# Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms

Falco Dürsch[1], Axel Stebner[1], Fabian Windheuser[1], Maxi Fischer[1], Tim Friedrich[1], Nils Strelow[1], Tobias Bleifuß[2], Hazar Harmouch[2], Lan Jiang[2], Thorsten Papenbrock[2], Felix Naumann[2]
Hasso Plattner Institute, University of Potsdam, Germany
[1]firstname.lastname@student.hpi.de, [2]firstname.lastname@hpi.de

## ABSTRACT

Inclusion dependencies are an important type of metadata in relational databases, because they indicate foreign key relationships and serve a variety of data management tasks, such as data linkage, query optimization, and data integration. The discovery of inclusion dependencies is, therefore, a well-studied problem and has been addressed by many algorithms. Each of these discovery algorithms follows its own strategy with certain strengths and weaknesses, which makes it difficult for data scientists to choose the optimal algorithm for a given profiling task.

This paper summarizes the different state-of-the-art discovery approaches and discusses their commonalities. For evaluation purposes, we carefully re-implemented the thirteen most popular discovery algorithms and discuss their individual properties. Our extensive evaluation on several real-world and synthetic datasets shows the unbiased performance of the different discovery approaches and, hence, provides a guideline on when and where each approach works best. Comparing the different runtimes and scalability graphs, we identify the best approaches for certain situations and demonstrate where certain algorithms fail.

## KEYWORDS

Inclusion Dependency Discovery, Data Profiling, Evaluation

**ACM Reference Format:**
Falco Dürsch, Axel Stebner, Fabian Windheuser, Maxi Fischer, Tim Friedrich, Nils Strelow, Tobias Bleifuß, Hazar Harmouch, Lan Jiang, Thorsten Papenbrock and Felix Naumann. 2019. Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms. In *The 28th ACM International Conference on Information and Knowledge Management (CIKM '19), November 3–7, 2019, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357384.3357916

## 1 INCLUSION DEPENDENCIES

An inclusion dependency (Ind) is a statement about a relational dataset indicating that all values of a certain attribute-combination are also contained in the values of another attribute-combination. This property makes Inds not only an important integrity constraint for relational databases [6] but also a prevalent notion in various data management use cases, such as foreign key detection [21, 26], query optimization [10], schema (re-)design [16], and data integration [18]. However, most datasets do not provide their Inds, either because the Inds have never been determined, because they have been lost during data integration, or simply because the data format does not allow the storage of such metadata. Whenever this is the case, data profiling algorithms are needed to efficiently discover the inclusion dependencies from the raw data.

**Problem Statement.** Ind discovery is one of the hardest tasks in data profiling: It is NP-hard [11], as well as one of the first real-world problems proven to be $W[3]$-complete [5]. The problem's complexity is mainly due to its exponentially large and complex search space combined with the expensive candidate checks that are required to verify Inds. For this reason, even the fastest Ind discovery algorithms take hours or even days to compute larger datasets, i.e., datasets of several Gigabyte size. Choosing the best algorithm is, therefore, crucial for the success of Ind profiling.

To deal with the high complexity, various Ind discovery algorithms pursue different search strategies that strive to prune and optimize the candidate checks. Each strategy has a different impact as well as strategy-specific costs and shortcomings depending on the input data and available computing resources. The difference in discovery performance is often several orders of magnitude so that picking one algorithm over another can make the difference of getting a job done in several minutes or failing in several hours. Unfortunately, no experimental study has ever compared all state-of-the-art Ind discovery approaches. The experiments published with each individual discovery algorithm cover only two, at best three approaches in a setting that is favourable for the published approach. So choosing the right approach is often a matter of luck.

**Contributions.** With this paper, we present the first comparative study that considers all state-of-the-art Ind discovery strategies. For this purpose, we survey, evaluate, and compare the most popular algorithms for Ind discovery namely Bell and Brockhausen [4], DeMarchi [7], Spider [3], S-indd [22], Binder [20], Sindy [15], Faida [14], Many [25], S-indd++ [24], Mind [8], ZigZag [9], Find2 [12], and Mind2 [23]. Each algorithm makes a significant contribution in at least one of the following areas:

- *Pruning strategies* that eliminate candidates from the search space without checking them
- *Traversal strategies* for the search space that maximize the pruning effects
- *Candidate checking techniques* that combine and/or shorten validation processes
- *Data management techniques* that avoid memory exhaustion

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*CIKM '19, November 3–7, 2019, Beijing, China*
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6976-3/19/11…$15.00
https://doi.org/10.1145/3357384.3357916



RIGHTSLINK

We support the choice of the right algorithm for a specific dataset by discussing the different strategies and techniques and by pinpointing their strengths and weaknesses. We measure the discovery times of all algorithms on real-world and synthetic datasets and make suggestions on when to use which algorithm based on the evaluation results. All algorithm (re-)implementations and all datasets are available online[1].

**Structure.** We start in Section 2 by providing the formal notations and terminology used in this paper. Section 3 then provides a survey of the uIND algorithms and Section 4 does the same for the nIND algorithms. In Section 5, we present our experimental evaluation and detailed discussions of the results. Section 6 summarizes the advantages and disadvantages of the algorithms and provides an outlook on possible future work in the field of IND discovery.

## 2 FOUNDATIONS

Formally, INDs are defined as follows [8]: Given two relational instances $r_i$ and $r_j$ of the relational schemata $R_i$ and $R_j$ respectively. When denoting tuples in $r_i$ and $r_j$ as $u$ and $v$ and attribute lists taken from $R_i$ and $R_j$ as X and Y, an IND $\sigma = R_i[\mathsf{X}] \subseteq R_j[\mathsf{Y}]$ (short: $\mathsf{X} \subseteq \mathsf{Y}$) is satisfied iff $\forall u \in r_i, \exists v \in r_j$ such that $u[\mathsf{X}] = v[\mathsf{Y}]$. Note that $R[\mathsf{X}]$ and $u[\mathsf{X}]$ denote the projection of schema $R$ and record $u$ on the attributes X.

We refer to the left-hand side X of an IND as *dependent* attribute(s) and the right-hand side Y as *referenced* attribute(s). The *arity* of an IND is defined as the number of its dependent attributes $n = |\mathsf{X}| = |\mathsf{Y}|$. Henceforth, a *unary* IND (uIND) is an IND with an arity of $n = 1$ and an *n-ary* IND (nIND) is an IND with an arity of $n > 1$. INDs of the form $R[\mathsf{X}] \subseteq R[\mathsf{X}]$ with exactly the same referenced and dependent attributes X are satisfied on any possible instance. Such *trivial* INDs do not need to be discovered.

According to the projection and permutation inference rules presented in [6], an IND $\sigma' = R_i[\mathsf{X}'] \subseteq R_j[\mathsf{Y}']$ *implies* an IND $\sigma = R_i[\mathsf{X}] \subseteq R_j[\mathsf{Y}]$ if X is a projection of any permutation of X′ and Y is the same projection of the same permutation of Y′. Applying the same permutation to both lists of attributes X and Y of an IND $\sigma$ always produces a new IND $\sigma'$ that is effectively the same IND as $\sigma$. Hence, it is common practice to fix the order of the dependent attributes X in each IND according to their order in the schema. It is also common practice to not consider INDs with repeating attributes (e.g., $R_i[\mathsf{A}, \mathsf{A}] \subseteq R_j[\mathsf{B}, \mathsf{C}]$), because such INDs are irrelevant for most use cases. Finally, an IND $\sigma$ is called *maximal*, if no *other* IND $\sigma'$ implies $\sigma$. The set of all maximal INDs is a *complete set* of INDs, because all non-maximal INDs can be derived from it.

