Extending Desbordante with Probabilistic Functional Dependency Discovery Support

Ilia Barutkin, Maxim Fofanov, Sergey Belokonny, Vladislav Makeev, George Chernishev
Saint-Petersburg University
Saint-Petersburg, Russia
{ilia.d.barutkin, max.fofanov, belokoniy, makeev.vladislav.d, chernishev}@gmail.com

Abstract—**Data profiling aims to extract complex patterns from data for further analysis and use that data in domains such as data cleaning, data deduplication, anomaly detection, and many more.**

Functional dependencies (FDs) are one of the most well-known patterns. However, they are poorly suited for these tasks, as real data is usually dirty, and the rigid definition of FDs does not allow algorithms to locate them. For this reason, there are several formulations aimed at relaxing FDs to support dirty data, with approximate functional dependency (AFD) being the most popular one. Another formulation is the Probabilistic Functional Dependency (pFD), which we aim to support inside Desbordante — a science-intensive, high-performance and open-source data profiling tool implemented in C++. However, pFDs are relatively poorly studied, compared to AFDs.

In this paper we study pFDs, both analytically and empirically. We start by assessing how different pFDs and AFDs are by studying cases in which pFDs have an edge over AFDs. Then, we implement the algorithm for pFD discovery, as well as study its run time and memory consumption. We also compare it with an AFD discovery algorithm. Lastly, we study the output of both algorithms to learn whether or not it is possible to use AFD discovery algorithm to get pFDs and vice versa.

I. Introduction

Currently, growing volumes of data pose a serious challenge to data analysts. Data, however, offers only a moderate value in and of itself, and it is instead the facts contained within that data that are of interest to analysts. The volumes of data in question far exceed the size that could be grasped by the human eye, so automatic approaches become more and more in demand.

Data profiling [1] aims to extract facts from data. There are two kinds of data profiling — naive and science-intensive. Naive approach concerns itself with simple statistics, such as the number of rows and columns, number of nulls in them, their mean and variance, etc. There are dozens of tools for this kind of profiling. On the other hand, science-intensive profiling aims to extract complex patterns represented by structures which we will refer to as primitives. Examples of such patterns are database dependencies (functional [2], inclusion [3]), association rules [4], algebraic constraints [5], inferred semantic data types [6], and others. Such patterns have many applications:

- for scientific data, they may indicate a presence of some regularity [7], which may promote the formulation of a hypothesis, which, in turn, may lead to a scientific discovery;
- for business data, it is possible [8] to use the discovered primitives for cleaning errors in data, finding inexact duplicates, performing schema matching, finding outliers, and solving many other problems;
- for machine learning, data primitives can help in feature engineering and in choosing the direction for the ablation study;
- for databases, they can help with validating and discovering various advanced integrity constraints.

Extracting and validating primitives is computationally expensive, which becomes a serious issue with the scaling of datasets. Therefore, it requires complex algorithms and efficient implementations. These are some of the major contributing factors as to why such kind of profiling is now a developing area and why science-intensive profilers are rare. Currently, there exist two science-intensive data profilers — Metanome [9] and Desbordante.

Desbordante (Spanish for *boundless*) [10] is a *science-intensive*, *high-performance* and *open-source* data profiling tool implemented in C++. To the best of our knowledge, Desbordante is currently the only profiler that possesses these three qualities. It is capable of discovering and validating many primitives, including functional dependencies (both exact and approximate), conditional functional dependencies, metric functional dependencies, and others. The full list can be found on the web-site [11].

One of the well-known primitives is the functional dependency, which states that if two records of the table are equal in attribute X, then they should be equal in attribute Y. The formal definition is given in Section II.

Primitives can be classified into three groups:

1) Exact by definition. These primitives define instances which hold over the whole dataset. Classic functional dependency is an example of the exact primitive.
2) Approximate by definition. In this case, approximate means that found instances hold over the whole dataset, but with some degree of error predefined by user at the start of the algorithm. Thus, there are records in the dataset that may not conform to the exact definition.
3) Approximate by discovery procedure. In this case approximate means that discovery algorithm returns primitive instances that may hold or may not hold. While such instances require verification, such approach may

be of use as it allows the speed up of discovery by up to an order of magnitude [12], [13].

In this paper, we will only consider dependencies that are approximate by definition, also called relaxed dependencies [14]. Such dependencies are of a particular interest for the end-users of science-intensive profilers. There is a simple reason for this: real-life data is always dirty — it contains inconsistencies, missing values, and other artifacts. Therefore, exact dependencies rarely hold on such data and discovery algorithm will not locate them.

For functional dependencies, there are several approximate variants that are built upon the family of g_1, g_2, g_3 metrics, proposed by J.Kivinen and H.Mannila in their seminal paper "Approximate inference of functional dependencies from relations" [15]. The most well-known variant is Approximate Functional Dependency (AFD) [16], which is based on an adaptation of the g_1 metric for defining maximum permissible error. One of the alternatives is the Probabilistic Functional Dependency [17], [18], which uses g_3.

We are considering the addition of pFD discovery functionality to Desbordante. Before this, it is necessary to evaluate pFDs, since they are significantly less studied that AFDs. At the same time, Desbordante supports discovery and validation of FDs and AFDs, so it is natural to compare them with each other. Our goal is twofold: firstly, it is essential to study how expensive in terms of run time and memory consumption pFDs are when compared to exact approaches. Secondly, it is also necessary to understand if pFD support provides value to the end-user. This includes answering questions "how different are pFDs from AFDs", and "how dependencies of both types that are returned by discovery algorithm relate to each other".

Overall, in our study we pose the following research questions (RQs):

RQ1 Are pFDs of interest to the end-user? How different are they from AFDs, what kind of FD violations are they more tolerant to? Does this definition allow for discovery of dependencies that could not be discovered by the AFD definition? And vice versa: how much AFDs are lost by it?

RQ2 How computationally expensive is the candidate validation procedure of pFD discovery algorithm compared to the AFD?

RQ3 How does maximum error threshold affect run time and memory consumption of the pFD discovery algorithm?

RQ4 What is the run time and memory expenses of pFD discovery, compared to AFD?

Overall, the contribution of the paper is the following:

- A discussion of pFDs, their comparison with AFDs.
- A survey of approximate primitives that are based on g_1, g_2, g_3 metrics, as it is the basis for the majority of existing approximate primitives, including AFDs and pFDs.
- An open-source C++ implementation of a pFD discovery algorithm, which — to the best of our knowledge — is the only one currently available.
- An empirical evaluation of the pFD discovery algorithm, and its comparison with the AFD one.

This paper is organized as follows. In Section II we formally present pFDs and AFDs. We compare them and discuss their differences, while providing examples. Next, in Section III we discuss related work concerning approximate dependencies based on g_1, g_2, g_3 metrics. In Section IV we describe the algorithm and discuss our modifications. We evaluate our implementation and compare pFD and AFD discovery in Section V. We conclude this paper with Section VI.

II. Background

Let us start with basic definitions imperative to understanding the paper's context.

A functional dependency [19] over a relation r with schema R is an expression denoted as $X \to A$, where $X \subseteq R$ and $A \in R$. We also denote set X as left-hand side (LHS), and the attribute A as right-hand side (RHS). The dependency is satisfied if, for all pairs of tuples $t, u \in r$, the following holds: if $\forall B \in X(t[B] = u[B])$, then $t[A] = u[A]$, or, equivalently, t and u agree on X and A. In this case, we also say that the functional dependency is correct or holds.

Let a certain relation r with schema R and a functional dependency $X \to Y$ over R be given. Then we assert that a pair (u, v) of tuples from r violates the dependency, or equivalently, is a violating pair, if $u[X] = v[X]$, but $u[Y] \neq v[Y]$. From this, it can be concluded that the dependency holds on the relation if the relation contains no violating pairs. A tuple u is termed violating if it is a part of a violating pair.

