SPIND: Scalable Partial Inclusion Dependency Discovery

Jakob Leander Müller
Marburg University
Marburg, Germany
me@jakob-l-m.de

Marcian Seeger
Marburg University
Marburg, Germany
marcian.seeger@uni-marburg.de

Thorsten Papenbrock
Marburg University
Marburg, Germany
papenbrock@informatik.uni-marburg.de

Abstract

Inclusion dependencies (INDs) play an important role in relational database theory because they describe foreign key relationships and serve a variety of data engineering tasks, such as data integration, query optimization, and integrity checking. Though, relational datasets often do not store their INDs explicitly due to technological restrictions or limited metadata maintenance capabilities. For this reason, data profiling algorithms are used to discover INDs dynamically when these are needed. The profiling of INDs in large datasets is, however, still very challenging, especially if these datasets are expected to contain a few errors so that, in fact, *partial* INDs (pINDs) need to be discovered. A pIND describes a subset relationship with a specific, not necessarily complete overlap and can, therefore, tolerate (smaller) data inaccuracies, which is particularly helpful in data cleaning and data integration scenarios.

In this paper, we present SPIND, a novel algorithm for the efficient and scalable discovery of both exact and partial INDs. SPIND combines new strategies for (p)IND discovery with known IND profiling practices, such as multi-column sort-merge joins and aggressive parallelization. With a special, chunking-based global sortation technique that pre-compresses multiple sortations into possibly few files, SPIND discovers (p)INDs much more efficiently than existing state-of-the-art IND profiling approaches and, therefore, also in much larger and more complex datasets. Our evaluation on different real-world and synthetic datasets shows that SPIND outperforms current state-of-the-art algorithms, such as AINDD, BINDER and SPIDER, in some cases by several orders of magnitude.

Keywords

Inclusion Dependencies, Database Constraints, Data Profiling, Data Dependencies, Structural Metadata, Data Linkage

1 Complexity of (p)IND discovery

Structural metadata are statements about schema elements. They can describe not only the properties of individual schema attributes, such as data types or value ranges, but also relationships between different attributes, such as order dependencies [38, 46] or matching dependencies [37, 52]. An *inclusion dependency (IND)* $A \subseteq B$ is a particularly important type of structural metadata that describes a set-superset relationship: All values of a schema element A are included in the values of another schema element B [7]. In this paper, we consider relational attributes as schema elements and their IND relationships as a particularly interesting structural property. INDs serve as integrity constraints in relational databases and support many popular data engineering use cases, such as schema design [28], data linkage [20, 40], query optimization [18], data quality assurance [14], schema mapping and

EDBT '26, Tampere (Finland)

Table 1: PokemonTypes (PT)

Name	Gender	Type
Pikachu	m/f	electric
Raichu	m/f	electric
Nidoran	m/f	poison
Dragonite	m/f	dragon
Mewtwo	⊥	psychic

Table 2: PokemonSizes (PS)

PName	PGender	PSize
Pikachu	m/f	0.4
Nidoran	m	0.5
Nidoran	f	0.4
Dragonite	m/f	2.2
Mewtwo	⊥	2.0

matching [19, 32] and many more [39]. Inclusion dependencies are also the basis of foreign key constraints [20, 40].

For various reasons, INDs are rarely stored explicitly with their datasets. These reasons include the typically large number of INDs, the high complexity of maintaining INDs, and technical limitation of storing structural metadata in, for example, CSV files. Hence, data profiling algorithms have been developed to (re-)discover (or mine) INDs on demand [3, 4, 9, 10, 22, 26, 27, 31, 36, 41–43, 49]. Current state-of-the-art profiling algorithms either efficiently discover exact n-ary INDs [36] or focus only on partial unary INDs [45]. Consequently, there is currently no solution for the automatic discovery of more complex *n-ary partial INDs*, even though n-ary keys are a common concept [15, 16]. In this paper, we propose the algorithm SPIND to fill this gap. SPIND overcomes current limitations with a more effective data representation on disk and a novel chunking-based global sortation technique, which solves the I/O bottlenecks of state-of-the-art algorithms, such as BINDER or SPIDER. Our algorithm also extends to discovery to partial INDs that indicate a set-superset relationships, in which only a portion, i.e., a specific percentage of the set-values are included in the superset. The pIND discovery takes a specific threshold ρ to find all pINDs with a value overlap of at least ρ. Discovering pINDs is a practical necessity in many data profiling scenarios to tolerate (minor) data errors; it is relevant for data cleaning [6], data integration [33] and many further use cases [30, 44].

Example: To illustrate the relevance of partial n-ary inclusion dependencies, we consider the example data integration scenario depicted in Table 1 and Table 2. The two tables list data about *Pokemon*, which are fantasy pocket monsters, with Type and Size information, respectively. The only exact IND in this example is PT[Gender] $\subseteq$ PS[PGender], which is not useful for our use case, because a join on Gender and PGender does not merge the data correctly. The semantically correct join is described by the binary pIND PT[Name,Gender] $\subseteq_{60\%}$ PS[PName,PGender]. As Table 2 shows, both the name and the gender are needed to describe a specific pokemon, because pokemon with the same name can have different properties. Furthermore, the partialization to, in this example, 60% is needed to tolerate the unique value "*Raichu*" in Table 1 and the gender differentiation of "*Nidoran*" in Table 2.

Contribution: In this paper, we present the novel (p)IND discovery algorithm SPIND that efficiently discovers all exact and/or partial n-ary inclusion dependencies. SPIND uses discovery techniques from the IND profiling algorithms BINDER [36]

10.48786/edbt.2026.40

and SPIDER [3], but adds the required features to discover also partial dependencies. This is made possible by a novel *counting scheme* that maintains repeated values and inclusion violations for both unary and n-ary pINDs. With SPIND, we furthermore propose a novel *chunking-based global sortation technique* that pre-compresses multiple sortations into possibly few files and, in this way, not only mitigates file handle and memory management weaknesses of existing approaches, but also improves the profiling performances significantly. For efficiency reasons, we devise novel, extremely compact *serialization formats* specifically designed for off-loading columnar data for (p)IND discovery on disk. SPIND is the first profiling algorithm for n-ary pINDs and our *exhaustive experiments* demonstrate its superiority over all competitors also in their restricted settings.

Structure: We first discuss related work (Section 2) and the foundations of pIND discovery (Section 3). We then introduce our novel (p)IND discovery algorithm SPIND (Section 4). Afterwards, we extend the two state-of-the-art algorithms SPIDER and BINDER into the algorithms pSPIDER and pBINDER enabling them to discover pINDs as well (Section 5). The extensions ensure a fair comparison with the SPIND algorithm in our evaluation that, finally, evaluates SPIDER, BINDER, their optimized versions, and SPIND on several datasets (Section 6).

2 Related Work

Inclusion dependencies are among the most researched types of structural metadata with well-known inference rules [7]. They provide the basis of *referential integrity* in the relational model [8] and it is proven that the discovery of INDs is PSPACE-complete if there is no limit on the size of the inclusion dependencies [7]. In this paper, we discuss how the three axioms *reflexivity*, *transitivity*, and *projection and permutation* of exact INDs relate to pINDs (Section 3.1). Because the terms *partial* and *simple* INDs have also been used to describe how null values are handled in IND discovery and SQL [25, 34], we define *pINDs* in the context of this line of research in Section 3 and discuss null handling in Section 3.2.

The BellAndBrockhausen [4] algorithm proposed a systematic IND discovery strategy that is based on a clever graph representation of the IND candidates as well as axiomatic and statistical candidate pruning rules. Meanwhile, more recent algorithms outperform this approach by orders of magnitude.

The DeMarchi [10] algorithm introduced an index-based all-column hash-join strategy to validate all unary IND candidates simultaneously. This approach improved the validation times significantly and, hence, the all-column join idea was adopted by different follow-up works. The SINDY [27] algorithm, for instance, translates the idea into a distributed transformation pipeline.

The SPIDER [3] algorithm uses a disk-backed, all-column sort-merge join with early termination to discover all unary INDs in a relational dataset. The validation idea is similar to the DeMarchi algorithm, but with a sort-based strategy and disk usage to avoid memory overflows. The discovery of pINDs with SPIDER was also discussed in the original publication, but never implemented or tested. In this paper, we improve SPIDER (see Section 5.1) and show that our proposed pIND discovery algorithm SPIND is still superior. After adding our lazy sorting, violation-aware validation and parallelization techniques also to this competitor, the chunking-based global sortation technique constitutes the main difference: While SPIDER uses a vertical partitioning (one file per column), SPIND uses horizontal chunking and merging (one file per relation).

The BINDER [36] algorithm translated the DeMarchi algorithm into a disk-backed, divide-and-conquer-based all-column hash-join. The algorithm's strategy of partitioning input datasets into smaller chunks, its improved hash-based validation technique, and its more aggressive pruning during validations make BINDER faster than SPIDER. BINDER can also discover n-ary INDs, but it does not consider pINDs and produces an exponentially growing number of files during n-ary IND discovery. In this paper, we improve BINDER and add support for pIND discovery to demonstrate that SPIND's novel profiling technique is faster.

The S-INDD [41] algorithm is an extension of SPIDER that merges the sorted files prior to the candidate validations to avoid file handle issues. The further optimized S-INDD++ [43] algorithms also added an improved version of BINDER's partitioning strategy to the IND discovery approach to maximize early stoppings gains. Because the average effectiveness of S-INDD++ has been measured to be somewhat between SPIDER and BINDER [12] and because the approach can discover only exact unary INDs, we do not cover it in our evaluations. Similar to S-INDD++, SPIND also merges sorted lists to reduce intermediate file numbers, albeit in a different way.

The AINDD [45] algorithm is a partition-aware unary pINDs discovery algorithm that limits and counts violations with early termination. It partitions all attributes into value-aligned *buckets*, similar to BINDER, and validates pIND candidates partition-wise with a three-layer filter: (I) a bit array, (II) a per-position distinct-value count, and (III) an exact value set. These filters define rules that validate or invalidate pINDs without scanning all tuples. Because AINDD focuses on only *unary* pINDs, it can sometimes discover them a bit faster than SPIND; our evaluation, though, shows that it is, in general, less efficient.