Figure 1 shows an example of two relations, namely *Movies* and *Showings*. The following unary and n-ary INDs are valid, the last one being the only maximum IND:

$$Showings[\mathsf{Movie}] \subseteq Movies[\mathsf{Title}]$$
$$Showings[\mathsf{Length}] \subseteq Movies[\mathsf{Time}]$$
$$Showings[\mathsf{Movie}, \mathsf{Length}] \subseteq Movies[\mathsf{Title}, \mathsf{Time}]$$

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

| **Showings** | | | **Movies** | | |
|---|---|---|---|---|---|
| Screen | Movie | Length | Title | Stars | Time |
| 1 | Titanic | 194 | Titanic | 7.8 | 194 |
| 2 | Titanic | 194 | Shrek | 7.9 | 90 |
| 3 | Shrek | 90 | Ben-Hur | 8.1 | 212 |
| 1 | Ben-Hur | 212 | Gladiator | 8.5 | 155 |

**Figure 1: Example instances of two relations**

**Candidate validation.** Every uIND discovery algorithm, except BELL AND BROCKHAUSEN, proposes its own, optimized candidate evaluation technique that we discuss in detail with each algorithm. BELL AND BROCKHAUSEN and all nIND discovery algorithms, except BINDER, MIND2, and FAIDA, however, rely on SQL (and hence a database) for their candidate checks. The three most popular validation queries for IND candidates either use an *outer join*, such as `LEFT OUTER JOIN`, a *set operation*, such as `NOT IN` or `MINUS`, or a *correlated subquery*, such as `NOT EXISTS` or `NOT IN`. The optimal query pattern for IND validations and, hence, the query pattern used for all algorithms in this experimental survey is the *outer join* for the following two reasons: First, *set operations* handle `null` values differently, i.e., as *unknown* values that can take the role of any missing left hand side value. For this reason, *set operations* find more and potentially wrong INDs when facing `null` values. Second, *outer joins* are highly optimized in most RDBMS, can use indexes if present, and support early termination with the `LIMIT 1` and `FETCH FIRST 1 ONLY` keywords. If no `null` values are involved, though, RDBMS might produce same execution plans for both *set operations* and *outer joins*. In our experiments, *outer joins* were in fact always the fastest approach – usually a few orders of magnitude faster than *correlated subqueries* and on par with *set operation* – and they always produced the expected results, in contrast to *set operation*. Hence, all our SQL-based algorithms use *outer joins*. For more details on SQL-based candidate validation, we refer to [1].

## 3 UNARY IND DISCOVERY

In this section, we discuss the various algorithms that have been proposed for the discovery of unary inclusion dependencies over the past years of research. Table 1 gives an overview of the nine algorithms that we consider in this study. Each of these algorithms contributes at least one unique technique that is often also used by their successor algorithms. Before we discuss the algorithms individually, we quickly go over their lineage.

### 3.1 History of unary IND discovery

In 1995, Bell and Brockhausen proposed the first pruning strategies that effectively reduce the search space for uIND discovery [4]. Their main idea was to use logical inference over already discovered INDs as well as basic column statistics to avoid many of the expensive candidate checks. To improve the efficiency of the individual candidate checks, DEMARCHI et al. suggested the use of an inverted index in 2002 [7]. Because this entire inverted index can become large and needs to be stored in main memory, Bauckmann et al. proposed a disk-based sort-merge-join algorithm called SPIDER in 2006 that not only overcomes the memory problem of DEMARCHI, but also



RIGHTSLINK

**Table 1: Overview of unary inclusion dependency discovery algorithms**

| Name | | Year | Validation | Storage | Special |
|---|---|---|---|---|---|
| Bell and Brockhausen | [4] | 1995 | SQL outer join | in-memory | extensive pruning with statistics |
| DeMarchi | [7, 8] | 2002 | hash join | in-memory | all-column hash join |
| Spider | [3] | 2006 | sort-merge join | disk-backed | all-column sort-merge join |
| S-indd | [22] | 2015 | sort-merge join | disk-backed | all-column sort-merge join and attribute clustering |
| Binder | [20] | 2015 | hash join | disk-backed | all-column hash join and divide & conquer |
| Sindy | [15] | 2015 | map-reduce | disk-backed | MapReduce-style parallel/distributed discovery |
| Faida | [14] | 2017 | HLL intersect | disk-backed | approximation via hashing, sampling, and sketching |
| Many | [25] | 2017 | hash set intersect | in-memory | BloomFilter-based candidate generation |
| S-indd++ | [24] | 2018 | sort-merge join | disk-backed | all-column sort-merge join and non-uniform bucketing |

improves the validation efficiency dramatically by checking several candidates simultaneously and using early termination for the candidate checks [3]. An issue in Spider, namely that it might need to open too many file handles simultaneously, was solved by Shaabani and Meinel's algorithm S-indd in 2015 [22]. Their approach solves the file handle problem by deriving clusters of attributes incrementally and utilizing different partitioning schemes in the process. At the same time, Papenbrock et al. introduced Binder, a disk-based hash-join approach that tackles the Ind discovery task with divide & conquer techniques [20]. A first fully distributable Ind discovery algorithm called Sindy was also presented in 2015 [15]. The approach proposed by Kruse et al. also extends the DeMarchi algorithm, but by making it parallel and distributed rather than using early termination techniques. In 2017, Kruse et al. proposed another Ind discovery algorithm that uses various approximation techniques to improve the discovery performance [14]. The algorithm cannot guarantee correct results, but it almost always provides correct results in practice. The Many algorithm by Tschirschnitz et al. is a uInd discovery algorithm of 2017 that is optimized for many, i.e., hundreds of thousands of tables with only very few records. It proposes a clever uInd candidate generation that avoids generating $n^2$ many candidates. The most recent algorithm S-indd++ by Shaabani et al. takes inspiration from S-indd and Binder and seeks to improve upon both of these algorithms [24].

## 3.2 Unary IND algorithms

**Bell and Brockhausen.** The unary Ind discovery algorithm Bell and Brockhausen [4] uses SQL-join statements to validate the various uInd candidates. Because SQL-based candidate validations are expensive, the algorithm tries to prune as many candidates as possible via *statistical pruning* and *transitivity pruning*. For statistical pruning, the algorithm generates only those candidates that have matching value ranges and data types, i.e., the uInd candidates A ⊆ B is initially generated from data statistics only if A and B have the same data type and the value range of B, which is $[min(B), max(B)]$, encloses the value range of A. After the candidate generation, the algorithm iteratively validates the candidates (via SQL) inserting every true uInd A ⊆ B as a directed edge (A, B) into a graph structure. For every inserted uInd, it tries to infer the (in-)validity of other uInds through transitivity properties, i.e., it applies transitivity pruning. For every two edges (A, B) and (B, C) in the graph, it adds the transitive edge (A, C) and removes the respective uInd candidate A ⊆ C from the candidate list, because this candidate needs to be true as well. Since the candidate list of uInds is sorted by their dependent attribute, the algorithm can, when inserting an edge (B, A), also prune any B ⊆ C where A ⊆ C has no edge in the graph, because these uInds must be invalid too.

