Partial Inclusion Dependency Discovery

Jakob Leander Müller
Philipps-Universität Marburg
Marburg, Germany
muelle5t@students.uni-marburg.de
me@jakob-l-m.de

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

Phd Mensch
Inria Paris-Rocquencourt
Rocquencourt, France
vb@rocquencourt.com

ABSTRACT

In a world where data grows exponentially and data sharing becomes more common, the need for efficient and flexible data profiling increases constantly. Past research dealing with inclusion dependencies (INDs) mostly overlooked the potential of partial INDs, meaning imperfect subsets. Such partial INDs can enable researchers and data engineers to understand databases, even if different sources do not match perfectly. Combining know strategies such as a sort-merge join and heavy parallelization we will introduce the algorithm *Spind*. *Spind* is not only able to find all partial INDs, it also outperforms the current state-of-the-art algorithm *BINDER* in both unary and n-ary setting by multiple magnitudes.

PVLDB Reference Format:
Jakob Leander Müller, Thorsten Papenbrock, and Phd Mensch. Partial Inclusion Dependency Discovery. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Jakob-L-M/partial-inclusion-dependencies.

1 INTRODUCTION

The amount of data that is being generated is growing constantly and at an ever increasing pace. All digital data is estimated to double about every two years [11]. This thesis will be focused around structured data, a subset of all digital data. Structured data refers to a type of data that is organized and formatted in a consistent manner, allowing for efficient search, retrieval, and analysis. More precisely, we will examine relational data, which is a type of structured data that is organized into tables or relations, where each table represents a set of entities or objects, and each row or tuple in the table represents a single instance of those entities. Example of efforts to collect relational data from the internet are the *Web Table Corpora* [1] or *Wikitables* [2]. These project collect tables but do provide insights that could be extracted from the data. There is also publicly available data from governments, other researchers and private businesses.

One of the most fundamental concepts in relational data are foreign key relations[6]. Foreign keys are a crucial aspect of relational databases as they help define the relationships between tables, maintain referential integrity, prevent errors, and improve the performance of operations pulling data from linked tables. They ensure that each record in one table corresponds to a valid record in another table, thereby promoting consistency and accuracy in the database. While foreign keys are not mandatory, they play a vital role in establishing clear relationships between tables and validating data as rows are added, updated, or removed. By linking data between tables, new insights can be extracted and previously hidden knowledge might get reviled. In today's economy, data profiling and therefore also the discovery of foreign key (and further inclusion dependencies), is a necessity which, if done by human experts, is connected to huge cost [13].

In the ever-expanding field of data management and analytics, the accurate representation and comprehension of relationships within datasets stand as central challenge. Inclusion dependencies encapsulate hierarchical connections between attributes, playing a crucial role in the integrity and normalization of data [6]. Understanding and discovering these dependencies have far-reaching implications for various applications, including database design [17], query optimization [12], and data quality assurance. As the volume and complexity of data continues to increase, there is an ever growing need for advanced methodologies and tools that can extract inclusion dependencies inherent in datasets.

This master thesis embarks on a comprehensive exploration of inclusion dependency discovery. Inclusion dependencies capture the relationships between attributes by specifying that the values in one set of attributes must be included in another. While traditional database design principles rely on the normalization process to ensure the minimization of redundancy and enhance data integrity, the discovery and exploitation of inclusion dependencies provide a nuanced perspective on data relationships, offering insights that extend beyond conventional normalization techniques. We will not only discuss state of the art algorithms but further expand the search to partial inclusion dependencies. This special kind of inclusion dependencies allows for (small) errors and opens the door for finding inclusions dependencies in non-perfect datasets with human errors, spelling differences or historically grown deviations.

Insights generated through different partial thresholds are not merely academic, they have practical implications for companies and governments alike. If a partial inclusion dependency at a 99% threshold is found, organizations could use this information to check for impurity in the given attributes.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

[1] https://webdatacommons.org/webtables/
[2] http://websail-fe.cs.northwestern.edu/TabEL/

2 DEFINITIONS

This sections expands the notation introduced by De Marchi et al. [19]. A relational instance r of a relational schemata R carries tuples of values, donated as u or v. Using an attribute list taken from R, typically denoted as X or Y, we can perform a projection on r, thereby selecting a subset of attributes. We notate it by $R[X]$. The same is possible for tuples u. Writing $u[X]$ references the selection of values in the tuple.