The SAWFISH [21] algorithm discovers *similarity inclusion dependencies (sINDs)*, which consider values as included if a sufficiently similar (not necessarily same) value is included in the superset. In this way, sINDs also tolerate data errors, but the necessary, in SAWFISH index-based similarity calculations are very expensive.

Approximate IND profiling algorithms, such as FAIDA [26] and the ideas of De Marchi and Petit [11], are orthogonal work to SPIND. An *approximate* IND is supposed to be exact, but due to the use of non-exact discovery strategies, such as Sampling, Bloomfilters or HyperLogLog structures, validity cannot be guaranteed; in contrast to pINDs, the overlap degree is neither known nor controllable.

Systems for data and join-discovery, such as Josie [54], Aurum [17] or MATE [13], discover column overlaps through, e.g., column similarities or ranking. These methods treat overlaps as undirected relationships that indicate semantic table matches rather than directed constraints; unlike pINDs, they do not consider completeness or exactness guarantees and, hence, answer an orthogonal question.

In contrast to all existing IND profiling algorithms, SPIND is the first algorithm that can discover *partial n-ary INDs*. BINDER [36] and FAIDA [26] can discover n-ary INDs, but without partialization and AINDD [45] discovers partial but not n-ary INDs. Our evaluation also shows that, apart from the partialization, SPIND's chunking-based global sortation technique makes it a competitive unary IND profiling algorithm and the fastest n-ary IND profiling algorithm.

3 Foundations of pINDs

This section follows the standard notation used by De Marchi et al. [31]. A relational instance r of a relational schema R is a collection of tuples, denoted by lower-case letters, such as u or v. Using an attribute list from R, denoted by upper-case letters $X \subseteq R$ or $Y \subseteq R$, we can specify a projection of r on X as $r[X]$ to select only a subset of attributes. Similarly, for tuples u, we write $u[X]$ to selection values from the tuple.

INDs represent relationships between attributes in a database schema. An IND specifies that the values of one relational projection are included in the values of another relational projection:

Definition 3.1 (Inclusion dependencies). Given two relational instances r_1 from R_1 and r_2 from R_2. An IND is defined as

$$r_1[X] \subseteq r_2[Y] \iff \forall\, u \in r_1, \exists\, v \in r_2 : u[X] = v[Y].$$

This condition can hold only if the cardinality of X is equal to the cardinality of Y, i.e., $|X| = |Y|$. We further call the left-hand side (here X) the *dependent* attribute(s) and the right-hand side (here Y) the *referenced* attribute(s).

Given a set of relational instances $\mathcal{I} = \{r_1, \ldots, r_n\}$, the IND profiling task is to enumerate all valid INDs $r_i[X] \subseteq r_j[Y]$ with $r_i, r_j \in \mathcal{I}$. With $m = \sum_{r_i \in \mathcal{I}} |R_i|$ attributes, this task is in $O(2^m \cdot m!)$) and, hence, an NP-complete problem [1]. It is even *"one of the first natural problems proven to be complete for the class W[3]"* [5], where W[k] is a hierarchy of parameterized hardness with larger k indicating greater intractability.

Partial INDs denote a relationship that holds for a certain threshold (ρ) over the relationship. More specifically, a pIND specifies that the values of one relational projection are *partialy* included in the values of another relational projection:

Definition 3.2 (Partial inclusion dependencies). A pIND denotes a partial inclusion relation and is written as $r_1[X] \subseteq_\rho r_2[Y]$ where $\rho \in (0, 1]$. For expressions of this form, we find two interpretations that differ in their consideration of duplicate values [3]: The *duplicate-aware* interpretation refers to lists of tuples (with list notation "$[\ldots]$") and, therefore, considers the cardinality of duplicate values. Hence, it defines a pIND is as

$$r_1[X] \subseteq_\rho r_2[Y] \iff \frac{|[u \in r_1[X] \mid \exists\, v \in r_2[Y] : u = v]|}{|r_1[X]|} \geq \rho.$$

The *duplicate-unaware* interpretation refers to sets of values (with set notation "$\{\ldots\}$") and defines a pIND as the set-based overlap

$$r_1[X] \subseteq_\rho r_2[Y] \iff \frac{|\{r_1[X]\} \cap \{r_2[Y]\}|}{|\{r_1[X]\}|} \geq \rho.$$

An IND *violation* refers to a left-hand side value $v \in r_1[X]$ that is not included in the right-hand side, i.e., $v \notin r_2[y]$. The maximum number of tolerable violations is $\lfloor (1-\rho) \cdot |[r_1[X]]| \rfloor$ for the *duplicate-aware* interpretation and $\lfloor (1-\rho) \cdot |\{r_1[X]\}| \rfloor$ for *duplicate-unaware*. Practically speaking, a pIND candidate is invalidated once the number of observed violations exceeds the respective maximum number of tolerable violations. For example, if $r_1[X] = [a, a, a, b]$ and $r_2[Y] = [a, c]$, then $r_1[X] \subseteq_{75\%} r_2[Y]$ is valid for the *duplicate-aware* interpretation, because $|[a, a, a]|/|[a, a, a, b]| = 3/4 = 75\%$, but it is invalid for the *duplicate-unaware* interpretation, because
$|\{a, a, a, b\} \cap \{a, c\}|/|\{a, a, a, b\}| = 1/2 < 75\%$. Our profiling algorithms support both interpretations and use *duplicate-aware* by default. The profiling task is to enumerate all valid pINDs w.r.t. a given ρ and a given duplicate interpretation. The possible values for ρ do not include 0, as this would mean that everything is a pIND of everything else, which is a trivial case. For mathematical reasons, we also assume $|r_1[X]| > 0$ and catch the case $|r_1[X]| = 0$ programmatically; $r_1[X] \subseteq_\rho r_2[Y]$ with $|r_1[X]| = 0$ is considered to be valid for any ρ. Because INDs are special pINDs with $\rho = 1$, discovering pINDs is at least as hard as discovering INDs and, hence, also NP-complete.

3.1 Properties of partial inclusion dependencies

Most strategies for IND discovery exploit IND properties that have been formalized as axioms [7, 35]. To also utilize them in this work, we now examine which axioms remain valid also for partial INDs. Related works on pIND discovery [3, 45] did not discuss these properties, presumably because their pruning strategies did not make use of them.

Reflexivity: For any $\rho \in (0, 1]$ the pIND $r[X] \subseteq_\rho r[X]$ is valid. This follows directly from Definition 3.2:

$$r[X] \subseteq_\rho r[X] \iff \frac{|r[X] \cap r[X]|}{|r[X]|} = \frac{|r[X]|}{|r[X]|} = 1 \geq \rho.$$

Because ρ is upper bound by 1, the pIND is always valid. Therefore, pINDs also adhere to reflexivity. Consequently, pINDs of the form $r[X] \subseteq_\rho r[X]$ are trivially valid and do not need to be tested. We note that Reflexivity is conditional on the null handling strategy, which we cover in Section 3.2.

Transitivity: Unlike INDs, pINDs generally do not adhere to transitivity. We demonstrate this claim with a proof-by-contradiction: Assume $r_1[X] = [1, 2, ..., 100]$, $r_2[Y] = [2, ..., 1000]$, and $r_3[Z] = [10, 11, ..., 1000]$. If transitivity would hold, then we should find that for any $\rho \in (0, 1]$ if $r_1[X] \subseteq_\rho r_2[Y]$ and $r_2[Y] \subseteq_\rho r_3[Z]$ are valid, then $r_1[X] \subseteq_\rho r_3[Z]$ is also always valid. However, for the given example and $\rho = 0.95$, we find a contradiction. Consequently, transitivity cannot be used as a pruning strategy to skip pIND test.

Projection: For any $\rho \in (0, 1]$, if $r_1[XY] \subseteq_\rho r_2[ZW]$ is valid, then also $r_1[X] \subseteq_\rho r_2[Z]$ and $r_1[Y] \subseteq_\rho r_2[W]$ are valid for any relations r_1 and r_2, attributes $X, Y \subseteq R_1$ and $Z, W \subseteq R_2$, and $X \cap Y = \emptyset$ and $Z \cap W = \emptyset$. To prove this, we assume $r_1[XY] \subseteq_\rho r_2[ZW]$ to be valid and consider only the portions $\tilde{r}_1 \subseteq r_1$ and $\tilde{r}_2 \subseteq r_2$ that satisfy $\tilde{r}_1[XY] \subseteq_1 \tilde{r}_2[ZW]$ and contain at least $|r_1| \cdot \rho$ entries. We can, then, use the projection property of INDs to conclude that $\tilde{r}_1[X] \subseteq_1 \tilde{r}_2[Z]$ and $\tilde{r}_1[Y] \subseteq_1 \tilde{r}_2[W]$ are valid. Because $|\tilde{r}_1| \geq \rho \cdot |r_1|$ and $\tilde{r}_2 \subseteq r_2$, it follows that for every tuple $t \in \tilde{r}_1$ we have $t[X] \in \tilde{r}_2[Z] \subseteq r_2[Z]$ (and also $t[Y] \in \tilde{r}_2[W] \subseteq r_2[W]$). Therefore, at least $\rho \cdot |r_1|$ tuples of r_1 witness the inclusions on X (and Y), which implies $r_1[X] \subseteq_\rho r_2[Z]$ and $r_1[Y] \subseteq_\rho r_2[W]$. Hence, both INDs and pINDs adhere to projection. Due to the projection property, $r_1[XY] \subseteq_\rho r_2[ZW]$ can be true only if all its projections are true. This characteristic enabled search space pruning when discovering n-ary (p)IND discovery, because we generate the candidate $r_1[XY] \subseteq_\rho r_2[ZW]$ only if all its projections are valid (*upwards pruning*) and can, vice versa, infer that all projections are valid, if it was validated (*downward pruning* [29].

3.2 Null handling

Relational databases sometimes include `null` (or simply *empty*) values. The presence of such values can be due to historical database expansions, missing or lost data, unknown values, or entries that do not apply to certain records. So `null` values generally refer to undefined or missing data [51]. To handle `null` values practically, Köhler et al. proposed *possible world* semantics under which keys and also INDs are considered valid if a replacement of `null` values exists that makes them valid [23–25]. This property is very expensive to test and hardly supported by profiling algorithms. Instead, most algorithms including SPIND offer the following `null` interpretations options:

Constant: With the *constant* interpretation, all `null` values are turned into a dedicated constant for pIND profiling. This simple interpretation results in `null` = `null` and `null` $\neq x$ for any non-`null` value x. It makes sense if, for example, `null` denotes an implicit value that cannot be expressed explicitly, such as *infinity* in numeric columns or *future* in date columns; it should, therefore, also be contained in any referenced column. Technically, we serve this strategy by simply replacing `null`s with a constant during preprocessing.

