Discovering Similarity Inclusion Dependencies

YOURI KAMINSKY*, Hasso Plattner Institute, University of Potsdam, Germany
EDUARDO H. M. PENA, Federal University of Technology – Paraná, Brazil
FELIX NAUMANN, Hasso Plattner Institute, University of Potsdam, Germany

Inclusion dependencies (INDs) are a well-known type of data dependency, specifying that the values of one column are contained in those of another column. INDs can be used for various purposes, such as foreign-key candidate selection or join partner discovery. The traditional notion of INDs is based on clean data, where the dependencies hold without exceptions. Unfortunately, data often contain errors, preventing otherwise valid INDs from being discovered. A typical response to this problem is to relax the dependency definition using a similarity measure to account for minor data errors, such as typos or different formatting. While this relaxation is known for functional dependencies, for inclusion dependencies no such relaxation has been defined.

We formally introduce *similarity inclusion dependencies*, which relax the inclusion by demanding the existence only of sufficiently similar values. Similarity inclusion dependencies can fulfill traditional IND use cases, such as foreign-key candidate discovery, even in the presence of dirty data. We present Sawfish, the first algorithm to discover all similarity inclusion dependencies in a given dataset efficiently. Our algorithm combines approaches for the discovery of traditional INDs and string similarity joins with a novel sliding-window approach and lazy candidate validation. Our experimental evaluation shows that Sawfish can outperform a baseline by a factor of up to 6.5.

CCS Concepts: • **Information systems → Information integration**; *Data mining*.

Additional Key Words and Phrases: data profiling, data lake, joinability

ACM Reference Format:
Youri Kaminsky, Eduardo H. M. Pena, and Felix Naumann. 2023. Discovering Similarity Inclusion Dependencies. *Proc. ACM Manag. Data* 1, 1, Article 75 (May 2023), 24 pages. https://doi.org/10.1145/3588929

1 INCLUSION DEPENDENCY

Data profiling is the process of extracting metadata from datasets. Data dependencies are an important type of metadata and, thus, have a crucial role in data profiling. There are different forms of data dependencies, e.g., functional dependencies (FDs) and inclusion dependencies (INDs). In particular, INDs express that the tuples of one column-combination are contained in the tuples of another column-combination. We call an IND unary, if it holds between two individual columns. INDs help data practitioners to understand and structure unknown data, in particular, in discovering foreign key candidates and joinable partners [20]. Moreover, INDs have assisted schema design in [15] and can improve query execution [12].

*Corresponding author (youri.kaminsky@hpi.de)

Authors' addresses: Youri Kaminsky, Hasso Plattner Institute, University of Potsdam, Germany, youri.kaminsky@hpi.de; Eduardo H. M. Pena, Federal University of Technology – Paraná, Campo Mourão, Paraná, Brazil, eduardopena@utfpr.edu.br; Felix Naumann, Hasso Plattner Institute, University of Potsdam, Germany, felix.naumann@hpi.de.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2836-6573/2023/5-ART75 $15.00
https://doi.org/10.1145/3588929

Traditional IND discovery assumes clean data: *all* tuples of the dependent column-combination must be exactly *equal* in the referenced column-combination. However, the ever-increasing volume of data also leads to more "dirty" data [18]. Thus, relaxed dependencies have been introduced to deal with erroneous data [4].

One example of relaxed dependencies is matching dependencies (MDs), which generalize the concept of functional dependencies (FDs) [23]. For a traditional FD, the tuples on the dependent side must be equal for all equal tuples on the determinant side. In contrast, MDs use similarity measures instead of the strict equality constraint. In other words, an MD holds if, for all *similar* tuples on the determinant side, all tuples on the dependent side are also *similar*. MDs allow data practitioners to address typical use cases of FDs, such as schema normalization, even in the presence of dirty data. Moreover, MDs can also be used for duplicate detection [23]. A feature of MDs is that they support arbitrary similarity measures and a configurable similarity threshold to balance the error tolerance and over generalization.

There is no corresponding notion yet for INDs to allow for similar values. Thus, we introduce similarity inclusion dependencies (sINDs). In contrast to traditional INDs, sINDs use a similarity measure to define inclusion. An sIND holds if, for all dependent values, there exists a referenced value that is at least *similar*. Like MDs, sINDs support arbitrary similarity measures and configurable similarity thresholds. For this work, we consider representatives of both edit-based and token-based similarity measures: the edit distance and the Jaccard similarity.

sINDs are a natural way to handle dirty data. Besides traditional IND use cases, sINDs can also identify sources of possibly erroneous data in data lakes. If a candidate IND does not hold, but its respective sIND holds, the data can be analyzed and used to identify erroneous relations. Either only one of the relations contains errors, so we can use the other relation to fix these, or both relations contain errors, and the data needs to be cleaned more thoroughly.

To illustrate the usefulness of sINDs, we present an example in Table 1. The tables are for a fictitious soccer tournament. While one table shows the results after all teams played against each other, the other table presents an aggregation of the participation forms of all goalkeepers in the tournament. We would assume that the values of the *club* column of Table 1b are contained in the values of the *name* column of Table 1a. However, multiple goalkeepers made minor spelling mistakes in their club name. Despite these mistakes, we want to discover the dependency. On the one hand, we are interested in joining these columns to know which goalkeeper played for the most successful team. On the other hand, we could perform a data cleaning task, because all mistakes are only in the *club* column of Table 1b. Thus, there is a matching counterpart in the other column to automatically correct the errors.

Table 1. Example relations of a soccer tournament

(a) Final Results

results	
name	points
SpVgg Beelitz	4
Potsdamer SC	9
SpVgg Bernau	4
VfL Potsdam	0

(b) Participating Goalkeepers

goalie	
p_id	club
202	SpVgg Beelitz
216	SpVgg Beelittz
469	Potsdamer SC
617	SpVgg Be_nau
692	ViL Potsdam
853	Potsdamer SC

There are other known extensions to INDs that can deal with dirty data. One example is partial INDs [2], which allow a certain portion of the dependent tuples to not be present in any form in the referenced tuples. Therefore, partial INDs are especially useful when dealing with incomplete columns. However, partial INDs provide no insight into what kind of matching failures occur. If a significant portion of tuples have a similar, but not equal counterpart, no partial INDs would be found. Since half of the goalkeepers in our example have misspelled their club names, we would need to set the error threshold above 50% to find a partial IND. In a typical database, this threshold would lead to many spurious dependencies. A special form of partial dependencies are conditional INDs (cINDs), which hold only for tuples that fulfil a certain condition [3, 1].

Another example are approximate INDs [13]. They are discovered on a sample of the complete data and show that an IND holds only with a certain probability. This relaxation is used to increase the discovery speed, but not the generality of the found IND. Moreover, approximate INDs also provide no insight into data errors. In our example, typical sampling strategies would lead to different value sets for the columns. Therefore, we would not find a dependency in our samples.

Apart from introducing the concept of similarity inclusion dependencies, we present **Sawfish**, the first approach to efficiently discover sINDs in a given dataset. In particular, we make the following contributions:

(1) We introduce the formal concept of similarity inclusion dependencies.
(2) We propose Sawfish, an efficient approach to discover all unary sINDs in a given dataset using the edit-distance and the Jaccard similarity measure.
(3) We offer an in-depth evaluation of Sawfish, comparing our approach to the state-of-the-art IND discovery algorithm Binder [22] and a naive baseline. We show how Sawfish scales with the size of the input data and different algorithm configurations.
(4) We present a case study that explores the usefulness of the discovered sINDs. We have manually built a ground-truth with over 1000 sINDs and publicly provide the annotations.
(5) We integrate Sawfish on the Metanome data profiling platform [21], so it can be easily used with a variety of datasets and compared to other data profiling algorithms. All code is made publicly available.

The remainder of this work is structured as follows. In Section 2 we present related research on inclusion dependencies, other relaxed dependencies and string similarity joins. We formally define sINDs in Section 3. Section 4 shows how to efficiently discover sINDs and presents the general principle of Sawfish. We present an exhaustive evaluation of Sawfish in Section 5. Finally, Section 6 draws a conclusion and gives insights into the limitations of Sawfish and provides an outlook on future work.

2 RELATED WORK

Although there has been no research on similarity inclusion dependencies yet, three research areas overlap with this work: traditional inclusion dependency discovery, other relaxations of dependencies, and string similarity joins.

Inclusion dependencies (INDs) are a well-known type of data dependency and have several algorithms to discover them from data— see Dürsch et al. for an overview of approaches [9]. There are approaches for both unary and n-ary IND discovery. De Marchi et al. presented an early algorithm to discover unary INDs by intersecting the attribute sets for each value [6]. Many also discovers unary INDs, but focuses on the problem of finding INDs in millions of tables [24]. A prominent representative for n-ary, i.e., inclusion between tuples of attribute lists, IND discovery is Zigzag [7]. It combines both up- and downwards pruning to discover maximal n-ary INDs efficiently.

In particular, Sawfish adapts a few techniques from the algorithm Binder [22]. Initially, Binder splits the input data into buckets for each attribute. It uses hash-partitioning to distribute the values evenly and place equal values into the same bucket. If main memory is exhausted, the largest buckets are written to disk, allowing Binder to scale to large datasets. To validate IND candidates, Binder loads data partitions into the main memory. Each partition contains one bucket from each attribute with the same bucket number. As the bucket number relates to the hash value, equal values of different attributes are in the same partition. If a partition is too large for main memory, it can be lazily refined into subpartitions. Binder deduces which INDs still hold in the data for each partition and prunes the other candidates. After processing every partition, Binder outputs all remaining candidates as valid INDs.

