Efficient Discovery of Matching Dependencies

PHILIPP SCHIRMER, Hasso Plattner Institute, University of Potsdam, Germany
THORSTEN PAPENBROCK, Hasso Plattner Institute, University of Potsdam, Germany
IOANNIS KOUMARELAS, Hasso Plattner Institute, University of Potsdam, Germany
FELIX NAUMANN, Hasso Plattner Institute, University of Potsdam, Germany

Matching dependencies (MDs) are data profiling results that are often used for data integration, data cleaning, and entity matching. They are a generalization of functional dependencies (FDs) matching *similar* rather than same elements. As their discovery is very difficult, existing profiling algorithms find either only small subsets of all MDs or their scope is limited to only small datasets.

We focus on the *efficient* discovery of all *interesting* MDs in real-world datasets. For this purpose, we propose HyMD, a novel MD discovery algorithm that finds all minimal, non-trivial MDs within given similarity boundaries. The algorithm extracts the exact similarity thresholds for the individual MDs from the data instead of using predefined similarity thresholds. For this reason, it is the first approach to solve the MD discovery problem in an exact and truly complete way. If needed, the algorithm can, however, enforce certain properties on the reported MDs, such as disjointness and minimum support, to focus the discovery on such results that are actually required by downstream use cases. HyMD is technically a hybrid approach that combines the two most popular dependency discovery strategies in related work: lattice traversal and inference from record pairs. Despite the additional effort of finding exact similarity thresholds for all MD candidates, the algorithm is still able to efficiently process large datasets, e.g., datasets larger than 3 GB.

CCS Concepts: • **Information systems → Incomplete data**; *Entity resolution*; *Data analytics*; *Data extraction and integration*; *Association rules.*

Additional Key Words and Phrases: Matching dependencies, functional dependencies, dependency discovery, data profiling, data matching, entity resolution, similarity measures

ACM Reference Format:
Philipp Schirmer, Thorsten Papenbrock, Ioannis Koumarelas, and Felix Naumann. 2020. Efficient Discovery of Matching Dependencies. *ACM Trans. Datab. Syst.* 1, 1, Article 1 (January 2020), 33 pages. https://doi.org/10.1145/3392778

1 MATCHING DEPENDENCIES

Functional dependencies (FDs) describe how columns in relational datasets depend on sets of other columns. Formally, an FD $X \rightarrow Y$ on a schema R with instance r states that whenever two records in r share the same values for the set of attributes $X \subseteq R$, they also agree in their $Y \subseteq R$ values. FDs are popular for supporting various data management tasks, such as query optimization [29], data integration [24], or schema normalization [7], but their reliance on value equality makes them

Authors' addresses: Philipp Schirmer, philipp.schirmer@hpi-alumni.de,Hasso Plattner Institute, University of Potsdam, Germany; Thorsten Papenbrock, thorsten.papenbrock@hpi.de,Hasso Plattner Institute, University of Potsdam, Germany; Ioannis Koumarelas, ioannis.koumarelas@hpi.de,Hasso Plattner Institute, University of Potsdam, Germany; Felix Naumann, felix.naumann@hpi.de,Hasso Plattner Institute, University of Potsdam, Germany.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
0362-5915/2020/1-ART1 $15.00
https://doi.org/10.1145/3392778

sensitive to data quality issues. For this reason, various derivations of FDs exist that relax different properties, such as similarity or extent [5].

Matching dependencies (MDs) [10] relax the equality constraint for the attributes. In other words, MDs generalize FDs by requiring pairs of records to be *similar* w.r.t. some similarity metric in the left- and right-hand side column values instead of being strictly equal. This relaxation allows MDs to tolerate data errors and express certain kinds of fuzzy dependencies. Consider, as an example, the dataset in Table 1, which violates the functional dependency `animal` → `diet`, due to a typo in the diet value "mead". This error prevents the FD to be discovered. Another error – the misspelling of "bear" as "beer" – is ignored by the FD `animal` → `diet`, i.e., the FD would hold despite this error. This data quality issue silently prevents the FD from matching the bears *Baloo* and *Pooh*. The MD $\texttt{animal}_{0.75} \rightarrow \texttt{diet}_{0.75}$, however, holds in the example dataset and can be automatically discovered. It specifies that whenever two records are at least 75% similar in their `animal` attribute (according to some similarity measure), they are also at least 75% similar in their `diet` attribute. In this way, the MD captures both errors: The `animal` relaxation matches "bear" ≈ "beer" and the `diet` relaxation matches "meat" ≈ "mead". Because the MD is not an FD, we can conclude that the data might be erroneous and the 75% thresholds would point us to the erroneous values.

name	**zoo**	**animal**	**diet**
Simba	berlin	lion	meat
Clarence	london	lion	mead
Baloo	berlin	bear	fish
Pooh	london	beer	fish

Table 1. Animals and their diet in different zoos.

The error tolerance and expressiveness of matching dependencies make these rules important for a variety of use cases. Most are centered around potentially dirty and erroneous data, such as the dataset shown in Table 1. The MDs of such a dataset are used to reveal both data errors and functional dependencies that hide behind these errors. MDs are, however, also considered on clean datasets to express more complex relationships, such as $\texttt{distance}_{0.9} \rightarrow \texttt{gas}_{0.8}$, which says that travel distances of about the same length consume about the same amount of gas. The relationship between `distance` and `gas` is not exact, i.e., it is not a functional dependency, because it depends, inter alia, on the type of car, the weather, the route and the driving behavior. A matching dependency, however, can capture it accurately.

The expressiveness of MDs also marks an important difference between matching dependencies and other popular relaxations of functional dependencies, such as conditional and approximate FDs: For conditional FDs, the FD violating tuples need to obey some (more or less informative) condition and, for approximate FDs, they can be arbitrary different; for MDs, though, the FD violating tuples need to be *similar* w.r.t. some specific similarity metric. This similarity is a strong assertion that increases the probability that the found dependencies are actually true FDs. The error of approximate FDs and the conditions of conditional FDs provide the same support but for different kinds of errors. Hence, the approaches complement one another in, for example, data cleaning use cases.

Because MDs subsume FDs, they can serve every use case that actually requires functional dependencies. If the data in Table 1 were clean, we would find $\texttt{animal}_{1.0} \rightarrow \texttt{diet}_{1.0}$, which is an FD, and could use it for, e.g., schema normalization [28] or query optimization [29]. As illustrated before with the MD $\texttt{distance}_{0.9} \rightarrow \texttt{gas}_{0.8}$, matching dependencies can also be utilized for data exploration. Most use cases for MDs, however, target data cleaning [13] scenarios. With our

example MD $\texttt{animal}_{0.75} \rightarrow \texttt{diet}_{0.75}$, for instance, we can identify and then resolve the matching values "bear" ≈ "beer" and "meat" ≈ "mead". Another popular data cleaning task that can utilize matching dependencies is the detection of duplicate entries in a dataset (see Section 6.5). Record matching [2, 12] and entity resolution [1, 35] are similar tasks that can utilize MD rules to match records within and across datasets. Assume, for example, that Table 1 is given in two versions and that $\texttt{name}_{0.95}, \texttt{animal}_{0.9} \rightarrow \texttt{zoo}_{1.0}, \texttt{diet}_{1.0}$ holds across these two versions. We could then merge the records in these two versions by similar name and animal attributes. More complex data integration scenarios exploit the fact that MDs can – as we show in Section 3 – match records with different schemata [10]. Matching dependencies have also been used in the area of fraud detection [10, 12] and integrity checking [15, 16], where the MDs represent data verification rules.

All these use cases assume the MDs of a certain dataset to be given. Because this is, in practice, not the case, data engineers and scientists need to be able to discover them. To avoid the difficult and usually imprecise manual search for MDs, automatic discovery algorithms have been proposed. Finding an *efficient* MD discovery algorithm is challenging, though, because MDs subsume FDs and FD discovery has already been proven to be NP-hard. If we assume that every attribute can take on n different similarity thresholds, then there are n^k many similarity threshold combinations for an MD with k attributes. So MD discovery is n^k times harder than FD discovery (if k is the average attribute length of all MD candidates). For this reason, existing MD profiling algorithms test only a few similarity thresholds, i.e., small n, and still cannot solve the discovery problem for large datasets.

Choosing the n thresholds is challenging not only for performance but also for precision. Assume we had chosen the thresholds 0.25, 0.5, 0.75, and 1.0. Then, instead of $\texttt{distance}_{0.9} \rightarrow \texttt{gas}_{0.8}$, which is the precise MD, we might have found $\texttt{distance}_{1.0} \rightarrow \texttt{gas}_{0.75}$ or $\texttt{distance}_{0.75} \rightarrow \texttt{gas}_{0.75}$, because the tightest threshold is never tested. Such inaccuracies impact all use cases that we discussed before, i.e., data exploration, cleaning, matching, and verification – most of them lose recall. In general, the n thresholds cannot be defined globally for all attributes, because every attribute needs its own thresholds. Still, this is what all state-of-the-art MD discovery algorithms do. Instead, the exact thresholds of one attribute should be found in the set of similarities that is to be calculated by pair-wise comparing all values of one attribute.

These observations lead to the following research questions for exact MD discovery: (1) How can we capture truly all MD candidates in a systematically enumerable way? (2) Which pruning strategies can we use to effectively reduce the potentially huge candidate spaces and result sets? (3) How can we efficiently validate MD candidates that incorporate complex similarity calculations? (4) How can we, in order to deal with the complexity of the discovery problem, apply hybrid search, being the currently most effective dependency discovery strategy?

In this paper, we answer these research questions with a novel MD discovery algorithm called HyMD, which automatically discovers all minimal, non-trivial MDs in large datasets. The algorithm infers the precise similarity thresholds from the data to find exact MDs – something that no other MD discovery algorithm achieves. This exact and much more difficult search is possible, because HyMD implements a highly efficient, hybrid discovery strategy that uses two complementary discovery algorithms. Because the result sets might become extremely large due to the exact search, the algorithm can be configured to search only within given similarity boundaries. We believe that, in the context of most use cases, specifying a range of similarities, such as 0.7 to 1.0, is not only more appropriate than specifying concrete similarity thresholds, it is also easier for the user. We also propose additional interestingness criteria to further focus the discovery on relevant MDs. In detail, we first give an overview of related work in Section 2 and then make the following contributions:

(1) *Search space model.* We use the triviality and minimality properties of MDs to define a partial order. With this order, we can structure the search space as a novel, complete MD candidate lattice that serves as a candidate traversal, storage, and pruning vehicle when used for MD discovery (Section 3.3).

(2) *Interestingness criteria.* We define five basic interestingness criteria for MDs and pruning rules based on these criteria. These rules serve to focus the discovery on relevant MDs and to control the number of discovered MDs. They also improve the performance of our profiling algorithm and can be turned off if, for some reason, *all* technically valid MDs should be reported (Section 3.4).

(3) *Discovery strategies.* We adapt the two most popular dependency discovery strategies, i.e., lattice traversal and inference from record pairs to the discovery of MDs. This is a challenge, because MDs are more complex than, for example, functional dependencies or unique column combination: They require similarity calculations, alternative index structures for validation, and different candidate generation and pruning approaches (Section 4).

(4) *Hybrid discovery.* We propose HyMD, an efficient and scalable algorithm that automatically discovers MDs within one or two relations. HyMD uses our novel search space model, pruning rules, and the two discovery strategies in a hybrid combination. While hybrid dependency discovery is a known technique, applying it to MDs discovery requires additional considerations when integrating record comparison results into the lattice data structure. Our hybrid algorithm also introduces a novel candidate validation technique for MDs and optimizes the search space traversal to maximize the impact of parallelization for candidate validations (Section 5).

(5) *Evaluation.* We provide detailed experiments and an in-depth analysis of HyMD's performance including a comparative evaluation and an evaluation of different discovery strategies. The measurements show that HyMD scales with the size of the input data and the size of the search space; it can, in particular, compute datasets of more than 3 GB size. We also exemplify the usefulness of the discovered MDs for duplicate detection applications (Section 6).

2 RELATED WORK

Matching dependencies were introduced by Fan [10]. The proposal to extract the decision boundary assignments from a given relational dataset was then made by Song and Chen [33]. In this work, we use a similar notation for our MDs and the same theoretical foundations. Song and Chen also proposed an approach for the automatic discovery of MDs in [33] and an improved version later in [34]. These approaches are technically restricted to only small MDs with few Lhs column matches, because their naive search approach supports no minimality pruning and, hence, generates a huge number of candidates. They therefore discover only a subset of the MDs that we discover. Our discovery algorithm does adopt the support pruning strategy proposed in [34], but it also adds a more effective candidate validation approach, a more systematic candidate generation technique, and minimality pruning rules. In addition to normal MDs, Song and Chen also seek to discover *partial* MDs, which let their algorithm discover very different MDs than our approach.

The works [12] and [11] use MDs for reasoning about key candidates and also propose rules that enable the inference of MDs from a small set of given MDs. With our lattice search space model, we follow an alternative, more systematic and complete candidate inference strategy.

Wang et al. recently introduced conditional matching dependencies (cMDs) as a combination of MDs and conditional FDs [38]. A cMD leverages the advantages of both relaxation dimensions, value similarity and extent. The authors provide a discovery algorithm that finds all such cMDs that have an identifying attribute on the Rhs, because the discovery is particularly tailored to the use case of entity matching. This special focus and the fact that decision boundaries are fixed rather than retrieved from the data reduces the search space significantly. On the other hand, introducing

an extent relaxation and conditions increases the complexity of the discovery task, so that their results and performance properties are not comparable to ours. For more details about relaxation dimensions, we refer to [5]. The algorithm of Wang et al. is also from a technical perspective hardly comparable to our approach: It uses a search strategy, pruning rules, and inference techniques that are specialized on cMD discovery and it is not clear how these approaches would work for non-conditional MDs with arbitrary RHS attributes.

