Determining the Largest Overlap between Tables

LUCA ZECCHINI*, University of Modena and Reggio Emilia, Italy
TOBIAS BLEIFUSS, Hasso Plattner Institute, University of Potsdam, Germany
GIOVANNI SIMONINI, University of Modena and Reggio Emilia, Italy
SONIA BERGAMASCHI, University of Modena and Reggio Emilia, Italy
FELIX NAUMANN, Hasso Plattner Institute, University of Potsdam, Germany

Both on the Web and in data lakes, it is possible to detect much redundant data in the form of largely overlapping pairs of tables. In many cases, this overlap is not accidental and provides significant information about the relatedness of the tables. Unfortunately, efficiently quantifying the overlap between two tables is not trivial. In particular, detecting their *largest overlap*, i.e., their largest common subtable, is a computationally challenging problem. As the information overlap may not occur in contiguous portions of the tables, only the ability to permute columns and rows can reveal it.

The detection of the largest overlap can help us in relevant tasks such as the discovery of multiple coexisting versions of the same table, which can present differences in the completeness and correctness of the conveyed information. Automatically detecting these highly similar, *matching* tables would allow us to guarantee their consistency through data cleaning or change propagation, but also to eliminate redundancy to free up storage space or to save additional work for the editors.

We present the first formal definition of this problem, and with it Sloth, our solution to efficiently detect the largest overlap between two tables. We experimentally demonstrate on real-world datasets its efficacy in solving this task, analyzing its performance and showing its impact on multiple use cases.

CCS Concepts: • **Information systems** → **Information integration**; **Deduplication**.

Additional Key Words and Phrases: table overlap, table matching, related tables, web tables, data lake

ACM Reference Format:
Luca Zecchini, Tobias Bleifuß, Giovanni Simonini, Sonia Bergamaschi, and Felix Naumann. 2024. Determining the Largest Overlap between Tables. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 48 (February 2024), 26 pages. https://doi.org/10.1145/3639303

1 OVERLAPPING TABLES

The Web contains a huge amount of structured data in tabular form. In 2008, Cafarella et al. [16] were already able to retrieve more than 14 billion HTML tables from the Web, of which more than 150 million contained high-quality relational data. A similar situation occurs in data lakes, where it is common to store a large number of tables, with the same table that might appear multiple times, possibly at different stages of its development and at different quality levels [3, 48, 49].

*Corresponding author (luca.zecchini@unimore.it)

Authors' addresses: Luca Zecchini, University of Modena and Reggio Emilia, Modena, Italy, luca.zecchini@unimore.it; Tobias Bleifuß, Hasso Plattner Institute, University of Potsdam, Potsdam, Germany, tobias.bleifuss@hpi.de; Giovanni Simonini, University of Modena and Reggio Emilia, Modena, Italy, giovanni.simonini@unimore.it; Sonia Bergamaschi, University of Modena and Reggio Emilia, Modena, Italy, sonia.bergamaschi@unimore.it; Felix Naumann, Hasso Plattner Institute, University of Potsdam, Potsdam, Germany, felix.naumann@hpi.de.

2836-6573/2024/2-ART48
https://doi.org/10.1145/3639303

Team	City	Stadium	Capacity
Arsenal	London	Emirates	60,704
Aston Villa	Birmingham	Villa Park	42,657
Liverpool	Liverpool	Anfield	53,394
Manchester	Manchester	Old Trafford	74,310

Stadium	Opened	Capacity
Anfield	1884	60,704
Craven Cottage	1896	22,384
Emirates	2006	60,704
Old Trafford	1910	74,310
Villa Park	1897	42,657

(a) A pair of tables about football teams and stadiums.

Liverpool	Liverpool	Anfield	53,394	
Arsenal	London	Emirates	60,704	2006
Aston Villa	Birmingham	Villa Park	42,657	1897
Manchester	Manchester	Old Trafford	74,310	1910
		Anfield	60,704	1884
		Craven Cottage	22,384	1896

Largest Overlap

(b) The largest overlap between the two tables.

Fig. 1. Example of largest overlap between tables.

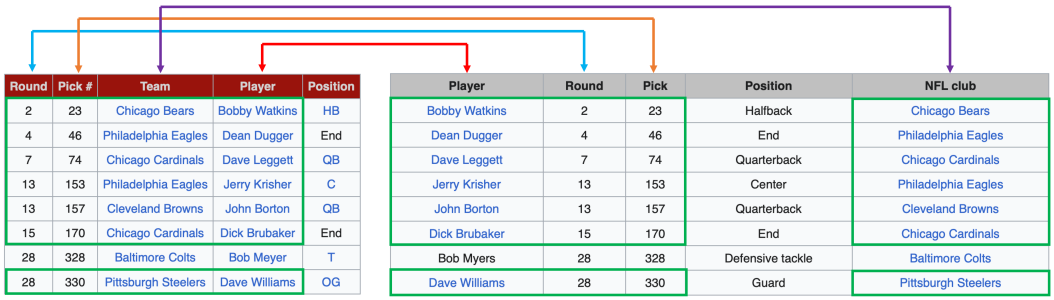
A particularly interesting case is represented by Wikipedia, where the tabular form is widely used to represent data. In fact, limited to its English version only, more than 60 million tables have appeared there throughout its history (up to September 1, 2019) [11]. Tables in this online encyclopedia have a very dynamic existence: they are frequently edited or updated, moved within their page or to another page, copied to related pages (e.g., when a topic is analyzed at different levels of detail, a table often appears both in the most generic page and in more specific ones) or elsewhere (even just to use them as a template), with frequent episodes of carelessness, conflicts among editors [14], or vandalism [53]. Considering this dynamism and the heterogeneity of the community of Wikipedia editors (which is composed of a huge number of people from all over the world), despite various attempts to automate the detection and resolution of some kinds of inconsistencies [5, 55], it can be very difficult to assess the quality of the tabular data appearing there, especially on less popular pages.

Among the 2.13 million tables existing at the time of the latest snapshot, we surprisingly discover that about 6.5 million pairs of tables present an overlap equal to at least half of the area of the smaller table, for an estimated redundancy of 63.49 MB. Even more surprisingly, we found that Wikipedia contains a huge amount of 5.9 million pairs of coexisting tables with identical content, highlighting the diffusion of copy-and-paste practices in such a scenario and the potential of centrally serving data for Wikipedia[1].

In particular, we focus on the *largest overlap* between the two tables, i.e., their largest common *rectangular* subtable, as depicted in Figure 1. Detecting the largest overlap is not trivial: the nature of tabular data allows changing the order of columns and rows (Figure 1b), making this task computationally challenging.

In many cases, the largest overlap between two tables is not accidental, but gives significant insights about the relatedness of the tables and the quality of the conveyed information. Let us consider a real example from Wikipedia. The tables presented in Figure 2, reporting the players selected from a US college football team in the 1955 NFL draft, appear in a page describing the

[1]All details about this analysis are presented in Section 4.3.



Round	Pick #	Team	Player	Position
2	23	Chicago Bears	Bobby Watkins	HB
4	46	Philadelphia Eagles	Dean Dugger	End
7	74	Chicago Cardinals	Dave Leggett	QB
13	153	Philadelphia Eagles	Jerry Krisher	C
13	157	Cleveland Browns	John Borton	QB
15	170	Chicago Cardinals	Dick Brubaker	End
28	328	Baltimore Colts	Bob Meyer	T
28	330	Pittsburgh Steelers	Dave Williams	OG

Player	Round	Pick	Position	NFL club
Bobby Watkins	2	23	Halfback	Chicago Bears
Dean Dugger	4	46	End	Philadelphia Eagles
Dave Leggett	7	74	Quarterback	Chicago Cardinals
Jerry Krisher	13	153	Center	Philadelphia Eagles
John Borton	13	157	Quarterback	Cleveland Browns
Dick Brubaker	15	170	End	Chicago Cardinals
Bob Myers	28	328	Defensive tackle	Baltimore Colts
Dave Williams	28	330	Guard	Pittsburgh Steelers

Fig. 2. Example of coexisting matching tables in related Wikipedia pages, presenting a different column order, a different convention for the Position column, and an inconsistency on the name of a player (Bob Meyer vs. Bob Myers).

previous season of the team[2] and in a page collecting the information about all draft picks during the team history[3], respectively. As highlighted by the arrows, the order of the columns is different and the two tables, which in principle should convey the same information, present some inconsistencies. In particular, different conventions for the player positions and a conflict on the surname of a player (Meyer vs. Myers) cause the removal of a whole column and a whole row from the largest overlap (since in this case including their common elements would lead to a smaller overlap size).

As made evident by this real-world example, the ability to detect the overlap between two tables, and in particular to retrieve pairs of highly similar tables, defined as *matching* or *duplicate* tables, can lead to several benefits. First, it allows checking the consistency of the information conveyed by the tables, pointing out cases of incompleteness or inconsistencies to be fixed by the editors. This aspect has a paramount importance in the context of an encyclopedia, where the information contained in the tables should always be correct, complete, and updated.

More generally, a scenario where a table can be duplicated at a certain point in time, with an independent development for the different copies, represents one of the main factors for the generation of inconsistencies. This is a common scenario that often occurs with enterprise data as well. For instance, when data scientists retrieve datasets from the enterprise's data lake, perform transformations (e.g., join, wrangling, etc.) for their analysis, then store back the new datasets into the data lake.

Depending on the context, a user might desire to leverage on the detected largest overlap to ensure the consistency of the information present in duplicate tables through operations of *data cleaning* [36] and *change propagation* [9], or it might be more convenient to directly prevent the rise of inconsistencies by eliminating this redundancy. In fact, avoiding redundancy not only allows to save disk space in the case of data lakes, where this phenomenon is quite frequent, but also to lighten the workload for website editors, who would just have to focus on a single consistent table instead of performing every editing multiple times, exposing to the risks shown by the example above. Whatever the purpose among those mentioned, one must first detect such matching tables.

The existing literature widely recognizes the importance of discovering related tables on the Web or in data lakes for enriching the conveyed information. Many approaches have therefore been proposed for the efficient detection of *unionable* tables [15, 41, 50], *joinable* tables [18, 27, 30, 63, 64], or more generally *related* tables [13, 19, 62], also providing insights about their subsequent integration [39]. Furthermore, representation learning has started to be used also for tables [4, 25,

[2]https://en.wikipedia.org/?oldid=1153086262
[3]https://en.wikipedia.org/?oldid=1160019836

26, 56, 60], and several large table corpora [29, 35, 42, 59] have been generated over the years and are widely adopted in these scenarios.

Yet, the task of detecting matching tables has been mostly overlooked by the existing research, where it was presented only in some specific or limited scenarios (e.g., to detect the subsequent versions of a table throughout the history of a Wikipedia page [10] or restricted to the most basic cases of perfect duplicates and inclusions [40]). Thus, this paper aims at filling this gap.

Our Contributions. We present Sloth, a novel method to determine the largest overlap between a given pair of tables, i.e., the maximal contiguous rectangular area of identical cells that can be achieved by reordering columns and rows of both tables. Sloth introduces the first algorithm designed for this task. First, our algorithm detects the pairs of attributes across the two tables that share some cell values. By combining these pairs of attributes, it is possible to obtain a complete overview of all potentially non-empty overlaps existing between the tables (i.e., the candidates to be the largest one) in the form of a *lattice* [7]. The combined pairs determine an upper bound for the area of the candidates. Thus, our algorithm aims to exploit this bounding mechanism to prioritize candidates and detect the largest overlap as soon as possible, minimizing the number of candidates for which we need to compute the actual area.