The input-handling of Sawfish also splits the column values into different buckets. However, it cannot use hash-partitioning due to the similarity measure, so it uses an adaptive main memory handling—presented in detail in Section 4.1. Sawfish also loads chunks of data into main memory and checks remaining sIND candidates to validate sIND candidates. We detail our validation strategy in Section 4.2.

Similarly to how sINDs extend the concept of INDs, there are examples of such relaxations of other dependencies— see Caruccio et al. for an overview of relaxed FD approaches [4]. As mentioned in the introduction, matching dependencies (MDs) are a prominent extension of functional dependencies (FDs). Like sINDs, they incorporate a similarity measure to find additional dependencies in the data. MDs were first introduced by Fan [10]. Schirmer et al. showed how to efficiently find all MDs in a database [23]. Their HyMD algorithm combines two techniques to find MDs: lattice traversal and inference from record pairs. The lattice comprises all candidate MDs in a sorted order. Therefore, it can be traversed to find minimal MDs. To quickly find counterexamples for an MD candidate, HyMD can compare record pairs and infer which MDs might still be minimal. After comparing every record pair, the inferred MDs are the correct solution set. We cannot use inference from record pairs, because we cannot infer the validity of an sIND from only a record pair. Since a single value can be similar to multiple other values, a record pair that shows that a value is not similar to another value does not disprove the validity of an sIND. HyMD computes the similarity of every value pair beforehand to access it quickly when validating MDs. Our approach avoids this preprocessing and computes the similarity while validating. Thus, we can save many unnecessary computations, because only those value pairs that we actually process are compared.

Finally, string similarity joins and sINDs share a related sub-problem. Like sINDs, for a large set of strings, string similarity joins need to identify similar strings based on some similarity measure to execute the join. Yu et al. provide an overview of different approaches [27]. Most methods either use specific substrings or a tree-like data structure to compute the similarity of two strings. For example, the algorithm TrieJoin uses a trie to efficiently calculate the similarity [11].

Sawfish uses the underlying method of PassJoin [17] to find similar strings for a dependent value when using the edit distance as its similarity measure. The author's observation is based on the pigeonhole principle: Assume an edit distance threshold τ, two strings x, y and $ED(x, y) \leq \tau$. If we split x into $\tau + 1$ disjoint segments, there exists a substring of y that is equal to one of the segments. Otherwise, we would need at least one edit operation for each segment to transform it to a substring of y. However, this violates our assumption that $ED(x, y) \leq \tau$. For example, given the two soccer club names, *Potsdamer SC* and *SpVgg Beelitz*, we want to know if their edit distance is within one, i.e. $\tau = 1$. Therefore, we segment Potsdamer SC in $\tau + 1 = 2$ equally-sized segments: `Potsda` and `mer SC`. We observe that there is no substring in `SpVgg Beelitz` that matches any of the segments. Therefore, we know that these club names are not similar without computing their actual edit distance. This segment filter effectively prunes dissimilar string pairs. Moreover, based on this filter, an inverted index of the segments to the indexed elements can be built. Additionally,

Li et al. presented techniques to reduce the number of substrings that need to be compared to the segments. They also improve the exact edit distance computation based on their segmentation filter. We detail our usage of the validation techniques in Section 4.

When using the Jaccard similarity, SAWFISH uses and adapts the SCANCOUNT [27] method. Assume a given Jaccard similarity threshold δ, a string x and a set of strings Y. First, for each $y \in Y$, it stores a mapping of each token to its parent string. For each token of x, SCANCOUNT retrieves the list of parent strings that contain that token and maintains a count of each occurrence of each parent string in all lists. Next, it computes a threshold $T = \frac{\delta}{1+\delta}(|\text{token}(x)| + |\min(\text{token}(y))|)$. Afterwards, all strings $y \in Y$ that have a count $\geq T$ are directly compared to x to compute their actual Jaccard similarity. We improve this version for our needs and show the modified version in Section 4. Note that we cannot use the popular MINHASH [26] to discover similar strings, because it is only an estimation of the Jaccard similarity.

3 SIMILARITY INCLUSION DEPENDENCY

Let R and S be two relations of a database D (with an instance I), and let A and B be two attributes. The notations $R[\mathsf{A}]$ and $S[\mathsf{B}]$ indicate the projections of R and S on A and B respectively. A unary inclusion dependency (IND) $R[\mathsf{A}] \subseteq S[\mathsf{B}]$ can be defined using quantifiers as follows:

$$R[\mathsf{A}] \subseteq S[\mathsf{B}] \iff \forall r \in I(R), \exists s \in I(S) : r[\mathsf{A}] = s[\mathsf{B}]$$

The values $R[\mathsf{A}]$ are called dependent values, whereas $S[\mathsf{B}]$ are called referenced values. R and S can be the same relation ($R = S$), but most typical use-cases are interested in INDs across relations.

We extend the definition of INDs to accommodate similarity measures, thereby introducing similarity inclusion dependencies (sINDs). Let $\sigma(x, y) \to [0, 1]$ be a similarity measure and let $\approx_\sigma$ be an operator that checks whether two values are similar for σ and a threshold. An sIND $R[\mathsf{A}] \subseteq_\sigma S[\mathsf{B}]$ can be defined as follows:

$$R[\mathsf{A}] \subseteq_\sigma S[\mathsf{B}] \iff \forall r \in I(R), \exists s \in I(S) : r[\mathsf{A}] \approx_\sigma s[\mathsf{B}]$$

In simpler terms, for each dependent value exists a *similar* referenced value given the similarity measure σ and some threshold.

We can identify trivial sINDs that correspond to trivial INDs. First, an empty column references every other column. Since there exist no values on the dependent side, every statement using the universal quantifier is trivially true. Second, every column trivially references itself: $R[\mathsf{A}] \subseteq_\sigma R[\mathsf{A}]$ always holds. Thus, we ignore such reflexive sIND candidates. Like in traditional IND discovery, we also ignore `null` values [22], i.e., in the presence of a null value in the dependent column, we do not demand a null value or a value similar to null in the referenced column.

SAWFISH supports both edit-based and token-based similarity measures. As a prominent representative of edit-based similarity measures, we explore the Levenshtein distance. The Levenshtein distance, also known as edit distance (*ED*), is defined as the minimum number of edit operations to transform one string into another [16]. There are three possible edit operations: substitutions can exchange any character of the string with another character; insertions allow the addition of a character at any position of the string; deletions allow the removal of any character of the string. To determine whether two strings are considered similar, the Levenshtein distance is compared to a user-defined threshold τ.

We identify another special case that is specific to sINDs and the Levenshtein distance. Let τ be the user-defined edit distance threshold. Each value with $\leq \tau$ characters is similar to every other value with $\leq \tau$ characters, because we can construct every word that consists of $\leq \tau$ characters with τ edit operations. Therefore, all pairwise columns that contain only values with $\leq \tau$ characters automatically form sINDs. However, these sINDs do not have any meaning other than that the

columns contain only short strings. We call these sINDs *simple* sINDs and exclude them from our analysis.

Besides supporting the absolute edit distance (ED), SAWFISH can also be used with a normalized edit distance (NED) threshold δ. The normalized edit distance is defined as follows:

$$NED(x, y) = \frac{ED(x, y)}{max(|x|, |y|)}$$

Applying this definition allows us to convert a normalized similarity threshold into an absolute edit distance value depending on the maximum length of the two involved strings. Given the longer string length l and the normalized threshold δ, we can calculate the absolute threshold τ as $\tau = (1 - \delta) \cdot l$.

Since we preprocess the data, we can calculate individual absolute thresholds for each occurring length beforehand. However, we observed that we discover fewer dependencies when using a normalized threshold. Normalization yields a minimum string length before we allow a single edit operation, i.e., $l \geq \frac{1}{\delta}$. This shortcoming of the normalized edit distance is especially noticeable in sINDs, because they are typically found in columns that contain shorter values.

Therefore, we created a hybrid mode for SAWFISH that always allows at least a single edit operation, but uses a normalized threshold for larger string lengths. Given two strings x and y and a normalized threshold δ, the hybrid mode of SAWFISH considers the strings to be similar as follows:

$$x \sim_\delta y = \begin{cases} ED(x, y) \leq 1 & \text{if } max(|x|, |y|) * (1 - \delta) \leq 1 \\ NED(x, y) \leq 1 - \delta & \text{otherwise} \end{cases}$$

As the representative of token-based measures, we choose the Jaccard similarity (JAC). The Jaccard similarity is defined as the number of tokens in the intersection divided by the number of tokens in the union of two token sets. To determine whether two strings are considered similar, the resulting ratio is compared to a user-defined threshold δ. There are multiple ways to tokenize strings, e.g. using n-grams. In this work, we tokenize strings by splitting them up at their whitespaces. There are no simple sINDs for the Jaccard similarity, because it is a relative similarity measure.

Lastly, we do not expect interesting or meaningful inclusion dependencies among very long values, e.g., abstracts of scientific papers, because we are not typically joining over those columns. Therefore, we ignore columns that include any value with more than 50 characters or 10 tokens.

