B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 207

Fast Approximate Discovery of Inclusion Dependencies

Sebastian Kruse[1] Thorsten Papenbrock[1] Christian Dullweber[2] Moritz Finke[2]
Manuel Hegner[2] Martin Zabel[2] Christian Zöllner[2] Felix Naumann[1]

Abstract: Inclusion dependencies (INDs) are relevant to several data management tasks, such as foreign key detection and data integration, and their discovery is a core concern of data profiling. However, *n*-ary IND discovery is computationally expensive, so that existing algorithms often perform poorly on complex datasets. To this end, we present Faida, the first approximate IND discovery algorithm. Faida combines probabilistic and exact data structures to approximate the INDs in relational datasets. In fact, Faida guarantees to find all INDs and only with a low probability false positives might occur due to the approximation. This little inaccuracy comes in favor of significantly increased performance, though. In our evaluation, we show that Faida scales to very large datasets and outperforms the state-of-the-art algorithm by a factor of up to six in terms of runtime without reporting any false positives. This shows that Faida strikes a good balance between efficiency and correctness.

Keywords: inclusion dependencies, data profiling, dependency, discovery, metadata, approximation

1 The Intricacies of Inclusion Dependency Discovery

It is a well-known fact that ever-increasing amounts of data are being collected. To put such large and complex datasets to use, be it for machine learning, data integration, or any other application, it is crucial to know the datasets' structure. Unfortunately, this information is oftentimes missing, incomplete, or outdated for all sorts of reasons. To overcome this quandary, the research area of *data profiling* has borne several algorithms to discover structural metadata of any given dataset.

A very important and fundamental type of structural metadata of relational databases are *inclusion dependencies (INDs)* [AGN15]. They form an integral component of foreign key (FK) discovery [Ro09], allow for query optimizations [Gr98], enable integrity checking [CTF88], and serve many further data management tasks. Intuitively, an IND describes that a combination of columns from one database table only contains values of another column combination, which might or might not be in the same table. Before looking at a concrete example, let us formalize this notion.

Definition 1 (Inclusion dependency) *Let r and s be two relational, potentially equal, tables with schemata $R = (R_1, \ldots, R_k)$ and $S = (S_1, \ldots, S_m)$, respectively. Further, let $\bar{R} = R_{i_1} \ldots R_{i_n}$ and $\bar{S} = S_{j_1} \ldots S_{j_n}$ be n-ary column combinations of distinct columns. We say that $\bar{R}$ is included in $\bar{S}$, i.e., $\bar{R} \subseteq \bar{S}$, if for every tuple $t_r \in r$, there is a tuple $t_s \in s$, such*

[1] Hasso Plattner Institute (HPI), Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, firstname.lastname@hpi.de
[2] Hasso Plattner Institute (HPI), Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, firstname.lastname@student.hpi.de

that $t_r[\bar{R}] = t_s[\bar{S}]$. $\bar{R}$ is called the dependent column combination and $\bar{S}$ the referenced column combination. With both of them having n columns, the IND is said to be n-ary.

Tab. 1 illustrates a lexicographical example dataset comprising a dictionary table that stores words of different languages, and a translation table that translates those words from one language to the other. Apparently, there are, amongst some others, two interesting, ternary INDs to be found in that example, namely *word1*, *lang1*, *type1* $\subseteq$ *word*, *lang*, *type* and *word2*, *lang2*, *type2* $\subseteq$ *word*, *lang*, *type*. Intuitively, these INDs require that all words in the translation table are found in the dictionary table. Note in particular the stronger semantics in comparison to INDs of lower arity, e.g., *word1* $\subseteq$ *word* and *word2* $\subseteq$ *word*. While the former, ternary INDs identify words not only by their literal but also their language and syntactical type, the latter, unary INDs merely consider the word literal, which does not suffice to uniquely identify a word (cf. *hat*). Due to these stronger semantics, it is worthwhile to discover INDs of the highest possible arity.

word	**lang**	**type**
hut	en	noun
hat	en	noun
has	en	verb
Hütte	de	noun
Hut	de	noun
hat	de	verb

(a) Dictionary table.

word1	**lang2**	**type1**	**word2**	**lang2**	**type2**	**fit**
hut	en	noun	Hütte	de	noun	$\perp$
hat	en	noun	Hut	de	noun	$\perp$
has	en	verb	hat	de	verb	$\perp$

(b) Translation table.

Tab. 1: A lexicographical example dataset with several INDs.

In the last years, several IND discovery algorithms have been proposed [Pa15, DMLP09, KR03, DMP03], pushing the boundaries in terms of efficiency and scalability. However, many real-world datasets cannot be processed by any of these algorithms within reasonable time, even on powerful hardware, for two main reasons: First, the number of valid *n*-ary INDs is often enormously large in real-world datasets. The result sets alone can, therefore, already exceed main memory limits [Pa15]. Second, and more commonly, the existing algorithms need to shuffle huge amounts of data to test IND candidates – in fact, the amount of shuffled data depends on the number of IND candidates and easily exceeds the inspected dataset in size. Some algorithms perform those shuffles out-of-core to overcome main memory limitations. Still, not only does this operation remain an efficiency bottleneck, but also the shuffled data can become so large that even disk storage limitations are exceeded!

We propose to tackle the latter issue by *approximating* the INDs of datasets, that is, for any given dataset we calculate a set of INDs that is complete but might contain false positives. However, the guarantee of correctness is traded for great performance improvements. Let us justify, why this trade is worthwhile: We observed that in real-world datasets any two column combinations are either related by an IND or their values are disjoint to a great extent. In other words, it is rare that the vast majority of values of one column combination are included in the other column combination, except for a small remainder. This clear cut allows to use more light-weight, approximate methods to test IND candidates without

risking severe accuracy losses. Nonetheless, in the few cases where two columns overlap in, say, 99 % of their values and an approximate method would indeed incorrectly report an IND, then this false positive is still a *partial IND*, i.e., it has only few violating values. We note that guarantueed and complete correctness of INDs is not required by many use cases, such as FK discovery [Ro09, Zh10] and data cleaning [Bo05].

To this end, we introduce Faida, the first approximate discovery algorithm for unary and n-ary INDs. Faida uses two different approximate data structures: a hash-based probabilistic data structure to characterize column combinations with a very small memory footprint, and a sampling-based inverted index to attenuate statistically expected inaccuracies. The combination of these two data structures offers high-precision results, because both compensate the other's weaknesses. In fact, we found Faida to report exact results in all our experiments. In addition, we characterize the novel class of *scrap INDs*, a sort of degenerate INDs that are not applicable to typical IND use cases but usually make up a considerable share of the INDs in a dataset. Faida identifies and prunes scrap INDs to narrow down the search space and achieve further performance improvements. This is particularly useful to prevail in situations where there are actually intractably many INDs, as mentioned above.