Unique: With the *unique* interpretation, every `null` represents a globally unique value. Hence, `null` $\neq$ `null` and `null` $\neq x$ for any value x. The *unique* handling can be used in pessimistic profiling scenarios, where `null` entries represent missing data that will be filled at some point and the profiling should not return pINDs that may soon be violated. To implement *unique*, we simply count all occurrences of `null` values in dependent attributes as violations.

Subset: With the *subset* interpretation, the `null` handling follows *more informative tuples* semantics [53] and *partial* `null` handling semantics in the SQL standard [34]. A `null` value in the left-hand side (lhs) of a pIND matches any value of the right-hand side (rhs) and a `null` value in the rhs matches only `null` values in the lhs. More specifically, a tuple t_2 is more informative than a tuple t_1 if the non-`null` values of t_1 are a positional subset of the non-`null` values of t_2. For example, t_1 = [Mewtwo, $\perp$, psychic] is a subset of t_2 = [Mewtwo, m/f, psychic] and t_1 in an IND's lhs is no violation, but t_2 in an IND's lhs would be a violation with only t_1 in the rhs. To implement the *subset* interpretation, we can simply ignore any `null` values or value combinations with `null` during level-wise bottom-up traversal, because they do not violate INDs in left-hand sides and they do not help right-hand sides to compensate for missing rhs values.

4 SPIND

In this section, we present SPIND, the first data profiling algorithm for the automatic discovery of all unary and n-ary pINDs in relational datasets. The algorithm name is an acronym for **S**calable **P**artial **IN**clusion **D**ependency discovery. It follows a level-wise candidate generation procedure that, first, generates and validates all unary pIND candidates and, then, processes candidates of progressively larger arity while pruning all those candidates from the search that cannot be valid. The main innovation of this algorithm lies in its candidate validation: In every cycle, it sorts all values into only one global run of values (or value combinations). By annotating these values with their corresponding attributes, we can compress and store all sorted values in only a very few files, which still enable the necessary pIND candidate validations. We propose a novel compression scheme with value counts and attribute annotations to minimize I/O times. For parallelization, we propose to chunk the input data so that every chunk can be processed in parallel; in a special merging step, every algorithmic cycle prepares the necessary combined validation data for precise and correct validations. The chunking procedure is a one-time effort that adds a small overhead, but clearly pays off in the discovery of n-ary pINDs. To save even more time in the candidate validations, we propose a probabilistic filtering step that can avoid some unnecessary validations.

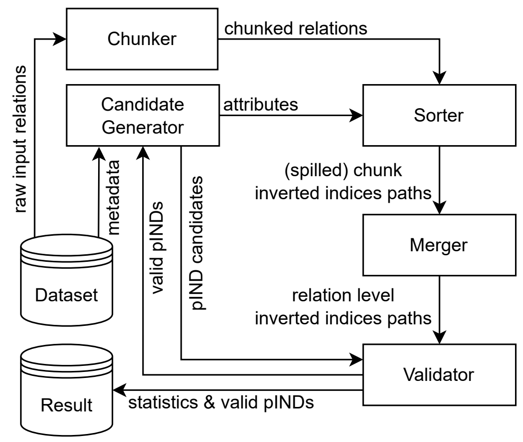

Figure 1: Conceptual overview of SPIND

Figure 1 and Algorithm 1 provide a more detailed overview of SPIND's (p)IND profiling steps: The *Chunker* first horizontally partitions every input relation into a number of chunks (Line 1); the chunks are bound in size and used only for parallelization; the chunking procedure is a one-time effort, for unary and also n-ary (p)IND discovery (see Section 4.1). The *Sorter* creates sorted chunks, which contain the original values of the chunks but turned into sorted inverted index structures (Line 5); sorted chunks essentially contain and sort all value–attribute pairs (or value-combination–attribute-combination pairs) of the input chunk (Section 4.2). The *Merger* performs a k-way merge on the sorted chunks to combine all sorted values of each relation in only one file (Line 6, Section 4.3). The *Validator* takes the (p)IND candidates, which is initially the cross product of all attributes, and the sorted value files of all relations as input to validate the candidates on the sorted values (Lines 7–8); the validation process uses parallelization (Section 4.4) and probabilistic filtering (Section 4.7). The *Candidate Generator* creates the initial, unary (p)IND candidates (Line 3) and successively also all n-ary (p)IND candidates with a bottom-up level-wise candidate generation procedure (Line 11); this procedure creates an iterative loop (Line 4) in SPIND's profiling approach (Section 4.5). Because the discovery of n-ary (p)INDs requires the serialization of n-ary value combinations, we also discuss an effective technique for this task (Section 4.6).

4.1 Chunking the input relations

SPIND heavily relies on sorting. As we learned from previous studies, though, sorting can become a bottleneck, because it does not allow for early termination [12]. This bottleneck prevents algorithms, such as SPIDER, from discovering n-ary INDs. Efficient parallelization on modern multi-core CPUs can, however, alleviate this issue. The challenge, though, is that the parallelization strategy needs to work for both unary and n-INDs. We therefore propose to vertically partition the input relations into *chunks*. The chunks keep the records intact and can, for this reason, be re-used for (p)IND validations of any size.

To implement this idea, SPIND's *Chunker* reads the input relations record-wise and splits them into partitions with at most

$$\#rows = \left\lceil \frac{CHUNK\_SIZE}{\#columns} \right\rceil$$

```
Input: Dataset $\mathcal{D}$, partial threshold $\rho$
Output: Set of valid maximal n-ary pINDs $\Sigma$
1  $chunks \leftarrow$ chunk($\mathcal{D}$);
2  $layer \leftarrow 1, \Sigma \leftarrow \emptyset, \Phi \leftarrow$ initBloomFilter();
3  $candidates \leftarrow$ getCandidates($\Sigma, layer$);
4  while $candidates \neq \emptyset$ do
5      $sortedValues \leftarrow$ sort($candidates, chunks, \Phi$);
6      $mergedValues \leftarrow$ merge($sortedValues$);
7      $validator \leftarrow$ new Validator($candidates, \rho, layer$);
8      $\Phi \leftarrow validator$.validate($mergedValues, \Phi$);
9      $\Sigma \leftarrow \Sigma \cup validator$.getValidPINDs();
10     $layer \leftarrow layer + 1$;
11     $candidates \leftarrow$ getCandidates($\Sigma, layer$);
12 return $\Sigma$;
```

Algorithm 1: SPIND's level-wise iterative pIND discovery

rows, where *CHUNK_SIZE* is a user-defined constant describing the approximate maximum number of values in a chunk. Larger chunks incur less file handling overhead, but smaller chunks increase the degree of parallelism; because the overall performance of SPIND is not sensitive to this parameter and a robust default value can be recommended (see Section 6). For the partitioning, the *Chunker* simply fills a chunk and, then, opens a new chunk; every relation is chunked independently. In this way, the number of chunks varies between different relations, but the maximum sizes are similar. Because the records are kept intact, chunking is performed only once at the start of an execution; the processing of n-ary (p)INDs of any size n, then, relies on the same chunked relations. Note that the proposed chunking is for parallelization purposes only and does, unlike BINDER, not use any hashing.

After the chunking is completed, we obtain a collection of chunks for each relation. Larger relations are divided into many chunks and smaller relations are often contained in only a single chunk.

4.2 Sorting Chunks

The *Sorter* takes as input the chunked relations and the attributes (or attribute combinations) that are needed for the next round of (p)IND candidate validations. For every input chunk, the *Sorter* creates one (or more as we will discuss later) sorted output chunks. An output chunk stores the values of the original chunk sorted in ascending order and annotated with the values' attributes (or attribute combinations) and occurrence counts.

Algorithm 2 shows the basic sorting strategy that SPIND applies in parallel to all chunks. The *attributes* are all attributes (or attribute combinations for n-ary INDs) that appear in the current set of IND candidates; we represent all attributes with unique numeric IDs and make the simplifying assumption that arbitrary records can be projected to these attribute IDs like *record[attribute]*. The cunking algorithm, first, creates a MultiMap that maps the values (or value combinations) of the chunk to the values' attribute(s) and the respective number of occurrences (Line 1). By inserting all values successively into this MultiMap, both duplicate and unprojected values are removed (Lines 2 to 7). Once the chunk has been processed, the values are sorted (Line 8) and written to disk into one sorted chunk file with their attribute and count annotations (Line 9). Finally, a *path* pointer to the sorted file is returned and passed on to the merging step.

The file format of the sorted output file uses two lines for each value: The first line stores the *value* and the second line the

```
Input: chunk and attributes
Output: path to the sorted inverted index file
1  val2attr2cnt = MultiMap of String to Integer to Integer
2  for record ∈ chunk do
3      for attribute ∈ attributes do
4          if (record[attribute], attribute) ∈ val2attr2cnt
5              val2attr2cnt.get(record[attribute], attribute) +=
                1
6          else
7              val2attr2cnt.put(record[attribute], attribute, 1)
8  sort(val2attr2cnt)
9  path = write(val2attr2cnt)
10 return path
```

Algorithm 2: SPIND's chunk sorting

attribute-count-pairs of all attributes that contain the value. If there are k attributes associated with a value, the attributes are serialized in the format $id_1, count_1; \ldots; id_k, count_k$. For example, if the value "electric" appears once in the attribute "3" and twice in the attribute "8", we would write "electric" on the first line and "3,1;8,2" on the second line. More details on how complex value combinations are serialized on the first line are discussed later in Section 4.6.

To avoid memory exhaustion, we limit the MultiMap's size by a constant called SORT_SIZE. If the map size reaches the defined limit (which is equally divided by the number of parallel threads, who all create one such map), the map is sorted and written to disk prematurely.