Because MDs are often used for entity matching, all rule-based entity matching systems that automatically mine their matching rules can be considered as related work. The work of Wang et al. [37], for example, addresses the problem of finding good similarity functions and similarity thresholds for simple conjunctive matching rules. Singh et al. go one step further and propose a new formalism, i.e., the *General Boolean Formula* (GBF) to describe entity matching rules [32]. Both of these approaches and all other automatic entity matching systems learn their matching rules from pre-labeled training data, which are sets of positive and negative matches. Our MD discovery algorithm also "learns" similarities from the data, but it uses the concept of attribute dependence to identify attribute matches and thresholds – it does not require pre-labeled training data. Matching dependency discovery is, therefore, a more general approach, whose discovered rules can also be used for purposes other than entity matching (e.g., error correction or integrity checking). The cost for being independent of training data is that with MDs we can find only such matching rules that have a dependent target column. General conjunctive matching rules and, in particular, extended rule formalisms, such as GBF and cMDs, can solve entity matching tasks more accurately, if training data are available. Overall, the mining of matching rules with a gold standard of matches is very different from general matching dependency discovery.

FDs are arguably the most popular data dependencies. Their discovery has, for this reason, gained a lot of attention in data profiling research. Because FDs are a specialization of MDs, many FD discovery strategies are – with the appropriate adaptations – also applicable to MD discovery. The algorithm TANE [18], for instance, proposes a lattice traversal technique that scales well with the number of records in a dataset, and the algorithm FDEP [14] infers dependencies from pair-wise record comparisons, which scales well with the number of attributes in the data. These two dependency discovery techniques in combination yield a powerful and robust profiling approach, as shown by the HYFD algorithm in [27]. In HYMD, we propose the same combination of lattice traversal and inference from record pairs strategies for the more complex task of MD discovery.

3 FOUNDATIONS

In this section, we formally define matching dependencies and introduce the foundations necessary for MD discovery. We discuss triviality and minimality properties of MDs, define a partial order on MDs to structure the search space, and introduce different notions of interestingness that reduce the solution size to MDs with special properties.

We use the abstract relational instance shown in Table 2 as a running example. The relation has three columns and four rows (*id* is considered as an implicit column). To discover all MDs in this relational instance, we use three column matches, i.e., each column is mapped to itself, with some column-specific, but for this example irrelevant similarity measures. All pair-wise similarities are listed in Table 3. In this work, we distinguish three main concepts: record identifiers (1, 2, 3, 4), column values (a_i, b_i, c_i) and similarities (0.0, 0.5, 0.7, 0.8, 1.0). The color coding should help to follow the concepts visually through the different figures.

3.1 Matching Dependencies

The core feature of matching dependencies, separating them from functional dependencies, is their ability to classify values as similar or dissimilar. Two values are classified as similar using a

id	**A**	**B**	**C**
1	a_1	b_1	c_1
2	a_2	b_2	c_2
3	a_3	b_2	c_3
4	a_4	b_3	c_2

Table 2. Example relation

records	**A**	**B**	**C**
(1,2)	0.0	0.8	0.0
(1,3)	1.0	0.8	0.5
(1,4)	1.0	0.0	0.0
(2,3)	0.7	1.0	1.0
(2,4)	0.7	0.0	1.0
(3,4)	0.7	0.0	1.0

Table 3. Similarity table

similarity measure and a *decision boundary*. A similarity measure $\approx$ is a function that determines the similarity of two values on a scale from 0.0 to 1.0, where 1.0 indicates maximum similarity (equality) and 0.0 maximal dissimilarity. Well-known similarity measures include Levenshtein [22], Jaccard [21], and Monge-Elkan [25]. Since many further similarity measures exist, we refer to the survey in [9]. A decision boundary λ (or ρ) decides for two values if they are similar or dissimilar: Similarities greater than or equal to λ are considered similar and lower similarities are considered dissimilar; hence, $\lambda = 0.0$ classifies any two values as similar. The decision boundaries of an MD are often referred to as the MD's *thresholds*. A similarity measure with a decision boundary forms a so-called *similarity classifier*.

Besides introducing a notion of similarity for the record comparisons, MDs also extend the definition of FDs to two relations: Given two (potentially same) relations R and S and a set of similarity measures $\approx$, an MD states a dependency on a set of *column matches* $C = \{C_1, C_2, \ldots, C_m\}$ over R and S, where $C_i = (A_i, B_i, \approx_i) \in R \times S \times \approx$. The set of column matches C usually defines a one-to-one mapping of attributes in R to attributes in S, based on a given schema mapping (or identity if $R = S$). MDs can, however, by definition match same column pairs multiple times with different similarity measures or not match certain columns at all. The MD discovery could, therefore, be started with all possible column matches, i.e., $C = R \times S \times \approx$, and by letting the discovery algorithm find the appropriate ones automatically. While this might be useful for data integration and schema matching use cases, the large number of possible combinations affects the discovery performance significantly. Therefore, we define C as a domain-dependent *subset* of all possible column matches, i.e., $C \subset R \times S \times \approx$, and expose it as an input parameter of the discovery process. If $r = s$, C usually maps all columns to themselves; if $r \neq s$, schema matching techniques can create the column matches in C [30]. Usually, the domain of a column match naturally suggests a similarity function, such as edit distance for strings, token distance for text, numeric distance for numbers, temporal distance for times, calendrical distance for dates, or Euclidean distance for coordinates. If the similarity function for some column match is not obvious, we define multiple column matches, one for each possible similarity function.

The two concepts, similarity classifier and column matches, lead to the following formal definition of MDs:

Definition 1. *Given a set of column matches $C \subseteq R \times S \times \approx$ over relations R and S and similarity measures $\approx$, where $C_i = (A_i, B_i, \approx_i)$. Then, an MD φ is defined as*

$$\left(\bigwedge_{i=1}^{m} R[A_i] \approx_{i,\lambda_i} S[B_i] \right) \rightarrow R[A_j] \approx_{j,\rho_j} S[B_j]$$

The left-hand side (Lhs) is a conjunction of m column similarity classifiers. Each column similarity classifier is represented as a column match C_i with a specific decision boundary $\lambda_i \in [0.0, 1.0]$. The set of Lhs decision boundaries is denoted as $\lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_m\}$. The right-hand side (Rhs) is a

single column similarity classifier for some $j \in [1, m]$. It consists of a column match C_j with similarity measure $\approx_j$ and a decision boundary $\rho_j \in [0.0, 1.0]$.

Given two relational instances $r \vDash R$ and $s \vDash S$, a pair of records $(r_k, s_l) \in r \times s$ matches the LHS X of an MD iff these records are classified as similar for each column match given an assignment λ. More formally, this is expressed as

$$(r_k, s_l) \vDash X \equiv \bigwedge_{i=1}^{m} (r_k[A_i] \approx_i s_l[B_i]) \geq \lambda_i$$

Similarly, a pair of records $(r_k, s_l) \in r \times s$ matches the RHS Y of an MD iff these records are classified as similar for each column match given an assignment ρ, or formally

$$(r_k, s_l) \vDash Y \equiv \left(r_k[A_j] \approx_j s_l[B_j]\right) \geq \rho_j$$

So an MD φ holds for two instances $r \vDash R$ and $s \vDash S$ iff

$$\forall (r_k, s_l) \in (r \times s) : (r_k, s_l) \vDash X \rightarrow (r_k, s_l) \vDash Y$$

According to this definition, every MD holds all possible column matches C on its LHS, i.e., the LHS always has a size of $|C| = m$. Only those $C_i \in C$ with $\lambda_i > 0.0$ are, however, relevant for the MD, because they can classify two records as dissimilar and, therefore, actually narrow the scope of the MD; a column match $C_i \in C$ with $\lambda_i = 0.0$ matches any two values and is, consequently, irrelevant for the MD's validity. For this reason, we omit each $\lambda_i = 0.0$ in our MD specifications and list only those LHS column matches with $\lambda_i > 0.0$.

Because an MD φ is fully characterized by λ and ρ_j given the column matches C, we sometimes denote an MD as $\varphi(\lambda, \rho_j)$. In Section 1, we also used the illustrative short notation that omits the similarity function and, if attributes are matches to themselves, shows only one attribute. For instance, we wrote $\texttt{animal}_{0.75} \rightarrow \texttt{diet}_{0.75}$, which means that if the similarity of two records in the column match represented by `animal` is greater than or equal to 0.75, the similarity in the columns of `diet` is at least 0.75. Note that MDs have – just like FDs – multiple LHS column matches but only one RHS column match. We can, nevertheless, group MDs with identical LHS column matches as $\varphi(\lambda, \rho)$, because they match the same record pairs. In illustrative short notation, this grouping looks like $\texttt{name}_{0.93} \rightarrow \texttt{zoo}_{1.0}, \texttt{animal}_{0.61}$ (Table 1) or $C_{0.5} \rightarrow A_{0.7}, B_{0.0}$ (Table 2).

The goal of MD discovery is to find all $\varphi(\lambda, \rho_j)$ or, in other words, to find all valid assignments for λ and the related ρ_j. The possible decision boundaries for the LHS column matches are to be derived from the data, which means that any existing similarity between two attribute values defines one $\lambda_i \in \lambda$. Because the decision boundaries are derived from the data, we call them *natural decision boundaries*. For each set of λ assignments, the corresponding maximum RHS decision boundary ρ_j, with which the MD still holds, then corresponds to the minimal similarity of matching records in their RHS attributes. In other words, all records that match on their LHS due to λ must also match on their RHS so that we need to set ρ_j as low as the lowest similarity of matching records on that side to form a correct MD. To illustrate this with an example, consider the MD $A_{0.7}B_{0.8} \rightarrow C_{0.5}$, which is true for our example Table 2. The LHS matches the record pairs (1,3) and (2,3), whose minimum similarity in attribute C is 0.5. Hence, $\rho_j = 0.5$ is the decision boundary for the RHS attribute C. A higher value for ρ_j would make the MD invalid. Also, a lower λ value for either A or B would make the MD match additional record pairs that also invalidate the MD. For this reason, $A_{0.7}B_{0.8} \rightarrow C_{0.5}$ is exactly the most specific MD supported by the data and, therefore, to be discovered.

The number of candidate MDs that a discovery algorithm needs to consider is extremely large: Let there be m column matches and t_i possible decision boundaries for each column match C_i. Then there are $\prod_{i=1}^{m} t_i$ possible LHSs (= λ assignments) that can be generated. This number is exponential in both the number of column matches and possible decision boundaries. In addition, the RHS can

have t_j different assignments for each ρ_j Rhs column match C_j. Therefore, the number of possible MDs is $\sum_{j=1}^{m} \left(t_j \cdot \prod_{i=1}^{m} t_i\right)$. Since the number of decision boundaries t_i depends on the number of rows in both r and s (natural decision boundaries), there are up to $|r| \cdot |s|$ possible t_i to be tested. With minimality pruning rules, we can control the search complexity a bit, but it remains NP-hard – much harder, in particular, than FD discovery, where $t_i \in \{0, 1\}$ and $|C| = |R|$.

All existing approaches for the discovery of matching dependencies including [34] and [38] use pre-defined decision boundaries, which fixes the number t to a constant. Although this limitation reduces the complexity of the MD discovery significantly, it also makes it impossible for the algorithms to discover *all* MDs. For example, $\texttt{animal}_{0.75} \rightarrow \texttt{diet}_{0.75}$ is an exact MD in our introductory example (see Table 1). With fixed decision boundaries, let's say 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0, we would not find this MD but $\texttt{animal}_{0.6} \rightarrow \texttt{diet}_{0.6}$ instead. This MD is less accurate, because both thresholds could be set 15% higher without violating the dependency. Depending on how fine-grained the thresholds are chosen, the discovered MDs are more or less accurate. By deriving the thresholds from the data, the discovery always finds the exact dependencies and, therefore, all dependencies. In the following section, Section 3.2, we discuss how any true MD can be derived from a complete set of MDs discovered with automatically derived thresholds.

3.2 Trivial and minimal MDs

Given an assignment λ for the Lhs decision boundaries, then all record pairs matching this Lhs have a similarity of at least λ_i in the respective column match C_i. Any MD with Rhs decision boundary $\rho_j \leq \lambda_j$ must, hence, be true regardless of the given data, i.e., we can simply generate all such *trivial* MDs without even inspecting the data:

Definition 2. *An MD $\varphi(\lambda, \rho_j)$ is* trivial *iff $\lambda_j \geq \rho_j$.*

This definition generalizes the definition of trivial FDs, which states that an FD is trivial if Rhs $\subseteq$ Lhs: If an attribute with index j is present on both sides of an FD, both ρ_j and λ_j are 1.0, which also makes the FD trivial according to Definition 2. If an attribute j is part of only the Rhs, then the respective λ_j is 0.0 and the FD is non-trivial.

All MDs with a Rhs decision boundary of 0.0 are by definition trivial. MDs with a non-zero Lhs decision boundary for the Rhs column match are, however, not necessarily trivial. The MD $A_{0.5}B_{0.6} \rightarrow A_{0.6}$, for instance, is non-trivial, because its Rhs decision boundary for attribute A is stricter than the Lhs decision boundary for A: A-values may match on the Lhs but not on the Rhs. Further examples for trivial and non-trivial MDs are shown in Table 4. An MD $\varphi(\lambda, \rho_j)$ is *disjoint* iff $\lambda_j = 0.0$. Hence, all MDs in Table 4 are non-disjoint. Any disjoint MD with $\rho_j \neq 0.0$ is non-trivial.

trivial	non-trivial
$A_{0.5} \rightarrow A_{0.5}$	$A_{0.5} \rightarrow A_{0.6}$
$A_{0.5} \rightarrow A_{0.4}$	
$A_{0.5}B_{0.6} \rightarrow A_{0.5}$	$A_{0.5}B_{0.6} \rightarrow A_{0.6}$
$A_{0.5}B_{0.6} \rightarrow A_{0.4}$	

Table 4. Examples for trivial and non-trivial MDs

The number of possible MDs in a relational dataset grows exponentially with the number of rows and columns. The MDs are, however, not independent of one another, i.e., we can infer valid MDs from already discovered MDs (minimality pruning). Hence, we now define a partial order for MDs in such a way that valid MDs also determine all their successor MDs as valid. We start by ordering all possible Lhss. Each set of Lhs decision boundaries is matched by a set of record pairs.