The remainder of the paper is organized as follows: In Sect. 2, we describe related work. We proceed to give an overview of Faida in Sect. 3, followed by a detailed description of its hybrid IND checking process in Sect. 4 and a formalization and rationale for scrap INDs in Sect. 5. Then, in Sect. 6, we compare Faida to the exact state-of-the-art IND discovery algorithm Binder and evaluate Faida's effectiveness and efficiency in detail. Eventually, we conclude in Sect. 7.

2 Related Work

The discovery of dependencies, such as functional dependencies, order dependencies, or inclusion dependencies, in a given database is considered an essential component of data profiling [AGN15]. In this section, we focus on related work that addresses the approximate and exact discovery of inclusion dependencies.

Approximate IND discovery. We define approximation as estimation of the set of the actual INDs in a dataset. This nomenclature is in contrast to "approximate INDs" (also: "partial INDs") that hold only on a subset of the rows [LPT02, DMLP09]. This orthogonal concern is not the focus of this paper.

The only existing approach to approximate IND discovery is described by Zhang et al. as part of foreign-key (FK) discovery [Zh10]. It uses bottom-k sketches and the Jaccard index to approximate the inclusion of two columns based on Jaccard coefficients. Their approach has several disadvantages: For each level of n-ary INDs the hashes for the bottom-k sketches have to be computed from all actual values. Furthermore, it suffers from a similar problem as the probabilistic data structure used by Faida when comparing a column c_1 that has only few distinct values with a column c_2 that has many distinct values. The two bottom sets have potentially only a small overlap and in the worst case, all bottom hashes of c_2 are smaller than the bottom hashes of c_1. While Faida's errors are limited to false positives, this effect

can additionally lead to false negatives. Moreover, the authors focus on FK candidates and apply the approach only to relatively few candidates where the right-hand side has to be a known primary key. For these reasons, the proposed algorithm produces significantly different result sets than Faida, so that a performance comparison between these algorithms does not make sense.

Exact IND discovery. In previous research, much attention has been paid to the discovery of unary INDs, i.e., INDs between single columns. Different discovery strategies have been proposed, based on inverted indices [DMLP09], sort-merge joins [Ba07], and distributed data aggregation [KPN15]. While efficient unary IND discovery is an important part of n-ary IND detection, the problem is not of exponential complexity and therefore a much simpler task. Of course, Faida can also be applied to the efficient discovery of unary INDs.

Research has also devised exact algorithms for the discovery of n-ary INDs, in particular Mind [DMLP09] and Binder [Pa15]. Both employ an Apriori-like discovery scheme to find IND candidates. While Mind tests these candidates individually against a database, Binder employs a more efficient divide-and-conquer strategy to test complete candidate sets in a single pass over the data. This makes Binder the current state-of the-art algorithm, which we compare against in our evaluation. Nevertheless, both strategies exhibit declined performance and increased memory consumption when testing IND candidates of high arity. In contrast, Faida, which also builds upon Apriori candidate generation, employs probabilistic data structures that do not suffer from this effect.

Besides Apriori-based approaches, depth-first algorithms have been proposed that optimize candidate generation towards inclusion dependencies of very high arity [KR03, DMP03]. However, these algorithms employ the same expensive IND checking mechanisms as Mind and are only applicable to pairs of tables, lacking a strategy to deal efficiently with whole datasets. Another recent approach avoids candidate generation entirely [SM16]. This is achieved by determining for every pair of tuples from two given tables, which unary INDs they support. These sets of unary INDs are then successively merged into maximal n-ary INDs. Again, no strategy is given to efficiently profile datasets with more than two tables.

Foreign key discovery. Although strongly connected, IND discovery and FK discovery are distinct problems: not every IND that holds on a given dataset is a FK relationship. Vice versa, in unclean databases, there might be semantically intended FK relationships whose corresponding INDs are violated by several tuples [Zh10]. However, INDs are a prerequisite for several FK discovery algorithms [Ro09, Zh10].

3 Overview of FAIDA

In this section, we present Faida, our Fast Approximate IND Discovery Algorithm, from a bird's eye view before giving more details in the following sections. Fig. 1 depicts Faida's general mode of operation. As for most n-ary IND discovery algorithms, Faida starts by identifying unary INDs, then uses an Apriori-style candidate generation process to generate binary IND candidates, and checks those in turn. This generate-and-test procedure

is repeated – i.e., the latest discovered, n-ary INDs are used to generate $(n+1)$-ary IND candidates, which are tested subsequently to retain the actual $(n+1)$-ary INDs – until the candidate generator produces no more IND candidates for some arity $n_{\max}$. In the following, we describe the various processing steps in more detail.

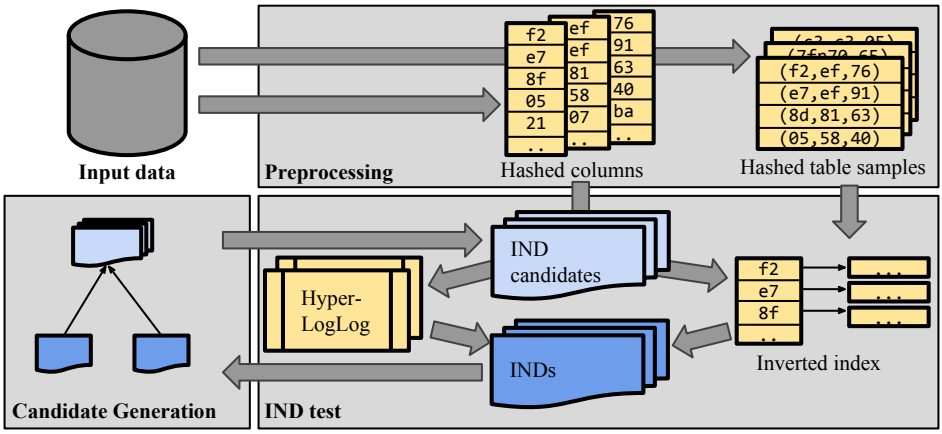

Fig. 1: Overview of Faida.

Preprocessing. The performance of IND discovery algorithms is mainly impacted by the fact that the input dataset has to be re-read and shuffled in every IND test phase. Faida attenuates this issue in a preprocessing step. At first, it converts the input dataset into *hashed columns*, i.e., the values in the input dataset tables are hashed and stored in a columnar layout. During the IND test phases, Faida will then resort to those hashed columns rather than the original input dataset, thereby greatly reducing the amount of data to be read, as we explain in the next paragraph. Furthermore, *hashed samples* of every table are stored – they are needed for bootstrapping the inverted index in the IND test phase. In Sect. 4.1, we explain the preprocessing step and its impact on performance and the IND result quality in greater detail. Still, we already want to remark that the use of compact hashes instead of actual values greatly improves performance of Faida in the subsequent phases, but it cannot guarantee exact results due to potential hash collisions. If two values share the same hash value, Faida will deem those two values equal. Nevertheless, this phenomenon can only cause false positive INDs but not miss out any INDs. Moreover, it is extremely unlikely that single hash collisions yield false positive INDs, because the distinction between INDs and non-INDs is usually not governed by single values only.