This task is computationally challenging, so we also propose a greedy variant for the algorithm based on *beam search* [47] to deal with critical pairs, when the exact algorithm struggles to produce a result in a reasonable time for the user.

Our experimental evaluation on real-world datasets assesses the efficiency of Sloth and the quality of the results produced by the greedy algorithm. Moreover, it highlights some relevant real-world use cases for Sloth, such as the detection of highly overlapping tables in a very popular context like Wikipedia, the recognition of potential copying between tables from different sources [44], and the discovery of candidate multi-column joins in a corpus of relational tables.

Outline. After we formalize the problem of detecting the largest overlap between two tables in Section 2, Section 3 provides an overview of Sloth, going into the details of the exact algorithm (Section 3.1) and its greedy variant (Section 3.2). In Section 4 we report the experimental evaluation of Sloth. Finally, we discuss related literature in Section 5, while Section 6 presents the future directions of our research and concludes the paper.

2 LARGEST OVERLAP DEFINITION

This section aims at laying the theoretical foundations for the detection of the largest overlap between two tables, presenting a formal definition of what we mean by *table overlap* (or simply *overlap* when evident from the context), describing how to compute its area, and evaluating the computational complexity of the problem.

The definition of table overlap relies on the concept of *attribute mapping*, defined as follows.

Definition 2.1 (Attribute Mapping). Given two tables $R(X)$ and $S(Y)$, where X and Y denote their schemas (i.e., their attribute sets), an *attribute mapping* (or simply *mapping*) between $R(X)$ and $S(Y)$ is defined as a bijective function M that maps a subset of X to a subset of Y:

$$M : X_M \subseteq X \rightarrow Y_M \subseteq Y$$

Due to the bijectivity of the mapping, the attribute sets have the same size ($|X_M| = |Y_M|$), no attribute in X is mapped to more than one attribute in Y, and no attribute in Y is mapped from more than one attribute in X. Each mapping determines a table overlap. Using $\uplus$ to denote the intersection under the bag semantics (which allows duplicates), we can define the table overlap and its area as follows.

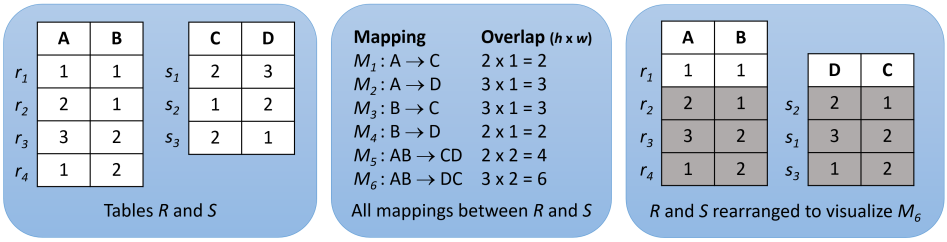


Fig. 3. Example to demonstrate the overlap between two tables determined by different mappings.

Definition 2.2 (Table Overlap). Given a mapping M, defined between tables $R(X)$ and $S(Y)$ considering the attribute subsets $X_M \subseteq X$ and $Y_M \subseteq Y$, the *table overlap* O_M is the bag intersection between the bags of the tuples obtained through the projection of $R(X)$ on X_M and $S(Y)$ on Y_M:

$$O_M = R[X_M] \Cap S[Y_M]$$

Definition 2.3 (Overlap Area). We define the *overlap area* A_M as the number of cells contained in the overlap O_M:

$$A_M = |X_M| \cdot |O_M|$$

where $|X_M|$ and $|O_M|$ represent the *width* and the *height* of the rectangle of overlapping cells between the two tables, respectively. We also refer to A_M as the area of mapping M.

Definition 2.4 (Largest Overlap). Let $\mathcal{O}$ be the set of overlaps determined by all possible mappings between $R(X)$ and $S(Y)$. We define the set of the *largest overlaps* $\mathcal{O}^* \subseteq \mathcal{O}$ as those overlaps that have the maximum area:

$$\mathcal{O}^* = \{O_{M^*} \in \mathcal{O} \mid A_{M^*} \geq A_M, \forall O_M \in \mathcal{O}\}$$

We refer to the mappings of $\mathcal{O}^*$ as *top mappings*, denoted as M^*. Note that the number of top mappings (in most cases just one) is equal to the number of largest overlaps. Figure 3 shows an example of two tables and the effects of different mappings.

Computational Complexity. Let us consider the `OVERLAP` decision problem, whose instance is composed of two tables R and S and a positive integer K. `OVERLAP` raises an affirmative answer if there exists a subtable O common to both R and S with area $A_O \geq K$. Note that the problem considers all overlaps between R and S, not only the largest ones. It is easy to see that `OVERLAP` $\in$ NP, since a nondeterministic algorithm only needs to guess a subset of row and column pairs from R and S and check in polynomial time that the two obtained subtables are equal and their area $A_O \geq K$.

The classical NP-complete `CLIQUE` decision problem [33], raising an affirmative answer if a graph $G = (V, E)$ contains a clique of at least C vertices, where $C \leq |V|$ is a positive integer, can be transformed into `OVERLAP`. In fact, considering the adjacency matrix $M : |V| \times |V|$ of the graph G, where $M_{i,j} = 1$ if $(V_i, V_j) \in E$ or $i = j$, G contains a clique of at least C vertices if and only if between M and $I : |V| \times |V|, I_{i,j} = 1, \forall i, \forall j$, there exists a (square) overlap O with area $A_O \geq C^2$ and the sets of indices for the columns and for the rows mapped from M are equal. The constraint on the indices discriminates between a clique and other structures determining square overlaps in M, such as bicliques. The transformation is polynomial (quadratic in the number of vertices). Hence, the `OVERLAP` decision problem is NP-complete.

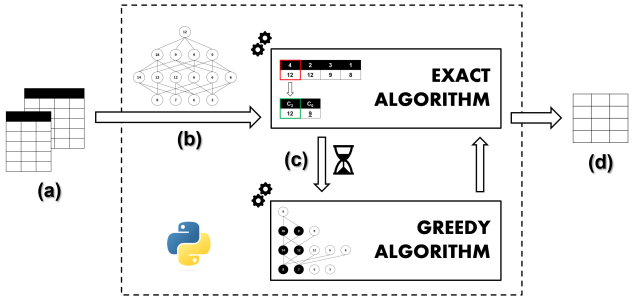


Fig. 4. An overview of the Sloth workflow, from a pair of tables (a) to their largest overlap (d) through our detection algorithm (b) and its greedy variant for critical cases (c).

Search Space. Every possible mapping M among the attributes of a table pair defines a table overlap. To obtain the size of the search space, we determine the number of such mappings. We can model this problem as a bipartite graph, where the attributes of each table represent one of the two nonadjacent vertex sets. For a worst-case analysis, let us assume that this bipartite graph is complete, i.e., we can consider mapping every attribute of one table to any attribute of the other table. Each mapping now corresponds to an independent edge set, i.e., a subset of the edges so that none of these are adjacent (i.e., share a common vertex). For a complete bipartite graph with vertex sets of size $|X|$ and $|Y|$, the number of independent edge sets (a.k.a. *Hosoya index*[4]) follows the formula[5]:

$$\sum_{k=0}^{\min(|X|,|Y|)} k! \cdot \binom{|X|}{k} \cdot \binom{|Y|}{k}$$

Note that according to our definition a mapping does not need to cover all attributes of either table. The number corresponds to choosing two k-sized subsets out of X and Y and then permuting one side while keeping the other side fixed. This grows super-exponentially in the number of attributes of the table with fewer attributes.

3 LARGEST OVERLAP DETECTION

In Figure 4, we represent the high-level design of Sloth. Implemented in Python, Sloth considers as input a pair of tables (Figure 4a) and returns the largest overlap detected between them (Figure 4d). In addition to this, Sloth can also return some additional insights on the original tables, such as the visualization for each table of the cells excluded from the result, making it easier for the user to point out the differences and the inconsistencies between the two tables.

To detect the largest overlap, Sloth implements our novel algorithm designed for this task (described in detail in Section 3.1) and its greedy variant inspired by the beam search technique for the most challenging pairs of tables (illustrated in Section 3.2). In particular, the exact algorithm is our default choice (Figure 4b), with a timeout mechanism defined to spot critical cases that would not provide a result in a reasonable time, activating the greedy algorithm (Figure 4c). We discuss this choice in more depth in Section 3.2.1. The timeout is set by the user and is aimed at the early detection of the described critical cases. The parameter for the greedy algorithm (i.e., the beam width β, clarified in Section 3.2) is set to a default value selected based on our experimental evaluation (Section 4.2) and can be edited according to the user's needs (e.g., a faster computation or a better accuracy).

[4]https://mathworld.wolfram.com/HosoyaIndex.html
[5]https://oeis.org/A086885

Algorithm 1: Largest overlap detection algorithm

Input: Two tables $R(X)$ and $S(Y)$; minimum area Δ (default 0)
Output: The set of the largest overlaps O^*

```
1  O* ← ∅                                                          // largest overlaps
2  Seeds ← findSeeds(R, S)
3  θ ← max(Δ, Seeds[0].A)                                          // pruning threshold
4  Levels ← initLevels(Seeds)                                      // priority queue to generate candidates
5  Candidates ← maxHeap(∅, key = A)                                // priority queue to verify candidates
6  while Candidates ≠ ∅ or Levels ≠ ∅ do
7      while Candidates.head().A < Levels.head().A do
8          Levels, Candidates ← genCand(Levels, Candidates, Seeds)  // generate more candidates
9      if Candidates ≠ ∅ then
10         topC ← Candidates.pop()                                  // top candidate
11         if topC.O ≠ ∅ then
12             O* ← O* ∪ topC.O                                     // largest overlap found!
13         else
14             Levels, Candidates ← verCand(R, S, topC, Levels, Candidates)  // verify top candidate
15 return O*
```

3.1 Exact Algorithm

In this section, we present our algorithm to detect the largest overlap(s) between two tables (Algorithm 1). The algorithm considers as input two tables (e.g., the two tables about football teams in Figure 5a), denoted as $R(X)$ and $S(Y)$, and returns the set of their largest overlaps O^*. If the two tables have no cells in common, the area of the largest overlap is equal to zero and an empty set is returned. The optional argument Δ represents the minimum area to consider the overlap as relevant and therefore to include it in the result. For instance, the user might consider the largest overlap as relevant if its area is at least equal to a certain percentage of the area of the smallest table. The definition of minimum/maximum thresholds for the width/height of the overlap is also supported, leading to the detection of the largest overlap among those complying with the defined boundaries. Since implementing this feature is straightforward, we do not go into the details of it to avoid overloading the formalization of the algorithm. We provide some examples of its application to real-world use cases in Sections 4.4 and 4.5.

Our algorithm needs to consider all mappings that can potentially determine the largest overlap between the two tables. These mappings are denoted as *candidates*, since they are candidates to be the top mapping. To identify the candidates, our algorithm first considers all possible *single-attribute mappings*, i.e., those mappings for which X_M is represented by a single attribute x, checking the area of the overlap between $R[x]$ and $S[M(x)]$. We call *seeds* those single-attribute mappings whose area is greater than zero, and we collect them in a dedicated list (Line 2), represented in Figure 5b, where they are sorted by descending area.