A relaxed functional dependency is a functional dependency that is almost satisfied. An example of this could be the relationship between columns "phone number" and "department", as several departments within company may share the same phone, albeit rarely. There are several ways to define the relaxation of functional dependency. The first one is the notion of Approximate Functional Dependency. In the original TANE paper [19] authors proposed to use the g_3 metric to define and discover AFDs, which is as follows:

$$g_3(X \to Y, r) = 1 - \frac{max\{|s| | s \subseteq r, s \models X \to Y\}}{|r|}$$

Almost two decades later S. Kruse and F. Naumann [16] developed PYRO — a novel AFD discovery algorithm. However, they have used a modified g_1 metric for their AFD definition, which is as follows:

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

Thus, currently there exist two algorithms for AFD discovery and two AFD definitions. Metrics which are used in these definitions can be put into both existing algorithms.

Desbordante has both TANE and PYRO implementations [20]. Having started our project, we have decided to stick to the modern definition, and thus in this paper we consider AFDs that are based on the modified g_1 metric. It is also worth mentioning that our implementation of TANE is a modified one, similarly to TANE implementation in the Metanome project [21].

The second relaxation approach is the Probabilistic Functional Dependency, which is defined as follows. Let R — a relation, X — a set of attributes, and A — an attribute in R. A probabilistic functional dependency [17] is denoted as $pFD : X \xrightarrow{p} A$, where p is the likelihood of the $X \to A$ being correct.

To define said probability, lets denote a set of unique not-null values of attributes of X as $D_X = \{t[X] \mid t \in R\}$, set of tuples with those values of X_1 attributes of X as $V_{X_1} = \{t \in R \mid t[X] = X_1\}$, and another set of tuples as $(V_Y, V_{X_1}) = \{t \in R \mid t[X] = X_1 \wedge t[Y] = \underset{Y_k \in r}{argmax}\{|V_{X_1} \cap V_{Y_k}|\}\}$. The probability of a dependency holding on a subset of tuples with value of attribute X equal to X_1 is therefore defined as $P(X \to Y, V_{X_1}) = \frac{|V_Y, V_{X_1}|}{|V_{X_1}|}$.

Finally, the probability of a functional dependency between attributes X and Y in R is defined via two formulas, namely PerValue and PerTuple:

$$P_{PerValue}(X \to Y, R) = \frac{\sum_{V_X \in D_X} P(X \to Y, V_X)}{|D_X|}$$

$$P_{PerTuple}(X \to Y, R) = \sum_{V_X \in D_X} \frac{|V_X|}{|R|} P(X \to Y, V_X)$$

It is evident PerValue is an average of the probabilities of a dependency being correct for each distinct value of X, whereas PerTuple metric accounts for the frequency of values of X amongst all tuples in a relation.

We also say that a pFD $\overline{X} \xrightarrow{p} Y$ is minimal if, for any proper subset $X' \subset \overline{X}$, $X' \xrightarrow{p} Y$ does not hold. A pFD is called trivial, if $Y \in \overline{X}$.

Note that it is possible to add an attribute to LHS of a pFD, and the resulting dependency will remain a pFD, if PerTuple metric is used. The same holds true for AFDs and their g_1 metric. However, this is not always true in case of pFD PerValue.

Finally, it is evident that PerTuple metric is the same as g_3, which is defined using notion of probability:

$$P_{PerTuple}(X \to Y, R) = 1 - g_3(X \to Y, R)$$

III. Related Work

In the world of relaxed dependencies, there are three major metrics used for defining how well a given relaxed dependency holds on a particular dataset. They are called g_1, g_2, g_3 and were proposed by J.Kivinen and H.Mannila in "Approximate inference of functional dependencies from relations" [15]. Despite the fact that the original paper considers relaxed functional dependencies, the concept is easily generalized owing to the flexibility of the provided definitions. As the result, these metrics gave rise to many other types of relaxed dependencies which we are going to survey in this paper.

A. g_1 and g_2 metrics

Let G_1 be defined as the number of violating pairs for the dependency $X \to Y$ in the relation r:

$$G_1(X \to Y, r) = |\{(u, v) | u, v \in r, \\ u[X] = v[X] \wedge u[Y] \neq v[Y]\}|.$$

Then, the metric g_1 represents a normalized version of G_1.

$$g_1(X \to Y, r) = G_1(X \to Y, r)/|r|^2$$

G_2 is the number of violating tuples for the dependency $X \to Y$ in the relation r.

$$G_2(X \to Y, r) = |\{u | u \in r, \\ \exists v \in r : u[X] = v[X] \wedge u[Y] \neq v[Y]\}|$$

The metric g_2, in turn, represents a normalized version of G_2.

$$g_2(X \to Y, r) = G_2(X \to Y, r)/|r|$$

The g_1 and g_2 metrics, as shown in the previously mentioned paper [15], are applied for defining approximate functional dependencies. However, due to their poorer generalizability and greater computational complexity, they are not as widely used as g_3 [22].

TABLE I. Example of g_1, g_2 and g_3

X	Y
a	1
b	2
a	3
c	3
d	4

Consider an example presented in Table I. In case of g_1 its value is calculated as follows. Since there is only a single violating tuple — $((a, 1), (a, 3))$, we get:

$$g_1(X \to Y, r) = \frac{1}{5^2} = 0.04.$$

For g_2, there are two values with different right-hand sides: $(a, 1)$ and $(a, 3)$, hence the value of the metric g_2 being:

$$g_2(X \to Y, r) = \frac{2}{5} = 0.4.$$

Now, let us consider various relaxed dependencies that are based on either g_1 or g_2.

1. Approximate functional dependencies. We have discussed the notion of AFDs in the Background section. An AFD example is presented in Table II.

PYRO [16] is an algorithm for discovery of AFDs that are based on the modern definition. In this algorithm, an adaptation of the g_1 metric, referred to by the authors of the article as e defined in equation II is employed.

PYRO demonstrates excellent performance due to employing several interesting optimizations, one of them being the error calculation approach.

TABLE II. AFD EXAMPLE: POSITION → SALARY (g_3 = 0.16).

ID	Position	Salary
1	Programmer	3000
2	Designer	2800
3	Programmer	3200
4	Manager	3000
5	Designer	2800
6	Programmer	3000

Let r be a relation with schema R and $X \subseteq R$ be a set of attributes. A cluster is defined as the set of all tuple indices from r that have identical values for X, or $c(t) = \{i | t_i[X] = t[X]\}$. The PLI for X is all such sets, excluding singleton clusters:

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

The size of the resultant index is denoted as $||\overline{\pi}(X)|| = \sum_{c \in \overline{\pi}(X)} |c|$.

Hence, the calculation of the error metric e is as follows: tuple pairs that agree on X and disagree on A (for the candidate $X \to A$) are considered violating pairs, which need to be counted. However, instead of counting them directly, PYRO employs a more efficient method. For each cluster $\overline{\pi}(X)$, the number of tuple pairs that also agree on A is calculated, and this result is then subtracted from the total number of tuple pairs in the cluster. This is achieved through v_A, a vector in which information about the content of the cluster is recorded in one-hot-encoding format. Summing up the errors for each cluster yields the final error. The pseudocode for this algorithm is presented in Listing 1.

Algorithm 1 Calculation of e for AFD using PYRO

```
Require: Set of tuples $\overline{\pi}(X)$, values of attribute $A$ as $v_A$
Ensure: Metric $e$ for AFD
  $e \leftarrow 0$
  for each cluster $c \in \overline{\pi}(X)$ do
    $counter \leftarrow$ dictionary with default value 0
    for each item $i \in c$ do
      for $v_A[i] \neq 0$ do
        $counter[v_A[i]] \leftarrow counter[v_A[i]] + 1$
      end for
    end for
    $e \leftarrow e + |c|^2 - |c| - \sum_{A \in counter} counter[A]^2 - counter[A]$
  end for
  return $e$
```

2. Approximate unique column combinations. Approximate Unique Column Combinations (AUCCs) represent another type of relaxed dependency that can be discovered using the PYRO algorithm.

Let r be a relation with schema R and attribute sets $X, Y \subseteq R$. According to [16], X is a Unique Column Combination (UCC) if, for all tuple pairs $t_1, t_2 \in r$, from $t_1[X] \neq t_2[X]$ it follows that $t_1[Y] = t_2[Y]$.