Inclusion Dependencies (INDs) represent a fundamental concept, denoting formal relationships between attributes in a database schema. An IND specifies that the values within one set of attributes are inherently included within the values of another set of attributes.

Definition 2.1 (Inclusion Dependencies). Given two relational instances r_i from R_i and r_j from R_j. An IND is defined as $r_i[X] \subseteq r_j[Y] \iff \forall\, u \in r_i[X], \exists\, v \in r_j[Y]$ such that $u[X] = v[Y]$. This condition can only hold if the cardinality of X is equal to the cardinality of Y. We further call the left hand side (here $r_i[X]$) the dependent attribute(s) and the right hand side the referenced attribute(s).

The complexity of discovering inclusion dependencies forms one of the hardest problems in computer science. More precisely, the discovery of all inclusion dependencies is W[3]-hard [5]. This makes IND discovery one of the hardest problems in computer science. The number of possible candidates for each attribute size can be calculated. Notice that the formula below assumes that all IND of layers before where valid. In natural language, we search for the number of attribute combinations where each attribute is present at most once, allowing all permutations.

Definition 2.2 (Candidate Space). Let α be the fixed integer size of all possible X_i. Let m be the number of attributes. Let k be the number of possible candidates ($X_i \cap X_j = \emptyset$ where $i \neq j$ given α).

$$k = \underbrace{\binom{m}{\alpha}}_{\text{Left hand side combinations}} \cdot \underbrace{\binom{m-\alpha}{\alpha} \cdot \alpha!}_{\text{Right hand side combinations}} .$$

This formula holds if $\alpha \leq \lfloor \frac{n}{2} \rfloor$ else k is 0. We do not permute the left-hand side, since the resulting INDs would only yield redundant information.

In a multi schema setting this calculation becomes more difficult. We now need to consider which schema can form which inter schema candidates while allowing intra schema candidates.

Definition 2.3 (Candidate Space (Multi-Schema)). We will first define a function $q_r(\alpha, \tau)$, which computes the candidates a single relation (r) can produce, given α, the size of combinations and τ an indicator, whether or not the schema was already used for the other side.

$$q_r(\alpha, \tau) = \begin{cases} 0, & \text{if } \alpha > |r| \bigvee (\alpha > \lfloor \frac{|r|}{2} \rfloor \bigwedge \tau) \\ \binom{|r|}{\alpha} \cdot \alpha!, & \text{if not } \tau \\ \binom{|r|-\alpha}{\alpha} \cdot \alpha!, & \text{if } \tau \end{cases}$$

Using this helper function we can calculate the possible candidates k for a fixed α over schema $r_1, \ldots, r_n$ by computing:

$$k = \sum_{i=1}^{n} q_{r_i}(\alpha, FALSE) \cdot \sum_{i=1}^{n} q_{r_i}(\alpha, TRUE).$$

Definition 2.4 (Partial Inclusion Dependencies). A partial inclusion dependency (pIND) is written as $r_1[X] \subseteq_\rho r_2[Y]$ where $\rho \in (0, 1]$ and the reaming notation is analog to Definition 2.1. The possible values for ρ do not include 0, since this would mean everything is a pIND of everything else, which is a trivial case. Further this notation refers to lists of tuples and takes the cardinality of duplicates into consideration. For the pIND $r_1[X] \subseteq_\rho r_2[Y]$ to be valid

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

needs to be true.

This definition uses the $\cap$ operator, which is usually used in set theory and defined for sets. The definitions adaptation of the operator is $r_1[X] \cap r_2[Y] = u \in r_1[X] \mid \exists\, v \in r_2[Y] : u = v$. In natural language the operator conducts the operation *"remove all elements from the dependent attribute, that are not present at least once in the referenced attribute"*. Therefore the referenced attribute could also be a set in a sense that the duplication distribution only matters for the dependent attribute.

