Detecting Inclusion Dependencies on Very Many Tables

FABIAN TSCHIRSCHNITZ, THORSTEN PAPENBROCK, and FELIX NAUMANN,
Hasso Plattner Institute

Detecting inclusion dependencies, the prerequisite of foreign keys, in relational data is a challenging task. Detecting them among the hundreds of thousands or even millions of tables on the web is daunting. Still, such inclusion dependencies can help connect disparate pieces of information on the Web and reveal unknown relationships among tables.

With the algorithm Many, we present a novel inclusion dependency detection algorithm, specialized for the very many—but typically small—tables found on the Web. We make use of Bloom filters and indexed bit-vectors to show the feasibility of our approach. Our evaluation on two corpora of Web tables shows a superior runtime over known approaches and its usefulness to reveal hidden structures on the Web.

CCS Concepts: • **Information systems → Incomplete data**; *Data extraction and integration*;

Additional Key Words and Phrases: Inclusion dependency discovery, foreign key discovery, data profiling, web data management

ACM Reference format:
Fabian Tschirschnitz, Thorsten Papenbrock, and Felix Naumann. 2017. Detecting Inclusion Dependencies on Very Many Tables. *ACM Trans. Database Syst.* 42, 3, Article 18 (July 2017), 29 pages.
https://doi.org/10.1145/3105959

1 JOINING THE WEB OF TABLES

The Web consists of an almost uncountable amount of small and relational structured datasets. We can find them in millions of tables embedded in HTML documents. If we take them on their own, they are mostly not very expressive due to their isolation. If we combine them, however, we might find new interesting information.

As an example, consider only the articles in the English Wikipedia category "Astronomical objects." It consists of 37 subcategories and hundreds of articles. Many of the articles contain tables, such as the one shown in Figure 1, which contains some interesting attributes of the planets in our solar system. The first column, Planet, lists the entities of this table and it can be considered the table's primary key.

In addition to this table, a user might stumble over several other tables while browsing through the articles, including those of Figure 2. The first one provides some more information about only the terrestrial planets, such as their density. For users, it would be useful to have an integrated view

Authors' addresses: F. Tschirschnitz, T. Papenbrock, and F. Naumann, Hasso Plattner Institute, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany; emails: fabian.tschirschnitz@student.hpi.de, thorsten.papenbrock@hpi.de, felix.naumann @hpi.de.
Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2017 ACM 0362-5915/2017/07-ART18 $15.00
https://doi.org/10.1145/3105959

Planet	Equatorial diameter[a]	Semi-major axis (orbit) (AU)	Orbital period (years)[a]	Inclination to Sun's equator (°)	Orbital eccentricity	Rotation period (days)	Confirmed moons	Rings
Mercury	0.382	0.31	0.24	3.38	0.206	58.64	0	no
Venus	0.949	0.72	0.62	3.86	0.007	−243.02	0	no
Earth	1.00	1.00	1.00	7.25	0.017	1.00	1	no
Mars	0.532	1.58	1.88	5.65	0.093	1.03	2	no
Jupiter	11.209	5.20	11.86	6.09	0.048	0.41	67	yes
Saturn	9.449	9.54	29.46	5.51	0.054	0.43	62	yes
Uranus	4.007	19.22	84.01	6.48	0.047	−0.72	27	yes
Neptune	3.883	30.06	164.8	6.43	0.009	0.67	14	yes

Fig. 1. An example Web table about the solar system's planets.

Object	Density (g cm^{-3})		Semi-major axis (AU)
	Mean	Uncompressed	
Mercury	5.4	5.3	0.39
Venus	5.2	4.4	0.72
Earth	5.5	4.4	1.0
Mars	3.9	3.8	1.5

Name	Satellite of	Difference in axes	
		km	% of mean diameter
Mimas	Saturn	33.4 (20.4 / 13.0)	8.4 (5.1 / 3.3)
Enceladus	Saturn	16.6	3.3
Miranda	Uranus	14.2	3.0
Tethys	Saturn	25.8	2.4
Io	Jupiter	29.4	0.8
Moon (*Luna*)	Earth	4.3	0.1

Fig. 2. A table of terrestrial planets and one of some natural satellites.

over the tabular data by joining related tables. In that way, they would see all related information at a glance. Such an integrated overview could be achieved by building a join graph that connects related tables and, hence, their information.

If users had an aggregated view on the data, they could also see that both underlying tables include an attribute with nearly the same name, which is Semi-major axis. For some planets, the values in these two attributes differ. This difference is a *data quality* issue within Wikipedia, which can already be observed among only the hundreds of tables from that limited topic. The column Satellite of in the second table of Figure 2 references the planets from the example table in Figure 1. In the terms of the relational model, it acts as a good foreign key candidate. By joining the two tables, users could easily query the resulting relation for "all moons of planets with rings," for example.

Tabular data is a great source for existing data that can be combined and queried, and it can be found all over the Web. For instance, the Web Data Commons project[1] extracted 154 million tables from the *Common Web Crawl*. Cafarella et al. (2008) showed interesting use cases for that dataset, such as specialized table search and table extensions. To build systems on the basis of Web table datasets, we need metadata, such as information about the relationships between them. Currently, this metadata does not exist. In contrast to many other datasets, this information did not simply get lost or outdated over the time. In fact, it never existed.

The question is how to find related tables in the extremely large number of candidates. An ideal system finds all related tables not only from a small subset of Wikipedia articles but also from the entire Web. However, web tables are not designed to be automatically linked and do not provide any clear hints for doing so. Such hints could be explicitly defined key and foreign key relationships, as we know them from Relational Database Systems (RDBMS). Additionally,

[1] http://webdatacommons.org/webtables *Accessed: 05/22/2017.*

Web tables are only “quasi” relational in that data types, schema information, and other structural information are missing. So, our goal is to analyze all of these isolated datasets to link them in a way that their information can be combined. For this task, we propose an *efficient and scalable inclusion dependency detection algorithm*. An inclusion dependency holds iff all values of one column are included in another column (of the same table or, more interestingly, of another table). These shared values suggest joins among the two tables. So far, IND detection has been attempted only for datasets with much fewer tables and columns. This article focuses on the mining of INDs in datasets with *several millions* of columns provided by the Web.

Contributions and structure. We solve the problem of IND detection on a very large number of tables and show why known approaches are incapable of handling such a large amount of tables or have poor efficiency in doing so. In particular, we introduce the MANY algorithm, a novel, efficient, and scalable approach for this task. Based on our findings, we introduce filter criteria on the input data as well as on the IND candidates to keep the result set size reasonable for later human processing. In addition, we take the filtered IND-set to visualize the join graph, providing exploration capabilities to the user. We finally evaluate our algorithm in comparison to known approaches and show how parameters and implementation details affect its efficiency.

2 FOUNDATIONS AND RELATED WORK

Inclusion dependencies were identified to be one of the most important dependency types in the field of data profiling (Marchi et al. 2002). They suggest joins among tables and can be used to recover foreign key relationships. In the following section, we provide a common notation and formal definition. Afterward, we cover different known detections approaches.

2.1 Foundations

Intuitively speaking, an inclusion dependency $R_1[X] \subseteq R_2[Y]$ holds for two instances of the relational schemata R_1 and R_2 with attributes X and Y if all values in the projection $R_1[X]$ are also included in $R_2[Y]$. To provide a more formal definition, we adopt the following notations (Marchi et al. 2002; Bauckmann et al. 2007; Papenbrock et al. 2015b):

Let $D = \{R_1, R_2, \ldots, R_m\}$ be a database schema as a set of relational schemata R_j. Let $d = \mathcal{I}(D) = \{r_1, r_2, \ldots, r_m\}$ be the instance of such a database schema. Here, r_j corresponds to a set of tuples $\{t_1, \ldots, t_k\}$ of schema R_j. Let $\pi_X(r_j)$ denote the projection of r_j on attribute X from R_j and $t[X]$ the restriction of tuple t to X so that $\pi_X(r_j) = \{t[X]\ s.t.\ t \in r_j\}$. The IND $R_a[X] \subseteq R_b[Y]$ between the two attributes X and Y is satisfied by instance d over D iff $d(R)[X] \subseteq d(R)[Y]$. The left-hand side X of an IND is called the *dependent* attribute and the right-hand side is called the *referenced* attribute. This definition can be extended to attribute sets and thus n-ary inclusion dependencies, which are not the focus of this article.

To verify one IND, each record of the projection on X must be proved to be contained in the projection on Y. However, the problem of detecting *all* INDs is inordinately harder. Ignoring trivial IND candidates of the form $R[X] \subseteq R[X]$, a database schema with a attributes already has $a(a-1)$ IND candidates that need to be verified or falsified. When searching for n-ary INDs, the candidate space even grows exponentially (Papenbrock et al. 2015b) and is infeasible for the enormous number of tables and attributes of our use-case (Bläsius et al. 2016).

The knowledge of INDs between relations can be used to find foreign keys in attributes of the relations. Each IND, together with the knowledge about the fulfilment of the uniqueness property of the referenced attribute or attribute sequence, forms a foreign-key candidate: Let $R_1[X] \subseteq R_2[Y]$ be a valid IND between two attributes X and Y of the relational schemata R_1 and R_2. We then call attribute X a *foreign key candidate* if attribute Y is a key candidate; that is, its values are

unique and do not include the null value. Attribute X is truly a foreign key only if its values also semantically reference values in Y. This latter condition is obviously not necessarily fulfilled if only the syntactically checkable part (the foreign-key candidate property) is given.

2.2 Related Work: IND Detection

Bell and Brockhausen (1995) introduced an IND detection algorithm on the basis of *SQL* using `join`-statements. To reduce the number of checks, they enumerated candidates from previously collected data statistics, such as data types and value ranges. During the test phase, already known INDs and transitivity are taken into account. However, the SQL-based IND validation is expensive: Loading a large number of tables into an RDBMS takes time, and the multitude of small tables, in particular, causes very many queries that become a dominating overhead.

De Marchi et al. proposed an algorithm that first divides the data according to its different data types, followed by the creation of an inverted index per data type (Marchi et al. 2002). In that index, each occurring value points to the set of all attributes containing that value. One can retrieve the set of all valid INDs by intersecting all of those attribute sets. This approach constitutes a powerful technique that builds the foundation for later algorithms in that field. The inverted index must, however, fit into the machine's main memory, which is not the case for most real-world datasets and especially not for our Web table scenario. For large numbers of attributes, the candidate sets also become extremely large and space-consuming. In particular, the value distributions that occur in Web tables where some values occur extremely frequently in a huge number of columns, cause very large candidate sets and costly intersection operations.

Based on the *sort-merge-join* algorithm, Bauckmann et al. proposed SPIDER (Bauckmann et al. 2007), which consists of two phases: The first phase reads all values for one attribute, sorts them, and writes them as a duplicate-free sorted list back to disk. The second phase then reads all attributes in parallel, keeping an open file handle on every sorted column file and sorting among the attributes by their current minimal value using a *min-heap*. When running over all values, SPIDER prunes a set of IND candidates with every set of same-value attributes, similar to de Marchi's algorithm. It is, however, able to prune entire attributes from the process if all their candidates have been invalidated. Because SPIDER needs an open file for each attribute, it is not capable of detecting INDs among several thousands of attributes. Its runtime is also often dominated by the sorting phase because tables are read multiple times (i.e., once for each attribute). Interestingly, the authors also evaluated the use of *Bloom filters* to prune the candidate space. In Bauckmann et al. (2010), the authors recognized that Bloom filters can be used as column signatures that preserve subset relationships. In contrast to our solution, they were not able to achieve promising efficiency gains due to pair-wise comparisons of the bit-signatures, which we avoid.