Since every *multi-attribute mapping* is a combination of some single-attribute mappings, the candidates can be considered as the combinations of the seeds and modeled as the nodes of a lattice, as depicted in Figure 5b, where every level n contains the combinations of n seeds. Moving up within the lattice increases the width of the overlap, but not necessarily its area, as its height may decrease as new columns are added. In particular, the seed with the minimum area (equal to its height) in the combination defines an upper bound for the height of the candidate, and therefore for its area. This bounding mechanism can be exploited both to prune the lattice and to prioritize the candidates based on their potential area. We define therefore the *pruning threshold* θ (Line 3),

Function 1: findSeeds() function

```
   Input: The two tables R(X) and S(Y)
   Output: The sorted list of the detected seeds
1  Seeds ← ∅
2  forall x in X do
3      forall y in Y do
4          seed.M ← M : x → y                                  // mapping
5          seed.A ← |R[x] ⋔ S[y]|                              // area
6          if seed.A > 0 then
7              Seeds.append(seed)
8  return Seeds.sort(A, desc)                                  // sort by increasing dominance
```

which always contains the maximum between Δ and the maximum actual area of a candidate that we know so far, which is initially the area of the first seed in the list and can be possibly updated every time we verify a new candidate, discovering its actual area.

Our algorithm leverages on the upper bound defined by the seeds to manage two priority queues, aiming to minimize both the number of candidates that need to be materialized and those among them whose actual area needs to be computed: *(i) Levels* (Line 4), containing the representations of the levels of the lattice, used to generate the candidates incrementally; *(ii) Candidates* (Line 5), containing the generated candidates, used to progressively verify their actual area and detect the largest overlap.

In particular, we iterate on the priority queues until both of them are emptied (Line 6), terminating early as soon as all largest overlaps are detected (or only the first one, if we are only interested in their area with no need to consider their content). At each iteration, first we need to ensure that at least one of the candidates with the potential largest overlap has been generated and inserted into *Candidates* (Lines 7-8), as depicted in Figure 6. Next, we can check the candidate at the top of *Candidates* (Lines 9-10). If it has already been verified (i.e., its actual overlap has already been computed), none of the other candidates (among both the ones already generated and the ones yet to generate) can present a greater area, hence it is one of the largest overlaps, and we can add it to the result set (Lines 11-12); otherwise, we need to compute its overlap (and therefore its actual area) and reinsert it into the priority queue if it can still be part of the result set (Lines 13-14).

In the next sections, we will go into the details of the different steps composing the algorithm, providing their formal representation and using the example introduced in Figures 5 and 6 to help their understanding.

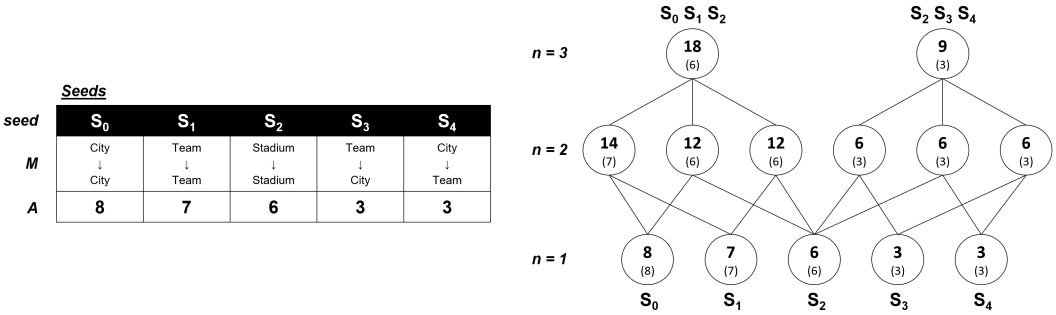
3.1.1 Detecting the Seeds. First, we need to detect the seeds. As stated in Line 2, this operation is delegated to the **findSeeds**() function (Function 1). Here the seeds, stored in a dedicated eponymous list (Line 1), are detected by verifying the area of all possible single-attribute mappings defined between the two tables (Lines 2-7). If its area is greater than zero, then a mapping is inserted into *Seeds* (Lines 6-7), keeping track of its area. We denote the number of detected seeds as s, i.e., $s = |Seeds|$. If an attribute from one table has cells in common with multiple attributes from the other table, that attribute appears in multiple seeds. Since a mapping has been defined as a bijective function, a valid multi-attribute mapping can include only one seed out of each cluster of seeds with a common attribute. We implicitly denote as mappings only the valid ones, while the invalid ones are automatically filtered out.

Example. *Considering the example tables $R(X)$ and $S(Y)$ in Figure 5a, the mappings inserted into Seeds are the ones depicted in Figure 5b. Here the mappings are represented by the attribute M (in*

Team	City	Stadium	Capacity
Arsenal	London	Emirates Stadium	60,704
Barcelona	Barcelona	Camp Nou	99,354
Bayern Munich	Munich	Allianz Arena	75,000
Inter Milan	Milan	San Siro	80,018
Liverpool	Liverpool	Anfield	53,394
Manchester United	Manchester	Old Trafford	74,310
Milan	Milan	San Siro	80,018
Real Madrid	Madrid	Santiago Bernabéu	81,044

Team	City	Country	Stadium	Founded
Real Madrid	Madrid	Spain	Santiago Bernabéu	1902
Milan	Milan	Italy	San Siro	1899
Bayern Munich	Munich	Germany	Allianz Arena	1900
Liverpool	Liverpool	England	Anfield	1892
Barcelona	Barcelona	Spain	Camp Nou	1899
Ajax	Amsterdam	Netherlands	Johan Cruyff Arena	1900
Inter Milan	Milan	Italy	Giuseppe Meazza	1908
Manchester United	Manchester	England	Old Trafford	1878
Chelsea	London	England	Stamford Bridge	1905
Juventus	Turin	Italy	Juventus Stadium	1897

(a) Two input tables $R(X)$ (on the left) and $S(Y)$ (on the right) about football teams.



(b) The sorted list of detected seeds (on the left) and the valid candidates in the lattice generated from the seeds (on the right). For every node in the lattice we report the upper bounds for its area and its height (between round brackets).

Fig. 5. Two input tables, the sorted list of detected seeds, and the lattice of valid candidates.

this example we use the headers instead of the indices to enhance readability) and their area by the attribute A. Thus, we see that the first seed, mapping the City attribute in R to the City attribute in S, has an area of 8 cells, since all cells from the first attribute find a match in the second one. The mappings involving the Team and Stadium pairs have smaller overlaps instead, due to the absence of Arsenal and its stadium from S and the use of two distinct denominations for the stadium of Inter Milan. Note that three teams (Milan, Liverpool, and Barcelona) carry the name of their city, causing the mappings from Team to City and from City to Team to be detected as seeds (denoted in Figure 5b as S_3 and S_4, respectively). By definition, S_1 and S_3 cannot occur together, since Team cannot be mapped to both Team and City (same for S_0 and S_3, S_0 and S_4, S_1 and S_4).

As stated above, the candidates can be seen as the nodes of the lattice representing the combinations of the seeds (depicted in Figure 5b), where every level n contains the mappings obtained by combining n seeds. Since the lattice includes only the valid mappings, it is allowed to present an incomplete shape (hence we can define it as a *semilattice* in case an attribute appears in more than one seed). The highest level containing at least one valid mapping can be identified considering the minimum number of distinct attributes from a table covered by the seeds, i.e., $min(|X_{Seeds}|, |Y_{Seeds}|)$. When every attribute appears in at most one seed, the complete lattice is composed of a number of nodes equal to:

$$\sum_{n=1}^{s} \binom{s}{n} = 2^s - 1$$

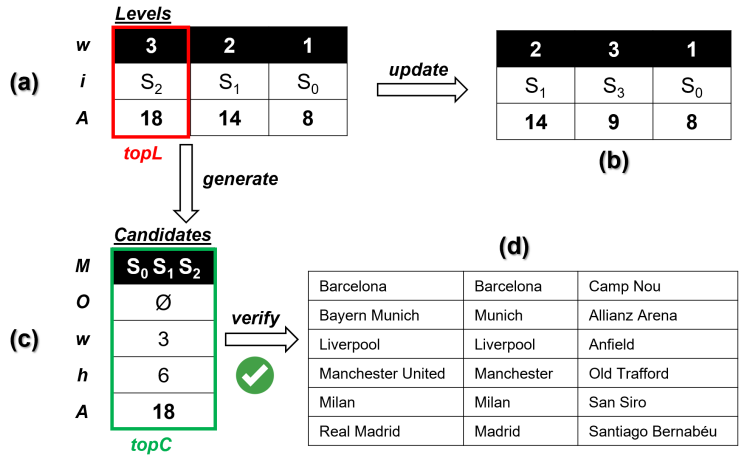


Fig. 6. *Levels* as initialized (a) and after the update (b), *Candidates* (c), and the detected largest overlap (d).

In the lattice, for every node we report the upper bounds for the height (between round brackets) and the area, where the latter is obtained as the product between the bounded height and the number of seeds combined to generate the candidate (i.e., $|X_M|$), representing the width of the overlap. These values are determined by the seeds. In fact, considering a candidate M, the maximum possible value for its height is determined by the seed with the smallest area appearing in the combination, since $|R[X_M] \pitchfork S[M(X_M)]| \leq min(\{|R[x] \pitchfork S[M(x)]|, \forall x \in X_M\})$. Therefore, we can say that a seed *dominates* the seeds with a greater area when they appear together in a candidate. Before being returned, the seed list is sorted by increasing dominance (Line 8), i.e., $Seeds[i].A \geq Seeds[i+1].A, 0 \leq i < s-1$. In the last position we find the seed with the minimum area, which dominates all other seeds. Note that the nodes in the bottom level of the lattice ($n = 1$), representing the seeds themselves, already report the actual values for the height and the area.

Example. *In our example, the seeds cover three distinct attributes from both tables, hence the top level of the lattice in Figure 5b is the one containing the combinations of three seeds. Let us consider the candidate combining S_0 and S_1 (i.e., the first node of level 2), with an area of 8 and 7 cells, respectively. While the width of its overlap is equal to 2, its height can be at most equal to 7, hence its area to 14.*

3.1.2 Prioritizing the Levels. Based on the number of detected seeds, the lattice can become significantly large. The possibility to define for every level an upper bound for the area of its candidates allows to generate the candidates incrementally according to their potential area, avoiding the materialization of the entire lattice. In fact, all candidates in a level n have a fixed width of n, and their height is bounded by the area of the seed in the position $n-1$ of the sorted *Seeds* list (i.e., the first when $n = 1$, the second when $n = 2$, etc.), since in a level the first $n-1$ seeds are always dominated by other seeds if they occur in a candidate. Thus, we can insert representations of the levels into *Levels*, the first of the two priority queues introduced above, to determine which level can produce the candidate with the maximum potential area and which seeds have to be combined for generating it (i.e., the seed bounding the height and the ones that precede it in the list).

The initialization of *Levels* is delegated to the **initLevels**() function (Function 2). This function initializes the *Levels* priority queue as an empty *max heap* structure (Line 1), then iterates over all possible levels the lattice may contain, starting from the bottom (i.e., level 1 that contains the seeds), until it reaches the top of the lattice, which is limited by the minimum number of attributes covered

Function 2: initLevels() function

Input: The sorted list of the detected seeds
Output: The priority queue for the lattice levels

```
1 Levels ← maxHeap(∅, key = A)
2 for n ← 1 to min(|X_Seeds|, |Y_Seeds|) do
3     level.w ← n                                   // width
4     level.i ← n − 1                               // seed pointer
5     level.A ← level.w · Seeds[level.i].A          // max area
6     if level.A ≥ θ then
7         Levels.push(level)
8 return Levels
```

by the seeds of either table, as stated above. For every level, the queue maintains the width of the candidates in that level (w) and a pointer i to the seed bounding their height, which consequently determines the upper bound for their area A (Lines 3-5). If this upper bound is equal to or greater than the pruning threshold θ, the level is inserted into *Levels* (Lines 6-7). Then, we can proceed with the upper levels.