IND test. To test IND candidates, Faida uses a hybrid approach that builds upon a probabilistic *HyperLogLog* structure [Fl07] to represent columns with many distinct values and a sampling-based inverted index to represent columns with few distinct values. For each level, i.e., for each IND arity, Faida passes once over the relevant hashed columns, inserts them into the two data structures, and finally jointly evaluates them to determine the actual INDs. The IND tests might also produce false positives in addition to those caused by the hashing during preprocessing, but it does not produce false negatives. Because the IND

tests and the hashing have consistent error characteristics, Faida is guaranteed to find all INDs in a dataset. Sect. 4.2 to 4.3 discuss our hybrid IND test strategy in detail.

Candidate generation. Faida uses the same Apriori-style candidate generation as existing algorithms [DMLP09, Pa15]. This procedure makes use of the downward closure property of INDs: It only generates an n-ary IND candidate $A_1A_2 \dots A_n \subseteq B_1B_2 \dots B_n$, if the n $(n-1)$-ary INDs $A_2A_3 \dots A_n \subseteq B_2B_3 \dots B_n$, $A_1A_3 \dots A_n \subseteq B_1B_3 \dots B_n$, ..., and $A_1A_2 \dots A_{n-1} \subseteq B_1B_2 \dots B_{n-1}$ are verified to hold on the profiled dataset. There are multiple reasons why we did not replace the candidate generation with an approximate version. At first, we found in our experiments that the candidate generation only takes a tiny fraction of the overall runtime of IND algorithms - so the overall gains of any performance improvement here would be marginal. Secondly, if an approximate candidate generation produces false positives, we would likely end up with inferior performance, because the algorithm would need to test those additional IND candidates as well. Finally, if an approximate candidate generation yields false negatives, i.e., if it misses out on some IND candidates, Faida cannot guarantee completeness of its results anymore, which would be a bad trade. However, Faida might still prune some candidates deliberately: In Sect. 5, we describe the class of *scrap INDs* that oftentimes make up a great share of all INDs in a dataset but that are mostly useless. We show how to detect scrap INDs, so as to remove them from the set of IND candidates for performance improvements.

4 Fast and Lean Inclusion Dependency Approximation

As stated in Sect. 3, Faida adapts the same workflow for IND discovery as most exact algorithms: First, it discovers all unary INDs and, then, iteratively generates and tests IND candidates of respectively next arity. However, Faida uses approximation techniques in this process to reduce the amount of data handled in each iteration and, ultimately, to improve performance. In the following, we explain the building blocks as well as the interplay of this approximation scheme in more detail.

4.1 Read-Optimized Input Data

Whenever there is a set of IND candidates to be tested, exact IND discovery algorithms (i) read the input dataset, (ii) extract the value combinations that belong to dependent or referenced column combination of any IND candidate, and then (iii) shuffle those value combinations to compare the column combinations of the IND candidates to determine the actual INDs. By dropping the guarantee of the correctness of the discovered INDs, Faida can use a completely different, more efficient, and more scalable approach to test IND candidates. Some activities of Faida's IND test can be factored out of the IND test loop and instead be done only once, before the first IND test, which further improves performance. We describe those in the following.

Hashing. Faida's IND test uses hashes of the values in the input dataset, rather than the actual values. This is obviously favorable w.r.t. performance and scalability, because hashes

are of a small, fixed size in contrast to the actual values. Thus, they consume less memory and can be efficiently compared. Moreover, Faida's IND test uses HyperLogLog [Fl07] data structures, which operate on hashes anyway. However, depending on the hash function, the hashing can be quite CPU-intensive. Of course, it is necessary to read the input data once before it can be hashed. Because re-reading the input dataset over and over again is costly in terms of disk I/O, Faida reads the input dataset only once, hashes its values, and writes the resulting hashes back to disk. Note that for testing n-ary INDs with $n \geq 2$, other IND discovery algorithms need to shuffle *combinations* of values. In contrast, Faida merges the individual value hashes of those value combinations into a new, single hash value using a simple bitwise `XOR`. Consequently, the descriptions of Faida's other components refer, without loss of generality, only to single hash values and not to combinations of hash values.

Columnar data layout. In most cases, relational data is organized in a row layout. When testing n-ary IND candidates with $n \geq 2$, most columns of the input dataset usually do not appear in all IND candidates. Still, in a row layout, those columns have to be read without being of any use. Faida avoids this inefficiency by storing above described hashes in a columnar layout, thereby allowing the IND test to read only those columns that are part of an IND candidate. For the example dataset from Tab. 1, Faida creates nine files, each containing the hashes of the values of one of the columns in that dataset.

Table samples. As mentioned in Sect. 3, Faida uses a hybrid IND test strategy with HyperLogLog structures and an inverted index. The inverted index operates on a small sample of the (hashed) input data. Our algorithm calculates this sample once in the beginning and, then, reuses it in every IND test phase. In fact, we have the following requirements for the sample: Given a sample size s (e.g., $s = 500$), we need a sample of each table, such that this sample table contains $min\{s, d_A\}$ distinct values for each column A with d_A being the actual number of distinct values of A. The simple rationale for this requirement is that for columns with only few distinct values, we aim to ensure that these are effectively processed in the inverted index. If we took a random sample instead, we would most likely capture only a subset of its actual values leading to an impaired performance of the inverted index. To generate this sample, we use a simple greedy proceeding that is depicted in Algorithm 1.

The sampling algorithm is applied to each table individually and can be piggy-backed onto the above described preprocessing steps. Note that it operates on the hashed values and, thus, benefits from the low memory footprint of its data structures. The algorithm starts by initializing two data structures, namely T_s, which collects the sample tuples, and *sampledValues*, which tracks for each of the columns in the table the values that have been sampled from it so far (Lines 1–2). Then, it iterates all the tuples of the table (Line 3) to decide for each tuple if it should be included in the sample. A tuple should be included if a column exists that does not yet have s different sampled values and the tuple provides a yet unseen value for that column (Lines 4–7). If so, the tuple is added to T_s and the samples in *sampledValues* are updated accordingly. As an example, consider Tab. 1a and assume $s = 2$. In that case, Algorithm 1 would sample the first four tuples: The first tuple is always sampled anyway; the second tuple provides a new value for *word*; the third for *type*; and the fourth for *lang*. Afterwards, the algorithm has sampled at least two values for each column, so no further tuple will be picked.