As touched on in the definition of pINDs, the handling of duplicate values needs to be decieded. Two different versions where already proposed in [3] and can be defined as *duplicateAware* and *duplicateUnaware*. For the *duplicateAware* version, the maximum number of violations that can occur for $r_1[X] \subseteq_\rho r_2[Y]$ to be valid is equal to $\lfloor (1 - \rho) \cdot |r_1[X]| \rfloor$. We examine if less than ρ percent of entries would need to be changed. In a *duplicateUnaware* setting we care about changed value, while not regarding their number of occurrences. We ask the question, do less than ρ percent of the unique values need to be modified, such that the IND $r_1[X] \subseteq r_2[Y]$ would be valid. In this setting, the maximum number of violations is equal to $\lfloor (1 - \rho) \cdot |\{r_1[X]\}| \rfloor$. Notice the set notation around the attribute, where we calculate the number of distinct values.

2.1 Properties of partial inclusion dependencies

Inclusion Dependencies follow Armstrong's Axioms [2]. We will now discuss which of the properties will also hold in a partial setting. Firstly, partial inclusion dependencies also respect reflexivity. For any $\rho \in (0, 1]$ the partial inclusion dependency $r_i[X_j] \subseteq_\rho r_i[X_j]$ is valid.

$$\frac{|r_i[X_j] \cap r_i[X_j]|}{|r_i[X_j]|} \geq \rho$$
$$\frac{|r_i[X_j]|}{|r_i[X_j]|} \geq \rho$$
$$1 \geq \rho$$

Since ρ is upper bounded by 1 the last statement will always be true.

Contrary to INDs, pINDs do not generally respect transitivity. We will proof this claim by contradiction. Assume $r_1[X] =$

$[1, 2, ..., 100], r_2[Y] = [2, ..., 1000], r_3[Z] = [10, 11, ..., 1000]$,. If transitivity would hold for any ρ, then we should find that for $\rho \in (0, 1]$ where $r_1[X] \subseteq_\rho r_2[Y]$ and $r_2[Y] \subseteq_\rho r_3[Z]$ are valid, $r_1[X] \subseteq_\rho r_3[Z]$ also needs to be valid. For the given example, if $\rho = 0.95$, we find a contradiction.

Lastly, both INDs and pINDs respect projection. We will now outline a proof for this claim. Consider the attributes $r_1[X]$, $r_1[Y]$, $r_2[Z]$, $r_2[W]$ where $r_1[X]$ and $r_1[Y]$ are in the same relation and $r_2[Z]$ and $r_2[W]$ are in the same relation, with $X \neq Y$ and $Z \neq W$. Assume $r_1[X], r_1[Y] \subseteq_\rho r_2[Z], r_2[W]$ is valid for some $\rho \in (0, 1]$. If projection holds, this implies that $r_1[X] \subseteq_\rho r_2[Z]$ and $r_1[Y] \subseteq_\rho r_2[W]$ have to be valid as well. If we now only consider the portion (with reduced size ρ%) which satisfies $r_1[X], r_1[Y] \subseteq_1 r_2[Z], r_2[W]$ we can use the known properties for INDs and conclude that for at least ρ% $r_1[X] \subseteq_1 r_2[Z]$ and $r_1[Y] \subseteq_1 r_2[W]$ has to be valid. This also directly implies that $r_1[X] \subseteq_\rho r_2[Z]$ and $r_1[Y] \subseteq_\rho r_2[W]$ will be true if the remaining $(1 - \rho)$% values are added again. This property is very important for search space pruning, which is the single most important task for (p)IND discovery [18].

The complexity of discovering partial inclusion dependencies is inherited form the base problem. Since the complexity is calculated under a worst case assumption, it does not change when switching into the partial setting.

2.2 Null handling

Relational databases can include *NULL* or *EMPTY* values. The allowance of such values can originate from my different sources like historic expansions of a database, missing or lost data, unknown values or entries which simply can not be filled with any concrete value. For the treatment of these values there are two main interpretations. Entries which are undefined and not relevant, and entires which are missing even tho they are relevant.Using these interpretations we will now define and discuss different approaches on how *NULL* entries are treated. None of the approaches is less valid than another, but one needs to choose a treatment.