Example. *In our example, the Levels priority queue is initialized with three elements (Figure 6a), one for each level of the lattice.*

3.1.3 Generating the Candidates. The *Candidates* priority queue is initialized as an empty *max heap* structure in Line 5 of Algorithm 1 to be filled incrementally with the generated candidates. At the beginning of an iteration, we might need to materialize further candidates to ensure that at least one of the candidates with the maximum potential area has been generated and inserted there. The generation process is described by the **genCand**() function (Function 3) and is depicted in Figure 6.

Considering the level n located at the top of *Levels* (Line 1), we generate the candidates corresponding to all combinations of n seeds composed of the seed pointed by the level and $n-1$ seeds selected among the ones preceding it in the list (Line 2). Note that this incremental process covers all candidates in the level, since:

$$\sum_{i=n-1}^{s-1} \binom{i}{n-1} = \binom{s}{n} \qquad n \neq 0$$

and generates them in the correct ordering based on their potential area, since $Seeds[i].A \cdot n \geq Seeds[i+1].A \cdot n, n-1 \leq i < s-1$.

Example. *In our example, the candidate with the maximum potential area is generated from level 3, located at the top of Levels. This level is currently pointing to the seed S_2, which occupies the third position in the seed list, preceded by S_0 and S_1. Since we are considering only three seeds, we only need to materialize the first node of the third level of the lattice in Figure 5b.*

For every generated candidate, we keep track of its width w (equal to n) and the upper bound for its height h, which determines its potential area A (Lines 4-6). The attribute O is initialized to be empty (Line 7) and is updated with the actual overlap once it is computed. Thus, when a candidate has the maximum priority, if its overlap is empty, we can infer that its attribute A describes its potential area, hence we still need to detect the overlap to verify its actual value. If the candidate is a seed, it is possible to directly initialize O with its actual overlap, which has already been computed. Finally, the candidate is inserted into the *Candidates* priority queue if its area is at least equal to θ or exceeds it (Lines 8-9).

Function 3: genCand() function

Input: The priority queues for the lattice levels and for the candidates, the sorted list of the detected seeds
Output: The updated priority queues

```
1  topL ← Levels.pop()                                              // top level
2  forall comb in combs(Seeds[: topL.i], topL.w) do
3      cand.M = comb                                                // mapping
4      cand.w = topL.w                                              // width
5      cand.h = Seeds[topL.i].A                                     // max height
6      cand.A = cand.w · cand.h                                     // max area
7      cand.O ← ∅                                                   // overlap
8      if cand.A ≥ θ then
9          Candidates.push(cand)
10 if topL.i < len(Seeds) − 1 then
11     topL.i ← topL.i + 1                                          // next seed
12     topL.A = topL.w · Seeds[topL.i].A
13     if topL.A ≥ θ then
14         Levels.push(topL)                                        // reinsert level
15 return Levels, Candidates
```

Example. *In our example, θ is still equal to 8, reflecting the maximum area among the seeds; thus, the generated candidate is inserted into Candidates, which was previously empty.*

Once generated the new candidates, the considered level has to be updated, pointing now to the next seed in the list (unless we were already pointing to the last one) and updating the upper bound for the area of the candidates yet to be generated consequently (Lines 10-12). If the updated upper bound is greater than or equal to θ, then the level is reinserted into *Levels* (Lines 13-14).

Example. *This update step is represented for our example in Figure 6b; from S_2, level 3 points now to S_3, which sets the upper bound for the height to 3 and for the potential area to 9 (which is greater than θ, so the level can be reinserted, but now level 2 can produce candidates with a greater potential area). If the reinserted level was located at the top of Levels in one of the next iterations, it would have to generate the candidates corresponding to all combinations containing S_3 (the pointed seed) and two seeds among S_0, S_1, and S_2, producing in principle three new candidates (in this case, they would be automatically filtered out, since they would all contain at least one seed between S_0 and S_1, not compatible with S_3).*

3.1.4 Verifying the Candidates. If the actual overlap of the candidate located at the top of *Candidates* has not been verified yet, we need to call the **verCand**() function (Function 4).

First, we need to compute the overlap according to the definitions provided in Section 2 to obtain the actual values for its height and its area (Lines 1-3). If the actual area is at least equal to the pruning threshold θ, the candidate is reinserted into *Candidates* with the updated values, since it might be the largest overlap. Moreover, we update θ to ensure it reflects the value of the maximum actual area that we know at the moment (Lines 4-6). This dynamic updating allows to prune the lattice in a more efficient way. In particular, when θ is updated, we can scan both priority queues to delete the elements that cannot reach its new value (Lines 7-12).

Example. *In our example, verifying the top candidate (Figure 6c) produces the overlap depicted in Figure 6d, presenting an area of 18 cells. Since its area is greater than θ, the candidate is reinserted into Candidates and θ is set to 18, causing the pruning of all elements in Levels (Figure 6b), since they cannot produce candidates whose area can reach this threshold. At the next iteration, the top candidate*

Function 4: verCand() function

Input: The two tables $R(X)$ and $S(Y)$, the top candidate, the priority queues for the lattice levels and for the candidates
Output: The updated priority queues

```
1  topC.O ← R[X_topC.M] ⋔ S[Y_topC.M]                          // overlap
2  topC.h ← |topC.O|                                          // actual height
3  topC.A ← topC.w · topC.h                                   // actual area
4  if topC.A ≥ θ then
5      θ ← topC.A
6      Candidates.push(topC)                                  // reinsert candidate
7      forall cand in Candidates do
8          if cand.A < θ then
9              Candidates.delete(cand)                        // prune candidate
10     forall level in Levels do
11         if level.A < θ then
12             Levels.delete(level)                           // prune level
13 return Levels, Candidates
```

has already been verified and is therefore added to the result set, completing the task after one single verification.

Coherently with the concept of dominance among seeds, since the height of a mapping is bounded by the seed with the smallest area appearing in the combination, adding further seeds to a mapping cannot increase this bound. Thus, we can state that the height of a node in the level n of the lattice is bounded by the height of the nodes from the level $n-1$ that are combined to generate it (we can equivalently say that they are *covered* by that node).

As a further optimization, when we compute the overlap of a candidate we can check if some elements in *Candidates* cover it and in this case bound their height based on the one of the verified candidate, i.e., $cand.h = min(topC.h, cand.h)$. Similarly, if we cache all verified heights with the related mappings, when we generate a candidate we can check if it covers some of those mappings and update its potential height consequently. These optimizations, overlooked in Algorithm 1 to improve readability but included in its implementation, make the potential area of the candidates closer to the actual one, enhancing the effectiveness of the pruning and the definition of the top candidate in the priority queue.

3.1.5 Computational Complexity. Let us consider tables T_1 and T_2, with r_1 and r_2 rows and c_1 and c_2 columns, respectively. For detecting the seeds, we need to consider all column combinations and compute their bag intersection. Using a hashmap, the cost is in $O\left(r_1 * c_1 + r_2 * c_2 + c_1 * c_2 * \min(r_1, r_2)\right)$: $\min(r_1, r_2)$ for the intersection itself and $r_{1(2)} * c_{1(2)}$ for the hashmap generation. The s detected seeds, $0 \leq s \leq \min(c_1, c_2)$, generate a lattice composed of $2^s - 1$ candidates, net of invalid mappings. In principle, considering the actual area A^* of the largest overlap, not known a priori, our algorithm needs to generate and verify all g candidates with a potential area $A \geq A^*$ (note that the optimizations described at the end of Section 3.1.4 can significantly reduce the number of candidates to verify). These candidates are generated from Σ *generating seeds*, i.e., the total number of seeds pointed by the items of the *Levels* priority queue, at most $s - l + 1$ for each level l. Hence, Σ generating seeds produce a number of candidates g equal to:

$$g = \sum_{\sigma \in \Sigma} \binom{i_\sigma}{l_\sigma - 1}$$

where l_σ denotes the level pointing the seed and i_σ its index in $Seeds$, $0 \leq i_\sigma \leq s-1$. In the worst case (i.e., if all seeds have the same area but no cell alignment), our algorithm might need to generate and verify through the bag intersection all candidates in the lattice (apart from the s seeds, whose actual area is already known), for a cost of:

$$O\left(\sum_{l=2}^{s}\binom{s}{l} * l * (r_1 + r_2)\right) = O\left(2^s * s * (r_1 + r_2)\right)$$

Hence, the computational complexity of the algorithm is in practice dominated by the exponential complexity in the number of seeds.

3.2 Greedy Algorithm

Algorithm 1 for the detection of the largest overlap between two tables, described in Section 3.1, generates the candidates by combining the seeds. If we need to apply the algorithm to a pair of wide tables with some values repeated across several columns in both (e.g., multiple columns containing Boolean values), it is possible that a significant number of seeds is detected, producing a huge lattice mostly composed of invalid nodes. In some cases, this situation can lead to the impossibility of generating the combinations for the new candidates in a reasonable amount of time.

While in principle it would be possible to reduce the number of seeds by post-processing them (e.g., to make an attribute appear in at most k seeds or reducing this situation to the *stable matching* problem [32], with the area of a seed determining its weight), pruning them a priori without knowing the actual area of any of the candidates in the upper levels can have a significantly negative impact on the correct detection of the largest overlap.

For a result as close as possible to the exact largest overlap, we designed a greedy variant for our algorithm inspired by *beam search*, a heuristic search algorithm widely applied in the speech recognition area [34] that performs a breadth-first search in a tree by only expanding the β most promising nodes at each level. The parameter β, denoted as *beam width*, is defined by the user based on the trade-off between efficiency (a smaller value for β requires to evaluate fewer candidates) and completeness (the case for $\beta = \infty$ would evaluate all candidates, producing the exact result).

Our greedy algorithm is designed to bottom-up traverse the lattice generated from the seeds. The set of the largest overlaps is initialized using the seeds with the maximum area (if it is greater than the minimum area Δ defined by the user). This value is used to initialize the pruning threshold θ, which can be updated (as well as the result set) every time we verify a new candidate. Before moving to the upper levels, we check if it is still possible for any remaining candidate to reach (or even surpass) θ. This is possible because the combination of bounded heights and width determines a bound for the area of their overlaps. If the current θ is larger than all such upper bounds of candidate areas, the algorithm saves on further exact area computations and terminates.

At the beginning, we consider the detected seeds and select a maximum of β candidates with the greatest area. To find candidates for the second level, we combine each of them with every other seed and drop repeated and invalid combinations. After this generating step, we verify all new candidates and again select only the β candidates with the greatest actual area among them. For the third and every further level, we combine the selected candidates of the previous level with every seed that is not already part of the candidate and then again verify their area and limit their number to β. Note that the selection of the best β candidates may affect the produced results; thus, it is important to rely on some tiebreaker strategies for the candidates that present the same actual area (e.g., favoring the ones whose seeds present the greatest total area).

To determine the computational complexity, let us again consider tables T_1 and T_2, with r_1 and r_2 rows and c_1 and c_2 columns, respectively. Since from level 2 up we generate the candidates

to verify by combining the top β ones from the previous level with all s seeds (net of generated duplicate candidates and seeds that would raise invalid ones), the cost of our greedy algorithm can be quantified as:

$$O\left(\sum_{l=2}^{s} \beta * s * l * (r_1 + r_2)\right) = O\left(s^3 * \beta * (r_1 + r_2)\right)$$

Combined with the (unchanged) initial seed detection, this leads to an overall computational complexity of $O\left(r_1 * c_1 + r_2 * c_2 + c_1 * c_2 * \min(r_1, r_2) + s^3 * \beta * (r_1 + r_2)\right)$, i.e., polynomial in the number of seeds. Our experimental evaluation (Section 4.2) demonstrates that, even in absence of approximation guarantees on the quality of its result, the greedy algorithm is generally able to detect largest overlaps of the same area as those discovered by the exact algorithm.