Algorithm 1: Create a hashed table sample

Input : hashed tuples T for a table with attributes $A_1 \dots A_n$
minimum values s to be sampled per column
Output sample T_s of the hashed tuples
:

```
1  T_s ← ∅; sampledValues ← arrayOfSize(n);
2  foreach 1 ≤ i ≤ n do sampledValues[i] ← ∅ ;
3  foreach t ∈ T do
4      addToSample ← false;
5      foreach 1 ≤ i ≤ n do
6          if |sampledValues[i]| ≤ s ∧ t[A_i] ∉ sampledValues[i] then
7              addToSample ← true;
8      if addToSample then
9          T_s ← T_s ∪ {t};
10         foreach 1 ≤ i ≤ n do sampledValues[i] ← sampledValues ∪ {t[A_i]} ;
```

4.2 Scalable Probabilistic Inclusion Dependency Test

As stated in the previous sections, the major bottleneck of exact IND discovery algorithms is the IND candidate testing, which requires shuffling large amounts of data, especially when many IND candidates of higher arities arise. Not only are there more value combinations of larger individual size with increasing arity, but the shuffling itself also becomes more expensive. That is because the shuffling can eliminate duplicate values. However, the likelihood of duplicate value combinations drastically declines with the arity. Using hashes instead of values, as described in Sect. 4.1, can only mitigate but not completely avoid this problem and it might also eventually succumb to memory limitations.

Faida avoids the shuffling completely and uses a probabilistic approach for IND testing. The main idea is to calculate a summary for each column combination in the IND candidates and then perform a heuristic IND test on those summaries. An obvious instance of this idea is to encode column combinations with Bloom filters and then check for each IND candidate if its referenced column combination Bloom filter has all bits of the dependent column combination Bloom filter set. However, Bloom filters are prone to oversaturation for very large columns. Therefore, we use set cardinalities: Let $X \subseteq Y$ be an IND candidate and s a function that maps a column combination to the set of all its contained value combinations. Then $X \subseteq Y$ is an IND, if and only if $|s(Y)| = |s(X) \cup s(Y)|$. The set cardinality of a multiset can be efficiently and effectively estimated with *HyperLogLog* [Fl07], which scales to very large cardinalities with arbitrary precision.

Before we describe how Faida employs HyperLogLog for IND tests, let us briefly explain how that counting scheme works. At the core, it makes use of the following observation: Let s be a sample from a uniform distribution of the values from 0 to $2^k - 1$ for some k and let n be the number of leading zeroes in the binary representation of $\min s$ using k digits. Then $|s|$ can be estimated as 2^n. Assuming a good hash function that produces uniformly

distributed, mostly collision-free hash values for its input data, we can count any values via their hashes. HyperLogLog extends this idea by partitioning the hashes by their prefixes into buckets and maintain for each bucket the largest number of leading zeroes observed in the residual suffix bits. The observations in the different buckets are, then, merged via harmonic mean with additional bias correction [Fl07]. Consider Fig. 2 as an example, where we use HyperLogLog to estimate the set cardinality of the *word* column from Tab. 1. We employ a 4-bit hash function using the first bit for partitioning and the residual three bits to count leading zeroes. Apparently, for the bucket for the prefix `0`, `hash`(*has*) = `00`11 provides the most leading zeroes in the suffix (namely one), while for the bucket with the prefix 1, `hash`(*hut*) = `1000` provides the most leading zeroes, namely three. Applying the harmonic mean and bias correction, HyperLogLog estimates four as the set cardinality of the input values. Note that HyperLogLog is, due to its stochastic nature, rather suited to estimate the set cardinality of larger datasets.

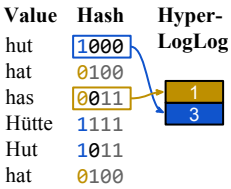

Fig. 2: Example HyperLogLog structure with two buckets.

Given this intuition of HyperLogLog, we proceed to show how we use this data structure for IND testing. As mentioned above, the basic idea is that for an IND candidate $X \subseteq Y$ to hold, the set cardinality of Y must be equal to the joint set cardinality of X and Y. A naïve implementation of this idea would use two HyperLogLog structures to estimate and compare both cardinalities, say HLL_Y and $\text{HLL}_{X \cup Y}$. This approach requires only very little processing effort in contrast to the shuffling of exact IND discovery algorithms: It is only necessary to scan once through the data and update the HyperLogLog structures, which themselves are of constant size. However, this is expected to work well only when (i) Y contains a stochastically relevant amount of elements and (ii) the differences of the count estimates of HLL_Y and $\text{HLL}_{X \cup Y}$ is greater than the expected estimation error of HLL_Y. The estimation error can be controlled via the number of buckets in the HyperLogLog structures. With our observation that the distinction of INDs and non-INDs is not governed by only few values, we can assume the second criterion to hold if we keep the expected estimation error small, e.g., around 0.1 %.

With that theoretical understanding on the applicability of HyperLogLog, we can now tailor it a bit more towards IND tests: The only case, where $\text{HLL}_{X \cup Y}$ would yield an estimate greater than that of HLL_Y, applies when there is some element in X that is not Y and that provides more leading zeroes to any partition in $\text{HLL}_{X \cup Y}$ than any element from Y does. Thus, we can instead maintain the two HyperLogLog structures HLL_X and HLL_Y and check if HLL_X has observed more leading zeroes than HLL_Y in any of the buckets. While this test is logically equivalent to the above naïve approach, it requires Faida to maintain only one HyperLogLog structure per column combination rather than up to two

HyperLogLog structures per IND, which is theoretically bounded only by the square of column combinations.

Tab. 2 exemplifies HyperLogLog structures with two buckets applied to the example dataset from Tab. 1. Note that in practical scenarios, more buckets should be used to enhance HyperLogLog's accuracy. Given that all column pairs are IND candidates, the above described IND test identifies all actual INDs correctly. For instance, *word1* $\subseteq$ *word* is correctly deemed to be an IND, because the HyperLogLog bucket values of *word2*, i.e., 1 and 3, are less then or equal to the respective bucket values of *word*, which are also 1 and 3. This result completeness is guarantueed, because if $X \subseteq Y$ is actually an IND, then all values of X must be considered in the HyperLogLog structure of Y. Correctness of the result cannot be guaranteed, though. For instance, *word* $\subseteq$ *word1* is deemed to be an IND judging from the HyperLogLog structures, although it is actually not. As mentioned above, HyperLogLog is not well-suited to compare columns with only few distinct values. Therefore, we complement it with a second IND test as presented in the next section.

