Efficient Discovery of Approximate Dependencies

Sebastian Kruse
Hasso Plattner Institute
Prof.-Dr.-Helmert-Str. 2–3
Potsdam, Germany
sebastian.kruse@hpi.de

Felix Naumann
Hasso Plattner Institute
Prof.-Dr.-Helmert-Str. 2–3
Potsdam, Germany
felix.naumann@hpi.de

ABSTRACT

Functional dependencies (FDs) and unique column combinations (UCCs) form a valuable ingredient for many data management tasks, such as data cleaning, schema recovery, and query optimization. Because these dependencies are unknown in most scenarios, their automatic discovery has been well researched. However, existing methods mostly discover only exact dependencies, i.e., those without violations. Real-world dependencies, in contrast, are frequently approximate due to data exceptions, ambiguities, or data errors. This relaxation to approximate dependencies renders their discovery an even harder task than the already challenging exact dependency discovery. To this end, we propose the novel and highly efficient algorithm Pyro to discover both approximate FDs and approximate UCCs. Pyro combines a separate-and-conquer search strategy with sampling-based guidance that quickly detects dependency candidates and verifies them. In our broad experimental evaluation, Pyro outperforms existing discovery algorithms by a factor of up to 33, scales to larger datasets, and at the same time requires the least main memory.

PVLDB Reference Format:
Sebastian Kruse, Felix Naumann. Efficient Discovery of Approximate Dependencies. *PVLDB*, 11 (7): 759-772, 2018.
DOI: https://doi.org/10.14778/3192965.3192968

1. THE EXCEPTION PROVES THE RULE

Database dependencies express relevant characteristics of datasets, thereby enabling various data management tasks. Among the most important dependencies for relational databases are *functional dependencies (FDs)* and *unique column combinations (UCCs)*. In few words, an FD states that some attributes in a relational instance functionally determine the value of a further attribute. A UCC, in contrast, declares that some columns uniquely identify every tuple in a relational instance. More formally, for a relation r with the schema R with attribute sets $X, Y \subseteq R$, we say that $X \rightarrow Y$ is an FD with *left-hand side (LHS)* X and *right-hand side (RHS)* Y if we have $t_1[X]{=}t_2[X] \Rightarrow t_1[Y]{=}t_2[Y]$ for all pairs of distinct tuples $t_1, t_2 \in r$. Likewise, X is a UCC if $t_1[X] \neq t_2[X]$ for all such tuple pairs. For instance, in a table with address data, the country and ZIP code might determine the city; and every address might be uniquely identified by its ZIP code, street, and house number.

Proceedings of the VLDB Endowment, Vol. 11, No. 7
Copyright 2018 VLDB Endowment 2150-8097/18/03.
DOI: https://doi.org/10.14778/3192965.3192968

FDs and UCCs have numerous applications from schema discovery [26] over data integration [34] to schema design [23], normalization [39], and query relaxation [35]. But because the FDs and the UCCs are unknown for most datasets, various algorithms have been devised over the last decades to automatically discover them [15,16,39,41]. Most of these algorithms discover only *exact* dependencies, which are completely satisfied by the data – without even a single violation. Real-world dependencies are all too often not exact, though. Table 1 exemplifies common reasons for this:
(1) Data errors: One might need to determine a primary key for the data in Table 1, and $\{First\_name, Last\_name\}$ seems to form a reasonable candidate. However, it is not a UCC: tuple t_4 is a duplicate of t_1. In fact, the table does not contain a single exact UCC.
(2) Exceptions: Most English first names determine a person's gender. There are exceptions, though. While Alex in tuple t_1 is male, Alex in t_5 is female. In consequence, the FD *First_name*$\rightarrow$ *Gender* is violated.
(3) Ambiguities: In contrast to first names and genders, a ZIP code is defined to uniquely determine its city. Still, we find that t_3 violates *ZIP*$\rightarrow$ *Town*, because it specifies a district rather than the city.

Table 1: Example table with person data.

	First name	Last name	Gender	ZIP	Town
t_1	Alex	Smith	m	55302	Brighton
t_2	John	Kramer	m	55301	Brighton
t_3	Lindsay	Miller	f	55301	Rapid Falls
t_4	Alex	Smith	m	55302	Brighton
t_5	Alex	Miller	f	55301	Brighton

These few examples illustrate why relevant data dependencies in real-world datasets are often not exact, so that most existing discovery algorithms fail to find them. To cope with this problem, the definition of exact dependencies can be relaxed to allow for a certain degree of violation [21]. We refer to such relaxed dependencies as *approximate dependencies.* Approximate dependencies can not only substitute their exact counterparts in many of the above mentioned use cases, but they also reveal potential data inconsistencies and thus serve as input to constraint-repairing systems [7,13,19,24,42]. Furthermore, approximate dependencies can help to improve poor cardinality estimates of query optimizers by revealing correlating column sets [17,27]; and they can support feature selection for machine learning algorithms (especially for those assuming mutual independence of features) by exposing dependent feature sets [10].

Unfortunately, their discovery is more challenging than is the case for their exact counterparts. The main challenge of both FD and UCC discovery is the huge search space that grows exponentially with the number of attributes in a dataset. To cope, exact discovery algorithms employ aggressive pruning: As soon as they find a single counter-example for a dependency candidate, they can immediately prune this candidate [39,41]. This obviously does not work when violations are permitted. Hence, a new strategy is called for to efficiently determine approximate dependencies.

We present Pyro, a novel algorithm to discover *approximate FDs (AFDs)* and *approximate UCCs (AUCCs)* with a unified approach based on two principal ideas. The first is to use focused sampling to quickly hypothesize AFD and AUCC candidates. The second idea is to traverse the search space in such a way, that we can validate the dependency candidates with as little effort as possible.

The remainder of this paper is structured as follows. In Section 2, we survey related work and highlight Pyro's novel contributions. Then, we precisely define the problem of finding AFDs and AUCCs in Section 3 and outline Pyro's approach to this problem in Section 4. Having conveyed the basic principles of our algorithms, we then proceed to describe its components in more detail. In particular, we describe how Pyro efficiently estimates and calculates the error of AFD and AUCC candidates in Section 5 and elaborate on its focused search space traversal in Section 6. Then, we exhaustively evaluate Pyro and compare it to three state-of-the-art discovery algorithms in Section 7. We find that Pyro is in most scenarios the most efficient among the algorithms and often outperforms its competitors by orders of magnitude. Finally, we conclude in Section 8.

2. RELATED WORK

Dependency discovery has been studied extensively in the field of data profiling [1]. The efficient discovery of exact FDs has gained particular interest [37]. Further, many extensions and relaxations of FDs have been proposed [9], e.g., using similarity functions, aggregations, or multiple data sources. Pyro focuses on approximate dependencies that may be violated by a certain portion of tuples or tuple pairs. Note that this is different from *dependency approximation algorithms* [8,22], which trade correctness guarantees of the discovered dependencies for performance improvements. In the following, we focus on those works that share goals or have technical commonalities with Pyro.

Approximate dependency discovery. While there are many works studying the discovery of FDs under various relaxations, only relatively few of them consider approximate FDs. To cope with the problem complexity, some discovery algorithms operate on samples of the profiled data and therefore cannot guarantee the correctness of their results [17,22,31] (that is, they only *approximate* the approximate FDs). This does not apply to Pyro. In addition, Cords discovers only *unary* AFDs [17], which is a much easier problem; and the authors of [31] detect only the *top k* AFDs according to an entropy-based measure without evaluating the qualitative impact of this restriction.

Another approach to harness the complexity is to use heuristics to prune potentially uninteresting AFD candidates [40]. Because this can cause the loss of interesting results, Pyro instead discovers *all* approximate dependencies for some given error threshold and leaves filtering or ranking of the dependencies to use-case specific post-processing. This prevents said loss and also frees users from the burden of selecting an appropriate interestingness threshold.

Along these lines, exact approaches for the discovery of AFDs and AUCCs have been devised. Arguably, the most adapted one is Tane [16], which converts the columns of a profiled relation into *stripped partitions (also: position list indices, PLIs)* and exhaustively combines them until it has discovered the minimal approximate dependencies. Being mainly designed for exact FD and UCC discovery, some of Tane's pruning rules do not work in the approximate case, leading to degraded performance. In fact, before discovering an approximate dependency involving n columns, Tane tests $2^n - 2$ candidates corresponding to subsets of these columns. Note that many works build upon Tane without changing these foundations [5,20,28]. Pyro avoids these problems by estimating the position of minimal approximate dependencies and then immediately verifying them.

Further approaches to infer approximate dependencies are based on the pairwise comparison of all tuples. The Fdep algorithm proposes (i) to compare all tuple pairs in a database, thereby counting any FD violations; (ii) to apply an error threshold to discard infrequent violations; and (iii) to deduce the AFDs from the residual violations [11,12]. We found this algorithm to yield incorrect results, though: Unlike exact FDs, AFDs can be violated by *combinations* of tuple pair-based violations, which Step (ii) neglects. In addition to that, the quadratic load of comparing all tuple pairs does not scale well to large relations [37]. In a related approach, Lopes et al. propose to use tuple pair comparisons to determine the most specific non-FDs in a given dataset whose error should then be calculated subsequently [30]. This approach is quite different from the aforementioned ones because it discovers only a small subset of all AFDs.

In a different line of work, an SQL-based algorithm for AFD discovery has been proposed [33]. As stated by the authors themselves, the focus of that work lies on ease of implementation in practical scenarios rather than performance.

Last but not least, the discovery of dependencies in the presence of `NULL` values has been studied [23]. This work considers replacements for `NULL` values, such that exact dependencies emerge. This problem is distinct from that of Pyro, which does not incorporate a special treatment of `NULL` but considers arbitrary dependency violations.