3.2.1 Coordinating the Algorithms. As described at the beginning of Section 3, Sloth adopts a *trial-and-error* approach to evaluate a pair of tables, favoring the exact algorithm whenever possible. The main reasons behind this choice are: *(i)* the guarantees of correctness and completeness of the result given by the exact algorithm (not guaranteed by its greedy variant, despite the empirical demonstration of its good accuracy); *(ii)* the minimal impact of timeouts on our experiments on coexisting tables from real-world scenarios (Sections 4.3 to 4.5), where only around 1.6% of all table pairs need to revert to the greedy algorithm. Further, Sloth can be seen as a unique *best-effort* solution, since several findings by the exact algorithm, such as the detected seeds and the computed actual heights and overlaps, can be reused by its greedy variant.

The main factor leading to timeouts is the number of seeds, which directly affects the size of the lattice, hence the number of candidates that potentially need to be generated. This aspect is strictly correlated to the width of the tables and the amount of repeated cell values across them (Figures 7a and 7b). Nevertheless, as depicted in Figures 7c and 7d, none of these factors defines a clear threshold to distinguish between successes and timeouts. The definition of rules for the automatic selection of the algorithm is therefore not trivial, while a classifier-based approach, able to detect more complex patterns among the described features, might seem more promising. However, to pursue such an approach, several factors need to be taken into account. First, we need training data, and, as stated above, in several table corpora timeouts are rather rare. Further, tables from different corpora might have very dissimilar features, as we show in Table 1, hence training the classifier on one dataset (e.g., the sample of wiki_history described in Section 4.1, where timeouts are quite frequent) might not cover many patterns occurring in other datasets, causing a poor accuracy. False positives might not only miss the possibility of detecting the exact result, but also impact negatively on the performance. As we show in Figure 7i, in many cases the exact algorithm can be faster than its greedy variant, since it can directly prioritize the evaluation of the most promising candidates from the lattice, while the latter proceeds level by level from bottom to top.

4 EXPERIMENTAL EVALUATION

In this section, we present the experimental evaluation of Sloth, aiming to assess: *(i)* what is the performance of Sloth at detecting the largest overlap between two tables based on the size and the features of those tables (Section 4.1); *(ii)* what is the accuracy of the results obtained using the greedy algorithm (Section 4.2); *(iii)* how many highly overlapping tables are present in a real-world relevant scenario such as Wikipedia and what we can learn from their analysis (Section 4.3); *(iv)* how Sloth can be applied to real-world use cases such as the detection of potential copying between tables from different sources (Section 4.4) or the detection of composite foreign keys in the relational context (Section 4.5).

Table 1. Statistics about the number and the size of the tables appearing in the used datasets.

Dataset	#D	Width (#columns)			Height (#rows)		
		MIN	MAX	AVG	MIN	MAX	AVG
wiki_history	55.97M	1	5694	5.92	1	17.38k	26.63
wiki_latest	2.13M	1	883	5.23	1	4670	11.47
uni_dwh	158	1	55	9.48	2	151.78k	5604.79
stock_raw	1.15k	4	69	16.18	221	1000	987.58
stock_clean	1.15k	3	17	12.49	221	1000	987.58
flight_clean	1.17k	3	7	5.75	6	1309	662.17

Datasets. In Table 1, we present the datasets used in our experimental evaluation. We denote as wiki_history the Wikipedia table matching dataset from the IANVS project[6]. In contrast to other corpora of Wikipedia tables (e.g., the WikiTables dataset[7] by Bhagavatula et al. [6], composed of 1.6M high-quality relational tables), it captures the evolution of all 3.5M tables present in the English Wikipedia throughout its entire history (until September 1, 2019), for a total amount of 55.97M different table versions stored in the JSON Lines text format (see [11] for more details). The composition of the *table lineages* (i.e., the collections of the subsequent versions of a table) was performed by Bleifuß et al. as described in [10]. We separately consider the most recent snapshot of Wikipedia tables from this dataset, denoted as wiki_latest.

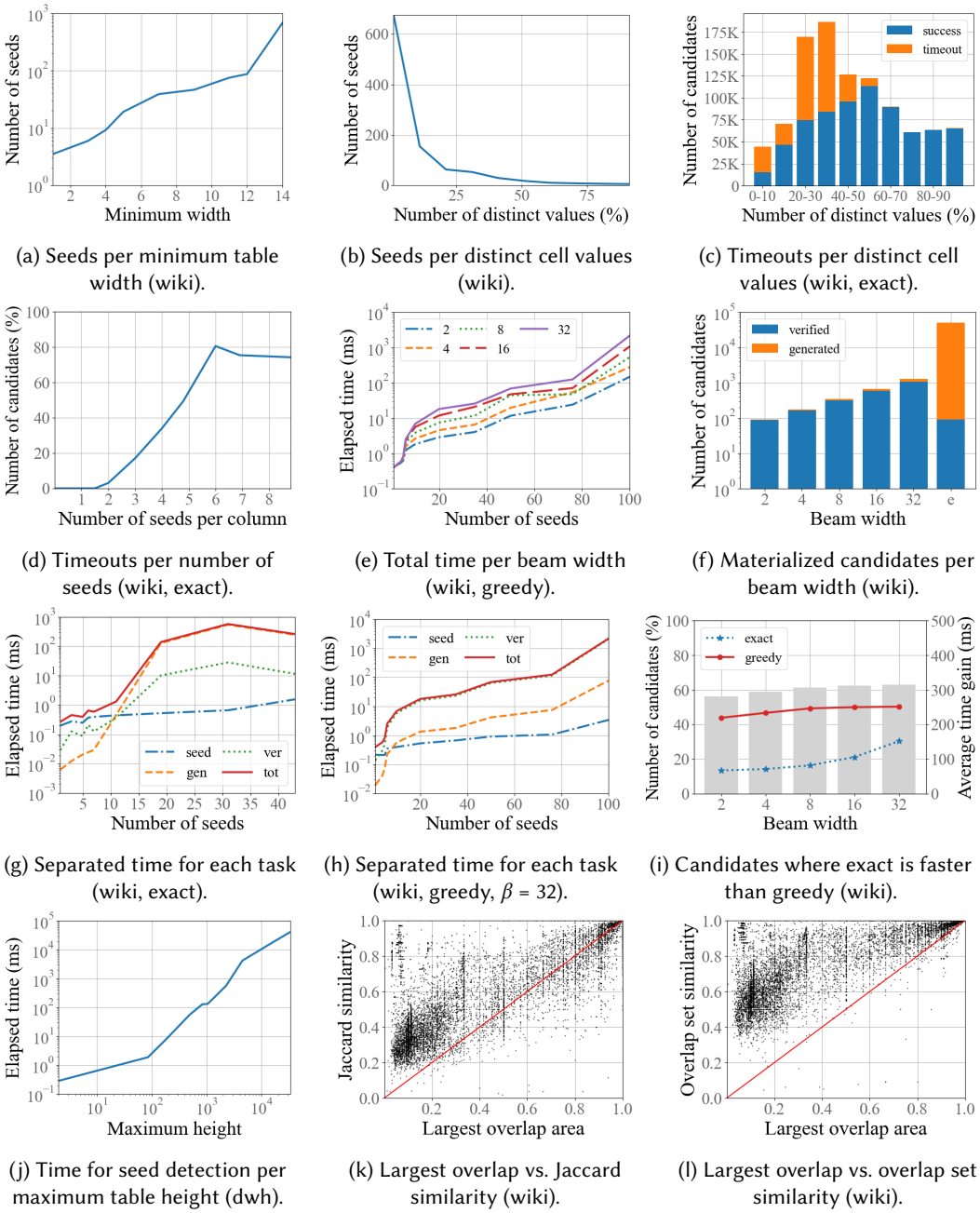
Beyond the Wikipedia scenario, in our experiments we also employ uni_dwh [18], a real-world university data warehouse, composed of relational tables with a significantly higher number of rows, and two datasets[8] (described in [44]) reporting the information about stock symbols and flights captured from different sources across multiple days (55 sources over 21 days and 38 over 31, respectively); thus, on every day there is a table for each source. For the stock dataset both the original tables (stock_raw) and their versions obtained through schema alignment (stock_clean) are available, while only the latter is provided for the flight dataset.

Setup. Sloth has been implemented in Python 3.7 and its code is available on GitHub[9]. We used a MongoDB instance to store the tables and their metadata. Our experiments were performed on a server machine equipped with 4 Intel Xeon E5-2697 @ 2.40 GHz (72 cores) processors and 216 GB of RAM, running Ubuntu 18.04. The default configuration for Sloth adopts a timeout of 3 seconds for the exact algorithm and 60 seconds for the greedy algorithm (with a default beam width of 32).

4.1 Performance of the Algorithms

The performance of Sloth has been evaluated on two of the datasets presented above: a representative subset of the wiki_history dataset and the uni_dwh dataset, chosen to cover both the case of Web tables and the one of relational tables collected from a database in our analysis. The subset was created by randomly picking from wiki_history at most 5 tables from each table lineage (to include in the evaluation also these cases of highly related tables) and filtering out the ones with less than 10 rows, for a total of 4.1M tables. We then produced 19.1M pairs of tables through LSH banding [43], using 16 bands and a minhash of 128 bits, and randomly selected 1M of these pairs to be used in our evaluation. Our results are reported in Figure 7, which we now explain in detail.

[6]https://hpi.de/naumann/projects/data-profiling-and-analytics/change-exploration.html
[7]http://websail-fe.cs.northwestern.edu/TabEL
[8]https://lunadong.com/fusionDataSets.htm
[9]https://github.com/dbmodena/sloth



(a) Seeds per minimum table width (wiki).

(b) Seeds per distinct cell values (wiki).

(c) Timeouts per distinct cell values (wiki, exact).

(d) Timeouts per number of seeds (wiki, exact).

(e) Total time per beam width (wiki, greedy).

(f) Materialized candidates per beam width (wiki).

(g) Separated time for each task (wiki, exact).

(h) Separated time for each task (wiki, greedy, β = 32).

(i) Candidates where exact is faster than greedy (wiki).

(j) Time for seed detection per maximum table height (dwh).

(k) Largest overlap vs. Jaccard similarity (wiki).

(l) Largest overlap vs. overlap set similarity (wiki).

Fig. 7. Performance of Sloth evaluated on the wiki_history dataset (a-i, k-l) and the uni_dwh dataset (j).

First, we examine the factors that determine the number of seeds and thus the size of the lattice. Figure 7a shows how the number of seeds increases with the number of columns, as expected. For table pairs with different number of columns, we consider the smaller number, which also bounds the height of the lattice. We plot the average number of seeds for 10 quantiles, positioning the

Table 2. Accuracy of the greedy algorithm results with different beam width (β) values, compared to the exact result and the largest greedy result as the ratio between their areas.