*The **unknown** interpretation.* In this interpretation the truth value of $x = y$ where x or y or both are *NULL* is considered as unknown [7]. It is not true or false, but rather lies under the assumption that the *NULL* entry could be filled using some, possibly distinct value. This interpretation of *NULL* entries causes inclusion or equality operations to also return neither true or false, but also an *unknown* truth statement. Our search for pINDs expects a static data input and would need to be re-run in the event of new or updated entries. Therefor the *unknown* interpretation does not provide us with usable inforation. We would rather have the expert user decide between one of the following interpretations.

*The **subset** interpretation.* This interpretation is similar to an interpretation known as *more informative tuples* [24]. A *more informative tuple* relation is a relation between two tuples t_1 and t_2 with attributes A and B where for every $a \in A$ where $t_1[a] \neq NULL$, $a \in B$ and $t_1[a] = t_2[a]$ are valid. Under this notation, t_2 carries the entire inforation of t_1, but potentially contains more inforation for the $NULL$ entries of t_1.

For pIND discovery we now consider a *subset* interpretation. Given two relational instances r_1 and r_2 with attribute sets $X \in r_1$ and $Y \in r_2$ where $|X| = |Y|$. We define $r_1[X] \subseteq_\rho r_2[Y]$ as valid if there are at least $\lfloor |r_1[X]| \cdot \rho \rfloor$ tuples in $r_1[X]$ for which a *more informative tuple* exists in $r_2[Y]$.

*The **foreign** interpretation.* A slight variation of the *subset* interpretation. The difference being the added constraint that the referenced side is not allowed to carry *NULL* values. The motivation for this interpretation is the detection of foreign keys.

*The **equality** interpretation.* This interpretation could be considered a somewhat optimistic version of the *unknown* interpretation in a sense that we consider all $NULL$ entries as being equal. The result is equal to the subset approach, if we consider $NULL$ as just another explicit value. Logically this interpretation should be chosen if $NULL$ implicitly defines an value which can not be expressed explicitly (e.g. infinity in a numeric column), but still holds meaning and is expected in the referenced for a valid pIND.

*The **inequality** interpretation.* The pessimistic version of the *unknown* interpretation. Here we consider every $NULL$ entry as an distinct value. Therefor the comparison between two entries x and y where one or both are $NULL$ always yields false. We are in a setting where $NULL$ entires are considered not yet filled and we choose a worst case scenario where the replacement values are all distinct form each other and the relation itself. In a non-partial setting this would restrict an attribute with at least one $NULL$ entry from forming any INDs. Applying a partial setting, there is still a chance, that attributes with only a few $NULL$ entries are able to form pINDs. This interpretation is equal to the certain world concept. No matter how the $NULL$ values would be set, the found pINDs will stay valid.

Possible World. In a possible world, all pINDs that could become valid under some $NULL$ replacement, are considered valid. This creates a whole new complexity layer when trying to choose the optimal $NULL$ replacements. If a referenced attribute has n $NULL$ entries, we could avoid n violations. If we do not care about deduplicates, this is easy to implementation, since we can increase the possible violations of a candidate by the number of $NULL$ in the referenced attribute. Should be care about deduplicates, this situation becomes more difficult. The optimal way of replacing $NULL$ would be to pick the n values in the dependant attribute with the most occurrences which do not appear in the referenced. That way we avoid the highest number of violations. For this to work, we would need to keep track of the top n highest occurring values, which create a violation. Since n might be very big, we might not be able to store a priority queue or something similar in main memory, and would there for need to store these values on disk. The described problems are certainly solvable, but are out of scope for the thesis. For this reason, the algorithms do not offer pIND discovery in a possible world setting.

3 SPIND OVERVIEW

We will now discuss our proposed algorithm *SPIND*, which stands for **S**calable **P**artial **IN**clusion **D**ependency discovery. Further the german word *Spind* is a special kind of closet and often multiple "Spinds" are placed next to each other. This is a metaphor to the algorithms procedure. Every input relation will be transformed

to a "Spind" of sorted values with connected attributes (attribute combinations), which is surely bigger than a *BINDER* bucket, but there are far fewer "Spinds" than *BINDER* would create buckets.