4 THE SAWFISH ALGORITHM

SAWFISH stands for **S**imilarity **AW**are **F**inder of **I**nclusion dependencies via a **S**egmented **H**ash-index. SAWFISH preprocesses the dataset, generates some metadata, and marks the sIND candidates requiring validation. Then, it performs the actual sIND discovery using an inverted index, which is used to identify possibly similar strings. We illustrate the discovery process by following the example from Table 1. We use the ED mode of SAWFISH and set an edit distance threshold $\tau = 1$.

SAWFISH currently supports both the Levenshtein distance and the Jaccard similarity, but it can easily be extended to any similarity measure that has the following properties. First, it must be possible to prune value pairs based on an ordered numeric measure, e.g., length of values for Levenshtein and number of tokens for Jaccard. Second, it must be possible to prune value pairs based on the inequality of subparts. This property allows the creation of an inverted index, where substrings point to their parent strings. If two values do not share a substring, they must be dissimilar according to the similarity measure. Thereby, SAWFISH can avoid validating dissimilar string pairs. Other similarity measures that fulfill these properties include the Hamming distance, the Cosine similarity, and the Dice similarity.

length	**results** name	**goalie** club	**indexed values when validating length** 11	12	13	14
11	VfL Potsdam	ViL Potsdam, SpVgg Benau				
12	Potsdamer SC, SpVgg Bernau	Potsdamer SC				
13	SpVgg Beelitz	SpVgg Beelitz				
14		SpVgg Beelittz				

Fig. 1. Example relation after preprocessing, including a visualization of the sliding length window; showing the indexed values for each currently validated length

4.1 Preprocessing

We transform the input into a particular data format and extract additional metadata. First, we group all distinct values of each column by their lengths. Here, length is the number of characters of the string for the *ED* case, whereas it is the number of tokens of the string for the *JAC* case. Figure 1 illustrates this grouping structure for our example relations and the *ED* case. The right-hand side of the figure visualizes the sliding length window that will be explained later.

We can discard any column containing at least one value with more than fifty characters or more than ten tokens. During input reading, we collect the minimum and maximum length of all values for each column.

To preprocess the data without requiring entire tables to fit into main memory, we can evict buckets by writing them to disk. The memory check occurs after reading a configurable number of values, which defaults to 100. If we need to free memory, we first identify the largest column. Therefore, we obtain the column sizes of the preprocessing routine and compare them against each other. We evict the largest column first because we free most of the memory with minimal disk I/O. To obtain the bucketized view of the data without having to read the entire column again later, we write the buckets separately to disk. We reset all data structures to free the memory that we have just written to disk. We repeat this process until we fall below the memory limit. After processing the entire input, we deduplicate all evicted buckets again. Due to their early eviction, there might be duplicate values that need to be eliminated.

There are up to n^2 unary sIND candidates. However, we can prune some candidates based on the collected metadata. First, we prune trivial sINDs, e.g., reflexive sINDs and sINDs with empty columns. Second, we prune the simple sINDs in the *ED* case, i.e., sINDs where both columns contain only values smaller than τ characters. Third, we prune sIND candidates that cannot hold due to their length difference. In the *ED* case, the maximum length difference between two similar strings is τ, because we can only perform τ insert or delete operations. Therefore, we can prune any candidate where the longest value of the dependent column is longer than the longest value of the referenced column plus τ. Similarly, we can prune any candidate, where the shortest value of the dependent column is shorter than the shortest value of the referenced column minus τ. In the *JAC* case, any value x can only be similar to values y where $|x| \cdot \delta \leq |y| \leq \frac{|x|}{\delta}$. Therefore, we can apply a similar pruning for the shortest and longest values of a column. We store the remaining candidates by assigning each referenced column all columns that possibly depend on it.

In our example, we generate two candidate sINDs for $\tau = 1$: *results[name]* $\subseteq$ *goalie[club]* and *goalie[club]* $\subseteq$ *results[name]*. All candidates containing *results[points]* and *goalie[p_id]* are pruned based on their length difference.

4.2 Basic Discovery Approach

As with the detection of traditional INDs, the general idea of SAWFISH is that it is sufficient to find a single counterexample to invalidate an sIND candidate. Due to the universal quantifier in the sIND definition, there needs to be at least one similar referenced value for each dependent value. However, it is infeasible to directly compare every dependent value to every referenced value. To reduce the number of comparisons of string pairs, SAWFISH uses an inverted index. After building the index for a referenced column, each dependent column is probed against the index. For each dependent value that matches an index entry, the similarity of the resulting value pair is validated. If each dependent value is similar to at least one referenced value, the sIND is emitted as a valid dependency.

4.3 Inverted Index

The inverted index is the centerpiece of SAWFISH. It stores a mapping of deduplicated substrings to their input strings. In *JAC* mode, we simply store a mapping of each token to its input string.

In *ED* mode, we need to segment every string and store the mapping of each segment to its input string. This procedure is based on the segment-based filter of the PassJoin algorithm [17]. SAWFISH does so in two steps. First, it generates the start positions of the segments. Since having segments with roughly the same size performs well in practice [17], the string length is divided by the number of segments. If the result has a remainder, we use shorter segments at the beginning and larger segments at the end. Due to the length grouping in the preprocessing, we need to compute the start positions only once for each length. Second, SAWFISH obtains the substrings by using the previously generated start positions. It maps each substring to its parent string. However, not all substrings are placed on the same map. There is a map for each segment position to access the segment sets individually.

Table 2 shows the inverted index for the `name` column in our example. We use $\tau = 1$, so every value is split into two segments. The segments within each segment position are deduplicated; thus, for instance, `SpVgg` is presented only once.

Table 2. Inverted index of the `name` column for $\tau = 1$

Segment 1	Segment 2	Original value
SpVgg	Bernau Beelitz	SpVgg Bernau SpVgg Beelitz
VfL P	otsdam	VfL Potsdam
Potsda	mer SC	Potsdamer SC

To reduce memory consumption of the inverted index, we employ a technique similar to `std::string_view` of the C++ standard. Since all strings and their substrings are immutable in the index, we do not need to copy the characters, but can use a view instead.

Sliding Length Window. Based on the intuition of the length filter, we only need to compare a dependent value y to the index values within a certain interval. We do not use the entire index; instead, we only use the index blocks required to validate the current dependent values. We can

iterate the dataset length-wise and build new indices only on-demand while removing unused indices. This technique resembles a sliding window through the occurring lengths of the dataset, as illustrated on the right-hand side of Figure 1.

We take advantage of the length buckets from preprocessing to iterate value lengths. We iterate from the longest to the shortest length because we expect fewer accidental matches for longer strings. Thus, we can prune candidates earlier. Let l be the length of the current iteration. In every iteration, we discard the index with length $l + \tau + 1$ or $\frac{l}{\delta} + 1$, respectively. Next, we load the bucket with length $l - \tau$, or $l\delta$, respectively, and create the inverted index. Afterwards, we validate the dependent values with length l. We loop until all lengths are processed, or there are no dependent candidates for our referenced column left.

There are two main advantages of the *sliding length window*. First, there is no need to build the entire index if a column is no longer referenced by any other column. Therefore, we can abort the execution early. Second, the indices in the main memory are smaller, so they are more likely to fit into the cache lines and can be accessed faster.

4.4 Dependent Value Validation

This section presents our approach to validating the individual values from the dependent columns of candidate sINDs. This phase shows the largest difference between the ED and JAC modes.

4.4.1 ED mode. To validate values in ED mode, we show how to generate substrings compatible with the inverted index. Then, we describe our method for efficiently using the inverted index in our use case. Also, we demonstrate how to use the inverted index to speed up similarity computation.

Substring Generation. If two strings x and y are similar, at least one substring of y matches a segment of x. Since we created the inverted index based on the segments, we need to generate all substrings that can match a segment. However, we do not want to probe the index for all substrings of y. Instead, we use the techniques by Li et al. to reduce the substring comparisons [17]. We compare only equally-sized substrings to the segments. Furthermore, we limit the start positions of the substrings to be close to the start positions of the segments. Finally, we employ a multi-match aware technique that skips unnecessary substrings.

Additionally, we need to compute substrings for multiple target lengths. Any string y can be similar to a string x, only if $|y| - \tau \leq |x| \leq |y| + \tau$. Therefore, we need to compute the substrings for every length in $[|y| - \tau, |y| + \tau]$. Due to these different target lengths, also the inverted index is divided into the different lengths of its values.

Index Probing. We need the generated substrings and their respective target length to probe the inverted index. Therefore, we iterate all possible length differences ld. For each target length, we first generate the substrings. Then we select the correct index for the target length. Finally, each substring for the i-th segment position is compared to the i-th segments of the selected index. If we find matching pairs of substring and segment for position i, we validate their actual similarity, as we describe in the next paragraph. Either one of the matches is similar to the dependent value, or we continue with checking the $i + 1$-th segment. To not process a referenced value multiple times, we keep a set of already validated values. Algorithm 1 shows the entire function.

This routine differs from the original PassJoin index probing because we need to find only *one* similar value, instead of *all* similar values [17]. Therefore, we reduce the number of unnecessary index accesses by validating the index matches earlier.

