Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms

Falco Dürsch¹, Axel Stebner¹, Fabian Windheuser¹, Maxi Fischer¹, Tim Friedrich¹, Nils Strelow¹, Tobias Bleifuß², Hazar Harmouch², Lan Jiang², Thorsten Papenbrock², Felix Naumann²
Hasso Plattner Institute, University of Potsdam, Germany
¹firstname.lastname@student.hpi.de, ²firstname.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

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 algorithms. 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
We support the choice of the right algorithm for a specific data-set 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.

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 unIND 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, \( \text{Ind} \)s 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 \( \text{Ind} \) \( \sigma = R_i[X] \subseteq R_j[Y] \) (short: \( X \subseteq Y \)) is satisfied iff \( \forall u \in r_i, \exists v \in r_j \) such that \( u[X] = v[Y] \). Note that \( R[X] \) and \( u[X] \) denote the projection of schema \( R \) and record \( u \) on the attributes \( X \).

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

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

Figure 1 shows an example of two relations, namely Movies and Showings. The following unary and \( n \)-ary \( \text{Ind} \)s are valid, the last one being the only maximum \( \text{Ind} \):

\[
\begin{align*}
\text{Showings}[\text{Movie}] & \subseteq \text{Movies}[\text{Title}] \\
\text{Showings}[\text{Length}] & \subseteq \text{Movies}[\text{Time}] \\
\text{Showings}[\text{Movie}, \text{Length}] & \subseteq \text{Movies}[\text{Title}, \text{Time}]
\end{align*}
\]

Candidate validation. Every unIND discovery algorithm, except BELL AND BROCKHAUSEN, proposes its own, optimized candidate validation 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 \( \text{Ind} \) 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 \( \text{Ind} \) 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 \( \text{Ind} \)s when facing null values. Second, outer joins are highly optimized in most RDBMSs, 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, RDBMSs 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 unIND discovery [4]. Their main idea was to use logical inference over already discovered \( \text{Ind} \)s 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
### 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 UND candidates. Because SQL-based candidate validation is 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 UND candidates $A \subseteq B$ 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 UND $A \subseteq B$ as a directed edge $(A, B)$ into a graph structure. For every inserted UND, it tries to infer the (in-)validity of other UNDs 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 UND candidate $A \subseteq C$ from the candidate list, because this candidate needs to be true as well. Since the candidate list of UNDs is sorted by their dependent attribute, the algorithm can, when inserting an edge $(B, A)$, also prune any $B \subseteq C$ where $A \subseteq C$ has no edge in the graph, because these UNDS must be invalid too.

**DeMarchi**. The DeMarchi 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 ($\Rightarrow$ dependent $A$) is mapped to the set of all other attributes ($\Rightarrow$ 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 UND $A \subseteq B$ 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 UND 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 $A \subseteq B$ where $A$ contains value $v$ but $B$ does not. All candidates that survive the intersection process are true UNDs and are reported as $\{A \subseteq B \mid B \in Ref_A\}$.

**Spider**. The UND 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 UND 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 UND candidates,

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Year</th>
<th>Validation</th>
<th>Storage</th>
<th>Special</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bell and Brockhausen</td>
<td>1995</td>
<td>SQL outer join</td>
<td>in-memory</td>
<td>extensive pruning with statistics</td>
</tr>
<tr>
<td>DeMarchi</td>
<td>[7, 8]</td>
<td>2002 hash join</td>
<td>in-memory</td>
<td>all-column hash join</td>
</tr>
<tr>
<td>S-INDD</td>
<td>[22]</td>
<td>2015 sort-merge join</td>
<td>disk-backed</td>
<td>all-column sort-merge join and attribute clustering</td>
</tr>
<tr>
<td>Binder</td>
<td>[20]</td>
<td>2015 hash join</td>
<td>disk-backed</td>
<td>all-column hash join and divide &amp; conquer</td>
</tr>
<tr>
<td>Sindy</td>
<td>[15]</td>
<td>2015 map-reduce</td>
<td>disk-backed</td>
<td>MapReduce-style parallel/distributed discovery</td>
</tr>
<tr>
<td>Faida</td>
<td>[14]</td>
<td>2017 HLL intersect</td>
<td>disk-backed</td>
<td>approximation via hashing, sampling, and sketching</td>
</tr>
</tbody>
</table>
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 \( U_c \) (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-merges the partition 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 \( R_{\bar{A}} \) and the dense index is used to select the attribute sets \( U_c \) 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 InD, 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 \( X \subseteq Y \) is valid, if the set cardinality of \( Y \), which is approximated by HLL, is equal to the joint set cardinality of \( X \) and \( Y \), which is \( \text{HLL}_{XY} \). 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.
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 α BloomFilters of length m into a m × 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-IND 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-IND, 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-monotonicity 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 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 Pettig proposed shortly after Mind another nIND algorithm, ZigZag, which estimates an optimistic and a pessimistic border of IND candidates and alternates between 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).