Prefix	**word**	**lang**	**type**	**word1**	**lang1**	**type1**	**word2**	**lang2**	**type2**	**fit**
0	1	1	0	1	1	0	0	0	0	0
1	3	1	2	3	0	1	1	1	1	0

Tab. 2: HyperLogLog structures for single columns of the example dataset.

4.3 Hybrid Inclusion Dependency Test

HyperLogLog is a stochastic counting approximation that works particularly well for IND tests where both column combinations have many distinct values. To fill its blind spot – IND candidates containing a column combination with only few values – the IND tests additionally use an inverted index. The IND testing with inverted indexes has first been introduced by De Marchi et al. [DMLP09]. The basic idea is to build an inverted index of the input dataset that maps each value to the columns it appears in. For instance, for our example dataset in Tab. 1 such an inverted index maps the value *en* to the set of columns {*lang*, *lang1*, *lang2*}. Now to find all columns that include a certain column X, it suffices to select all column sets that contain X and intersect them. Applying this procedure for every column X yields all the INDs in the dataset.

To avoid scaling problems, Faida cannot create such an inverted index on the entire dataset. However, it is possible to create such an inverted index for a subset of the values (or rather a subset of the hashes as described in Sect. 4.1) in the dataset and apply the said IND test to it. This idea seems promising, because we can control the sample size and yet focus the sample in such a manner that it comprises especially the values of columns with only few distinct values (cf. Sect. 4.1), i.e., those cases where HyperLogLog is not so well-suited. Moreover, this approach still preserves result completeness: If $X \subseteq Y$ is an IND, then $\sigma(X) \subseteq \sigma(Y)$ has to hold, too, where σ selects only those values that are in the sample.

Algorithm 2 implements this idea. It starts by taking the sample tuples for each table (cf. Sect. 4.1). Then, it creates an inverted index for all column combinations that appear

in any of the IND candidates (Lines 1–7). Next, Algorithm 2 builds a HyperLogLog structure for each column combinations. Afterwards, the algorithm needs to initialize a flag in *isCovered* to keep track of whether all values for a certain column combination are actually found in the sample and, thus, in the inverted index (Lines 8–12). Having initialized all relevant data structures, the algorithm iterates all values of all column combinations (Line 13). If a value is included in the table samples and, thus, a key of the inverted index, the corresponding index entry is updated with the column combination of that value (Lines 14–16); otherwise, the algorithm updates the corresponding HyperLogLog structure with that value (Line 17). In the latter case, the algorithm also notes that the respective column combination is not completely covered by the inverted index (Line 19). Whether or not a column combination is covered becomes relevant in the subsequent phase where the IND candidates are actually tested. In the beginning of that phase, only those IND candidates are retained that hold on the inverted index (Line 20). Now, if the dependent

Algorithm 2: Hybrid IND test

Input : set of IND candidates I_c
samples of hashed tuples for each table $\mathcal{T}_s$
hashes for all value combinations V
Output verified INDs I
:

```
1  invertedIndex ← mapping(defaultValue = ∅);
2  foreach T_s ∈ 𝒯_s do
3      C ← relevantColumnCombinations(T_s, I);
4      foreach t ∈ T_s do
5          foreach c ∈ C do
6              v ← t[c];
7              invertedIndex[v] ← invertedIndex[v] ∪ {c}
8  isCovered ← mapping();
9  hlls ← mapping();
10 foreach c ∈ allColumnCombinations(I_c) do
11     isCovered[c] ← true;
12     hlls[c] ← hyperLogLog();
13 foreach v ∈ V do
14     c ← columnCombination(v);
15     C ← invertedIndex[v];
16     if C ≠ ∅ then invertedIndex[v] ← C ∪ {c};
17     else
18         insert(v into hlls[c]);
19         isCovered[v] ← false;
20 I'_c ← testAll(I_c on invertedIndex);
21 foreach ⟨X ⊆ Y⟩ ∈ I'_c do
22     if isCovered[X] ∨ (¬isCovered[Y] ∧ test(⟨X ⊆ Y⟩ on hll[X] and hll[Y])) then
23         I ← I ∪ {⟨X ⊆ Y⟩};
```

column combination X of any retained IND candidate $X \subseteq Y$ is covered by the inverted index, we can directly promote it as a valid IND (Lines 21–23); otherwise, if neither X nor Y is covered, we additionally perform the HyperLogLog-based IND test to verify the candidate. Note that, if Y is covered but X is not, then there must be some value in X that violates $X \subseteq Y$. Thus, we do not add it to the set of actual INDs I in that case.

In summary, the presented IND test follows a hybrid strategy using HyperLogLog and an inverted index. The sampling-based inverted index reliably discerns INDs between column combinations with only few distinct values. If, in contrast, IND candidates between column combinations with many distinct values need to be tested, it automatically switches to the stochastic HyperLogLog-based test that scales well because of its constant memory footprint regardless of the size of the input data. Still, the inverted index reinforces the test as a "control sample".

5 Scrap Inclusion Dependencies

Exact IND discovery algorithms, and also Faida, deliberately exclude some uninteresting INDs in their result sets, even though the respective dataset actually satisfies them. Those INDs have certain *syntactical* properties: First, there are *trivial* INDs $X \subseteq X$ with equal dependent and referenced column combinations, which always hold. Second, discovery algorithms respect permutability of INDs, i.e., if $AB \subseteq CD$ is a valid IND, then $BA \subseteq DC$ must also hold. Thus, it is sufficient to check (and report) only a single IND candidate from such permutation classes. Omitting these two kinds of INDs reduces both the amount of resources needed during the discovery process and the size of the output that usually needs to undergo further, often manual, processing.

On the face of those benefits, we propose to extend the criteria for omissible INDs from syntactical properties to instance-based properties, i.e., properties of the data comprised in the columns of an INDs. Specifically, we argue that columns that contain only NULL values (which we call *NULL columns*) and columns that contain only a single distinct value (which we call *constant columns*) are only contained in INDs that are degenerate and not actually useful for typical IND use-cases, such as those described in Sect. 1. Thus, it is fair to omit those *scrap INDs* – and it is also significant, because in our experiments we found scrap INDs to appear quite frequently.

NULL columns. There are several ways to interpret NULL values, e.g., using possible-world semantics or simply treating it as another domain value [Kö16]. Another approach is to treat NULLs as "no value", which conforms to the semantics of foreign keys in SQL. Under that interpretation, a column with only NULLs basically contains no values at all; its value set is the empty set. Because the empty set is a subset of all other sets, for a NULL column A and *any* other column B, $A \subseteq B$ is a valid IND. However, not describing an actual inclusion of values, this IND is unlikely useful. Furthermore, any other IND $X \subseteq Y$ can be extended to $XA \subseteq YB$, where A and X, as well as B and Y lie in the same respective tables. Again, this extension is not useful, because it does not refine $X \subseteq Y$, i.e., AX does not discern tuples beyond X. NULL columns are a common phenomenon. They can occur

when schemata provide overly detailed column sets or simply when the data for a column cannot be ascertained. Therefore, FAIDA detects NULL columns during the preprocessing (cf. Sect. 4.1), removes them from candidate generation, and reports them in the end. In this way, no INDs involving NULL columns will be discovered and the user is informed why.