The error metric for AUCC is defined as follows:

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

For Approximate UCC, unlike AFD, the error calculation for $\overline{\pi}(X)$ is trivial. This happens because all tuple pairs within each cluster are the violating tuples themselves.

Algorithm 2 Calculation of e for AUCC using PYRO

```
Require: Set of tuples $\overline{\pi}(X)$, total number of tuples $|r|$
Ensure: Metric $e$ for AUCC
  $e \leftarrow \sum_{c \in \overline{\pi}(X)} \frac{|c|^2 - |c|}{|r|^2 - |r|}$
  return $e$
```

Approximate Unique Column Combinations are utilized in tasks such as data cleaning, database normalization, and query optimization.

3. Denial Constraints. Denial constraint (DC) is a type of an integrity constraint used in databases to ensure data quality. DC describes conditions that must not occur within the database. For example, it might state that two rows in a table cannot have certain value combinations. If an insertion or an update of a row violates a DC, the operation is generally aborted.

Approximate denial constraints in databases are a form of constraint that permits a degree of flexibility or exceptions. Unlike exact denial constraints that rigorously prohibit certain data value combinations, approximate denial constraints allow for a limited number of violations.

In this case, the g_1 metric is utilized for calculating the error measure [23].

DCs and their approximate variants are essential for upholding data consistency and reliability within a database, as they avert the introduction of invalid or conflicting information.

B. g_3 metric

Let G_3 represent the number of tuples for the dependency $X \to Y$ within the relation r that must be removed to establish an exact dependency. Formally:

$$G_3(X \to Y, r) = |r| - max\{|s| : s \subset r, s \models X \to Y\}$$

$$g_3(X \to Y, r) = G_3(X \to Y, r)/|r|$$

The g_3 metric is acknowledged as an industry standard and is applied in the context of various approximate dependencies: Approximate Functional Dependencies, Approximate Inclusion Dependencies, Probabilistic Functional Dependencies.

Its calculation is as follows. For the example presented in table I:

$$g_3(X \to Y, r) = \frac{5 - 4}{5},$$

this is because it is sufficient to remove one tuple for the "exact" dependency to be satisfied. This example illustrates the practical utility of the g_3 in assessing the degree of violation of a dependency within a dataset. Now, let us consider various relaxed dependencies that are based on this metric.

1. Approximate functional dependencies. Despite the fact that modern AFD discovery papers utilize the g_1 metric, the initial paper proposing the AFD concept employed g_3. This paper also proposed the TANE algorithm [19], designed for mining exact functional dependencies which can also be modified for mining approximate functional dependencies. The metric was defined as follows:

$$e(X \to A) = min\{\frac{|s|}{|r|} : s \subset r \text{ and } X \to A \text{ holds in } r \setminus s\}$$

Another algorithm [24] utilizing the g_3 metric is called $DiM\varepsilon$. This highly-optimized algorithm employs a level-wise approach in candidate generation, starting with singleton sets at level zero. Additionally, the authors claim that the algorithm can be adapted for use with other metrics and even different types of dependencies.

Functional and approximate functional dependencies are instrumental in database normalization, data cleaning, and also aid analysts in uncovering hidden trends within data. Their implementation and optimization in algorithms like TANE and $DiM\varepsilon$ highlight their significance in managing and analyzing large data sets efficiently.

2. Approximate inclusion dependencies. An Inclusion Dependency [16] (IND) over a schema R is a statement of the form $R_i[X] \subseteq R_j[Y]$, $R_i, R_j \in R$, $X \subseteq R_i$, $Y \subseteq R_j$. The size (or arity) of such a dependency is denoted as $i = R[X] \subseteq R[Y]$, where $|i| = |X| = |Y|$. Inclusion dependencies of size one are commonly referred to as unary inclusion dependencies.

An inclusion dependency is satisfied if all values from the left side are present in the right side. To assess the degree of approximation, a variant of the g_3 metric, denoted as g_3', is used. This version is adapted for inclusion dependencies and conveys essentially the same meaning.

Despite the lack of separate algorithms for detecting approximate inclusion dependencies, several algorithms for finding "exact" dependencies have been adapted for this task, such as MIND [25], Spider [26], or S-indd [27].

MIND employs a level-wise approach, where candidates of size $i+1$ are generated from already discovered dependencies of size i. In the case of approximate dependencies, during the candidate validation stage, approximate dependencies that meet a user-defined threshold for g_3' are also considered.

The primary application area for both "exact" and approximate inclusion dependencies is in the identification of foreign keys in databases [28]. This is crucial for database design, integrity, and normalization processes, facilitating effective data management and interrelation of different data sets within a database system. An example of AIND is presented in Table III.

3. Graph Entity Dependencies. A Graph Entity Dependency (GED) is a constraint within a property graph G, expressed as a pair $\phi = (Q[\bar{u}], X \to Y)$. It states that for any instance of the pattern in the graph $Q[\bar{u}]$ within G, the

TABLE III. AIND EXAMPLE: USER EMAIL $\to$ REGISTERED EMAIL ($\varepsilon = 0.34$)

TID	User Email	ID	Registered Email
T001	example@email.com	C123	example@email.com
T002	sample@email.com	C124	sample@email.com
T003	missing@email.com	C125	-

dependency $X \to Y$ must be upheld. This denotes that if specific conditions defined by X are met within a pattern instance, then other conditions outlined by Y must also be satisfied. The metric g_3 is employed in its original form as a measure of approximation for GED [29].

Graph Entity Dependencies are employed for several key objectives within the realm of graph databases and data management. They ensure data integrity and consistency and aid in the optimization of complex queries.

4. Approximate Interval-based Temporal Dependencies. Approximate Interval-based Temporal Functional Dependencies (AITFDs) [30], [31] are a type of constraint in temporal databases. They extend the concept of functional dependencies to consider the temporal aspect of data, specifically focusing on time intervals. They use g_3 as a metric of approximation as follows.

Let X and Y be sets of atemporal attributes of a temporal relation schema $R = R(U, B, E)$, an Allen's Interval relation and ϵ a real number $0 \leq \epsilon \leq 1$. An instance r of R satisfies an ITFD $X \to Y$ with approximation ϵ if there exists a subset $r' \subseteq r$ for which $r \setminus r' \models X \to Y$ and $|r'| \leq \epsilon \cdot |r|$.

AITFDs are used for maintaining data integrity in temporal databases by ensuring that relationships among data attributes adhere to specified patterns over time. They are particularly useful for analyzing historical data, identifying trends, and predicting future values by understanding the temporal dynamics of data relationships.

Wrap-up. Concluding this section, we can state that, to the best of our knowledge, there were no studies where comparison between pFDs and AFDs was performed. The reasons for this are the following:

1) Both notions were developed long before the era of data profiling began.
2) Each notion was developed by a different research group and for a particular task.
3) The notions were assessed by its applicability to this particular task only, or no comparisons were performed at all.

Currently, data profiling is gaining traction, and it is imperative to catalogue all available tools. Thus, it is essential to compare pFDs and AFDs with each other.

IV. ALGORITHMS AND IMPLEMENTATION

This paper considers an implementation of pFDTane algorithm designed to discover minimal non-trivial probabilistic functional dependencies.

The new algorithm is based on TANE [19], which is a graph-traversing algorithm in which a graph — called lattice — is comprised of vertices representing all possible sets of attributes and edges connecting nodes of a form X and XA, where X — set of vertices, and A — another attribute. This way every edge represents a functional dependency $X \to A$. The algorithm consecutively checks for the existence of functional dependencies between neighboring levels of lattice, excluding vertices whenever possible.

Integration. In Desbordante, FDs discovery algorithms are implemented by inheriting FDAlgorithm or its subclasses and overriding ExecuteInternal method. Tane and PFDTane shown on the diagram in Figure 1 inherit PliBasedAlgorithm, in which relation loading method is additionally overridden. PositionListIndex (PLI) is a useful data structure comprised of stripped partitions [19]. This means that the structure contains a set of equivalence classes, built with respect to the equality of attribute values. Stripped means that classes containing a single attribute are dropped to reduce memory consumption. In Desbordante, this set is represented by a double-ended queue — namely, std::deque. More specifically, inheritance and related classes are shown on Fig. 1.