The BINDER approach by Papenbrock et al. bucketizes the column values from the input with the help of a hash-function (Papenbrock et al. 2015b). The algorithm assures that a single partition fits into main memory by iterative refinement and automatic memory management. In the validation phase, the algorithm creates for each partition two indices: One index maps the single attributes to their value lists; and one inverted index, which is similar to de Marchi's index, maps every value from a partition to all the attributes it occurs in. The partitioning and the two indices allow BINDER to circumvent memory limitations and to prune much more aggressively than other related algorithms. We perform a *comparative evaluation* of our approach with both SPIDER and BINDER.

2.3 Related Work: Foreign Keys

Our main motivation for this work is to discover unknown relationships in the form of INDs among web tables. The classification whether an IND indeed is also a foreign key is addressed in different works.

Rostin et al. achieved good results with a machine learning–based approach (Rostin et al. 2009). They built a 10-dimensional feature vector for each IND and trained different classifiers on real-world relational datasets. They show that a classifier trained on one specific dataset can achieve a good *F-measure* on a different dataset as well. However, a foreign key detection approach that relies exclusively on machine-learning and the complete set of INDs in the dataset is not applicable for our domain. The sheer amount of valid INDs among the relations requires some prefiltering. To this end, we make use of several of the features described in Rostin et al. (2009).

Lopes et al. follow a different path, not relying on the knowledge of all INDs but on a provided *SQL*-workload (Lopes et al. 2002). From this workload, they deduce the set of foreign keys by assuming that join operations are usually performed by equating a key with a foreign key. As there is no existing workload for even a small fraction of tabular Web data, this approach is not applicable for us.

Another approach that does not rely on the knowledge of the set of INDs is described by Zhang et al. (2010). The main idea of their approach is to determine if the dependent side of a foreign key candidate is a "good" sample of the referenced part. One issue with this detection process is that it cannot well handle the small size of Web tables: Two tables might be related by an IND, but this IND might not indicate a foreign key relationship according to that approach. Consider, for example, a table collecting certain economic information about the Baltic countries. A foreign key relationship between the "Country" column of that table and a table that collects knowledge of all countries in the world is preferable to be detected. The approach by Zhang et al. would likely not detect this foreign key relationship because the sample of the three Baltic states is extremely small.

2.4 Related Work: Web Tables

Carafella et al. analyzed 14.1 billion *HTML* tables from Google's general purpose Web crawl (Cafarella et al. 2008). The authors derived the *Attribute Correlation Statistics Database* (AcsDB), which stores statistics on co-occurrences of schema elements. With the help of that metadata, they developed novel applications, such as schema auto-completion, attribute-synonyme finding, and, most interestingly to us: join-graph traversal. The latter enables navigation between extracted schemas based on shared attributes (with same label) across tables. In contrast, our IND-based approach finds links between two schemas in an instance-based manner and does not rely on schema information, which is usually not available anyway.

Bhagvatula et al. extracted nearly 1.4 million tables from the English Wikipedia for the *WikiTables* project (Bhagavatula et al. 2013). To join tables, the authors tailored a hybrid approach: In a first step, they heuristically filtered most of the more than 7 million columns because they were not considered as possible key or foreign key columns (e.g., integer columns). Then they computed a pairwise *MatchPercentage* for the remaining 1.75 million columns, keeping those with a score > 50%. When a user queries for related columns based on a particular table of interest, their system computes a set of candidate columns and ranks them with the help of a machine learning algorithm. The main problem with this approach is its quadratic runtime in the number of columns. In contrast, we propose bit-signatures for the enumeration of probable candidates to avoid a pairwise comparison of all columns.

Yakout et al. created the *InfoGather* system on a huge Web table dataset (Yakout et al. 2012), gathering 573 million Web tables from a *Bing* crawl. Their goal was to augment entities by providing a desired attribute name for a given entity (e.g., brand for a certain camera model) or by providing other entities from the same domain together with values of the desired attribute (camera models with their brand). For that, they developed an "attribute discovery" mechanism for entities as inputs. The proposed system takes entities wrapped in *query-relations* to augment them with related

Table 1. Sizes of Tabular Datasets Extracted from the Web

		Columns		Rows	
Dataset	# Tables	Ø	min–max	Ø	min–max
WDC	147,636,113	3.49	2–2,368	12.41	1–70,068
WikiTables	1,398,105	5.34	0–2,349	10.85	0–4,670

tables. While they also take value overlaps between the relations into account, they do not explicitly detect INDs. Our goal is not to find information relevant to a user-specified query-relation, but to create a join graph for Web tables based on INDs among them that suggest meaningful joins.

Similar to the approach of Yakout et al. (2012); Abedjan et al. (2015) also utilize Web tables to complete binary query-relations asked by the user. They evaluated two different storage and querying approaches for Web tables: One uses a column-oriented RDBMS and the other the Lucene information retrieval framework. The latter stores each column from the Web tables as an indexed document vector. To detect tables that include parts of the query-relation, a single-keyword query is issued against Lucene for each value in a column from the query-relation. For both approaches, the reported runtimes are prohibitive for our use-case.

3 THE ANATOMY OF WEB TABLES

In terms of the numbers of attributes and relations, the *WDC Webtables* dataset is bigger than any other by an order of five magnitudes (Cafarella et al. 2008). Not only the sheer amount of tables makes such datasets so special. Their origin from thousands of authors implies heterogeneity and poor data quality. This section first introduces the peculiarities of these tables that affect IND detection. Then it presents our designated use-case, finding join paths that allow us to prune the search space to a feasible size.

3.1 Web Table Properties

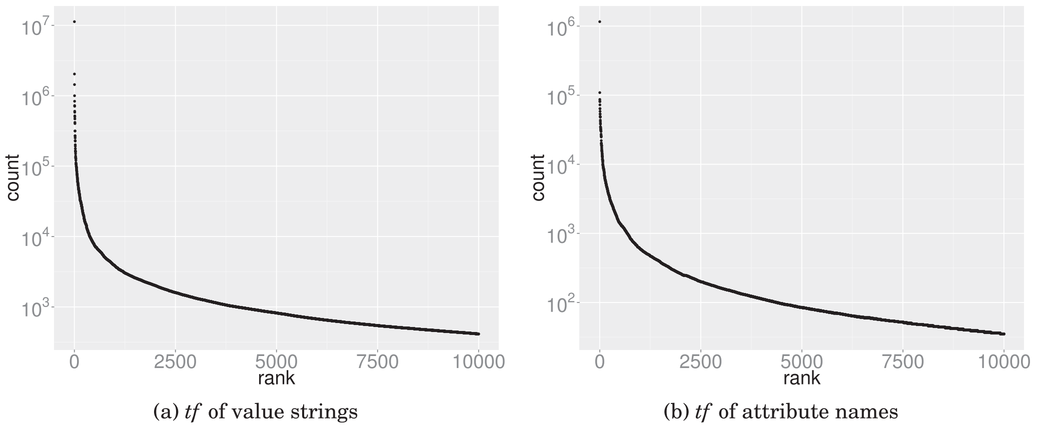
We focus on both already-mentioned datasets; *WDC Webtables* (Cafarella et al. 2008) and *WikiTables*.[2] Some basic statistics can be found in the Table 1.

Although the *WDC Webtables* dataset is larger than the *WikiTables* dataset by two orders of magnitudes, their shapes are similar: On average their tables have only a handful of columns and around a dozen rows, so are small compared to other relational real-world datasets. Nevertheless, the range for the number of columns and rows is surprisingly large.

The average number of tuples per table is important for our later approach, because it directly influences the algorithm's parametrization: We generate signatures for every column in the input, and those signatures are less expressive the higher the relation of distinct values are in the column.

We exemplarily discuss the values from the *WikiTables* dataset, which contains 477,074 different attribute names and 13,742,963 distinct values. All values together occur 88,808,438 times. Figure 3 provides plots on the frequencies of the values as well as the attribute names in the *WikiTables* dataset. Both distributions closely follow Zipf's law. The by far most frequent string in the dataset for both values and attributes is the empty string. The 25 most frequent values are the numbers "1" (2nd) to "22" (25th) in order and the single characters "—" (6th), "-" (11th) and "–" (19th) among them. Together, they occur 24,131,756 times and therefore ~27% of all values are numeric or some representation of what we consider as `null`. As a side note: The occurrences of all integer numbers closely follow Benford's law (Benford 1938); that is, 1 occurs as the leading digit in numbers about 30.1% of the time, while larger digits occur less frequently as leading digits.

[2]http://downey-n1.cs.northwestern.edu/public *Accessed: 05/22/2017.*



(a) *tf* of value strings

(b) *tf* of attribute names

Fig. 3. Term frequency of the 10,000 most frequent strings in the *WikiTables* dataset.

Table 2. Values Considered as `null` Values

value	*tf* rank	frequency	value	*tf* rank	frequency
“”	1st	11,279,606	“- -”	176th	27,898
“—”	6th	724,917	“n/a”	182nd	26,596
“-”	11th	501,114	“●”	198th	23,574
“–”	19th	252,008	“- - -”	578th	6,619
“N/A”	33th	131,339	“.”	604th	6,367
“?”	49th	97,236	“??”	645th	6,053
“Unknown”	146th	32,484	“(n/a)”	915th	4,228

Table 3. Quadratic Growth in the Number of INDs in Random Subsets of the *WikiTables* Dataset

#tables	1,000	2,500	5,000	10,000	25,000	50,000
#INDs	122k	774k	2.5m	10m	63m	259m

Many Web tables suffer from missing or `null` values. These are manifest in many different representations: While RDBMSs provide an explicit mechanism for such values, we consider the empty string (“”) in tabular datasets as the most intuitive way to represent the absence of a value in a tuple for a certain attribute. However, among the 1,000 most frequent values, we find 13 other representations that we consider as `null` values, as shown in Table 2.

3.2 Finding INDs for Meaningful Joins

The number of INDs in a Web table dataset is enormous. Table 3 shows experimentally how the result set grows for increasing sizes of sets of randomly selected input tables from the *WikiTables* dataset. By extrapolating to the entire dataset, consisting of more than 1 million tables, it is obvious that discovering (and storing) them all is infeasible.

A closer look at the result sets reveals that most INDs hold for two reasons: A significant portion of columns is completely empty or contains only `null` values. An empty column is included in every other column in the dataset, while a column containing only `null` values is included in every other column that has at least one `null` value (assuming the usual `null` = `null` semantics of IND detection). Both cases lead to a large number of INDs that can be considered useless. The second

and even more important reason for the vast amount of INDs lies in the distribution of values: Very many columns are lists of integer values, often starting from *"1."* Consider tables containing ordered lists of things, such as rankings, for example. Those columns frequently depend on similar columns, but the IND is spurious. Based on these insights, we devise a set of four filter conditions for IND candidates and valid INDs so that the remaining INDs only suggest meaningful joins or possible foreign keys between Web tables. Note that these filters are not an essential part of the algorithm—for smaller inputs they could, in principle, be dropped.