Exact dependency discovery. Many algorithms for the discovery of exact FDs and UCCs have been devised, e.g., [15,16,39,41]. These algorithms can generally be divided into (i) those that are based on the pairwise comparisons of tuples and scale well with the number of attributes and (ii) those that are based on PLI intersection and scale well with the number of tuples [37].

The algorithms Ducc [15] and the derived Dfd [2] belong to the latter and resemble Pyro in that they use a depth-first search space traversal strategy. Still, both exhibit substantial differences: While Ducc and Dfd perform a random walk through the search space, Pyro performs a sampling-based best-first search along with other techniques to reduce the number of dependency candidate tests. Interestingly, the authors of DFD suggest that it could be modified to discover AFDs and we will therefore consider a modified version of this algorithm in our evaluation.

The recent HyFD algorithm manages to scale well with growing numbers of tuples and columns by combining tuple comparisons and PLI intersections [38]. Pyro also combines these two base techniques. However, HyFD aggressively prunes FDs as soon as it discovers a violation of the same. While this pruning is key to HyFD's efficiency, it is not applicable to approximate dependencies. Instead, Pyro uses tuple comparisons to hypothesize dependency candidates rather than falsifying them and its search space traversal is adaptive rather than bottom-up.

Lattice search. In a broader sense, Pyro classifies nodes in a power set lattice, as we explain in Section 3. Apart from dependency discovery, several other problems, such as frequent itemset mining [3, 14], belong in this category and can be tackled with the same algorithmic foundations [32]. For instance, AFD discovery can be modeled as a frequent itemset mining problem; however, such adaptations require additional tailoring to be practically usable [40].

3. PROBLEM STATEMENT

Before outlining our algorithm Pyro, let us formalize the problem of finding the AFDs and AUCCs in a dataset. When referring to approximate dependencies, we need to quantify the degree of approximation. For that purpose, we use a slight adaptation of the well-established g_1 error [21] that ignores reflexive tuple pairs.

Definition 1 (AFD/AUCC error). Given a dataset r and an AFD candidate $X \rightarrow A$, we define its error as

$$e(X \rightarrow A, r) = \frac{|\{(t_1, t_2) \in r^2 \mid t_1[X]{=}t_2[X] \wedge t_1[A]{\neq}t_2[A]\}|}{|r|^2 - |r|}$$

Analogously, the error of an AUCC candidate X is defined as

$$e(X, r) = \frac{|\{(t_1, t_2) \in r^2 \mid t_1 \neq t_2 \wedge t_1[X]{=}t_2[X]\}|}{|r|^2 - |r|}$$

Example 1. For Table 1, we can calculate $e(\mathit{First\_name} \rightarrow \mathit{Gender}, r) = \frac{4}{5^2-5} = 0.2$ (violated by (t_1, t_5), (t_4, t_5), and their inverses) and $e(\{\mathit{First\_name}, \mathit{Last\_name}\}, r) = \frac{2}{5^2-5} = 0.1$ (violated by (t_1, t_4) and its inverse).

These error measures lend themselves for Pyro for two reasons. First, and as we demonstrate in Section 5, they can be easily calculated from different data structures. Second, and more importantly, they are *monotonous.* That is, for any AFD $X \rightarrow Y$ and an additional attribute A we have $e(X \rightarrow Y) \geq e(XA \rightarrow Y)$. Likewise for an AUCC X, we have $e(X) \geq e(XA)$. In other words, adding an attribute to the LHS of an AFD or to an AUCC can only remove violations but not introduce new violations. We refer to those $XA \rightarrow Y$ and XA, respectively, as *specializations* and to $X \rightarrow Y$ and X, respectively, as *generalizations.* With these observations, we can now precisely define our problem statement.

Problem Statement . Given a relation r and error thresholds e_ϕ and e_υ, we want to determine all *minimal* AFDs with a single RHS attribute and all *minimal* AUCCs. A minimal AFD has an AFD error of at most e_ϕ, while all its generalizations have an AFD error greater than e_ϕ. Analogously, a minimal AUCC has an AUCC error of at most e_υ, while all its generalizations have an AUCC error greater than e_υ.

Example 2. Assume we want to find all minimal AUCCs in Table 1 with the error threshold of $e_\upsilon = 0.1$. Amongst others, $\upsilon_1 = \{\mathit{First\_name}, \mathit{Last\_name}\}$ and $\upsilon_2 = \{\mathit{First\_name}, \mathit{Last\_name}, \mathit{Gender}\}$ have an AUCC of $0.1 \leq e_\upsilon$. However, υ_1 is a generalization of υ_2, so υ_2 is not minimal and need not be discovered explicitly.

Let us finally explain, why we exclude AFDs with *composite* RHSs, i.e., with more than one attribute. For exact dependency discovery, this exclusion is sensible because the FD $X \rightarrow AB$ holds if and only if $X \rightarrow A$ and $X \rightarrow B$ hold [4]. For AFDs as defined in Definition 1, this is no longer the case. However, considering composite RHSs potentially increases the number of AFDs drastically and might have serious performance implications. Furthermore, it is not clear how the use cases mentioned in Section 1 would benefit from such additional AFDs, or whether they would even be impaired by their huge number. Hence, we deliberately focus on single RHSs for pragmatic reasons and do so in accordance with related work [12, 16, 30]. Nevertheless, it can be shown that an AFD $X \rightarrow AB$ can hold only if both $X \rightarrow A$ and $X \rightarrow B$ hold. Hence, AFDs with a single RHS can be used to prune AFD candidates with composite RHSs.

4. ALGORITHM OVERVIEW

Before detailing Pyro's individual components, let us outline with the help of Algorithm 1 and Figure 1 how Pyro discovers all minimal AFDs and AUCCs for a given dataset and error thresholds e_ϕ and e_υ. For simplicity (but without loss of generality), we assume $e_\phi = e_\upsilon = e_{\max}$ for some user-defined $e_{\max}$. Furthermore, we refer to AFD and AUCC candidates with an error $\leq e_{\max}$ as *dependencies* and otherwise as *non-dependencies.* Now, given a dataset with n attributes, Pyro spawns $n{+}1$ *search spaces* (Line 1): one search space per attribute to discover the minimal AFDs with that very attribute as RHS; and one search space for AUCCs. The AUCC search space is a powerset lattice of all attributes, where each attribute set directly forms an AUCC candidate. Similarly, each AFD search space is a powerset lattice with all but the RHS attribute A, where each attribute set X represents the AFD candidate $X \rightarrow A$. In other words, each attribute set in the powerset lattices forms a unique dependency candidate. We may therefore use attribute sets and dependency candidates synonymously.

In a second preparatory step before the actual dependency discovery, Pyro builds up two auxiliary data structures (called *agree set sample (AS) cache* and *position list index (PLI) cache*; Lines 2–3), both of which support the discovery process for *all* search spaces by estimating or calculating the error of dependency candidates. We explain these data structures in Section 5.

Eventually, Pyro traverses each search space with a *separate-and-conquer* strategy to discover their minimal dependencies (Lines 4–5). Said strategy employs computationally inexpensive error estimates (via the AS cache) to quickly locate a promising minimal dependency candidate and then efficiently checks it with only few error calculations (via the PLI cache). As indicated in Figure 1, the verified (non-) dependencies are then used to prune considerable parts of

Algorithm 1: PYRO's general workflow.

```
  Data: Relation schema $R$ with instance $r$, AFD error
        threshold $e_\phi$, AUCC error threshold $e_v$
  ▷ Section 4
1 search-spaces ← {create-aucc-space($R, e_v$)}∪
    $\bigcup_{A \in R}$ create-afd-space($R \setminus \{A\}, A, e_\phi$)
  ▷ Section 5
2 pli-cache ← init-pli-cache($r$)
3 as-cache ← init-as-cache($r$, pli-cache)
  ▷ Section 6
4 foreach space ∈ search-spaces do
5     traverse(space, pli-cache, as-cache)
```

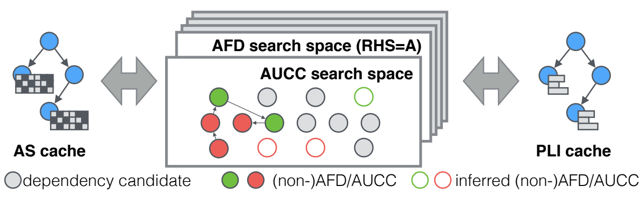

Figure 1: Intermediate state of Pyro while profiling a relation with the schema $R = (A, B, C, D)$.

the search space and as a result PYRO needs to inspect only the residual dependency candidates. Notice that our traversal strategy is sufficiently abstract to accommodate both AFD and AUCC discovery without any changes.

5. ERROR ASSESSMENT

As PYRO traverses a search space, it needs to estimate and calculate the error of dependency candidates. This section explains the data structures and algorithms to perform both operations efficiently.

5.1 PLI Cache

As we shall see in the following, both the error estimation and calculation involve *position list indices (PLIs)* (also known as *stripped partitions* [16]):

Definition 2 (PLI). Let r be a relation with schema R and let $X \subseteq R$ be a set of attributes. A *cluster* is a set of all tuple indices in r that have the same value for X, i.e., $c(t) = \{i \mid t_i[X] = t[X]\}$. The PLI of X is the set of all such clusters except for singleton clusters:

$$\bar{\pi}(X) := \{c(t) \mid t \in r \wedge |c(t)| > 1\}$$

We further define the *size* of a PLI as the number of included tuple indices, i.e., $\|\bar{\pi}(X)\| := \sum_{c \in \bar{\pi}(X)} |c|$.

Example 3. Consider the attribute *Last_name* in Table 1. Its associated PLI consists of the clusters $\{1, 4\}$ for the value *Smith* and $\{3, 5\}$ for the value *Miller*. The PLI does not include the singleton cluster for the value *Kramer*, though.