**DeMarchi.** The DeMarchi [7, 8] algorithm uses a novel candidate validation technique based on an inverted index. Similar to Bell and Brockhausen, the algorithm first groups the attributes by their data types into *extraction contexts* to generate (and validate) only candidates with matching attributes. To generate all candidates, every attribute (= dependent A) is mapped to the set of all other attributes (= referenced $Ref_A$) of the same extraction context. For every extraction context, the algorithm then builds an inverted index, which is held in main memory, such that every value $v$ is mapped to a set of all attributes $U_v$ containing that value. A uInd A ⊆ B is valid if every attribute set $U_v$ in the inverted index that contains A also contains B. Hence, to evaluate all candidates, DeMarchi's uInd inference step scans all $U_v$ of the inverted index intersecting every $Ref_A$ with $U_v$ where $A \in U_v$. In other words, every intersection removes all candidates A ⊆ B where A contains value $v$ but B does not. All candidates that survive the intersection process are true uInd and are reported as $\{A \subseteq B | B \in Ref_A\}$.

**Spider.** The uInd discovery algorithm Spider [3] is a disk-backed all-column sort-merge join with early termination. In the sorting step, the algorithm reads the input relation(s) attribute-wise, sorts and de-duplicates each column, and writes the sorted lists of values back to disk into individual files – one per attribute. The authors of Spider propose to sort the columns via in-database SQL `ORDER BY` statements, but non-SQL-based sorting is also possible with Spider as shown in [20]. After generating all uInd candidates (just like DeMarchi as a map of dependent attributes A to sets of referenced attributes $Ref_A$ all of same data type), Spider opens all sorted value files simultaneously creating one file iterator per attribute. By inserting all head values with their respective attributes into a min-heap data structure (min by head value), the algorithm can read a set of attributes with same value $v$ from the head of the min-heap. This attribute set is equivalent to an index value $U_v$ in DeMarchi's inverted index. Hence, for candidate validation, Spider intersects the retrieved $U_v$ with all $Ref_A$ attribute sets with $A \in U_v$. Then, it reads the next value for all attributes in $U_v$, which updates the min-heap and lets the algorithm read the attribute set $U_{v'}$ for the next value $v'$. If an attribute has been removed from all uInd candidates,



RIGHTSLINK

its iterator can be closed. When an iterator reaches the end of a value list, Spider can close it and, when all iterators are closed, all remaining candidates must be true uInds.

**S-indd.** The S-indd [22] is an extension of Spider that seeks to reduces the number of simultaneously opened file handles with an iterative merging phase. Similar to Spider, S-indd first transforms the input into sorted value files, but it adds an attribute label to each value in these files, i.e., instead of writing the values *a*, *b*, and *c*, it writes the tuples (*a*,A), (*b*,A), and (*c*,A) into the file of attribute A. Then, instead of opening all files simultaneously, S-indd opens *k* files and to merge their sorted lists into one sorted list of value-attributelist pairs, such as (*a*,A), (*b*,AB), (*c*,AB), and (*d*,B). The merge process continues until fewer than *k* lists remain. To cope with large input datasets, S-indd also proposes to horizontally partition the data in the merge process and to merge the partitions in the last merge step. The algorithm could now open the remaining files and run the sort-based intersections from Spider, but instead S-indd introduces another trick: The algorithm merges all remaining sorted-value lists into one list of attribute lists $U_v$ (= the values of DeMarchi's inverted index) and de-duplicates this list. Only these duplicate-free attributes lists are then used for the intersection-based candidate validation process.

**Binder.** The algorithm Binder [20] is an all-column hash-join approach similar to DeMarchi but with three major improvements: First, Binder hash-partitions the data into smaller chunks that fit into main memory before actually checking for Inds; second, it uses an additional, dense index to avoid redundant intersect operations during candidate validation; and, third, it adopts the *apriori-gen*-based [2] bottom-up lattice traversal strategy of the Mind algorithm, which we introduce in Section 4, to discover not only unary but also n-ary Inds. The algorithm's overall discovery strategy follows the divide & conquer paradigm: The *divide phase* reads all input relations and splits the values in each column via hash-partitioning into a fixed number of buckets; all values with the same hash are placed into the same bucket and duplicate values are removed. In the end, all buckets are written to disk. The *conquer phase*, then, reads the buckets successively back into main memory for candidate validation. Every read operation fetches all such buckets that share the same hash. If these buckets do not fit into main memory, Binder re-partitions them as described in the divide phase. Once successfully loaded, the data is transformed into an inverted index (see DeMarchi) and a dense index that points values to the lists of attributes that they occur in. During candidate validation, the inverted index is used to intersect the candidates' referenced sets $Ref_{\mathsf{A}}$ and the dense index is used to select the attribute sets $U_v$ from the inverted index, such that redundant intersections are avoided. Processing the buckets successively avoids memory overflows and allows the algorithm to prune buckets of attributes whose Ind candidates have all been falsified (see Spider). In the end, only valid Inds remain. The divide-and-conquer cycle then repeats for each arity of n-ary Ind candidates using a variation of the *apriori-gen* algorithm for candidate generation (see Section 4).

**Sindy.** The algorithm Sindy [15] is a distributed and, hence, fully parallel uInd discovery algorithm. It is designed around the functional programming primitives *map* and *reduce* and, therefore, well suited for data flow frameworks, such as Apache Hadoop, Apache Spark, and Apache Flink. The core idea is similar to the DeMarchi algorithm, but expressed in distributable terms: In the index construction phase, Sindy reads all records to first `map` their values to the attributes they occur in and then `group` these mappings into what is DeMarchi's inverted index. For the candidate generation phase, Sindy takes the attribute sets from the inverted index and, then, uses a `map` to transform each attribute set into all uInd candidates that can be deduced from this attribute set. Marching on with the candidates, Sindy's validation phase `groups` all candidates by their dependent attribute and, then, intersects their lists of referenced attributes. Again, only the true uInds survive the intersection process. Due to the distributed nature of Sindy's discovery strategy, however, no early termination optimizations are being applied.

**Faida.** In contrast to all other algorithms in this survey, Faida [14] is not an exact but an approximate approach to the discovery of Inds. Although approximate algorithms also *try* to find all valid Inds, their results are not guaranteed to be correct and/or complete. Waiving certain result guarantees allows the use of more efficient discovery strategies, such as sampling or sketching. The Faida algorithm in particular sacrifices correctness for efficiency, but it still guarantees completeness, which means that all true Inds are discovered but some of them might actually be false positives. This is an important feature, because re-validating a given Ind is much cheaper than finding missing Inds. Furthermore, the experiments in [14] and our own experiments show that Faida's false positives rate is really small – the algorithm reported exact results in most of our experiments. The three approximation techniques that Faida uses are hashing, sampling, and sketching. In a preprocessing step, the algorithm first hashes every value in the input relation to compact the data and to accelerate its later processing. The hashed values are then stored column-wise on disk to not exhaust main memory. At the same time, Faida bootstraps a small sample of the hashed records. After hashing and sampling, Faida generates all uInd candidates. To validate them, the algorithm creates probabilistic *HyperLogLog* (*HLL*) structures from the hashed columns and an inverted index (see DeMarchi) from the sample. A candidate is then validated on the inverted index, if its dependent side is of low cardinality, and on the *HLL* sketches, otherwise. A candidate $\mathsf{X} \subseteq \mathsf{Y}$ is valid, if the set cardinality of Y, which is approximated by $HLL_{\mathsf{Y}}$, is equal to the joint set cardinality of X and Y, which is $HLL_{\mathsf{X}\cup\mathsf{Y}}$. The cardinality-based test is imprecise if the cardinalities are low; hence, low cardinality attributes are tested using the inverted index. Similar to other validation strategies, this approximate test can be applied to unary and n-ary Ind candidates. For this reason, Faida can – similar to Binder– successively generate and validate candidates of higher arity to discover both uInds and nInds.