β	Timeout	Accuracy (exact)		Accuracy (greedy)	
		1	**$\geq$ 0.75**	**1**	**$\geq$ 0.75**
2	0.040%	98.612%	99.668%	72.988%	96.707%
4	0.041%	99.351%	99.857%	80.940%	98.718%
8	0.042%	99.835%	99.992%	87.326%	99.392%
16	0.046%	99.917%	99.998%	92.025%	99.700%
32	0.127%	99.940%	99.999%	96.311%	99.915%

values at the beginning of the interval covered by the quantile on the x-axis. A second dimension that has a high impact on the number of seeds is the percentage of distinct cell values in the two tables. Given a pair of tables, we consider the two sets of cell values, then divide the sum of their sizes by the total number of cells of the two tables. In Figure 7b, aggregating this value into 10% buckets, we show that the number of seeds tends to significantly increase when the tables contain many repeated values. These are also the most challenging pairs for Sloth, as the distribution of the cases for which the exact algorithm exceeds the timeout shows in Figure 7c. In fact, these 291k pairs fall almost entirely in the initial half of the plot and mostly reflect the scenario in which two tables contain the repetition of few distinct cell values in several different alignments, producing many seeds and requiring our exact algorithm to generate many candidates before discovering the largest overlap. In Figure 7d, we equivalently show how the percentage of timeouts increases with the average number of seeds per column (computed on the table with the smaller width, always considering 10 quantiles).

In case of timeout, Sloth activates the greedy algorithm based on beam search. The impact of the beam width is illustrated for 10 quantiles in Figure 7e. Using a larger beam width tends to produce more reliable results (see Section 4.2), but it also implies materializing more candidates, whose area needs to be verified to select the most promising ones, as highlighted by Figure 7f, hence requiring more time. Thus, the candidate verification represents by far the most expensive operation for the greedy algorithm, as illustrated in Figure 7h, which analyzes the time (once again as the average value for 10 quantiles) required by the three main tasks performed by Sloth (the other two being seed detection and candidate generation). For our exact algorithm, the most critical task is the candidate generation instead, as highlighted by the last column of Figure 7f and by Figure 7g, while the use of the priority queues allows minimizing the number of candidates whose actual area needs to be verified out of the generated ones. In Figure 7i, the bars show for each beam width the percentage of candidates for which the exact algorithm (when it does not exceed the timeout) is faster than its greedy variant. The lines show for each algorithm the average time gain on the candidates for which it is the fastest.

Finally, the number of rows can affect those tasks that require to perform the bag intersection, in particular the seed detection. While the effect is very limited in the case of Wikipedia, it becomes much more evident when moving to the relational tables of uni_dwh, as presented in Figure 7j, where the average number of rows is significantly larger (see Table 1) and the time required for detecting the seeds can exceed 5 minutes in some extreme cases.

4.2 Accuracy of the Greedy Algorithm

In Table 2, we assess the accuracy of the largest overlaps detected by the greedy algorithm, computed on the 1M random table pairs introduced in Section 4.1. In particular, we consider five different values for the beam width (β) and we report for each of them: *(i)* the percentage of pairs for which

Table 3. Types of overlapping tables in Wikipedia.

Type	#Table Pairs	Estimated Size
Perfect duplicates	5.91M (85.521%)	39.55 MB
Inclusions	351.24k (5.085%)	6.05 MB
Additional rows	59.75k (0.865%)	4.35 MB
Additional columns	289.97k (4.198%)	1.91 MB
Additional rows and columns	1.53k (0.022%)	0.56 MB
Partial overlaps	648.91k (9.394%)	39.16 MB
$\geq$ *50% of the smallest table*	252.80k (3.660%)	35.52 MB
$<$ *50% of the smallest table*	396.12k (5.735%)	6.60 MB
Total ($\geq$ 50%)	**6.51M (94.265%)**	**63.49 MB**

the computation exceeds the timeout; *(ii)* the percentage of pairs for which the ratio between the area of their largest overlap and the one obtained by the exact algorithm is equal to 1 (i.e., same area) or at least 0.75 (i.e., close to the exact area), computed on the pairs where the exact algorithm provides a solution (708.2k) and its greedy variant does not exceed the timeout; *(iii)* the same percentage considering the ratio w.r.t. the largest overlap with the maximum area among those produced by the different beam width values (also considering the case with $\beta = 64$), to study the accuracy of the greedy algorithm on the pairs where the exact algorithm exceeds the timeout.

As depicted in Table 2, in the first case (third and fourth columns) even smaller values for the beam width lead to a very high accuracy. Differently, in the second one (last two columns), they struggle to reproduce the exact result (despite producing a good approximation); hence, values such as 16 or 32 appear to be a more reliable choice (note that these values reach a very high accuracy of at least 0.9 on 97.148% and 99.013% of the pairs, respectively), even if these configurations require more computational time, as highlighted by their increasing (yet marginal) timeout rate.

4.3 Overlapping Tables in Wikipedia

With Sloth, we are able to detect overlapping tables coexisting in the latest snapshot of the Wikipedia table matching dataset, composed of 2.13M tables (see Table 1). To avoid comparing all pairs, we first performed LSH banding on the dataset, using 8 bands and a minhash of 128 bits[10]. This operation reduced the candidate set to 6.91M pairs of tables (we ignored the trivial tables with only one distinct cell value). Among the remaining candidates, the greedy algorithm was required only for 110.88k of them (1.605% of the cases), and for only 4 the timeout of 60 seconds was exceeded. The average time for evaluating a candidate was 153.7 ms, and the process could be easily parallelized, distributing the candidates over several workers.

In Table 3, we report the results of our analysis, categorizing the evaluated candidates according to the following types of overlap: *(i) perfect duplicates*, if the two tables have identical content (with the possible reordering of columns and rows); *(ii) inclusions*, if the smaller table is contained in the larger one, which presents some additional rows and/or columns; *(iii) partial overlaps*, if both tables present some cells that are excluded from the largest overlap. For the estimated memory occupation, we rely on the average size of the cells in this snapshot, equal to 14.53 bytes.

The number of pairs with an overlap of at least 50% of the smaller table is surprisingly high. In particular, Wikipedia contains a huge amount of perfect duplicates; this situation denotes a wide diffusion of the copy-and-paste practice across multiple pages, which can foster inconsistencies, as highlighted by the relevant presence of inclusions and partial overlaps (e.g., we notice how even

[10]This approximates a Jaccard similarity threshold of about 0.8; we observed in preliminary experiments, by manually inspecting samples of candidates, that under that threshold the table overlap occurs mostly by chance in this dataset.

Table 4. Size of the clusters of sources with potential copying.

Dataset	SLOTH *(min_height)*	Jaccard *(threshold)*	Li et al. [44]
stock_raw	13, 2 *(0.85)*	12, 2 *(0.80)*	11, 2
stock_clean	13, 2 *(0.95)*	12, 2 *(0.85)*	
flight_clean	5, 4, 3, 3, 3 *(0.95)*	5, 4, 3, 2, 2 *(0.80)*	5, 4, 3, 2, 2

the frequent and apparently trivial case of the legend tables defined for a certain category of pages can lead to the rise of differences as they evolve over time), and requires a significant editing effort by the users to achieve coherency in the encyclopedia.

Finally, we show how the area of the largest overlap detected by SLOTH, normalized by the area of the smaller table, differs from traditional metrics based on set semantics, such as Jaccard similarity [37] and overlap set similarity [24, 63]. Set semantics cannot consider the repetition of cell values and their alignment in the table; moreover, Jaccard similarity presents a bias against sets of different sizes. Figures 7k and 7l show on a random sample of 10k pairs (from the 1M random table pairs introduced in Section 4.1) how these substantial differences can lead Jaccard and overlap set similarity (normalized by the size of the smaller set) to very dissimilar results from SLOTH. For instance, with a 0.8 threshold SLOTH would detect 321k pairs out of 1M, while Jaccard and overlap set similarity 339k and 557k (268k and 318k in common with SLOTH). For 365k and 601k pairs, SLOTH's result differs by at least 0.2 from Jaccard and overlap set similarity, while the unnormalized value for the latter is more than double or less than half the largest overlap area in 334k cases. We collected in our GitHub[11] repository 50 representative example pairs from the wiki_history dataset depicting typical cases where the analyzed metrics differ significantly.

4.4 Potential Copying Detection

As a further real-world use case, in this section we focus on the detection of potential copying across different sources. In their work on truth finding [44], Li et al. detect clusters of sources with potential copying in the datasets about stocks and flights introduced in Table 1, relying on criteria such as claimed dependencies or query redirections, and computing several measures on them. The right column of Table 4 shows the size of those clusters.

To perform potential copying detection with SLOTH, we configured the algorithm with a high threshold for the minimum height of the largest overlap to be detected (e.g., 0.9 of the table with fewer rows in the pair) and a minimum width of 2 (since all tables share the column with the object identifiers). This way, the algorithm finds pairs of tables with a very large overlap on a subset of columns. Note that we consider the stock dataset both in the raw and in the clean version that is obtained through schema alignment. Due to the limited size of the datasets and the early stopping introduced by the defined bounds, we simply consider for each day all the pairs of tables obtained through the Cartesian product, net of identity and reflexivity. SLOTH finishes all computations for each single day in less than one minute. Finally, we consider those pairs of tables with an overlap width of at least 0.8 of the table with fewer columns (considering the lower bound for the schema similarity of the clusters detected in [44]) during the whole period, with a tolerance of 3 days, to be potential copies. As a baseline, we adopt Jaccard similarity (on which Li et al. rely for multiple measures), computed between the sets of cell values of the tables.

As depicted in Table 4, both solutions are not only able to detect all clusters in [44], but also to retrieve some additional sources with potential copying in both domains. While a source with almost exactly the same schema and content is easily detected even by Jaccard similarity as a part

[11] https://github.com/dbmodena/sloth/tree/main/examples

of the larger cluster of the stock domain, Sloth can retrieve three more additional sources: first, a second source for the same cluster with a significantly wider schema and different labels (which can be retrieved by Jaccard similarity only when dropping its additional columns in the aligned version, with a threshold of 0.75); second, two more sources in the flight domain, composed of a limited amount of copied rows. Indeed, Jaccard similarity struggles with sets of different sizes; its set semantics, ignoring the repetition and the alignment of cell values, might even lead to the detection of false positives. Thus, using Sloth we were able to detect all clusters of sources with potential copying with a minimum effort, without needing schema alignment (as highlighted through the stock_raw dataset), also leading to the discovery of meaningful additional sources.

4.5 Discovery of Candidate Multi-Column Joins

Sloth can support practitioners in the fundamental yet challenging task of automatically discovering candidate multi-column joins in a corpus of tables. To demonstrate it, we employ the real-world uni_dwh dataset, composed of 158 tables, making 12.4k table pairs. The average time for computing the largest overlap is about 9.6 seconds per pair (mainly due to the seed detection, as pointed out in Section 4.1, not counted towards the timeout) and the process could be easily parallelized.

To detect reasonable multi-column joins, we limit the largest overlap to between 2 and 5 columns. We also require the height of the overlap to be at least 90% of the table with fewer rows: this ensures high coverage of its values, while not looking for a perfect containment, which would be limiting for the join discovery scenario. For instance, say we find that two tables have a largest overlap involving 90% of the rows of one table and three attributes ⟨First Name, Last Name, Position⟩; then, we might use these attributes for joining the two tables.

A largest overlap that complies with the defined settings is detected for only 243 pairs (out of 5.6k pairs that would be obtained without defining restrictions on the overlap size). It is easy to determine the cardinality of the join by counting the duplicates in the original tables of the attributes that participate in the overlap: for each table in the pair, if there are no duplicates in the projection on those attributes, then the table participates in the join with cardinality one. Thus, among the detected overlaps, we find that 48, 64, 131 correspond to *one-to-one*, *one-to-many*, *many-to-many* joins, respectively. For 34 out of these pairs, the overlap detects a *composite key* (primary or alternate) for at least one table.