String Similarity Computation. After obtaining possibly similar string pairs from the inverted index, we validate their similarity by computing their exact edit distance. There is a well-known dynamic programming-based algorithm to calculate edit distances [25]. However, it is possible

Algorithm 1 Index Probing in *ED* mode

```
    Data: indices, value
    Result: existsSimilarString
 1: function ProbeIndex
 2:     for ld ← [−τ, τ] do
 3:         substrings ← GenerateSubstrings(value, ld, τ + 1)
 4:         index ← indices[len(value) + ld]
 5:         alreadyValidated ← ∅
 6:         for i ∈ [1, τ + 1] do
 7:             segmentMap ← index[i]
 8:             matches ← ∅
 9:             for all sub ∈ substrings[i] do
10:                 for all match ∈ segmentMap[sub] do
11:                     if match ∉ alreadyValidated then
12:                         matches ← ∪ match
13:                         alreadyValidated ← ∪ match
14:             if matches ≠ ∅ then
15:                 existsSimilarString ← ValidateMatches(value, matches)
16:                 if existsSimilarString then
17:                     return existsSimilarString
18:     return False
```

to use a technique from Li et al. to improve this algorithm by avoiding computing all matrix entries [17]. First, we are only interested in whether two strings are similar. Thus, we can abort the computation if we cannot obtain an edit distance below τ. Second, we know about a matching part between the strings due to the segment filter. Therefore, we can split the edit distance computation into the left and right parts of the match separately. Thus, we can use tighter thresholds. Finally, we compare the obtained edit distance to the user-defined edit distance threshold τ to decide whether the two strings are similar.

4.4.2 JAC mode. The method for validating the Jaccard similarity is significantly simpler than that for the Levenshtein distance. Since we know the number of tokens for the entries in the inverted index and the number of tokens of the dependent value, we can calculate the minimum number of tokens that need to match to be similar. This threshold can be computed as $T = \frac{\delta}{1+\delta}(|y| + |index|)$. We use the scan count method to identify similar strings [27]. After determining T, we simply count the number of exact matches between the tokens of the dependent value y and each index entry $index_i$. If there are more than T matches for any i, we can directly return that there is a sufficiently similar value for y and validate the next dependent value. Algorithm 2 shows the procedure in detail.

4.5 sIND Discovery

Sawfish combines the building blocks above – Algorithm 3 presents the entire procedure. For each referenced column, we build an index and validate all possible dependent columns. This approach explains the order in which we store the *candidates* in preprocessing. It allows us to build the index only once and go through all possible dependent columns without rebuilding it. We probe the inverted index for each dependent value and save the matches. However, if we do not find a single match for all substrings, we can directly discard the sIND candidate because we only need one

Algorithm 2 Index Probing in *JAC* mode

```
    Data: indices, value
    Result: existsSimilarString
 1: function PROBEINDEX
 2:     for indexLength ← [⌈δ|value|⌉, ⌊|value|/δ⌋] do
 3:         index ← indices[indexLength]
 4:         T ← δ/(1+δ) (|value| + indexLength)
 5:         counts ← {}
 6:         tokens ← TOKENIZE(value)
 7:         for all tok ∈ tokens do
 8:             for all match ∈ index[tok] do
 9:                 count[match]++
10:                 if count[match] ≥ T then
11:                     return True
12:     return False
```

counterexample. Otherwise, we validate the individual matches using the similarity measure. If one of the matches is indeed similar to the dependent value, we can jump to the next dependent value. Otherwise, we also can discard the candidate. If the dependent column is still in the *candidates* set of the referenced column after validating all dependent values, we can output a valid sIND between the dependent and the referenced column.

Validating one referenced column at a time is not optimal. While there might be situations where only one index fits into main memory, for most real-world datasets, multiple indexes do fit into main memory. Since we require the largest index to fit into the main memory, smaller indexes can coexist in the main memory. Therefore, we can build the inverted index for multiple columns at once, enabling us to use the available memory better. Moreover, since the columns of the indices might depend on each other, we do not need to load them into main memory just for validating the values, but we can also build the index for them. Nonetheless, we still need to decide which indices to fit into the main memory and model the decision after the bin packing problem.

The bin packing problem is NP-hard [19]. However, there exist multiple approximation techniques. The First Fit Decreasing (FFD) algorithm works by first sorting the integers (here: index sizes) and then selecting fitting elements. Dósa proved the tight bound of $FFD(b) \leq 11/9 OPT(b) + 6/9$, where *FFD(b)* is the number of bins used by FFD and *OPT(b)* the number of bins of the optimal solution [8]. We use FFD to select the columns that we process in each iteration. Thus, our column batching selection method works well because even in the worst-case scenario, our algorithm needs to run only slightly more often than in an optimal case.

5 EXPERIMENTAL EVALUATION

Our evaluation of Sawfish includes a comparison with two competitors, and we investigate the scaling behavior of Sawfish. Finally, we look into the actual dependencies that Sawfish produces, and examine their usefulness. We publicly provide all source code[1].

5.1 Experimental Setup

All experiments were run on an Ubuntu-based (20.04 LTS) server, equipped with two Intel Xeon E5-2650 processors and 256 GB of RAM. All algorithms are single-threaded, running on Java 11

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

Algorithm 3 Candidate Validation

```
    Data: candidates, columnBucketMap, τ                                ▷ From Preprocessing
    Result: all valid unary sINDs
 1: processed ← ∅
 2: while |processed| ≠ |columns| do
 3:     refColumns ← GetReferencedColumns(columnBucketMap, processed)
 4:     for l ← [50, 1] do
 5:         if ∀ ref ∈ refColumns: candidates[ref] = ∅ then
 6:             break
 7:         for all ref ∈ refColumns do
 8:             if candidates[ref] ≠ ∅ then
 9:                 columnIndices[ref][l + τ + 1] ← ∅
10:                 columnIndices[ref][l − τ] ← BuildIndex(columnBucketMap[ref][l − τ], τ + 1)
11:         for all ref ∈ refColumns do
12:             indices ← columnIndices[ref]
13:             for all dep ∈ candidates[ref] do
14:                 for all value ∈ columnBucketMap[dep] do
15:                     exisitsSimilarString ← ProbeIndex(indices, value)
16:                     if ¬ existsSimilarString then
17:                         candidates[ref].drop(dep)
18:                         break
19:                 if ¬ existsSimilarString then
20:                     break
21:     for all ref ∈ refColumns do
22:         for all dep ∈ candidates[ref] do
23:             output dep ⊆̃ ref
```

and reading their input data from CSV files. We do not use the entire main memory, but set explicit memory limits for the Java Virtual Machine. We use a 4 GB limit for all datasets, except for the *IMDB* dataset, which uses 32 GB. Moreover, we set a time limit of 24 hours, aborting executions that exceeded that time threshold. Unless stated otherwise, all experiments were run three times, and we present the mean of the runs.

Sawfish is compatible with the *Metanome* data profiling platform [21], which handles both file and database backed inputs and provides a unified view for the algorithms. We used the *metanome-cli* to conduct the experiments in a repeatable and efficient way.

We use four publicly available datasets[2]: three real-world datasets, and the synthetic TPC-H dataset. Table 3 shows the datasets and their characteristics. The number in parentheses indicates the number of ignored attributes (due to values containing more than 50 characters or more than ten tokens). The number of distinct values across all columns excludes the values of the ignored columns.

Because there exists no other sIND detection algorithm, we compare Sawfish to a traditional IND detection algorithm and a baseline. For the former, we choose the state-of-the-art algorithm Binder [22] using the authors' implementation. Binder solves a specialized task in comparison to Sawfish, implicitly setting $\tau = 0$. Thus, it can use techniques more tailored to the traditional IND discovery, as we explained in Section 2.

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

Table 3. Summary of datasets used in the experiments.

	size (MB)	cols (ignored)	rows	distincts
CENSUS	112	42 (0)	199 524	101 937
WIKIPEDIA	570	14 (3)	14 024 428	6 950 343
TPC-H	1430	61 (6)	6 001 215	9 547 611
IMDB	3790	108 (14)	36 244 344	47 539 970

Also, we compare Sawfish to a baseline solution for each similarity measure – an approach based on the original string similarity join algorithms PassJoin [17] and ScanCount [27]. Applying the definition of sINDs, we simply similarity-join each value of the dependent column one by one with the referenced column. If each value of the dependent column finds at least one join partner, the sIND between the dependent and the referenced column holds. For PassJoin, we based our implementation on the `EditDistanceJoiner` by the database group of Tsinghua University[3]. The provided algorithm is capable of handling only strings that are longer than the edit distance threshold. Therefore, Sawfish also ignores shorter strings when comparing both algorithms. Since Sawfish can handle strings that are shorter than the edit distance threshold, we process all strings up to length 50 in the other experiments. Additionally, we implement the commonly known length filter, which compares the minimum and maximum lengths of column pairs. For ScanCount, we did not find a suitable Java implementation, so we implemented a plain Java version ourselves.

5.2 Runtime Scalability

This section examines how Sawfish scales in different dimensions. We investigate the row and column scalability. Additionally, we analyze the runtime impact of the edit distance threshold.

5.2.1 Row Scalability. This experiment investigates the runtime scalability with the number of rows. We used random samples: starting with 10% of the input dataset and then gradually growing the sample size by 10%. We repeated this experiment 10 times and plotted the mean and standard deviation.