4.3 Merging Inverted Indices

Given the sorted files with inverted indices, SPIND's *Merger* now creates one sorted inverted index file per input relation by merging all files from the same relations. In this way, the merged files contain all values of one relation sorted and annotated with their respective attributes (and counts). In other words, this is the union of all columns of the relation sorted into a single sequence. This consolidation ensures that the subsequent validation computed the complete inverted indices and, in this way, guarantees that no pINDs are missed due to the initial chunking. The fully sorted columns are necessary for the validation step, which later on iterates the attribute (combination) values systematically.

The merging process is executed for each relation in parallel. To limit the number of simultaneously open file handles, we introduce a constant MERGE_SIZE that restricts the number of files being merged simultaneously. Algorithmically, the *Merger* reads up to MERGE_SIZE files at the same time and merges their key-value pairs with the help of a priority queue and a k-way merge [47] into another single output file. At the end of the merge phase, we obtain a single sorted file for each relation. Through effective latency hiding, this merge strategy avoids I/O bottlenecks as observed in related IND profiling algorithms [36].

4.4 Validation of Candidates

At this point of SPIND's execution, the *Validator* receives a set of (p)IND candidates from the *Candidate Generator* and one sorted inverted index file for each relation from the *Merger*. Before the candidate validations start, the algorithms introduce one last constant parameter called VALIDATION_SIZE that specifies how many values can be pre-loaded in total over all relations for the validation; in SPIDER, this is only one value per attribute, and in BINDER, this is an entire level of buckets. Specifying the size limits VALIDATION_SIZE, CHUNK_SIZE, SORT_SIZE, and MERGE_SIZE statically minimizes memory management overhead and still enables efficient parallelization and latency hiding; SPIND's performance is, in general, not sensitive to these parameters so that we can provide robust default values; the values are also fairly easy to guess manually, because all the parameters are about numbers of values that should kept in main memory for specific purposes.

4.4.1 Parallel Pruning Job Construction. With the VALIDATION_SIZE parameter, we can now have a look at Figure 2 that shows the six steps of SPIND's candidate validation. The validation starts in step (a) that reads all input relations $\mathcal{R}$ in parallel. For every relation, the *Validator* allocates a buffer of size

$$bufferSize = \left\lfloor \frac{VALIDATION\_SIZE}{|\mathcal{R}|} \right\rfloor.$$

The file readers, then, read their top header values (with their respective attribute and count annotations) until the assigned buffers are full or the readers reach the end of their files; in the later case, the respective relation has been finished and both the file and the file reader can be deleted. For latency hiding, the reading process is parallelized to $|\mathcal{R}|$ many threads. In step (b), the *Validator* finds across all buffers the smallest value x (the smallest head element) and the smallest largest value y (the smallest tail element). This can be done in $O(|\mathcal{R}|)$ time. Hence, the value range $[x, y]$ is now loaded from all relations into main memory and ready for validation. In step (c), we join the $[x, y]$ ranges of all remaining buffers on their values and concatenate their *attribute-count*-pair lists. For the join, the *Validator* uses a simple in-memory hash-join, because the order of the values is irrelevant after the join. The result of the join contains up to (but usually much less than) VALIDATION_SIZE many values in the range of $[x, y]$. Step (d) considers every value with its assigned *attribute-count*-pairs as a pruning job and processes these pruning jobs in parallel.

4.4.2 Parallel Pruning Job Processing. The pruning jobs need to synchronize their actual edits of the (p)IND candidates in set (e), because the candidates are stored in a shared data structure. Step (d) can, however, already parse the *attribute-count*-pairs and prepare the attribute sets for candidate pruning in parallel: If an attribute is not a dependent or referenced attribute of any (p)IND candidate, it is removed from the job's attribute set (the dependent/referenced tests are in $O(1)$ time); if no attribute of the set is a dependent side attribute, the job is stopped, because it does not affect any candidate (the dependent side test is also in $O(1)$ time); and if at least one attribute is a dependent attribute, the value of this job is inserted into a synchronized probabilistic Bloom Filter data structure in step (f) – we describe the purpose of this data structure in Section 4.7.

4.4.3 Violation-aware Validation. All pruning jobs that are actually found to be relevant for pruning are handed to step (e). In this step, a single thread takes all the attribute sets and applies them to the (p)IND candidate set, which is represented as a HashMap data structure. SPIND follows De Marchi's inverted validation [31], which iterates over the global domain of sorted values to check attribute memberships. However, instead of pruning a candidate $A \subseteq B$ on the first instance where $v \in A$ and $v \notin B$, SPIND increments a violation counter. The counter increases by

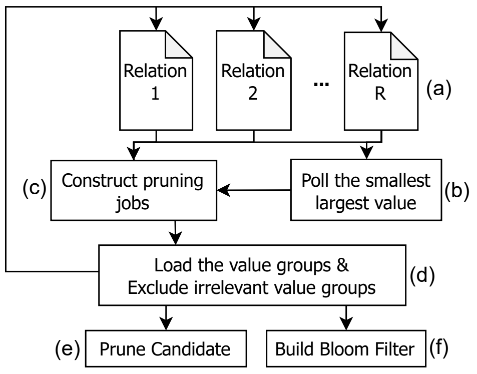

Figure 2: Parallelized Validation on a relational level.

one (*duplicate-unaware*) or by the occurrences of v (*duplicate-aware*), and the candidate is only removed when *maxViolations* is exceeded. To calculate the maximum number of allowed violations w.r.t. a partial degree ρ of the to-be-discovered INDs, SPIND needs to track the number of unique values *#unique* and the total number of values *#total* per attribute. In a *duplicate-aware* setting, the number of allowed violations is

$$maxViolations = \lfloor (1 - \rho) \cdot \#unique \rfloor,$$

while in the *duplicate-unaware* setting it is

$$maxViolations = \lfloor (1 - \rho) \cdot \#total \rfloor.$$

If *#violations* > *maxViolations*, a pIND candidate is pruned. With the number of violations, SPIND can, in the end, also provide the exact partial degree of every individual pIND. Once the attribute sets have been processed, the $[x, y]$ value ranges are released from memory and step (a) refills the buffers. The iterative process ends when all relation files are read entirely; all remaining pIND candidates are, then, valid and can be returned by the *Validator*.

Note that the entire process works with both unary and n-ary (p)IND candidates. For n-ary (p)INDs, the process simply considers lists of attributes and lists of values. More details on candidate generation and n-ary values are provided in the following sections.

4.5 Partial N-ary Candidate Generation

When SPIND is started, its *Candidate Generator* first reads the input's schema information and, then, creates all unary pIND candidates as the cross product of all attributes. The attributes are passed to the *Sorter* to inform it about the columns of values that are to be sorted; the pIND candidates are forwarded to the *Validator* to inform it about the candidates that are to be validated (see Figure 1). So far, we basically discussed how the *Sorter*, *Merger* and *Validator* discover unary pINDs with these inputs. To discover (n+1)-ary pINDs, the *Candidate Generator* waits for the n-ary candidates to be discovered and, then, generates the next level of (n+1)-ary pIND candidates in a *bottom-up candidate search space traversal* approach [12]. This iterative process of generating and validating ever larger candidates from the previous level follows the idea of the apriori-gen algorithm [2]. More specifically, the *Candidate Generator* receives the set of all n-ary pINDs Σ_n and then creates the (n+1)-ary pIND candidates Σ_{n+1} with the following rules: The pIND candidate $r_i[XA] \subseteq_\rho r_j[YB]$ is formed if and only if

1) $\forall C \in X : C < A$

```
    Input: tuple (a list of n strings)
    Output: value (representing the tuple as a string)
1   if $n == 1$
2       return tuple[1]
3   value = "" ∪ length(tuple[1])
    /* Append the length encoding */
4   for index ∈ {2, ..., n - 1} do
5       value = value ∪ ":" ∪ length(tuple[index])
6   value = value ∪ "|"
    /* Append the inner values */
7   for innerValue ∈ tuple do
8       value = value ∪ innerValue
9   return value
```

Algorithm 3: SPIND's n-ary tuple serialization

2) $XA \cap YB = \emptyset$ and $B \notin Y$
3) $\forall k \in \{1, \ldots, n\} : r_i[XA \setminus X_k] \subseteq_\rho r_j[YB \setminus Y_k] \in \Sigma_n$

To illustrate the generation, we again consider our example in Tables 1 and 2 with a threshold $\rho = 0.6$ and assume that SPIND has already discovered the unary pINDs σ_1 : PT[Name] $\subseteq_{0.6}$ PS[PName] and σ_2 : PT[Gender] $\subseteq_{0.6}$ PS[PGender]. The algorithm then generates the next level of 2-ary candidates by combining unary pINDs and applying the rules. Combining σ_1 and σ_2 creates the candidate σ_3 : PT[Name, Gender] $\subseteq_{0.6}$ PS[PName, PGender]. Rule 1 prevents the generation of σ_4 : PT[Gender, Name] $\subseteq_{0.6}$ PS[PGender, PName] by enforcing the left-hand side attributes to be sorted (e.g., by attribute ID) to avoid redundant candidates (here $\sigma_4 = \sigma_3$). Rule 1 and 2 together prevent the generation of any PT[Name, Name] $\subseteq_{0.6}$ PS[PName, PName] with repeating attributes by ensuring that all attributes are disjoined – a restriction that basically all n-ary IND discovery algorithms make for practical reasons [12]. Rule 3 prevents the generation of any pIND with an invalid projection, such as PT[Name, Type] $\subseteq_{0.6}$ PS[PName, PGender]. It exploits the projection property introduced in Section 3.1 and ensures that only potentially valid candidates are generated (upwards pruning).

Some previous algorithms, including BINDER, do not fully enforce Rule 3 and test only for projections with the same prefix, which is faster to compute but often generates many unnecessary but expensive candidates. Furthermore, some related work algorithms require B to be non-empty, which makes sense for *subset* null semantics (see Section 3.2), but because SPIND offers different null handling semantics, the algorithm allows empty attributes for B.

4.6 Serialization of N-ary Tuples

The discovery of n-ary pINDs, such as $r_i[ABC] \subseteq_\rho r_j[DEF]$ requires SPIND to serialize, hash and compare n-ary tuple values, such as $t_i[ABC] \in r_i[ABC]$ and $t_j[DEF] \in r_j[DEF]$. SPIND could implement custom comparison and hashing functions for these tuples, but then all components of SPIND would need to implement all their functionality twice, once for strings and once for tuples. For this reason, SPIND serializes these tuples to string values and, then, operates n-ary tuple values in the exact same efficient, global sorting and merging way used for unary IND discovery. The serialization is performed in the *Sorter* module, which reads the chunked relations and forms the values for every round of candidate validations.