Increasing any of these decision boundaries also increases the selectivity of the respective classifier so that it matches only a subset of the previously matched record pairs. We say that λ *subsumes* λ' iff

$$\lambda \preceq \lambda' \equiv \forall i \in [1, m] : \lambda_i \leq \lambda'_i$$

In other words, λ is a *generalization* of λ', and λ' is a *specialization* of λ. Given that $\varphi(\lambda, \rho_j)$ is a valid MD, we know that any other MD $\varphi'(\lambda', \rho'_j)$ with $\lambda \preceq \lambda'$ matches a subset of the record pairs matched by φ. If for $\varphi(\lambda, \rho_j)$ and $\varphi'(\lambda', \rho'_j)$ also $\rho'_j \leq \rho_j$ is true, the MD φ' needs to be valid as well, because ρ'_j matches at least the same values as ρ_j and can, therefore, not introduce new violating value pairs on the Rhs. We say that an MD φ *subsumes* another MD φ' iff

$$\varphi(\lambda, \rho_j) \preceq \varphi['(\lambda', \rho'_j)] \equiv \lambda \preceq \lambda' \wedge \rho_j \geq \rho'_j$$

Again, φ is a *generalization* of φ', and φ' is a *specialization* of φ. If an MD holds, all of its specializations hold as well and we can easily infer them. The subsumption relationship defines a partial order on the set of all MDs Φ.

Below are three examples for valid subsumption relationships. Intuitively, raising an Lhs decision bounary λ_i (1)(2) or reducing an Rhs decision boundary ρ_j (3) generates valid MDs from existing MDs; any other decision boundary manipulation generates incomparable MD candidates.

$$A_{0.5} \rightarrow C_{0.6} \preceq A_{0.6} \rightarrow C_{0.6} \tag{1}$$

$$A_{0.5} \rightarrow C_{0.6} \preceq A_{0.5}B_{0.1} \rightarrow C_{0.6} \tag{2}$$

$$A_{0.5} \rightarrow C_{0.6} \preceq A_{0.5} \rightarrow C_{0.5} \tag{3}$$

Because many MDs can be inferred from others, we can restrict the discovery to only those MDs that cannot be inferred. We call such MDs *minimal*.

Definition 3. *A matching dependency $\varphi \in \Phi$ is* minimal *iff $\nexists \varphi' \in \Phi : \varphi' \neq \varphi \wedge \varphi' \preceq \varphi$.*

This definition is a generalization of the definition of minimal FDs. Like in FD discovery, the set of minimal dependencies is much smaller than the set of all dependencies and still defines the complete set of dependencies. Our discovery approach, therefore, discovers only minimal MDs.

3.3 Lattice

The partial order of MDs defined above describes a power set lattice of column classifiers, i.e., column matches and their decision boundaries. More precisely, it describes one power set lattice for every possible Rhs column match, because MDs with different Rhs column matches are independent. Similar to the discovery of other types of dependencies, the lattice serves as a structure to systematically traverse the search space of all possible MD candidates.

The lattice for any Rhs column match C_j starts with the root MD $\emptyset \rightarrow C_{j,1.0}$. Here, $\emptyset$ represents the decision boundary assignment λ with $\lambda_i = 0.0$ for all i, i.e., all decision boundaries are set to 0.0, and $C_{j,1.0}$ is the lattice specific Rhs column match C_j with maximum decision boundary $\rho_j = 1.0$. To generate a direct successor of this (or any other) lattice node, we either increase one λ_i or decrease ρ_j.

To increase or decrease a decision boundary, we use the natural decision boundaries of the respective column match, i.e., the sorted list of value similarities that are actually present in the respective column match. This guarantees that only potentially minimal MD candidates are generated. The similarity table in Table 3 lists these natural decision boundaries for every attribute of our example relation.

For our running example, Figure 1 depicts the full power set lattice of MD candidates for the Rhs column match C (note that there is another such lattice for A and B). Every column match generates

one dimension and the length of each dimension is determined by the number of possible decision boundaries. The arrows denote direct specialization relationships, i.e., they change only one decision boundary by one natural step. We omitted all trivial and non-disjoint MDs for simplicity; these MDs are, nevertheless, usually part of the lattice. Figure 1 also presents a consistent classification of all MD candidates: The valid MDs are at the top, whereas the invalid MDs are below. The only two minimal MDs for the Rhs column match C, $B_{1.0} \rightarrow C_{1.0}$ and $A_{0.7}B_{0.8} \rightarrow C_{0.5}$, are highlighted in the middle. Since minimal MDs are never preceded by a valid MD, we can traverse the lattice from root to top, prune all specializations of valid MDs, and, in this way, discover all minimal MDs.

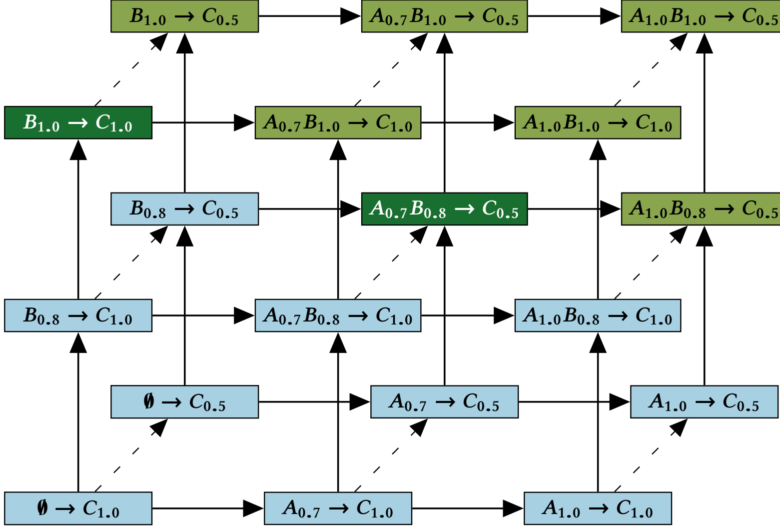

Fig. 1. The search space lattice for Rhs column match C w.r.t. the example relation shown in Table 2 and its three column matches A, B, and C. Solid and dashed arrows indicate Lhs and Rhs specializations, respectively.

Although two MDs from different lattices, i.e., with different Rhss, are order-wise unrelated, they might have identical Lhss and match the same record pairs. Our algorithm leverages this fact by calculating each Lhs only once.

We define the *depth* of an MD candidate as the length of the shortest path from the root to the MD. Steps in the Rhs dimension do not add to the length of a path so that all MDs with identical Lhs have the same depth. The MD $A_{0.7}B_{0.8} \rightarrow C_{0.5}$, for instance, has a depth of two. All nodes with different Lhs but same depth are independent of each other, i.e., they neither generalize nor specialize one another. We later use this observation to parallelize discovery.

3.4 Interestingness

Although we aim to discover only minimal, non-trivial matching dependencies, the result sets are often unreasonably large, because every combination of Lhs and Rhs attributes forms valid MDs for some decision boundary assignments. Not all of these MDs are, however, actually *interesting*. We propose different criteria for interestingness that can be used to prune (or prioritize) certain MDs. All proposed pruning strategies are optional so that our algorithm can, technically, still discover all MDs. Section 6 shows the effects, i.e., result size and runtime reduction, of using the pruning strategies.

Cardinality. The *cardinality* $|\varphi|$ of an MD $\varphi(\lambda, \rho_j)$ is the number of non-zero Lhs decision boundaries: $|\varphi| := |\{\lambda_i | \lambda_i \in \lambda \wedge \lambda_i \neq 0.0\}|$. MDs with low cardinality are statistically more likely to be true

and also easier to understand. Consider, for example, the two MDs φ_1 : $\texttt{animal}_{0.75} \rightarrow \texttt{diet}_{0.75}$ and φ_2 : $\texttt{name}_{0.6}$ $\texttt{zoo}_{0.5}$ $\texttt{habitat}_{0.4}$ $\texttt{weight}_{0.95} \rightarrow \texttt{diet}_{0.6}$. The MD φ_2 is clearly harder to understand than the MD φ_1, because it correlates five instead of only two attributes. The long MD φ_2 is also semantically weaker, because it takes four diet-unrelated attributes to draw a dependency with diet. Works like [27], therefore, successively prune large, i.e., high-level dependencies during dependency discovery if memory is exhausted. For MD discovery, we propose the same dynamic pruning behavior: If the entire result is not computable, which is when it does not fit into memory, then cardinality pruning starts successively discarding candidates on the highest lattice-levels until the available memory allows the discovery to continue. Cardinality pruning is therefore a last resort strategy. However, if a certain use case requires only MDs of a certain size, e.g. if a data cleaning framework can parse only rules with at most 10 predicates, then we can also give an initial maximum cardinality as a parameter. In our experiments, we do not specify any maximum cardinality and, since the algorithm did not exhaust the available memory, this pruning was effectively never used.

Support. We define the *support* σ of an MD $\varphi(\lambda, \rho_j)$ as the number of record pairs that match the MD's LHS: $\sigma(\varphi) := |\{(r_k, s_l)|(r_k, s_l) \in r \times s \wedge (r_k, s_l) \vDash \varphi\}|$. If $r = s$, the support of each MD is at least $|r|$, because every record matches itself; if $r \neq s$, the support of a valid MD may be 0. Such MDs have basically no support in the data and cannot be considered interesting. Even MDs with only a few record pairs support are doubtfully interesting. The MD φ_1 : $\texttt{animal}_{0.75} \rightarrow \texttt{diet}_{0.75}$, for instance, has a support of six record pairs (four self-matches, and (r_1, r_2) and (r_3, r_4)) in our example dataset of Table 1, while the MD φ_3 : $\texttt{name}_{0.2} \rightarrow \texttt{animal}_{0.75}$ has a support of only five record pairs (four self-matches and (r_3, r_4)). So (r_1, r_2) provides evidence that φ_1 is true but no evidence whether or not φ_3 is true. This makes φ_1 more interesting and statistically significant than φ_3. To consider the statistical relevance of the MDs in the discovery, works like [33] propose a *minimal support* threshold for pruning: All candidates with less support are ignored. Because specializations of unsupported MDs have the same or even less support, they can be pruned as well. Choosing a good support threshold depends on the given data and use case. A data cleaning use case, for example, might already find a data quality issue with an MD of one record pair support, i.e., that particular record pair could be faulty; an entity linkage use case, on the other hand, might want to link records from r to s and, hence, should require a support of close to $|r|$. Due to the use case dependence of this parameter, we use a conservative minimal support of $|r| + 1$ if $r = s$ and 1 otherwise, so that at least one non-reflexive record pair needs to match, i.e., support the MD.

Disjointness. As stated in Section 3.2, MDs with non-disjoint LHS and RHS attribute sets are not necessarily trivial. They are, however, less interesting than *disjoint* MDs, because they make statements about RHS attributes that are also used as a premise: Record pairs that match the LHS of an MD already have a certain similarity in the corresponding RHS. Consider, for example, the non-disjoint MD φ_4 : $\texttt{animal}_{0.75} \rightarrow \texttt{animal}_{0.75}$. Clearly, φ_4 is not an interesting dependency, because it holds on any instance. The MD φ_5 : $\texttt{animal}_{0.9} \rightarrow \texttt{animal}_{0.75}$ and any other non-disjoint MD with a higher LHS than RHS threshold does provide some insight, i.e., all records in the classes of 90% similar values are actually 75% similar, but any such MD is not minimal so that we would not discover them anyway. For this reason, we propose disjointness pruning that allows to restrict the discovery on MDs with disjoint LHS and RHS attribute sets. Because non-disjoint MDs seem to have no special relevance for any use case, we experimentally show the additional costs of discovering non-disjoint MDs in Section 6 and use this pruning rule by default.

Decision boundaries (value). Low decision boundaries λ_i match quite dissimilar values. This is quite converse to what MDs are usually used for, namely to indicate similarities. Because the profiling process infers the MDs from the data, it needs to assume that everything that is not

violated by the data could be true. Low decision boundaries are, therefore, mostly the result of a lack of counter examples, i.e., data. Our example MD $\varphi_3 : \texttt{name}_{0.2} \rightarrow \texttt{animal}_{0.75}$, for instance, has a very low Lhs decision boundary, because the example instance is extremely short. In reality, it is quite likely that two different animals have the same name, so that the φ_3 is rather spurious. To focus on more reliable MDs, we introduce a *minimal decision boundary* threshold and prune candidates with smaller decision boundaries from the search space. Note that we still fetch the λ_i values as natural decision boundaries from the data, but retain only those that are larger than the global minimal threshold; the decision boundary 0.0 needs to be retained, though, to represent that a column match is irrelevant. Depending on the given use case, a lower or higher minimal decision boundary threshold is advisable. The simple question that a user has to answer is "*How dissimilar can two values in my use case be to be still considered as the same value?*" Because the discovery algorithm automatically checks all higher similarities, this question can be answered with a conservatively low decision boundary threshold. For this reason, we define the minimal decision boundary to be 0.7 in our experiments so that two values need to be at least 70% similar w.r.t. their column match specific similarity measure. Considering values with only 70% similarity as matches, i.e., as equal, is a conservative setting that should well represent most use cases of MDs. While some use cases, such as data exploration, might actually require such a low decision boundary threshold to find certain interesting but weak relationships, other use cases, such as data integration, require much stricter, e.g., 90% and higher, similarity constrains.