![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.](image-url)
The algorithm starts by validating the maximal candidates from level \( k \). The algorithm uses the true \( \text{INdS} \) from level \( k-1 \). In short, \( X' \subseteq Y' \) is a valid candidate if all its implied \( \text{INdS} X \subseteq Y \) with \( |X'| = |X| - 1 \) and \( |Y'| = |Y| - 1 \) are true \( \text{INdS} \) 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 \textsc{Binder} and \textsc{Faida}.

\textbf{ZigZag.} The ZigZag algorithm [9] also uses the candidate lattice as a search space model, but it alternates between using the \textsc{apriori-gen}-based bottom-up traversal and a top-down traversal, ‘zigzagging’ between bottom and top \( \text{INdS} \) candidates. The algorithm takes not only the unary \( \text{INdS} \) as input but all \( \text{INdS} \) of the first \( k \) lattice levels; as proposed by the authors, we use \( k = 2 \) as default. The valid \( \text{INdS} \) are used to initialize a negative border, a positive border, and an optimistic positive border. The negative border contains all known \textit{minimal invalid} \( \text{INdS} \) and the positive border contains all \textit{maximal valid} \( \text{INdS} \). Whenever an \( \text{INdS} \) candidate is tested to be true or false, it is added to the corresponding border. The optimistic positive border, in contrast, contains the \textit{maximal candidate} \( \text{INdS} \), which are derived as largest compositions of valid, positive cover \( \text{INdS} \).

During \( \text{INdS} \) discovery, ZigZag jumps back and forth between the negative/positive borders and the optimistic positive border. The algorithm starts by validating the maximal \( \text{INdS} \) 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 \( \text{INdS} \), i.e., the \( g'_{\text{IND}} \) error measure [17]. Because ZigZag needs the number of violating rows, the SQL queries cannot use early termination when validating \( \text{IND} \) candidates. In case the error measure is below a certain threshold \( \epsilon \), the algorithm checks all direct subset \( \text{INdS} \) 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 \textsc{apriori-gen}-style.

\textbf{Find2.} The bi-directional \( \text{INdS} \) discovery algorithm \textsc{Find2} [12] translates the problem of \( \text{INdS} \) discovery to the discovery of cliques in \( k \)-uniform-hypergraphs, where each edge must connect exactly \( k \) nodes. The given \( \text{UNdS} \) are the nodes in these graphs and the \( \text{INdS} \) at level \( k \) of the search space lattice are the \( k \)-hyperedges connecting the \( \text{UNdS} \) from which they are composed. Each level of the lattice, which is a set of \( k \)-ary \( \text{INdS} \), is represented as a \( k \)-uniform-hypergraph. Just like ZigZag, \textsc{Find2} takes the first \( k \) levels of true \( \text{INdS} \) 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 minimum \( \text{INdS} \) candidates. Finding maximum cliques in \( k \)-uniform-hypergraphs is NP-hard, but \textsc{Find2}’s HyperClique algorithm solves the problem efficiently for sparse graphs with only a few cliques, which is usually the case for \( \text{INdS} \) discovery scenarios. After generating all cliques, i.e., maximum \( \text{INdS} \) candidates, \textsc{Find2} validates them using SQL validation queries. If an \( \text{INdS} \) candidate evaluates to true, the algorithm has discovered a maximal \( \text{INdS} \) and prunes all implied candidates from the search space (downward-pruning); otherwise, the candidate is broken into \( \text{INdS} \) 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 \( k + 1 \). \textsc{Find2} terminates if no new cliques are found.