**Many.** The Many algorithm [25] is a uInd discovery approach optimized for a high number of short input relations, i.e., millions of attributes with only a few dozens values (see, for instance, web table data). If the number of attributes is high, materializing all possible uInd candidates is infeasible due to their quadratic memory consumption. For this reason, Many's main contribution is a clever candidate generation strategy that is followed by a simple candidate validation step: At first, the algorithm reads the all input tables running each relational column through a *BloomFilter*. Then, it successively generates all possible uInd candidates from these



RIGHTSLINK

BloomFilters. A ⊆ B is a uInd candidate if and only if all bits in A's BloomFilter are also set in B's BloomFilter. To efficiently generate all candidates that meet this condition, Many transforms all $n$ BloomFilters of length $m$ into a $m \times n$ matrix of bitsets, i.e., the column-based BloomFilters are turned into row-based bitsets. The algorithm then considers every attribute A as a potential dependent side attribute and collects all those bitsets from the matrix that hold a 1-bit for attribute A. After intersecting these (few dozen) bitsets, the resulting bitset holds a 1-bit for all those attributes B that form potential uInds A ⊆ B with A. Because the input tables are expected to be short, Many quickly validates each such candidate via in-memory hashing-based set intersection on the actual values. For this intersection, the tables are dynamically loaded from disk while a least-recently-used cache is used to minimize disk I/O.

**S-indd++.** The algorithm S-indd++ [24] is an improvement of the algorithm S-indd in that it introduces a new partitioning approach for the sorted value-attribute(lists). In the sorting phase, S-indd++ starts by partitioning the data using a hash-function. In contrast to Binder and S-indd, S-indd++ does not aim for equally sized partitions and instead deliberately creates a skew in their size: The first level of partitions should be small and further level increasingly larger. The intuition is that, usually, some attributes can be pruned early on when they get disconnected from all their Ind candidates, which makes subsequent partitions naturally smaller anyways. Hence, by processing buckets of smaller size first, S-indd++ tries to avoid the merging of many value lists.

## 4 N-ARY IND DISCOVERY

This section discusses four well established discovery techniques for n-ary inclusion dependencies – an overview is given in Table 2. All nInd algorithms start with the uInds that they either computed themselves or gathered from one of the uInd algorithms discussed before. Then, all except Mind2 traverse a search space that is modeled as a *candidate lattice*. In the following, we first introduce this search space model, then discuss the lineage of n-ary discovery techniques, and finally survey each technique individually.

**Search space model.** A lattice is a partially ordered set of attribute lists where each pair of elements has a unique supremum and infimum. The first level of the lattice contains all *valid* unary Inds, which are to be discovered with one of the more specialized uInd algorithms. All further levels combine (n−1)-ary candidates from the previous level to n-ary candidates as seen in Figure 2, such that an edge between nodes means that the larger Ind implies the smaller one. This property, which is also known as anti-monotony, is useful in Ind-discovery, because whenever an Ind is known to be unsatisfied, all Inds that are reachable by only following edges upwards, are also known to be unsatisfied.

### 4.1 History of n-ary IND discovery

The algorithm Mind by De Marchi et al. was proposed in 2002 and constitutes the first published algorithm for nInd discovery [7]. It proposes an apriori-gen-based approach to generate nInd candidates and, hence, traverses the lattice bottom-up using the anti-monotony property of the candidates to prune invalid ones from the lattice using their invalid generalizations (*upward-pruning*). The extension with which Binder and Faida are later also able to discover

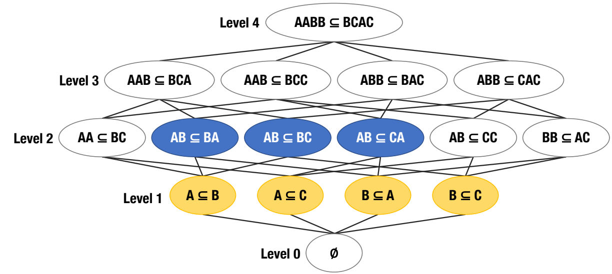


**Figure 2: A 5-level example lattice: Each node is one Ind candidate. The four unary Inds are given; the three marked nInds are to be checked; the white nodes can be disregarded.**

nInds is the same bottom-up lattice traversal as used by Mind. They, however, continue with optimized candidate validation strategies. Because it is also possible to infer the validity of uInd candidates from their valid specializations (*downward-pruning*), De Marchi and Petit proposed shortly after Mind another nInd algorithm, ZigZag, which estimates an optimistic and a pessimistic border of Ind candidates and alternates between them using bottom-up and top-down traversals [9].

Also in 2003, Koeller et al. pursued a similar strategy with their algorithm Find2 [12]. Instead of estimating an optimistic border, their algorithm first models the candidates as a hypergraph. With this representation, Find2 can calculate hypercliques that serve to deduce large nInd candidates early on; it then traverses the lattice in an aggressive bottom-up fashion using smaller hypercliques. In 2016, Shaabani and Meinel presented an alternative search space traversal technique, which infers all maximum uInds from so-called uInd coordinates [23]: When iteratively intersected, these coordinates generate all maximum uInds directly from the data; hence, the algorithm does not need to validate candidates individually.

Mind, ZigZag, and Find2 use SQL for candidate validation, while the other three use their own strategies. Furthermore, Find2, ZigZag, and Mind2 require all unary Inds (and sometimes even more Inds) as input – they cannot discover them. And finally, Find2, Mind2, and ZigZag discover only maximal Inds while Mind, Binder, and Faida report all n-ary Inds due to their apriori candidate generation – the final result set sizes therefore usually differ between these two groups.

### 4.2 N-ary IND algorithms

**Mind.** The nary Ind discovery algorithm Mind [8] is an *apriori-gen*-based bottom-up lattice traversal algorithm using SQL candidate validation. Mind takes all true uInds as input. Starting with the uInds, it traverses the lattice level-wise, generating and testing ever larger nInd candidates. The candidate generation uses a variant of the *apriori-gen* algorithm, proposed by Agrawal and Srikant for frequent itemset mining [2]. The *apriori-gen* algorithm is applicable to the generation of nInd candidates due to their anti-monotonicity property: Every Ind candidate that is *implied* by a valid Ind, i.e., that is a projection of a valid Ind, must be a valid Ind as well (*downward-pruning*). Vice versa, every Ind candidate that implies an invalid Ind must be invalid, too (*upward-pruning*).



RIGHTSLINK

**Table 2: Overview of n-ary inclusion dependency discovery algorithms**

| Algorithm | | Year | Validation | Pruning | Traversal | Special |
|---|---|---|---|---|---|---|
| MIND | [8] | 2002 | SQL outer join | upwards | bottom-up | apriori-gen based candidate generation |
| ZIGZAG | [9] | 2003 | SQL outer join | up-/downwards | bi-directional | pruning via optimistic and pessimistic borders |
| FIND2 | [12] | 2003 | SQL outer join | up-/downwards | bi-directional | clique-finding in hypergraphs |
| BINDER | [20] | 2015 | hash join | upwards | bottom-up | apriori-gen based candidate generation |
| MIND2 | [23] | 2016 | sort-merge join | upwards | inference | iterative merging of UIND coordinates |
| FAIDA | [14] | 2017 | HLL intersect | upwards | bottom-up | apriori-gen based candidate generation |

MIND's variant of *apriori-gen* generates NIND candidates in such a way that only possibly true candidates are created – false NINDS are automatically inferred and (upward-)pruned in the process. To generate all candidates for level $k$, the algorithm uses the true INDS from level $k-1$: In short, $X' \subseteq Y'$ is a valid candidate if all its implied INDS $X \subseteq Y$ with $|X| = |X'| - 1$ and $|Y| = |Y'| - 1$ are true INDS of the previous level. After generating all candidates for level $k$, they are validated using SQL queries against a database. The process repeats for every level $k' = k + 1$ until no more candidates are created. The same candidate generation strategy but with different validation approaches is also used by the algorithms BINDER and FAIDA.