Algorithm 3 Calculation of PerValue [17] metric

```
Require: Relation $R$, attributes X and A
Ensure: Metric PerValue for $X \to A$
  $c \leftarrow t_1(X); |\pi(X)| \leftarrow 1; count(c) \leftarrow 0$
  $c' \leftarrow t_1(X, A); count(c') \leftarrow 0; maxCount(c) \leftarrow 0$
  $sum \leftarrow 0$
  for each $t \in R$ do
    if $t(X) == c$ then
      $count(c) \leftarrow count(c) + 1$
      if $t(X, A) == c'$ then
        $count(c') \leftarrow count(c') + 1$
      else
        if $maxCount(c) < count(c')$ then
          $maxCount(c) \leftarrow count(c')$
        end if
        $c' \leftarrow t(X, A); count(c') \leftarrow 0$
      end if
    else
      $sum \leftarrow sum + maxCount(c)/count(c)$
      $c \leftarrow t(X); |\pi(X)| \leftarrow |\pi(X)| + 1$
      $count(c) \leftarrow 0; maxCount(c) \leftarrow 0$
    end if
  end for
  return $sum/|\pi(X)|$
```

The class PFDTane uses LatticeLevel and LatticeVertex data structures, which contain the level and vertex information respectively. PFDTane generates lattice levels and handle its life time, so there an aggregation dependency with LatticeLevel is shown. Meanwhile ExecuteInternal method uses LatticeVertex and PLI, LatticeLevel consists of instances of LatticeVertex and each PLI instance corresponds to LatticeVertex, which is shown as composition on the diagram.

Algorithm 4 Calculation of PerTuple metric

```
Require: Relation $R$, attributes X and A
Ensure: Metric PerTuple for $X \to A$
  $c \leftarrow t_1(X); count(c) \leftarrow 0$
  $c' \leftarrow t_1(X, A); count(c') \leftarrow 0; maxCount(c) \leftarrow 0$
  $sum \leftarrow 0$
  for each $t \in R$ do
    if $t(X) == c$ then
      $count(c) \leftarrow count(c) + 1$
      if $t(X, A) == c'$ then
        $count(c') \leftarrow count(c') + 1$
      else
        if $maxCount(c) < count(c')$ then
          $maxCount(c) \leftarrow count(c')$
        end if
        $c' \leftarrow t(X, A); count(c') \leftarrow 0$
      end if
    else
      $sum \leftarrow sum + maxCount(c)$
      $c \leftarrow t(X); |\pi(X)| \leftarrow |\pi(X)| + 1$
      $count(c) \leftarrow 0; maxCount(c) \leftarrow 0$
    end if
  end for
  return $sum/|R|$
```

Candidate Validation. Error measurement functions used for candidate validation is the essentially only part which had to be changed in order to adapt the existing TANE implementation for pFD discovery. The functions implements the algorithms presented in listings 3 and 4. Implemented functions for non-zero FDs take PLI of LHS attributes of dependency and PLI of a union of LHS attributes and RHS attribute as arguments. Sorting performed in the first lines of code in Listings 3 and 4 is done on the latter argument, i.e. union of PLIs. Thus, it is then possible to iterate over PLI clusters, calculating probability in linear time. Because of using PLI, the algorithm does not iterate over single value clusters, which has positive impact on the algorithms run time.

V. Evaluation and Discussion

A. Methodology and Experimental Setup

Methodology. In order to answer research questions posed in the introduction, we have decided to perform quantitative and qualitative studies. For the former, we are going to analyze pFDs using examples and conduct an extensive literature review. For the latter, we are going to run a series of experiments, featuring AFD, pFD PerTuple, and pFD PerValue discovery algorithms. All these algorithms were implemented in Desbordante, and, more specifically, we used our TANE implementation. It is necessary to mention that, similarly to Metanome in Desbordante, TANE algorithm can cause a larger search space than necessary. However, due to the specifics of implementation, this functionality incurs almost negligible RAM overhead. Turning to performance, we want to stress the fact that this also does not negatively affects our study,

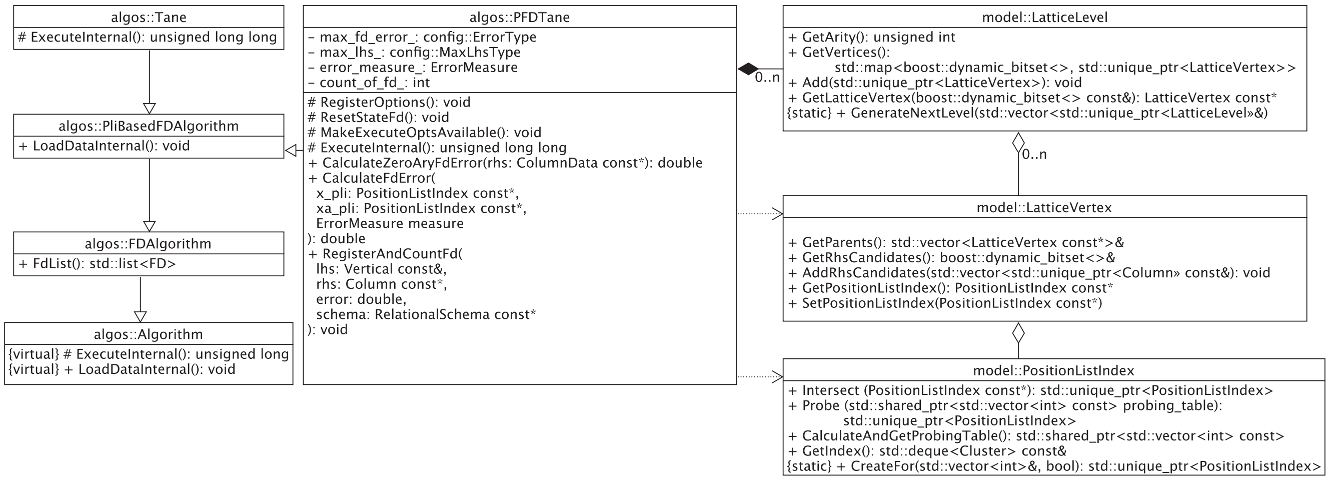

algos::Tane
# ExecuteInternal(): unsigned long long
algos::PliBasedFDAlgorithm
+ LoadDataInternal(): void
algos::FDAlgorithm
+ FdList(): std::list<FD>
algos::Algorithm
{virtual} # ExecuteInternal(): unsigned long
{virtual} + LoadDataInternal(): void
algos::PFDTane
– max_fd_error_: config::ErrorType
– max_lhs_: config::MaxLhsType
– error_measure_: ErrorMeasure
– count_of_fd_: int
# RegisterOptions(): void
# ResetStateFd(): void
# MakeExecuteOptsAvailable(): void
# ExecuteInternal(): unsigned long long
+ CalculateZeroAryFdError(rhs: ColumnData const*): double
+ CalculateFdError(x_pli: PositionListIndex const*, xa_pli: PositionListIndex const*, ErrorMeasure measure): double
+ RegisterAndCountFd(lhs: Vertical const&, rhs: Column const*, error: double, schema: RelationalSchema const*): void
0..n
model::LatticeLevel
+ GetArity(): unsigned int
+ GetVertices(): std::map<boost::dynamic_bitset<>, std::unique_ptr<LatticeVertex>>
+ Add(std::unique_ptr<LatticeVertex>): void
+ GetLatticeVertex(boost::dynamic_bitset<> const&): LatticeVertex const*
{static} + GenerateNextLevel(std::vector<std::unique_ptr<LatticeLevel»&)
0..n
model::LatticeVertex
+ GetParents(): std::vector<LatticeVertex const*>&
+ GetRhsCandidates(): boost::dynamic_bitset<>&
+ AddRhsCandidates(std::vector<std::unique_ptr<Column» const&): void
+ GetPositionListIndex(): PositionListIndex const*
+ SetPositionListIndex(PositionListIndex const*)
model::PositionListIndex
+ Intersect (PositionListIndex const*): std::unique_ptr<PositionListIndex>
+ Probe (std::shared_ptr<std::vector<int> const> probing_table): std::unique_ptr<PositionListIndex>
+ CalculateAndGetProbingTable(): std::shared_ptr<std::vector<int> const>
+ GetIndex(): std::deque<Cluster> const&
{static} + CreateFor(std::vector<int>&, bool): std::unique_ptr<PositionListIndex>

Fig. 1. UML class diagram

TABLE IV.
EXAMPLE CASE 1

X	Y
0	1
0	2
0	3
0	4
0	5
...	...
1	1
2	2
3	3
4	4
5	5
6	6

TABLE V.
EXAMPLE CASE 2

X	Y
0	1
0	1
0	1
0	1
0	1
1	1
1	2
2	3
2	4
3	5
3	6

since all experiments either compare methods relatively, or they compare algorithm output.