1. **Null-column-filter**. Empty columns or those containing only `null` values are ignored.
2. **Integer-column-filter**. We have shown that many columns in our considered datasets contain integer lists. INDs among those columns are almost always of a random nature and lead to huge result sets. While foreign key relationships over integer columns that represent IDs are the rule for traditional RDBMS, such IDs are hardly present on web data because there is no issuing authority. Bhagvatula et al. also apply such a filter rule (Bhagavatula et al. 2013).
3. **Non-unique-filter**. The definition of a foreign key requires that the referenced attribute is a primary key inside its relation. Hence, only attributes with *unique* values should be taken into account as referenced candidates.
4. **Coverage-filter**. Inspired by a feature of Rostin et al. (2009), a machine learning approach to classify INDs as foreign keys or non-foreign keys, we filter INDs with a too low coverage of the values in the referenced attribute's value set based on a user defined threshold (default 20%). That is, the dependent column should cover at least a certain percentage of the distinct values in the referenced column because dependent columns with a low coverage usually arise by accident and do not represent real foreign keys.

With these filters in mind, we now move to the actual detection algorithm.

4 BIT-SIGNATURES FOR IND DETECTION

In this section, we first provide foundations by describing how to create subset-relationship preserving signatures using a probabilistic data structure. We then describe our IND detection workflow and show how to improve its efficiency by parallelization and other optimizations.

4.1 Bloom Filters

The main challenge we have to tackle is the number of potential candidates. Taking simple metadata into account alone for the candidate set generation, such as the minimal and maximum value of each column or the data type, still would lead to a vast amount of candidates to check. Our algorithm does not explicitly enumerate all IND candidates and thus keeps the initial candidate set of INDs to check small. Instead, we create bit-signatures for each of the columns in the input that act as fingerprints. Those fingerprints must fulfill one important criterion: If the value set of one column A in an instance of R_x is a subset of another column B in R_y (IND $R_x.A \subseteq R_y.B$ holds), the bit-signature of A shall also be a subset of the bit-signature of B. Similar hashing approaches are already known as *simhash* (Charikar 2002).

We decided to use Bloom filters (Bloom 1970), which are space-efficient probabilistic data structures often used to test if an element already occurred in a data stream. Intuitively, a Bloom filter is a bit-array where certain bits are set and some are not. If a value is added to a Bloom filter, a certain number of bits representing that value are set. If we want to know if a value already occurred in the data stream, we check if every bit is set to 1 that would have been set if the value were added. Bloom filters do not produce false negatives, but false-positive matches are possible. More formally:

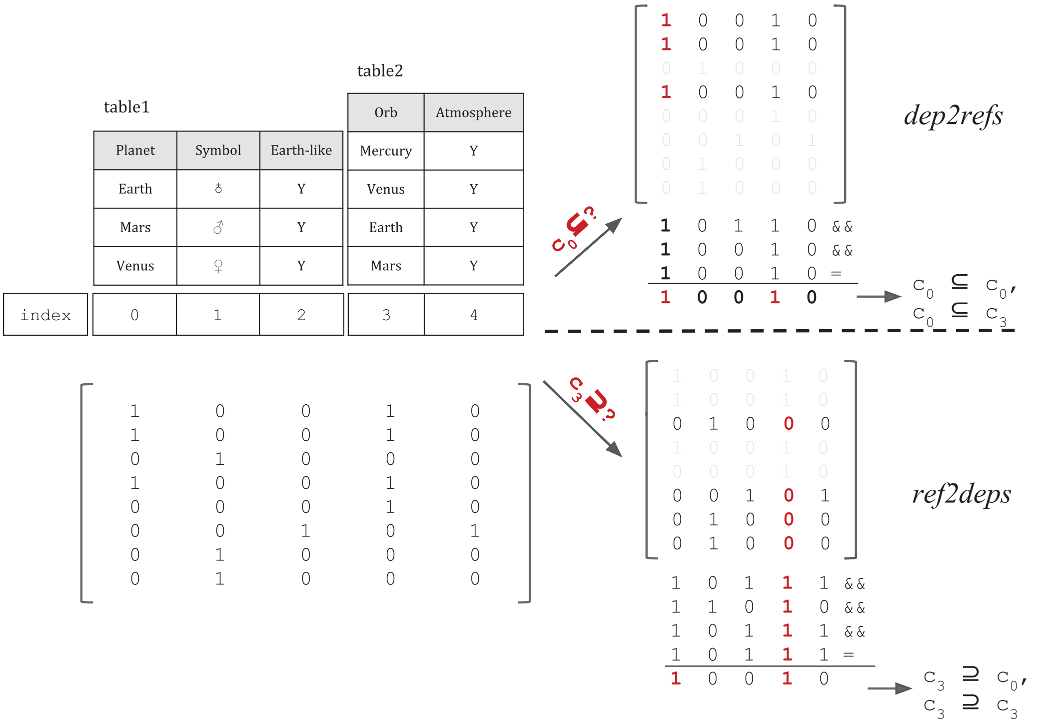


Fig. 4. Finding referenced attribute candidates for a dependent attribute candidate (*dep2refs*, top) and vice versa (*ref2deps*, bottom).

Let $B = (b_1, b_2, \ldots, b_m)$ be a binary vector of size m with all bits initially set to 0. Let $H = \{h_1, h_2, \ldots, h_k\}$ be a set of k hash-functions. Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of n different data values. The hash-functions in H are defined as $h_x : S \rightarrow [0, m-1]$. For every value in S each of the k hash-functions is computed, and, accordingly, the ith bit is set in B with i being the output of one hash-function. After all values have been added, B is the Bloom filter for S. To query a Bloom filter with a string q, one computes all hash-functions on q and verifies whether all corresponding bits are set. If that is the case, the string q *may* be in S; if not, it *cannot* be in S. A Bloom filter B_1 is a subset of another Bloom filter B_2 (denoted as $B_1 \subseteq B_2$) iff all bits set in B_1 are also set in B_2. Bloom filters fulfill the desired property of preserving subset relationships between two sets: If $S_1 \subseteq S_2$ then $B_1 \subseteq B_2$, but not vice versa.

So far, we generated only fingerprints for our columns in the dataset. Our initial goal is still to efficiently enumerate the candidate set of INDs that are worthwhile to check. Currently, we would still need to check every Bloom filter pairwise for containment. Once we observe containment in the Bloom filters, we would propose their corresponding attributes as an IND candidate. If there is no containment of one bit-array in another, we know that there is no IND between the signature generating attributes.

The pairwise comparison of Bloom filters can be substituted by a much more efficient computation, as suggested in Figure 4: First, we combine all Bloom filters into an $(m \times n)$-matrix. In this matrix, n is the number of columns in our input dataset and m is the length chosen for our Bloom filter signatures. If we want to know, e.g., in which other Bloom filters the Bloom filter of the attribute's table1.Planet value-set is included, we can check where the corresponding column c_0 in the matrix is included.

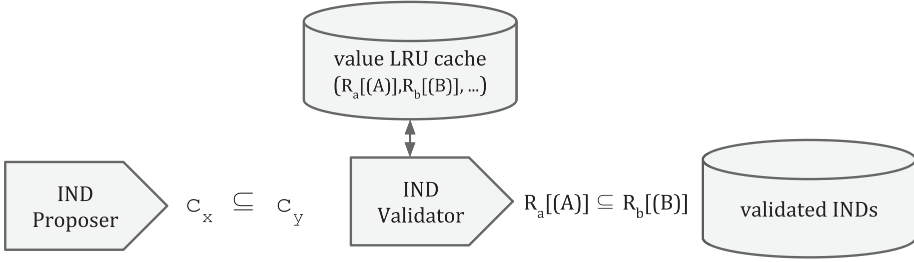


Fig. 5. Process of the IND validation in MANY.

To determine containment of a column in others in the matrix, we make use of simple and efficient bitwise and operations. For the and operation, we must consider only those lines that have their c_0-bit set because we have to check only the existence of every element in the set to be contained in the superset candidate. Hence, we consider only the rows 0, 1, and 3 from our example matrix to check inclusions for c_0.

In the example, the result of the and operations as shown in Figure 4 (top) is 10010. It indicates that the Bloom filter represented by the first column in the matrix is included trivially in itself and in the Bloom filter generated by the values of attribute table2.Orb. Therefore, we propose the IND candidate table1.Planet $\subseteq$ table2.Orb. We do not know yet whether there is indeed an IND between these two attributes, but it is likely.

Because our bit-based signature is a binary representation of the column values, we are able to find not only subset relationships, but also (without greater effort) the superset relationships for a specific column. If we want to know if an attribute is possibly referenced by others, we simply have to check if its Bloom filter is a superset of another attribute's Bloom filters. Figure 4 (bottom) shows how to enumerate potentially dependent attributes for a possibly referenced attribute. We take the same example as seen in Figure 4, but this time we want to know which columns might be contained in the attribute table2.Orb. The Bloom filter representation of this attribute's values can be found in the fourth column, c_3, of our bit-matrix.

Again, to find subsets of this bit-array, we have to logically link only those rows of the matrix where c_3 itself is set to 0. The superset relation $c_3 \supseteq c_x$ can be violated merely in those rows of the matrix where c_3 is set to 0. Instead of combining all relevant rows by an and, as before, we first take the inverse of each row and then combine them with an and. This is equivalent to applying logical or operations, but we provide an optimization for the and operation of our bit-vector implementation in Section 4.5.

As a result, in the example, we obtain the bit-vector 10010, indicating that the column c_3 of the bit-matrix is only a superset of itself and of c_0, which corresponds to the attribute table1.Planet. Thus, we know that the value set of table2.Orb might be a superset of the values in table1.Planet and propose the IND candidate table1.Planet $\subseteq$ table2.Orb.

We refer to the two different strategies to enumerate IND candidates described in Figure 4 as *dep2refs* and *ref2deps*, respectively. The part of our algorithm that generates the IND candidate is in the following called *IND-Proposer*.

After candidate preparation, the *IND-Validator* checks the correctness of every candidate by checking the subset relation of the underlying actual value sets and outputs only valid INDs. This part of the process from the final algorithm is sketched in Figure 5. To prevent redundant reads (from disk) of the same data values, a cache is used. This cache stores the projections of the tables on the single attributes.

4.2 The MANY Algorithm

After providing the theoretical foundations, we now provide an overview of our entire IND detection process. MANY takes relational data as input. It then uses the *Signature-Generator* to calculate a Bloom filter–based bit-signature for every column in the dataset. Thereafter, the *IND-Proposer* determines the set of all IND candidates containing all valid and possibly some false-positive INDs. In a final step, the *IND-Validator* checks every candidate for the necessary value containment of the dependent attribute's values. Algorithm 1 outlines this process.

ALGORITHM 1: MANY

Require: Database schema $D = \{R_1, R_2, \ldots, R_s\}$, instance of database schema $d = \mathcal{I}(D) = \{r_1, r_2, \ldots, r_s\}$, Bloom filter size $m \in \mathbb{N}$, hash-function count $k \in \{1, \ldots, m\}$, $strategy \in \{dep2refs, ref2deps\}$
Ensure: All valid INDs *uinds*

```
1:  colIndicesMap ←BUILDCOLINDICESMAP(D)
2:  function MANY(D, d)
3:      uinds ← ∅
4:      sigMatrix ← GENERATESIGNATURES(D, d)
5:      candidates ← GENERATECANDIDATES(sigMatrix)
6:      for all candidate ∈ candidates do
7:          if ISVALID(candidate, d) then
8:              uinds ← uinds ∪ {candidate}
9:      return uinds
10: return MANY(D, d)
```

MANY receives a database schema $D = \{R_1, R_2, \ldots, R_s\}$ consisting of a finite set of relational schemas as input as well as an instance $d = \mathcal{I}(D) = \{r_1, r_2, \ldots, r_s\}$ of this schema. Note that only the essential data structures for each function are provided as parameters. Parameters, such as m (Bloom filter size), k (hash-function count), or $strategy$ ({*dep2refs*, *ref2deps*}) are provided as global state from outside the main function MANY(D, d). Supporting data structures, such as the mapping from column identifiers to global indices (*colIndicesMap*), are also globally accessible.