**ZIGZAG.** The ZIGZAG algorithm [9] also uses the candidate lattice as a search space model, but it alternates between using the *apriori-gen*-based bottom-up traversal and a top-down traversal, 'zigzaging' between bottom and top IND candidates. The algorithm takes not only the unary INDS as input but all INDS of the first $k$ lattice levels; as proposed by the authors, we use $k = 2$ as default. The valid INDS are used to initialize a *negative border*, a *positive border*, and an *optimistic positive border*. The negative border contains all known *minimal invalid* INDS and the positive border contains all *maximal valid* INDS. Whenever an IND candidate is tested to be true or false, it is added to the corresponding border. The optimistic positive border, in contrast, contains the *maximal candidate* INDS, which are derived as largest compositions of valid, positive cover INDS.

During NIND discovery, ZIGZAG jumps back and forth between the negative/positive borders and the optimistic positive border. The algorithm starts by validating the maximal IND candidates in the optimistic positive border. If a candidate is true, all implied candidates must be true as well and can be pruned (downward-pruning). If the tested candidate is large, this pruning might have a high impact. However, if the candidate is false, ZIGZAG calculates the percentage of rows that dissatisfy the IND, i.e., the $g_3'$ error measure [17]. Because ZIGZAG needs the number of violating rows, the SQL queries cannot use early termination when validating IND candidates. In case the error measure is below a certain threshold $\varepsilon$, the algorithm checks all direct subset INDS of the invalid candidate (top-down traversal); otherwise or after evaluating all optimistic positive border candidates, ZIGZAG jumps to the positive border to generate and validate the next level in bottom-up *apriori-gen*-style.

**FIND2.** The bi-directional NIND discovery algorithm FIND2 [12] translates the problem of NIND discovery to the discovery of cliques in $k$-uniform-hypergraphs, where each edge must connect exactly $k$ nodes. The given UINDS are the nodes in these graphs and the NINDS at level $k$ of the search space lattice are the $k$-hyperedges connecting the UINDS from which they are composed. Each level of the lattice, which is a set of $k$-ary INDS, is represented as a $k$-uniform-hypergraph. Just like ZIGZAG, FIND2 takes the first $k$ levels of true INDS as input (we use $k = 2$ by default). After constructing the $k$-uniform-hypergraph for the largest given $k$, the algorithm calculates all maximum cliques in that graph; these cliques correspond to maximum NIND candidates. Finding maximum cliques in $k$-uniform-hypergraphs is NP-hard, but FIND2's HYPERCLIQUE algorithm solves the problem efficiently for sparse graphs with only a few cliques, which is usually the case for NIND discovery scenarios. After generating all cliques, i.e., maximum NIND candidates, FIND2 validates them using SQL validation queries. If an IND candidate evaluates to true, the algorithm has discovered a maximal IND and prunes all implied candidates from the search space (downward-pruning); otherwise, the candidate is broken into INDS of arity $k + 1$, which are evaluated thereafter. The valid ones are then used in the next iteration that starts by finding maximum cliques in level $k + 1$. FIND2 terminates if no new cliques are found.

**MIND2.** The NIND discovery algorithm MIND2 [23] requires all valid UINDS as input and then discovers all maximum inclusion dependencies by deriving them from so-called *UIND coordinates*. The coordinates of a UIND $A \subseteq B$ is the set of all tuple pairs $(i, j)$ for which the value of A at index $i$ is equal to the value of B at index $j$. MIND2 groups these UIND coordinates by their $i$-index resulting in a mapping of $i$-indices to lists corresponding $j$-indices. By transforming value lists into coordinates, MIND2 formulates an index structure for the NIND discovery process that relies on simple integers rather than complex (string) values. The creation of the coordinates is done via SQL inside a database. Afterwards, however, MIND2 stores the coordinate mapping (sorted by $i$-index) for each UIND in a separate file on disk. It then infers an initial set of maximum INDS from the given UINDS and opens a file reader to each UIND coordinates file.

The intuition for the next step is the following: $AC \subseteq BD$ is valid, iff for all $i$ in $A \subseteq B$ and $C \subseteq D$'s coordinate files the intersection of the respective $j$-lists is not empty. In other words, $\forall\, r_i[AC]\ \exists\, r_j[BD] : r_i[AC] = r_j[BD]$. MIND2 checks this property by iterating all coordinate files simultaneously. After reading the next mapping of $i$ to a list of $j$-values from all files, the algorithm constructs maximum sets of dependent attributes (and according referenced attributes) such that their intersection of $j$-value lists is not empty. These maximum INDS at position $i$ are used to update, i.e., prune the working set of maximum INDS. MIND2 then reads and processes the next position $i + 1$ until all coordinate files are finished. In the end, the working set contains all maximal INDS.



RIGHTSLINK

# 5 EVALUATION

This section analyzes and compares the state-of-the-art IND discovery algorithms of the previous sections. After introducing our experimental setup, we first present the results for uIND discovery and then those for nIND discovery.

## 5.1 Experimental setup

We implemented all algorithms for our *Metanome* data profiling framework[2], which defines standard interfaces for different kinds of profiling algorithms [19]. All common tasks, such as input parsing, result formatting, performance measuring, and algorithm parametrization, are standardized by the framework, and decoupled from the algorithms to ensure a uniform test environment for all thirteen implementations. Our implementations, additional documentation, and the datasets used in the experiments are available online[3].

**Datasets.** We evaluate all algorithms on several real-world and three synthetic datasets, namely TPC-H, TESMA, and GHAIND. Table 3 lists all datasets and their characteristics. FAIDA's approximate IND counts are given in brackets if they differ from the exact results. Many of these datasets already served evaluation purposes in the original publications. The numerous selected datasets reflect a broad range of application domains, such as biology, medicine, literature, and business.

**`Null` semantics.** Depending on the interpretation of `null` values, certain INDs might be true or false. Finding the right interpretation depends on the use case and can be more [13] or less [14] complex. Because `null` semantics for INDs are still an open research topic and the focus of this evaluation is on performance rather than IND semantics, we decided to interpret all `null` values as empty strings, which basically bypasses the interpretation issue: `null` values are equal to each other but different to all other values. We also discard completely empty columns, because they trivially generate nINDs with all other INDs.

**Hardware and software.** All experiments are executed on a Dell PowerEdge R620 running CentOS 6.10. The test machine has two Intel Xeon E5-2650 (2.00 GHz, Octa-Core) processors and 128 GB DDR3-1600 RAM. All algorithms (besides SINDY) are single-threaded and use only one processor core. The algorithms run on Oracle's JDK 64-Bit Server VM 1.8.0_151 and read their input data from a PostgreSQL 9.3.23 database.

## 5.2 UIND experiments

We first evaluate the runtime of the unary IND discovery algorithms on different datasets and then analyze the algorithms' runtime behavior for increasing dataset sizes. The experiments will, in summary, show the following: The in-memory algorithm DEMARCHI performs best for all datasets that easily fit into main memory; if the datasets are larger than main memory or at least a few gigabytes large, BINDER is the most efficient approach in general; however, FAIDA is faster than BINDER, if we can tolerate false positives in the results, and SINDY is even faster than FAIDA (and, hence, the fastest algorithm over all), if the hardware has at least eight cores.