Constant columns. We call a column constant if it stores the same non-null value for every tuple. During the analysis of several real-world datasets, we found that in all cases of such constant columns, the value in question (e.g., "−1" or an empty string) either is a surrogate for a NULL value or a default value (in the sense of SQL's `DEFAULT` keyword). Arguably, INDs containing constant columns are omissible: If the constant is a NULL surrogate, then the same rationale as for NULL columns applies; in any other case, constant columns still do not provide much value, because they do not discern the tuples of their table. Such INDs with constant columns can bloat the IND search and result space. In particular, two constant columns A and B with the same value can be added to any IND $X \subseteq Y$ and form the valid IND $XA \subseteq YB$ where A and X as well as B and Y are from the same table. Thus, FAIDA also detects constant columns in order to report and remove them.

By excluding the two described kinds of scrap INDs, FAIDA often gains significant performance improvements and, at the same time, enhances the quality of the discovered INDs. Note that the removal of scrap INDs is an additional, optional improvement of FAIDA and not a necessity to run the algorithm.

6 Evaluation

In our evaluation, we demonstrate both the efficiency and effectiveness of FAIDA. Regarding efficiency, we want to answer two main questions: (i) *How does FAIDA compare to an exact state-of-the-art IND discovery algorithm, namely BINDER?* (ii) *How well does FAIDA scale to large datasets?* To investigate the effectiveness, we address the following questions: (iii) *How good is FAIDA's result quality and to what extent is it influenced by its parameterization?* (iv) *What are the effects of omitting the scrap INDs?* We first briefly describe our experimental setup and then answer these questions in various experiments.

6.1 Experimental setup

Hardware. All experiments were run on a machine with an Intel Core i5-4690 CPU with 1600 MHz, 8 GB of main memory, and a Seagate Barracuda ST3000DM001 3 TB hard disk. We used Ubuntu 14 and the Oracle JRE 1.8u45 with a maximum 6 GB heap size.

Datasets. The datasets used for evaluation are all publicly available. Some details about those datasets are listed on the left-hand side of Tab. 3. Further information and links for all datasets, as well as an implementation of FAIDA, can be found at `https://hpi.de/naumann/projects/repeatability/data-profiling/metanome-ind-algorithms.html`.

Parameterization. FAIDA is configured via two parameters: The sampling-based inverted index requests the number of values to sample from each column, and the HYPERLOGLOG structures require a desired accuracy of their count estimates, which effectively designates their number of buckets. While FAIDA is guaranteed to find all INDs, it potentially reports incorrect INDs due to its approximative nature. Thus, the configuration of the two parameters impacts FAIDA's output quality: larger samples and more HYPERLOGLOG counters reveal violations in IND candidates more accurately. In our experiments, we set the sampling parameter to a default of 500 and the HYPERLOGLOG accuracy to a default of 0.1 %, which roughly allocates 640 KiB of main memory for 1,000,000 buckets per HYPERLOGLOG structure. Sect. 6.4 investigates FAIDA's sensitivity w.r.t. this parameterization and shows that our defaults are a rather conservative and robust choice that incur no or only very few false positive INDs. Thus, our defaults are a reasonable choice for the following comparison with BINDER.

6.2 Comparison of FAIDA and BINDER

The premise of approximate IND discovery is that a little loss in result quality can be traded for large performance improvements. Even though FAIDA always discovered exactly the correct INDs in our experiments, it relinquishes correctness guarantees, and, in turn, it should be more efficient than exact IND discovery algorithms. To verify this, we compare FAIDA's runtimes on various datasets with those of the state-of-the-art algorithm for exact IND discovery, BINDER [Pa15]. Note that FAIDA prunes scrap INDs, as introduced in Sect. 5. This novel pruning technique is not restricted to approximate IND discovery and can be applied to other IND discovery algorithms as well. To allow a fair comparison, we modified BINDER to also prune scrap INDs and provide a separate evaluation of the scrap IND pruning in Sect. 6.5.

In addition, we considered simple approximate IND discovery baselines. To determine the impact of using hashes rather than actual value combinations, we modified BINDER to hash long value combinations and operate on those hashes then. This modification always was approximately 20 % slower, because the additional hashing costs could not be redeemed. Also, we considered FAIDA without its inverted index, thereby detecting INDs solely using HYPERLOGLOG. However, while the performance overhead of the inverted index is small, leaving it out often causes false positive INDs. Those yield unnecessary IND candidates, so that eventually performance declines (see Sect. 6.4). With these modifications being inferior, we focus only on FAIDA and BINDER in the following.

Tab. 3 shows the results and runtimes of both algorithms to discover the INDs in various datasets. FAIDA outperforms BINDER consistently by a factor of five to six. Both algorithms generated and tested exactly the same IND candidates, which means that FAIDA's data preprocessing and approximate IND tests are more efficient than BINDER's exact, hash partition-based IND test.

The reason for this improvement is two-fold. At first, FAIDA tests INDs using compact hashes rather than the actual values from the datasets, which allows for more efficient comparisons

and reduces memory requirements. For instance, the TESMA dataset contains a lot of long string values. Although, this dataset contains only unary INDs, FAIDA's hashing approach can drastically reduce the computation load and easily redeems its data preprocessing overhead. In addition to that, value combinations of n-ary IND candidates often become quite long. Again, FAIDA represents those by a single hash value. The second reason for the performance improvement is found in the fact that FAIDA summarizes large datasets with small HYPERLOGLOG structures and does not need to do any out-of-core execution. BINDER, in contrast, needs to spill data to disk when processing large datasets. The I/O effort can slow it down severely – in particular for long values and value combinations, respectively.

Dataset	Size	Non-constant columns	Non-scrap n-ary INDs	Max. arity	Runtime BINDER	Runtime FAIDA
CENSUS	117 MB	48	222	6	39 sec	6 sec
WIKIRANK	730 MB	25	118	6	2 min 44 sec	26 sec
TESMA	1.2 GB	114	2	1	1 min 36 sec	25 sec
TCP-H 70	79.4 GB	60	111	3	9 h 32 min	1 h 47 min

Tab. 3: Comparative evaluation for n-ary IND detection.

6.3 Scalability

On the face of ever-growing datasets, scalability is an important property of IND discovery algorithms. In particular, we investigate two scalability dimensions, namely the number of rows and the numbers of columns in a dataset, and compare FAIDA with BINDER along these dimensions.