Decision boundaries (number). The number of natural decision boundaries can be high, i.e., $min(|r|, |s|)$ per attribute in the worst case, which has a significant impact on the size of the search space. Moreover, excessively fine grained intervals (e.g., in fractions of thousands) might not even matter with regard to interestingness. The exact distinction between $\texttt{animal}_{0.75} \rightarrow \texttt{description}_{0.989245}$ and $\texttt{animal}_{0.75} \rightarrow \texttt{description}_{0.990926}$ might, for instance, not be relevant for data exploration use cases, because both MDs basically state the same insight, which is that the `description` values match *almost perfectly*. We therefore propose to limit the number of possible decision boundaries for each attribute, if the result does not need to be perfectly precise: Given the natural decision boundaries, we select k decision boundaries by choosing every $\frac{t_i}{k}$th decision boundary or, if the distribution of decision boundaries is skewed, we select k uniformly distributed decision boundaries from the range of natural decision boundaries. For our evaluation, we decided not to apply this pruning strategy and use *all* natural decision boundaries by default, because their number and their selection is highly use-case dependent: While a data exploration process might cope with larger similarity intervals, fine-grained integrity checking requires more precise decision boundaries. Reducing the number of decision boundaries is also a necessary tool to forcefully reduce the discovery time, if the discovery time is restricted. For this reason, we support it in our discovery approach.

4 TWO DISCOVERY APPROACHES

For the discovery of data dependencies, we find two search strategies in related work: *Lattice traversal* and *inference from record pairs*. The former models the search space as a power set lattice of attribute combinations to classify the candidates within the lattice as either true or false dependencies; different traversal strategies and pruning rules aim to reduce the number of candidate checks. The latter strategy compares all (necessary) pairs of records in search of non-dependencies from which it derives all true dependencies; refraining individual candidate checks, this strategy controls the search space size more easily, but the costs for record comparisons are quadratic in the number of records.

Algorithm 1: Lattice traversal

```
Data: Relational instances r, s. Column matches C. Minimal RHS decision boundaries ρ_min.
      Minimal support minSup.
Result: Set of all minimal MDs Φ.
1  m ← |C|;
2  Φ ← {φ(∅, (1.0, j))|j ∈ [1, m]};
3  l ← 0;
4  while l ≤ getMaxLevel(Φ) do
5      foreach φ(λ, ρ) ∈ getLevel(l, Φ) do
6          Φ ← Φ \ φ;
7          λ_lower ← getLowerBoundaries(λ, Φ);
8          ρ′, σ ← validate(λ, ρ, r, s, C, λ_lower, ρ_min);
9          if σ < minSup then
10             markUnsupported(λ);
11         else
12             foreach ρ′_j ∈ ρ′ do
13                 if ρ′_j > λ_j ∧
14                    ρ′_j ≥ ρ_min[j] ∧
15                    ρ′_j > λ_lower[j] then
16                     add(φ(λ, ρ′_j), Φ)
17             foreach i ∈ [1, m] do
18                 if canSpecializeLhs(λ, i) then
19                     λ′ ← specializeLhs(λ, i);
20                     if isSupported(λ′) then
21                         foreach ρ_j ∈ ρ do
22                             if ρ_j > λ′_j then
23                                 addIfMin(φ(λ′, ρ_j), Φ);
24     l ← l + 1;
25 return Φ;
```

In this section, we propose one MD discovery algorithm for each strategy. Although both algorithms are already fully self-contained MD discovery solutions, we later combine them into the even more efficient hybrid solution HyMD.

4.1 Lattice traversal

Our lattice traversal algorithm is a bottom-up, breadth-first approach, which is based on the FD discovery algorithm Tane [18]: Starting with the most general MDs at the lattice root (see Figure 1 in Section 3.3), the algorithm walks its way up level-by-level validating all candidates it passes; during traversal, the algorithm prunes all non-minimal MD candidates. Note that there is conceptually one lattice for each Rhs column match, but we traverse them simultaneously handling all MDs with the same Lhs at once for efficiency reasons. Algorithm 1 explains the traversal in more detail.

The algorithm starts by initializing the lattice with the most general MDs (line 2). For now, an MD's depth is regarded as its level; we later re-define the level of an MD with a different function to improve parallelization. After retrieving the current level from the lattice, the algorithm handles each MD candidate in that level (line 5). The goal is to determine all maximal Rhs decision boundaries for which the current Lhs forms valid MDs. Due to minimality pruning (see later), all such MDs are minimal. First, we remove the candidate from the lattice (line 6). Before validating the MD, we retrieve the maximum Rhs decision boundary that exists in the lattice for an MD with a more general Lhs than the current (line 7). To find a new minimal MD with the current Lhs, its Rhs decision boundary must be higher than the respective lower bound. To retrieve these lower boundaries, the algorithm needs to check all MDs in the lattice with a more general Lhs. Section 5 presents an implementation of the lattice and this checking operation.

Given the lower boundaries, the algorithm now determines the maximal Rhs decision boundaries for the current candidate, i.e., we validate it (line 8): We identify all matching record pairs, determine their minimal similarities in the Rhs column matches, and calculate their support. Again, we postpone the detailed discussion of an efficient validation algorithm to Section 5.3.2. Having determined the support and the maximal Rhs decision boundaries, the algorithm distinguishes two cases using the minimal support threshold: (a) the Lhs is supported or (b) it is not supported. If it is not supported, we mark the Lhs as unsupported and stop the traversal for this candidate (line 10); if it is supported, we continue inferring new MD candidates. First, we check for every Rhs decision boundary if it results in an interesting, non-trivial, and minimal MD. If that is the case, we add the MD to the lattice (line 16). The algorithm can also immediately emit this MD as a result. Next, we infer specializations that are not yet covered by any minimal MD. It creates these specializations by increasing each Lhs decision boundary λ_i to the next possible value (line 19). The old Rhs decision boundary is retained and the specialization is added to the lattice if it is minimal (line 23).

In the discovery process, Algorithm 1 has marked some MD candidates as *unsupported* (line 10) and it has checked whether certain new candidates are *supported* (line 20). To do this efficiently, the algorithm stores unsupported Lhss λ in an additional lattice and checks the support of a new Lhs λ' via generalization look-up in that lattice. Like the main candidate lattice, this additional lattice is implemented as a prefix tree, but instead of annotating Rhs decision boundaries, we annotate if the Lhs is supported or not.

After processing each MD of a current level, the algorithm proceeds to the next level. When all levels are processed, i.e., when the lattice holds no more candidates in higher levels, all minimal MDs have been found.

4.2 Inference from record pairs

A second discovery approach is the inference from record pairs: We systematically infer non-dependencies from record pair comparisons and use them to derive all valid dependencies. This technique was first proposed by the FD discovery algorithm Fdep [14]. The inference is very efficient w.r.t. the size of the candidate space, because finding one invalidating record pair suffices to mark a dependency as invalid.

To understand which MD candidates a specific pair of records invalidates, we consider two records r and s and their concrete similarity value $\mathsf{sim}_i \in \mathsf{sim}$ in each column match $C_i \in C$. An MD candidate φ with Lhs decision boundaries λ *matches* the record pair, iff for all $\lambda_i \in \lambda$ it holds that $\lambda_i \leq \mathsf{sim}_i$, i.e., if all Lhs decision boundaries are smaller than or equal to the observed similarities in the record pair. We can now infer the (in-)validity of φ as follows: If the MD candidate φ matches some record pair (r, s) and its Rhs decision boundary ρ_j is larger than the observed similarity, i.e., $\rho_j > \mathsf{sim}_j$, then this record *violates* φ.

Algorithm 2: Inference from record pairs

Data: Relational instances r, s. Column matches C. Minimal Rhs decision boundaries ρ_{min}.
Result: Set of all minimal MDs Φ.

```
1  m ← |C|;
2  Φ ← {φ(∅, (1.0, j))|j ∈ [1, m]};
3  foreach (r_k, s_l) ∈ r × s do
4      sim ← calculateSimilarity(r_k, s_l, C);
5      violated ← findViolated(sim, Φ);
6      foreach φ(λ, ρ_j) ∈ violated do
7          Φ ← Φ \ φ;
8          if sim[j] ≥ ρ_min[j] ∧ sim[j] > λ_j then
9              ρ'_j ← sim[j];
10             addIfMin(φ(λ, ρ'_j), Φ);
11         foreach i ∈ [1, m] do
12             if canSpecializeLhs(λ, i, sim[i]) then
13                 λ' ← specializeLhs(λ, i, sim[i]);
14                 if ρ_j > λ'_j then
15                     addIfMin(φ(λ', ρ_j), Φ);
16 return Φ;
```

Suppose Φ is the set of all MD candidates that we assume to be true and that we are given a record pair (r, s) that was not yet considered. We can reduce Φ by invalidating all $\varphi \in \Phi$ that have an Lhs decision boundary subset $(\forall \lambda_i \in \lambda : \lambda_i \leq \text{sim}_i)$ but a larger Rhs decision boundary $(\rho_j > \text{sim}_j)$. For example, record pair $(1, 3)$ in Table 3 matches and invalidates the six MD candidates $\emptyset \to C_{1.0}$, $A_{0.7} \to C_{1.0}$, $A_{1.0} \to C_{1.0}$, $B_{0.8} \to C_{1.0}$, $A_{0.7}B_{0.8} \to C_{1.0}$, and $A_{1.0}B_{0.8} \to C_{1.0}$, which are shown blue in Figure 1. Once an MD candidate has been invalidated by some record pair, we can infer that this MD and all generalizations are invalid; the MD's specializations, however, can still be valid and are potentially also minimal. These are those MDs that are closest to the prior MD but not yet violated. To reduce Φ, we therefore remove every invalidated φ and add its nearest, still valid and minimal specializations w.r.t. the search space lattice of MD candidates.

Algorithm 2 describes the entire MD inference process: Analogously to the lattice traversal approach, we first initialize the lattice with the most general MDs (line 2). Then, the algorithm iterates all pairs of records in the input relation (line 3). We later introduce a *sampling* strategy that avoids comparing all record pairs. For each selected pair of records (r_k, s_l), the algorithm calculates the similarities sim for all column matches C (line 4). With these similarities, the algorithm searches the lattice for all violated MDs as explained above (line 5); in Section 5, we present an implementation of this operation. Any violated MD is first removed from the lattice (line 7) and then used to infer new MD candidates, which are specializations of the current MD, by either lowering the Rhs decision boundary (lines 8 – 10) or increasing some Lhs decision boundary (lines 11 – 15).

Since an MD with the same Lhs as the current can hold for a weaker Rhs, we lower the Rhs decision boundary to the similarity of the records in the respective columns. If this specialization is minimal, it is added to the lattice (line 10). To generate the other specializations, we specialize the Lhs so that the current violating record pair does not match it anymore. For this purpose, we

increase the decision boundary for each Lhs column match to the next possible value above the similarity of the records in the respective columns (line 13). If a new MD candidate, consisting of the specialized Lhs and the old Rhs, is minimal and non-trivial, the algorithm adds it to the lattice (line 15). We retain the Rhs decision boundary, because it was inferred from another record pair that an MD on this path must use to be minimal; if a higher Rhs decision boundary is in fact possible, it is inferred by a different MD on another path through the lattice. The algorithm terminates after processing all pairs of records; Φ then contains all valid, minimal MDs.

To exemplify this approach, we use the data shown in Tables 2 and 3 and infer the minimal MDs according to Figure 1. The initial MD candidate is $\emptyset \rightarrow C_{1.0}$. Pair $(1, 2)$ matches this MD and we, therefore, infer its specializations: Because $(1, 2)$ has a similarity of 0.0 in the columns of C, no non-trivial MD with $\emptyset$ as the Lhs is inferred. However, we infer $A_{0.7} \rightarrow C_{1.0}$ and $B_{1.0} \rightarrow C_{1.0}$. $(1, 3)$ violates the former and we infer $A_{0.7} \rightarrow C_{0.5}$. $A_{0.7}B_{1.0} \rightarrow C_{1.0}$ is also inferred but not minimal. $(1, 4)$ violates $A_{0.7} \rightarrow C_{0.5}$ and, hence, $A_{0.7}B_{0.8} \rightarrow C_{0.5}$ is inferred. From thereon, no candidates are invalidated anymore by any record pair and the minimal MDs $B_{1.0} \rightarrow C_{1.0}$ and $A_{0.7}B_{0.8} \rightarrow C_{0.5}$ are discovered.

5 A HYBRID APPROACH: HyMD

The two MD discovery approaches, inference from record pairs and lattice traversal, both have their strengths and weaknesses: While the former scales better with the number of attributes, the latter scales better with the number of rows in the input relation. Both approaches are, still, quite similar w.r.t. their general search model: They both start with the most general assumption, which is that every possible MD holds, and then gradually specialize this assumption. Lattice traversal does this by validating the candidates step by step making only small but continuous progress in the lattice. Inference from record pairs, on the other hand, checks records pair-wise for MD violations that may invalidate many candidates in the lattice at once.

We now combine the two approaches in a hybrid algorithm called HyMD, which leverages the advantages of both search techniques and mitigates their weaknesses. For this to work, we need efficient data structures that can be used by both approaches and a strategy to switch between the two approaches that guarantees mutual support. In the end, HyMD shall output all minimal MDs.

Figure 2 gives an overview of the data structures and components of HyMD. The *Plis* and *dictionary compressed records* represent a compressed version of the data and the *similarity matrix* and *similarity index* data structures capture similarities between different data values. The *lattice* data structure is a special prefix tree for MDs and holds the current MD candidates. The components *traversal* and *validation* implement the level-wise lattice traversal and the components *sampling* and *inference* implement an adaption of the inference from record pairs technique. The arrows indicate the interplay between the five components and access to certain data structures by certain components.

In the following, Section 5.1 first introduces the four upper data structures and the preprocessing step of HyMD that creates them. Section 5.2 then proposes the lattice data structure for handling intermediate MD candidates. Finally, Section 5.3 discusses the hybrid discovery and the use of the different data structures in detail.