PYRO (and many related works, for that matter) employ PLIs for various reasons. First, and that is specific to PYRO, PLIs allow to create focused samples on the data, thereby enabling precise error estimates of dependency candidates. Second, PLIs have a low memory footprint because they store only tuple indices rather than actual values and omit singleton clusters completely. Third, the g_1 error can be directly calculated on them, as we show in the next section. Finally, $\bar{\pi}(XY)$ can be efficiently calculated from $\bar{\pi}(X)$ and $\bar{\pi}(Y)$ [16], denoted as *intersecting PLIs*. In consequence, we can represent any combination of attributes as a PLI.

For clarity, let us briefly describe the PLI intersection. As an example, consider the data from Table 1 and assume we want to intersect the PLIs $\bar{\pi}(\mathit{Firs\_name}) = \{\{1, 4, 5\}\}$ and $\bar{\pi}(\mathit{Last\_name}) = \{\{1, 4\}, \{2, 5\}\}$. In the first step, we convert $\bar{\pi}(\mathit{Last\_name})$ into the *attribute vector* $v_{\mathit{Last\_name}} = (1, 0, 2, 1, 2)$, which simply is a dictionary-compressed array of the attribute *Last_name* with one peculiarity: All values that appear only once are encoded as 0. This conversion is straight-forward: For each cluster in $\bar{\pi}(\mathit{Last\_name})$, we simply devise an arbitrary ID and write this ID into the positions contained in that cluster. In the second step, the *probing*, we group the tuple indices within each cluster of $\bar{\pi}(\mathit{First\_name})$. Concretely, the grouping key for the tuple index i is the i-th value in $v_{\mathit{Last\_name}}$ unless that value is 0: In that case the tuple index is dropped. For the cluster $\{1, 4, 5\}$, we obtain the groups $1 \to \{1, 4\}$ and $2 \to \{5\}$. Eventually, all groups with a size greater than 1 form the new PLI. In our example, we get $\bar{\pi}(\mathit{First\_name}, \mathit{Last\_name}) = \{\{1, 4\}\}$. Because t_1 and t_4 are the only tuples in Table 1 that agree in both *First_name* and *Last_name*, our calculated PLI indeed satisfies Definition 2.

That being said, intersecting PLIs is computationally expensive. Therefore, PYRO puts calculated PLIs into a *PLI cache* (cf. Figure 1) for later reuse. Caching PLIs has been proposed in context of the DUCC algorithm [15] (and was adopted by DFD [2]), however, a description of the caching data structure has not been given. It has been shown, however, that the set-trie of the FDEP algorithm [12] is suitable to index and look up PLIs [43].

As exemplified in Figure 2, PYRO's PLI cache adopts a similar strategy: It is essentially a trie (also: prefix tree) that associates attribute sets to their respective cached PLI. Assume we have calculated $\bar{\pi}(\{C, E\})$. Then we convert this attribute set into the list (C, E), which orders the attributes according to their order in the relation schema. Then, we index $\bar{\pi}(\{C, E\})$ in the trie using (C, E) as key.

However, when PYRO requests some PLI $\bar{\pi}(X)$, it may well not be in the cache. Still, we can leverage the cache by addressing the following criteria:

(1) We want to obtain $\bar{\pi}(X)$ with only few PLI intersections.
(2) In every intersection $\bar{\pi}(Y) \cap \bar{\pi}(Z)$, where we probe $\bar{\pi}(Y)$ against v_Z, we would like $\|\bar{\pi}(Y)\|$ to be small.

While Criterion 1 addresses the number of PLI intersections, Criterion 2 addresses the efficiency of the individual intersections, because probing few, small PLI clusters is beneficial performance-wise. Algorithm 2 considers both criteria to serve PLI requests utilizing the PLI cache. As an example, assume we want to construct the PLI $\bar{\pi}(ABCDE)$ with

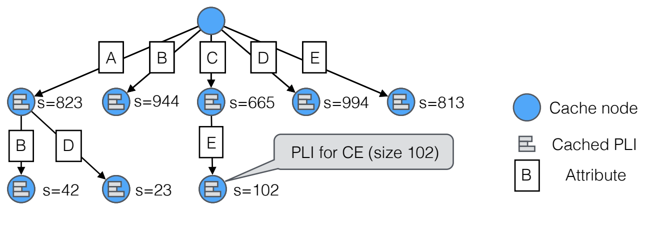

Figure 2: Example PLI cache.

Algorithm 2: Retrieve a PLI from the PLI cache.

```
   Data: PLI cache cache, attribute set $X$
1  $\Pi \leftarrow$ lookup PLIs for subsets of $X$ in cache
2  $\bar{\pi}(Y) \leftarrow$ pick the smallest PLI indices from $\Pi$
3  $\mathcal{Z} \leftarrow$ new list, $C \leftarrow Y$
4  while $C \subset X$ do
5      $\bar{\pi}(Z) \leftarrow$ pick PLI from $\Pi$ that maximizes $|Z \setminus C|$
6      append $\bar{\pi}(Z)$ to $\mathcal{Z}$
7      $C \leftarrow C \cup Z$
8  sort $\mathcal{Z}$ by the PLIs' sizes, $C \leftarrow Y$
9  foreach $\bar{\pi}(Z) \in \mathcal{Z}$ do
10     $\bar{\pi}(C \cup Z) \leftarrow \bar{\pi}(C) \cap \bar{\pi}(Z)$, $C \leftarrow C \cup Z$
11     if coin flip shows head then put $\bar{\pi}(C)$ into cache
12 return $\bar{\pi}(C)$
```

the PLI cache from Figure 2. At first, we look up all PLIs for subsets of $ABCDE$ in the cache (Line 1). This look-up can be done efficiently in tries. Among the retrieved PLIs, we pick the one $\bar{\pi}(Y)$ with the smallest size (Line 2). In our example, this is the case for $\bar{\pi}(AD)$ with a size of 23. This smallest PLI shall be used for probing in the first PLI intersection. The resulting PLI, which cannot be larger in size, will then be used for the subsequent intersection's probing and so on. This satisfies Criterion 2.

Next, we need to determine the remaining PLIs to probe against. Here, we follow Criterion 1 and repeatedly pick whatever PLI provides the most new attributes to those in the already picked PLIs (Lines 3–7). In our example, we thus pick $\bar{\pi}(CE)$, which provides two new attributes, and then $\bar{\pi}(B)$. Finally, all attributes in $ABCDE$ appear in at least one of the three selected PLIs. Note that PYRO always maintains PLIs for the single attributes in the PLI cache and can therefore serve any PLI request.

Having selected the PLIs, we intersect them using small PLIs as early as possible due to Criterion 2 (Lines 8–10). For our example, this yields the intersection order $(\bar{\pi}(AD) \cap \bar{\pi}(CE)) \cap \bar{\pi}(B)$. Compared to intersecting PLIs of single attributes, we save two out of four intersection operations. Additionally, we can use the PLI $\bar{\pi}(AD)$, which is much smaller than any single-attribute PLI. Hence, the PLI cache is useful to address both Criteria 1 and 2.

Finally, we cache randomly selected PLIs (Line 11). Caching *all* calculated PLIs would quickly fill the cache with redundant PLIs or those that will not be needed again. Our random approach, in contrast, caches frequently needed PLIs with a higher probability – with virtually no overhead.

5.2 Evaluating Dependency Candidates

PLIs are vital to calculate the error of an AFD or AUCC, respectively. Having shown how to efficiently obtain the PLI for some attribute set X, let us show how to calculate the g_1 error (see Definition 1) from $\bar{\pi}(X)$ in Algorithm 3.

For an AUCC candidate X, the error calculation given $\bar{\pi}(X)$ is trivial: We merely count all tuple pairs inside of each cluster because these are exactly the violating tuple pairs (Lines 1–2). In contrast, the error calculation of an AFD candidate $X \rightarrow A$ is a bit more complex. According to Definition 1, those tuple pairs violate $X \rightarrow A$ that agree in X and disagree in A. We do not count these tuple pairs directly. Instead, for each cluster of $\bar{\pi}(X)$ we calculate the

Algorithm 3: Error calculation for AFDs and AUCCs.

```
1  Function $e(X, r) =$ calc-AUCC-error$(\bar{\pi}(X), r)$
2      return $\sum_{c \in \bar{\pi}(X)} \frac{|c|^2 - |c|}{|r|^2 - |r|}$
3  Function $e(X \rightarrow A, r) =$ calc-AFD-error$(\bar{\pi}(X), v_A, r)$
4      $e \leftarrow 0$
5      foreach $c \in \bar{\pi}(X)$ do
6          counter $\leftarrow$ dictionary with default value 0
7          foreach $i \in c$ do
8              if $v_A[i] \neq 0$ then increase counter$[v_A[i]]$
9          $e \leftarrow e + \left(|c|^2 - |c|\right) - \sum_{c_A \in counter} \left(c_A^2 - c_A\right)$
10     return $\frac{e}{|r|^2 - |r|}$
```

number of tuple pairs also agreeing in A (Lines 4–8) and then subtract this number from all tuple pairs in the cluster (Lines 9–10). For this calculation, we need the attribute vector v_A of attribute A (cf. Section 5.1), in addition to $\bar{\pi}(X)$. Note that we must not count zeros in v_A, because they represent singleton values. By summing the errors of all clusters in $\bar{\pi}(X)$, we finally obtain $e(X \rightarrow A, r)$.

5.3 Estimating Dependency Errors

A key idea of PYRO is to avoid costly PLI-based error calculations by estimating the errors of dependency candidates and only then conduct a few targeted error calculations. As a matter of fact, an error *calculation* can be orders of magnitudes slower than an error *estimation*. Generally speaking, we can estimate dependency errors by comparing a subset of tuples – or better: a subset of tuple pairs – and extrapolate the number of encountered violations to the whole relation. Such error estimation is related to (but far more efficient than) algorithms that exhaustively compare *all* tuple pairs to discover dependencies [12,30]. The basis for this approach are *agree set samples (AS samples)*.