For pFD PerValue and pFD PerTuple algorithms, each dataset have been run with error thresholds ranging from 0 to 1 (inclusively) with an increment of 0.025. For TANE, datasets have been run with error thresholds located in captions of Tables IX–XII. This subset was selected due to the fact that error values close to zero are of more value to user and are expected to be used much more frequently.

A total of 10 iterations per error threshold value had been run, after which the average query execution time and maximum memory usage were calculated with confidence interval of 95%. Due to large confidence intervals for the run time of measures_v2.csv, an additional set of 20 iterations have been performed for that specific dataset in order to get a more accurate data.

Datasets. To perform experimental evaluation, we used datasets presented in Table VIII. Links to the datasets are available in the GitHub repository [32]. To perform comprehensive evaluation, we tried to select a collection of datasets with different properties. In this table we list the number of rows and attributes, file size and file source, as well as the number of minimal non-trivial FDs, AFDs, pFDs (both PerTuple and PerValue). The AFDs and pFDs were calculated with error threshold set to 0.01.

We have divided these datasets into two groups, which we present separately in two distinct figures. The reason for this is the dataset size difference, which will make them poorly readable if we put them in the same figure.

Experimental setup. Experiments were performed using the following hardware and software configuration. Hardware: AMD® Ryzen 5 7600X CPU @ 5.453GHz (6 cores), 32GB RAM. Software: Ubuntu 22.04 LTS, Kernel 6.5.0-15-generic (64-bit).

B. RQ1: Are pFDs of interest to the end-user? How much of a difference there is between pFDs and AFDs, and what kind of FD violations are pFDs more tolerant to? Does this definition allow for discovery of dependencies that could not be discovered by the AFD definition? And vice versa: how much AFDs are lost by it?

Let us start with the qualitative comparison of AFDs and pFDs.

Observation 1. First, lets consider a simplified dataset R presented in Table IV and $X \to Y$ dependency.

pFD with PerValue metric is not affected by the frequency of X. Indeed, consider $|V_0| \to \infty$. In this case $P_{PerValue}(X \to Y, R)$ tends to $\frac{6}{7}$ and its respective error $1 - P_{PerValue}(X \to Y, R)$ tends to $\frac{1}{7}$. At the same time, $g_3(X \to Y, R)$ and $e(X \to Y, R)$ tends to 1.

Thus, this metric can account for "faulty" LHS, if there are not too much of them. It allows having a lot of violating records if they correspond to relatively few distinct LHS values. For example, such "local" error may arise if a single sensor of an overall healthy set started to report faulty data.

Observation 2. Now, consider data presented in Table V and the same dependency.

The dependency is less likely to hold with the PerValue metric: $P_{PerValue}(X \to Y, R) = \frac{5}{8} = 0.625$, and $1 - P_{PerValue}(X \to Y, R) = 0.375$. At the same time $g_3(X \to Y, R) = \frac{3}{11} = 0.27$, and $e(X \to Y, R) = \frac{3}{55} = 0.05$.

Thus, pFD's PerValue will report larger error when there are a lot of individual LHS where dependency does not hold. It will ignore the positive contribution of "0", regardless of their number.

For a data scientist who explores data, such behavior may be undesirable and lead to valuable facts being missed. Suppose that there are one million of those "0" in this table, and the other six entries stay the same. This table will result in PerValue error of 0.375, which is rather large and therefore the pattern described by this pFD can be ignored. However, a more probable interpretation is the following: one million records are correct (since there is one million of them) and these six values are anomalies which should be deleted. At the same time, such interpretation can be located using AFDs.

FD guessing problem. pFDs were extensively used for problems where the goal was to "guess" FDs [17], [33], [34] from low-quality data. In these studies true FDs (gold standard) were known beforehand and had to be discovered using pFDs. Authors who originally proposed the pFD concept have performed experiments [17] which have shown that PerTuple tends to yield better results than PerValue in finding correct dependencies in cases where data is of a lower quality (due to noise).

However, their subsequent experiments [18] with an improved version of TANE that uses transitivity rule have had PerValue outperforming PerTuple in the majority of cases. The authors measured recall, precision, and F-measure using a gold-standard collection.

Recently, the PerValue metric demonstrated [33], [34] better results for the problem of FD discovery in datasets containing missing values.

AFDs vs pFDs, quantatively. The above-mentioned studies have not considered AFDs and their difference from pFDs. In our qualitative study we have demonstrated cases where pFDs can be of use and where they are inferior to AFDs. Now, let us turn to quantative part, which aims to answer the rest of the RQ1: "Does this definition allow for discovery of dependencies that could not be discovered by the AFD definition? And vice versa: how much AFDs are lost by it?".

Table VI contains the results of a search for three different dependency types in the monkeypox.csv dataset: AFDs, pFDs with PerValue, and pFDs with PerTuple. The table shows that AFD fails to find some pFDs when run with certain error thresholds, despite the dataset containing a comparable number of minimal non-trivial AFDs.

Though the minimal sets indeed differ, it doesn't immediately imply that the complete sets of pFDs and AFDs do. For example, for a fixed threshold, you may have found the following minimal dependencies: $pfd_1 : XZ \to A$, $pfd_2 : XY \to A$ and $afd_1 : X \to A$. But afd_1 infers all other AFDs that have X in LHS. That implies pfd_1, pfd_2 are in set of *all* AFDs. In order to highlight the essential difference

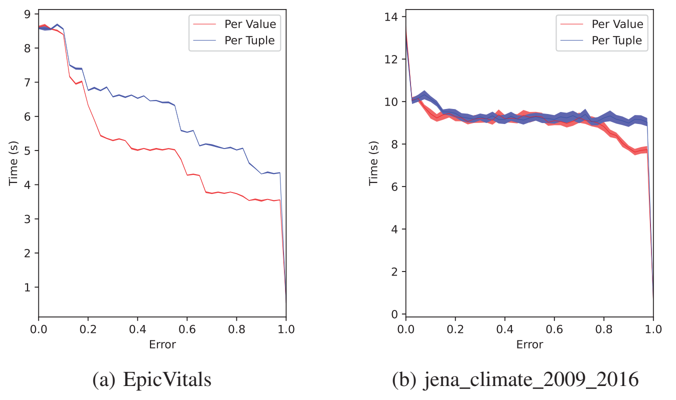


Fig. 2. pFDTane running times

of pFDs, we have also included in the table the number of minimal pFDs that are neither in the set of minimal AFDs nor inferable from it.

Concluding this RQ, we can say that pFDs have their own strengths, and that they are different from AFDs. Specifically, having fixed error threshold, pFDs are not a mere subset of AFDs, nor are AFDs a subset of pFDs in general. Finally, existing studies have demonstrated that pFDs have found applications for the FD guessing problem. However, in those studies comparison with AFDs was not performed, and it is outside of scope of this paper.

C. RQ2: How computationally expensive is the candidate validation procedure of pFD discovery algorithm compared to the AFD?

To compare the run time and memory consumption of validation functions of pFDTane and AFDTane, the corresponding algorithms had been run with error set to 0. This setting guarantees that all algorithms traverse the same part of the lattice and they all are on the level playing field. The results presented in Table VII showcase the fact that both PerValue and PerTuple prove to work slower than the validation with g_1. On the other hand, PerValue and PerTuple do not demonstrate a significant difference in either run time or memory consumption.