Row Scalability. To analyze the row scalability of FAIDA, it makes sense to reduce the impact of other factors affecting its runtime, such as value distributions and the number of INDs among the test datasets. To keep those other impact factors steady across datasets of different size, we use the TPC-H dataset generator to create datasets with varying numbers of rows but with the same schema and the same foreign keys. Nevertheless, we observed a few more, likely spurious, INDs, as the randomly generated data volume increases. Because their number is very small, they hardly affect runtime, though: TCP-H 1 has 104 INDs while TCP-H 100 has 113.

Fig. 3 displays the results of the row scalability experiment for FAIDA and BINDER. While both algorithms exhibit a linear scaling behavior, FAIDA is always around five times faster than BINDER. In other words, the larger the dataset is, the greater are the absolute time savings of FAIDA compared to BINDER.

Column Scalability. To test the runtime behavior with regard to the number of columns, we used a subset of the PDB dataset, namely 15 tables with at least 20 columns each. Then, we executed FAIDA and BINDER 20 times on those tables, thereby only taking into account the first k columns for each $1 \leq k \leq 20$. Incrementing the number of considered columns

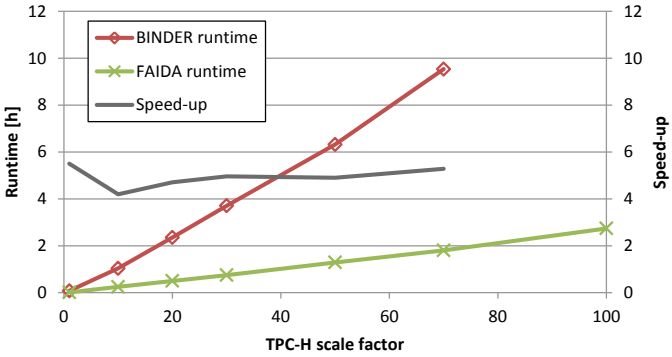

Fig. 3: Runtime scaling of BINDER and FAIDA with the number of rows.

in each table, rather than just incrementing the number of considered tables, mitigates the runtime impact of varying numbers of tuples in the tables and yields a smooth increase of the processed data volume.

Fig. 4 shows the runtime of FAIDA and BINDER together with the number and distribution of discovered INDs. Apparently, both algorithms scale somewhat linearly w.r.t. the number of INDs. Nevertheless, FAIDA scales a lot better in the presence of n-ary INDs, where its approximation schemes take particular effect: At first, FAIDA resorts to its hashed column store to test IND candidates, while BINDER has to re-read the complete input dataset multiple times to test IND candidates of different arities. Second, FAIDA works exclusively on compact hashes; BINDER in contrast concatenates values and shuffles the larger value combinations to test n-ary IND candidates. Finally, FAIDA's HYPERLOGLOG structures keep its memory footprint relatively small, while BINDER at some point needs to spill the mentioned value combinations to disk in order to shuffle them. This spilling causes BINDER's drastic runtime increase for more than 250 columns. All of the above experiments demonstrate that FAIDA trades result correctness for considerable performance gains.

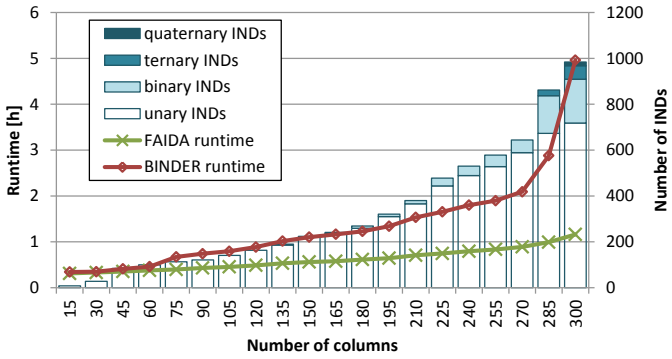

Fig. 4: Runtime scaling of BINDER and FAIDA with the number of columns.

6.4 Result Correctness

FAIDA uses two approximate data structures to test IND candidates: a sampling-based inverted index and HYPERLOGLOG structures, both of which can trade main memory requirements for accuracy. Hence, it is important to size them in a way that lets FAIDA yield accurate results without straining main memory too much. To explore this trade-off, we executed FAIDA with different HYPERLOGLOG accuracies (see Sect. 4.2) and column sample sizes (see Sect. 4.3), thereby measuring the *false-positive rate*, i.e., the ratio of incorrectly reported INDs.

Tab. 4 displays the *maximum* false-positive rate of FAIDA across the seven datasets COMA, CENSUS, BIOSQLSP, WIKIRANK, CATH, TESMA, and TPC-H[3], and reveals two interesting insights. First, it is clearly visible that the sampling-based inverted index and the HYPERLOGLOG structures complement one another. In particular, the HYPERLOGLOG structures alone did not always yield exact results and the inverted index has to be quite large to achieve full correctness. However, in combination, the two data structures exhibit superior performance. As a second insight, it becomes apparent that a reasonably sized inverted index and HYPERLOGLOG structures can robustly provide exact IND results. As a matter of fact, the column sample size of 500 and the HYPERLOGLOG accuracy of 0.1 % that we used in our efficiency experiments turn out to be a rather conservative choice. FAIDA is quite robust with respect to parameter settings. While these results, of course, do not imply that it will discover exactly the correct INDs on any given dataset, they do indicate a high confidence in FAIDA's results.

	HLL accuracy		
Sample size	10 %	1 %	0.1 %
1	6.000	0.082	0.024
10	0.243	0.047	0.012
100	0.094	**0.000**	**0.000**
1,000	0.036	0.000	0.0000
10,000	**0.000**	0.000	0.0000

Tab. 4: Maximum false positive rate of FAIDA over various datasets under different parameterizations.

Having shown that the combination of a sampled inverted index and HYPERLOGLOG yields high precision, it is intriguing to investigate how FAIDA behaves when HYPERLOGLOG is replaced with other data summarization techniques. For this purpose, we repeated the above experiment with bottom-k sketches, as proposed in [Zh10], and with Bloom filters. To make these techniques comparable to HYPERLOGLOG, we configured the size of the Bloom filter and the number hashes in the bottom-k sketch, respectively, such that they consume as much main memory as HYPERLOGLOG for the various accuracy settings from Tab. 4. We found that bottom-k sketches are not a good choice: Although still yielding good results, bottom-k sketches performed *at most* as well as (but often worse than) HYPERLOGLOG and Bloom

[3] See `https://hpi.de/naumann/projects/repeatability/data-profiling/metanome-ind-algorithms.html` for downloads and details of these datasets.

filters under all parameterizations. This is because bottom-k sketches do not partition the hash space, which would allow a pairwise comparison of their hash values, as is the case for bits in a Bloom filter or buckets in a HyperLogLog structure. However, we also found that Bloom filters performed similarly well as HyperLogLog and are an eligible replacement.

