FD. This approach allows the algorithm to skip steps in the minimization of the FD, because the initial left hand sides are already smaller. In line 9, the algorithm minimizes the reduced left hand sides. After this step, all FDs in the dataset are discovered. In the following, we describe the methods *removeUCCs()* and *minimizeFDs()* in detail.

Shadowed FD pruning: As previously described, we are not interested in left hand sides that contain UCCs, because they cannot yield minimal FDs. Algorithm 3 shows the minimization of left hand sides by removing parts of UCCs. The resulting left hand sides do not contain any UCC. For a single non-minimized left hand side multiple reduced left hand sides may be generated.

We use tasks and a queue to avoid recursive method calls that is shown in line 3. Then, the algorithm calculates all UCCs that are contained in the left hand side in line 4. For each task, we save the next UCC that must be removed (*pos*) and the columns that should be removed from the initial left hand side to create a UCC-free left hand side(*remCol*). The algorithm iterates over the tasks in line 5-15. Lines 7-9 show a task that already iterated over all contained UCCs. Thus, the calculation of the reduced left hand side is finished and the reduced left hand side is added to the result in line 8. If at least a single UCC must be removed from the current left hand side, the algorithm obtains this UCC in line 10. Lines 11-13 check if the current UCC is completely removed due to the columns in *remCol*. If this is the case, MUDS continues with the next UCC by adding the task with an increased *pos* index in line 12. Lines 13-15 show the actual removal of columns. The algorithm iterates over the columns in the UCC in line 14. For each column, a new task is generated in line 15, where the current column is removed. When all tasks are processed, the algorithm returns the reduced left hand sides in line 16.

Algorithm 4 Discover and minimize potential FDs

```
Require: shadowed Tasks $tasks$, map(lhs,rhs) of minimal functional dependencies $FDs$
 1: procedure MINIMIZEFDS($tasks$, $FDs$)
 2:     while !$tasks$.isEmpty() do
 3:         $task \leftarrow tasks$.remove()
 4:         $currentRhs = task$.rhs
 5:         for $subset \in task$.lhs.getDirectSubsets() do
 6:             $validFD \leftarrow$ checkFDs($subset$, $task$.rhs)
 7:             if $validFDs$.isEmpty() then
 8:                 continue
 9:             $currentRhs \leftarrow currentRhs \setminus validFDs$
10:             $tasks$.add($\langle$lhs$\leftarrow subset$, rhs$\leftarrow validFDs\rangle$)
11:         $FDs[task$.lhs$] \leftarrow FDs[task$.lhs$] \cup currentRhs$
```

Shadowed FD minimizing: MUDS calls Algorithm 4 in Algorithm 2 to minimize FDs. The algorithm also uses a task queue to avoid recursive method calls. For every shadowed FD task containing a potential left and right hand side, MUDS generates all direct left hand side subsets (line 5). These subsets might already determine the right hand side. Afterwards, the algorithm checks, which FDs actually hold on the current subset's left hand side and then removes these dependencies from the superset's right hand side, because they are not minimal (lines 6-9). Then, MUDS creates new tasks for all direct subsets (lines 10). The set of minimal FDs is updated with the known superset's minimal FDs (line 11). When all tasks have been processed, the algorithm terminates. At this stage, all shadowed FDs have been discovered and minimized.

5.4 Subset pruning tree

As described in Section 5.3, MUDS also finds shadowed FDs. For each left hand side X, it needs to look-up all minimal UCCs that are subsets of X. The naïve implementation of this operation iterates over the list of minimal UCCs and performs a subset check for each minimal UCC. With an increasing number of attributes, the number of minimal UCCs increases as well, which makes this operation increasingly expensive. Therefore, we organize all minimal UCCs in a prefix tree that guarantees an efficient look-up of UCC subsets.

The concept of prefix trees was first presented by De La Briandais [7]. For the construction of our prefix tree, we assume that the columns in all column combinations are sorted (e.g. by their position in the dataset). Figure 5 depicts an exemplary prefix tree for UCCs. Level 1, which is the root node of the prefix tree, stores all columns that occur as the first column in any column combination. For each entry in level 1, a node in level 2 may exist. These nodes store the second columns of column combinations that share the same first column. This pattern continues for the other levels. The last column of a column combination can be identified as an entry that has no child node.