Existing algorithms, such as BINDER [36] and DeMarchi [31], also rely on a simple form of serialization that concatenates the individual values with a special character, such as *"#"*. For example, the tuple *("Marburg", "35037", "06421")* would be transformed into the string *"Marburg#35037#06421"*. However, concatenating values in this way can produce incorrect results if some of the tuple values contain the concatenation string. For this reason, we propose an injective serialization $f : T \longrightarrow S$ from the space of all string tuples T to the space of all strings S. The function f accepts a tuple $t \in T$ consisting of n strings $t = (s_1, ..., s_n)$ and creates a single string $s \in S$ as output. Simple concatenation functions are not injective, because the tuples $(1\#1, 1)$ and $(1, 1\#1)$, for example, both produce the same string *"1#1#1"*. Although these cases are rare in real-world datasets, they are ever more confusing if they actually happen.

For SPIND, we propose the injective n-ary tuple serialization function shown in Algorithm 3. The function takes an n-dimensional tuple and, then, conducts a length encoding for the first $n-1$ values. These lengths are concatenated with the separator ":" and added to the target value. We terminate the sequence of lengths with the separator "|". The choice of ":" and "|" is arbitrary and any non-numerical character would work here. After the length encoding, all inner values of the input tuple are concatenated without a separator and added to the target value, which concludes the serialization.

Algorithm 3 transforms the three example tuple from before as follows: *("Marburg", "35037", "06421")* $\longrightarrow$ *"7:5|Marburg3503706421"*. In this transformation, "7" is the length of the first value and "5" is the length of the second. To show that the algorithm is injective, we construct the inverse function f^{-1} and show that the chain $f^{-1} \circ f$ is the identity function of T. Our inverse function searches for the first occurrence of "|", which marks the end of the length encoding. Then, it splits the substring before the "|" at ":" to retrieve the indexes for splitting the substring after the "|". Given an n-ary tuple of strings $(s_1, \ldots, s_n)$, f produces the string "$len(s_1)$:$len(s_2)$:...:$len(s_{n-1})|s_1 s_2 \ldots s_n$". The inverse function f^{-1} cuts the second part at the specified indices and reproduces the tuple $(s_1, \ldots, s_n)$. In this way, the original tuple is re-created and the injectivity of Algorithm 3 is shown.

4.7 Probabilistic Filtering

We now consider the set of all values $\mathcal{X}$ that occur in at least one dependent attribute. Given the set of all valid pINDs Σ, we can define $\mathcal{X}$ as follows:

$$\mathcal{X} = \bigcup_{x \in \{r_i[X] \mid r_i[X] \subseteq_\rho r_j[Y] \in \Sigma\}} x$$

If we knew $\mathcal{X}$ up front, we could utilize it during pIND discovery to directly filter out all values $y \notin \mathcal{X}$ in the reading process, because they are irrelevant for sorting, merging, and validation. However, $\mathcal{X}$ is usually too large to be stored in memory and calculating $\mathcal{X}$ also requires the pINDs Σ to be known. For this reason, we suggest to compute $\mathcal{X}$ incrementally with a probabilistic Bloom filter [48]: During the validation of the unary pIND candidates, SPIND's *Validator* inserts every value that was found in a dependent attribute into a Bloom filter (see step (f) in Figure 2). Then, when SPIND moves to the next level of n-ary pIND candidates (see loop in Figure 1), its *Sorter* can test every read value for containment in this Bloom filter. If the value is contained, it is used to form n-ary value tuples for the current n-ary pIND candidates; if the value is not contained, it can directly be discarded, because no combination with this value is contained in a dependent attribute combination.

The incremental, approximate computation of $\mathcal{X}$ likely inserts more values into the Bloom filter than $\mathcal{X}$ actually contains. This is because the *Validator* continuously adds values of dependent attributes into the Bloom filter before their pIND candidates are fully verified, which leads to some false positives. Additionally, the Bloom filter might naturally add some false positives due to hash collisions. These false positives reduce the effectiveness of the filter a bit, but because all true $\mathcal{X}$ are contained in the Bloom filter, no relevant values are filtered out once the Bloom filter-based value removal is activated for the n-ary pINDs of size n > 1. Because the Bloom filter is populated while profiling the unary pINDs, it improves the performance of only the n-ary pIND discovery. In Section 6, we evaluate the performance impact of the probabilistic value filtering; we also evaluate, if the filter should be populated only once during unary pIND discovery or if continuous updates with n-ary values are beneficial.

For the construction of the Bloom filter, we opted for a default configuration that expects 100 million values and specifies a 1% false positive rate. The resulting Bloom filter takes about 400 MB of main memory. Depending on the available memory, a user might want to in- or decrease the size of the Bloom filter.

5 Partializing existing IND Algorithms

The general idea of SPIND is similar to the algorithms BINDER [36] and SPIDER [3], but waiving all the algorithms' weaknesses: SPIND also chunks and sorts the input data, but the result is only one sorted file per input relation, which decreases file handling overheads significantly especially for n-ary IND discovery; the chunking and sorting strategy does not require active, usually costly memory monitoring; and the use of aggressive parallelization makes the sorting-based discovery competitive to the purely hashing-based approach. The general validation approach of SPIND is also similar to those of BINDER and SPIDER (eventually all three are based on the strategy of De Marchi [31]), but we add counting capabilities for pINDs and an efficient candidate generation techniques for n-ary (p)INDs, as well as probabilistic filtering and chunked parallelization for increased performance.

In this section, we introduce necessary modifications to the existing IND discovery algorithms SPIDER [3] (Section 5.1) and BINDER [36] (Section 5.2) that enable the discovery of pINDs. Note that the techniques *lazy sorting*, *violation-aware validation*, *parallelized execution*, and *strict candidate generation* are our improvements that we also utilize in SPIND.

5.1 Partial SPIDER

The original SPIDER algorithm [3] discovers only unary INDs. It uses a two-step process with a sort phase and a merge phase across all columns. First, it sorts the values of each attribute and stores each sorted value list in a separate file. Then, it performs a k-way merge while simultaneously verifying all unary IND candidates.

To extend SPIDER's capabilities, Bauckmann et al. already conceptualized a pINDs discovery variant. The idea uses counters to track violations and invalidates IND candidates if violation numbers exceed a predefined threshold. However, this algorithm idea could handle only the *duplicate-unaware* `null`-handling case and was never actually implemented and tested. The pSPIDER modification, which we introduce in this section, supports both *duplicate-aware* and *duplicate-unaware* scenarios. It also integrates the lazy sorting, hash-based deduplication, and parallelization, which are contributions of this paper, in order to achieve a runtime improvement of up to 14 times compared to the original SPIDER.

5.1.1 Lazy Sorting. The original SPIDER implementation runs a *Two-phase Merge Sort* with a `SortedSet` as its central data structure during the attribute sorting: Upon reading a new record, each value is placed directly into a corresponding `SortedSet`. If, at some point, the main memory consumption of the executing machine surpasses a specific threshold, the values are written to disk. This process is called *spilling*. Once written to disk, the sorted sets of values are released from main memory. When all attributes have been sorted, SPIDER merges the sorted chunks into one sorted value list per attribute. The use of a `SortedSet` keeps the values sorted at all times and also directly deduplicates the contained values, but *lazy sorting* combined with *hash-based deduplication* as introduced in Section 2 is much more efficient. For this reason, pSPIDER uses simple, fixed-size `HashMaps` to capture and deduplicate the attribute values; it also tracks duplicate occurrences with complexity of $O(1)$ to enable *duplicate-aware* partialization strategies. If the fixed size of a value set in a `HashMap` is reached, the algorithm sorts all values in the map and spills them to disk. The lazy sorting reduces the sorting times by approximately 75%. To track the value counts, we store the sorted values with their individual counts using the same compressed file format as SPIND. The merge procedure is kept unchanged.

5.1.2 Violation-aware Validation. To calculate the maximum number of allowed violations w.r.t. a partial degree ρ of the INDs, pSPIDER tracks the number of unique values *#unique* and the total number of values *#total* per attribute. It then uses the *violation-aware validation technique* introduced in Section 4.4.3 to validate the pIND candidates: Only if *#violations* > *maxViolations*, a pIND candidate is pruned. Apart from this extra check, the validation processes of pSPIDER and SPIDER are almost identical.

5.1.3 Parallelized Execution. SPIDER runs in two phases: a sorting phase and a validation phase. Because the algorithm spends most of its time in the sorting phase and the validation phase is not easy to parallelize, we applied SPIND's parallelization only to the sorting phase of pSPIDER. The algorithm, first, reads all input relations (i.e., CSV files or database tables) in parallel. For each relation, it splits the records vertically into the attribute-wise value sets discussed in Section 5.1.1 and stores the value sets into (unsorted) files (one per attribute). Then, the sorting strategy sorts the value files in parallel by reading and sorting multiple files simultaneously. The sorting and job scheduling in this phase largely follow the parallelization strategy of SPIND: Parallel sorting jobs are scheduled in a first-in-first-out fashion to available threads and if the values of an attribute cannot be sorted in one go due to the limited available main memory, a *Two-Phase Multiway Merge Sort* strategy is applied. With parallelization, we measured runtime reductions of 15% to 75% on our datasets with 8 threads on an 8 core CPU.

5.2 Partial BINDER

The BINDER algorithm uses a divide-and-conquer approach to efficiently discover both unary and n-ary INDs [36]. Because BINDER outperformed many other exact, non-distributed state-of-the-art unary and n-ary IND discovery algorithms [12], we also translate BINDER into a pINDs discovery algorithm called pBINDER. The necessary violation-aware validation extensions cause only a negligible computation overhead. However, we also apply optimizations in pBINDER's candidate generation that offer an enhanced performance for n-ary (p)IND discovery. A major advantage of BINDER over SPIDER is that BINDER waives the costly sorting step. We did, however, not find an efficient parallelization for BINDER.