Line 1 initializes MANY for the given schema D: It computes a mapping of column identifiers to their later global column index in the bit-signature matrix and stores this in the global variable *colIndicesMap*. Algorithm 2 shows the implementation of BUILDCOLINDICESMAP. In Line 4, the *Signature-Generator* computes the signature for each input column and puts the result in the corresponding place inside the *sigMatrix*. Based on this matrix, the *IND-Proposer* then computes the complete set of IND candidates. If the *IND-Validator* approves a candidate in Line 7, the IND is added to the output set. After testing all candidates, MANY outputs the complete set of valid INDs *uinds*.

The BUILDCOLINDICESMAP procedure in Algorithm 2 takes the database schema D as input and first initializes the later output *colIndicesMap* with an empty bidirectionally accessible map. This map stores a mapping from column identifiers to global column indices in the form of key-value pairs, such as $(R_x.A_y, z)$. Because it is a bidirectional map, it can be queried in both directions, either for a column index of a given column identifier or vice versa. The function mainly consists of a nested `for` loop in Lines 4 and 5: The outer loop iterates over the set of relational schemata and the inner loops over each attribute A_j of such a schema R_i, saving A_j into the map together with the current global counter *globalColumnId*, which is continuously incremented.

ALGORITHM 2: Build Column to Indices Mapping

```
1: function BUILDCOLINDICESMAP(D)
2:     colIndicesMap ← [R_x.A_y : z ∈ ℕ+]                ▷ empty bidirectional map
3:     globalColumnId ← 0
4:     for all R_i ∈ D do
5:         for all A_j ∈ R_i do
6:             colIndicesMap[R_i.A_j] ← globalColumnId
7:             globalColumnId ← globalColumnId + 1
8:     return colIndicesMap
```

The *Signature-Generator* is described in more detail in Algorithm 3. It takes both the database schema D and the instance d as inputs and initializes an empty list of Bloom filters as a starting point. Afterward, in Lines 3–7, the algorithm iterates over all columns in the dataset with the help of the attribute set stored in the key set of the *colIndicesMap*. For each column, a new Bloom filter with size m and hash-function count k is created. Subsequently, each value of the projection of r_i on the current attribute A_j is added to the Bloom filter signature. The Bloom filter is finally stored in the *bloomFilters* list at the corresponding global *columnId* position.

ALGORITHM 3: *Signature-Generator*

```
1:  function GENERATESIGNATURES(D, d)
2:      bloomFilters ← [colIndicesMap.length]
3:      for all R_i.A_j ∈ columnIndices.keys do
4:          bFilter ← BloomFilter(m, k)
5:          for all value ∈ π_{A_j}(r_i) do
6:              add value to bFilter
7:          bloomFilter[colIndicesMap[R_i.A_j]] ← bFilter
8:      sigMatrix ← [m][colIndicesMap.length]
9:      for all bFilter_i ∈ bloomFilters do
10:         for all bit_j ∈ bFilter_i do
11:             sigMatrix[j][i] ← bit_j
12:     return sigMatrix
```

Because we have only a list of decoupled Bloom filters after that step, we merge them together in the *sigMatrix*. Hence, by looping over the bits of each Bloom filter's backing bit-array in Lines 9–11, the ith Bloom filter's jth bit is written to the ith column at the jth row of the bit-signature matrix. Finnally, the filled matrix is returned.

The *sigMatrix* is the foundation for the computations of the *IND-Proposer* (Algorithm 4), which loops over all columns by their global column indices in the value set of the *colIndicesMap*. Depending on the chosen *strategy*, the *IND-Proposer* applies either *dep2refs* or *ref2deps* on each column. In both cases, a candidate subset is returned by DEP2REFS or REF2DEPS that is combined with the current candidate set. Finally, the *candidateSet* is returned.

In the introduction of this section, we already discussed the theory behind both strategies for IND candidate generation. We now give more detailed explanations based on the pseudo-codes in Algorithms 5 and 6. Both shown functions have the same signature: They take the index of a current column and the signature matrix as input and return IND candidate sets, where the

ALGORITHM 4: *IND-Proposer*

```
1: function generateCandidates(sigMatrix)
2:     cand ← ∅
3:     for all colIdx ∈ colIndicesMap.values do
4:         if strategy = dep2refs then
5:             cand ← cand ∪ dep2Refs(colIdx, sigMatrix)
6:         else if strategy = ref2deps then
7:             cand ← cand ∪ ref2Deps(colIdx, sigMatrix)
8:     return candidateSet
```

currently considered column is either the dependent attribute (*dep2refs*) or the referenced attribute (*ref2deps*).

ALGORITHM 5: *dep2refs* Strategy

```
Require: Index of the dependent candidate column colIdx, Bit-signature matrix sigMatrix
Ensure: Set of IND candidates candidateSet = {R_b.A_c ⊆ R_d.A_e, ...}
 1: function dep2Refs(colIdx, sigMatrix)
 2:     referencedCandidates ← ∅
 3:     candidateBitArray ← [colIndicesMap.length]
 4:     for all row_j ∈ sigMatrix do
 5:         if row_j[colIdx] = 1 then
 6:             if candidateBitArray = [] then
 7:                 candidateBitArray ← row_j
 8:             else
 9:                 candidateBitArray ← candidateBitArray ∧ row_j
10:     dep ← colIndicesMap[colIdx]
11:     for all bit_k ∈ candidateBitArray do
12:         if bit_k = 1 and k ≠ colIdx then
13:             ref ← columnIndices[k]
14:             referencedCandidates ← referencedCandidates ∪ {dep ⊆ ref}
15:     return referencedCandidates
```

The first strategy is shown in Algorithm 5. The first two lines of function DEP2REFS initialize the *referencedCandidateSet* and temporarily set the *candidateBitArray* to an empty bit-array. Afterward, we iterate over all rows in the input matrix, considering only those rows where row_j has the bit at position *colIdx* set to 1. Figure 4 already explained the reason for that. The first matching row is copied as the initial value for our result bit-vector *candidateBitArray*. It is combined with an and with every following matching row in the matrix. The final state of this bit-vector represents the potentially referenced attributes in D. For simplicity, we store the column identifier for the currently tested column in the variable *dep*. We finally iterate over all bits in our result vector that are set to 1 and add a corresponding IND candidate to the candidates found so far. The second condition in the if-clause of Line 12 ensures that IND candidate $R_x.X_y \subseteq R_x.X_y$, which are trivial due to the reflexivity of INDs, are not proposed.

The strategy *ref2deps* represented in Algorithm 6 works analogously to *dep2refs*. According to the explanation in Figure 4, there are only three differences: We consider rows row_j with a 0 at

ALGORITHM 6: *ref2deps* Strategy

Require: Index of the referenced candidate column *colIdx*, Bit-signature matrix *sigMatrix*
Ensure: Set of IND candidates $candidateSet = \{R_b.A_c \subseteq R_d.A_e, \ldots\}$

```
 1: function REF2DEPS(colIdx, sigMatrix)
 2:     dependentCandidates ← ∅
 3:     candidateBitArray ← [colIndicesMap.length]
 4:     for all row_j ∈ sigMatrix do
 5:         if row_j[colIdx] = 0 then                        ▷ ref2deps checks for 0
 6:             if candidateBitArray = [] then
 7:                 candidateBitArray ← ¬(row_j)             ▷ ref2deps links with inverted row_j
 8:             else
 9:                 candidateBitArray ← candidateBitArray ∧ ¬(row_j)
10:     ref ← colIndicesMap[colIdx]
11:     for all bit_k ∈ candidateBitArray do
12:         if bit_k = 1 and k ≠ colIdx then
13:             dep ← columnIndices[k]
14:             referencedCandidates ← referencedCandidates ∪ {dep ⊆ ref}
15:     return dependentCandidates
```

position *colIdx* (Line 5), we take the inverse of the rows (i.e., $\neg(row_j)$) before combining them with logical and operations (Lines 7 and 9), and the order in building candidates is reversed in the sense that the currently evaluated column is the referenced column and the algorithm finds possibly dependent columns (Lines 10 and 13).

The *IND-Validator* finally checks the validity of each IND candidate as outlined in Algorithm 7: At first, the function stores the value sets of the dependent and referenced attribute by retrieving the corresponding projections for r_k and r_j, respectively. The actual check happens in Line 4, where the subset relation of the dependent value set is checked against the value set of the referenced column candidate.

ALGORITHM 7: *IND-Validator*

Require: IND candidate $R_b.A_c \subseteq R_d.A_e$, instance of database schema $d = \mathcal{I}(D) = \{r_1, r_2, \ldots, r_n\}$
Ensure: Validity of IND candidate

```
1: function ISVALID(candidate)
2:     dependentSet ← π_{A_c}(r_b)
3:     referencedSet ← π_{A_e}(r_d)
4:     return dependentSet ⊆ referencedSet
```

Instead of recomputing the projections for each and every validation, the projections are actually computed only once and then stored and retrieved from a cache. Because most attribute value sets are required for different checks, recurring I/O operations for the same values can in this way be avoided. The value sets in the cache are maintained using a *Least Recently Used (LRU)* strategy, so that values can be removed from memory if memory is exhausted. To avoid multiple reads of the data, this cache is filled initially by the *Signature-Generator* when this component reads the single values for insertion into the Bloom filter signatures. If the entire dataset fits into main memory, which in fact was possible for almost all our evaluated datasets on our test machine, no data have to be read again during validation. Otherwise, the cache is filled with a best-effort strategy to a certain

memory level that still allows further computations. If during the *IND-Validator* computation phase a value set is not present inside the cache, the value set is read from disk and stored in the cache. The *LRU* cache ensures that frequently needed value sets (e.g., those of frequently referenced attributes) are kept in memory as long as possible.

When the *IND-Validator* finishes the last candidate from the set of candidates provided by the *IND-Proposer*, the set of valid inclusion dependencies is complete and can be returned by the main function.

4.3 Multi-Hashing

The number of false positive candidates generated by the *IND-Proposer* is mainly influenced by the parameters that are chosen for the Bloom filter. Consider a relatively small Bloom filter (small m) to which we add many values. In the worst case, every bit is set in the signature. Then, every other signature is a subset of this particular "degenerated" Bloom filter, leading to many false-positive candidates. For that reason, a greater length of the underlying bit-array (large m) should be used for the Bloom filter signature. As a drawback, longer Bloom filters result in a larger signature matrix and, in turn, to more logical operations. Furthermore, by choosing a larger k, the higher number of logical conjunctions might decelerate the runtime of the Many algorithm. Obviously, there is a trade-off between the false-positive rate and Bloom filter length together with hash-function count.

We discuss the correct choice of m and k, the Bloom filter length and hash-function count, in the evaluation. Here, we introduce an idea to improve Bloom filter performance while keeping the false-positive rate small. The main idea is to initially create *multiple* smaller bit-signatures for the columns and then combine these signatures in the signature matrix. In this way, we enforce a more uniform distribution of set bits in the combined signatures, which on average allows Many's *IND-Proposer* to prune columns much earlier during bitset-intersections.