[2] http://www.metanome.de

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

**Table 3: Dataset characteristics**

| Dataset | Size | Attributes | uINDs | nINDs |
|---|---|---|---|---|
| SCOP | 16 MB | 22 | 39 | 36 |
| CATH 4.0 | 16 MB | 25 | 50 | 81 |
| CENSUS | 112 MB | 42 | 39 | 89 |
| WIKIPEDIA | 540 MB | 14 | 2 | 0 |
| BIOSQL | 560 MB | 77 | 348 | 507 |
| WIKIRANK | 697 MB | 29 | 15 | 103 |
| LOD | 830 MB | 41 | 258 | Unknown |
| ENSEMBL | 836 MB | 130 | 364 | 100 |
| †GHAIND | 822 MB | 36 | 18 | 203 |
| †TESMA | 1 GB | 114 | 2 | 0 |
| †TPC-H 1 | 1 GB | 61 | 96 (99) | 8 |
| †TPC-H 10 | 10 GB | 61 | 97 | 11 |
| MUSICBRAINZ | 27 GB | 1 054 | 49 829 (49867) | Unknown |

†synthetic datasets

**Runtimes on different datasets.** Table 4 lists the measured runtimes of all nine uIND discovery algorithms for all evaluation datasets. Despite its clever candidate pruning strategies, BELL AND BROCKHAUSEN performs worst, being the only approach that relies on SQL. DEMARCHI performs remarkably well on small datasets, even though its validation strategy has been used and optimized by all other algorithms, because it saves the time these other algorithms need to spill intermediate data structures to disk. On datasets with only a few attributes and few distinct values (e.g., TPC-H, which is generated from a fixed-size seed), DEMARCHI's redundant set intersections are much less expensive than any disk writing costs; however, on large datasets, such as MUSICBRAINZ, the redundancy in the candidate validations is more expensive than the disk I/O. SPIDER does not require the input dataset to fit into main memory and still offers a reliably good performance. The algorithm is very easy to implement and works well as long as the number of attributes is below the operating system's maximum number of open file handles. The S-INDD algorithm does not have SPIDER's file handle limitation. It is, though, not a clear performance improvement over SPIDER: S-INDD is sometimes faster, because the algorithm explicitly de-duplicates the attribute lists $U_v$, which prunes many redundant list-intersect operations; S-INDD is also sometimes slower, because it writes more data to disk (values and attribute instead of only values), it might read the data more often (iterative merge process), and it needs to calculate additional hashes for the data partitioning.

The BINDER algorithm outperforms SPIDER and S-INDD on most datasets, because its hash-join approach is more efficient than the sort-merge-join approach ($O(n)$ instead of $O(n \log n)$) and because BINDER reads the input relation only once, while the sort-based approaches read every attribute once. It performs more attribute-list intersections than S-INDD, but it reads and writes less data from/to disk. S-INDD++ improves the disk I/O of S-INDD and, hence, achieves better runtimes; its overall performance still slightly falls behind BINDER. The approximate IND discovery algorithm FAIDA sacrifices the correctness guarantee of its results for clearly better runtimes than all other (single-threaded) algorithms. The approximation techniques clearly outperform the exact algorithms especially on large datasets, where data management is not trivial. Although FAIDA



RIGHTSLINK

**Table 4: uInd performance on real-world datasets (minutes)**

| Datasets | B&B | DeMarchi | Spider | S-indd | Binder | S-indd++ | Faida | Many | Sindy 1 worker | Sindy 8 workers | Sindy 32 workers |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Scop | 0.14 | **0.04** | 0.08 | 0.09 | 0.07 | 0.08 | 0.05 | 0.08 | 0.49 | 0.35 | 0.36 |
| Cath | 0.11 | **0.02** | 0.05 | 0.05 | 0.04 | 0.05 | 0.03 | 0.04 | 0.45 | 0.38 | 0.37 |
| Census | 1.05 | **0.09** | 0.15 | 0.17 | 0.14 | 0.14 | 0.12 | 0.14 | 0.66 | 0.38 | 0.38 |
| Wikipedia | >4h | 1.02 | 1.47 | 1.56 | 1.22 | 1.25 | 1.11 | 1.34 | 1.93 | 0.59 | **0.51** |
| BioSql | 4.98 | 0.76 | 1.30 | 1.48 | 1.41 | 1.15 | 0.9 | 1.15 | 2.14 | 0.63 | **0.51** |
| Wikirank | 2.90 | 0.73 | 1.53 | 1.44 | 1.23 | 1.04 | 0.87 | 0.97 | 2.33 | 0.85 | **0.55** |
| Lod | 0.34 | **0.25** | 0.45 | 0.41 | 0.30 | 0.37 | 0.69 | 0.36 | 1.81 | 0.68 | 0.46 |
| Ensembl | 23.52 | 2.1 | 3.04 | 3.70 | 2.39 | 3.05 | 1.76 | 2.85 | 3.50 | 0.91 | **0.62** |
| Tesma | >4h | 3.66 | 3.30 | 4.75 | 4.27 | 6.13 | 3.11 | 3.71 | 2.57 | 0.69 | **0.54** |
| TPC-H 1 | 17.79 | 1.96 | 3.88 | 3.58 | 2.96 | 2.72 | 2.08 | 3.00 | 5.55 | 1.34 | **0.78** |
| TPC-H 10 | >4h | 22.81 | 44.43 | 36.54 | 28.21 | 28.39 | 19.19 | >4h | 57.78 | 10.22 | **4.89** |
| Musicbrainz | >4h | 136.03 | 61.42 | 106.26 | 45.69 | 71.22 | 27.67 | 105.49 | 175.97 | 30.22 | **15.38** |

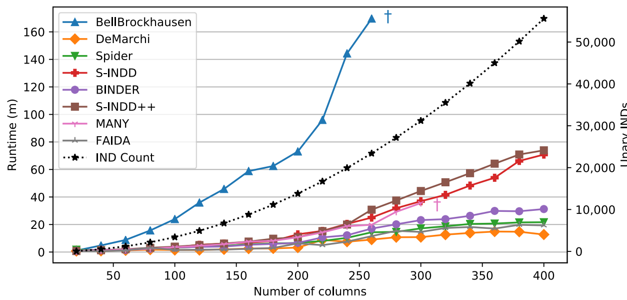


**Figure 3: Column scalability of uInd algorithms**

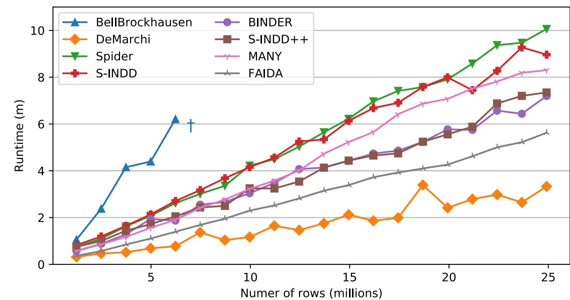


**Figure 4: Row scalability of uInd algorithms**

does not guarantee correctness, it reported correct results for most datasets (see Table 3). The Many algorithm is optimized for datasets with many attributes but only few rows. It can process the datasets in this evaluation only because the datasets fit into main memory, where simple set-based intersections can be used. On smaller datasets, the algorithm even competes well with those competitors that use optimized Ind validation techniques, because Many prunes the candidate space very effectively. Sindy, the parallel uInd discovery algorithm, is the second slowest approach if only one CPU core can be used for the discovery, because its parallelization capabilities with Apache Flink introduce some overhead for, e.g., work scheduling, data transformation, and framework startup. However, because the algorithm scales very well with the number of cores, it surpasses all other discovery approaches on larger datasets given at least eight cores (on small datasets, the startup-time for a Flink cluster is higher than the actual discovery times).