5.1 Preprocessing

Before HyMD starts the hybrid discovery, it transforms the two input relations r and s into four compact, static data structures that are kept in memory during MD discovery. This transformation is done in a preprocessing step:

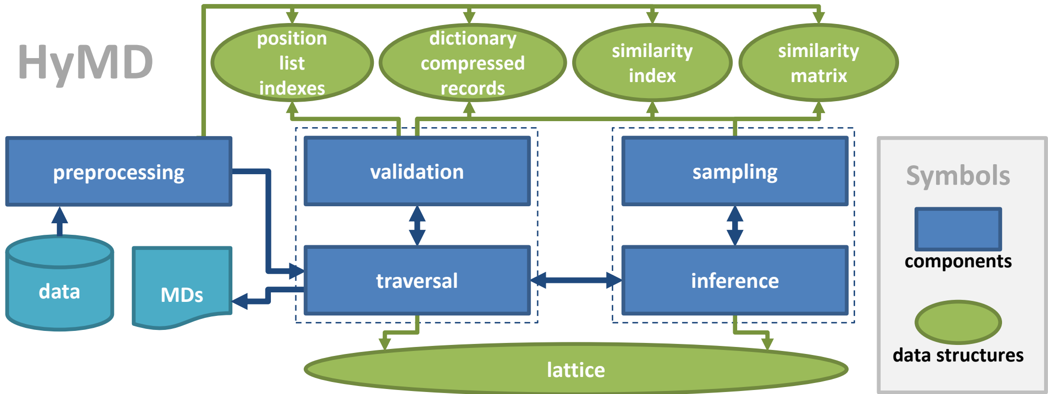

Fig. 2. Overview of the components of HyMD

Position list indexes (Plis). The input data is, first, transformed into inverted indexes that point for each column and every distinct value of that column to the list of positions at which this value occurs. These indexes, denoted with π, are called *position list indexes* (Plis) or *stripped partitions* in the literature [8, 18, 27]. For their construction, the preprocessing reads the input data record-wise adding each value-to-position mapping to the Pli of the corresponding attribute. Because the dependency discovery requires only the lists of positions that refer to the same values and not the actual values themselves, we retain only these lists in the Plis. With respect to our example data in Table 2, the Plis are $\pi_A = \{\{1\}, \{2\}, \{3\}, \{4\}\}$, $\pi_B = \{\{1\}, \{2, 3\}, \{4\}\}$, and $\pi_C = \{\{1\}, \{2, 4\}, \{3\}\}$; the index of each cluster serves as an implicit *value identifier* and is, therefore, used synonymously for its value in the algorithm.

Dictionary compressed records. While constructing the Plis record by record, HyMD also compresses these records via dictionary encoding into compact representations [31]. The compression process uses the Plis as dictionaries, i.e., to substitute each value with its cluster number in the corresponding Pli. The resulting *dictionary compressed records* are needed later on for record comparisons and MD candidate validations. The data structure is implemented as an array of record arrays that looks like the relation depicted in Table 2 but with value identifiers a'_i, b'_i, and c'_i instead of actual values a_i, b_i, and c_i.

Similarity matrices. The discovery of MDs requires the calculation of similarities between the various values of a column match C_i. Because these similarities need to be calculated repeatedly for different MD candidates, and because their calculation is expensive, we propose to pre-calculate and store all necessary similarities in *similarity matrices*: one matrix of similarities for every given column match C_i. The two columns of a column match define the two dimensions of that matrix: The x-dimension of the matrix are the value identifiers of one column and the y-dimension are value identifiers of the other column of the column match (note that both columns can be the same). The value identifiers represent unique values in each column and are derived from the Plis calculated earlier. Each cell in the matrix stores the similarity of two values w.r.t. the similarity measure $\approx_i$ of C_i. Figure 3 shows two such similarity matrices for our running example of Table 2. Considering the column match C_2, the similarity of the values with identifier b'_1 and b'_2 can, in this way, easily be looked-up as 0.8.

C_1	a'_1	a'_2	a'_3	a'_4
a'_1	1.0	0.0	1.0	1.0
a'_2	0.0	1.0	0.7	0.7
a'_3	1.0	0.7	1.0	0.7
a'_4	1.0	0.7	0.7	1.0

C_2	b'_1	b'_2	b'_3
b'_1	1.0	0.8	0.0
b'_2	0.8	1.0	0.0
b'_3	0.0	0.0	1.0

Fig. 3. Two similarity matrices for the column matches $C_1 = (A, A, \approx_{levenshtein})$ and $C_2 = (B, B, \approx_{jaro})$.

To calculate the similarity matrices, HyMD requires the actual data values for the similarity calculation and the value identifiers for the matrix construction. For this reason, it calculates the similarity matrices *after* the Pli construction but *before* throwing away the keys of the Plis.

Because the similarity calculations are both expensive and independent of one another, the preprocessing algorithm parallelizes them. Furthermore, we check all values on equality before executing a complex similarity function, because many values are not only similar but in fact equal. In the worst case, though, $O(|r| \times |s|)$ similarities need to be computed for every column match C_i. If the input relations are too large to compute all similarities, we propose the use of partitioning techniques, which compute the similarities for only those value pairs that are actually similar. The similarity matrix then stores a similarity of 0.0 for all value pairs that were not compared. A well-known partitioning technique in record linkage and duplicate detection research [6, 9] is the *sorted neighborhood method* [17], which compares each value to its most similar neighbor values w.r.t. prefix similarity. With our minimal decision boundary threshold, though, the discovery algorithm knows exactly which similarities are relevant for each column match, i.e., all similarities above this threshold. For this reason, we propose the use of a more sophisticated string similarity search approach that is able to calculate all similarities larger than the given minimal decision boundary threshold, such as *Ed-Join* [39], *Pass-Join* [23], or *FixPrefixScheme* [36]. Note, however, that the applicability of a partitioning approach depends on the similarity measure. The partitioning is also an approximation technique that might prevent the discovery of all MDs, which is why we do not use it in our experiments and always calculate *all* similarities.

To keep the size of the similarity matrices small, we consider the interestingness criterions for MDs as presented in Section 3.4: Similarities that are both below the minimal Lhs and Rhs decision boundary are not relevant to the discovery, so that the algorithm does not need to store them. *Any non-existent similarity is implicitly* 0.0. Not storing these similarities makes the matrices sparse so that we can save much space representing the matrix as an array of hash maps, i.e., a continuous sequence of x-value identifiers (array-indexes) holding y-value identifiers (map-keys) that point to similarities (map-values).

Similarity index. The similarity matrix helps to determine the similarity for a given pair of values w.r.t. their column match. During MD discovery, however, another operation frequently retrieves for a given value identifier all record IDs with at least some specific similarity. To support this operation, we propose an inverted index. This index is called *similarity index*. It points any value identifier (of r) to all similarities of that identifier, and then each similarity to the list of record IDs (of s) with at least that similarity to the initial value identifier – in this way, the similarity index basically inverts the value-to-similarity mapping of the similarity matrix. It is, hence, calculated from the similarity matrices. Figure 4 shows the similarity index for the column match C_2, whose similarity matrix is depicted in Table 2. To retrieve all records with at least 0.8 similarity to the value identified by b_1, we first lookup b_1, then 0.8 in the resulting map, and finally retrieve the record IDs 1, 2, and 3.

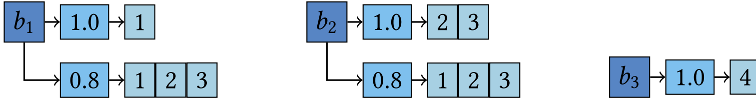

Fig. 4. Flat similarity indexes

5.2 Lattice

The lattice stores all current MD candidates in a compact format and offers certain operations that help to, for instance, efficiently retrieve generalizations of violated candidates. For this purpose, it is implemented as a prefix tree: The root node represent the empty set, all further nodes represent column matches, paths from the root to any node in the tree represent LHSs of MDs, and annotations attached to the nodes represent valid RHSs column matches of MDs with that node specific LHS path. By uniquely ordering the LHS nodes on each path (see Section 3.3 for ordering rules), any path uniquely identifies its MDs. A similar data structure was proposed for storing FDs [14, 27].

The initial prefix tree in HYMD contains only the root node without any children and, as annotation, all column matches with $\lambda_i = 1.0$. It then grows with every specialization of its candidates. In the following, we discuss the supported operations:

`add` inserts an MD into the lattice by traversing a path in the prefix tree according to the new MD's LHS. If a node on the path does not exist, it is created. At the end of the path, the operation adds the MD's RHS as an annotation.

`addIfMin` inserts an MD into the lattice using the `add` operation if this MD is minimal, i.e., no generalizations exist in the lattice. To check for generalizations, the operation applies a depth-first tree search for subsets of the MD's LHS.

`findViolated` retrieves all MDs from the lattice that are violated by a given record pair, i.e., MDs whose LHS matches this record pair and whose RHS is higher than the respective similarity of the records. For this purpose, the operation applies a depth-first search for matching LHSs and checks *all* RHSs on the way reporting those that are violated.

`getLowerBoundaries` determines the maximum RHS decision boundaries for any generalization of a given MD. Hence, it also traverses all generalizations of the MD's LHS via depth-first search capturing the maximum decision boundary for each RHS column match across all visited nodes. If all lower boundaries are 1.0, we abort the operation, because these boundaries cannot become higher anymore.

`getLevel` retrieves all MDs of a certain level. The level can be defined as either its depth (as we did in Section 4.1) or its cardinality (as we do in Section 5.3). Regardless of this definition, the operator uses a depth-first search to collect all nodes with exactly the specified depth or cardinality.

5.3 Hybrid Discovery

Having introduced the two search strategies as well as their data structures, we now present the hybrid algorithm HYMD that combines them. The components of HYMD have already been shown in Figure 2: After the preprocessing, HYMD proceeds to the inference strategy in the sampling component. Rather than selecting all records right away, the sampling component selects the record pairs successively. With each selected record pair, the inference component derives non-MDs that are used to specialize the MD candidates in the lattice. When the sampling becomes inefficient, it switches to the traversal component that selects candidates from the lattice (level-wise, bottom-up). These candidates are validated by the validation component and updated in the lattice accordingly. After each round of validations, HYMD continues with the inference strategy. To improve the

efficiency of this phase, the traversal strategy collects record pairs as inference recommendations for the inference strategy. By exchanging MD candidates and inference recommendations, both phases support one another.

5.3.1 Inference from record pairs. In comparison to its archetype of Section 4.2, the inference from record pairs algorithm changes only slightly when used in the hybrid setup. The first two changes are simple: First, we do not initialize the set of minimal MDs Φ with every method call but reuse and share the same set of MDs with the lattice traversal phase. Second, we add one more parameter that transports the inference recommendations from the traversal phase to this inference phase; these recommendations are checked before we enter the main loop in line 3 of Algorithm 2.

The third change counts the number of record pair *comparisons* (r_k, s_l) (line 3) and the number of *refined* MDs φ, which are those that could actually be taken from Φ (line 7). The quotient of these two numbers, i.e., *refined*/*comparisons* decreases over time since ever fewer refinements are to be discovered. It therefore reflects the efficiency of the inference-based search and triggers a phase switch after processing some record pair, if it falls below a certain threshold. The authors of HyFD have shown that for hybrid FD discovery any efficiency threshold between 0.1% and 10% works well, because the efficiency drops quickly [27]; we observed the same for MD discovery and, hence, start with a threshold of 1%. With every switch into the inference phase, HyMD halves this threshold to become efficient again.

The fourth and final change is that the algorithm avoids processing redundant record pairs, which are record pairs with the same set of similarities as previously processed record pairs. Because redundant pairs produce the same invalidations, HyMD remembers the unique sets of similarities that were processed to not process them again.

5.3.2 Lattice traversal. The lattice traversal strategy used in HyMD is an extension of the level-wise, bottom-up traversal algorithm of Section 4.1. The extensions are primarily geared towards letting this algorithm switch to and from the inference strategy in the hybrid setting, but we also contribute some optimizations that are independent of the hybrid execution strategy.

The inference phase of HyMD is responsible for initializing the set of minimal MDs Φ, which is the search space lattice. Since Φ is already calculated by the inference phase that always executes first, we remove line 2 in Algorithm 1 and make Φ a parameter.

As mentioned earlier, HyMD switches from the traversal to the inference phase whenever one level of candidates has been validated. In contrast to [27], we do not measure the efficiency of the traversal phase and switch more pro-actively, because if the switch was not needed, the inference phase returns quickly anyway. To implement the switch, we replace the while-loop in line 4 with an if-statement and switch to the inference phase at the end of the algorithm.

To support the inference phase after a switch, the traversal phase collects possibly many inference recommendations, which are pairs of records (r_k, s_l) that invalidated some MD candidate. These record pairs have not been compared before, because otherwise the candidate would not have been checked. It is also likely that they affect other candidates in the same way, which makes their inspection promising. Because the validation function in line 8 finds these pairs of records naturally, HyMD can simply collect them. The collected record pairs are, then, handed over to the inference phase as comparison suggestions with every phase switch.