\textbf{MIND2.} The \( \text{INdS} \) discovery algorithm \textsc{MIND2} [23] requires all valid \( \text{UNdS} \) as input and then discovers all maximum inclusion dependencies by deriving them from so-called \( \text{UNdS} \) coordinates. The coordinates of a \( \text{UNdS} \ A \subseteq B \) 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 \( j \). \textsc{MIND2} groups these \( \text{UNdS} \) coordinates by their \( i \)-index resulting in a mapping of \( i \)-indices to lists corresponding \( j \)-indices. By transforming value lists into coordinates, \textsc{MIND2} formulates an index structure for the \( \text{INdS} \) 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, \textsc{MIND2} stores the coordinate mapping (sorted by \( i \)-index) for each \( \text{UNdS} \) in a separate file on disk. It then infers an initial set of maximum \( \text{INdS} \) from the given \( \text{UNdS} \) and opens a file reader to each \( \text{UNdS} \) coordinate file.

The intuition for the next step is the following: \( A C \subseteq BD \) is valid, iff for all \( i \) in \( A \subseteq B \) and \( C \subseteq D \), \( i \)-coordinate files of the respective \( j \)-lists is not empty. In other words, \( \forall r_i[AC] \exists r_j[BD] : r_i[AC] = r_j[BD] \). \textsc{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 \( \text{INdS} \) at position \( i \) are used to update, i.e., prune the working set of maximum \( \text{INdS} \). \textsc{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 \( \text{INdS} \).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Year</th>
<th>Validation</th>
<th>Pruning</th>
<th>Traversal</th>
<th>Special</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIND</td>
<td>[8]</td>
<td>2002</td>
<td>SQL outer join</td>
<td>upwards</td>
<td>bottom-up</td>
</tr>
<tr>
<td>ZigZag</td>
<td>[9]</td>
<td>2003</td>
<td>SQL outer join</td>
<td>up-/downwards</td>
<td>bi-directional</td>
</tr>
<tr>
<td>Find2</td>
<td>[12]</td>
<td>2003</td>
<td>SQL outer join</td>
<td>up-/downwards</td>
<td>bi-directional</td>
</tr>
<tr>
<td>Binder</td>
<td>[20]</td>
<td>2015</td>
<td>hash join</td>
<td>upwards</td>
<td>bottom-up</td>
</tr>
<tr>
<td>Mind2</td>
<td>[23]</td>
<td>2016</td>
<td>sort-merge join</td>
<td>upwards</td>
<td>inference</td>
</tr>
<tr>
<td>Faida</td>
<td>[14]</td>
<td>2017</td>
<td>HLL intersect</td>
<td>upwards</td>
<td>bottom-up</td>
</tr>
</tbody>
</table>
5 EVALUATION

This section analyzes and compares the state-of-the-art \textsc{Ind} discovery algorithms of the previous sections. After introducing our experimental setup, we first present the results for \textsc{uInd} discovery and then those for \textsc{nInd} discovery.

5.1 Experimental setup

We implemented all algorithms for our \textsc{Metanome} data profiling framework\footnote{http://www.metanome.de}, which defines standard interfaces for different kinds of profiling algorithms [19]. All common tasks, such as input parsing, result formatting, performance measuring, and algorithm parameterization, 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\footnote{https://lpi.de/naumann/projects/repeatability/data-profiling/metanome-ind-algorithms.html}.

Datasets. We evaluate all algorithms on several real-world and three synthetic datasets, namely TPC-H, \textsc{Tesma}, and \textsc{ghaIND}. Table 3 lists all datasets and their characteristics. \textsc{Faida}'s approximate \textsc{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 \textsc{nInd}s 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 \textsc{Ind}s are still an open research topic and the focus of this evaluation is on performance rather than \textsc{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 \textsc{nInd}s with all other \textsc{Ind}s.

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 \textsc{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 \textsc{uInd} experiments

We first evaluate the runtime of the unary \textsc{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 \textsc{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, \textsc{Binder} is the most efficient approach in general; however, \textsc{Faida} is faster than \textsc{Binder}, if we can tolerate false positives in the results, and \textsc{SIndy} is even faster than \textsc{Faida} (and, hence, the fastest algorithm over all), if the hardware has at least eight cores.