Definition 3 (AS sample). Given a relational instance r with schema R and two tuples $t_1, t_2 \in r$, their *agree set* [6] is $ag(t_1, t_2) := \{A \in R \mid t_1[A] = t_2[A]\}$.[1] Further, let $s \subseteq r^2$ be a sample of tuple pairs of the relational instance r. Then, s induces the AS sample

$$\mathsf{AS} := \{(a, c(a)) \mid \exists (t_1, t_2) \in s \colon a = ag(t_1, t_2)\}$$

where $c(a) := |\{(t_1, t_2) \in s \mid a = ag(t_1, t_2)\}|$ counts the number of occurrences of each agree set in s.

Example 4. Assume that we randomly sample three tuple pairs from Table 1, e.g., (t_1, t_3), (t_1, t_5), and (t_2, t_3). This gives us the AS sample $\mathsf{AS} = \{(\{Gender, Town\}, 1), (\{First\_name, Town\}, 1), (\{Gender, ZIP\}, 1)\}$.

Now to estimate AFD and AUCC errors from an AS sample AS, we define a query that reports the number of agree sets in AS that include some attribute set *inc* and do not contain any attribute of a further attribute set *exc*:

$$\texttt{count}(\mathsf{AS}, inc, exc) := \sum_{(a,c) \in \mathsf{AS}} \begin{cases} c & \text{if } inc \subseteq a \wedge exc \cap a = \emptyset \\ 0 & \text{otherwise} \end{cases}$$

[1]For improved memory and computation efficiency, we calculate agree sets from cached attribute vectors (see Section 5.1) rather than the original input dataset.

Pyro stores agree sets efficiently as bit masks using one-hot encoding. This allows to keep AS samples in memory and perform count efficiently with a full scan over the AS sample. So, to estimate the error of an AFD candidate $X \rightarrow A$, we could count its numbers of violating agree sets in AS as $\texttt{count}(\mathsf{AS}, X, \{A\})$ and divide the result by $\|\mathsf{AS}\|$. Likewise, for an AUCC candidate X, we could count the violations $\texttt{count}(\mathsf{AS}, X, \emptyset)$ and, again, divide by $\|\mathsf{AS}\|$:

Lemma 1. Let $\hat{e}$ be the estimated error of an AFD or AUCC candidate using the AS sample AS. Further, let e denote the actual dependency error. Then $\hat{e}$ is unbiased, i. e., its expectation value is exactly e, and the probability that $|e - \hat{e}| \leq \varepsilon$ for some user-defined ε is given by

$$P_\varepsilon(\mathsf{AS}, e) := \sum_{i=\lceil (e-\varepsilon)\cdot\|\mathsf{AS}\|\rceil}^{\lfloor (e+\varepsilon)\cdot\|\mathsf{AS}\|\rfloor} \binom{\|\mathsf{AS}\|}{i} e^i (1-e)^{\|\mathsf{AS}\|-i}$$

Proof. Sampling n tuple pairs and testing whether they violate an AUCC or AFD candidate follows a binomial distribution whose probability parameter is exactly the dependency error as defined in Definition 1. The mean of this distribution, i. e., the expected number of violations in AS, is $e \cdot \|\mathsf{AS}\|$ $(= \mathbb{E}[\hat{e}] \cdot \|\mathsf{AS}\|)$ and the above error bounds can be immediately derived from the cumulative distribution function of the binomial distribution. □

Interestingly, the accuracy of our error estimates does not depend on the size of the input relation, which makes it highly scalable. Instead, we observe an influence of the actual dependency error e. Indeed, the variance of the binomial distribution (and thus the uncertainty of our estimator) is maximized for $e = 0.5$. However, for typical error thresholds we need accurate estimates only when e approaches 0 to tell apart partial dependencies and non-dependencies. For instance, for $e = 0.01$ and $\|AS\| = 1,000$ our error estimates are off by at most 0.006 with a probability of 0.96.

However, if we had only a single AS sample for the whole relation, this sample might need to be huge to achieve high precision estimates when needed: The standard deviation of our bionomially distributed error estimator, $\sqrt{\frac{e(1-e)}{\|\mathsf{AS}\|}}$, is inversely proportional to the square root of the sample size. Intuitively, one might suspect that the above accuracy for $\|AS\| = 1,000$ is sufficient to discover partial AFDs and AUCCs with an error threshold of 0.01, but that is not necessarily the case. As an example, assume a relation with n attributes A_i $(1 \leq i \leq n)$, each having an AUCC error of 0.0101, while their combination $A_1 \ldots A_n$ has an error of 0.0999. In this scenario, any set of two or more attributes might be a minimal AUCCs and, for that matter, there are $2^n - (n + 1)$ such sets. Obviously, we would need samples with much more than the above 1,000 agree sets to reasonably predict where the minimal AUCCs might be, which would come at a high cost.

To provide high precision error estimates from small AS samples, Pyro uses *focused sampling.* However, the resulting samples must still be random, so as to preserve the above explained unbiasedness of our estimator. To solve this conflict, we sample only such agree sets a that are supersets of some given attribute set X, i.e. $a \supseteq X$. Such a sample can be created efficiently: We obtain the PLI $\bar{\pi}(X)$ and then sample only such tuple pairs that co-occur in some cluster $c \in \bar{\pi}(X)$. As an example, consider the PLI $\bar{\pi}(\{Zip\}) = \{\{1, 4\}, \{2, 3, 5\}\}$ for Table 1. This PLI restricts the sampling to tuple pairs from $\{t_1, t_4\}$ or $\{t_2, t_3, t_5\}$.

In detail, to sample a tuple pair that agrees in X, we first select a cluster $c' \in \bar{\pi}(X)$ with a probability of $\frac{|c'|^2-|c'|}{pairs(X)}$ where $pairs(X) := \sum_{c\in\bar{\pi}(X)} |c|^2 - |c|$ denote the number of overall tuple pairs agreeing in X. That is, the probability of picking c' is proportional to the number of its tuple pairs. Then, we randomly sample two distinct tuples from c', so that each tuple pair within c' is sampled with the probability $\frac{1}{|c'|^2-|c'|}$. In consequence, *any* tuple pair with tuples agreeing in X from the input relation has the same probability $\frac{1}{pairs(X)}$ of being sampled. Finally, we calculate the agree sets for the sampled tuple pairs and obtain a focused, yet random, AS sample, denoted AS_X.

Based on AS_X, we can now estimate the error of any AFD candidate $Y \rightarrow A$ and AUCC candidate Y if $Y \supseteq X$. In fact, the error of the AUCC candidate Y in a relation r can be estimated as

$$\hat{e}(Y, r) := \frac{\texttt{count}(\mathsf{AS}_X, Y, \emptyset)}{\|\mathsf{AS}_X\|} \cdot \frac{pairs(X)}{|r|^2 - |r|}$$

and the error of the AFD candidate $Y \rightarrow A$ as

$$\hat{e}(Y \rightarrow A, r) := \frac{\texttt{count}(\mathsf{AS}_X, Y, \{A\})}{\|\mathsf{AS}_X\|} \cdot \frac{pairs(X)}{|r|^2 - |r|}$$

where $\|\mathsf{AS}_X\| := \sum_{(a,c)\in\mathsf{AS}_X} c$.

Theorem 1. Given an AUCC candidate Y or AFD candidate $Y \rightarrow A$ $(Y \supseteq X)$, our focused estimators based on sample AS_X are unbiased and the probability that $|e - \hat{e}| \leq \varepsilon$ for an actual dependency error e, error estimate $\hat{e}$, some user-defined ε is given by $P_\varepsilon(\mathsf{AS}_X, e\frac{|r|^2-|r|}{pairs(X)})$.

Proof. The first terms of the estimators estimate the ratio of the tuple pairs violating the dependency candidate among all tuple pairs agreeing in X; Lemma 1 shows their unbiasedness and error bounds. Because all violating tuple pairs must agree in X, the additional terms *exactly* extrapolate this "focused" estimate to the whole relation, thereby preserving the unbiasedness and shrinking the error bounds by a constant factor. □

Theorem 1 explains why focused samples are effective. Consider again the *ZIP* column in Table 1: Out of all 10 tuple pairs, only 4 agree in their *ZIP* values, so that a *ZIP*-focused estimator is $\frac{10}{4} = 2.5\times$ more precise than an unfocused estimator with the same sample size and confidence level. This effect is even stronger in larger, real-world data sets. For instance, a focused estimator for an attribute with 1,000 equally distributed values shrinks the error bounds by a factor of 10^6. Hence, it is more efficient to create and use multiple focused samples rather than one highly extensive one. In fact, Pyro operates only on focused samples – initially one for each attribute.

Having explained focused AS samples and how to estimate AFD and AUCC errors with them, it remains to be shown how Pyro serves an actual request for an error estimate of some dependency candidate. Without loss of generality, assume that the dependency candidate in question is the AUCC candidate Y. As for the PLIs, whenever Pyro creates an AS sample, it caches it in a trie (cf. Figure 1 and Section 5.1). Pyro first determines all AS samples with focus X for some $X \subseteq Y$, which can be done efficiently because the AS cache is a trie. Recall that the focus of the AS sample must be a subset of the dependency candidate to obtain an unbiased estimate. Then, PYRO picks the AS sample with the highest *sampling ratio*, i. e.,the ratio of sampled agree sets to the population; formally $\frac{\|\mathsf{AS}_X\|}{pairs(X)}$. The reason is that larger AS samples and smaller sampling foci yield more precise error estimates (see Theorem 1). What is more, when the population for the sample AS_X is very small (because X is almost a UCC), then the sample can even be exhaustive and, hence, the error estimate is known to be exact and need not be calculated with PLIs anymore.

6. SEARCH SPACE TRAVERSAL

To discover dependencies, it is not only important to efficiently assess individual dependency candidates as explained in the above section; a search space traversal strategy that combines efficient error estimations and final error calculations is crucial, too. In contrast to TANE, which systematically traverses large parts of the search space, and in contrast to DUCC/DFD, which perform random walks through the search space, PYRO employs a novel separate-and-conquer strategy: It separates a part of the search space, estimates the position of the minimal dependencies within that subspace, and then validates this estimate. Afterwards, considerable parts of the search space can be pruned.