We present the results for the *CENSUS*, *WIKIPEDIA*, *TPC-H*, and *IMDB* (JAC mode only) datasets in Figure 2. We do not show the *IMDB* dataset in *ED* mode, as processing the entire dataset exceeds our time limit of 24 hours, as shown in Figure 7a. We set the edit distance threshold $\tau = 1$ and the Jaccard similarity threshold $\delta = 0.4$. The solid yellow and blue lines show the runtime of the JAC mode and *ED* mode, respectively. The numbers of sINDs are presented as dashed lines in the respective colors.

We observe that Sawfish exhibits a near perfect linear scaling, both in *ED* mode and JAC mode. We also observe that the runtime is not dependent on the result set size: in the *CENSUS* and the *WIKIPEDIA* datasets the result set size is almost constant, but the runtime increases similarly to the *TPC-H* and the *IMDB* datasets, where the result set size continuously grows. However, this independence of runtime and result set size is related to our experimental setup. Sawfish benefits from discarding sIND candidates early. Verifying an sIND candidate that is valid is most demanding because, for every dependent value, it probes the index and computes the exact edit distance at least once. Since we sampled from the input dataset, for all sINDs that do not appear in the result set of smaller data portions, there is a missing value in the referenced column. The validation effort for other values does not change because we cannot check the dependent value that references the

[3]https://github.com/lispc/EditDistanceClusterer

missing value first. Thus, the validation effort decreases only linearly. Nonetheless, we can expect Sawfish to scale linearly to even longer datasets.

Interestingly, the variance is higher in *ED* mode for the *WIKIPEDIA* dataset, while it is higher for the *TPC-H* dataset in *JAC* mode. For the *CENSUS* dataset, the overall variance is low due to the short runtime. The results for the *IMDB* dataset also exhibit little variance and follow the linear trend of the other datasets. There is even a slight decrease in mean runtime in *JAC* mode on the *TPC-H* dataset. However, these longer running samples do not necessarily contain more sINDs. These artifacts can be linked to the individual dataset characteristics. If we find dependent values that have many matches in the inverted index, but are in fact dissimilar to all referenced values, the validation takes longer. By increasing row set sizes, we also increase the number of "strong" counterexamples, i.e., counterexamples with no index matches. We observe a similar behavior in Section 5.2.3.

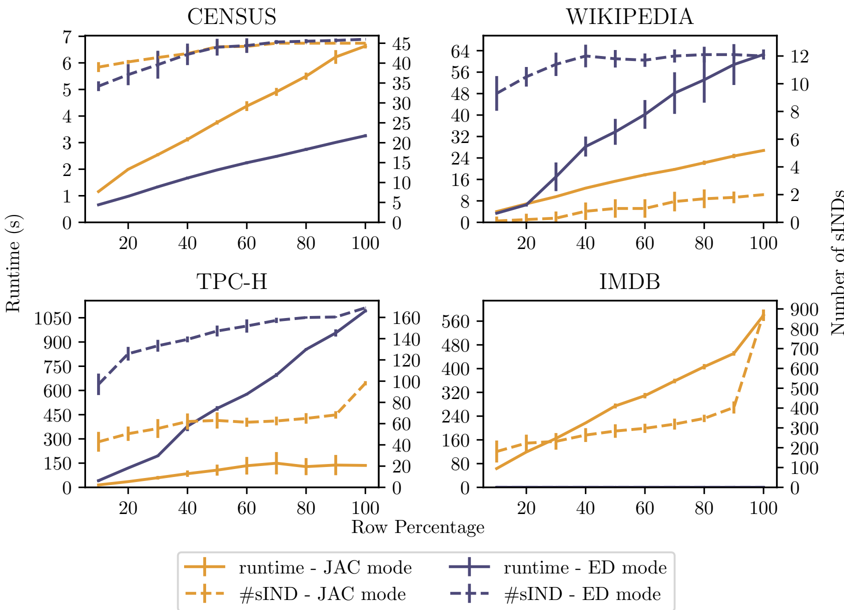


Fig. 2. Row scalability on different datasets (for *ED* mode $\tau = 1$, for *JAC* mode $\delta = 0.4$)

5.2.2 Column Scalability. Next, we investigate the scalability of Sawfish with an increasing number of columns. For this experiment, we randomly sampled column sets for each of the ten column set sizes. We used 30 samples for each size to accommodate the high variance from different column sets, because the runtime depends on the number of valid sINDs that happen to hold in that sample. To explain the variance, we take another look at our introductory example, regarding two sets of two columns each. Let the first set consist of column *results*[`name`] and column *goalie*[`p_id`], and the second consist of column *results*[`name`] and column *goalie*[`club`]. As the sIND *results*[`name`] $\subseteq$ *goalie*[`club`] as well as the symmetric counterpart are valid, Sawfish needs to iterate through both columns, build indices, and validate all dependent values. On the other hand, we can already determine after preprocessing that there are no sINDs in the first column set, because of their length difference. Thus, Sawfish can process it much faster. We investigate the impact of valid dependencies in Section 5.3. To emphasize the variance between the results,

we present boxplots for each sample size. Figure 3a shows the results for the column scalability of the *CENSUS*, *WIKIPEDIA* and *TPC-H* datasets in *ED* mode. We omit the *IMDB* dataset for the same reasons as in our row-scaling experiment.

Since the number of sIND candidates grows quadratically in the number of columns, one could expect a quadratic scaling. However, we observe a linear scaling. In our experimental setup of sampling from an input dataset, the number of valid sIND candidates on average grows linearly with the number of columns. Additionally, we find that SAWFISH's runtime in general does not grow over-proportionally with rising data amounts. This shows the effectiveness of SAWFISH's memory handling.

While the *CENSUS* dataset shows almost no variance due to its small size, the runtimes vary widely for the *WIKIPEDIA* and the *TPC-H* dataset. For the *WIKIPEDIA* dataset, we observe a small runtime variance for column sets of sizes 10 and 11, as only a few sets have such column counts. The remaining variance can be attributed to the differences between the three experiment runs. However, we observe some outliers. On the one hand, there are two outliers for column set size 10 that have a drastically reduced runtime. On the other hand, there are two outliers for column set sizes 3 to 5 that have a drastically increased runtime. For column set size 2, there are also a few outliers. These are artifacts related to the difference between processing a valid sIND and quickly discarding an invalid candidate, as we explained before. However, they are also related to the dataset "shape". The *WIKIPEDIA* dataset consists of two tables: Table 1 is wider, while Table 2 is longer. There exists the valid sIND *Table1[*`column9`*]* $\subsetapprox$ *Table2[*`column1`*]*. It takes roughly 40 seconds to validate this sIND alone, explaining the extreme outlier for column set size 2. This sIND is also responsible for the two outliers for column set size 10. There are similar explanations for the other outliers.

When using *JAC* as similarity measure, the results in Figure 3b for the *WIKIPEDIA* dataset look quite similar. While we can process the data faster overall, the scalability differs not much. We observe similar outliers because they are not specific to the *ED* mode.

For the *TPC-H* dataset, SAWFISH's column scaling behaves similarly. Figure 3 also shows some outliers in *ED* mode. However, these exist for the same reasons, as in the *WIKIPEDIA* dataset. Similarly to the *WIKIPEDIA* dataset, the general scaling seems to be linear as well. We observe another artifact of our random sampling: the mean runtime for the column set size 40 is in fact lower than the mean runtime for column set size 35. Due to the large number of possible column sets of these sizes, such a deviation can occur because of the differences in runtime of individual column combinations. Interestingly, there seems to be a column set of size 50 that takes longer to process than the entire dataset. The reason for this result is simply normal measuring deviation.

In comparison to the *ED* mode, the column scalability on the *TPC-H* dataset in Figure 3b in *JAC* mode presents a near perfect linear scaling of the mean runtime. We still observe outliers, but the overall variance is lower. Due to the lower overall runtime, the time spent on processing the input has a larger share of the entire runtime. Because the input handling is less dependent on specific data characteristics, it differs less than the validation time for equal sample sizes. Even on the largest *IMDB* dataset, the observed column scaling is linear. We also observe some outliers that are present for similar reasons as in *WIKIPEDIA* and *TPC-H* datasets. However, the overall variance compared to the absolute runtime is lower. This experiment shows that SAWFISH's column scaling behavior can be replicated on larger datasets.

5.2.3 Similarity Measure Thresholds. This section examines the impact of the only functional configuration parameter: the similarity threshold. For the *ED* mode, this is the number of allowed edits τ. For the *JAC* mode, this is the similarity threshold δ.