The MUDS algorithm uses the prefix tree to efficiently find subsets of minimal UCCs: Given a column combination X, we need to find all subsets of X that are contained in the prefix tree. To find them, we iterate over the sorted columns of X. For each column in X, we check whether the column is contained in level 1 of the prefix tree. If the column is not contained, we discard that column and continue with the

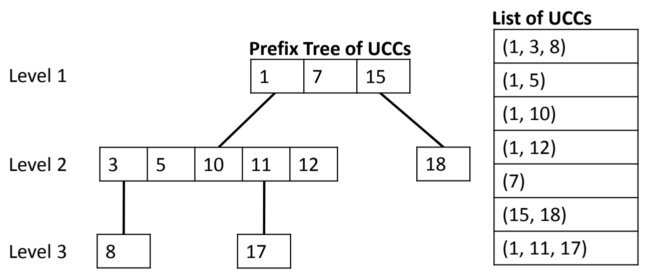

Figure 5: Prefix tree for seven column combinations

next column of X. If it is contained, we visit the associated node in level 2. There, we perform the same check with the remaining columns of X. Then, we go to level 3 and so on. Two conditions can stop the search: First, we find a matching node entry that has no associated node in the next level. This means that we found a subset of X. Second, there are no remaining columns in X, because we discarded all of them, but we have not reached the end of a path in the prefix three. Then, the current path does not lead to a valid subset of X. In both cases, we need to trace back and start over with the remaining columns of X.

6. EVALUATION

We compare MUDS (Sec. 5), HOLISTIC FUN (Sec. 3.2), and the composition of baseline algorithms (Sec. 2) using the Metanome data profiling framework[1]. The framework provides a standardized execution environment for pre-packaged profiling algorithms. In this way, common tasks like file I/O, user interaction, and result handling are decoupled from the algorithms allowing for fair comparisons. Metanome and all algorithms use OpenJDK 1.7 64-Bit. We show that the algorithms have different characteristics and sweet spots, but that MUDS is preferable in general.

Hardware. We execute our experiments on a Dell PowerEdge R620 with CentOS 6.4. The machine has 128 GB DDR3 RAM, of which 120 GB are assigned to the Java virtual machine, and two Intel Xeon E5-2650 (2.00 GHz, Octa-Core) CPUs. Because the implementations of our algorithms are all single threaded, only one of the cores is actually used.

Datasets. For the row scalability experiment in Section 6.1, we use the Universal Protein Resource[2] (*uniprot*) dataset. It is a public dataset about protein sequences and their functions and contains 539,165 rows and 223 columns. For the column scalability experiment in Section 6.2, we use the *ionosphere* [3] dataset. This dataset contains radar data of the ionosphere and features 351 rows and 34 columns (and many dependencies). Our third dataset, the North Carolina Voter Registration Statistics[3] (*ncvoter*) dataset, contains non-confidential data about 7,503,575 voters from North Carolina and features 94 columns. We use this dataset for the evaluation in Section 6.4. For our experiments described in Section 6.3, we use additional real world datasets from the UCI machine learning repository [3]. The section also compares the performance of MUDS with the performance of TANE [11], the most popular FD discovery algorithm, to show that MUDS can also compete with non-holistic FD algorithms. Finally, we discuss dataset properties that MUDS' FD discovery is optimized for in Section 6.5.

[1]`www.metanome.de`
[2]`www.uniprot.org` Accessed: 2015-03-03
[3]`ftp://alt.ncsbe.gov/data/` Accessed: 2015-03-03

6.1 Scalability on rows

In the following experiment, we evaluate the scalability of our algorithms MUDS and HOLISTIC FUN with respect to the number of rows in the input dataset and compare them to the baseline algorithm, which is the sequential execution of SPIDER, DUCC, and FUN. Because of its length, the *uniprot* dataset is best suited for row-scalability experiments. Figure 6 depicts the execution times of the different algorithms on *uniprot* with ten columns while varying the dataset's number of rows.