We call the method of using multiple Bloom filters *multi-hashing*. For its implementation, we introduce a new parameter p (passes) in the Many algorithm. For $p > 1$, multiple Bloom filter signatures with different hash-function sets are generated per column. The signatures of these Bloom filters can simply be concatenated and treated as one before they are merged into the signature matrix by the *Signature-Generator*. The algorithm that we discussed so far virtually had p set to 1. Thus, the resulting signature matrix had a size of $(m \cdot n)$ bits, with n being the number of attributes in the input and m the Bloom filter size. Through the introduction of p, the resulting matrix is $((p \cdot m) \cdot n)$ bits large.

The technique of concatenating the p signatures into one works because the rows in the signature matrix are logically linked by `and` operations. Also, introducing the parameter needs only a slight change in Many, where an additional `for`-loop is introduced around Lines 3–7 in Algorithm 3.

4.4 Parallelization

No previous ind detection approach utilizes the parallelization capabilities of modern hardware. The two main challenges in the parallelization of algorithms is the communication overhead between the parallel tasks and the locking due to synchronization of concurrent write accesses to shared data while concurrent read operations are lock-free. Indeed, the Many algorithm already contains an inherent parallelization: The bit-wise `and` operations of methods dep2Refs and ref2Deps in Algorithms 5 and 6 check the Bloom filter containment of a particular attribute against all other attributes at the same time. In fact, depending on the underlying hardware architecture (e.g., 64-bit register width) and the used implementation strategy for the bit-vectors (e.g.,

`long`-array based), this means that for one set bit in the currently checked Bloom filter it is probed against 64 other attributes' Bloom filters simultaneously.

But, so far, only one CPU core out of the possibly many available is used. Notably, all operations on the bit-signature matrix during the *IND-Proposer* phase are read-accesses only. Instead of letting one thread sequentially check each column in the matrix, we can divide the workload and let multiple threads check subsets of matrix columns. The necessary algorithm modifications are shown in the following listings starting with the main function shown in Algorithm 8.

ALGORITHM 8: Parallel MANY

Require: ⟨see Algorithm 1⟩, number of available CPUs n
Ensure: All valid INDS *uinds*

1: $colIndicesMap$ ←BUILDCOLINDICESMAP(D)
2: **function** MANYPARALLEL(D, d)
3: $uinds \leftarrow \emptyset$
4: $sigMatrix \leftarrow$ GENERATESIGNATURES(D, d)
5: **for all** $i \in \{1, \ldots, n\}$ **do**
6: $lo \leftarrow \lceil (i-1) * (colIndicesMap.length/n) \rceil$
7: $up \leftarrow \lfloor i * (colIndicesMap.length/n) \rfloor$
8: $uinds \leftarrow uinds \cup$ INDDETECTIONWORKER$_i$($sigMatrix, lo, up$)
9: **return** *uinds*
10: **return** MANYPARALLEL(D, d)

The main difference to the sequential version of the algorithm is that the generation, iteration, and validation of IND candidates was moved into *IND-Detection-Workers*. The algorithm now splits the candidate space evenly into n distinct parts and distributes these parts to n worker threads via *asynchronous* calls to the *IND DETECTION WORKER* function. Each *IND Detection Worker* produces a validated subset of the final complete set of INDS. Hence, the only blocking synchronization point during the computation is the union of the intermediate result subsets in Line 8 after a worker terminates. For the rest of the time the worker threads run independently.

The *IND-Detection-Worker*, which is shown in Algorithm 9, combines the functionality of the *IND-Proposer* and *IND-Validator* components. We decided to integrate the validation of IND candidates inside the multiple threads because it is also easily parallelizable. Inside the outer `for` loop of Line 3, a worker loops over all assigned column indices and executes the candidate-generating function depending on the chosen strategy in Line 5 or 9. Each generated candidate is then immediately checked and added to the subset of the final result if the validation succeeds.

Now everything but the initialization and the bit-signature generation is parallelized. A parallelization of the latter does not make sense because it is mainly I/O bound due to reading all values from disk.

4.5 Indexed Bit-Vectors

One of the main strengths of the discussed IND detection approaches is the ability to avoid unnecessary checks during the computation. Checks can become unnecessary if one column includes a value that occurs in no other column. Therefore, this column cannot depend on any other column and thus can be disregarded by further processing. We now discuss how to implement such a pruning technique by changing the implementation of the bit-arrays that are used by the Bloom filters and the bit-signature matrix.

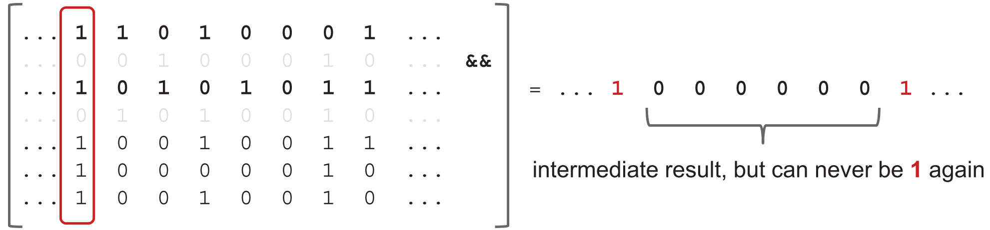


Fig. 6. After combining two matrix rows, a consecutive part of the intermediate result is already cleared (0).

To check a column for containment in other columns with Many (*dep2refs* strategy), we check which columns' bit-signatures have the same bits set as that column. We do this check by logically linking all rows in the signature matrix with a conjunction where the concerned signature's bit is set to `1`. If this operation changes a bit from `1` to `0` in the intermediate result vector, the column that corresponds to the index of that cleared bit *cannot* be a superset of the currently checked matrix column. This is because the `0` indicates that a bit is set in the signature of the currently checked column that was not present in the signature of the column with the cleared bit in the intermediate result.

The Many algorithm avoids checking bit-signatures for subset relationships pairwise. Instead, it actually checks X columns at once, where X depends on the underlying hardware architecture, which is 64 in our case. This parallel check prevents us from omitting certain single columns (i.e., columns whose bit in the intermediate result vector became `0` during the inclusion-checking process). However, our focus on datasets with many tables and, in particular, Web table datasets, comes to the rescue: The often millions of columns make the lengths of the signature matrix rows and the (intermediate) result vectors accordingly large. For this reason, we represent the bit-vectors for the matrix rows and result vectors as multiple concatenated `long` values. We further observe that the row and result vectors are mostly extremely sparse in terms of bits set to 1 because the input tables are typically short and do not have much overlap. For this reason, not only single bits in the bit-vectors are `0`, but also often long consecutive sequences. An example for this situation is illustrated in Figure 6.

ALGORITHM 9: IND-Detection-Worker

Require: Bit-signature matrix *sigMatrix*, lower l and upper u attribute index limits
Ensure: Subset of all valid INDs $uinds_i$

1: **function** INDDetectionWorker$_i$($sigMatrix, l, u$)
2: $\quad uinds_i \leftarrow \emptyset$
3: $\quad$ **for all** $colIdx \in \{l, \ldots, u\}$ **do**
4: $\quad\quad$ **if** $strategy = dep2refs$ **then**
5: $\quad\quad\quad$ **for all** $candidate \in$ dep2Refs($colIdx, sigMatrix$) **do**
6: $\quad\quad\quad\quad$ **if** isValid($candidate$) **then**
7: $\quad\quad\quad\quad\quad uinds_i \leftarrow uinds_i \cup \{candidate\}$
8: $\quad\quad$ **else if** $strategy = ref2deps$ **then**
9: $\quad\quad\quad$ **for all** $candidate \in$ ref2Deps($colIdx, sigMatrix$) **do**
10: $\quad\quad\quad\quad$ **if** isValid($candidate$) **then**
11: $\quad\quad\quad\quad\quad uinds_i \leftarrow uinds_i \cup \{candidate\}$
12: $\quad$ **return** $uinds_i$

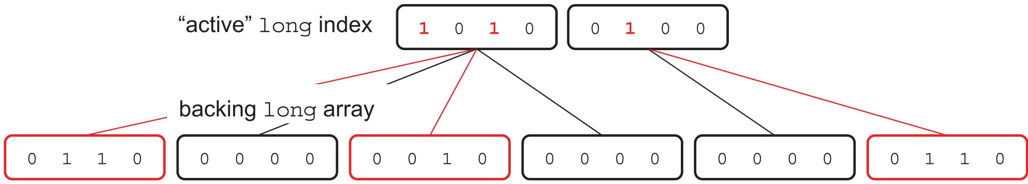


Fig. 7. Visualization of indexed bit-vectors. In this example, long values have a length of 4 bits.

Now, if certain long values are completely covered by a 0-sequence (i.e., all their bits are 0), then the intersection algorithm can discard them from all following checks for a currently considered column. Logically linking these empty long values by a conjunction with any other value cannot change them. Nevertheless, a naïve bit-vector implementation spends CPU cycles on these zeroes.

The main idea to implement the pruning of empty long values is to introduce an index over the long-array vectors. In this index, we keep only those long values that are not equal to 0. We call these values *"active"* in the following. The intersection process uses this index to link only those parts of the result vector with corresponding parts from the signature matrix that are still of interest. The idea is sketched in Figure 7.

To implement the index, we need to extend the naïve bit-vectors. First, we store which long values are still active using another simple bit-vector. Second, we extend the bit-vector methods that influence the state of the vector. For example, if a bit in the vector is set to 1 inside an area marked as inactive, this area must also be changed to active in the index. Furthermore, and most importantly, the and operation must be changed: So far, the method iterated over all long values in the backing array and logically linked them by a conjunction with the corresponding value of the operand; now, the method searches for the next set bit in the index and performs the conjunction only for active values.[3]

4.6 Applying Filters

Because the IND result sets can grow quadratically with the number of input columns, it is essential to reduce the output set early on in the candidate detection and the validation phase. In this section, we show when and how we apply the filters introduced in Section 3.2.

The filters (1) *filterNullColumns* and (2) *filterIntegerColumns* affect referenced candidates as well as dependent attribute candidates. Therefore, if attributes are found that match these filters, it is unnecessary to further check them. The filter (3) *filterNonUniqueRefCandidates*, in contrast, affects only the referenced attribute of a candidate so that we can prune candidates with this rule but not (directly) entire attributes. Filter (4) *filterRefLowCoverage* measures the value overlap of dependent and referenced attributes. This is a costly operation that is useful to reduce the result size but not to improve the discovery performance. Therefore, we implement it as a post-processing filter after a candidate has been validated.

Algorithm 10 shows the changes made to Algorithm 8. The only difference is prior to the actual algorithm's function ManyParallel(D, d). For the filter *filterRefLowCoverage*, a new parameter τ is introduced as the minimal fraction of coverage on the referenced attribute's values. Furthermore, we introduce two new bit-arrays with a length of the number of all attributes from the input (Line 2). Those two arrays indicate if certain attributes were filtered. In the first one, *filteredCols*, a bit is set if an attribute is completely excluded from further consideration. For that reason, the results of the *filterNullColumns* and *filterIntegerColumns* filter rules are stored together inside this

[3]Modern x86 architectures offer special hardware instructions to efficiently find the index of the next set bit.

variable. In *filteredRefs*, we mark only attributes that can still reference others but must no longer be considered as possible referenced attributes.