SPIND solves multiple problems which no other algorithm solves. The architecture does not need to monitor main memory consumption or estimate the size of some objects. *BINDER* uses heuristics to estimate if the content of a file can be loaded without overflowing the available main memory. In some rare cases this heuristic can fail leading to an execution failure. Another issue is the creation of many files. On datasets with a lot of pINDs algorithms like *BINDER* or *SPIDER* will create one file per attribute (combination) regardless of the dataset size. This can lead to millions of files being created, which total a few megabytes. On top of the problems that *SPIND* solves, it also outperforms *BINDER* and *SPIDER* computationally by a factor of up to multiple magnitudes.

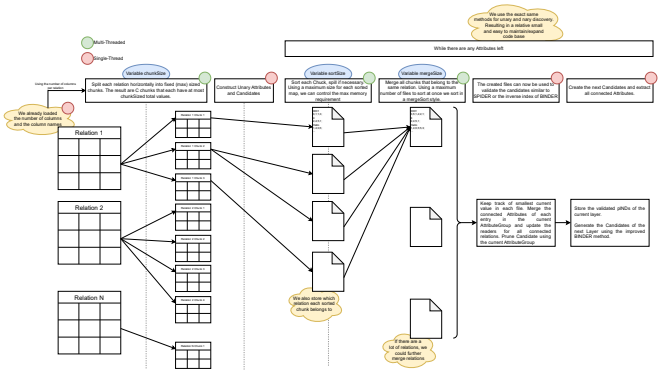

Figure 1: Conceptual overview of *SPIND*

3.1 Chunking the input relations

Modern CPUs feature multiple cores capable of executing tasks concurrently. Previous studies have largely overlooked the potential of multi-threading. In contrast, *SPIND* will leverage multi-threading extensively to maximize hardware utilization. Achieving this objective necessitates the identification of independent tasks capable of being processed in parallel.

The execution starts by fetching some very basic information about the input relations. For each relational instance (input table) *SPIND* will extract the header column (if present) and store the number of unary attributes (columns) each table has. Using a constant *CHUNK_SIZE* which is set by the user, we spilt each relation into somewhat equal parts. Each chunk will consist of at most $\lfloor \frac{CHUNK\_SIZE}{\#cloumns\ in\ relation} \rfloor$ rows. The complexity of processing a relation directly correlates to the number of total values in that relation. While this may be an oversimplification since there are many more factors, like the distribution of duplicate values, the raw size is easy to modify and a heuristic that can be applied without any specific dataset knowledge. Each of the resulting chunks is associated with exactly one relation and carries a subset of that relations rows.

Chunking is done exactly once at the very start and is not repeated for n-ary layers. We reuse the same chunks in every layer of the n-ary pIND discovery.

Since *SPIND* almost always operates on the relation layer, a hash based partitioning, similar to *BINDER*, is not feasible. For the validation, we need to descend to the attribute layer and there we need to know which attributes (attribute combinations) share at least one value. A partitioning would therefore be required to split the dataset into n groups $G_1, \ldots, G_n$ such that for every tuple of values t_i generate by some row, it holds that if $t_i \in G_j$ than

1) $\forall\, t_k \in G \setminus G_j : t_i \cap t_k = \emptyset$
2) and $t_i \cap t_k \neq \emptyset\ \forall t_k \in G_j$.

To find such groups we would need knowledge of all values in a relation. Even if we expand the groups row by row and merge when necessary, solving this grouping problem would create a lot of new complexity. This leads to the decision to use a plain horizontal splitting approach.

At the end of the chunking phase we have a set of chucks for each relation, that all carry at most *CHUNK_SIZE* values. Larger relations have been split into more chunks while smaller relations are probably contained within a single chunk.

3.2 Probabilistic Filtering

Assume we would know the set R of all values, which only occur in referred attributes. Such a set could be used to filter the values that would need to be sorted, merged and validated. How the filter is filled and where exactly it is used will be discussed in the sorting and validation sections. This section will explain the reason why such a filter can be used in the first way and which benefits it brings.

Mathematically we require the set R to hold the property $\forall\, ref \in candidates : ref \cap R = \emptyset$. Trivially, the empty set would be valid for R. Our goal is to find an R which best approximates the union of all values present in referenced attributes (attribute combinations) without the values present in any dependant attributes.