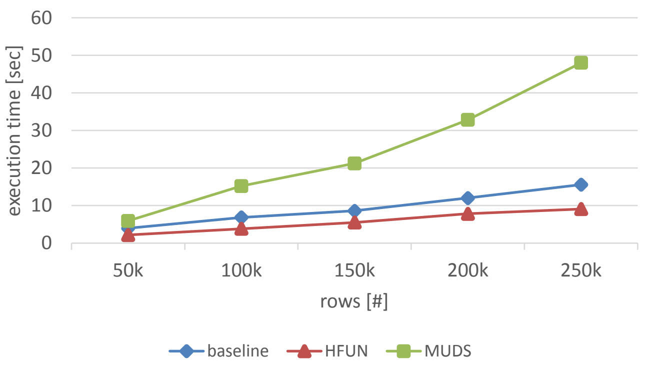

Figure 6: Scalability with regard to the number of rows on the *uniprot* dataset

The measurements show that all three algorithms scale almost linearly with the number of rows. On this particular dataset, HOLISTIC FUN is the fastest algorithm; it is about 1/3 faster than the baseline, because HOLISTIC FUN shares the cost for I/O among the three profiling tasks and discovers UCCs directly with the FDs. MUDS, on the contrary, is the slowest algorithm on *uniprot*, because the discovery of shadowed FDs is particularly expensive on this dataset; if many shadowed FDs are to be discovered, the costs for this step also scale linearly with the number of rows.

6.2 Scalability on columns

The following experiment evaluates the algorithms with regard to the number of columns: Using the *ionosphere* dataset, we successively include more and more columns from the original dataset in each run. The *ionosphere* dataset is particularly interesting for column-scalability experiments, because it contains many and large FDs – a challenge for any FD discovery algorithm and a test of its pruning capabilities. The results are shown in Figure 7, which also presents the number of discovered FDs in the dataset.

The experiment shows that the execution times of all three algorithms scale exponentially with the number of columns, which is due to the fact that the search space for UCCs and FDs also grows exponentially. However, MUDS scales clearly better with the number of columns on the *ionosphere* dataset than both HOLISTIC FUN and the baseline, because due to the UCC-first discovery approach MUDS searches a much smaller search space and the number of shadowed FDs is manageable on *ionosphere*. The baseline and HOLISTIC

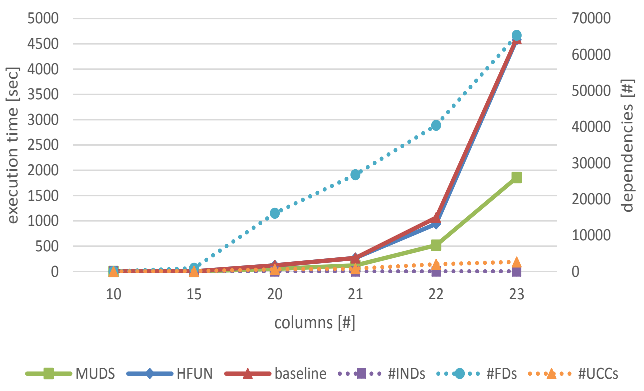

Figure 7: Scalability over the number of columns on the *ionosphere* dataset.

FUN both spend 99% of their runtime on FD discovery for 23 columns, which gives MUDS' improved FD discovery a significant advantage. For the same reason, HOLISTIC FUN performs only slightly better than the baseline; it optimizes those algorithmic parts that make up only 1% of the overall runtime.

6.3 Performance on various Datasets

The scalability experiments evaluated the performance of HOLISTIC FUN and MUDS on the *uniprot* and the *ionosphere* dataset. To verify the observations that we made on these two datasets, we now evaluate the algorithms on various real-world datasets from the UCI machine learning repository [3]. These UCI datasets have been used in most related work benchmarks and thus allow for comparability of results. We also add the TANE algorithm to this comparison in order to investigate how MUDS performs in comparison to state-of-the-art non-holistic FD discovery. Table 3 lists the execution times of the four profiling algorithms.

As in the scalability experiments, HOLISTIC FUN always performs slightly better than the baseline algorithm showing that a holistic approach should always be preferred over the sequential execution of individual profiling algorithms. The experiment also shows that it strongly depends on the properties of the given input dataset whether MUDS or HOLISTIC FUN is the overall best algorithm: For small numbers of columns and higher numbers of rows, HOLISTIC FUN appears to be the faster algorithm and for higher numbers of columns and small numbers of rows, MUDS performs best. The number of columns, thereby, determines the algorithms' performance much stronger than the number of rows as the *adult* and the *letter* dataset show, on which MUDS is up to factor 48 faster than HOLISTIC FUN.

When analyzing the algorithms' performance on the different datasets in detail, we find that some minimal FDs in datasets where MUDS is clearly faster than HOLISTIC FUN have very large left hand sides. The HOLISTIC FUN algorithm, then, must traverse many nodes in the lattice, which is expensive. MUDS, in contrast, utilizes the minimal UCCs for pruning and the depth first search strategy in order to traverse a much smaller part of the lattice. This leads to a clear performance advantage. So in general, the performance of HOLISTIC FUN and MUDS depends on the position of the FDs in the lattice. If the minimal FDs have small left hand sides, HOLISTIC FUN is the better algorithm, because the overhead of finding shadowed FDs in MUDS is unproportionately large. But if there are FDs with large left hand sides in the dataset, MUDS is more efficient. Because the average size of minimal FDs correlates with number of columns, we can choose MUDS or HOLISTIC FUN based on the number of columns. Section 6.5 discusses the difference between HOLISTIC FUN and MUDS in more detail.