Let us outline PYRO's traversal strategy more concretely with the example in Figure 3 before elaborating on its individual phases in detail. The traversal can be thought of as a firework display – which inspired the name PYRO. It consists of multiple rounds starting from the single attributes, the *launchpads*. In our example, PYRO selects A as launchpad[2] and *ascends* (like a skyrocket) in the search space until it detects the dependency $ABCD$ (Step (1), Section 6.1). From this dependency, called *peak*, PYRO *trickles down* (like an exploded skyrocket) and estimates the position of all *minimal* dependencies that generalize the peak (Step (2), Section 6.2), which is the case for CD. Then, it verifies the estimate by checking the *complementary*, untested dependency candidates, namely ABD (Step (3), Section 6.3).

This completes the first search round and, as shown in Figure 3, PYRO uses both discovered non-dependencies and dependencies to drastically narrow down the search space for subsequent search rounds. In fact, discovered (non-) dependencies are stored in a trie (cf. Section 5.1) to efficiently determine whether following dependency candidates are already pruned. Finally, in the next search round, PYRO might pick up a pruned launchpad; in Figure 3, we pick again A. In that case, PYRO *escapes* the launchpad into the unpruned part of the search space (Step (4), Section 6.4).

6.1 Ascend

The ascend phase should efficiently determine some dependency in a given search space, which will then form the input to the subsequent trickle-down phase. Algorithm 4 depicts how PYRO achieves that. The ascend starts at the *launchpads*, where are minimal dependency candidates with an unknown state. Hence, the single attributes are the initial launchpads. PYRO estimates their error and picks the one with the smallest error, e.g., attribute A as in Figure 3, assuming it to lead to a dependency quickly (Line 1).

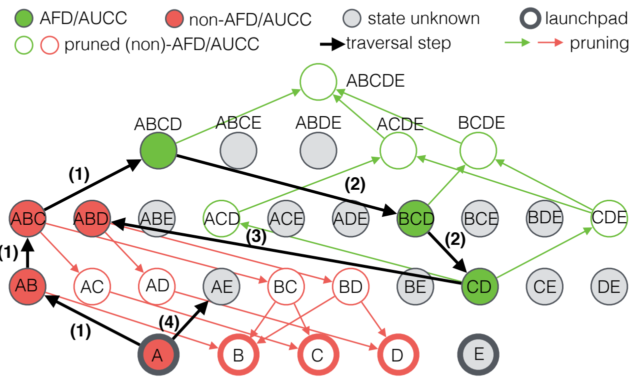

Figure 3: Example of a search space traversal round.

Algorithm 4: The ascend phase.

```
   Data: launchpads $L$, maximum error $e_{\max}$
1  $(X, \hat{e}_X) \leftarrow$ pick launchpad with smallest error estimate $\hat{e}_X$ from $L$
2  while True do
3      if $\hat{e}_X \leq e_{max}$ then
4          if $\hat{e}_X$ is not known to be exact then
5              $\hat{e}_X \leftarrow$ calculate error of $X$
6          if $\hat{e}_X \leq e_{max}$ then break
7      $A \leftarrow \arg\min_{A \in R \setminus X} \hat{e}_{XA}$ with $XA$ is not pruned
8      if no such $A$ then break
9      $X \leftarrow XA$
10     $\hat{e}_X \leftarrow$ estimate error of $X$
11 if $\hat{e}_X$ is not known to be exact then $\hat{e}_X \leftarrow e_X$
12 if $e_X \leq e_{max}$ then
13     trickle down from $X$
14 else
15     declare $X$ a maxmimum non-dependency
```

Then, PYRO greedily adds that attribute to the launchpad that reduces the (estimated) dependency error the most (Line 7), until either a dependency is met (Line 6) or no attribute can be added anymore (Line 8). The latter case occurs when there simply is no attribute left to add or when all possible candidates are already known dependencies from previous rounds. In this case, we declare it as a maximum non-dependency for pruning purposes and cease the current search round (Line 15). However, if we meet a dependency, as is the case for $ABCD$ in Figure 3, we proceed with the trickle-down phase starting from that dependency (Line 13).

6.2 Trickle down

Given a dependency from the ascend phase, called *peak* P, PYRO trickles down to estimate the position of *all minimal* dependencies that generalize P. Algorithm 5 outlines how PYRO performs this estimation. First, P is placed into a new priority queue that orders peaks by their estimated dependency error (Line 2). PYRO then takes the smallest element (which initially is P) without removal (Line 4), checks

[2]Recall from Section 4 that we use dependency candidates and attribute sets synonymously: A can be an AUCC candidate or an AFD candidate $A \rightarrow R$ for some RHS R. As a result, the traversal works for both dependency types.

Algorithm 5: Estimate position of minimal dependencies.

```
   Data: peak $P$, maximum error $e_{\max}$
1  $\mathcal{M} \leftarrow$ new trie
2  $\mathcal{P} \leftarrow$ new priority queue with $P$
3  while $\mathcal{P}$ is not empty do
4      $P' \leftarrow$ peek from $\mathcal{P}$
5      $\mathcal{M}' \leftarrow$ look up subsets of $P'$ in $\mathcal{M}$
6      if $\mathcal{M}' \neq \emptyset$ then
7          remove $P'$ from $\mathcal{P}$
8          foreach $H \in$ hitting-set($\mathcal{M}'$) do
9              if $P' \setminus H$ is not an (estimated) non-dependency then
10                 add $P' \setminus H$ to $\mathcal{P}$
11     else
12         $M \leftarrow$ trickle-down-from($P', e_{\max}$)
13         if $M \neq \bot$ then add $M$ to $\mathcal{M}$ else remove $P'$ from $\mathcal{P}$
14 Function trickle-down-from($P', e_{max}$)
15     if $|P'| > 1$ then
16         $\mathcal{G} \leftarrow$ error-based priority queue with generalizations of $P'$
17         while $\mathcal{G}$ is not empty do
18             $G, \hat{e}_G \leftarrow$ poll from $\mathcal{G}$
19             if $\hat{e}_G > e_{max}$ then break
20             $C \leftarrow$ trickle-down-from($G, e_{\max}$)
21             if $C \neq \bot$ then return $C$
22     $e_{P'} \leftarrow$ calculate error of $P'$
23     if $e_{P'} \leq e_{max}$ then return $P'$
24     create and cache AS sample with focus $P'$
25     return $\bot$
```

whether it is pruned by some already estimated minimal dependencies (Line 5) (which is initially not the case), and then invokes the function `trickle-down-from` with P whose purpose is to estimate the position of exactly one minimal dependency that generalizes P or return $\bot$ if P and none of its generalizations are estimated to be a dependency (Line 12).

In our example, we initially invoke `trickle-down-from` with our peak $P = ABCD$. It now creates a priority queue that orders the immediate generalizations of P, i.e., ABC, ABD etc., by their estimated dependency error (Line 16). These generalizations are potential minimal dependencies, therefore any of them with an estimated error of less than $e_{\max}$ is recursively trickled down from (Lines 17–21). If the recursion yields a minimal dependency candidate, Pyro immediately reports it. In Figure 3, we recursively visit BCD and then CD. Neither C nor D is estimated to be a dependency, so CD might be a minimal dependency. Eventually, we calculate the error of CD to make sure that it actually is a dependency and return it (Lines 22–23). If, in contrast, we had falsely assumed CD to be a dependency, we would create a focused sample on CD so as to obtain better error estimates for dependency candidates between CD and the peak $ABCD$ and continue the search at BCD.

Finally, we add CD to the estimated minimal dependencies $\mathcal{M}$ (Line 13) and peek again from the peak priority queue (Line 4), which still contains the original peak $ABCD$. However, now there is the alleged minimal dependency CD,

Algorithm 6: Calculate minimal hitting sets.

```
   Data: attribute sets $\mathcal{S}$, relation schema $R$
1  Function hitting-set($\mathcal{S}$)
2      $T \leftarrow$ set trie initialized with $\emptyset$
3      $L_{\mathcal{S}} \leftarrow$ list of elements in $\mathcal{S}$
4      sort $L_{\mathcal{S}}$ ascending by set size
5      foreach $S \in L_{\mathcal{S}}$ do
6          $\overline{S} \leftarrow R \setminus S$
7          $\mathcal{V} \leftarrow$ remove all subsets of $\overline{S}$ from $T$
8          foreach $V \in \mathcal{V}$ do
9              foreach $A \in S$ do
10                 if no subset of $VA$ is in $T$ then
11                     add $VA$ to $T$
12     return $T$
```

which explains $ABCD$. Still, we must not simply discard this peak, because there might be further minimal dependencies generalizing it. Therefore, we identify all maximal dependency candidates that are a subset of $ABCD$ but not a superset of CD. As detailed below, Pyro determines those candidates by calculating the *minimal hitting sets* of CD, namely C and D, and removing them from $ABCD$ (Line 8–10), which yields ABD and ABC. These form the new peaks from which the search for minimal dependencies is continued. In our example, we estimate both to be non-dependencies and remove them from the queue (Line 13).

Let us now explain the hitting set calculation in more detail. Formally, a set is a hitting set of a set family $\mathcal{S}$ (here: a set of attribute sets) if it intersects every set $S \in \mathcal{S}$. It is *minimal* if none of its subsets is a hitting set. The calculation of minimal hitting sets is employed in the following traversal steps, too, and furthermore constitutes an NP-hard problem [18]. Facing this computational complexity, the problem should be solved as efficiently as possible. Algorithm 6 displays how Pyro calculates *all* minimal hitting sets for a set of attribute sets.