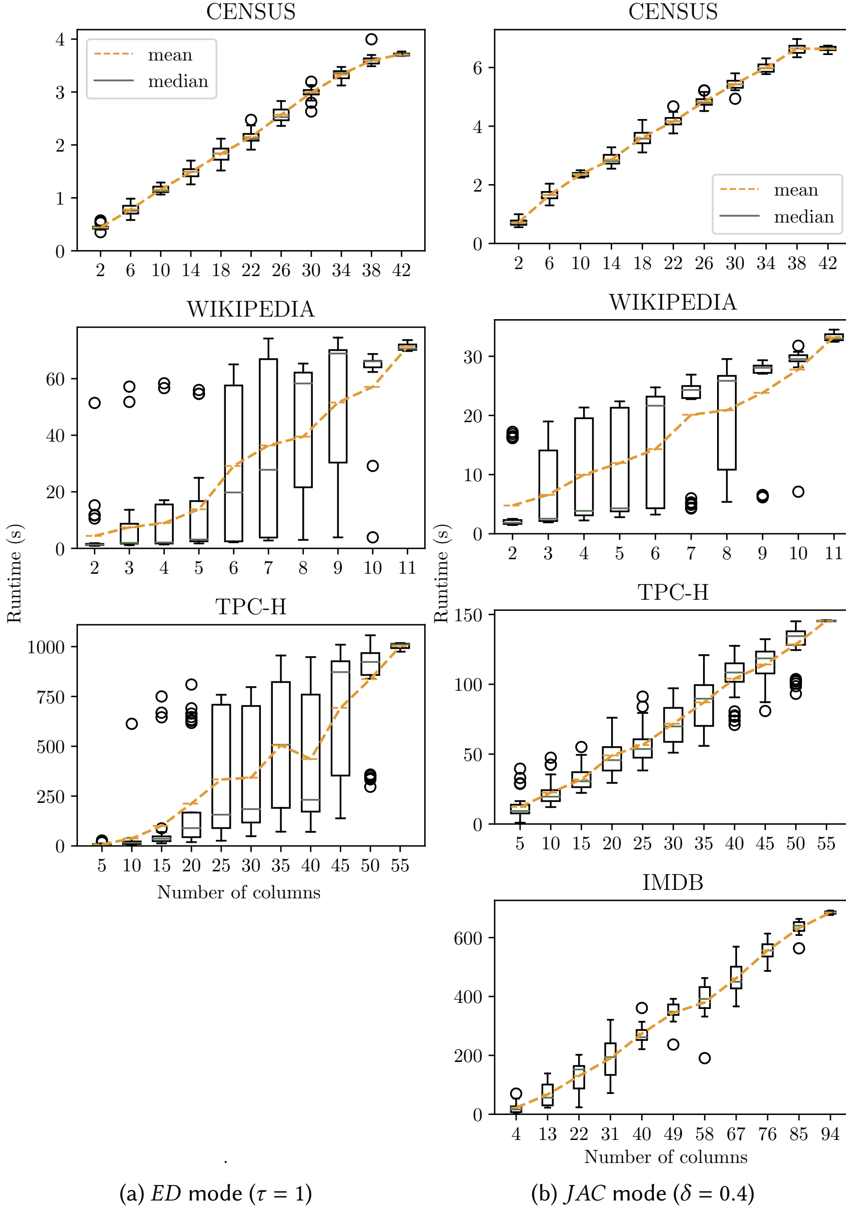
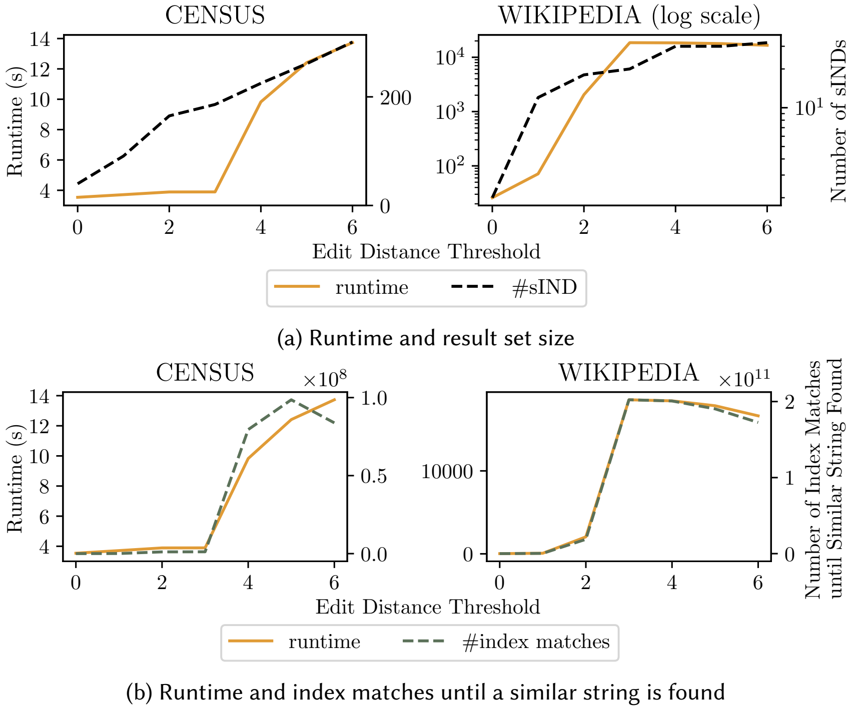


Fig. 3. Column scalability on different datasets

Edit Distance Threshold. We investigate the edit distance threshold on two datasets: *CENSUS* and *WIKIPEDIA*. We run Sawfish with an edit distance threshold τ between 0 and 6. We choose 6 as the upper bound, because beyond that threshold a majority of the strings consists of less than τ characters. We do not evaluate the other datasets, as they cannot be processed for $\tau = 6$ within our time limit of 24 hours. Figure 4a shows the results for the *CENSUS* dataset. We plot the runtime and the number of valid sINDs. While the runtime stays almost constant for lower edit distance



(a) Runtime and result set size

(b) Runtime and index matches until a similar string is found

Fig. 4. Edit distance threshold scaling on different datasets

thresholds, it quickly rises for $\tau \geq 4$. In contrast, the result set grows over the entire experiment, despite a growing number of simple sINDs (all value lengths are $\leq \tau$).

Interestingly, the jump in runtime does not relate to a jump in the result set size. To investigate the increasing runtime, we look at the number of matches in our index until a similar string was found. Figure 4b shows the runtime compared to the number of index matches. We can observe a clear correlation. There are two factors that influence the number of matches: the number of dependent values that we need to validate for the candidate sINDs, and the edit distance, which influences the pruning power of the inverted index. Since the segment length shrinks, there are more coincidental matches. Both of these factors are dependent on the internal structure of the dataset. For the *CENSUS* dataset, we observe that there is an increase in coincidental matches for edit distance $\tau \geq 4$ because the number of index matches grows more than the number of discovered sINDs. For $\tau = 6$, the number of index matches decreases again, because we can validate some strings earlier due to the greater edit distance. However, the effort for each edit distance computation increases, so overall the runtime still increases.

The results for the *WIKIPEDIA* dataset are on the right-hand side of Figure 4a. As before, we plot the runtime and the number of valid sINDs, here in a logarithmic scale to better show the differences in runtime. Sawfish's runtime increases significantly for lower edit distance thresholds and reaches it peak at $\tau = 3$. Afterward, it decreases slightly for larger edit distance thresholds, while the result set size stays almost constant.

To investigate both the stark increase and the following decrease in runtime, we regard the index matches once again. We use linear scaling in Figure 4b to better visualize the similarity of the lines. Interestingly, the two curves have a nearly identical shape. In contrast to the *CENSUS* dataset, the

early increase in the number of valid sINDs coincides with the increase in the number of index matches. Due to the higher number of valid candidates, we have to validate more dependent values. However, the number of index matches for $\tau = 3$ is significantly higher per dependent value than for all other edit distances. For each dependent value, we find an average of 17 000 index matches for $\tau = 3$, while there are only 8500 matches on average for $\tau = 4$. This means that there are many index matches for $\tau = 3$ that cannot be validated. Thus, we have to iterate the complete list of index matches for each dependent value. In contrast, we can validate more values for $\tau = 4$, so we do not have to process all index matches. Additionally, this explains the increase in valid sINDs for $\tau = 4$. This trend of earlier validation continues for $\tau = 5$ and $\tau = 6$. It also indicates that the number of coincidental matches does not increase significantly. As explained for the *CENSUS* dataset, the number of index matches decreases over-proportionally to the decrease in runtime, because each individual edit distance computation takes longer.

In conclusion, Sawfish is able to discover sINDs also for higher edit distance thresholds. However, the runtime is dependent on the internal structure of the dataset, namely the number of valid sINDs and the number of coincidental index matches.

Jaccard Similarity Threshold. We investigate the effect of different similarity thresholds δ on the *TPC-H* dataset. We focus on this dataset, because it contains strings with varying token counts; thus the difference for different thresholds is more noticeable.

Figure 5 shows the runtime and the result set size for different δ, ranging from 0.1 to 1.0. The reason for the drop in sINDs between $\delta = 0.5 \rightarrow 0.6$ is the content of the discovered sINDs. For $\delta \leq 0.5$, we find matches between single token and two token strings that form the extra sINDs.

The runtime behavior is related to the validation characteristics, especially for values with more tokens. For $\delta = 0.1$, for every string we need at most two matching tokens to validate the similarity, because we limit the number of tokens to 10. For $\delta = 0.2$ and $\delta = 0.3$, we cannot validate each string so easily, but the threshold is still low. Thus, many coincidental matches occur, which slow down the discovery process. For $\delta \geq 0.4$, the number of coincidental matches decreases, but the validation effort for each individual value grows, because we need to find more matching tokens. Nonetheless, the runtime converges. This behavior is different from the edit distance case, because we discover fewer sINDs overall, but also the number of coincidental matches is lower.