6.5 Omitting scrap INDs

Scrap INDs are those INDs that involve either NULL columns (columns containing no values other than NULL) and/or constant columns (columns containing only a single value). In Sect. 5 we argue that such INDs are not meaningful for the typical IND-based applications and ignoring them could save much computation. It remains to show that the class of scrap INDs is common and its dedicated treatment worthwhile.

To this end, we analyzed the different types of INDs in various datasets. The results are displayed in Fig. 5. Approximately two thirds of all INDs in this experiment are scrap INDs. While the majority of scrap INDs involve NULL columns, we observe that datasets can also comprise many scrap INDs related to constant columns, such as ENSEMBL. Furthermore, we measured the runtime of Faida with and without pruning of scrap INDs and found it to be beneficial. While for three out of the seven datasets, performance was not affected, for the other four datasets, the pruning indeed yielded a performance improvement. On the EMDE dataset, particularly, we observed a speed-up of factor 20. In consequence, it seems appropriate to detect and prune scrap INDs already during the IND discovery process.

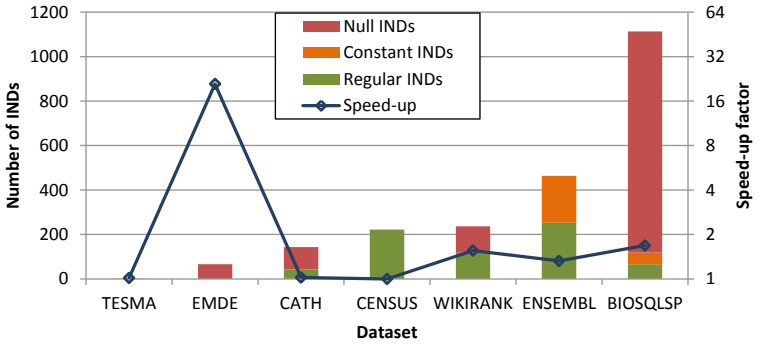

Fig. 5: Break-down of IND types for various datasets.

7 Conclusion

We presented Faida, an approximate algorithm for the n-ary IND discovery problem. Faida uses a symbiotic combination of data preprocessing, hashes, a sampling-based inverted index, and HyperLogLog to test INDs in a highly efficient and scalable manner. In our experiments, we found our algorithm to be as much as six times faster than the exact state-of-the-art IND discovery algorithm Binder. Besides performance aspects, Faida further guarantees result completeness, i.e., it will find all INDs in a given dataset.

Although incorrect INDs might be reported, FAIDA did not yield any false positives in our experiments. This shows the effectiveness of our hybrid IND test. A promising direction for future research is to adapt FAIDA for incremental IND discovery: With the low memory footprint of its data structures, FAIDA might be a particularly good fit to maintain a set of INDs on evolving, dynamic datasets. However, especially updating those data structures on value deletions or changes is a challenging task.

Acknowledgements. This research was partially funded by the German Research Society (DFG grant no. FOR 1306).

References

[AGN15] Abedjan, Ziawasch; Golab, Lukasz; Naumann, Felix: Profiling relational data: a survey. VLDB Journal, 24(4):557–581, 2015.

[Ba07] Bauckmann, Jana; Leser, Ulf; Naumann, Felix; Tietz, Véronique: Efficiently detecting inclusion dependencies. In: Proceedings of the International Conference on Data Engineering (ICDE). pp. 1448–1450, 2007.

[Bo05] Bohannon, Philip; Fan, Wenfei; Flaster, Michael; Rastogi, Rajeev: A cost-based model and effective heuristic for repairing constraints by value modification. In: Proceedings of the International Conference on Management of Data (SIGMOD). pp. 143–154, 2005.

[CTF88] Casanova, Marco A.; Tucherman, Luiz; Furtado, Antonio L.: Enforcing Inclusion Dependencies and Referencial Integrity. In: Proceedings of the International Conference on Very Large Databases (VLDB). pp. 38–49, 1988.

[DMLP09] De Marchi, Fabien; Lopes, Stéphane; Petit, Jean-Marc: Unary and n-ary inclusion dependency discovery in relational databases. Journal of Intelligent Information Systems, 32(1):53–73, 2009.

[DMP03] De Marchi, Fabien; Petit, Jean-Marc: Zigzag: a new algorithm for mining large inclusion dependencies in databases. In: Proceedings of the International Conference on Data Mining (ICDM). pp. 27–34, 2003.

[Fl07] Flajolet, Philippe; Fusy, Éric; Gandouet, Olivier; Meunier, Frédéric: HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In: Proceedings of the International Conference on Analysis of Algorithms (AofA). pp. 127–146, 2007.

[Gr98] Gryz, Jarek: Query folding with inclusion dependencies. In: Proceedings of the International Conference on Data Engineering (ICDE). pp. 126–133, 1998.

[Kö16] Köhler, Henning; Leck, Uwe; Link, Sebastian; Zhou, Xiaofang: Possible and certain keys for SQL. VLDB Journal, 25(4):571–596, 2016.

[KPN15] Kruse, Sebastian; Papenbrock, Thorsten; Naumann, Felix: Scaling out the discovery of inclusion dependencies. In: Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW). pp. 445–454, 2015.

[KR03] Koeller, Andreas; Rundensteiner, Elke: Discovery of high-dimensional inclusion dependencies. In: Proceedings of the International Conference on Data Engineering (ICDE). pp. 683–685, 2003.

[LPT02] Lopes, Stéphane; Petit, Jean-Marc; Toumani, Farouk: Discovering interesting inclusion dependencies: application to logical database tuning. Information Systems, 27(1):1–19, 2002.

[Pa15] Papenbrock, Thorsten; Kruse, Sebastian; Quiané-Ruiz, Jorge-Arnulfo; Naumann, Felix: Divide & conquer-based inclusion dependency discovery. Proceedings of the VLDB Endowment (PVLDB), 8(7):774–785, 2015.

[Ro09] Rostin, Alexandra; Albrecht, Oliver; Bauckmann, Jana; Naumann, Felix; Leser, Ulf: A Machine Learning Approach to Foreign Key Discovery. In: Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB). 2009.

[SM16] Shaabani, Nuhad; Meinel, Christoph: Detecting Maximum Inclusion Dependencies without Candidate Generation. In: Database and Expert Systems Applications (DEXA). pp. 118–133, 2016.

[Zh10] Zhang, Meihui; Hadjieleftheriou, Marios; Ooi, Beng Chin; Procopiuc, Cecilia M; Srivastava, Divesh: On multi-column foreign key discovery. Proceedings of the VLDB Endowment (PVLDB), 3(1-2):805–814, 2010.