The function getLevel() in line 5 of Algorithm 1 retrieves all MD candidates of a specific level from the search space lattice. Since all retrieved candidates are independent of one another, we can validate them in parallel, i.e., parallelize the for-loop in line 5 to multiple threads. The inference strategy, however, reduces the number of candidates in each level significantly as it prunes large portions of the lattice. Because this lowers the effectiveness of parallelization, we redefine the notion of a lattice level *for the hybrid approach*: Instead of the *depth*, we use the *cardinality* (see

Algorithm 3: Validation

Data: Decision boundaries λ, ρ. PLIS of r $\pi[r]$. Relational instances r, s. Column matches C. Maximal LHS and minimal RHS boundaries λ_{lower}, ρ_{min}. Similarity indexes simIndex.
Result: Maximal decision boundaries ρ'. Support σ.

```
1  ρ', σ ← ρ, 0;
2  if |λ| = 0 then
3      return getMinSims(ρ, simIndex), |r| · |s|;
4  if |λ| = 1 then
5      foreach value, r' ∈ π[r][i] do
6          s' ← getSimRecs(value, λ_i, simIndex[i]);
7          σ ← σ + |r'| · |s'|;
8          foreach r_k, s_l ∈ r' × s' do
9              ρ' ← computeMaxRhs(r_k,s_l,ρ',λ,λ_lower,ρ_min);
10 else
11     foreach rep, r' ∈ groupByValue(π[r], λ, C) do
12         s' ← getSimRecs(rep[j], λ_j, simIndex[j]);
13         foreach λ_i ∈ λ \ λ_j do
14             s' ← s' ∩ getSimRecs(rep[i], λ_i, simIndex[i]);
15         σ ← σ + |r'| · |s'|;
16         foreach r_k, s_l ∈ r' × s' do
17             ρ' ← computeMaxRhs(r_k,s_l,ρ',λ,λ_lower,ρ_min);
18 return ρ', σ;
```

Section 3.4) of the MDs as the function to determine their level, because the number of same cardinality candidates is larger than the number of same depth MDs. The lattice in Figure 1, for instance, has 2 MD candidates of distance one (see LHSS $A_{0.7}$ and $B_{0.8}$) and 4 MD candidates of cardinality one (see LHSS $A_{0.7}$, $A_{1.0}$, $B_{0.8}$, and $B_{1.0}$) – note that only the LHSS count; all RHSS are calculated dynamically. With cardinality as level function, one level might contain candidates that specialize others, such as $A_{0.7} \rightarrow C_{1.0}$ and $A_{1.0} \rightarrow C_{1.0}$. To discover only minimal MDs, we need to validate all generalizations before their specialization. Therefore, HYMD first analyses each level for specializations by checking the candidates pair-wise to then defer the validation of specializations. So in our example, the LHS $A_{0.7}$ is finished before $A_{1.0}$ and $B_{0.8}$ is finished before $B_{1.0}$. Still any other combination of LHSS, such as $A_{1.0}$ and $B_{0.8}$, can be evaluated in parallel. Despite the overhead of specialization analysis within each level, the gain in parallelization outweighs these costs, as we show in Section 6. Many candidates, such as $A_{1.0} \rightarrow C_{1.0}$ and $B_{0.8} \rightarrow C_{1.0}$ in Figure 1, for instance, are independent of each other but have different depths.

Before validating the set of all (specialization-free) MD candidates in parallel, we group them by their LHS λ. This provides us with a set of RHSS ρ that HYMD can validate simultaneously: The algorithm first retrieves all record pairs matching λ and then checks all ρ against these records. In other words, it calculates the maximum RHS decision boundaries ρ' for the LHS decision boundaries λ so that the traversal algorithm can check these ρ'. Algorithm 3 shows this validation algorithm in detail. It provides three different validation approaches depending on the MD's cardinality, which is either 0, 1, or larger.

MDs with a cardinality of 0 have all Lhs decision boundaries set to 0.0 (line 2). They are matched by all record pairs and, therefore, have a support σ of $|r| \cdot |s|$. The decision boundaries ρ' of these MDs are the minimal similarities in the respective Rhs column matches and can be retrieved with one linear scan of the similarity index (line 3).

For MDs with cardinality 1 (line 4), there is only one Lhs column match C_i to be considered. So HyMD starts by iterating all values and their positions r' in r using the Pli $\pi[r]$ of this column match, i.e., with index i (line 5). For each r-value, Algorithm 3 collects all positions s' of s-values that are similar to the r-value w.r.t. the Lhs decision boundary λ_i using the similarity index, i.e., it selects the records that have a similarity greater than or equal to the sole Lhs decision boundary (line 6). Since all records in r' and s' match pair-wise, $|r'| \cdot |s'|$ is added to the support σ (line 7). HyMD then compares all records r_k and s_l in $r' \times s'$ w.r.t. their Rhs similarities and stores the *minimal* similarities in ρ'; these minimal similarities constitute the maximal Rhs decision boundaries for the validated MD candidates (line 9).

If the input MDs have a cardinality > 1 (line 10), Algorithm 3 cannot directly read the values of r from its Plis. It therefore groups the r-records by their Lhs value combinations on the fly by intersecting the Plis of all Lhs column matches (line 11). Each cluster in the resulting Pli defines a representative value combination rep that the algorithm requires to find similar value combinations in s. To find all positions s' of s-records that are similar in *all* Lhs column matches, HyMD collects for each individual Lhs column match the positions of similar records w.r.t. this column match using the similarity index and intersects all these sets (line 12–14). The sets r' and s' of matching r- and s-record positions are then used to update ρ' and σ just like in the cardinality-1-case (lines 15–17).

Even with the changes explained above, all MDs discovered by Algorithm 1 are valid and minimal. For this reason, HyMD ends in the lattice traversal phase when all MD candidates have been processed yielding Φ as the final result.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate various aspects of our algorithm HyMD: We first measure its runtime on different datasets (Section 6.1); then we measure the impact of some major implementation details and configurations on the algorithm's runtime and the result size (Section 6.2); next, we investigate HyMD's scalability for increasing numbers of column matches and rows (Section 6.3); we also compare our algorithm to the FD discovery algorithm HyFD [27] (Section 6.4); in the end, we briefly evaluate and discuss the usefulness of the discovered MDs in the context of duplicate detection (Section 6.5).

Experimental setup. HyMD is implemented in Java 1.8 and based on open-source data profiling tool Metanome [26]. The code for HyMD is available online[1]. We ran all experiments on an OpenJDK 64-bit Server 1.8.0_151 JVM and CentOS 6.8 64-bit. The hosting machine features 128 GB RAM and two Intel Xeon E5-2650 2 GHz CPUs; we use all 16 hyperthreaded cores for parallelization.

Default configuration. If not stated differently, we use the natural decision boundaries for each column match and set the minimal Lhs and Rhs decision boundaries to 0.7; the minimal support is set to $|r| + 1$ if $r = s$ and 1 otherwise. For runs on a single relation, all columns are matched to themselves; for two relations, we use fixed schema mappings. Our default similarity measure is the Levenshtein similarity and HyMD is configured to calculate all similarities for each column match, i.e., no approximation is used.

[1]https://github.com/HPI-Information-Systems/metanome-algorithms

dataset	cols [#]	rows [#]	size	lhs [#]	MDs [#]	pre [sec]	disc [sec]	execution
adult	15	32,561	3.5 MB	10^5	91	2.0	12.8	14.8 sec
restaurant	6	864	63.0 KB	10^6	7	1.1	0.7	1.8 sec
VTTS	14	10,042,044	588.0 MB	10^6	76	269.8	11,957.7	3.4 h
NCVoters*	12	466,388	45.7 MB	10^8	472	13.9	133.5	2.5 min
NCVoters	12	32,413,515	3.1 GB	10^9	435	1,627.7	32,338.2	9.5 h
hospital*	10	9,342/4,830	2.7 MB	10^9	188	9.5	3.0	12.5 sec
HGI	14	4,830	0.7 MB	10^{10}	289	10.0	5.3	15.3 sec
PCM	14	9,342	2.0 MB	10^{10}	557	9.3	6.4	15.7 sec
CORA	16	1,879	367.0 KB	10^{12}	2,001	5.6	379.4	6.5 min
flight	38	1,000	187.0 KB	10^{12}	14,170	0.9	65.4	1.1 min
hospital	134	9,342/4,830	2.7 MB	10^{22}	3,544	30.3	83.5	1.9 min

Table 5. Performance of HYMD on various datasets

6.1 Datasets and Runtimes

Table 5 lists the seven datasets[2] that we use for our experiments: *adult* with general information about people, *restaurant* describing US restaurants, *VTTS* with ERP data from an SAP R3 system, *NCVoters* about voting preferences in North Carolina, *hospital* consisting of the *Hospital General Information* (HGI) relation and the *Preventive Care Measures* (PCM) relation both providing information about hospitals in the US, *CORA* with bibliographic information, and finally *flight* with details on US flights.

For *hospital*, we use a hand-crafted schema mapping for creating the column matches, because both tables of the *hospital* dataset contain records about hospitals and the attributes have clear correspondences. For the *CORA* dataset, we use Monge-Elkan instead of Levenshtein similarity and we also limit the number of decision boundaries for each column match to 5 instead of using all natural decision boundaries, because *CORA* contains particularly many different and very similar values; the number of *CORA*'s possible LHSs is still one of the largest.

The number of possible MD LHSs in Table 5 shows that the search spaces are huge and that the discovered MDs are only small fractions of them. Considering the fact that we find on average $\frac{column}{2}$ MDs for every possible LHS (by simply selecting the largest RHS threshold with which the MD is still true), the numbers of actually discovered MDs are tiny. This shows that our interestingness criteria, which are only minimum decision boundary, (number decision boundaries,) and disjointness in this experiment, are effective. Using stricter interestingness criteria values than those we used in our experiments is realistic for many MD use cases and reduces the number of discovered MDs significantly further. For example, increasing the minimum decision boundary to 0.9 and the minimum support to 1.05 times the relation sizes reduces the number of discovered MDs from 3,544 to 320 MDs on *restaurant* and from 14,170 to 2,295 on *flight*. After their discovery, the MDs can be ranked also by the interestingness criteria to show only the most relevant ones to a user.

The two columns *pre* and *disc* list HYMD's runtime, broken down to preprocessing time (pre) and discovery time (disc). They show that the preprocessing time, which is in particular the calculation of the similarities, actually dominates the discovery time for datasets with many different values and MDs that are easy to find by either of the two discovery strategies. The last column reports the overall execution time of HYMD.

[2]https://hpi.de/naumann/projects/repeatability/data-profiling/mds.html

Despite their relatively small size, the listed datasets present real challenges for MD discovery algorithms due to their large candidate search spaces. HyMD however could process all datasets with ease in a few seconds to minutes; only the *VTTS* and *NCVoter* dataset took longer than a few minutes, which is acceptable considering their much larger size of 588.0 MB and 3.1 GB, respectively. The *flight* and *hospital* dataset have not only the most column matches, they also contain the most MDs of all datasets; their processing times are still manageable, because the inference-based discovery works very well on only a few thousand records. The adult, *VTTS*, and *NCVoter* dataset, on the other hand, are particularly long, but due to their small number of column matches and MDs, the traversal-based discovery processes them efficiently.

The measurements in Table 5 also show that the number of discovered MDs can be very large. In the only 187 KB *flight* dataset, for instance, HyMD discovered more than 14 thousand MDs, which are way more MDs than a data analyst could process manually. Fortunately, some algorithms that consume MDs to solve certain use cases can process the MDs automatically. For use cases that require a manual analysis of the discovered MDs, such as data exploration, we suggest to define stricter interestingness pruning thresholds. In our experiments, we defined overly conservative thresholds to show what the algorithm is capable of. By demanding, for instance, a maximum cardinality of three attributes, a minimum support of 1.5 times the dataset size, a minimum decision boundary of 90%, and a maximum number of decision boundaries of 100, all result-sets shrink to less than 100 MDs.

Many of the discovered MDs are not only syntactically correct but also semantically interesting. While syntactic correctness means that an MD is valid for some relational instance, semantic interestingness is given if the described rule has a true meaning w.r.t. the relation's domain. Because MDs can be syntactically correct by chance and due to a lack of counter examples in the data, not all discovered rules are semantically interesting. Many of them, however, are interesting. The MDs listed in Table 6, for example, describe rules that we can intuitively agree to: Hospitals with matching `address` and `phone` number should also match in their `names` (φ_1), restaurants with matching `name` and `address` should also match in their `phone` numbers (φ_2), and publications with matching `title` and `booktitle` should have been written by matching `authors` (φ_4). Furthermore, MD φ_3 teaches us that we can add the `type` of a restaurant to rule φ_2 to better distinguish similarly named restaurants at similar addresses. The pruning rules introduced in Section 3.4 help HyMD to focus the discovery process on exactly such *interesting* MDs, as they usually have low cardinality, high support, disjoint Lhs and Rhs sides, and high decision boundaries. For our performance measurements, however, we reduced the interestingness pruning to a minimum according to Section 3.4 in order to see HyMD's worst-case runtimes; this let to the result counts listed in Table 5.

	dataset	**MD**
φ_1	hospital	$\texttt{address_1}_{0.86}\ \texttt{phone}_{1.0} \rightarrow \texttt{name}_{0.95}$
φ_2	restaurant	$\texttt{name}_{1.0}\ \texttt{address}_{0.98} \rightarrow \texttt{phone}_{0.85}$
φ_3	restaurant	$\texttt{name}_{0.7}\ \texttt{address}_{0.7}\ \texttt{type}_{0.71} \rightarrow \texttt{phone}_{0.85}$
φ_4	CORA	$\texttt{title}_{0.81}\ \texttt{booktitle}_{0.96} \rightarrow \texttt{author}_{0.33}$

Table 6. Some *interesting* MDs discovered by HyMD

6.2 In-depth Experiments

In this paper, we proposed to apply the hybrid dependency search strategy to the discovery of MDs. Because the search space for MDs is significantly larger than the search space for FDs and the lattice traversal is accordingly more expensive, it is not obvious that the hybrid search is actually

effective in this setting. For this reason, we now evaluate the three search approaches, which are traversal, inference, and hybrid, separately. We also evaluate the level function and the disjointness pruning in detail as both are important for the performance of the lattice traversal. In a fourth and final in-depth experiment, we evaluate the impact of choosing different similarity functions on HyMD's runtime and the number of discovered MDs.

Hybrid search. HyMD combines lattice traversal with inference from record pairs. Because both strategies are able to discover all MDs on their own, we compared their performances with the performance of their hybrid combination: On *restaurant*, we measured the three discovery times 37.0 sec (traversal), 1.2 sec (inference), and 0.7 sec (hybrid); on *hospital*, we measured 65.1 sec (traversal), 11.7 sec (inference), and 2.8 sec (hybrid); and similar ratio for the other datasets. Because lattice traversal scales poorly with the number of decision boundaries, we limited their number for this experiment to 10 and 5, respectively. Since both datasets are relatively short, the inference strategy clearly outperforms the traversal strategy. The hybrid strategy, however, outperforms both individual approaches – even the optimal approach for the given dataset – due to its synergy effects.