**Runtimes on different row and column numbers.** To measure the algorithms' scalability w.r.t. the number of columns and rows in the input dataset, we use the `editor_sanitised` relation of the Musicbrainz dataset, which has 20 columns and 1 245 661 rows, is 270 MB in size, and offers 48 uInds. To scale the number of columns, we start by measuring the runtime for `editor_sanitised` alone and then successively replicate the same relation, such that the first run processes 20 columns, the second run 40, and so on. In this way, the number of rows is fixed. Figure 3 shows the result of this experiment. The † symbol at the end of a runtime curve indicates that the algorithm exceeded the four-hour time limit.

The measurements show that all algorithms scale similarly with the number of columns, which is, the relative runtimes stay about the same and no algorithm surpasses others when more columns are being added. In general, S-indd, S-indd++, and Many should profit from an increasing number of columns: While all other algorithms generate all $n^2$ uInd candidates (just to clear most of them later on), these three algorithms generate the candidates directly from the data, i.e., they generate fewer candidates. On the 400 columns in this experiment and on up to 1054 columns (see Musicbrainz) in other experiments, we could not measure this advantage, because the candidate generation and pruning is, although quadratic, still highly optimized and dominated by I/O. On datasets with many thousand attributes, as were used in the evaluations of [22] and [25], the algorithms should however show some advantage, because the allocation and removal of many millions of candidates is costly.

To assess the algorithms' scalability with the number of rows, we again took the `editor_sanitised` relation and, this time, extended the relation via repeated copy-and-append. Figure 4 shows the runtimes of the different algorithms when successively increasing the length of the relation. Overall, the runtimes for all algorithms scale about linearly with the length of the input. This is mainly due to the fact that the I/O costs, which dominate the algorithms' runtimes, increase about linearly for all algorithms; non-linear aspects, such as



RIGHTSLINK

re-partitioning and sorting processes, do not have a noticeable impact in this experiment. The runtime spikes in the linear-trending curves are a result of the hashing-based candidate pruning and no measuring artifact: Sometimes, the algorithms read important pruning information from their intermediate data structures earlier, sometimes later, depending on how the intermediate data structures have grown. Like in the column scalability experiment, the relative performance of the different algorithms is, apart from the curves' noise, relatively stable and no algorithm takes a particular advantage from longer datasets.

## 5.3 NIND experiments

In this section, we evaluate the runtimes of the six n-ary IND discovery algorithms MIND, BINDER, ZIGZAG, FIND2, MIND2, and FAIDA on the datasets described in Table 3. The experiments will show that the bottom-up lattice traversal algorithm BINDER is the most efficient approach for n-ary IND discovery on real-world datasets, but the bi-directional lattice traversal approach FIND2 can outperform BINDER on certain synthetic datasets that have many NINDS of high arity; however, if false positives are acceptable, FAIDA is several orders of magnitude faster than all other algorithms and, hence, the most efficient discovery approach.

Note that the optimized, disk-backed validation strategies of BINDER, FAIDA, and MIND2 do have a problem that we cannot show in the runtime charts: Their memory consumption on disk is significant and can become much larger than the input dataset itself. The number of value pairs as well as the number of attribute combinations that BINDER and FAIDA need to consider grows exponentially; the coordinate files of MIND2 grow only quadratically with the length of the data (if the attributes share many values), but by simultaneously opening one file handle for every UIND, MIND2 cannot processes datasets with more UINDS than the operating system's file handle limit. Solving these issues is still a topic for future work.

The algorithms MIND, ZIGZAG, FIND2, and MIND2 require the first (and second) level of INDS as input. Although different algorithms can provide these inputs, we use SPIDER for the UINDS and MIND for the binary INDS due to their SQL-nature (SQL-based sorting and SQL candidate validations), which is similar to the nature of the NIND algorithms. We add the initial UIND discovery times to the NIND discovery time so that all reported runtimes reflect the total time needed to discover the n-ary INDS of all arities. BINDER and FAIDA compute the UINDS themselves. For parameterization, we used the algorithm's proposed default settings, which are an $\epsilon$ of 1.0 for ZIGZAG and ten buckets per attribute for BINDER.

**Runtimes on different datasets.** Table 5 shows the runtimes for all NIND discovery algorithms. Because most of the INDS found in the real-world datasets have low arity, we added the synthetically generated GHAIND dataset to evaluate how well the algorithms deal with deep search spaces, i.e., many INDS of high arity. The generated dataset contains several 5-, 6-, and 7-ary maximum INDS.

The measurements show that MIND's bottom-up lattice traversal performs well on all datasets. It though fails on long datasets (e.g. TPC-H 10) where the SQL-validation queries become very expensive, and it fails on datasets with very many candidate NINDS (e.g. LOD and MUSICBRAINZ). BINDER is a bit faster than MIND on most datasets due to its own, not SQL-based validation technique. However, if the number of NIND candidates is very small (see TESMA) or if the data is dense and contains many different value-combinations (see GHAIND and TPC-H 1), BINDER's validation is slower than simple SQL queries, because the validation requires the algorithm to generate many, often very large and, hence, I/O intensive value partitions. The expensive generation of these partitions does actually pay off on larger datasets, because it's performance scales better than the performance of SQL queries (see TPC-H 10). Because ZIGZAG uses an $\epsilon$ of 1.0, the algorithm aggressively prefers top-down over bottom-up search. If the NINDS are of high arity (see WIKIRANK and GHAIND), this strategy pays of; but if the UINDS are actually of low cardinality, which is true for most real-world datasets, the algorithm performs worse than simple buttom-up approaches. The same is true for FIND2, but the algorithm's clique finding approach for selecting high arity IND candidates is more precise than ZIGZAG's optimistic border estimation (see ENSEMBL). Due to its SQL-based candidate validation, though, FIND2 fails to compute large datasets and those with very many NINDS. MIND2 also performs very well on datasets with high-arity INDS (see GHAIND) due to its implicit candidate generation that avoids traversing the otherwise large search space. The algorithm however struggles processing most other datasets, because the amount of UIND-coordinates grows quadratically with the data if many columns of UINDS share the same values. It is also not applicable to CENSUS, because it requires at least two relations as input. The FAIDA algorithm clearly outperforms all other NIND algorithms with its naive bottom-up search space traversal and its various approximation techniques. The algorithm generates a considerable amount of candidates, but it is able to validate them very quickly. Due to the sampling, the summary data structures, and their efficient combination, FAIDA processes even high-arity IND candidates efficiently. Still, FAIDA also failed to process LOD and MUSICBRAINZ in our time limit due to their enormous candidate space. The algorithm's precision in theory reduces with every higher lattice level, but all its NIND results for the tested datasets were correct.