5.2.1 Strict Candidate Generation. The original implementation of BINDER uses a candidate generation technique that does not utilize the full pruning potential of n-ary INDs: Given the set of n-ary INDs, BINDER generates all (n+1)-ary candidates by combining all n-ary INDs with common (n-1)-long prefix in their dependent attributes to (n+1)-ary IND candidates. In pBINDER, we create an (n+1)-ary candidate if and only if *all* its n-ary subset INDs are valid (more details in Section 4.5). This *strict candidate generation* makes the candidate generation process a bit more costly due to the additional subset checks, but it often creates fewer candidates and, therefore, reduces the expensive file handling overheads significantly.

5.2.2 Violation-aware Validation. Similar to the IND validations in SPIDER, BINDER's validator component also invalidates an IND candidate with the first conflicting value that it finds. To validate pINDs, we also apply the violation-aware validation technique discussed in Section 4.4.3 in pBINDER: We count *#unique* and *#total* in the divide phase and track the number of *#violations* in the conquer phase; if a pIND candidate's violations exceed the threshold defined by ρ, it is invalidated.

6 Experimental Evaluation

In this section, we evaluate the performance of SPIND and compare its profiling times to the profiling times of existing algorithms. We begin with a short description of the experimental setup. Afterwards, we evaluate SPIND's performance on various datasets relative to the performance of existing profiling algorithms (Section 6.1), SPIND's hyperparameter sensitivity (Section 6.2), its scalability w.r.t. the number of given columns, rows, and relaxation (Section 6.3), and the impact of SPIND's probabilistic filtering (Section 6.4).

Algorithms: We compare SPIND with the baseline algorithms AINDD [45], SPIDER [3], and BINDER [36], our improved versions pSPIDER and pBINDER, and the two related IND profiling FAIDA [26] and SAWFISH [21]. All algorithms have been implemented in Java 21 with the same performance-optimizing libraries. Hence, all measurements reflect algorithmic differences. Further evaluations of other IND profiling algorithms can be found in related work [12]. In the scope of this paper, we cannot evaluate the quality or robustness of (discovered) pINDs, we refer the interested reader to the work of Su et al. who have already analyzed the effectiveness of pIND discovery in the presence of errors [45].

Setup: We perform all experiments on a computer with an AMD Ryzen 7 98003DX CPU (8 physical cores with hyper-threading), 64GB 6000MHz DDR5 RAM, and a Samsung 980 PRO SSD. The experiments use a memory limit of 20GB RAM for each run. SPIND and its competitors are implemented in Java and executed on an OpenJDK JVM version 23. All experiments are executed with the *subset* null interpretation and, if pINDs are discovered, *duplicate-aware* semantics (see Section 3); SPIND also supports other null interpretations and duplicate semantics, which do impact its performance, but we measured that a change in these settings does not change SPIND's performance relative to its competitors.

Datasets: For our experimental evaluation, we selected 12 datasets from different domains and with varying characteristics. These datasets are listed in Table 3. The detailed information about how each dataset was obtained or generated is documented on our dataset website [50]. The datasets include the real-world datasets Cars (retail data on cars), ACNH (video-game data from *Animal Crossing*), T2D (benchmarking data for matching web tables to *DBpedia*), WebTables (miscellaneous web table data sampled from the *WebDataCommons* crawl), US (government data from the United States), EU (government data from the European Union), Population (data about worldwise demographics from 1950 to 2025), Musicbrainz (entertainment data about music, artists and brands), and UniProt (vertebrate genome data acquired from *Ensembl*), as well as the synthetically generated datasets Tesma and TPC-H.

6.1 Absolute Profiling Times

In this first experiment, we run all six profiling algorithms with their optimal default configurations on all 12 datasets and measure their absolute profiling times. For a fair comparison, all algorithms discover exact INDs; we evaluate SPIND's runtime increase for pINDs later. The measured runtimes for unary and n-ary IND discovery are listed in Table 3.

For the discovery of unary pINDs, AINDD is often the fastest algorithm on smaller datasets, such as Cars and TPC-H 1. However, it fails to scale to larger datasets, where it either runs out of memory or does not finish. On most larger datasets, pSPIDER is actually the fastest approach for the discovery of unary pINDs. Lazy sorting, parallelization and a few updated, more efficient data structures allow the algorithm to beat the original SPIDER algorithm by a factor of up to 14x (UniProt) on our hardware. The proposed improvements make pSPIDER even faster than all other profiling algorithms. This is mainly because pSPIDER spends most time on sorting and file I/O, which are the two highly optimized and parallelized activities. BINDER's and pBINDER's hashing and file I/O are not parallelized and, hence, both algorithms now perform worst for unary IND discovery. The measurements show that SPIND is on average only a bit slower than pSPIDER and AINDD for the discovery of unary (p)INDs. This is because of SPIND's extra chunk- and merge-steps, which split input files for parallelization purposes and, afterwards, re-combine them into one file per relation. The chunking is a one-time preprocessing step that only reveals its gains in the n-ary IND discovery. pSPIDER's approach of sorting all attributes directly and considering them independently from each other works very well for unary INDs, but it would create an exponentially growing amount of files for n-ary INDs. For this reason, SPIND's value management in horizontal chunks instead of vertical columns is a necessity for the profiling of n-ary INDs.

Because AINDD and SPIDER are designed only for unary pIND discovery, the discovery of n-ary pINDs is the task where BINDER particularly excels. With our extensions to pBINDER, the competitor also became more efficient. However, both approaches still partition the data horizontally and vertically into many bucket files. Due to SPIND's more efficient and scalable file management, it clearly outperforms both BINDER versions despite its more elaborate value serialization (see Section 4.6). The runtime gab between SPIND and pBINDER becomes increasingly larger

Table 3: Datasets, their characteristics, and the runtime measurements of the IND profiling algorithms for exact INDs. † marks user-defined limits; *OOM* are *Out Of Memory* errors; *DNF* are *Did Not Finish* timeouts.

Dataset Metadata				IND Counts			Unary Runtimes						N-ary Runtimes		
Name	Size	#Rec.	#Attr.	Unaries	N-aries	n_{max}	AINDD	SPIDER	pSPIDER	BINDER	pBINDER	SPIND	BINDER	pBINDER	SPIND
Cars	6.1 MB	118k	117	281	91	4	**0.4s**	1.2s	0.7s	1.1s	1.7s	1.0s	4.8s	6.1s	**1.0s**
ACNH	3.5 MB	19k	630	8,686	20,908,814	12	2.6s	2.3s	2.3s	7.8s	9.0s	**0.9s**	*DNF*	*DNF*	**8m 15s**
T2D	4 MB	65k	3125	362,604	9,301,847	8	56.5s	11.4s	11.7s	56.6s	50.5s	**2.6s**	*DNF*	*DNF*	**23.0s**
WebTables	5.3 MB	52k	18,663	19,924,741	-	$1^{†}$	DNF	12m 36s	**15.6s**	9m 34s	2m 20s	34.6s	-	-	-
US	1.2 GB	5.4M	255	753	215,308	7	12.1s	3m 57s	**7.6s**	27.5s	27.5s	12.5s	3h 17m	3h 56m	**48.0s**
EU	1.8 GB	12.4M	624	18,752	54,634	6	12s	1m 35s	**9.8s**	30.9s	29.6s	16.5s	43.2s	42s	**17.5s**
Population	1.7 GB	3.1M	109	236	1	2	35.3s	17m 7s	**19.4s**	57.4s	1m 1s	47.6s	2h 35m	3h 11m	**33m 46s**
Musicbrainz	16.7 GB	191M	1291	1,843	2,739,733	$4^{†}$	OOM (>56m)	29m 6s	10m 35s	10m 3s	11m 59s	**5m 27s**	*DNF*	*DNF*	**47m 17s**
UniProt	633 MB	6.5M	2367	420,412	1,174,863	5	1m 25s	46.7s	**12.5s**	56.8s	57.0s	14.0s	3m 47s	4m 49s	**22.8s**
Tesma	80 MB	1.0M	12	4	1	2	**1.2s**	4.1s	**1.2s**	2.1s	2.1s	2.8s	3.4s	3.5s	**3.0s**
TPC-H 1	1 GB	8.7M	61	96	8	2	**13.1s**	1m 36s	15.3s	24.5s	25.7s	19.1s	1m 55s	2m 5s	**1m 16s**
TPC-H 10	10.5 GB	86.6M	61	97	11	3	2m 46s	16m 19s	**2m 39s**	4m 11s	4m 46s	2m 46s	25m 50s	28m 40s	**12m 39s**

the more n-ary INDs are to be found. BINDER was not able to finish the datasets ACNH, T2D and Musicbrainz within 6 hours, because it spent too much time on writing many files. By consolidating many values in only a limited number of files, SPIND could process these datasets successfully.

Next, we compare SPIND's performance to the profiling algorithms FAIDA (approximate IND discovery) and SAWFISH (similarity IND discovery). Both algorithms solve a different task than SPIND (FAIDA's task is easier and SAWFISH's task is harder), but the comparison is interesting nonetheless: Surprisingly, FAIDA is on average over all datasets 1.5 times *slower* than SPIND and its results are 35% different from the exact results due to the approximation. This is because the all-column join validation of SPIND with the effective pre-aggregation is more effective than FAIDA's sketch-based validations. Non-surprisingly, though, SAWFISH is on average 4.8 times slower than SPIND due to the expensive similarity calculations.

6.2 Hyperparameter Optimization

SPIND exposes five hyperparameters that affect the algorithm's performance: the size of the initially generated chunks (CHUNK_SIZE), the maximum number of values retained in main memory during the sorting phase (SORT_SIZE), the maximum number of files merged simultaneously (MERGE_SIZE), the total queue size across all relations during candidate validation (VALIDATION_SIZE), and the level of parallelization (PARALLELISM) in all phases of the execution. With the following set of experiments, we propose robust default settings for these hyperparameters and show that SPIND's performance is not sensitive to the proposed settings.