We can also note that almost all datasets have had less memory consumed by pFDTane when compared to AFDTane. The exception — which was the SEA.csv dataset — used approximately the same amount of memory.

D. RQ3: How does maximum error threshold affect run time and memory consumption of the pFD discovery algorithm?

The pFDTane algorithm have been run with various error thresholds ranging from 0 to 1 on eight different datasets depicted in Table VIII. Figure 2a and Figure 2b show two different patterns of behaviours of pFDTane. When it's supplied with the jena_climate_2009_2016.csv dataset, run time does not demonstrate a noteworthy difference past the 0.25 error threshold. Contrary to that, when run on the EpicVitals.csv dataset, the algorithm gets progressively faster as the error

TABLE VI. MINIMAL NON-TRIVIAL PFDS AND AFDS FOUND IN MONKEYPOX.CSV

	Total			$\|pFDs \setminus AFDs\|$		Non-inferable pFDs		$\|AFDs \setminus pFDs\|$		$\|pFDs \cap AFDs\|$	
Error	AFD	pFD PerValue	pFD PerTuple	PerValue	PerTuple	PerValue	PerTuple	PerValue	PerTuple	PerValue	PerTuple
0.01	126	142	134	133	124	3	1	117	116	9	10
0.05	73	69	71	62	64	2	2	66	66	7	7
0.1	55	81	64	72	55	2	0	46	46	9	9
0.2	69	168	70	153	60	29	0	54	59	15	10
0.3	63	70	51	61	42	30	2	54	54	9	9

TABLE VII. EXACT FDS DISCOVERY TIME AND MEMORY

Datasets	Time (s)			Memory (MB)		
	pFDTane per_value	pFDTane per_tuple	AFDTane	pFDTane per_value	pFDTane per_tuple	AFDTane
BKB_WaterQualityData_2020084	2.013	2.018	0.935	216	216	260
EpicVitals	8.609	8.583	4.266	807	807	1490
jena_climate_2009_2016	12.711	12.720	6.205	530	530	1490
measures_v2	16.245	16.334	15.278	1758	1758	1785
nuclear_explosions	2.198	2.203	0.795	189	189	1490
parking_citations	25.908	25.953	7.360	1381	1381	1491
SEA	1.963	1.956	1.193	279	279	270
games	3.207	3.222	1.492	399	399	1490

TABLE VIII. DATASETS USED FOR EXPERIMENTS

Dataset	Rows	Attributes	Size	Source	pFD PT count	pFD PV count	AFD count
EpicVitals.csv	1246303	7	33MB	EPF	10	13	21
BKB_WaterQualityData_2020084.csv	2370	17	180KB	U.S. FWS	3389	3712	901
games.csv	20058	16	7.67MB	kaggle	2264	1810	266
jena_climate_2009_2016.csv	420550	15	43.16MB	kaggle	3003	3148	210
measures_v2.csv	1330816	13	300.06MB	kaggle	642	573	144
nuclear_explosions.csv	2046	16	220KB	tidytudesday repository	2795	3619	1459
parking_citations.csv	95433	13	10MB	norfolk opendata	224	269	565
SEA.csv	1000000	4	33MB	openml.com	3	3	9
monkeypox.csv	5875	14	516KB	who.int	134	142	126

threshold increases. In our experiments six out of eight datasets behaved similarly to EpicVitals.

The number of steps in Tane is determined by the number of vertices in the lattice. However, the algorithm discards some vertices during its execution due to the nature of the task of searching for the minimal functional dependencies. Thus, the observed trend could be explained by the difference between 0.2 and 0.8 error threshold not generating any new dependencies, which would subsequently lead to an inability to discard additional vertices in the lattice.

Finally, as could be observed from Figure 3, PerValue yields better results in terms of run time on every dataset but SEA.csv when compared to pFDs with PerTuple.

E. RQ4: What is the run time and memory expenses of pFD discovery, compared to AFD?

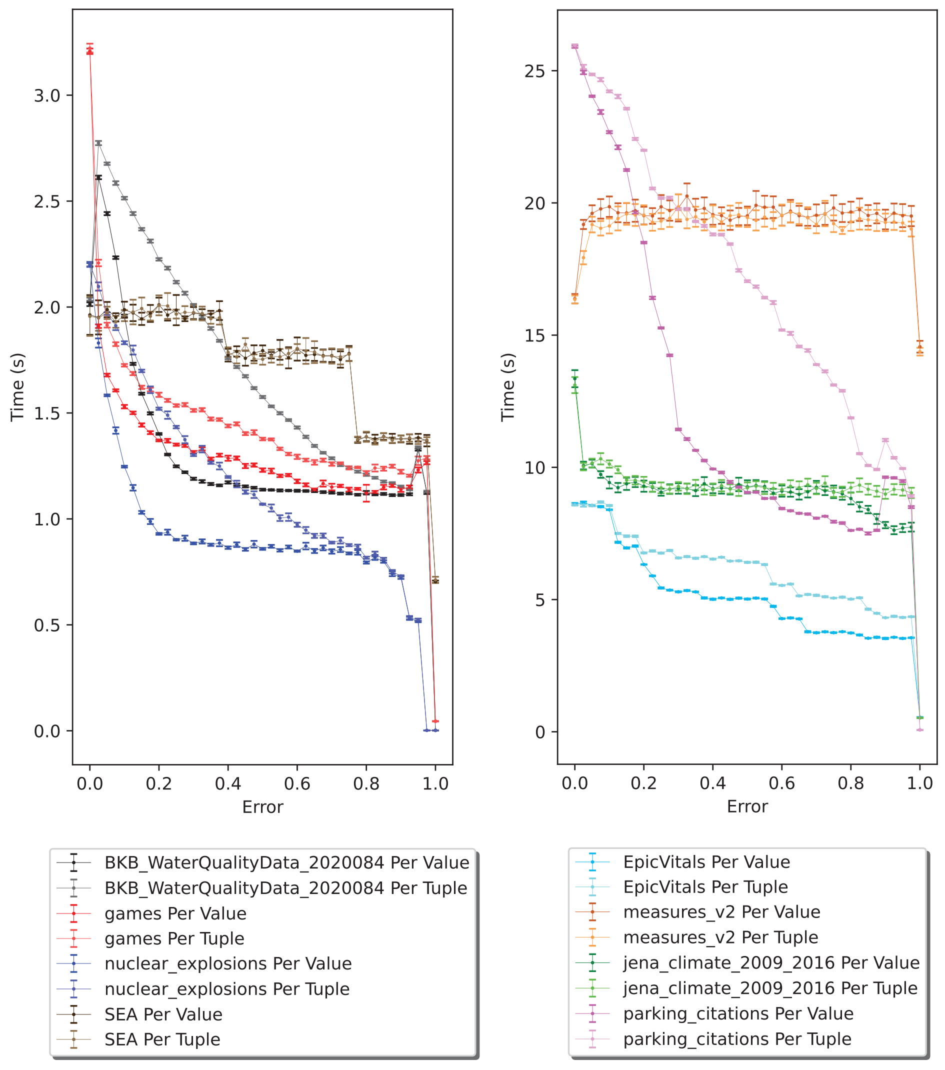
To compare pFDTane and AFDTane algorithms in the probabilistic and approximate dependency discovery tasks, the algorithms' implementations have been tested with different error thresholds. The results are presented in Table IX, Table X for the run time and in Table XI and Table XII for the memory consumption respectively. Each cell contains the ratio of pFDTane to AFDTane respective measurements. For the ease of understanding we have plotted the maximum amount memory consumed graph in Figure 4.

Almost all of the datasets depict AFDTane as a faster discovery algorithm when compared to pFDTane, with an exception of measures_v2.csv. The results suggest a decrease in performance difference with the error threshold exceeding 0.1.

Memory consumption have been observed to be lower for pFDTane on all datasets but SEA.csv. In contrast to the run time metric, the memory consumption seems to be equal for pFDTane and AFDTane for each of error threshold values.