Dataset	Col	Row	FDs	basel.	HFUN	MUDS	TANE
iris	5	150	4	**.1s**	**.1s**	**.1s**	.6s
balance	5	625	1	.3s	**.1s**	**.1s**	.9s
chess	7	28k	1	2.0s	**.9s**	1.5s	2.0s
abalone	9	4k	137	1.3s	**.6s**	1.1s	1.0s
nursery	9	12k	1	2.3s	**1.9s**	3.1s	3.1s
b-cancer	11	699	46	.8s	.6s	**.5s**	1.4s
bridges	13	108	142	.8s	.7s	**.6s**	1.3s
echocard	13	132	538	1.0s	**.6s**	1.6s	.8s
adult	14	48k	78	126s	118s	**9.9s**	81.2s
letter	17	20k	61	706s	636s	**13.2s**	326.0s
hepatitis	20	155	8k	462s	450s	88.1s	**10.9s**

Table 3: Runtime comparison on 11 real world datasets

When comparing the runtimes of MUDS with the runtimes of TANE, we observe up to 8 times slower runtimes on the *hepatitis* dataset, where many shadowed FDs exist. However, we also see that the holistic algorithm can outperform the non-holistic algorithm by orders of magnitude: It is 8 times faster on the *adult* dataset and 24 times faster on the *letter* dataset. This is possible if MUDS finds favorable pruning conditions, i.e., FDs with large left hand sides. The algorithm can, then, prune much more efficiently than TANE, which makes MUDS also the better FD algorithm here.

6.4 Analysis of MUDS' phases

Our previous experiments identified MUDS' last phase, namely the discovery of shadowed FDs, to be its most computationally expensive phase. We now analyze the different phases in more detail. Figure 8 depicts the execution time for each phase of the algorithm. We ran this experiment on 20 columns and 10,000 rows of the *ncvoter* dataset.

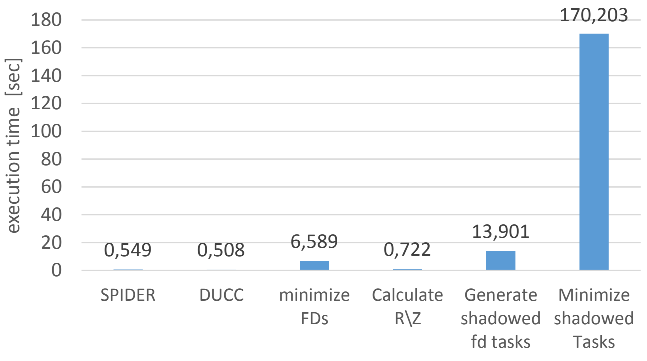

Figure 8: Runtime of Muds' different phases on the *ncvoter* dataset with 10,000 rows and 20 columns

As the measurement show, the computational effort spent on SPIDER and DUCC is almost negligible. The last two phases that detect shadowed FDs, however, are the performance bottleneck of MUDS in this experiment. We observe a 22 times higher execution time than in the previous phases. The *generate shadowed tasks* phase iterates over the already found FDs and creates tasks for possibly shadowed FDs. Each task immediately checks if the FDs holds. These FD checks consume 78% of the time in this phase. Because the found FDs are not necessarily minimal, MUDS then minimizes the valid FDs in the *minimize shadowed tasks* phase. The FD checks in this phase take 90% of the phase's time. If the number of discovered shadowed FDs is high, the minimization is very expensive, because minimizing FDs corresponds to a top-down FD discovery that scales exponentially with the number of columns. In both phases, the primary time-consuming operation is the PLI intersect, which is necessary to check whether an FD holds. However, the execution time distribution between the phases of the MUDS algorithm is dataset specific and especially disadvantageous in this experiment. Nevertheless, we observed that the shadowed FD finding part is the most expensive phase of MUDS on all datasets.

6.5 Favorable dataset properties for MUDS

From our previous experiments, we learned that MUDS usually performs best on datasets with ten or more columns. But the number of columns is only one possible indicator for the performance of MUDS over HOLISTIC FUN. In both algorithms, the discovery of FDs is the most expensive part. So to understand the difference between these two approaches better, we have to compare the two FD search strategies in more depth. Depending on the size of these search spaces on the given dataset, one or the other algorithm is superior.