First, we initialize a set trie (cf. Section 5.1) with the empty set as initial solution (Line 2). Next, we order the attribute sets by size (Lines 3–4). If $\mathcal{S}$ contains two sets A and B with $A \subseteq B$, we want to process A first, because any intermediate hitting set that intersects A will also intersect B. When processing B, we do not need to update the intermediate solution. Then, we iterate the ordered input attribute sets one after another (Line 5). Assume, we have $S = CD$ as above. Then, we remove all current hitting sets that do not intersect with S by looking up subsets of its inversion (Line 6–7); recall that we can perform subset lookups on tries efficiently. In our example, the inversion of CD is ABE and the initial solution in T, the empty set, is a subset of it. Eventually, we combine all removed sets with all attributes in S to re-establish the hitting set property (Line 8–11). For instance, combining $V = \emptyset$ with $S = CD$ yields the two new hitting sets C and D. However, these new hitting sets might not be minimal. Therefore, before adding a new hitting set H back to the trie, we check if there is an existing minimal hitting set in the trie that is a subset of H. Again, subset look-ups can be performed efficiently on tries. After all attributes from $\mathcal{S}$ have been processed, the trie contains all minimal hitting sets.

6.3 Validate

While the estimated set of minimal dependencies $\mathcal{M}$ from the trickle-down phase contains only verified dependencies, it is not known whether these dependencies are minimal and whether $\mathcal{M}$ is complete. For instance, we might have incorrectly deemed several dependencies to be non-dependencies in the in ascend or trickle-down phase. Pyro validates the completeness of $\mathcal{M}$ with as few error calculations as possible. $\mathcal{M}$ is complete if and only if any dependency candidate $X \subseteq P$ is either a specialization of some allegedly minimal dependency $Y \in \mathcal{M}$, i.e., $X \supseteq Y$, or X is a non-dependency.

To test this, it suffices to test the *maximal* alleged non-dependencies "beneath" the peak, i.e., those dependency candidates that generalize P and whose specializations are all known dependencies. If these maximal candidates are indeed non-dependencies, then so are all their generalizations and $\mathcal{M}$ is complete. As in Algorithm 5, Pyro calculates these candidates, denoted as $\overline{\mathcal{M}}$, by calculating the minimal hitting sets of all elements in $\mathcal{M}$ and removing them from P. For our example, we have $\texttt{hitting-sets}(\{CD\}) = \{C, D\}$ and thus need to check $P \backslash C = ABD$ and $P \backslash D = ABC$.

Pyro checks all candidates in $\overline{\mathcal{M}}$ with two possible outcomes. If those candidates are non-dependencies, then $\mathcal{M}$ is indeed complete. If, however, $\overline{\mathcal{M}}$ contains dependencies, $\mathcal{M}$ is not complete. Nonetheless, we can use this result to narrow the focus our search.

Let $\mathcal{D} \subseteq \overline{\mathcal{M}}$ denote said dependencies in $\overline{\mathcal{M}}$ and assume that ABD turned out to be a dependency, i.e., $\mathcal{D} = \{ABD\}$. Any dependency not covered by $\mathcal{M}$ must be a generalization of some dependency in $\mathcal{D}$, because any candidate $X \subseteq P$ is either a superset of some element in $\mathcal{M}$ or a subset of some element $\overline{\mathcal{M}}$. Further, let $\mathcal{N} = \overline{\mathcal{M}} \setminus \mathcal{D}$ denote the non-dependencies in $\overline{\mathcal{M}}$. In our modified example, we have $\mathcal{N} = \{ABC\}$. These are maximal w.r.t. the peak P, i.e., all their supersets that are also subsets of P are known dependencies. We can now determine the dependency candidates that are not subsets of any such maximal non-dependency in $\mathcal{N}$, denoted as $\overline{\mathcal{N}}$: We invert all elements in $\mathcal{N}$ w.r.t. P and calculate their minimal hitting sets. For $\mathcal{N} = \{ABC\}$, we get $\overline{\mathcal{N}} = \{D\}$. It follows that any dependency not covered by $\mathcal{M}$ is a specialization of some dependency candidate in $\overline{\mathcal{N}}$ and a generalization of some dependency in $\mathcal{D}$, i.e., in our example the unknown minimal dependencies must be a superset of D and a subset of ABD.

As a result, Pyro can create a search sub-space with exactly those dependency candidates and recursively process it, including all steps presented in this section. In addition, Pyro *boosts* the size of AS samples while working in this sub-space so as to decrease the probability of new mispredictions about the minimal dependencies. Still, in the case of another misprediction Pyro can recursively create new sub-spaces. The recursion is guaranteed to terminate, though, because the sub-spaces are continuously shrinking. However, in our experiments we rarely saw a recursion depth of even 2. After the recursion, Pyro eventually needs to check for which dependencies in $\mathcal{M}$ the recursion has not yielded a generalizing minimal dependency. Those dependencies were minimal all along and must be reported as such.

6.4 Escape

After each search round, great parts of the search space can be pruned. As shown in Figure 3, also launchpads might now be in the pruned space. Unless these launchpads have been found to be (minimal) dependencies, we must not discard them, though: There might still be undiscovered minimal dependencies that are supersets of that launchpad.

Whenever Pyro picks up a pruned launchpad, it needs to *escape* it out of the pruned search space part by adding a minimum amount of attributes to it. Let us assume that Pyro picks up the launchpad A once more. To determine whether it is pruned, Pyro determines all previously visited peaks that are supersets of A, which is $ABCD$ in our case. Again, a hitting set calculation can now determine minimum attribute sets to add to A, such that it is not a subset of $ABCD$ anymore: Pyro calculates the hitting sets of $R \setminus ABCD = E$, which is simply E. By adding E to A, we get the only minimal escaping (cf. Step (4) in Figure 3). Note that this operation is the exact inverse of the relocation of peaks in Algorithm 5. However, because we have the launchpad E, which is a subset of AE, we finally have to discard AE and, as a matter of fact, all unknown dependency candidates are indeed supersets of E.

Because Pyro maintains the launchpads, such that they form exactly the minimal untested dependency candidates, a search space is completed when it contains no more launchpads. As a result, Pyro will eventually terminate with the complete set of dependencies of the search space.

7. EVALUATION

Pyro aims for efficient and scalable discovery of approximate dependencies in given datasets. To evaluate in how far Pyro attains this goal, we empirically investigate how Pyro compares to (extensions of) three state-of-the-art algorithms in terms of efficiency and scalability and furthermore conduct several in-depth analyses of Pyro. A theoretical comparison of the algorithms would be of limited value, because in the worst case AUCC/AFD discovery is of exponential complexity w. r. t. the number of profiled attributes due to the possibly exponential numbers of dependencies [2, 29] – even a brute-force algorithm that checks every dependency candidate would meet that complexity. An average-case complexity analysis, on the other hand, would have to resort to strong, perhaps unjustifiable, assumptions on the data due to the adaptivity of the algorithms. That being said, we close the evaluation with an investigation of the interestingness of AUCCs and AFDs.

7.1 Experimental setup

We have carefully (re-)implemented Pyro, Tane, Fdep, and Ducc/Dfd in Java, so that their runtimes are comparable. For easy repeatability of our experiments, all algorithms are integrated with the profiling frameworks Metanome [36] and Metacrate [25]. Our Pyro prototype implements a simple parallelization strategy ("Pyro (parallel)") that traverses multiple search spaces in parallel and also executes multiple search rounds in a single search space simultaneously, when there are fewer search spaces than available cores. Pyro further monitors the memory usage of the PLI and AS caches and halves them when running low on memory. Additionally, we fixed a known issue regarding Tane's key-pruning [37] and extended both Tane and Fdep to output also AUCCs. For Fdep and Ducc/Dfd specifically, we consulted the original implementations whenever we found that their respective publications were not specific enough. Eventually, we modified Ducc/Dfd to discover approximate dependencies as suggested in [2]. To our knowledge, this

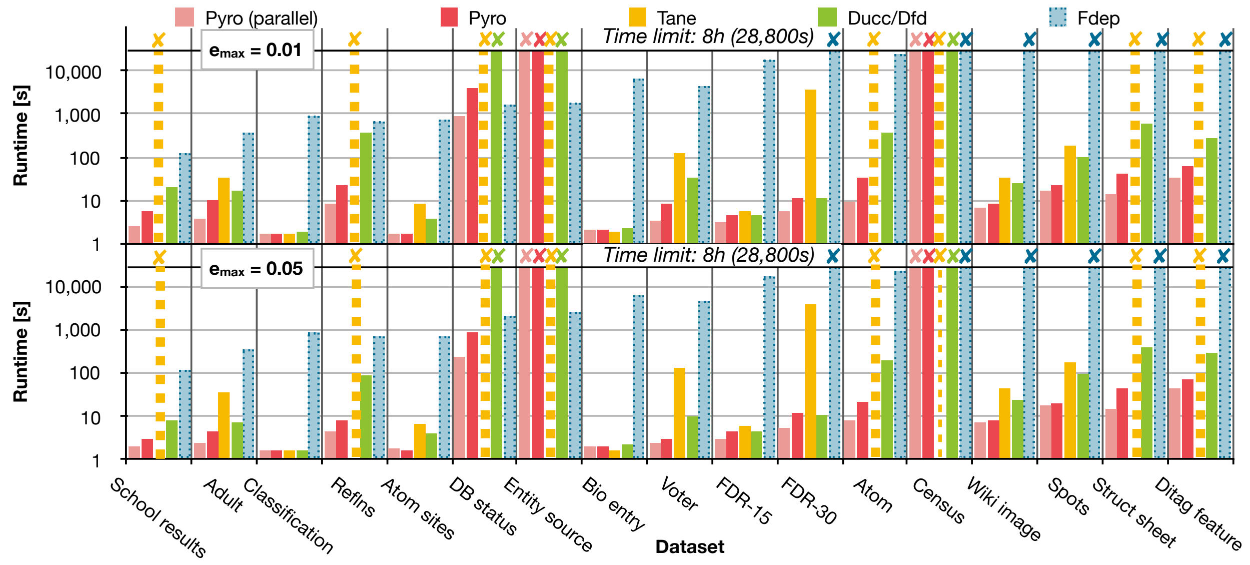

Figure 4: Runtime comparison of Pyro, Tane, Ducc/Dfd, and Fdep. The crosses indicate that an algorithm either ran out of time or memory.

modification was never implemented or evaluated. Therefore, we believe that the respective experiments are interesting in itself.