Table 3: Dataset characteristics

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Attributes</th>
<th>\textsc{uInds}</th>
<th>\textsc{nInds}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textsc{Scop}</td>
<td>16 MB</td>
<td>22</td>
<td>39</td>
<td>36</td>
</tr>
<tr>
<td>\textsc{Cath 4.0}</td>
<td>16 MB</td>
<td>25</td>
<td>50</td>
<td>81</td>
</tr>
<tr>
<td>\textsc{Census}</td>
<td>112 MB</td>
<td>42</td>
<td>39</td>
<td>89</td>
</tr>
<tr>
<td>\textsc{Wikipedia}</td>
<td>540 MB</td>
<td>14</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>\textsc{BioSql}</td>
<td>560 MB</td>
<td>77</td>
<td>348</td>
<td>507</td>
</tr>
<tr>
<td>\textsc{WikiRank}</td>
<td>697 MB</td>
<td>29</td>
<td>15</td>
<td>103</td>
</tr>
<tr>
<td>\textsc{Lod}</td>
<td>830 MB</td>
<td>41</td>
<td>258</td>
<td>Unknown</td>
</tr>
<tr>
<td>\textsc{Ensembl}</td>
<td>836 MB</td>
<td>130</td>
<td>364</td>
<td>100</td>
</tr>
<tr>
<td>\textsc{Tesma}</td>
<td>1 GB</td>
<td>114</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>\textsc{TPC-H 1}</td>
<td>1 GB</td>
<td>61</td>
<td>96 (99)</td>
<td>8</td>
</tr>
<tr>
<td>\textsc{TPC-H 10}</td>
<td>10 GB</td>
<td>61</td>
<td>97</td>
<td>11</td>
</tr>
<tr>
<td>\textsc{Musicbrainz}</td>
<td>27 GB</td>
<td>1054</td>
<td>49,829 (49,867)</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

\textbf{Runtimes on different datasets.} Table 4 lists the measured runtimes of all nine \textsc{uInd} discovery algorithms for all evaluation datasets. Despite its clever candidate pruning strategies, \textsc{Bell} and \textsc{Brockhausen} performs worst, being the only approach that relies on SQL. \textsc{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), \textsc{DeMarchi}'s redundant set intersections are much less expensive than any disk writing costs; however, on large datasets, such as \textsc{Musicbrainz}, the redundancy in the candidate validations is more expensive than the disk I/O. \textsc{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 \textsc{s-Indd} algorithm does not have \textsc{Spider}'s file handle limitation. It is, though, not a clear performance improvement over \textsc{Spider}: \textsc{s-Indd} is sometimes faster, because the algorithm explicitly de-duplicates the attribute lists \textit{Uv}, which prunes many redundant list-intersect operations; \textsc{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 \textsc{Binder} algorithm outperforms \textsc{Spider} and \textsc{s-Indd} on most datasets, because its hash-join approach is more efficient than the sort-merge-join approach \textit{O(n)} instead of \textit{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 \textsc{s-Indd}, but it reads and writes less data from/to disk. \textsc{s-Indd++} improves the disk I/O of \textsc{s-Indd} and, hence, achieves better runtimes; its overall performance still slightly falls behind \textsc{Binder}. The approximate \textsc{Ind} discovery algorithm \textsc{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 \textsc{Faida}
Table 4: uIND performance on real-world datasets (minutes)