ALGORITHM 10: Filtering ManyParallel

```
Require: ⟨same as in Algorithm 8⟩, minimum coverage τ ∈ [0, 1]
Ensure: All valid INDs uinds
 1: colIndicesMap ←Initialize(D)
 2: filteredCols, filteredRefs ← [colIndicesMap.length]
 3: filteredCols ←filterNullColumns(D, d)
 4: filteredCols ← filteredCols ∨ filterIntegerColumns(D, d)
 5: filteredRefs ←filterNonUniqueRefCandidates(D, d)
 6: function ManyParallel(D, d)
    ... see Algorithm 8
```

For all filters, changes are also made to the INDDetectionWorker, as shown in Algorithm 11. We still have one outer loop iterating over all assigned columns in the bit-signature matrix. If such a column is marked in the *filteredCols* array, we proceed to the next one. For strategy *ref2deps*, the algorithm then checks if the column is still a possible referenced candidate. As a last step, we modify the isValid function: After validating a candidate, the function checks the overlap of the two value sets, and, with the minimum coverage parameter τ, it decides whether to keep or prune the current inclusion dependency. More concretely, if $\frac{|dependentSet|}{|referencedSet|} \geq \tau$, then the attribute pair is kept.

ALGORITHM 11: Filtering INDDetectionWorker

```
Require: Bit-signature matrix sigMatrix, lower l and upper u attribute index limits, minimum
    coverage τ
Ensure: Subset of all valid INDs uinds_i
 1: function INDDetectionWorker_i(sigMatrix, l, u, τ) {
 2: uinds_i ← ∅
 3: for all colIdx ∈ {l, . . . , u} do
 4:     if filteredCols[colIdx] = 1 then
 5:         continue
 6:     if strategy = ref2deps then
 7:         if filteredRefs[colIdx] = 0 then
 8:             continue
 9:         for all cand ∈ Ref2Deps(colIdx, sigMatrix) do
10:             if isValid(cand, τ) then
11:                 uinds_i ← uinds_i ∪ {cand}
12:     else
13:         analogously to the code above
14: return uinds_i }
```

5 EVALUATION

We first assess Many's behavior with respect to parameters and different input datasets to find an optimal parametrization. Then, we show how parallelization and bit-vector indexing influence

Table 4. Datasets Used in the Evaluation of Many

Dataset	# tables	# attr.	Ø attr.	Ø tuples
WikiTables	1,398,105	7,470,443	5.34	10.85
WDCWebTables1	528,106	1,849,065	3.5	12.27
PlistaStatistics	1	63	63	101,305,267
BTCFreebase	11,118	22,236	2	9,105

Table 5. Runtime of Many and the Number of INDs on Several Real-world Datasets

Dataset	# INDs	runtime in seconds
WikiTables	5,056,820∗	7,867
WDCWebTables1	243,224,068∗	10,438
PlistaStatistics	921	`OutOfMemoryError`
BTCFreebase	202,331	523

the algorithm's efficiency. Finally, we compare our approach to existing inclusion dependency detection approaches on different real-world datasets.

5.1 Test Environment and Datasets

We executed the experiments on a Dell Poweredge R620 with two Intel Xeon E5-2650 and CentOS 6.4. The Java execution environment is the 64-bit OpenJDK in version 1.7.0_65. The Java virtual machine is granted 120GB of total 128GB main memory for heap space allocations.

All tested algorithms are implemented on the publicly available *Metanome* framework (Papenbrock et al. 2015a) with its implementations of the Spider and Binder algorithms. Unless stated otherwise, Many's filters are not activated. The measuring of the runtime was done using Java's `System.nanoTime()` functionality. We conducted three runs of each experiment with empty caches and always report the minimal runtimes to remove noise.

The collection of our four evaluated real-world datasets can be found in Table 4 alongside some basic statistics. The datasets were chosen for their quite diverse shapes.

The datasets *WikiTables* and *WDCWebTables1* both contain a huge collection of Web tables. The former is provided by the *WikiTable* project in form of an *SQL* dump,[4] which we converted to separate csv-files. *WDCWebTables1* is a subset of tables from the *WDC WebTable*[5] project containing approximately half a million relational instances as csv-files; here, we used *part1-5000*.[6] From the *Plista* dataset (Kille et al. 2013), which is a collection of streamed anonymized Web log data, we used only the Statistics relation with more than 100 million tuples. The *BTCFreebase*[7] dataset is a collection of RDF triples in relational representation. Each relation in *BTCFreebase* corresponds to an RDF predicate and has two attributes: *subject* and *object*. Because some predicates are used more frequently than others, the number of rows of the relations varies greatly. While nearly 11,000 of the instances contain only 10,000 tuples or fewer, the 18 largest instances contain between 1 and 10 million tuples.

Table 5 summarizes the runtimes of the Many algorithm on the complete datasets. It also shows the actual number of INDs. The counts marked with an asterisk (∗) are the numbers when all filters are activated. For the *PlistaStatistics* dataset, no final runtime result could be determined due to

[4] http://downey-n1.cs.northwestern.edu/public/ *Accessed: 05/22/2017.*
[5] http://webdatacommons.org/webtables *Accessed: 05/22/2017.*
[6] http://data.dws.informatik.uni-mannheim.de/webtables/2014-02/englishCorpus *Accessed: 05/22/2017.*
[7] http://km.aifb.kit.edu/projects/btc-2012 *Accessed: 05/22/2017.*

	WikiTables				WDCWebTables1			
rank	m	k	p	runtime in s	m	k	p	runtime in s
1	650	6	2	32.14	500	4	1	48.13
2	200	7	3	32.29	550	11	3	48.28
3	1300	15	4	32.39	500	4	2	49.67
4	300	9	1	32.62	750	2	1	50.28
5	300	1	3	32.68	800	8	1	50.55
6	1050	3	1	32.74	850	13	2	50.76
7	850	3	3	32.90	650	5	1	51.09
8	650	6	1	32.99	750	4	1	51.35
9	1400	10	4	33.07	650	13	2	51.61
10	1450	8	2	33.10	900	10	1	51.62
...	...	...	...	...	...	...	...	...
2246	50	9	1	76.12	50	5	1	104.35
2247	50	15	1	76.61	50	13	1	107.39
2248	50	7	1	79.56	50	14	1	107.99
2249	50	1	1	80.08	50	12	2	117.91
2250	50	13	1	82.15	50	10	1	118.02

Fig. 8. Ranked runtimes of MANY on subsets of 10,000 tables from *WikiTables* and *WDCWebTables1* with different parameter combinations ($m \in \{50, 100, \ldots, 1500\}, k \in \{1, 2, \ldots, 15\}, p \in \{1, 2, \ldots, 5\}$, red=large, yellow=medium, green=small).

the size of the dataset, which exceeded the available main memory. Here, the number of INDs was determined with the BINDER algorithm of Papenbrock et al. (2015b), which is specialized for this use-case.

5.2 Parametrization of MANY

The MANY algorithm takes three different configuration parameters as its input: the size of the Bloom filter m, the number of hash-functions k, and the number of passes p. As candidate enumeration strategy, we use the *dep2refs* strategy if not stated differently. We set the number of worker threads to 16 because this provides the lowest runtimes, as shown later. Now, we evaluate the influence of the three parameters and their interdependence to find an optimal parametrization and explore MANY's sensitivity.

In general, we expect a trade-off between the configurations. Larger Bloom filters may produce more unique bit-signatures for single columns and might, therefore, decrease the false-positive rate. A low false-positive rate reduces the number of unnecessary IND checks and, hence, the runtime spent for the validation. However, large m and k result in a large bit-matrix, resulting in a longer runtime for logical linkage of bit-signature matrix rows.

We ran experiments on two different subsets of the *WikiTables* and *WDCWebTables1* datasets with a sample size of 10,000 tables to achieve runtimes that are short enough to test *all* 2,250 parameter combinations. This sample size also guarantees runtimes with significantly measurable differences. We choose m from $\{50, 100, \ldots, 1500\}$, k from $\{1, 2, \ldots, 15\}$, and p from $\{1, 2, \ldots, 5\}$.

Figure 8 shows the 10 best and 5 worst configurations for both datasets, ranked by the overall runtime. The parameters are colored according to their relative value within the chosen range. To gain a better understanding of the distribution of runtimes, we also plotted the runtimes in ascending order in Figure 9.

The results for both datasets are similar: The top 10 configurations lie closely together regarding their runtime. None of the three parameters has an obviously high or low setting, which mean that fast runtimes can be achieved with both small and large parameter values. It is particularly surprising that also huge Bloom filters and multiple passes can result in fast overall computation times.

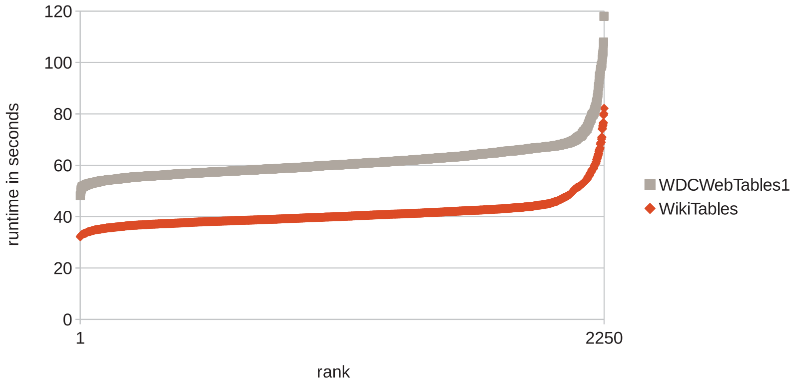


Fig. 9. Distribution of runtimes of different parametrizations in ascending order.

In particular, Figure 9 shows an overall low sensitivity for the settings of all three parameters—it is unlikely to choose a combination that performs particularly poorly.

We also observe that the use of multi-hashing is not always outperforming configurations with only one bit-signature per input column, that is, the top 10 results for *WDCWebTables1* contain solely parametrizations with $p \leq 3$, and six times only was one pass applied. However, almost all worst configuration settings only use $p = 1$ and the smallest value for m. With this observation, we might not be able to guess the best parametrization for Many in advance, but we can easily avoid the worst runtimes shown in Figure 9 by setting $m > 100$ and $p > 1$. For the following evaluations, we chose for each dataset the best available parametrization.

5.3 Scale-up Evaluation

Many is the first inclusion dependency detection algorithm that uses multiple CPU cores through parallelization. To evaluate the parallelization, we ran a scalability experiment on the complete *WikiTables* dataset by varying Many's parameter n that specifies the number of *"IND Detection Workers."* We choose 32 as an upper limit for n because it equals the number of logical cores in our machine (16 physical cores). We also activated all filters described in Section 3.2 in this experiment to keep the result size manageable. The effect of these filters is discussed later.

Figure 10 illustrates the runtime results for the scale-up experiment: The top line in the chart is the overall runtime, and the lower line represents the runtime of the serial (nonparallelizable) preprocessing part, namely, the population of the Bloom filters and their merge into the signature matrix. Compared to the overall runtime, the serial part is negligible (approximately 10 seconds in all runs).

As we expected, the graph shows a near linear speed-up in the number of worker threads for one to four threads because the individual worker threads are lock-free and need synchronization only at the end of their computation life cycle, when their results are written to the set of all INDs. This synchronization in the end, together with an increased overhead for thread creation and management, reduces the performance gains for thread numbers between four and eight. Still, Many gets faster here and achieves the best runtime with 16 threads. At this point, the overall speed-up factor is approximately 6.5 (instead of the optimal speed-up of 16). With more than 16 threads, Many spawns more worker threads than physical cores exist, and the algorithms performance decreases. Because each thread makes heavy use of physical instruction units for the bit-operations, the reason for the performance loss is probably that these instructions do not work well with hyper-threading.