We conducted our experiments on a Dell PowerEdge R430 with an Intel Xeon E5-2630 v4 (2.2 GHz, 10 cores, 20 threads, 25 MB cache), 32 GB RAM (2.4 MT/s), and 2 hard disks (7.2 kRPM, SATA) running Ubuntu 16.04.1 and Oracle JDK 1.8.0_112 with a maximum heap size of 24 GB. Unless specified otherwise, we considered an error threshold of 0.01 for both AFDs and AUCCs and used an initial AS sample size of 1,000 and a boost factor of 2 with Pyro. The partition cache size for Ducc/Dfd was set to 1,000 in accordance to the suggested value and our main memory size [2]. Our datasets were stored as CSV files on disk. Their sizes can be found in Table 2. For repeatability purposes, our implementations and pointers to the datasets can be found on our webpage: `https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html`.

Table 2: Overview of the evaluation datasets sorted by their size (columns × tuples).

Dataset			AFDs + AUCCs	
Name	Cols.	Rows	$e_{max} = 0.01$	$e_{max} = 0.05$
School results	27	14,384	3,408	1,527
Adult	15	32,561	1,848	1,015
Classification	12	70,859	119	1311
Reflns	37	24,769	9,396	3,345
Atom sites	31	32,485	79	78
DB status	35	29,787	108,003	45,617
Entity source	46	26,139	*(unknown)*	*(unknown)*
Bio entry	9	184,292	29	39
Voter	19	100,001	647	201
FDR-15	15	250,001	225	225
FDR-30	30	250,001	900	900
Atom	31	160,000	1,582	875
Census	42	199,524	*(unknown)*	*(unknown)*
Wiki image	12	777,676	92	74
Spots	15	973,510	75	79
Struct sheet	32	664,128	1,096	1,458
Ditag feature	13	3,960,124	187	260

7.2 Efficiency

Let us begin with a broad comparison of all four algorithms to show in how far Pyro advances the state of the art in AUCC and AFD discovery. For that purpose, we ran all algorithms on the datasets from Table 2 on two different, typical error thresholds. Note that we report runtimes for Pyro in a parallel version (to show its best runtimes) and a non-parallel version (for comparison with the other algorithms). We also report runtimes for the logically flawed Fdep algorithm (see Section 2) to include an algorithm that is based on the comparison of all tuple pairs in a dataset. The results are shown in Figure 4.

Obviously, Pyro is the most efficient among the correct algorithms: While the non-parallel version is outperformed on some easy-to-process datasets by at most 0.6 seconds, Pyro outperforms its competitors by at least a factor of 2 in 59 % of the configurations. For that matter, for hard-to-process datasets we observe the greatest speed-ups. For instance on the *DB_status* dataset for $e_{\max} = 0.05$, Pyro outperforms the best competitor, Ducc/Dfd, *at least* by a factor of 33 (or 7.7 h in absolute terms) – for the parallel version, we even measured a speed-up factor of at least 123. The actual speed-up might even be greater, because Ducc/Dfd did not complete the profiling within the 8 h time limit. This shows that Pyro's approach to estimate and verify dependency candidates is much more efficient than Tane's breadth-first search and Ducc/Dfd's random walk search.

The above comparison omits Fdep, because it reports incorrect results (see Section 2). Nonetheless, we find it to be regularly inferior in terms of performance. The reason is that it compares all possible tuple pairs in a dataset, thereby incurring quadratic load in the number of tuples, which is prohibitive on larger datasets. On the other hand, the few scenarios where Fdep is faster than Pyro can be attributed to its logical flaw: For instance on the *DB_status* for $e_{\max} = 0.01$, Fdep reports only 15,138 supposed dependencies – 7× fewer than there actually are. And what is more, 67 % of these supposed dependencies are incorrect. Correct-

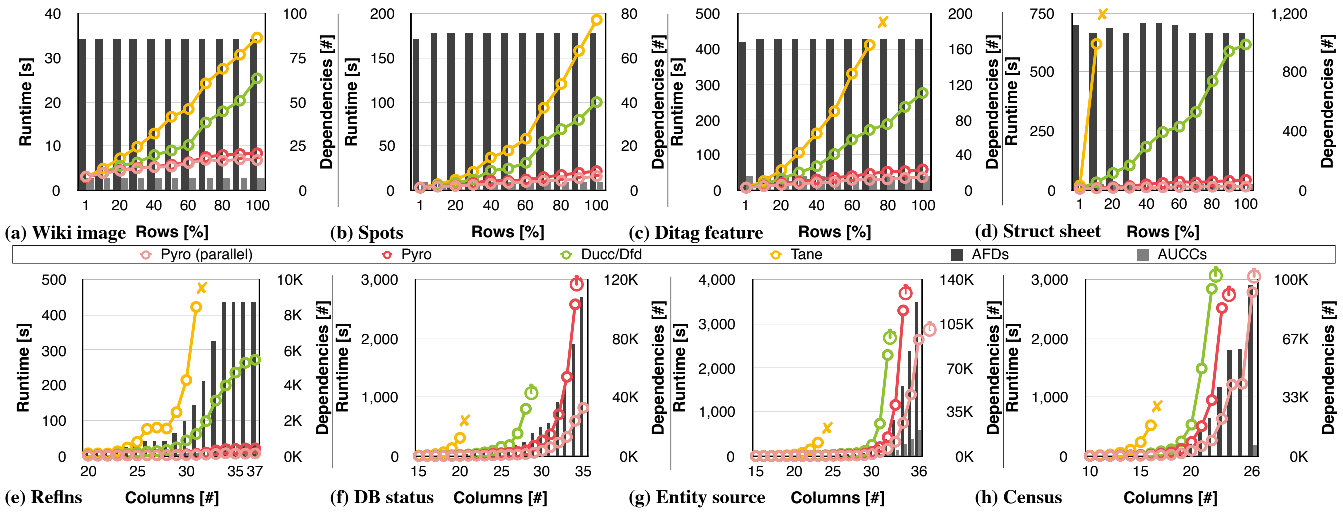

Figure 5: Row and column scalability of Pyro, Ducc/Dfd, and Tane ($e_{\max} = 0.01$). Crosses indicate that an algorithm ran out of memory and stopwatches mean that the algorithm exceeded the 1 h time limit.

ing those results would require additional time. Note that, although PYRO also compares tuple pairs, it does not run into the quadratic trap because it requires only a constant number of comparisons (cf. Theorem 1) and reports correct results because it does not deduce the final dependencies from those tuple comparisons.

Clearly, PYRO is superior in terms of performance. Nevertheless, no algorithm could profile the complete *Entity source* and *Census* datasets within the time limit. The next section shows the reason: These datasets entail tremendous dependency sets.

7.3 Scalability

Our next experiment analyzes the algorithms' scalability, i. e., how they compare as the profiled datasets grow in the number of tuples and columns. Note that we again exclude FDEP from these and following detail analyses and include both the non-parallel and parallel version of PYRO.

To determine the row scalability of the algorithms, we execute them on various samples of the four datasets with the most rows; Figures 5(a–d) depict the results. Because the number of dependencies is mainly stable across the samples, each algorithm tests more or less the same dependency candidates, regardless of the sample size. While all algorithms exhibit a somewhat linear scaling behavior, DUCC/DFD is usually up to 20 times faster than TANE, and PYRO is up to 15 times faster than DUCC/DFD. In both cases, the speed-up increases as the sample size increases, which is because on larger datasets the overhead, e. g., for initially loading the data and maintaining AS samples to avoid PLI-based error calculations, is more effectively redeemed.

In our column scalability experiments on the four datasets with the most columns, PYRO also shows the best scaling behavior, as can be seen in Figures 5(e–h): DUCC/DFD is up to 22 times faster than TANE and PYRO up to 12 times faster than DUCC/DFD– the more columns are considered, the greater is the speed-up. A particularly interesting observation is that all algorithms' runtime curves somewhat follow the number of dependencies. Such behavior is optimal, in the sense that any algorithm has to be at least of linear complexity in its output size; still the algorithms differ greatly in terms of absolute runtimes. Also, the number of columns that can be processed by each algorithm differs. TANE always fails first due to its high memory demand. In contrast, PYRO and DUCC/DFD fail only when they exceed the 1 h time limit of this experiment. They are less susceptible to run out of main memory because of their dynamic caches. Still, PYRO outperforms DUCC/DFD because of its high efficiency and, as a result, scales further.

Nonetheless, PYRO can only *mitigate*, but not *completely avoid* the effects of the exponential complexity of the dependency discovery problem, of course. Figure 6 breaks down PYRO's runtime from Figures 5(f–h), thereby showing that the growing number of dependencies requires PYRO to calculate more dependency errors, which dominates the runtime. Avoiding the exponential complexity altogether would require a relaxation of the problem itself, which is not the goal of this paper. However, the above experiments clearly demonstrate PYRO's improved scalability, thereby making it an excellent basis for relaxed algorithms.

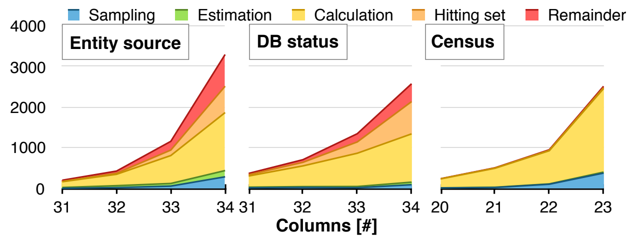

Figure 6: Runtime breakdown of Pyro ($e_{\max} = 0.01$).