VI. CONCLUSION

We started with qualitative analysis of pFDs, as well as showing cases in which they have the edge over AFDs and vice versa. Essentially, we demonstrated that data interpretation and data context leave room for both of them, since neither can substitute the other one. Ultimately, it's up to a data scientist to decide what to consider a violation of an exact FD, and the two concepts allow its user to target different cases. Experiments have also shown that pFD is capable of discovering some dependencies that AFD fails to find. The results featured in Table VI demonstrate that fact by counting



(a) First Batch (b) Second Batch

Fig. 3. Running time by error threshold

TABLE IX. RATIO OF RUNNING TIME OF AFDTANE AND PFDTANE PERVALUE ALGORITHMS

Error threshold / Dataset	0.025	0.05	0.075	0.1	0.15	0.2	0.25	0.3	0.4	0.5
BKB_WaterQualityData_2020084	2.529	2.199	2.010	1.763	1.432	1.267	1.128	1.076	1.050	1.027
EpicVitals	2.671	2.637	2.623	2.589	2.145	1.952	1.679	1.636	1.548	1.547
jena_climate_2009_2016	1.455	1.475	1.470	1.359	1.387	1.316	1.308	1.322	1.320	1.360
measures_v2	0.962	0.972	0.951	0.921	0.946	0.953	0.992	0.985	0.949	0.915
nuclear_explosions	2.160	1.862	1.668	1.466	1.220	1.101	1.070	1.048	1.024	1.015
parking_citations	3.358	3.261	3.177	3.079	2.882	2.513	2.073	1.550	1.351	1.231
SEA	1.745	1.815	1.767	1.805	1.752	1.821	1.794	1.773	1.615	1.674
games	1.785	1.587	1.517	1.446	1.361	1.299	1.277	1.217	1.212	1.162

TABLE X. Ratio of running time of AFDTane and pFDTane PerTuple algorithms

Dataset \ Error threshold	0.025	0.05	0.075	0.1	0.15	0.2	0.25	0.3	0.4	0.5
BKB_WaterQualityData_2020084	2.685	2.415	2.320	2.262	2.127	2.007	1.908	1.804	1.578	1.417
EpicVitals	2.642	2.636	2.677	2.638	2.283	2.087	2.086	2.031	2.017	1.974
jena_climate_2009_2016	1.465	1.479	1.514	1.487	1.410	1.343	1.326	1.396	1.316	1.368
measures_v2	0.894	0.952	0.904	0.892	0.928	0.948	0.961	0.965	0.943	0.914
nuclear_explosions	2.475	2.316	2.251	2.153	2.010	1.802	1.701	1.542	1.419	1.264
parking_citations	3.386	3.373	3.344	3.289	3.197	2.987	2.740	2.681	2.557	2.318
SEA	1.742	1.800	1.747	1.780	1.796	1.819	1.777	1.776	1.626	1.636
games	2.064	1.810	1.723	1.630	1.529	1.503	1.451	1.399	1.356	1.300

TABLE XI. Ratio of consumed memory of AFDTane and pFDTane PerValue algorithms

Dataset \ Error threshold	0.025	0.05	0.075	0.1	0.15	0.2	0.25	0.3	0.4	0.5
BKB_WaterQualityData_2020084	0.931	0.932	0.922	0.911	0.894	0.885	0.873	0.871	0.874	0.873
EpicVitals	0.542	0.542	0.542	0.542	0.521	0.521	0.486	0.486	0.486	0.486
jena_climate_2009_2016	0.356	0.356	0.356	0.356	0.356	0.356	0.356	0.356	0.356	0.356
measures_v2	0.985	0.985	0.985	0.985	0.985	0.985	0.985	0.985	0.985	0.985
nuclear_explosions	0.127	0.128	0.127	0.127	0.127	0.127	0.127	0.127	0.127	0.127
parking_citations	0.926	0.926	0.926	0.926	0.926	0.926	0.926	0.925	0.925	0.925
SEA	1.029	1.030	1.029	1.030	1.030	1.030	1.030	1.030	1.030	1.046
games	0.170	0.162	0.160	0.160	0.159	0.159	0.159	0.159	0.159	0.159

TABLE XII. Ratio of consumed memory of AFDTane and pFDTane PerTuple algorithms

Dataset \ Error threshold	0.025	0.05	0.075	0.1	0.15	0.2	0.25	0.3	0.4	0.5
BKB_WaterQualityData_2020084	0.942	0.940	0.935	0.934	0.927	0.918	0.915	0.911	0.901	0.890
EpicVitals	0.542	0.542	0.542	0.542	0.543	0.542	0.542	0.521	0.521	0.521
jena_climate_2009_2016	0.356	0.356	0.356	0.356	0.356	0.356	0.356	0.356	0.356	0.356
measures_v2	0.985	0.985	0.985	0.985	0.985	0.985	0.985	0.985	0.985	0.985
nuclear_explosions	0.127	0.128	0.127	0.127	0.127	0.126	0.126	0.126	0.127	0.127
parking_citations	0.926	0.926	0.926	0.926	0.926	0.926	0.926	0.926	0.926	0.926
SEA	1.029	1.030	1.029	1.030	1.030	1.030	1.030	1.030	1.030	1.046
games	0.184	0.169	0.165	0.163	0.161	0.160	0.160	0.160	0.159	0.159

discovered dependencies with the same error threshold for both algorithms.

As for performance, the discovery of both pFD types is almost always considerably slower than AFD. However, the memory consumption shows the opposite trend, with pFD using less memory compared to AFD. The difference between run time of pFD and AFD decreases as the error threshold increases, though useful information is primarily mined with a low error threshold. Experiments have also shown similar run time and memory consumption for both pFD PerTuple and pFD PerValue.

Overall, we have introduced pFD discovery functionality into Desbordante for both PerValue and PerTuple metrics, as we had shown that their utility depends on data interpretation and context. At the same time, while building a science-intensive data profiler it is an imperative to expand the catalogue of available tools. Therefore, we hope that this primitive will become another useful tool which will allow our users to uncover knowledge hidden in data. Source code of the implementation is available in the GitHub repository (PR 300) [11].

References

[1] Z. Abedjan, L. Golab, F. Naumann, and T. Papenbrock, *Data Profiling*. Morgan & Claypool Publishers, 2018.

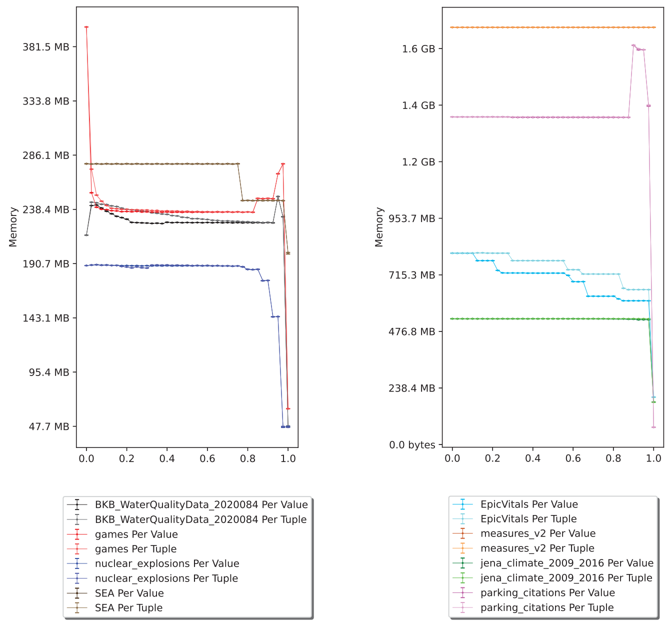
[2] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann, "Functional dependency discovery: An experimental evaluation of seven algorithms," *Proc. VLDB Endow.*, vol. 8, no. 10, p. 1082–1093, Jun. 2015. [Online]. Available: https://doi.org/10.14778/2794367.2794377

[3] F. Dürsch, A. Stebner, F. Windheuser, M. Fischer, T. Friedrich, N. Strelow, T. Bleifuß, H. Harmouch, L. Jiang, T. Papenbrock, and F. Naumann, "Inclusion dependency discovery: An experimental evaluation of thirteen algorithms," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, ser. CIKM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 219–228. [Online]. Available: https://doi.org/10.1145/3357384.3357916

[4] C. C. Aggarwal and J. Han, *Frequent Pattern Mining*. Springer Publishing Company, Incorporated, 2014.