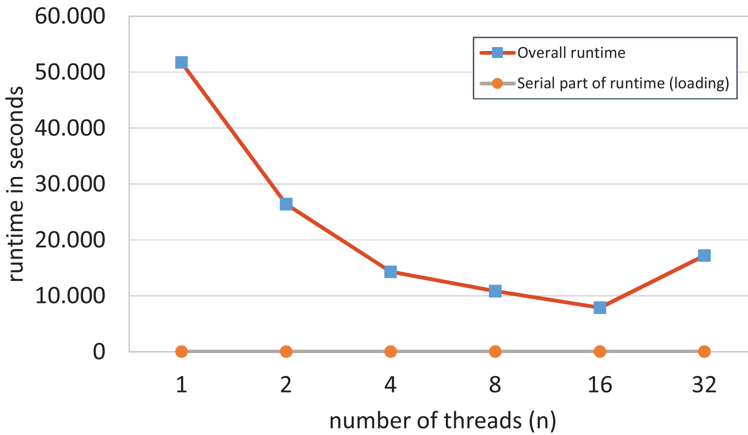


Fig. 10. Scale-up behavior of MANY on *WikiTables* with $m = 650$, $k = 6$, $p = 2$, *strategy* = *dep2refs*, and all filters.

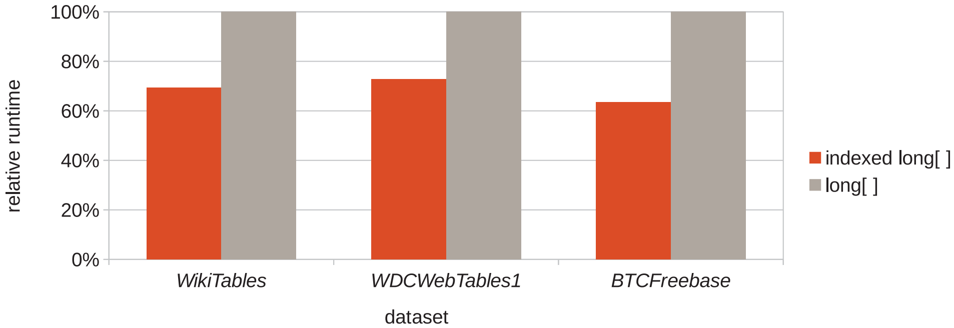


Fig. 11. The comparison of indexed `long` arrays and simple `long` arrays used as bit-vector implementations in MANY.

In summary, we observed that MANY scales well with the number of threads and utilizes the provided cores efficiently. The speed-up is sublinear due to thread synchronization overhead and can be achieved only up to a thread count that equals the number of physical CPU cores on a machine.

5.4 Evaluation of Indexed Bit-Vectors

In Section 4.5, we introduced an idea to avoid unnecessary intersect operations in the IND candidate enumeration phase by indexing the bit-vector implementations. The following experiment compares the runtimes using simple `long` array-backed bit-vectors with the runtimes using our indexed bit-vectors. As test sets, we take the full *WikiTables* and *WDCWebTables1* datasets with activated filters and the *BTCFreebase* dataset without any filters (because result filtering is due to the much smaller number of results technically not necessary here). For the parameters, we chose the ones with the best runtimes from Section 5.2 and for *BTCFreebase,* the same as in the previous section.

The results of this experiment are listed in Figure 11: For all three datasets, the indexed version of the bit-vectors leads to better runtime results. The achieved speed-ups vary only slightly between the datasets (i.e., the runtimes are about 30% to 40% lower).

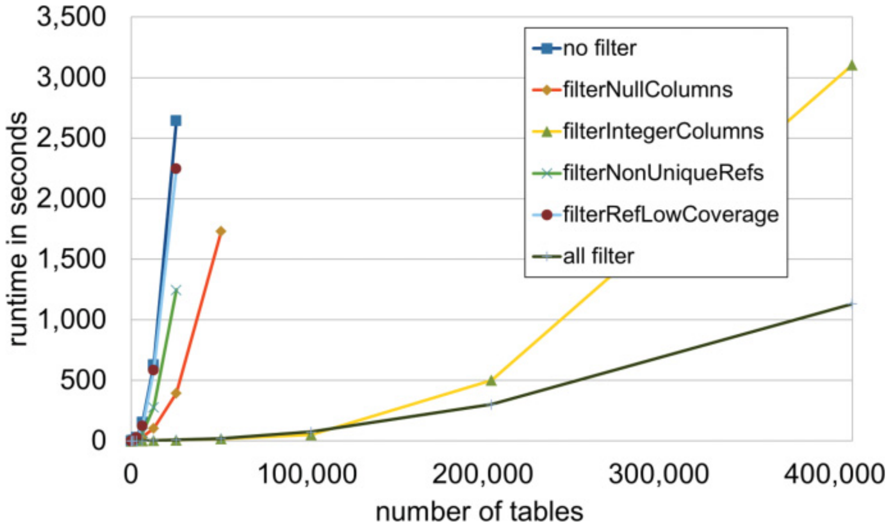


Fig. 12. Runtime of Many with different filters on different *WikiTables* subsets ($m = 650, k = 6, p = 2$).

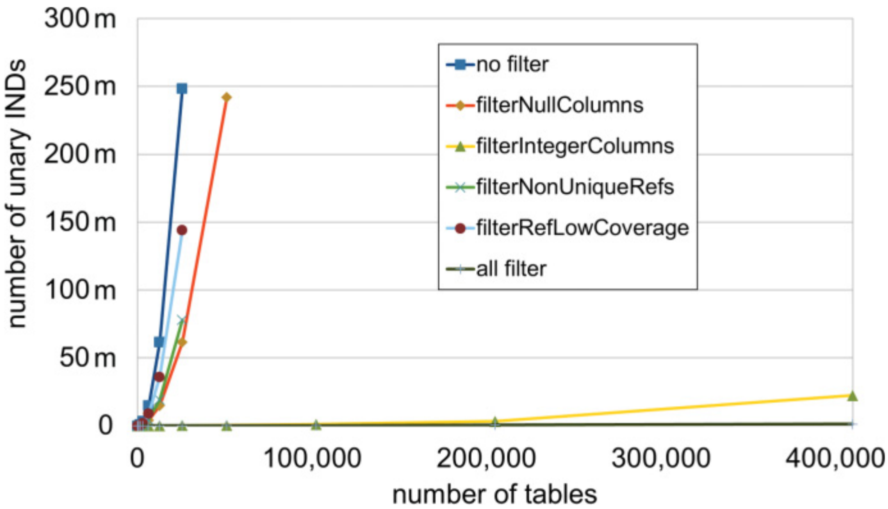


Fig. 13. INDs that Many detects with different filters on different *WikiTables* subsets ($m = 650, k = 6, p = 2$).

5.5 Efficiency Effects of Filtering

In Section 3.2, we discussed that the number of INDs grows quadratically with the number of columns in the input tables. Thus, we suggested different filters to decrease the number of INDs early on. In the following experiment, we investigate how the use of individual filters influences the number of discovered INDs and also how it influences the runtime. For this purpose, we ran the Many algorithm on different subsets of the *WikiTable* dataset. We increase the number of tables and measure the runtime for each filter individually, for no filters, and for all filters together. Figure 12 depicts the different runtime measurements, while Figure 13 shows the corresponding number of detected INDs. We aborted measurements that exceeded more than 1 hour of runtime.

The runtimes clearly correlate with the number of detected INDs. For all filters except for the *filterIntegerColumns* filter and the combination of all filters, we can observe a rapid increase of the runtime as well as an increase in the number of found dependencies. None of those filters alone discards enough INDs to keep the runtime reasonably small for more than 50,000 tables.

The *filterIntegerColumns* filter excludes so many columns from the input set from further consideration that it is able to detect INDs among more than 400,000 tables in under 1 hour. For smaller subsets, it even slightly outperforms the runtimes measured with all filters activated, due to their own overhead.

The most important observation from that experiment is the fact that a combination of all filters clearly performs best for a large number of tables. Even if most of the single filters do not significantly decrease the runtime, their combination, together with the most effective filter *filterIntegerColumns,* speeds up the computation. Also, they keep the result set at a reasonable size: Among the columns of the 409,600 tables, about 1.3 million valid INDs are detected when all filters are applied. So the combination of all filters is the only efficient way to detect all relevant INDs among hundreds of thousands of tables.

From this experiment, we learned that the number of valid INDs (without filtering) easily exceeds any reasonable number and therefore makes an efficient persistence in a human-readable format for interpretation impossible. The enormous increase can be avoided only by the exclusion of columns that merely contain integer numbers. A combination of all filters makes the Many algorithm capable of handling several hundreds of thousand Web tables.

5.6 Comparison to Known IND Algorithms

We compared Many to two known inclusion dependency detection algorithms, namely Binder and Spider. We ran experiments to examine how the three algorithms scale with respect to the number of tuples as well as with the number of attributes in the input. For a fair comparison, we disabled all filters in the Many algorithm in this experiment and found the same complete set of INDs with all three algorithms.

5.6.1 Scaling the Number of Attributes. We first discuss how the approaches scale with the overall number of attributes in the input. This is especially important for our use case because we want to detect inclusion dependencies among thousands of tables and, thus, an immense amount of attributes.

We start by providing each of the algorithms the same subset of 100 randomly chosen tables from the *WikiTables* dataset as their input. Then, we add more and more tables to the input, always keeping the tables from the previous run. Because of the random choice of tables and the nearly even distribution of attribute-counts among the tables, we increase the number of attributes fairly evenly.

The runtime results are plotted in Figure 14. As expected, the runtimes of all three algorithms grows quadratically because the candidate set grows quadratically as well, and all algorithms spend most of their time in checking these candidates. We see that the last measurement for Spider could be obtained for an input size of only 1,000 tables. For 2,000 and more tables, the execution aborted for too many open files. Two thousand relations contain approximately 10,680 attributes (which is about 5.34 attributes per relation), and, therefore, Spider exceeds the typical number of open file handles allowed per process in operating systems.

The Binder algorithm scales up to an input of several thousand tables. It aborts with an `OutOfMemoryError` for 4,000 tables because the generated candidate set, which Binder represents in bulky HashMaps, exceeded 90 GB of memory. For input sizes where all three algorithms terminated successfully, one can observe that Many is always the fastest algorithm, followed first

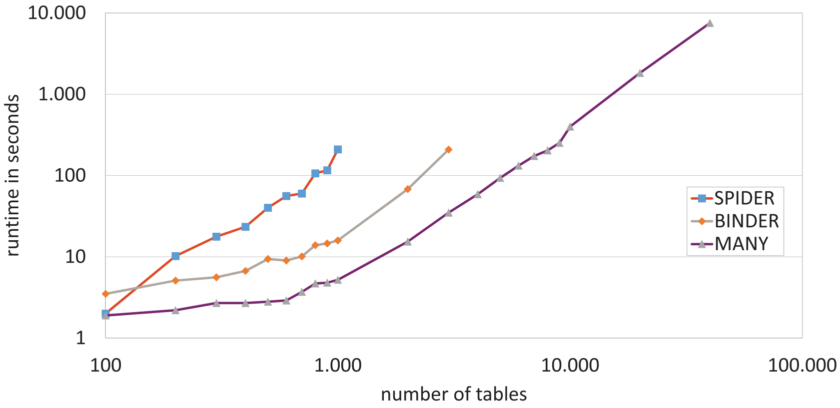


Fig. 14. Scaling over the number of tables/attributes in the input set. Parameters used for Many: $m = 650, k = 6, p = 2$.