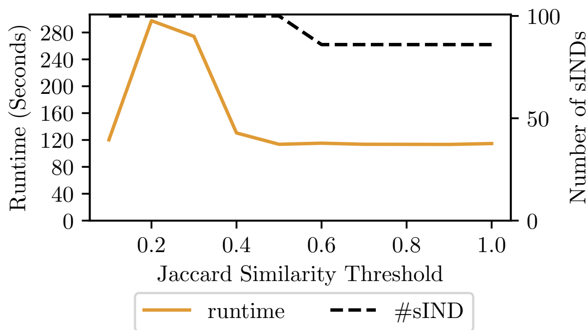


Fig. 5. Jaccard similarity scalability on the *TPC-H* dataset

5.3 Impact of Valid Dependencies

The column scalability experiment illustrated how much of the overall runtime depends on individual columns or even individual sIND candidates. Thus, we explore this effect in more detail. Valid

dependencies contribute to the runtime especially, because we need to validate each dependent value and cannot abort early.

Figure 6 shows the runtime for each potential sIND candidate in *CENSUS*, *WIKIPEDIA* and *TPC-H* datasets. For clarity, we only show the results of *ED* mode, but the findings are similar for the *JAC* mode. We highlight valid candidates and order the candidates by the number of distinct dependent values. As usual, we filtered any candidates that were discarded during preprocessing.

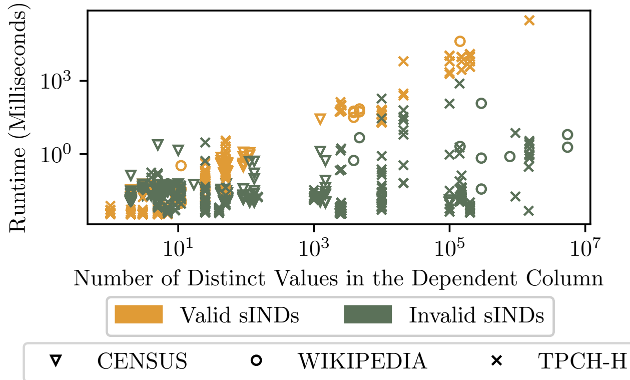


Fig. 6. Difference in runtime of valid and invalid candidate validation (log-scale for both axes)

We can observe two expected phenomena. Larger dependent columns and valid dependencies need more time to validate. The scaling behavior is more interesting. While the runtime for valid dependencies continuously increases for larger dependent columns, the scaling is not clear for invalid dependencies. This is because Sawfish needs to find a dependent value for which no similar referenced value exists. Therefore, the runtime for invalid dependencies is dependent on the position of the counterexample in the dependent column and thus almost random. This behavior can be observed especially for larger invalid dependencies, where the position differences between counterexamples are more noticeable.

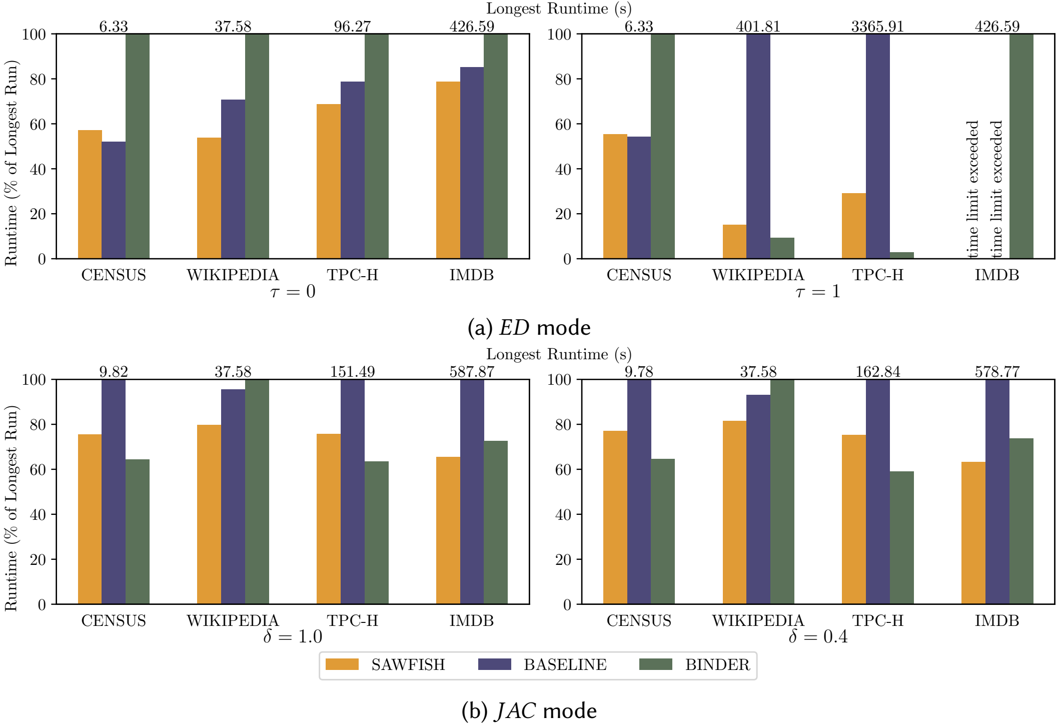
5.4 Comparison to Related Work

The experiments in this section compare Sawfish with the baselines using all datasets.

Edit Distance Mode. The following experiment compares the runtimes for an edit distance threshold $\tau = 0$. In this setting, all performance results are comparable since all algorithms discover traditional INDs. The left-hand side of Figure 7a shows the results for all datasets, scaled per dataset to the longest running algorithm. Interestingly, for the CENSUS dataset, our naive baseline is the fastest algorithm, while Binder needs the most time to complete the search. Since the dataset is relatively small, the benefit of Sawfish's preprocessing does not outweigh its overhead. Nonetheless, Sawfish is relatively close to the runtime of the baseline.

For the *WIKIPEDIA*, the *TPC-H* and the *IMDB* datasets, Sawfish's preprocessing combined with the improved index access pattern improves the performance over the baseline. Binder is the slowest in this experiment, because the two other algorithms can operate entirely in main memory if the entire dataset fits. Binder eagerly writes every bucket to disk after preprocessing, because it anticipates larger dataset sizes.

Next, we set $\tau = 1$, i.e., allow sINDs with an edit distance of up to 1 for each value. We present them on the right-hand side of Figure 7a. The results for Binder remain in the figures for comparison purposes, even though Binder cannot discover such dependencies. The result for the *CENSUS*



(a) *ED* mode

(b) *JAC* mode

Fig. 7. Comparison of Sawfish to related work

dataset is comparable in runtime to the IND discovery experiment. However, the results for the *WIKIPEDIA* dataset and the *TPC-H* dataset show the superiority of Sawfish. For the *TPC-H* dataset, we observe the higher complexity of the sIND discovery compared to the IND discovery. The runtime of Sawfish is about one third of the baseline's runtime. This visualizes the improvements that we accomplish by modifying PassJoin's approach. The effect is even larger in the *WIKIPEDIA* dataset: Sawfish outperforms the baseline by a factor of around 6.5. Sawfish was not able to discover all sINDs for the *IMDB* dataset, as it exceeded the time limit of 24 hours. This emphasizes the difficulty of sIND discovery, because Sawfish discovered all INDs (without allowing similarity) in around 6 minutes. We discuss ideas to improve the validation speed in Section 6.

Our results show that Sawfish can efficiently discover sINDs for reasonably large datasets. While the sIND discovery is a hard problem, Sawfish manages to process some datasets in a comparable time to the IND discovery. For all larger datasets, Sawfish outperforms the baseline.

Jaccard Similarity Mode. Figure 7b shows the results for $\delta = 1.0$ on the left-hand side. Binder performs best for the *CENSUS* and the *TPC-H* datasets, while Sawfish performs best on the other two datasets, *WIKIPEDIA* and *IMDB*. Besides Binder's algorithmic advantages, the overhead for tokenization inhibits the performance of the other two algorithms. While Sawfish and the baseline could have found more dependencies due to the order insensitivity of the Jaccard similarity, we did not observe such effects in our dataset. Nonetheless, they have to tokenize each value. On the other hand, Sawfish and the baseline can once again operate entirely in main memory. Therefore, they gain an advantage in datasets that do not create many tokens per value, e.g., *WIKIPEDIA* and *IMDB*.

For the next experiment, we set $\delta = 0.4$ and show the results on the right-hand side of Figure 7b. In general, the performance characteristics remain similar. The gap between SAWFISH and BINDER widens a bit for the *TPC-H* dataset, while it closes a bit for the *WIKIPEDIA* dataset. For the *IMDB* dataset, SAWFISH and the baseline improve their runtime, because they can validate string pairs of traditional INDs earlier due to the lower threshold.

When using the Jaccard similarity, SAWFISH can discover sINDs in a time that is comparable to the state-of-the-art IND discovery algorithm BINDER. We also show that SAWFISH outperforms a baseline for all datasets.

5.5 Joinability Case Study

In this experiment, we want to investigate whether the discovered similarity inclusion dependencies can indeed indicate joinability in the presence of typos or other data errors. Joinability means that two columns can be linked together because they contain similar data from the same domain[5]. To investigate the performance on a more heterogeneous data source, we ran SAWFISH on a subset of the *2015 Web Table Corpus (WTC)*[14]. This corpus consists of automatically crawled web tables. While these web tables are typically smaller than traditional database tables, more web tables are usually processed simultaneously. Our experiment uses an edit distance threshold $\tau = 1$ and the random sample of 10 000 tables from the original source[4]. However, we process only the 2516 relational tables in that set. We omit numeric columns to visualize the results better. SAWFISH runs in *ED* mode, because we find more and also more interesting results.