None of the existing solutions supports the automatic discovery of candidate multi-column joins (see Section 5 for more details). As a baseline, we can therefore adapt Josie [63], which uses the overlap set similarity to detect single-column joins, to consider as a set the entire tables instead of the single columns. Differently from Sloth, such a baseline cannot take into account the structure of the table (including the repetition and the alignment of cell values), preventing the definition of the bounds that ensure the detection of multi-column joins. In fact, only 106 of the candidate multi-column joins discovered by Sloth (48, 37, and 21 for each join type, 27 covering a composite key) are present among the top 243 table pairs according to their overlap set similarity (normalized by the size of the smaller set) retrieved by the baseline. Instead, the baseline detects several single-column joins and many invalid candidates, such as pairs with a low row coverage or where the values from a column are matched by multiple columns on the other side.

5 RELATED WORK

In this section, we review the literature closely related to Sloth, moving in four main directions: *(i)* the algorithms for the efficient discovery of joinable, unionable, and related tables; *(ii)* the existing solutions facing the problem of detecting duplicate tables; *(iii)* the case of partial *n*-ary inclusion dependencies from data profiling research [1]; *(iv)* the use of similarity measures based on the overlap between two matrices in different scientific domains.

Related Table Discovery. As stated in Section 1, a plethora of algorithms have been designed for the efficient discovery of unionable [15, 41, 50], joinable [18, 23, 27, 61, 64], and related tables [13, 19, 62]. These algorithms cannot compute the actual value of the largest overlap, but in some cases they can produce an upper bound for it; thus, they can be exploited to scale to large table corpora, passing to Sloth only the most promising pairs.

For example, Josie [63] is designed to efficiently detect joinable tables in massive data lakes through an *overlap set similarity search*. In particular, given a query table and selected its join column, Josie finds the top-k joinable tables, i.e., those tables with a column whose set of cell values presents the largest intersection with the one of the join column. Since the similarity (which reflects the size of this intersection) is computed on a single column using the set semantics, it is impossible for Josie to obtain the actual size of the largest overlap. Nevertheless, if we consider the entire tables under the bag semantics instead of the single columns, the similarity would reflect the number of common cells between the two tables, which represents an upper bound for the size of the largest overlap. Thus, such an adapted version of Josie might be used to detect for a query table the top-k tables that are most promising to present the maximum largest overlap, then pass them to Sloth to compute the actual values. Note that differently from Jaccard similarity, and therefore LSH, overlap set similarity is not biased against pairs of sets of different size.

Mate [30] is the only join-discovery system that allows to discover multi-column joins. Mate exploits a dedicated hashing function named XASH to generate a super-key for every table row, which is employed as a Bloom filter [12] to efficiently check if a row might contain a specific value combination. Mate requires the user to provide a set of columns as input; then, it retrieves all possible tables with a set of columns that might join with the user-provided set. Thus, Mate cannot be easily employed to detect the largest overlap, since the set of columns that yields the largest overlap is not known by the users up-front.

Duplicate Table Detection. While a lot of attention has been devoted to unionable, joinable, and related tables, duplicate tables are barely touched by the existing literature. Koch et al. [40] provide a definition to this problem and propose to use XASH to detect duplicate tables in data lakes, designing a pipeline for both query table and data lake deduplication scenarios. The authors define two tables as duplicates if they contain the same sets of tuples, possibly after permuting the columns of one table. Thus, they only tackle the basic cases of perfect duplicates or row containment, ignoring both column containment and mostly those cases where the overlap misses cells from both tables, which pose the most significant challenges and usually provide the most meaningful insights.

A similar task is performed by Bleifuß et al. [10] to detect matching tables across subsequent versions of a Wikipedia page. However, the proposed approach, relying on a multi-stage matching based on Jaccard similarity (also in a relaxed form), is specifically designed for one-to-one matches among a limited number of tables sharing the same context, also exploiting specific aspects such as the position of the tables inside the page, hence not generalizable.

In the literature [31, 45, 54], the expression *table matching* has also been employed to refer to the task of matching Web tables with knowledge bases, i.e., to annotate their cells, rows, and columns with entities and properties from a knowledge base (such as YAGO [52] or DBpedia [8]). By identifying a common link through the knowledge base, these methods could be adapted to detect overlaps among tables. The major drawback of this approach would be that all the cells that are not linked with an entity in the knowledge base (e.g., YAGO) would not be considered for computing the table overlaps—hence making it highly inaccurate in many scenarios.

Partial n*-ary Inclusion Dependencies. Inclusion dependencies* (INDs) are a kind of data dependency expressing that the set of tuples of one column combination is contained in the set of tuples of another column combination, hence allowing to discover foreign keys among relational tables.

INDs are a prominent topic in data profiling research and many approaches have been proposed to detect them efficiently [20, 38, 58]. In particular, our definition of largest overlap shows affinity with the case for *partial* n-*ary INDs*. Partial INDs [46] allow a defined amount of tuples to violate the containment constraint (since the largest overlap does not need to cover all rows), while *n*-ary INDs [21, 22, 51] consider combinations covering more than one column (as usually happens for the largest overlap). Therefore, an algorithm for detecting partial *n*-ary INDs should return the largest overlap among the detected dependencies. Unfortunately, even if some algorithms for detecting INDs have been extended to separately deal with both partial and *n*-ary INDs [28], no solution has been proposed yet to find INDs presenting both of these features. Note that the algorithms for discovering INDs adopt the set semantics, while our problem would require the bag semantics to correctly evaluate the overlap area.

Overlap-Based Similarity Measures. While Sloth is the first solution to detect the largest overlap between tables, some related approaches in different research areas use the *largest common submatrix* to estimate the similarity between two matrices, e.g., as a distance measure between images [2] or for determining coevolving proteins [57]. However, the case of tables is unique due to the possibility of changing the order of columns and rows, making this problem significantly more complex.

6 CONCLUSION AND FUTURE WORK

We presented Sloth, a method to efficiently detect the largest (*rectangular*) overlap between two tables. Knowing the largest overlap allows detecting matching tables, leading to several benefits both on the Web and in data lakes. For instance, it allows spotting and solving common data quality issues, such as inconsistent or incomplete information. Also, it helps eliminate redundancy to free up storage space or to save additional work for the editors, preventing the insurgence of data quality problems. Through our experimental evaluation we assessed the performance of Sloth in real-world scenarios, considering Web tables from Wikipedia and relational tables from a data warehouse, up to use cases such as the detection of potential copying between different sources and the discovery of candidate multi-column joins in a corpus of relational tables.

Moving beyond the results presented in this paper, we plan to broaden our research in two main directions. Firstly, we want to design updatable indexes to enable table-overlap-based discovery at scale, i.e., to allow users to provide a table as a query and to retrieve the top-k tables presenting a large overlap with it. Secondly, we want to enable Sloth to detect not only the largest, but also the *best* overlap between two tables, defining quality metrics to capture the meaningfulness of an overlap based for instance on its width and height or on the entropy of its content. A further challenge is the relaxation of the cell matching operation, including in the overlap not only identical cells but also highly similar ones (e.g., based on an edit distance threshold or the closeness of their embeddings [17]), and the possibility of masking inconsistencies with NULL values to avoid losing partially matching information.

ACKNOWLEDGEMENTS

We would like to warmly thank Donatella Firmani for her help in the NP-completeness proof and the members of the data profiling team at the Hasso Plattner Institute (Leon Bornemann, Youri Kaminsky, and Daniel Lindner) for following the development of our project and sharing their ideas and suggestions. This work was partially supported by the "Enzo Ferrari" Engineering Department (University of Modena and Reggio Emilia) within the project FAR2023DIP, by Cineca within the project ISCRA C (code HP10CSFAPN), and by MUR within the project "Discount Quality for Responsible Data Science: Human-in-the-Loop for Quality Data" (code 202248FWFS).

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. 2018. *Data Profiling.* https://doi.org/10.1007/978-3-031-01865-7

[2] Alessia Amelio and Clara Pizzuti. 2013. Average Common Submatrix: A New Image Distance Measure. In *Proceedings of the International Conference on Image Analysis and Processing (ICIAP), Part 1.* 170–180. https://doi.org/10.1007/978-3-642-41181-6_18

[3] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał Świtakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proceedings of the VLDB Endowment (PVLDB)* 13, 12 (2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[4] Gilbert Badaro, Mohammed Saeed, and Paolo Papotti. 2023. Transformers for Tabular Data Representation: A Survey of Models and Applications. *Transactions of the Association for Computational Linguistics (TACL)* 11 (2023), 227–249. https://doi.org/10.1162/tacl_a_00544

[5] Malte Barth, Tibor Bleidt, Martin Büßemeyer, Fabian Heseding, Niklas Köhnecke, Tobias Bleifuß, Leon Bornemann, Dmitri V. Kalashnikov, Felix Naumann, and Divesh Srivastava. 2023. Detecting Stale Data in Wikipedia Infoboxes. In *Proceedings of the International Conference on Extending Database Technology (EDBT).* 450–456. https://doi.org/10.48786/edbt.2023.36

[6] Chandra Sekhar Bhagavatula, Thanapon Noraset, and Doug Downey. 2015. TabEL: Entity Linking in Web Tables. In *Proceedings of the International Semantic Web Conference (ISWC), Part 1.* 425–441. https://doi.org/10.1007/978-3-319-25007-6_25

[7] Garrett Birkhoff. 1940. *Lattice Theory.* https://doi.org/10.1090/coll/025

[8] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. 2009. DBpedia: A crystallization point for the Web of Data. *Journal of Web Semantics* 7, 3 (2009), 154–165. https://doi.org/10.1016/j.websem.2009.07.002

[9] Tobias Bleifuß, Leon Bornemann, Theodore Johnson, Dmitri V. Kalashnikov, Felix Naumann, and Divesh Srivastava. 2018. Exploring Change: A New Dimension of Data Analytics. *Proceedings of the VLDB Endowment (PVLDB)* 12, 2 (2018), 85–98. https://doi.org/10.14778/3282495.3282496

[10] Tobias Bleifuß, Leon Bornemann, Dmitri V. Kalashnikov, Felix Naumann, and Divesh Srivastava. 2021. Structured Object Matching across Web Page Revisions. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE).* 1284–1295. https://doi.org/10.1109/ICDE51399.2021.00115

[11] Tobias Bleifuß, Leon Bornemann, Dmitri V. Kalashnikov, Felix Naumann, and Divesh Srivastava. 2021. The Secret Life of Wikipedia Tables. In *Proceedings of the Workshop on Search, Exploration, and Analysis in Heterogeneous Datastores (SEA Data @ VLDB).* 20–26. https://ceur-ws.org/Vol-2929/paper4.pdf

[12] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[13] Alex Bogatu, Alvaro A. A. Fernandes, Norman W. Paton, and Nikolaos Konstantinou. 2020. Dataset Discovery in Data Lakes. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE).* 709–720. https://doi.org/10.1109/ICDE48307.2020.00067

[14] Siarhei Bykau, Flip Korn, Divesh Srivastava, and Yannis Velegrakis. 2015. Fine-Grained Controversy Detection in Wikipedia. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE).* 1573–1584. https://doi.org/10.1109/ICDE.2015.7113426

[15] Michael J. Cafarella, Alon Halevy, and Nodira Khoussainova. 2009. Data Integration for the Relational Web. *Proceedings of the VLDB Endowment (PVLDB)* 2, 1 (2009), 1090–1101. https://doi.org/10.14778/1687627.1687750

[16] Michael J. Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. 2008. WebTables: Exploring the Power of Tables on the Web. *Proceedings of the VLDB Endowment (PVLDB)* 1, 1 (2008), 538–549. https://doi.org/10.14778/1453856.1453916