7.4 Memory requirements

A particular concern of data profiling algorithms is their memory requirement, because computers vary greatly in the amount of available main memory. And after all, performance improvements might be just an advantage gained by using the available main memory more extensively. Thus, we compare PYRO's, DUCC/DFD's, and TANE's memory requirements. In detail, we execute all three algorithms with a maximum heap size of 32 MB and continuously double this value until the respective algorithm is able to process the given dataset. As can be seen in Figure 7, PYRO always requires the least amount of memory. In fact, we find PYRO to run out of memory only while loading the data. In contrast to TANE and DUCC/DFD, PYRO pins only very few larger data structures in main memory, while managing PLIs and AS samples in caches that adapt themselves to the amount of available main memory. This renders PYRO highly robust.

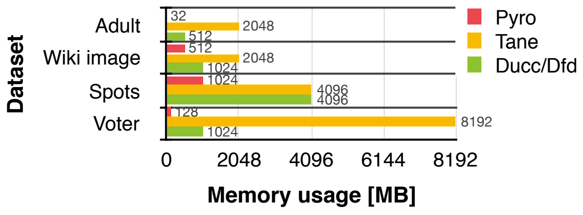

Figure 7: Memory requirements of Pyro, Tane, and Ducc/Dfd ($e_{\max} = 0.01$).

7.5 Sampling

One of PYRO's principal approaches is to save costly dependency error calculations by estimating the error of dependency candidates. Even though Theorem 1 describes the accuracy of PYRO's estimator depending on the AS sample size, it is unknown, which accuracy works best to find the minimal dependencies in real-world datasets. To investigate this matter, we execute PYRO with different AS sample sizes on hard-to-profile datasets and display the results in Figure 8.

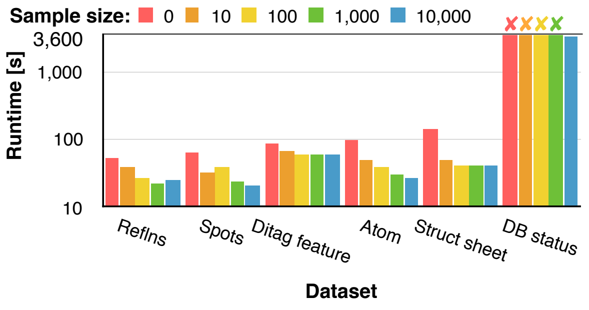

Figure 8: Comparison of Pyro's sampling strategies ($e_{\max} = 0.01$).

Apparently, not using AS samples (and thus error estimations) is always the worst option, while AS sample sizes of 1,000 and 10,000 work well on all tested datasets. That being said, using a tiny sample is consistently better than using no sampling at all to guide the search space traversal. For instance, on the *Atom* dataset, using a sample of only 10 agree sets reduces the number of dependency error calculations from 15,708 to 7,920. A sample size of 10,000 further reduces this value to 4,562. Note that the latter number leaves only little room for improvement: The *Atom* dataset has 1,582 minimal dependencies and about as many maximal non-dependencies, all of which need to be verified by an individual error calculation in the optimal case.

7.6 Ranking

Table 2 reveals that certain datasets entail an unwieldy number of approximate dependencies. While some use cases, such as query optimization and feature selection (see Section 1), can make use of all those dependencies regardless of their semantics, other use cases, such as data cleaning, require semantically meaningful dependencies. This raises the question whether users can be supported to quickly spot relevant dependencies.

For that purpose, we developed a simple ranking scheme for AFDs. For an AFD $X \to A$, our metric models the overlap in tuple pairs agreeing in either X or A with the hypergeometric distribution and measures how many standard deviations the actual overlap and the mean overlap are apart – let h denote this value. Intuitively, if h is large, X and A strongly correlate. Additionally, we consider the conditional probability that a tuple pair agrees in A given it agrees in X. This is similar to the confidence of association rules, thus let c denote this probability. Now, we can assign each AFD $c \cdot \operatorname{sgn} h \cdot \log |h|$ as ranking score, where sgn is the signum function.

We applied this ranking to the *Voter* dataset, for which we have column headers, and report the highest ranked AFDs: *(1+2)* The AFDs *voter_id→voter_reg_num* and *voter_reg_num →voter_id* uncover that both are equivalent voter identifications. Note that they are not keys, though. *(3)* The AFD *zip_code→city* reflects a rule of the data domain and lends itself to data cleaning. *(4)* The AFD *first_name→gender* also exposes a latent rule of the data domain, thereby supporting knowledge discovery. These examples demonstrate that it is possible to quickly draw relevant dependencies from large dependency sets. Note that, although often only few of the discovered dependencies are meaningful, we must still discover *all* of them: Unlike the error measures from Definition 1, our ranking criterion lacks the monotonicity required for pruning during the discovery process.

8. CONCLUDING REMARKS

We presented PYRO, an efficient and scalable algorithm to discover all AFDs and all AUCCs in a given dataset. Using a separate-and-conquer discovery strategy, PYRO quickly approaches minimal dependencies via samples of agree sets, efficiently verifies them, and effectively prunes the search spaces with the discovered dependencies. In addition, PYRO has a small memory footprint and benefits from parallel execution. In our evaluation with state-of-the-art dependency discovery algorithms, we found PYRO to be up to 33× more efficient than the best competitor. Parallelization regularly allows even greater speed-ups.

PYRO is formulated in an abstract manner that fits both the AUCC and AFD discovery problem. It is hence interesting to investigate whether PYRO can be used as a framework to solve other lattice-based problems efficiently, e. g., frequent itemset mining or hypergraph transversal calculation. We are also investigating how to incorporate semantic measures for pruning: Besides filtering spurious dependencies, considerably reduced result sizes might proportionally improve PYRO's performance.

Acknowledgements

This work was funded by the German Research Society (DFG fund FOR 1306).

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

[4] W. W. Armstrong. Dependency structures of data base relationships. In *IFIP Congress*, pages 580–583, 1974.

[5] J. Atoum. Mining approximate functional dependencies from databases based on minimal cover and equivalent classes. *European Journal of Scientific Research*, 33(2):338–346, 2009.

[6] C. Beeri, M. Dowd, R. Fagin, and R. Statman. On the structure of Armstrong relations for functional dependencies. *Journal of the ACM*, 31(1):30–46, 1984.

[7] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3(1-2):197–207, 2010.

[8] T. Bleifuß, S. Bülow, J. Frohnhofen, J. Risch, G. Wiese, S. Kruse, T. Papenbrock, and F. Naumann. Approximate discovery of functional dependencies for large datasets. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1803–1812, 2016.

[9] L. Caruccio, V. Deufemia, and G. Polese. Relaxed functional dependencies – a survey of approaches. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(1):147–165, 2016.

[10] G. Chandrashekar and F. Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014.

[11] P. A. Flach and I. Savnik. FDEP source code. `http://www.cs.bris.ac.uk/~flach/fdep/`.

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

[13] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC data-cleaning framework. *PVLDB*, 6(9):625–636, 2013.

[14] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1–12, 2000.

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

[16] 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.

[17] 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.

[18] R. M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations*, pages 85–103, 1972.

[19] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and S. Yin. Bigdansing: A system for big data cleansing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1215–1230, 2015.

[20] R. S. King and J. J. Legendre. Discovery of functional and approximate functional dependencies in relational databases. *Journal of Applied Mathematics and Decision Sciences*, 7(1):49–59, 2003.

[21] J. Kivinen and H. Mannila. Approximate dependency inference from relations. *Proceedings of the International Conference on Database Theory (ICDT)*, pages 86–98, 1992.

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

[23] H. Köhler, S. Link, and X. Zhou. Possible and certain SQL keys. *PVLDB*, 8(11):1118–1129, 2015.

[24] S. Kolahi and L. V. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 53–62, 2009.

[25] S. Kruse, D. Hahn, M. Walter, and F. Naumann. Metacrate: organize and analyze millions of data profiles. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, 2017.

[26] S. Kruse, T. Papenbrock, H. Harmouch, and F. Naumann. Data anamnesis: admitting raw data into an organization. *IEEE Data Engineering Bulletin*, 39(2):8–20, 2016.

[27] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[28] W. Li, Z. Li, Q. Chen, T. Jiang, and Z. Yin. Discovering approximate functional dependencies from distributed big data. In *Asia-Pacific Web Conference*, pages 289–301, 2016.

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

[30] 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.

[31] P. Mandros, M. Boley, and J. Vreeken. Discovering reliable approximate functional dependencies. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 355–363, 2017.

[32] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.

[33] V. Matos and B. Grasser. SQL-based discovery of exact and approximate functional dependencies. *ACM SIGCSE Bulletin*, 36(4):58–63, 2004.

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

[35] U. Nambiar and S. Kambhampati. Mining approximate functional dependencies and concept similarities to answer imprecise queries. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB)*, pages 73–78, 2004.

[36] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data profiling with Metanome. *PVLDB*, 8(12):1860–1863, 2015.

[37] 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. *PVLDB*, 8(10):1082–1093, 2015.

[38] 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.

[39] T. Papenbrock and F. Naumann. Data-driven schema normalization. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 342–353, 2017.

[40] D. Sánchez, J. M. Serrano, I. Blanco, M. J. Martín-Bautista, and M.-A. Vila. Using association rules to mine for strong approximate dependencies. *Data Mining and Knowledge Discovery*, 16(3):313–348, 2008.

[41] 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.

[42] S. Thirumuruganathan, L. Berti-Equille, M. Ouzzani, J. A. Quiané-Ruiz, and N. Tang. UGuide: user-guided discovery of FD-detectable errors. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2017.

[43] J. Zwiener. Quicker ways of doing fewer things: improved index structures and algorithms for data profiling. Master's thesis, Hasso Plattner Institute, University of Potsdam, 2015.