[5] P. G. Brown and P. J. Hass, "Bhunt: Automatic discovery of fuzzy algebraic constraints in relational data," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03. VLDB Endowment, 2003, p. 668–679.

[6] M. Hulsebos, K. Hu, M. Bakker, E. Zgraggen, A. Satyanarayan, T. Kraska, c. Demiralp, and C. Hidalgo, "Sherlock: A deep learning approach to semantic data type detection," in *SIGKDD'19*. New York, NY, USA: Association for Computing Machinery, 2019, p. 1500–1508.



(a) First Batch (b) Second Batch

Fig. 4. Maximum memory consumption by error threshold

[7] P. Tsurinov, O. Shpynov, N. Lukashina, D. Likholetova, and M. Artyomov, "Farm: Hierarchical association rule mining and visualization method," in *Proceedings of the 12th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*, ser. BCB '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3459930.3469499

[8] I. Koumarelas, T. Papenbrock, and F. Naumann, "Mdedup: duplicate detection with matching dependencies," *Proc. VLDB Endow.*, vol. 13, no. 5, p. 712–725, jan 2020. [Online]. Available: https://doi.org/10.14778/3377369.3377379

[9] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann, "Data profiling with Metanome," *Proc. VLDB Endow.*, vol. 8, no. 12, p. 1860–1863, Aug. 2015. [Online]. Available: https://doi.org/10.14778/2824032.2824086

[10] G. Chernishev *et al.*, "Desbordante: from benchmarking suite to high-performance science-intensive data profiler," *CoRR*, vol. abs/2301.05965, 2023.

[11] *Desbordante GitHub repository*. [Online]. Available: https://github.com/Mstrutov/Desbordante

[12] 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 *CIKM'16*. New York, NY, USA: Association for Computing Machinery, 2016, p. 1803–1812. [Online]. Available: https://doi.org/10.1145/2983323.2983781

[13] S. Kruse, T. Papenbrock, C. Dullweber, M. Finke, M. Hegner, M. Zabel, C. Zöllner, and F. Naumann, "Fast Approximate Discovery of Inclusion Dependencies," in *BTW 2017*, ser. LNI, B. Mitschang and et al., Eds., vol. P-265. GI, 2017, pp. 207–226. [Online]. Available: https://dl.gi.de/20.500.12116/629

[14] L. Caruccio, V. Deufemia, and G. Polese, "Relaxed functional dependencies—a survey of approaches," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 1, pp. 147–165, 2016.

[15] J. Kivinen and H. Mannila, "Approximate inference of functional dependencies from relations," *Theoretical Computer Science*, vol. 149, no. 1, pp. 129–149, 1995, fourth International Conference on Database Theory (ICDT '92). [Online]. Available: https://www.sciencedirect.com/science/article/pii/030439759500028U

[16] S. Kruse and F. Naumann, "Efficient discovery of approximate dependencies," *Proc. VLDB Endow.*, vol. 11, no. 7, p. 759–772, mar 2018. [Online]. Available: https://doi.org/10.14778/3192965.3192968

[17] D. Z. Wang, X. L. Dong, A. D. Sarma, M. J. Franklin, and A. Y. Halevy, "Functional dependency generation and applications in pay-as-you-go data integration systems," in *12th International Workshop on the Web and Databases, WebDB 2009, Providence, Rhode Island, USA, June 28, 2009*, 2009. [Online]. Available: http://webdb09.cse.buffalo.edu/papers/Paper18/webdb09.pdf

[18] D. Z. Wang, M. Franklin, L. Dong, A. D. Sarma, and A. Halevy, "Discovering functional dependencies in pay-as-you-go data integration systems," Tech. Rep. UCB/EECS-2009-119, 2009. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-119.pdf

[19] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen, "Tane: An efficient algorithm for discovering functional and approximate dependencies," *The Computer Journal*, vol. 42, no. 2, pp. 100–111, 1999.

[20] M. Strutovskiy, N. Bobrov, K. Smirnov, and G. Chernishev, "Desbordante: a framework for exploring limits of dependency discovery algorithms," in *2021 29th Conference of Open Innovations Association (FRUCT)*, 2021, pp. 344–354.

[21] *TANE implementation in the Metanome project*. [Online]. Available: https://github.com/HPI-Information-Systems/pyro/blob/master/pyro-metanome/src/main/java/de/hpi/isg/pyro/algorithms/TaneX.java

[22] S. Vilmin, P. Faure-Giovagnoli, J. Petit, and V. Scuturici, "Functional dependencies with predicates: What makes the g3-error easy to compute?" in *ICCS'23*, ser. Lecture Notes in Computer Science, M. O. et al., Ed., vol. 14133. Springer, 2023, pp. 3–16. [Online]. Available: https://doi.org/10.1007/978-3-031-40960-8_1

[23] E. H. Pena, E. C. De Almeida, and F. Naumann, "Discovery of approximate (and exact) denial constraints," *Proceedings of the VLDB Endowment*, vol. 13, no. 3, pp. 266–278, 2019.

[24] L. Caruccio, V. Deufemia, and G. Polese, "Mining relaxed functional dependencies from data," *Data Mining and Knowledge Discovery*, vol. 34, no. 2, pp. 443–477, 2020.

[25] F. D. Marchi, S. Lopes, and J.-M. Petit, "Unary and n-ary inclusion dependency discovery in relational databases," *Journal of Intelligent Information Systems*, vol. 32, pp. 53–73, 2009.

[26] J. Bauckmann, U. Leser, and F. Naumann, *Efficient and Exact Computation of Inclusion Dependencies for Data Integration*, ser. Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam. Univ.-Verlag, 2010. [Online]. Available: https://books.google.ru/books?id=A2UqdXlpVOEC

[27] N. Shaabani and C. Meinel, "Scalable inclusion dependency discovery," in *Database Systems for Advanced Applications*, M. Renz, C. Shahabi, X. Zhou, and M. A. Cheema, Eds. Cham: Springer International Publishing, 2015, pp. 425–440.

[28] F. De Marchi, S. Lopes, and J.-M. Petit, "Efficient algorithms for mining inclusion dependencies," in *Advances in Database Technology — EDBT 2002*, C. S. Jensen, S. Šaltenis, K. G. Jeffery, J. Pokorny, E. Bertino, K. Böhn, and M. Jarke, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 464–476.

[29] G. Zhou, S. Kwashie, Y. Zhang, M. Bewong, V. M. Nofong, J. Hu, D. Cheng, K. He, S. Liu, and Z. Feng, "Fastageds: fast approximate graph entity dependency discovery," in *International Conference on Web Information Systems Engineering*. Springer, 2023, pp. 451–465.

[30] C. Combi and P. Sala, "Mining approximate interval-based temporal dependencies," *Acta Informatica*, vol. 53, 09 2015.

[31] P. Sala, "Approximate interval-based temporal dependencies: The complexity landscape," in *2014 21st International Symposium on Temporal Representation and Reasoning (TIME)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2014, pp. 69–78. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TIME.2014.20

[32] *Links to the datasets used in experiments*. [Online]. Available: https://gist.github.com/iliya-b/f67cf0aa8397ec0a5ab7849376ec8a31

[33] L. Berti-Équille, H. Harmouch, F. Naumann, N. Novelli, and S. Thirumuruganathan, "Discovery of genuine functional dependencies from relational data with missing values," *Proc. VLDB Endow.*, vol. 11, no. 8, pp. 880–892, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p880-berti-equille.pdf

[34] H. Harmouch, "Single-column data profiling," Ph.D. dissertation, University of Potsdam, Germany, 2020. [Online]. Available: https://publishup.uni-potsdam.de/frontdoor/index/index/docId/47455