HOLISTIC FUN searches for FDs by starting at the bottom of the lattice and, then, traverses the nodes level-wise until all minimal FDs are found. Thereby, it classifies visited elements into *free sets* and *non-free sets* to prune non-minimal FD candidates. The size of HOLISTIC FUN's search space is, hence, defined by the height of the lattice levels that hold the minimal FDs; the higher the algorithm has to search, the longer it takes

MUDS first computes all minimal UCCs and, then, searches for FDs in two different search spaces: The first search starts from the minimal UCCs and traverses the lattice in a top-down strategy until all minimal FDs are found (see Section 5.1). All nodes that MUDS traverses in this way are subsets of minimal UCCs and *non-free sets* in the classification of FUN. Because this search space approaches the minimal FDs from the top down, it differs greatly from the search space of HOLISTIC FUN. The second search space of MUDS uses only the columns in $R \setminus A$ for the construction of sub-lattices that are then traversed bottom-up, depth-first (see Section 5.2). This search space overlaps with HOLISTIC FUN's search space, but it is much smaller depending on the size of $R \setminus A$. Furthermore, MUDS' depth-first search reaches minimal FDs on high lattice levels much faster than HOLISTIC FUN's breath-first search.

So MUDS also performs best, if the minimal FDs and minimal UCCs are located on low lattice levels. In comparison to HOLISTIC FUN and any bottom-up, breath-first FD discovery algorithm MUDS performs best when:

1. The minimal UCCs lie close to the minimal FDs in the lattice.
2. The minimal UCCs lie on high lattice levels so that the minimal FDs lie on high lattice levels as well.
3. Many columns participate in at least one minimal UCC, i.e., the column set $R \setminus A$ is small.

Criterion 1 is intuitively clear, because the size of MUDS' first search space shrinks with the distance of minimal UCCs and FDs. Criterion 2 seems to be a disadvantage for MUDS, because it increases the size of the second search space; but MUDS' second search space is *(a)* much smaller than HOLISTIC FUN's search space and *(b)* the depth-first search outperforms the breath-first search if this search depth is large. Criterion 3 is a big advantage for MUDS, because its bottom-down search is much faster than the bottom-up search (see Section 6.4).

So now it becomes clear, why MUDS scales better with the number of columns than HOLISTIC FUN: The average distance between minimal UCCs and minimal FDs growths only slightly with the number of columns so that Criterion 1 always holds; the average height and number of both minimal UCCs and minimal FDs growths constantly so that Criterion 2 becomes increasingly advantageous for MUDS; the chances for columns being part of some minimal UCC with some other columns also increases with the number of columns so that the size of column set $R \setminus A$ increases only slightly making Criterion 3 hold.

The number of columns is, however, only an indirect indicator for the performance of MUDS. If the Criteria 1 to 3 do not hold in a particular dataset or if the dataset comprises many rows (see Section 6.1), a lot more columns would be needed to make MUDS faster than HOLISTIC FUN. So we could, instead, use the number and size of minimal UCCs to choose the FD discovery strategy: Because MUDS calculates the minimal UCCs before it starts the FD discovery, one could choose MUDS' FD discovery part if many, large UCCs have been found or the FUN algorithm if few, small UCCs are found. In practice, however, making the decision based on the number of columns is easier and similarly precise, because the properties "many" and "large" depend on the actual number of rows and columns so that they are non-trivial to define.

7. RELATED WORK

Many data profiling applications, such as data integration and data exploration, need to calculate various metadata at once, but almost all existing algorithms focus on only one task. We already introduced the most important related algorithms for holistic data profiling in Section 2, so we now give a more comprehensive overview on existing IND, UCC, and FD discovery algorithms:

De Marchi et al. [8] presented one of the first algorithms for IND detection. The algorithm constructs an inverted index upon the values of all attributes to check them for inclusions. This technique has been outperformed by the SPIDER algorithm [4] that discards attributes early on. As SPIDER is the currently most efficient algorithm for IND detection, we integrated it in our holistic approaches.