**Table 5: NIND performance on real-world datasets (minutes)**

| Datasets | MIND | BINDER | ZIGZAG | FIND2 | MIND2 | FAIDA |
|---|---|---|---|---|---|---|
| SCOP | 0.36 | 0.30 | 0.50 | 0.37 | 1.85 | **0.07** |
| CATH | 3.15 | 4.18 | 3.23 | 3.24 | 29.16 | **0.05** |
| CENSUS | 2.01 | 0.68 | 3.37 | 2.28 | n.a. | **0.16** |
| WIKIPEDIA | 1.48 | 1.40 | 1.54 | 1.50 | 1.47 | **1.11** |
| BIOSQL | 5.36 | 3.65 | 4.88 | 5.51 | >4h | **0.96** |
| WIKIRANK | 8.89 | 8.10 | 2.96 | 2.99 | >4h | **1** |
| LOD | >4h | >4h | >4h | >4h | >4h | >4h |
| ENSEMBL | 7.13 | 6.31 | 206.94 | 8.33 | >4h | **1.90** |
| GHAIND | 27.94 | 177.65 | 13.13 | 13.28 | 13.45 | **3.97** |
| TESMA | 3.38 | 4.35 | 3.39 | 3.39 | 8.79 | **3.1** |
| TPC-H 1 | 7.01 | 9.97 | 12.37 | 7.49 | >4h | **2.37** |
| TPC-H 10 | >4h | 121.09 | >4h | >4h | >4h | **22.65** |
| MUSICBR. | >4h | >4h | >4h | >4h | >4h | >4h |

**Runtimes on different column numbers.** To evaluate the algorithms' scalability with the number of columns, we start with the `artist` and `artist_alias` relations of MUSICBRAINZ (35 attributes



RIGHTSLINK

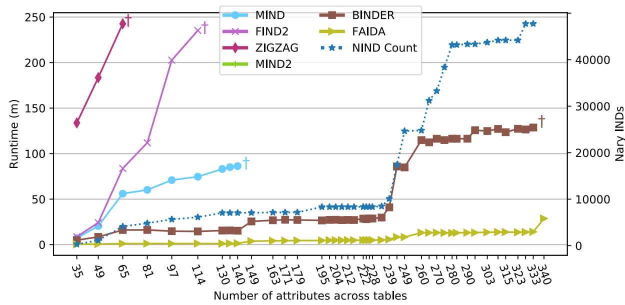


**Figure 5: Attribute scalability of nInd algorithms**

and 175 nInds) and successively add additional relations following logical join-paths in MusicBrainz: `artist`, `artist_ alias`, `area`, `area_alias`, `label`, `label_alias` etc. Figure 5 shows the runtimes for the six nInd algorithms. The † again denotes that an algorithm did not finish within the time limit of four hours. Mind2 does not show up in the graph, because the algorithm already exceeds the time limit for the first relation pair due its large I/O overhead. We also see all three SQL-based algorithms, i.e., Mind, ZigZag, and Find2, exceed the four hour time limit early on, because their validation queries are expensive. Mind survives longer than ZigZag and Find2, because most real-world Inds are small and ZigZag and Find2 overestimate their size. Mind suddenly fails when adding the 10th relation, because that relation is large and generates many, mostly false nInd candidates. With its all-column hash join, Binder can finish this relation in still less than 30 minutes. Binder then exceeds the time limit at 340 attributes. Due to its approximation techniques, Faida again shows the best performance albeit actually making a few mistakes on this dataset.

## 6 CONCLUSION AND FUTURE WORK

In this research project, we implemented and evaluated thirteen state-of-the-art algorithms for the discovery of unary and n-ary inclusion dependencies. With this paper, we provide a survey and detailed analysis of all important Ind discovery strategies. We measured the algorithms’ execution times on several datasets and investigated their scalability.

In summary, we can draw the following conclusions: An in-memory algorithm, such as DeMarchi, is the most efficient discovery approach as long as the data fits into main memory – even if the validation procedure is not perfect. The Binder algorithm is the most efficient general purpose solution for unary and n-ary Ind discovery. If the risk of getting a false positive Ind is tolerable, Faida can discover all unary and n-ary Inds even faster than Binder. However, the by far fastest approach for unary Ind discovery is Sindy, if at least 8 cores are available.

Our experiments also showed that the discovery of Inds is still a challenge that requires approximation and parallelization to be solved. Efficient techniques to handle intermediate data in memory and on disk are still needed, because even the fastest algorithms (i.e., Binder, Faida, and Sindy) tend to exhaust these resources. Furthermore, most use cases require only a certain subset of all discoverable Inds. The selection of *relevant* Inds at discovery time could therefore further improve the efficiency of Ind profiling.

## REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. 2018. *Data Profiling*. Morgan & Claypool Publishers.
[2] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. 487–499.
[3] Jana Bauckmann, Ulf Leser, Felix Naumann, and Veronique Tietz. 2007. Efficiently Detecting Inclusion Dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1448–1450.
[4] Siegfried Bell and Peter Brockhausen. 1995. Discovery of Data Dependencies in Relational Databases. In *Statistics, Machine Learning and Knowledge Discovery in Databases, ML–Net Familiarization Workshop*. 53–58.
[5] Thomas Bläsius, Tobias Friedrich, and Martin Schirneck. 2016. The Parameterized Complexity of Dependency Detection in Relational Databases. In *International Symposium on Parameterized and Exact Computation (IPEC)*, Vol. 63. 6:1–6:13.
[6] Marco A Casanova, Ronald Fagin, and Christos H Papadimitriou. 1984. Inclusion dependencies and their interaction with functional dependencies. *J. Comput. System Sci.* 28, 1 (1984), 29–59.
[7] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. 2002. Efficient Algorithms for Mining Inclusion Dependencies. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 464–476.
[8] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. 2009. Unary and n-ary inclusion dependency discovery in relational databases. *Journal of Intelligent Information Systems* 32, 1 (2009), 53–73.
[9] Fabien De Marchi and Jean-Marc Petit. 2003. Zigzag: a new algorithm for mining large inclusion dependencies in databases. In *Proceedings of the International Conference on Data Mining (ICDM)*. 27–34.
[10] Jarek Gryz. 1998. Query folding with inclusion dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 126–133.
[11] Martti Kantola, Heikki Mannila, Kari-Jouko Räihä, and Harri Siirtola. 1992. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems* 7, 7 (1992), 591–607.
[12] Andreas Koeller and Elke A Rundensteiner. 2003. Discovery of high-dimensional inclusion dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 683–685.
[13] Henning Köhler, Uwe Leck, Sebastian Link, and Xiaofang Zhou. 2016. Possible and certain keys for SQL. *VLDB Journal* 25, 4 (2016), 571–596.
[14] Sebastian Kruse, Thorsten Papenbrock, Christian Dullweber, Moritz Finke, Manuel Hegner, Martin Zabel, Christian Zöllner, and Felix Naumann. 2017. Fast approximate discovery of inclusion dependencies. In *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW)*. 207–226.
[15] Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. 2015. Scaling out the discovery of inclusion dependencies. In *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW)*. 445–454.
[16] Mark Levene and Vincent. 2000. Justification for inclusion dependency normal form. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* (2000).
[17] 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.
[18] Renée J Miller, Mauricio A Hernández, Laura M Haas, Ling-Ling Yan, CT Howard Ho, Ronald Fagin, and Lucian Popa. 2001. The Clio project: managing heterogeneity. *SIGMOD Record* 30, 1 (2001), 78–83.
[19] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. 2015. Data Profiling with Metanome (demo). *Proceedings of the VLDB Endowment (PVLDB)* 8, 12 (2015), 1860–1863.
[20] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. 2015. Divide & conquer-based inclusion dependency discovery. *Proceedings of the VLDB Endowment (PVLDB)* 8, 7 (2015), 774–785.
[21] 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 SIGMOD Workshop on the Web and Databases (WebDB)*.
[22] Nuhad Shaabani and Christoph Meinel. 2015. Scalable Inclusion Dependency Discovery. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*. 425–440.
[23] Nuhad Shaabani and Christoph Meinel. 2016. Detecting Maximum Inclusion Dependencies without Candidate Generation. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*. 118–133.
[24] Nuhad Shaabani and Christoph Meinel. 2018. Improving the Efficiency of Inclusion Dependency Detection. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. ACM, 207–216.
[25] Fabian Tschirschnitz, Thorsten Papenbrock, and Felix Naumann. 2017. Detecting Inclusion Dependencies on Very Many Tables. *ACM Transactions on Database Systems (TODS)* 1, 1 (2017), 1–30.
[26] 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 (PVLDB)* 3, 1-2 (2010), 805–814.



RIGHTSLINK