Level function. Section 5.3.2 raised the subject of using different level functions to retrieve MD candidates for validation: depth and cardinality. With cardinality, more candidates can be validated in parallel, but an additional step that orders the candidates becomes necessary. The level function in HyMD is, therefore, configurable. In our experimental setup, we measured the discovery times 4.6 sec (depth) and 3.0 sec (cardinality) on *hospital*, 13.6 sec (depth) and 12.8 sec (cardinality) on *adult*, and further similar runtimes in favor of cardinality on the other datasets. The increased degree of parallelization, which we achieved on 16 cores, easily compensates the additional ordering costs.

Disjointness pruning. In Section 3.2, we discussed non-disjoint MDs and proposed to prune them from the search space although they are not necessarily trivial. When switching the disjointness pruning in HyMD off, more MDs can be found, at the cost of discovery time: On *hospital* we find 188 MDs in 3.0 sec (on) or 974 MDs in 18.7 sec (off); and on *adult*, we find 91 MDs in 12.8 sec (on) or 138 MDs in 52.7 sec (off). Hence, the discovery times increase faster than the result counts.

Similarity functions. HyMD can use any similarity function that can effectively compare the values of the given column matches. The algorithm also allows to mix similarity functions and use different functions for same column pairs. For this reason, we now evaluate the impact of the chosen similarity functions on the discovery time and number of discovered MDs. For this experiment, we compare the two similarity functions Levenshtein and Monge-Elkan on *NCVoters* and *CORA*. The results are shown in Table 7.

Because calculating Monge-Elkan similarities is more expensive than Levenshtein similarities, the pre-processing time takes longer with this similarity function. The discovery, however, uses the pre-calculated similarities and is, therefore, not affected by the costs of the chosen similarity functions. It is, though, affected by the number of MDs that can be discovered with these similarities. Monge-Elkan is a hybrid similarity function that also uses token similarity. It is, hence, less sensitive against character conversions than Levenshtein and calculates higher similarities for various values. The number of true MDs w.r.t. Monge-Elkan is therefore much higher, which impacts the discovery time clearly. This impact is, in particular, much larger than the impact caused by the calculation costs of the similarity function. So in summary, the more values a similarity function can match, the higher the MD discovery costs become.

dataset	function	MDs [#]	pre [sec]	disc [sec]	execution
NCVoters*	Levenshtein	472	13.9	133.5	2.5 min
	Monge-Elkan	1,219	20.9	382.2	6.7 min
CORA	Levenshtein	997	2.1	145.7	2.5 min
	Monge-Elkan	2,001	5.6	379.4	6.5 min

Table 7. Performance of HyMD with different similarity functions

6.3 Scalability

Three properties of input datasets influence the runtime behavior of HyMD: their number of records, column matches, and decision boundaries. With growing volumes of data, usually all three numbers increase. For this reason, we investigate HyMD's runtime when scaling up each of them on the *CORA* and *NCVoters** dataset.

Number of records. We start for both datasets with a 10% random sample of records and add further 10% records with every measurement step. The number of column matches and decision boundaries is fixed to the corresponding numbers in the last slice to scale only the number of records. By adding further records to the dataset, the number of minimal MDs can both increase or decrease: It increases if the added records introduce MD violations. It decreases if candidates are pushed into lattice regions with fewer candidates.

Figure 5 shows the result of this experiment. We see that different forces either favor or impede HyMD's runtime although only the number of records is changed: In general, more records add more values to the preprocessing and validation efforts, i.e., the similarity calculation effort increases, the index structures become larger, the inference needs to compare more record pairs, the traversal needs to proceed to more candidates on higher lattice levels, and the validations need to check more values. On the other hand, newly introduced MD violations might let the inference phase identify non-MDs faster and decrease the number of minimal MDs that need to be discovered (and checked).

For *NCVoters**, we first see that the runtime grows about linearly. The preprocessing time indeed grows quadratically, but the discovery time still dominates, i.e., we mainly see the candidate validation costs increasing. Some easy-to-validate MDs exist in the beginning and their vanishing does not effect the runtime; the vanishing of some larger MDs in the end, however, effects the runtime clearly.

For *CORA*, the preprocessing costs are negligible with less than 2% runtime. We still see the runtime increasing (10-40% and 70-100%) when MD candidates are pushed up higher in the lattice and become harder to evaluate; the runtime, however, also drops significantly when larger MDs vanish or when they become easier to infer.

It is, in summary, hard to predict HyMD's runtime based on the number of records. The trend, however, is that more records increase the runtime and reduce the number of MDs.

Number of column matches. We start for both datasets with two column matches and continuously add more until all are used. In practice, the number of column matches increases if the MD discovery is applied to datasets with more columns and it also increases if additional similarity functions should be considered. Adding column matches might introduce additional MDs, but it does not affect existing ones; the search space lattice grows exponentially and, on most datasets, the number of minimal MDs also grows exponentially.

Figure 6 shows the results of this experiment. As expected, the number of MDs increases roughly exponentially with the number of column matches for both datasets. Although the preprocessing

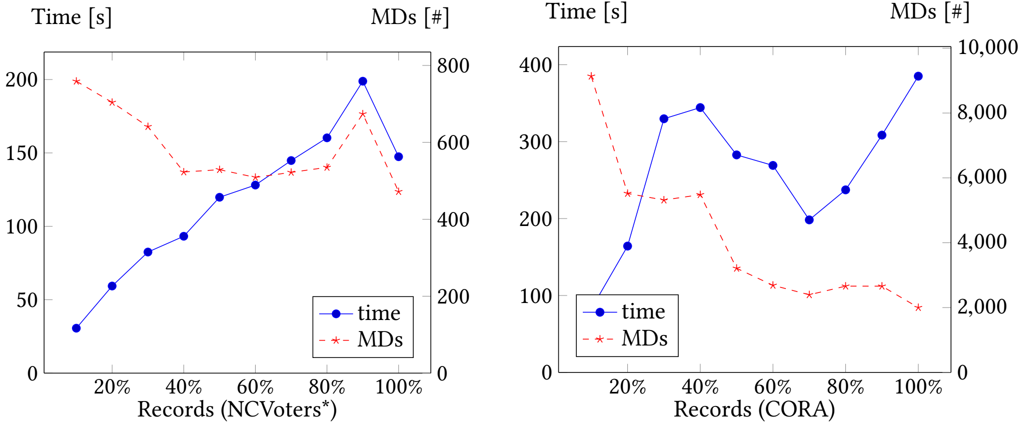

Fig. 5. Scaling the number of records

effort increases only linearly with every column match, the runtimes for both inference and traversal increase exponentially. Adding the column matches in different orders yields the same runtime curves. Hence, we can expect both runtime and number of MDs to increase exponentially w.r.t. the number of column matches.

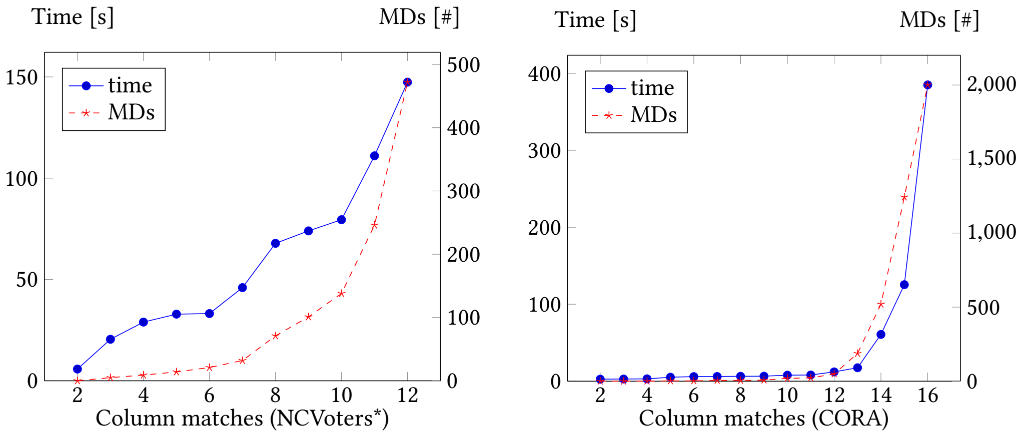

Fig. 6. Scaling the number of column matches

Number of decision boundaries. We select for each column match a number of uniformly distributed decision boundaries from its natural decision boundaries. This number is linearly scaled from one to five decision boundaries. By adding additional decision boundaries, the search space grows exponentially, because we basically add one step to the dimension of each attribute.

The results on *CORA* (Figure 7) confirm this theoretical consideration with an overall exponential runtime increase. Although the number of minimal decision boundaries grows even less than linear, the search space that HyMD needs to inspect grows exponentially. We do not see that exponential runtime behavior for *NCVoters**, because some of its column matches contain fewer than five decision boundaries and thus do not increase the search space size. Overall, with increasing number

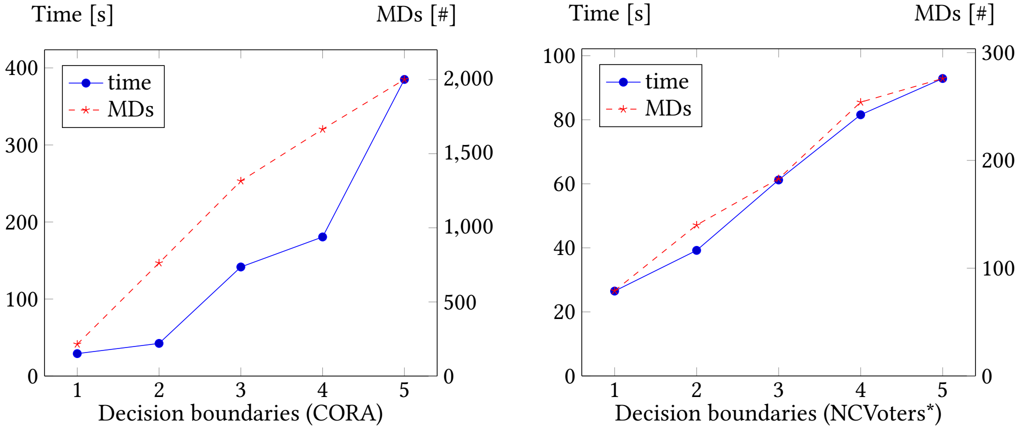

Fig. 7. Scaling the number of decision boundaries

of decision boundaries the runtime first increases exponentially and then converges to the runtime of using all natural decision boundaries of a dataset.

Minimum decision boundary. The minimum decision boundary threshold is another parameter that controls the number of decision boundaries used during MD discovery. In all previous experiments, we used a minimum decision boundary of 0.7 for all LHS and RHS column matches. Considering only 70% similar values as matches is rather unrealistic in most use cases but it serves to demonstrate the performance of our discovery approach. To show the algorithm's performance for higher minimal decision boundary thresholds, this experiment measures HYMD's discovery time for various values.

Figure 8 shows the results for *NCVoters* and *CORA*. We see that for both datasets increasing the minimum decision boundary reduces not only the discovery time but also the number of discovered MDs significantly. More specifically, increasing the threshold from 0.7 to 0.9 results in about 3 times faster discovery times on *NCVoters* and 23 times faster discovery times on *CORA*; the number of discovered MDs drops by 57.6% and 97.5%, respectively. Hence, the minimum decision boundary can be used to effectively reduce the discovery time and narrowed the search to MDs with higher similarity constraints.

6.4 Comparative Experiments

None of the MD discovery algorithms discussed in Section 2 discovers the same results as our algorithm HYMD for two reasons: First, they use pre-defined similarity thresholds for the candidate generation and not the similarity threshold that can actually be found in the data. For this reason, they do not find *all* MDs. Depending on how many similarity threshold are pre-defined, the candidate space can be smaller or larger than the actual candidate space that is given by the data; in any case, not extracting the actual threshold from the data saves some time. The second reason why algorithms from related work produce different results is that they also consider partial and conditional MDs. Depending on the search strategy and the data that is being profiled, this can also be an advantage or disadvantage for the performance: Validating partial/conditional dependencies is in general more costly, because more than one violation to an MD needs to be found (and resolved), but there may be fewer candidates to be validated, because valid, minimal partial/conditional MDs are smaller and occur earlier in the search space (assuming systematic bottom-up lattice search).

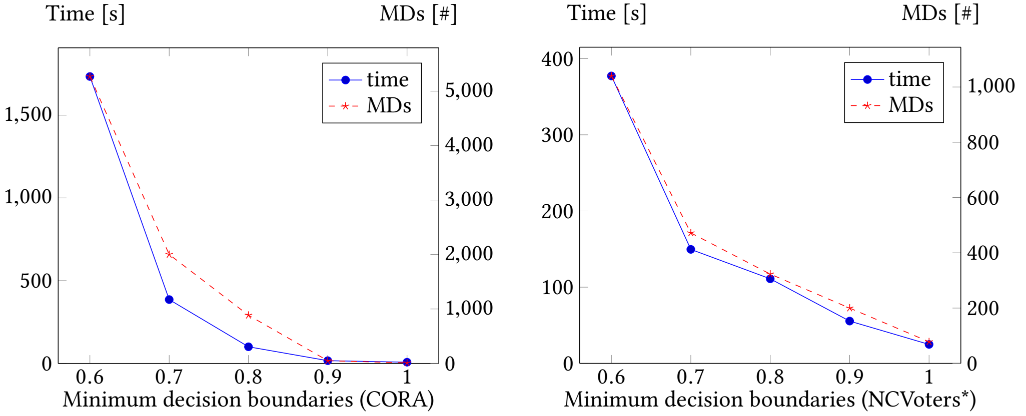

Fig. 8. Scaling the minimum decision boundaries

Because the results are so different and (in the case of related work) dependent on the algorithm configuration, comparing our profiling times to those of other MD discovery algorithms is largely inconclusive. The most obvious algorithm for a performance comparison with HyMD is the FD profiling algorithm HyFD, which implements the same hybrid search strategy.