The algorithms for UCC discovery can be separated in two groups: row-based and column-based techniques. Giannella et al. [9] presented a column-based algorithm that generates relevant column combinations to check their uniqueness. An improved version of this algorithm, HCA [1], uses an optimized candidate generation and additional statistical pruning to find UCCs. In both algorithms the check for uniqueness is costly. GORDIAN [16], in contrast, is a row-based UCC discovery algorithm that organizes the data in a prefix tree to check for UCCs. It traverses the tree to determine maximal non-UCCs, which are then used to calculate the minimal UCCs. This is also costly if the number of maximal non-UCCs is large. The currently most efficient UCC algorithm Ducc [10] implements a combination of row-based and column-based techniques: It builds upon efficient data structures and uses UCCs and non-UCCs simultaneously for pruning. Due to the efficiency of Ducc's random walk strategy, we used it in our holistic algorithms.

While Tane [11], proposed by Huhtala et al., is the most popular FD algorithm, it is often outperformed by Fun [14]. Tane and Fun both use partition refinement to identify FDs and apriori-gen to traverse the search space. As Fun additionally incorporates a cardinality inference method that reduces the necessary partition intersect operations, it needs to traverse an overall smaller number of candidates. The CORDS algorithm by Ilyas et al. [12] is capable of identifying various correlations and *soft* FDs. As the algorithm's identification process builds upon sampling techniques, it only approximates the real result.

Although current profiling algorithms focus on one specific profiling task, there are some algorithms that already leverage the knowledge about one type of metadata to draw conclusions about another. The first work in this context was contributed by Beeri and Bernstein [5] who used the defined FDs in a relational database to find additional keys. The algorithms Fun and HCA both use dependencies between FDs and UCCs for pruning. However, they use discovered FDs to derive UCCs, while our algorithm Muds uses discovered UCCs to derive FDs.

8. CONCLUSION

We investigated the problem of simultaneously discovering three types of metadata: INDs, UCCs, and FDs. By interleaving their calculation, we demonstrated how to share common costs for I/O operations and the traversal of data structures in order to reduce the overall profiling time. Furthermore, we introduced new inter-task pruning and inference rules, in particular rules that derive possible FDs from discovered UCCs. For the analysis of our holistic techniques, we proposed the two algorithms Holistic Fun and Muds, which jointly discover unary INDs, minimal UCCs, and minimal FDs. Holistic Fun always outperforms the sequential execution of the Spider, Ducc, and Fun, the fastest of algorithms for the respective tasks, by avoiding duplicate work. The Muds algorithm, on the other hand, also facilitates the additional inter-task pruning and inference rules giving it a significant performance advantage on different real world datasets: Our evaluation shows that Muds is up to 48 times faster than Holistic Fun and the baseline if favorable (and common) pruning conditions (many FDs and UCCs with long left hand sides) are given. On such datasets, Muds performs even faster than the fastest pure FD algorithm. The number of columns and the size of discovered UCCs can both be used to decide whether Muds or Holistic Fun should be used.

9. REFERENCES

[1] Z. Abedjan and F. Naumann. Advancing the discovery of unique column combinations. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1565–1570, 2011.

[2] A. Andreeva, D. Howorth, J.-M. Chandonia, S. E. Brenner, T. J. P. Hubbard, C. Chothia, and A. G. Murzin. Data growth and its impact on the SCOP database: new developments. *Nucleic Acids Research*, 36(1):419–425, 2008.

[3] K. Bache and M. Lichman. UCI machine learning repository. `http://archive.ics.uci.edu/ml`, 2013. Accessed: 2015-03-03.

[4] J. Bauckmann, U. Leser, and F. Naumann. Efficiently computing inclusion dependencies for schema discovery. In *ICDE Workshops*, page 2, 2006.

[5] C. Beeri and P. A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems (TODS)*, 4(1):30–59, 1979.

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

[7] R. De La Briandais. File searching using variable length keys. In *Proceedings of the ACM Western Joint Computer Conference*, pages 295–298, 1959.

[8] F. De Marchi, S. Lopes, and J.-M. Petit. Efficient algorithms for mining inclusion dependencies. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 464–476, 2002.

[9] C. Giannella and C. Wyss. Finding minimal keys in a relation instance. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.7086&rep=rep1&type=pdf`, 1999. Accessed: 2015-03-03.

[10] A. Heise, J.-A. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 301–312, 2013.

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

[12] I. F. Ilyas, V. Markl, P. 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.

[13] F. Naumann. Data profiling revisited. *SIGMOD Record*, 32(4):40–49, 2013.

[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] H. Saiedian and T. Spencer. An efficient algorithm to compute the candidate keys of a relational database schema. *The Computer Journal*, 39(2):124–132, 1996.

[16] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. Gordian: efficient and scalable discovery of composite keys. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 691–702, 2006.