<table>
<thead>
<tr>
<th>Datasets</th>
<th>B&amp;B</th>
<th>DeMarchi</th>
<th>SPIDER</th>
<th>S-INDD</th>
<th>BINDER</th>
<th>S-INDD++</th>
<th>FAIDA</th>
<th>MANY</th>
<th>1 worker</th>
<th>8 workers</th>
<th>32 workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOPE</td>
<td>0.14</td>
<td>0.04</td>
<td>0.08</td>
<td>0.09</td>
<td>0.07</td>
<td>0.08</td>
<td>0.05</td>
<td>0.08</td>
<td>0.49</td>
<td>0.35</td>
<td>0.36</td>
</tr>
<tr>
<td>Cath</td>
<td>0.11</td>
<td>0.02</td>
<td>0.05</td>
<td>0.05</td>
<td>0.04</td>
<td>0.05</td>
<td>0.03</td>
<td>0.04</td>
<td>0.45</td>
<td>0.38</td>
<td>0.37</td>
</tr>
<tr>
<td>Census</td>
<td>1.05</td>
<td>0.09</td>
<td>0.15</td>
<td>0.17</td>
<td>0.14</td>
<td>0.14</td>
<td>0.12</td>
<td>0.14</td>
<td>0.66</td>
<td>0.38</td>
<td>0.38</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>&gt;4h</td>
<td>1.02</td>
<td>1.47</td>
<td>1.56</td>
<td>1.22</td>
<td>1.25</td>
<td>1.11</td>
<td>1.34</td>
<td>1.93</td>
<td>0.59</td>
<td>0.51</td>
</tr>
<tr>
<td>BioSql</td>
<td>4.98</td>
<td>0.76</td>
<td>1.30</td>
<td>1.48</td>
<td>1.41</td>
<td>1.15</td>
<td>0.9</td>
<td>1.15</td>
<td>2.14</td>
<td>0.63</td>
<td>0.51</td>
</tr>
<tr>
<td>WikiBank</td>
<td>2.90</td>
<td>0.73</td>
<td>1.53</td>
<td>1.44</td>
<td>1.23</td>
<td>1.04</td>
<td>0.87</td>
<td>0.97</td>
<td>2.33</td>
<td>0.85</td>
<td>0.55</td>
</tr>
<tr>
<td>LOD</td>
<td>0.34</td>
<td>0.25</td>
<td>0.45</td>
<td>0.41</td>
<td>0.30</td>
<td>0.37</td>
<td>0.69</td>
<td>0.36</td>
<td>1.81</td>
<td>0.68</td>
<td>0.46</td>
</tr>
<tr>
<td>Ensembl</td>
<td>23.52</td>
<td>2.1</td>
<td>3.04</td>
<td>3.70</td>
<td>2.39</td>
<td>3.05</td>
<td>1.76</td>
<td>2.85</td>
<td>3.50</td>
<td>0.91</td>
<td>0.62</td>
</tr>
<tr>
<td>Tesma</td>
<td>&gt;4h</td>
<td>3.66</td>
<td>3.39</td>
<td>4.75</td>
<td>4.27</td>
<td>6.13</td>
<td>3.11</td>
<td>3.71</td>
<td>2.57</td>
<td>0.69</td>
<td>0.54</td>
</tr>
<tr>
<td>TPC-H 1</td>
<td>17.79</td>
<td>1.96</td>
<td>3.88</td>
<td>3.58</td>
<td>2.96</td>
<td>2.72</td>
<td>2.08</td>
<td>3.00</td>
<td>5.55</td>
<td>1.34</td>
<td>0.78</td>
</tr>
<tr>
<td>TPC-H 10</td>
<td>&gt;4h</td>
<td>22.81</td>
<td>44.43</td>
<td>36.54</td>
<td>28.21</td>
<td>28.39</td>
<td>19.19</td>
<td>57.78</td>
<td>10.22</td>
<td>4.89</td>
<td></td>
</tr>
<tr>
<td>Musicbrainz</td>
<td>&gt;4h</td>
<td>136.03</td>
<td>61.42</td>
<td>106.26</td>
<td>45.69</td>
<td>71.22</td>
<td>27.67</td>
<td>105.49</td>
<td>175.97</td>
<td>30.22</td>
<td>15.38</td>
</tr>
</tbody>
</table>

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
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 unInd, Mind2 cannot process datasets with more unInds 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 unInds 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 unInd 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 unInds 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 Musicbrainz2). 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 off; but if the unInds are actually of low cardinality, which is true for most real-world datasets, the algorithm performs worse than simple bottom-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 unInd-coordinates grows quadratically with the data if many columns of unInds 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.

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

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Mind</th>
<th>Binder</th>
<th>ZigZag</th>
<th>Find2</th>
<th>Mind2</th>
<th>Faida</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-H 1</td>
<td>1.48</td>
<td>1.40</td>
<td>1.54</td>
<td>1.50</td>
<td>1.47</td>
<td>1.11</td>
</tr>
<tr>
<td>TPC-H 10</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>1.90</td>
</tr>
<tr>
<td>Musicbrainz</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>22.65</td>
</tr>
<tr>
<td>TPC-H 1</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>23.77</td>
</tr>
<tr>
<td>TPC-H 10</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
<td>&gt;4h</td>
</tr>
</tbody>
</table>

Table 5: nInd performance on real-world datasets (minutes)
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 runtime for the six NIND algorithms. The \( g \) again denotes that an algorithm did not finish within the time limit of four hours. NIND2 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. BIND 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