[17] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* 1335–1349. https://doi.org/10.1145/3318464.3389742

[18] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Sam Madden, and Michael Stonebraker. 2018. Aurum: A Data Discovery System. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE).* 1001–1012. https://doi.org/10.1109/ICDE.2018.00094

[19] Anish Das Sarma, Lujun Fang, Nitin Gupta, Alon Halevy, Hongrae Lee, Fei Wu, Reynold Xin, and Cong Yu. 2012. Finding Related Tables. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* 817–828. https://doi.org/10.1145/2213836.2213962

[20] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. 2002. Efficient Algorithms for Mining Inclusion Dependencies. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 464–476. https://doi.org/10.1007/3-540-45876-X_30

[21] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. 2009. Unary and n-ary inclusion dependency discovery in relational databases. *Journal of Intelligent Information Systems* 32, 1 (2009), 53–73. https://doi.org/10.1007/s10844-007-0048-x

[22] Fabien De Marchi and Jean-Marc Petit. 2003. Zigzag: a new algorithm for mining large inclusion dependencies in databases. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*. 27–34. https://doi.org/10.1109/ICDM.2003.1250899

[23] Dong Deng, Albert Kim, Samuel Madden, and Michael Stonebraker. 2017. SilkMoth: An Efficient Method for Finding Related Sets with Maximum Matching Constraints. *Proceedings of the VLDB Endowment (PVLDB)* 10, 10 (2017), 1082–1093. https://doi.org/10.14778/3115404.3115413

[24] Dong Deng, Yufei Tao, and Guoliang Li. 2018. Overlap Set Similarity Joins with Theoretical Guarantees. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 905–920. https://doi.org/10.1145/3183713.3183748

[25] Li Deng, Shuo Zhang, and Krisztian Balog. 2019. Table2Vec: Neural Word and Entity Embeddings for Table Population and Retrieval. In *Proceedings of the ACM International Conference on Research and Development in Information Retrieval (SIGIR)*. 1029–1032. https://doi.org/10.1145/3331184.3331333

[26] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2020. TURL: Table Understanding through Representation Learning. *Proceedings of the VLDB Endowment (PVLDB)* 14, 3 (2020), 307–319. https://doi.org/10.14778/3430915.3430921

[27] Yuyang Dong, Kunihiro Takeoka, Chuan Xiao, and Masafumi Oyamada. 2021. Efficient Joinable Table Discovery in Data Lakes: A High-Dimensional Similarity-Based Approach. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 456–467. https://doi.org/10.1109/ICDE51399.2021.00046

[28] Falco Dürsch, Axel Stebner, Fabian Windheuser, Maxi Fischer, Tim Friedrich, Nils Strelow, Tobias Bleifuß, Hazar Harmouch, Lan Jiang, Thorsten Papenbrock, and Felix Naumann. 2019. Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*. 219–228. https://doi.org/10.1145/3357384.3357916

[29] Julian Eberius, Katrin Braunschweig, Markus Hentsch, Maik Thiele, Ahmad Ahmadov, and Wolfgang Lehner. 2015. Building the Dresden Web Table Corpus: A Classification Approach. In *Proceedings of the IEEE/ACM International Symposium on Big Data Computing (BDC)*. 41–50. https://doi.org/10.1109/BDC.2015.30

[30] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. 2022. MATE: Multi-Attribute Table Extraction. *Proceedings of the VLDB Endowment (PVLDB)* 15, 8 (2022), 1684–1696. https://doi.org/10.14778/3529337.3529353

[31] Ju Fan, Meiyu Lu, Beng Chin Ooi, Wang-Chiew Tan, and Meihui Zhang. 2014. A Hybrid Machine-Crowdsourcing System for Matching Web Tables. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 976–987. https://doi.org/10.1109/ICDE.2014.6816716

[32] David Gale and Lloyd S. Shapley. 1962. College Admissions and the Stability of Marriage. *American Mathematical Monthly* 69, 1 (1962), 9–15. https://doi.org/10.2307/2312726

[33] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*.

[34] Xuedong Huang, James Baker, and Raj Reddy. 2014. A Historical Perspective of Speech Recognition. *Communications of the ACM* 57, 1 (2014), 94–103. https://doi.org/10.1145/2500887

[35] Madelon Hulsebos, Çağatay Demiralp, and Paul Groth. 2023. GitTables: A Large-Scale Corpus of Relational Tables. *Proceedings of the ACM on Management of Data (PACMMOD)* 1, 1, Article 30 (2023), 17 pages. https://doi.org/10.1145/3588710

[36] Ihab F. Ilyas and Xu Chu. 2019. *Data Cleaning*. https://doi.org/10.1145/3310205

[37] Paul Jaccard. 1912. The Distribution of the Flora in the Alpine Zone. *New Phytologist* 11, 2 (1912), 37–50. https://doi.org/10.1111/j.1469-8137.1912.tb05611.x

[38] Youri Kaminsky, Eduardo H. M. Pena, and Felix Naumann. 2023. Discovering Similarity Inclusion Dependencies. *Proceedings of the ACM on Management of Data (PACMMOD)* 1, 1, Article 75 (2023), 24 pages. https://doi.org/10.1145/3588929

[39] Aamod Khatiwada, Roee Shraga, Wolfgang Gatterbauer, and Renée J. Miller. 2022. Integrating Data Lake Tables. *Proceedings of the VLDB Endowment (PVLDB)* 16, 4 (2022), 932–945. https://doi.org/10.14778/3574245.3574274

[40] Maximilian Koch, Mahdi Esmailoghli, Sören Auer, and Ziawasch Abedjan. 2023. Duplicate Table Detection with Xash. In *Proceedings of the Conference on Database Systems for Business, Technology and Web (BTW)*. 367–390. https://doi.org/10.18420/BTW2023-18

[41] Oliver Lehmberg and Christian Bizer. 2017. Stitching Web Tables for Improving Matching Quality. *Proceedings of the VLDB Endowment (PVLDB)* 10, 11 (2017), 1502–1513. https://doi.org/10.14778/3137628.3137657

[42] Oliver Lehmberg, Dominique Ritze, Robert Meusel, and Christian Bizer. 2016. A Large Public Corpus of Web Tables containing Time and Context Metadata. In *Proceedings of the International World Wide Web Conference (WWW),*

Companion Volume. 75–76. https://doi.org/10.1145/2872518.2889386
[43] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2020. *Mining of Massive Datasets*. http://www.mmds.org
[44] Xian Li, Xin Luna Dong, Kenneth Lyons, Weiyi Meng, and Divesh Srivastava. 2012. Truth Finding on the Deep Web: Is the Problem Solved? *Proceedings of the VLDB Endowment (PVLDB)* 6, 2 (2012), 97–108. https://doi.org/10.14778/2535568.2448943
[45] Girija Limaye, Sunita Sarawagi, and Soumen Chakrabarti. 2010. Annotating and Searching Web Tables Using Entities, Types and Relationships. *Proceedings of the VLDB Endowment (PVLDB)* 3, 1 (2010), 1338–1347. https://doi.org/10.14778/1920841.1921005
[46] Stéphane Lopes, Jean-Marc Petit, and Farouk Toumani. 2002. Discovering interesting inclusion dependencies: application to logical database tuning. *Information Systems* 27, 1 (2002), 1–19. https://doi.org/10.1016/S0306-4379(01)00027-8
[47] Bruce T. Lowerre. 1976. *The HARPY Speech Recognition System*. Ph.D. Dissertation. Carnegie Mellon University.
[48] Fatemeh Nargesian, Ken Q. Pu, Erkang Zhu, Bahar Ghadiri Bashardoost, and Renée J. Miller. 2020. Organizing Data Lakes for Navigation. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1939–1950. https://doi.org/10.1145/3318464.3380605
[49] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data Lake Management: Challenges and Opportunities. *Proceedings of the VLDB Endowment (PVLDB)* 12, 12 (2019), 1986–1989. https://doi.org/10.14778/3352063.3352116
[50] Fatemeh Nargesian, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. 2018. Table Union Search on Open Data. *Proceedings of the VLDB Endowment (PVLDB)* 11, 7 (2018), 813–825. https://doi.org/10.14778/3192965.3192973
[51] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. 2015. Divide & Conquer-based Inclusion Dependency Discovery. *Proceedings of the VLDB Endowment (PVLDB)* 8, 7 (2015), 774–785. https://doi.org/10.14778/2752939.2752946
[52] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian Suchanek. 2020. YAGO 4: A Reason-able Knowledge Base. In *Proceedings of the Extended Semantic Web Conference (ESWC)*. 583–596. https://doi.org/10.1007/978-3-030-49461-2_34
[53] Martin Potthast, Benno Stein, and Robert Gerling. 2008. Automatic Vandalism Detection in Wikipedia. In *Proceedings of the European Conference on Information Retrieval (ECIR)*. 663–668. https://doi.org/10.1007/978-3-540-78646-7_75
[54] Dominique Ritze, Oliver Lehmberg, and Christian Bizer. 2015. Matching HTML Tables to DBpedia. In *Proceedings of the ACM International Conference on Web Intelligence, Mining and Semantics (WIMS)*. Article 10, 6 pages. https://doi.org/10.1145/2797115.2797118
[55] Paolo Sottovia, Matteo Paganelli, Francesco Guerra, and Yannis Velegrakis. 2019. Finding Synonymous Attributes in Evolving Wikipedia Infoboxes. In *Proceedings of the European Conference on Advances in Databases and Information Systems (ADBIS)*. 169–185. https://doi.org/10.1007/978-3-030-28730-6_11
[56] Kavitha Srinivas, Julian Dolby, Ibrahim Abdelaziz, Oktie Hassanzadeh, Harsha Kokel, Aamod Khatiwada, Tejaswini Pedapati, Subhajit Chaudhury, and Horst Samulowitz. 2023. LakeBench: Benchmarks for Data Discovery over Data Lakes. *arXiv preprint* (2023). https://doi.org/10.48550/arXiv.2307.04217
[57] Elisabeth R. M. Tillier and Robert L. Charlebois. 2009. The human protein coevolution network. *Genome Research* 19, 10 (2009), 1861–1871. https://doi.org/10.1101/gr.092452.109
[58] Fabian Tschirschnitz, Thorsten Papenbrock, and Felix Naumann. 2017. Detecting Inclusion Dependencies on Very Many Tables. *ACM Transactions on Database Systems (TODS)* 42, 3, Article 18 (2017), 29 pages. https://doi.org/10.1145/3105959
[59] Liane Vogel and Carsten Binnig. 2023. WikiDBs: A Corpus Of Relational Databases From Wikidata. In *Proceedings of the VLDB Workshops*. https://ceur-ws.org/Vol-3462/TADA3.pdf
[60] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. 8413–8426. https://doi.org/10.18653/v1/2020.acl-main.745
[61] Jiang Zhan and Shan Wang. 2007. ITREKS: Keyword Search over Relational Database by Indexing Tuple Relationship. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*. 67–78. https://doi.org/10.1007/978-3-540-71703-4_8
[62] Yi Zhang and Zachary G. Ives. 2020. Finding Related Tables in Data Lakes for Interactive Data Science. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1951–1966. https://doi.org/10.1145/3318464.3389726
[63] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 847–864. https://doi.org/10.1145/3299869.3300065
[64] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH Ensemble: Internet-Scale Domain Search. *Proceedings of the VLDB Endowment (PVLDB)* 9, 12 (2016), 1185–1196. https://doi.org/10.14778/2994509.2994534

Received July 2023; revised October 2023; accepted November 2023