In total, we observe 1044 sINDs that consist only of strings and are not traditional INDs, i.e., the dependent column contains at least one value for which only a similar value can be found in the referenced column.

We manually annotated the sINDs to assess their genuineness. Overall, we found 161 (15%) meaningful sINDs, while 671 (64%) sINDs are coincidental. Coincidental sINDs are dependencies where the two columns contain data from different domains, but the values happen to be similar purely by chance. For example, we discover an sIND between the cost column of a university audiobook chapter list and the first name column of a table of probate files. Since it is a university audiobook, all chapters are “`Free`”. Thus, the entire value set of the column contains only this value. In turn, in the probate file table, we find the first name `Fred`. The remaining 212 (20%) sINDs are caused by errors in the header detection of the underlying dataset.

Additionally, we investigated the *characteristics* of meaningful and coincidental sINDs. For our example dataset, we observe that there are two criteria that reduce the number of coincidental sINDs significantly. First, the maximum number of characters of any value of the dependent side is above three. Second, the portion of dependent values that match only similarly to a value on the referenced side is below 30%. Given these two filters, the number of coincidental sINDs is reduced by 638 to 33 (20%). The number of meaningful sINDs that fulfill these criteria is 101 (64%).

To exemplify the discovered sINDs, we showcase some meaningful sINDs. For example, there is an sIND between the data type column of an API description and the data type column of the input parameters of a program. While, one column uses the uppercase versions, such as `Float` and `String`, the other column uses the lowercase versions, such as `float` and `string`. Despite these columns obviously containing data of the same domain, they do not form a traditional IND. Nonetheless, they could be joined to investigate usages for the same data type.

We present another example of a meaningful sIND in Table 4. There are multiple tables inside the sample that contain data about college athletes. Typically, students are classified into four categories: freshman, sophomore, junior, and senior – multiple columns contain abbreviations of

[4]http://data.dws.informatik.uni-mannheim.de/webtables/2015-07/sample.gz

these categories. However, as the table shows, they do not match perfectly, so they do not form a traditional IND. Interestingly, we only discover the sINDs $VCU \subsetneq CFB$ and $VCU \subsetneq NCAA$ (and their symmetric partners). To also discover $CFB \subsetneq NCAA$, we need to increase the edit distance threshold to 2. Nevertheless, all of this data belongs to the same domain. Moreover, if we want to compare students from the same category, it makes sense to join these values.

Table 4. Example sINDs in the *WTC* dataset

VCU Football	CFB: Tulsa Golden Hurricane	NCAA Baseball
Fr	FR	Fr.
So	SO	So.
Jr	JR	Jr.
Sr	SR	Sr.

6 CONCLUSION

This work introduces the formal concept of similarity inclusion dependencies (sINDs), extending traditional inclusion dependencies with a similarity measure. First, we identified use cases for sINDs, which include discovering foreign-key candidates and join partners. Second, we formalized an sIND definition that extends traditional INDs with an arbitrary similarity measure. Third, we presented SAWFISH, the first efficient approach to automatically discover sINDs from data. It finds all unary sINDs based on the edit-distance and the Jaccard similarity measure. SAWFISH combines approaches of traditional IND discovery and string similarity joins with a novel sliding-window approach and lazy candidate validation. Fourth, we evaluated SAWFISH, showing that it scales well in the number of rows, and in the number of columns. Compared to a baseline implementation, we outperformed it by a factor of up to 6.5. Finally, we examined the sINDs discovered by SAWFISH and observed real-world examples indicating joinability.

While sIND discovery is a harder problem than traditional IND discovery, the runtime could be further improved by multithreading or distributing the process. As SINDY [9] demonstrated, distribution can significantly improve the performance. However, we observed that a single column or even a single sIND candidate can dominate the runtime. Therefore, it is not trivial to scale SAWFISH's approach to multiple threads or nodes. Further future work shall extend SAWFISH to discover n-ary sINDs.

REFERENCES

[1] Jana Bauckmann, Ziawasch Abedjan, Ulf Leser, Heiko Müller, and Felix Naumann. 2012. Discovering conditional inclusion dependencies. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, 2094–2098.

[2] Jana Bauckmann, Ulf Leser, Felix Naumann, and Veronique Tietz. 2007. Efficiently detecting inclusion dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1448–1450.

[3] Loreto Bravo, Wenfei Fan, and Shuai Ma. 2007. Extending dependencies with conditions. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 243–254.

[4] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2016. Relaxed functional dependencies — a survey of approaches. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28, 1, 147–165. DOI: 10.1109/TKDE.2015.2472010.

[5] Pern Hui Chia, Damien Desfontaines, Irippuge Milinda Perera, Daniel Simmons-Marengo, Chao Li, Wei-Yen Day, Qiushi Wang, and Miguel Guevara. 2019. Khyperloglog: estimating reidentifiability and joinability of large data at scale. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 350–364. DOI: 10.1109/SP.2019.00046.

[6] 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)*. Volume 2287, 464–476.

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

[8] György Dósa. 2007. The tight bound of first fit decreasing bin-packing algorithm is $FFD(I) \leq 11/9 OPT(I) + 6/9$. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, 1–11. DOI: 10.1007/978-3-540-74450-4_1.

[9] 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 International Conference on Information and Knowledge Management (CIKM)*, 219–228. DOI: 10.1145/3357384.3357916.

[10] Wenfei Fan. 2008. Dependencies revisited for improving data quality. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 159–170. DOI: 10.1145/1376916.1376940.

[11] Jianhua Feng, Jiannan Wang, and Guoliang Li. 2012. Trie-join: a trie-based method for efficient string similarity joins. *VLDB Journal*, 21, 4, 437–461. DOI: 10.1007/s00778-011-0252-8.

[12] Jarek Gryz. 1998. Query folding with inclusion dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 126–133. DOI: 10.1109/ICDE.1998.655768.

[13] Sebastian Kruse, Thorsten Papenbrock, Christian Dullweber, Moritz Finke, Manuel Hegner, Martin Zabel, Christian Zöllner, and Felix Naumann. 2017. Fast approximate discovery of inclusion dependencies. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, 207–226. https://dl.gi.de/20.500.12116/629.

[14] 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 Conference Companion on World Wide Web*. International World Wide Web Conferences Steering Committee, 75–76. DOI: 10.1145/2872518.2889386.

[15] Mark Levene and Millist W. Vincent. 2000. Justification for inclusion dependency normal form. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 12, 2, 281–291. DOI: 10.1109/69.842267.

[16] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady* number 8. Volume 10, 707–710.

[17] Guoliang Li, Dong Deng, and Jianhua Feng. 2013. A partition-based method for string similarity joins with edit-distance constraints. *ACM Transactions on Database Systems (TODS)*, 38, 2. DOI: 10.1145/2487259.2487261.

[18] Richard Marsh. 2005. Drowning in dirty data? it's time to sink or swim: a four-stage methodology for total data quality management. *Journal of Database Marketing and Customer Strategy Management*, 12, 105–112. DOI: 10.1057/palgrave.dbm.3240247.

[19] Silvano Martello and Paolo Toth. 1990. Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Mathematics*, 28, 1, 59–70. DOI: 10.1016/0166-218X(90)90094-S.

[20] Renée J. Miller, Mauricio A. Hernández, Laura M. Haas, Lingling Yan, C. T. Howard Ho, Ronald Fagin, and Lucian Popa. 2001. The clio project: managing heterogeneity. *SIGMOD Rec.*, 30, 1, 78–83. DOI: 10.1145/373626.373713.

[21] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. 2015. Data profiling with Metanome. In *PVLDB* number 12. Volume 8, 1860–1863. DOI: 10.14778/2824032.2824086.

[22] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. 2015. Divide & conquer-based inclusion dependency discovery. In *PVLDB* number 7. Volume 8, 774–785. DOI: 10.14778/2752939.2752946.

[23] Philipp Schirmer, Thorsten Papenbrock, Ioannis Koumarelas, and Felix Naumann. 2020. Efficient discovery of matching dependencies. *ACM Transactions on Database Systems (TODS)*, 45, 3. DOI: 10.1145/3392778.

[24] Fabian Tschirschnitz, Thorsten Papenbrock, and Felix Naumann. 2017. Detecting inclusion dependencies on very many tables. *ACM Transactions on Database Systems (TODS)*, 42, 3. DOI: 10.1145/3105959.

[25] Robert A Wagner and Michael J Fischer. 1974. The string-to-string correction problem. *J. ACM*, 21, 1, 168–173. DOI: 10.1145/321796.321811.

[26] Wei Wu, Bin Li, Ling Chen, Junbin Gao, and Chengqi Zhang. 2022. A review for weighted minhash algorithms. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 34, 6, 2553–2573. DOI: 10.1109/TKDE.2020.3021067.

[27] Minghe Yu, Guoliang Li, Dong Deng, and Jianhua Feng. 2016. String similarity search and join: a survey. *Frontiers of Computer Science*, 10, 3, 399–417. DOI: 10.1007/s11704-015-5900-5.

Received July 2022; revised October 2022; accepted November 2022