As we have seen in Table 3, SPIND profiled small datasets, such as Cars, T2D, Tesma, UniProt, EU and US, in less than a minute. For these datasets, the hyperparameter settings hardly matter, because (I) even conservative settings usually end up simply loading everything at once and (II) the profiling times are so fast that ±20% runtime makes no difference in practice. We therefore include only the larger datasets in the hyperparameter analysis. The experimental setup executed a parameter grid search on all five hyperparameters. In Figure 3, we plot for every hyperparameter the relative runtime measurements of SPIND on all larger datasets in extensive value ranges while keeping all non-scaled hyperparameters at their static optimum. We use relative runtime measurements based on the respective fastest execution (i.e., runtime 1.00) to show the runtime differences more clearly and combine all dataset curves in one plot each. All plots except the PARALLELISM plot share the same y-axis. The chosen "optimal" hyperparameter settings are marked with a vertical, dashed line.

The experimental results show that SPIND's performance is relatively insensitive to the settings of its SIZE-hyperparameters: Except from very small or very high parameter settings, the runtime is affected to less than 20%, which is in the range of regular measurement fluctuations and insignificant w.r.t. SPIND's general runtime improvements (see US, Musicbrainz, and Population in Table 3). Overly small settings for any SIZE-parameter increase file- and job-management overheads and, therefore, ultimately cause noticeable increases in runtime. Overly large settings put more stress on the JVM's garbage collector, but have a much smaller impact on runtimes. Robust default settings are, therefore, 1000 for MERGE_SIZE, 50 000 for VALIDATION_SIZE, 100 000 for CHUNK_SIZE, and 2 000 000 for SORT_SIZE. All other experiments in this evaluation are executed with these default settings. Users of SPIND may increase or decrease the proposed default settings based on their available main memory, but there is no major performance implication and the proposed values are so conservatively small that they never caused SPIND to fully use the provided 20GB RAM. Optimizing the parameters based on specific hardware availabilities is generally difficult, because the pINDs always consume an unpredictable amount of memory. With a potentially factorial growing number of n-ary pINDs, certain datasets may require SPIND to provide more memory for candidates, ultimately forcing the user to restrict the number of n-ary pINDs (e.g. by their maximum size) rather than SPIND's SIZE-hyperparameters. In Java implementations, the memory management via fixed buffer sizes in SPIND has significant performance advantages over e.g. BINDER's dynamic memory management and it is not less safe, because the dynamic memory management also sometimes failed with OutOfMemory exceptions in our experiments.

In contrast to the SIZE-hyperparameters, we observe significant performance implications for the degree of PARALLELISM: On large datasets, additional threads reduce SPIND's runtime by up to four times. Hence, we recommend the use of as many threads as possible, which is 16 in our setup.

6.3 Scalability Across Varying Dimensions

In the following set of experiments, we evaluate the scalability of both SPIND and pBINDER w.r.t. the number of input rows (via TPC-H's scale factor), the number of input columns (via WebTables' #relations), and the degree of partialization (via the ρ threshold). The respective measurements are plotted in Figure 3.

To measure row scalability, we generated five versions of the TPC-H datasets with scaling factors of 1, 10, 30, 50 and 70. Then, we measured the profiling times of SPIND and pBINDER on each

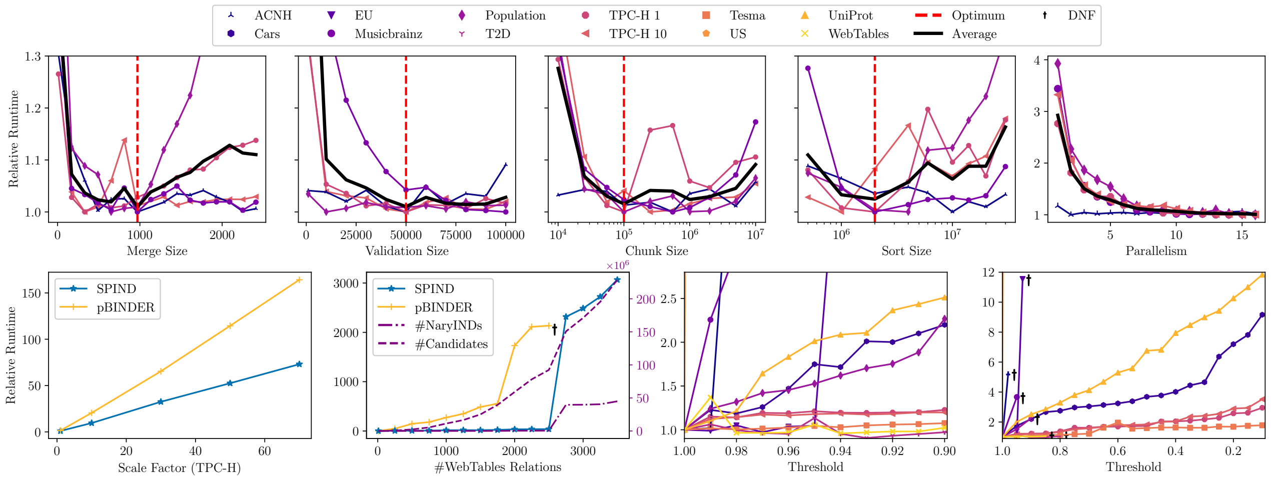

Figure 3: First row: hyperparameter analysis for `MERGE_SIZE`, `VALIDATION_SIZE`, `CHUNK_SIZE`, `SORT_SIZE`, and `PARALLELIZATION` with the chosen optimal settings; second row: scalability analysis w.r.t. rows, columns, and partialization thresholds.

of them and plotted the relative runtimes in Figure 3. As the measurements show, both SPIND and pBINDER scale linearly with the number of rows. This is because the data management costs in both algorithms scales linearly with the lengths of the input dataset and, furthermore, on TPC-H the number of to-be-discovered INDs also hardly changes with the lengths of the dataset. Most noticeable, the relative performance advantage of SPIND over pBINDER stays about constant w.r.t. the number of rows. At scale factor of 70 (≈70GB), pBINDER takes almost 3.8 hours, while SPIND only takes 1.7 hours.

To evaluate the column scalability, we use an increasing number of relations from the WebTables dataset, because (I) the dataset comprises many attributes and (II) web tables have a relatively stable average length. The relative runtimes are again visualized in Figure 3. As we already know, the runtime complexity of (partial) inclusion dependency discovery is in $O(2^m \cdot m!)$ with m attributes [1]. This is because the number of (p)IND candidates and potential (p)INDs growths factorial with the number of attributes. Our measurements in Figure 3 reflect this theoretical assessment and, indeed, show that the algorithms are output bound as their runtimes strongly correlate with the number of to-be-discovered INDs: The runtimes increase only slightly (SPIND) or moderately (pBINDER) with up to 2 500 relations but, then, explode for both algorithms with the almost 45 million INDs of the 2 750 relations set. The increasing number of largely invalid IND candidates up to 2 500 relation is relatively irrelevant for SPIND, due to its efficient validation strategy; the candidates impact pBINDER's runtime more, because of the divide-and-conquer-based candidate validation, which is quite ineffective on short web tables.

In our final scalability experiment, we measure SPIND's runtime on all datasets when gradually decreasing the partialization degree ρ on all datasets. With a smaller ρ, both the validation costs and the number of pINDs increases. This is clearly reflected also in the measurements in Figure 3: For some datasets, the number of pIND does hardly increase so that SPIND's runtime stays relatively low; for other datasets, the number of pINDs (and with it SPIND's runtime) explodes. In the dataset US, for example, SPIND discovers 215 308 exact n-ary INDs within ~2 minutes, but with a $\rho = 0.99$ SPIND already discovers nearly 10x as many n-ary pINDs (2 601 253) and takes over 2.5 hours. If the number of pINDs stays about constant, which is particularly true for synthetic datsets, such as Tesma and TPC-H, a decreasing ρ threshold impacts the profiling time only slightly. The datasets with a timeout † show that partialization can increase the profiling complexity significantly already for threshold above $\rho = 0.8$. Partialization should, therefore, be used with caution – also with an efficient pIND discovery algorithm, such as SPIND.

6.4 Filter Evaluation

In Section 4.7, we introduced a probabilistic filter in two versions: One version builds the filter only *once* after the unary IND discovery step, and the second version *refines* the filter on every n-ary layer. Figure 4 shows the relative execution time changes when enabling probabilistic filtering on the large datasets; runs without probabilistic filtering are the baseline for this experiment.

For some datasets, such as TPC-H or Population, filtering has hardly any effect and, for other datasets, such as US and Musicbrainz, the runtime improvements of −34% and −80% are significant. This is because the former datasets contain only few values that never occur in dependent attributes and/or only few n-ary pINDs, while the latter ones contain many such values and many n-ary pINDs. The linear hashing overhead only pays of if it actually reduces the validation data for n-ary pIND candidate tests; otherwise, it may even add a small runtime penalty. Because the potential runtime gains clearly outweigh the small overheads, we consider this filtering technique viable in practice. The performance difference between calculating the filter *once* or *refining* it with every level is small and inconsistent: Sometimes the additional pruning capabilities of the refinements outweigh the update costs, sometimes they do not. Because the *refine* mode can reduce some read and write operations, which is more healthy for SSD discs, we opted for this filter version as a default setting.

7 Conclusion

In this paper, we presented SPIND, a novel profiling algorithm for the discovery of (partial) inclusion dependencies in large datasets. SPIND outperforms the state-of-the-art IND profiling algorithms

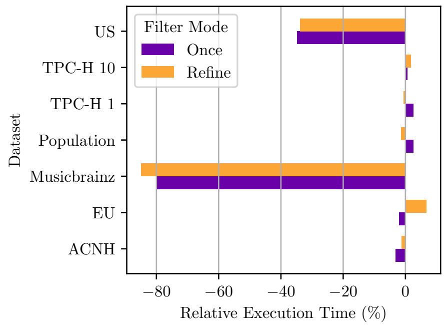

Figure 4: Execution times changes when enabling probabilistic filtering in *once* or *refine* mode for n-ary pIND discovery.

SPIDER and BINDER, as well as their partialized versions pSPIDER and pBINDER, which have also been introduced in this paper, in both efficiency and scalability. SPIND supports different null semantics and exposes a few parameters to control its parallelization, partialization and memory usage. Our experimental results showed that SPIND is particularly effective for discovering n-ary (p)INDs.

In future work, we aim to explore alternative candidate generation techniques, such as the ones proposed with the n-ary IND discovery algorithms Find2 [22], ZigZag [10], and Mind2 [42]. The bottom-up, level-wise candidate generation technique of Mind [31], which is used by SPIND, has been the most effective strategy to date, but a bi-directional bottom-up and top-down candidate generation strategy would also work fine with SPIND's validation approach.