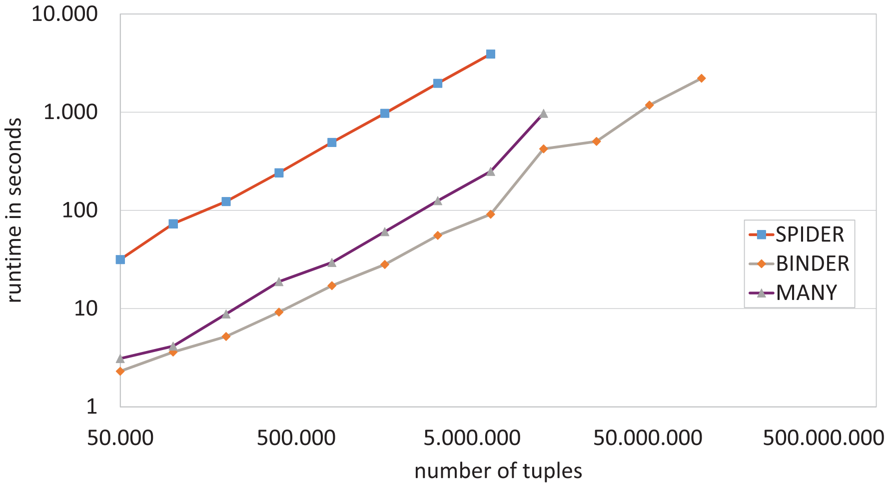


Fig. 15. Scaling over the number of tuples. Parameters used for Many: $m = 10,000, k = 4, p = 2$.

by Binder and then by Spider. In fact, it is up to 10 orders of magnitude faster than Binder. This advantage becomes even greater when more physical cores are available so that Many can spawn more than the currently used 16 worker threads. Furthermore, we disabled all filters for this experiment to compute the same results; enabling Many's filters would also further speed the algorithm up.

5.6.2 Scaling the Number of Tuples. We now evaluate Many's ability to scale with an increasing number of tuples in comparison to Spider and Binder using the *PlistaStatistics* dataset. In this experiment, we incrementally increase the number of tuples of a single relation by doubling the number of tuples from one measurement to the next. We always take the n first tuples from the relation. The complete relation includes 921 INDs, but the number of INDs varies slightly from one measurement to the next.

Figure 15 depicts the runtime results of the experiment. Because an increase in the number of tuples influences the validation, but not the candidate space, we observe that all three algorithms scale nearly linearly with the number of tuples. We stopped the experiment for Spider at 6.4 million

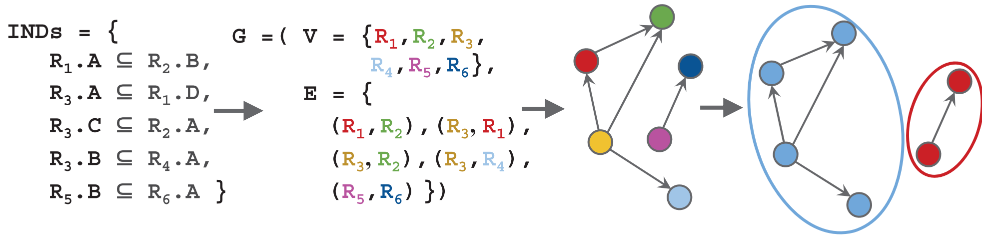


Fig. 16. Conversion of INDs into graph representation.

tuples because its runtime exceeded 3 hours, which was our time limit for this experiment. Both Many and Binder, then, show a small outlier between 6.4 and 12.8 million tuples because the internal structure of the data (e.g., number of distinct values and inclusion dependencies) changed and caused by chance more and longer validation operations.

Overall, Spider is more than one order of magnitude slower than Many because it comes with higher I/O cost due to the writing and reading of sorted value lists.

Binder also spends additional time on writing and reading bucket files, but its pruning strategies work most efficiently in a many rows experiment. For this reason, it outperforms the Many algorithm on very long datasets and scales to larger numbers of rows. Because Many does not implement memory management techniques like Binder does, it throws an `OutOfMemoryError` as soon as two single columns do not fit into the limited space of the validation cache. This is at about 100 million tuples for the *PlistaStatistics* dataset and 120GB of RAM. To handle such huge numbers of tuples, a more efficient caching strategy is needed that does not rely on keeping complete columns in memory for the validation. Note, however, that this extreme scenario (few very large tables) is not the primary goal of Many because it specializes on very many small tables.

We summarize that Many scales linearly with the number of tuples. It is considerably faster than Spider, but slightly slower than Binder. Because of the high memory capacities of modern machines and the use case focus on short tables, Many works completely in-memory, but further improvements in the caching strategy are needed if long datasets are to be analyzed.

6 VISUALIZING THE IND GRAPH

To explore the resulting INDs, we have developed a prototypical interactive graphical system of the graph of IND relationships among tables. The system first transforms the discovered INDs into a directed graph $G = (V, E)$. To keep semantically related attributes together, we choose tables as the most fine-grained elements (i.e., vertices V in the graphical representation). Figure 16 shows the transformation of a set of INDs into such a directed graph: For every relation participating in an IND, we add a vertex to V; for each IND, we add a directed edge from the vertex containing the dependent attribute to the vertex containing the referenced attribute. As our evaluated datasets contain millions of INDs, we further group them by their connected components.

Figure 17 illustrates our graphical tool to explore the resulting connected components and their constituent graphs, tables, and columns. Here, we visualize the complete set of filtered INDs from the *WikiTables* dataset. The circles to the left represent the identified connected components inside the table relationship graph. The sizes represent the number of included tables per component. The set of thousands of connected components is clearly dominated by a handful of outstanding huge components. Those huge components combine tens of thousands of tables. The user can drill into each of these components to see the actual join graphs. By selecting a vertex, she then sees the actual INDs that link this table to others.

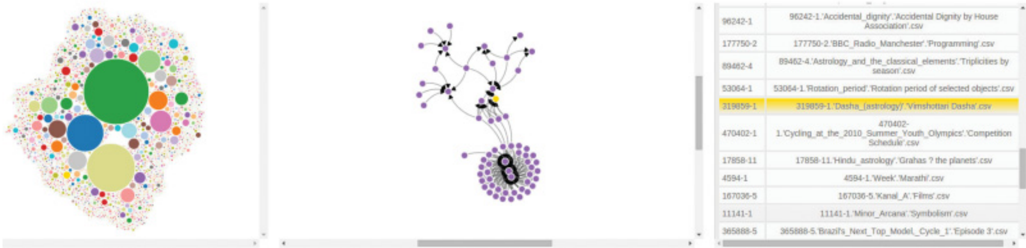


Fig. 17. User exploration of IND relationships between tables extracted from the entire Wikipedia. From left to right: All connected components (sized for number of tables); a connected component with tables as vertices; list of tables therein.

7 CONCLUSION

We addressed the problem of IND detection on more than a million tables that were extracted from Web pages. We provided use cases that motivate the detection of inclusion dependencies in such datasets, such as the exploration of related information, and we analyzed the limitations of existing approaches for IND detection on so many tables. We proposed the MANY algorithm, which overcomes these limitations by employing an intelligent candidate proposal strategy. MANY creates a space-efficient bit-signature for each column in the input dataset using Bloom filters. We showed that the subset relationships between columns in the dataset are retained by those signatures. Furthermore, a strategy to parallelize our approach and an optimization for sparse bit-vector intersection is introduced.

In our evaluation, we studied several aspects of our algorithm on different real-world datasets. We found that the sensitivity to parameters that concerns the Bloom filter size is surprisingly low. Furthermore, the algorithm scales well with the number of provided CPU cores. In comparison with known approaches, MANY shows especially good performance in the scale over the number of attributes.

Some issues and extensions are left for future work. First, we are working on a GPGPU-based implementation to achieve higher efficiency. Second, we plan to apply the full set of heuristics from Rostin et al. (2009) to classify INDs as foreign keys; in this context, we also evaluate the semantic value of INDs in other use cases, such as query optimization, data cleaning, and schema design, which have alternative notions of semantic relevance for INDs. Third, we plan to extend our approach to detect not only exact INDs, but also partial INDs; that is, INDs that are not fully valid: In a Web setting, even meaningful inclusions might be violated by incorrect or incomplete data. An extension of our approach to multi-valued INDs is tempting but increases the problem complexity significantly, and we do not expect meaningful multi-valued relationships among small Web tables.

REFERENCES

Ziawasch Abedjan, John Morcos, Michael Gubanov, Ihab F. Ilyas, Michael Stonebraker, Paolo Papotti, and Mourad Ouzzani. 2015. DataXFormer: Leveraging the web for semantic transformations. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org.

Jana Bauckmann, Ulf Leser, and Felix Naumann. 2010. *Efficient and Exact Computation of Inclusion Dependencies for Data Integration*. Technical Report. Universitätsverlag Potsdam.

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

Siegfried Bell and Peter Brockhausen. 1995. *Discovery of Data Dependencies in Relational Databases*. Technical Report. Universität Dortmund.

Frank Benford. 1938. The law of anomalous numbers. *Proceedings of the American Philosophical Society* 78 (1938), 551–572.

Chandra Sekhar Bhagavatula, Thanapon Noraset, and Doug Downey. 2013. Methods for exploring and mining tables on wikipedia. In *Proceedings of the ACM SIGKDD Workshop on Interactive Data Exploration and Analytics*. ACM, New York, NY, 18–26.

Thomas Bläsius, Tobias Friedrich, and Martin Schirneck. 2016. The parameterized complexity of dependency detection in relational databases. In *International Symposium on Parametrized and Exact Computation (IPEC)*. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13, 7 (1970), 422–426.

Michael J. Cafarella, Alon Y. Halevy, Daisy Z. Wang, Eugene Wu, and Yang Zhang. 2008. WebTables: Exploring the power of tables on the web. *Proceedings of the VLDB Endowment* 1, 1 (2008), 538–549.

Moses S. Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Symposium on Theory of Computing (STOC)*. ACM, New York, NY, 380–388.

Benjamin Kille, Frank Hopfgartner, Torben Brodt, and Tobias Heintz. 2013. The plista dataset. In *Proceedings of the International Workshop and Challenge on News Recommender Systems*. ACM, New York, NY, USA, 16–23.

Stéphane Lopes, Jean-Marc Petit, and Farouk Toumani. 2002. Discovering interesting inclusion dependencies: Application to logical database tuning. *Information Systems* 27, 1 (2002), 1–19.

Fabien De Marchi, Stphane Lopes, and Jean-Marc Petit. 2002. Efficient algorithms for mining inclusion dependencies. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. Springer, Berlin, 464–476.

Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. 2015a. Data profiling with metanome. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1860–1871.

Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Naumann. 2015b. Divide & conquer-based inclusion dependency discovery. *Proceedings of the VLDB Endowment* 8, 7 (2015), 774–785.

Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. 2009. A machine learning approach to foreign key discovery. In *Proceedings of the ACM Workshop on the Web and Databases (WebDB)*. ACM, Providence, RI.

Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. 2012. InfoGather: Entity augmentation and attribute discovery by holistic matching with web tables. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, New York, NY, 97–108.

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

Received February 2016; revised April 2017; accepted May 2017