Functional dependencies are a special case of matching dependencies where all columns are matched to themselves and only "=" is used as similarity function. Hence, HyMD can also discover functional dependencies with these two restrictions. To evaluate its capability as FD discovery algorithm, we test it on the row-heavy *adult* and the column-heavy *flight* dataset and compare the measured runtimes with the runtimes of HyFD.

Table 8 shows the result of this experiment: The restriction on FDs, in fact, improves the runtime of HyMD, because the similarity calculations are faster and fewer MDs are to be discovered. The overall runtime is, however, still clearly slower than HyFD's runtime, because HyMD's preprocessing, data structures, and validation strategies are optimized for checking MDs, which are more complex than FDs. It still performs this task reasonably well.

data	**MDs** [#]	**FDs** [#]	**HyMD** [sec]	**HyMD FD** [sec]	**HyFD** [sec]
adult	91	78	14.8	13.5	1.5
flight	14,170	6,811	66.3	15.9	1.6

Table 8. Comparison between HyMD and HyFD

6.5 Duplicate detection

In this section, we aim to illustrate the usefulness of the discovered MDs with a concrete example. From the multitude of use cases, we chose duplicate detection [10, 38] for this illustration, because it is one of the most popular and well-researched use cases for MDs. The purpose of this experiment is not to propose a novel, more effective duplicate detection approach, but to demonstrate in a real-world setup that the discovered MDs are actually useful. In a follow-up project [20], we developed a fully automatic duplicate detection system called `MDedup` that is based on the algorithm HyMD and the insights presented in this last evaluation. Duplicate detection processes identify similar representations of same real-world entities in relational datasets and are, therefore, common tasks

in data cleaning and record linkage. For the experiment, we use the *CORA* and *restaurant* datasets, because both provide a gold standard: *CORA* contains 64,578 duplicate record pairs and *restaurant* contains 112 duplicates.

To identify duplicates in data, domain experts usually define rules (or features to learn such rules) that should match two records if these records describe the same entity. The same is true for matching dependencies: They are rules that match similar records, which can be labeled as duplicates. The difference is, however, that MDs do not need to be defined manually as HyMD can automatically discover them. To test the usefulness of the discovered MDs w.r.t. the duplicate detection use case, we evaluate *precision*, *recall*, and f_1*-measure* for a selection of discovered MDs and compare the results with state-of-the-art machine learning approaches: a *support vector machine* (SVM) approach [4] and a *random forest* (RF) approach [3].

Various related works, such as [10] and [38], have used MDs for duplicate detection before, but they propose supervised learning approaches that rely on pre-labeled data to train the MDs. More specifically, these approaches require a special attribute that is used as Rhs for all MDs and that pre-labels duplicates on a training dataset. Our approach differs from their approach, because we can discover the MDs with HyMD from unlabelled raw data. The use of a dedicated target column that indicates whether two tuples are duplicates or not, would be a helpful hint for the MD discovery. But because we want to show that the MDs are useful also without this hint, we decided for a different duplicate detection approach: For the discovery, we use the default parameterization from our experiments. We then rank the MDs by cardinality (primary sort) and average decision boundaries (secondary sort), present the top 20 MDs to a user, and let her pick the rules for duplicate detection. The picking process is clearly a subject for future improvement, but it is feasible (in contrast to labelling thousands of training records) and – as we show later – sufficient to achieve reasonable results. For *restaurant*, the user picked φ_2 and φ_3 listed in Table 6 and, for *CORA*, she picked only φ_4. All record pairs matching these MDs are classified as duplicates.

To train and evaluate the baseline RF and SVM model, we create all pairs of records within the data, construct a similarity matrix as described in Section 5.1, and split the record pairs into training and test data. We then perform a 10-fold cross validation [19], which means that the data is divided into 10 equally-sized folds and each of these folds is used as testing once, while the remaining data forms the training data. The RF implementation is provided by the R package `randomForest`[3] and the SVM implementation by the R package `e1071`[4].

To ensure that the results are comparable, all three approaches, RF, SVM, and MD, use the same features: They consider all attributes apart from the `id`-columns, and Levenshtein similarity for each attribute. All three approaches are able to consider more similarity functions and would produce better results in that way, but this experiment aims to give a relative performance comparison under same pre-conditions rather than an absolute performance evaluation – we leave the development of a novel MD-based duplicate detection algorithm and its careful evaluation to future work.

For all three approaches, MD, RF, and SVM, we compare the detected duplicates to the gold standard and calculate their precision, recall, and f-measure. Figure 9 shows the results. Although the MD approach achieves perfect precision for *restaurant*, only about half of the duplicates are discovered. Adding further MDs does not improve this recall, because the missing duplicates are hard to describe via matching dependency rules. Consider, for example, the *restaurant* records depicted in Table 9. The record pair (5,6) is a duplicate that our MDs were not able to detect, and (145,550) is a non-duplicate that was also not detected. A duplicate detection rule needed to correctly classify (5,6) as a duplicate and (145,550) as non-duplicate is $\texttt{address}_{1.0}$, $\texttt{city}_{1.0}$, $\texttt{name}_{0.15}$, $\texttt{phone}_{1.0}$, $\texttt{type}_{1.0}$

[3]https://cran.r-project.org/web/packages/randomForest/randomForest.pdf
[4]https://cran.r-project.org/web/packages/e1071/e1071.pdf

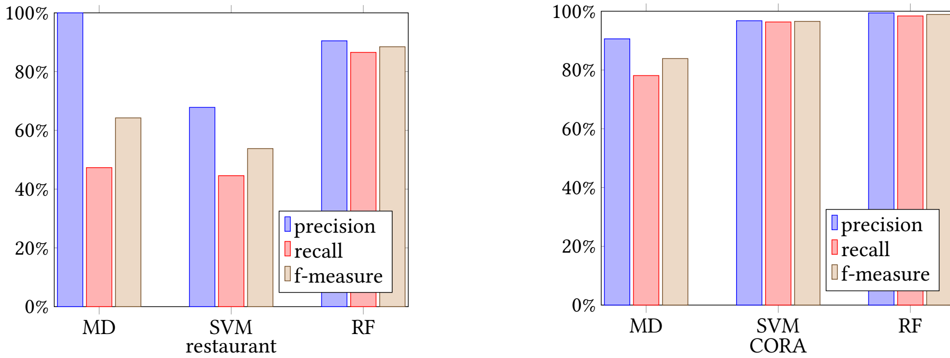

Fig. 9. Performance of the MD-, SVM-, and RF-based duplicate detection approaches

(note that both name value pairs have the same Levenshtein distance of 11 and the id attribute is ignored in general). This rule, however, cannot be found as a LHS of an MD, because there is no attribute left to be the RHS of that MD. However, assume that $\texttt{address}_{1.0}, \texttt{city}_{1.0}, \texttt{name}_{0.15}, \texttt{phone}_{1.0}$ (without type) would be a correct classifier for most (say 99%) of the matches. Then, we would need to find the actually true MD $\texttt{address}_{1.0}, \texttt{city}_{1.0}, \texttt{name}_{0.15}, \texttt{phone}_{1.0} \longrightarrow \texttt{type}_{0.08}$. Both (5,6) and (145,550) would be classified as duplicates, which lowers precision a bit but clearly increases recall. Our approach, however, does not report this MD due to its low RHS similarity threshold of 0.08; MDs with low RHS thresholds do not provide any evidence that their LHS is a good duplicate classifier. If we set the RHS threshold small enough, any LHS eventually forms a valid MD. So only MDs with high RHS thresholds indicate interesting duplicate classifiers – otherwise we could search for the classifiers directly ignoring the matching dependency construct all together. For this reason, some rules and, hence, some duplicates are not discovered.

id	address	city	name	phone	type
5	701 stone canyon rd	bel air	hotel bel air	310 472 1211	californian
6	701 stone canyon rd	bel air	bel air hotel	310 472 1211	californian
145	2880 las vegas blvd s	las vegas	steak house	702 734 0410	steak houses
550	2880 las vegas blvd s	las vegas	circus circus	702 734 0410	buffets

Table 9. Four records from the *restaurant* dataset; (5,6) is a duplicate and (145,550) a non-duplicate.

Compared to the other two approaches, the MD-based duplicate detection approach still performs surprisingly well considering that *no pre-labeling* of the data was needed: It stays about 20% f_1-measure behind RF, but competes well with SVM (about 10% f_1-measure better on *restaurant* and 10% worse on *CORA*). With 82% f_1-measure on *CORA* and 64% f_1-measure on *restaurant*, the absolute results are also fine, especially considering that MD rules tend to favor precision over recall. High precision values indicate high confidence in the discovered duplicates and, hence, support our claim that the selected MDs are in fact relevant and interesting. The experiment has therefore shown that HYMD does discover relevant MDs and that, although the result sets can be large (see *CORA*), a user can select relevant MDs easily with our interestingness features and result ranking. In [20], we show that the selection process can be learned and, hence, automated.

7 CONCLUSION

We presented HyMD a hybrid algorithm for the discovery of all minimal, non-trivial MDs within given similarity boundaries. The algorithm uses two search strategies, namely lattice traversal and inference from record pairs, to combine their strengths and minimize their weaknesses. Due to the resulting synergy effects, HyMD is significantly faster than both individual approaches and can, therefore, discover the MDs with exact decision boundaries. This makes HyMD the first MD discovery algorithm that can detect all MDs in one or across two different datasets. Because the result sets can grow accordingly large, we also proposed five pruning techniques to focus the discovery on interesting MDs.

The main contributions of HyMD are its new search space model that systematically captures all possible MD candidates, novel lattice traversal and candidate pruning techniques, and the new candidate validation and inference algorithms for MDs. Our experiments have shown that HyMD could easily process relations larger than 3 GB, which makes the algorithm applicable to many real-world datasets.

REFERENCES

[1] Z. Bahmani, L. Bertossi, and N. Vasiloglou. ERBlox: Combining matching dependencies with machine learning for entity resolution. In *International Conference on Scalable Uncertainty Management*, pages 399–414. Springer, 2015.
[2] Z. Bahmani, L. E. Bertossi, S. Kolahi, and L. V. Lakshmanan. Declarative entity resolution via matching dependencies and answer set programs. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2012.
[3] M. Bilenko, R. Mooney, W. Cohen, P. Ravikumar, and S. Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Systems*, 18(5):16–23, 2003.
[4] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
[5] 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.
[6] P. Christen. *Data matching*. Springer, 2012.
[7] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
[8] S. S. Cosmadakis, P. C. Kanellakis, and N. Spyratos. Partition semantics for relations. *Journal of Computer and System Sciences*, 33(2):203–233, 1986.
[9] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: a survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(1):1–16, 2007.
[10] W. Fan. Dependencies revisited for improving data quality. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 159–170. ACM, 2008.
[11] W. Fan, H. Gao, X. Jia, J. Li, and S. Ma. Dynamic constraints for record matching. *VLDB Journal*, 20(4):495–520, 2011.
[12] W. Fan, X. Jia, J. Li, and S. Ma. Reasoning about record matching rules. *PVLDB*, 2(1):407–418, 2009.
[13] W. Fan, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. *Journal of Data and Information Quality (JDIQ)*, 4(4):16, 2014.
[14] P. A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI Communications*, 12(3):139–160, 1999.
[15] J. Gardezi, L. Bertossi, and I. Kiringa. Matching dependencies with arbitrary attribute values: semantics, query answering and integrity constraints. In *Proceedings of the International Workshop on Logic in Databases*, pages 23–30. ACM, 2011.
[16] J. Gardezi, L. Bertossi, and I. Kiringa. Matching dependencies: semantics and query answering. *Frontiers of Computer Science*, pages 1–15, 2012.
[17] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 127–138, 1995.
[18] 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.
[19] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 14, pages 1137–1145, 1995.
[20] I. Koumarelas, T. Papenbrock, and F. Naumann. MDedup: Duplicate detection with matching dependencies. *PVLDB*, 13(5):712–725, 2020.
[21] M. Levandowsky and D. Winter. Distance between sets. *Nature*, 234(5323):34–35, 1971.

[22] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.

[23] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264.

[24] 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.

[25] A. E. Monge and C. Elkan. The field matching problem: algorithms and applications. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 267–270, 1996.

[26] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data profiling with Metanome. *PVLDB*, 8(12):1860–1871, 2015. Demo.

[27] 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.

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

[29] G. N. Paulley. *Exploiting Functional Dependence in Query Optimization.* PhD thesis, Waterloo, Ont., Canada, Canada, 2000. AAINQ51220.

[30] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[31] K. Sayood. *Introduction to data compression.* Morgan Kaufmann, 5th edition, 2017.

[32] R. Singh, V. V. Meduri, A. K. Elmagarmid, S. Madden, P. Papotti, J. Quiané-Ruiz, A. Solar-Lezama, and N. Tang. Synthesizing entity matching rules by examples. *PVLDB*, 11(2):189–202, 2017.

[33] S. Song and L. Chen. Discovering matching dependencies. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1421–1424. ACM, 2009.

[34] S. Song and L. Chen. Efficient discovery of similarity constraints for matching dependencies. *Data and Knowledge Engineering (DKE)*, 87:146–166, 2013.

[35] U. ul Hassan, S. O'Riain, and E. Curry. Leveraging matching dependencies for guided user feedback in linked data applications. In *Proceedings of the International Workshop on Information Integration on the Web (IIWeb)*, pages 1–6, 2012.

[36] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: An adaptive framework for similarity join and search. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 85–96. ACM, 2012.

[37] J. Wang, G. Li, J. X. Yu, and J. Feng. Entity matching: How similar is similar. *PVLDB*, 4(10):622–633, 2011.

[38] Y. Wang, S. Song, L. Chen, J. X. Yu, and H. Cheng. Discovering conditional matching rules. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 11(4):46, 2017.

[39] C. Xiao, W. Wang, and X. Lin. Ed-Join: An efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.