Acknowledgments

This work was supported by the LOEWE initiative (Hesse, Germany) Research Fund of emergenCITY center [LOEWE/1/12/519/03/05.001(0016)/72].

Artifacts

All artifacts, including source code, experimental scripts, and datasets used in this work, are available in our public code repository: https://github.com/UMR-Big-Data-Analytics/partial-inclusion-dependencies. The repository also contains detailed instructions for reproducing our results.

References

[1] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. 2018. Data Profiling. In *Synthesis Lectures on Data Management (SLDM)* (1 ed.). Springer Cham. doi:10.2200/S00878ED1V01Y201810DTM052
[2] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, Vol. 1215. Santiago, 487–499.
[3] Jana Bauckmann, Ulf Leser, Felix Naumann, and Véronique Tietz. 2006. Efficiently detecting inclusion dependencies. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 1448–1450.
[4] Siegfried Bell and Peter Brockhausen. 1995. *Discovery of data dependencies in relational databases.* Citeseer.
[5] Thomas Bläsius, Tobias Friedrich, and Martin Schirneck. 2017. The parameterized complexity of dependency detection in relational databases. In *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[6] Philip Bohannon, Wenfei Fan, Michael Flaster, and Rajeev Rastogi. 2005. A cost-based model and effective heuristic for repairing constraints by value modification. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 143–154.
[7] Marco A Casanova, Ronald Fagin, and Christos H Papadimitriou. 1982. Inclusion dependencies and their interaction with functional dependencies. In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*. 171–176.
[8] CJ Date. 1981. Referential integrity. In *Proceedings of the seventh international conference on Very Large Data Bases-Volume 7*. 2–12.
[9] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. 2002. Efficient algorithms for mining inclusion dependencies. In *International Conference on Extending Database Technology*. Springer, 464–476.
[10] Fabien De Marchi and J-M Petit. 2003. Zigzag: a new algorithm for mining large inclusion dependencies in databases. In *Third IEEE International Conference on Data Mining*. IEEE, 27–34.
[11] Fabien De Marchi and Jean-Marc Petit. 2005. Approximating a set of approximate inclusion dependencies. In *Intelligent Information Processing and Web Mining: Proceedings of the International IIS: IIPWM'05 Conference held in Gdansk, Poland, June 13–16, 2005*. Springer, 633–640.
[12] Falco Dürsch, Axel Stebner, Fabian Windheuser, Maxi Fischer, Tim Friedrich, Nils Strelow, Tobias Bleifuß, Hazar Harmouch, Lan Jiang, Thorsten Papenbrock, and Felix Naumann. 2019. Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM '19)*. ACM, 219–228. doi:10.1145/3357384.3357916
[13] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. 2021. MATE: multi-attribute table extraction. *arXiv preprint arXiv:2110.00318* (2021).
[14] Wenfei Fan. 2008. Dependencies revisited for improving data quality. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 159–170.
[15] Martin Faust, Martin Boissier, Marvin Keller, David Schwalb, Holger Bischoff, Katrin Eisenreich, Franz Färber, and Hasso Plattner. 2016. Footprint reduction and uniqueness enforcement with hash indices in SAP HANA. In *International Conference on Database and Expert Systems Applications*. Springer, 137–151.
[16] Martin Faust, David Schwalb, and Hasso Plattner. 2013. Composite group-keys: Space-efficient indexing of multiple columns for compressed in-memory column stores. In *International Workshop on In-Memory Data Management and Analytics*. Springer, 139–150.
[17] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1001–1012.
[18] Jarek Gryz. 1998. Query folding with inclusion dependencies. In *Proceedings 14th International Conference on Data Engineering*. IEEE, 126–133.
[19] Jan-Eric Hellenberg, Fabian Mahling, Lukas Laskowski, Felix Naumann, Matteo Paganelli, and Fabian Panse. 2024. PRISMA: a privacy-preserving schema matcher using functional dependencies. In *International Conference on Extending Database Technology (EDBT)*. 297–309.
[20] Lan Jiang and Felix Naumann. 2020. Holistic primary key and foreign key detection. *Journal of Intelligent Information Systems* 54 (2020), 439–461.
[21] Youri Kaminsky, Eduardo HM Pena, and Felix Naumann. 2023. Discovering Similarity Inclusion Dependencies. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–24.
[22] Andreas Koeller and Elke A Rundensteiner. 2003. Discovery of high-dimensional inclusion dependencies. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE, 683–685.
[23] Henning Köhler, Uwe Leck, and Sebastian Link. 2013. *Possible and certain SQL keys*. Technical Report. Department of Computer Science, The University of Auckland, New Zealand.
[24] Henning Köhler, Uwe Leck, Sebastian Link, and Xiaofang Zhou. 2016. Possible and certain keys for SQL. *The VLDB Journal* 25 (2016), 571–596.
[25] Henning Köhler and Sebastian Link. 2015. Inclusion dependencies reloaded. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. 1361–1370.
[26] Sebastian Kruse, Thorsten Papenbrock, Christian Dullweber, Moritz Finke, Manuel Hegner, Martin Zabel, Christian Zollner, and Felix Naumann. 2017. Fast approximate discovery of inclusion dependencies. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*. Gesellschaft für Informatik, Bonn, 207–226.
[27] Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. 2015. Scaling out the discovery of inclusion dependencies. In *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*. Gesellschaft für Informatik eV, 445–454.
[28] Mark Levene and Millist W Vincent. 2000. Justification for inclusion dependency normal form. *IEEE Transactions on Knowledge and Data Engineering* 12, 2 (2000), 281–291.
[29] Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. 2010. Discover dependencies from data—a review. *IEEE Transactions on Knowledge and Data Engineering* 24, 2 (2010), 251–264.
[30] 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.
[31] 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 (2009), 53–73.
[32] Lacramioara Mazilu, Nikolaos Konstantinou, Norman W Paton, and Alvaro AA Fernandes. 2020. Schema Mapping Generation in the Wild: A Demonstration with Open Government Data.. In *International Conference on Extending Database Technology (EDBT)*. 615–618.

[33] Lacramioara Mazilu, Norman W Paton, Alvaro AA Fernandes, and Martin Koehler. 2019. Dynamap: Schema mapping generation in the wild. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*. 37–48.

[34] Jim Melton and Alan R Simon. 1993. *Understanding the new SQL: a complete guide*. Morgan Kaufmann.

[35] John C Mitchell. 1983. The implication problem for functional and inclusion dependencies. *Information and Control* 56, 3 (1983), 154–173.

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

[37] Philipp Schirmer, Thorsten Papenbrock, Ioannis Koumarelas, and Felix Naumann. 2020. Efficient Discovery of Matching Dependencies. *ACM Transactions on Database Systems (TODS)* 45, 3 (2020), 1–33. doi:10.1145/3392778

[38] Sebastian Schmidl and Thorsten Papenbrock. 2022. Efficient Distributed Discovery of Bidirectional Order Dependencies. *The VLDB Journal (VLDBJ)* 31, 1 (2022), 49–74. doi:10.1007/s00778-021-00683-4

[39] Marcian Seeger and Thorsten Papenbrock. 2025. DPQL: Applications for Holistic Data Profiling. *Proceedings of the Conference Database Systems for Business, Technology and Web (BTW)* (2025), 155–168. doi:10.18420/BTW2025-08

[40] Marcian Seeger, Sebastian Schmidl, Alexander Vielhauer, and Thorsten Papenbrock. 2023. DPQL: The Data Profiling Query Language. *Proceedings of the Conference Database Systems for Business, Technology and Web (BTW)* 20 (2023), 319–415. doi:10.18420/BTW2023-19

[41] Nuhad Shaabani and Christoph Meinel. 2015. Scalable inclusion dependency discovery. In *Database Systems for Advanced Applications: 20th International Conference, DASFAA 2015, Hanoi, Vietnam, April 20-23, 2015, Proceedings, Part I 20*. Springer, 425–440.

[42] Nuhad Shaabani and Christoph Meinel. 2016. Detecting maximum inclusion dependencies without candidate generation. In *International Conference on Database and Expert Systems Applications*. Springer, 118–133.

[43] Nuhad Shaabani and Christoph Meinel. 2018. Improving the efficiency of inclusion dependency detection. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 207–216.

[44] Shaoxu Song and Lei Chen. 2023. *Integrity Constraints on Rich Data Types*. Springer.

[45] Qingdong Su, Zhikang Wang, Zijing Tan, and Shuai Ma. 2024. Discovering Approximate Inclusion Dependencies. *Proceedings of the VLDB Endowment* 18, 4 (2024), 1210–1222.

[46] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. 2017. Effective and Complete Discovery of Order Dependencies via Set-based Axiomatization. *Proceedings of the VLDB Endowment (PVLDB)* 10, 7 (2017), 721–732. doi:10.14778/3067421.3067422

[47] D. Taniar, C.H.C. Leung, W. Rahayu, and S. Goel. 2008. *High-Performance Parallel Database Processing and Grid Databases*. Wiley. 85–87 pages. https://books.google.de/books?id=xp-mB7hk9nAC

[48] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2011. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials* 14, 1 (2011), 131–155.

[49] Fabian Tschirschnitz, Thorsten Papenbrock, and Felix Naumann. 2017. Detecting inclusion dependencies on very many tables. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–29.

[50] Marburg University. 2025. *Evaluation Data for Partial Inclusion Dependency Discovery Algorithms*. https://github.com/UMR-Big-Data-Analytics/partial-inclusion-dependencies/tree/main/data

[51] Yannis Vassiliou. 1979. Null values in data base management a denotational semantics approach. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 162–169.

[52] Yihan Wang, Shaoxu Song, Lei Chen, Jeffrey Xu Yu, and Hong Cheng. 2017. Discovering conditional matching rules. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 11, 4 (2017), 1–38. doi:10.1145/3070647

[53] Carlo Zaniolo. 1982. Database relations with null values. In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*. 27–33.

[54] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J Miller. 2019. Josie: Overlap set similarity search for finding joinable tables in data lakes. In *Proceedings of the 2019 International Conference on Management of Data*. 847–864.