Since we may not be able to actually keep all of the values in main memory, we need a different approach of filtering the values. An important observation being, that we need to be certain, that all values which are present in at least one dependant attribute need to make it through the filter. If values which are only in dependant (or irrelevant) attributes make it through the filter, that creates more computational complexity, but will not create any incorrect results.

These constraints can be full filled by building a probabilistic filter in which all values that appear in an dependant attribute at least once are stored. Such a filter can than be queried using a value to check if that value was added. If the value was added previously it will always return true. If the value was not added it may create a false positive at a rate of typically less than 5%. There are multiple options for such filters.

Research regarding probabilistic data structures focuses on two primary metrics govern research evaluation: the computational efficiency of insertion and query operations, and the required number

dataset	values in filter
data.gov	-
musicbrainz	-

Table 1: The number of items that are inserted into the bloom filter for each dataset.

of bits for storing individual values within the structure[10]. Bloom filters are initialized with an anticipated number of elements and a target false positive rate. However, if additional values are incorporated beyond the initial expectation, Bloom filters may become oversaturated, leading to an approaching false positive rate of 1 [23]. Bloom filters use about 44% ($log_2(e)$) more space per key than the theoretical lower bound. There are probabilistic data structures which are closer to the theoretical lower bound, but these structures impose more complex strategies which are out of scope of this thesis [10].

The size of the Bloom Filter is set to 100 million expected values with a false positive rate of 1% by default. With these settings the filter will consume TODO bytes of main memory. If the available memory is very limited, the filter should be initialized with a smaller size or higher false positive rate. Table 1 shows how many values are inserted into the bloom filter during u-nary pIND discovery for each dataset. In section 3.5 we will discuss in detail how the values are inserted into the filter.

3.2.1 Procedual Refining. After each layer the set of values which are present in dependant attributes will be a subset or equal to the set of values of any of the preciding layers. If we construct the probabilistic filter after unary pINDs have been validated, we might can only masked out values based on the unary pINDs. At fours layer, the set of values that make up dependant attribute combinations might have shrunk significantly, but the filter could not accomondate for that. We could overcome this issue by reconstructing the filter during the validation of every layer. While this creates more computational effored it holds the potential of saving read and write operations during the sort and merge phases. To rebuild the filter in n-ary layer we use the inverse transformation (see Algorithm 2) to extract the strings contained in the serialized value and compute the hashes. These steps are executed in the parallized section of the validation, while the insertion into the filter needs to be synchronized between threads.

3.2.2 Filter evaluation. In Figure TODO we find the execution times of all datasets (expect the most complex ones) using the bloom filter by constructing it once during u-nary discovery (*Once*), using and refining the filter on every level (*Refining*) and not using a probabilistic filter at all (*None*). The y-Axis is relative to the longest execution time for each dataset and the displayed times are averaged over three executions for each mode. We find that using a filter that was build once or a refinded filter results in very similar execution times which are usually a bit faster than not using a filter. Since a refinded filter can save some read and write operations we will chose this version to put a smaller workload on the disk. Notice that the total execution time of *Cars* is under one second for n-ary discovery, which results in the time it takes to construct the file to become quite significant. We will not give much weight to this observation since the dataset is so quick to solve anyway.

3.3 Sorting Chunks

Our objective is to arrange each chunk in sorted order, a prerequisite for subsequent merging and validation procedures. Consistently, each chunk undergoes identical processing steps, executed concurrently.

We employ a *CSVReader*[3] for each chunk, specifying the relevant attribute combinations. Line by line processing of the chunk occurs, during which we update all associated attributes to their new value. In Section **??** we discuss exactly how the values of n-ary attributes are assigned. For now we will take the existence of a proper string value for both unary and n-ary attributes as a given. To manage these values and their associations, we employ a hash based mapping structure where values are mapped to their respective attributes along with the frequency of occurrences, thus forming a nested map. To prevent main memory overflows, we constrain the size of this map using a constant denoted as *SORT_SIZE*. If the map size reaches the specified threshold (which is divided by the number of threads) or when sorting is complete, the contained data is serialized to disk.

This storage process will first sort the key set of the outer map. Afterwards we iterate over the sorted entries and write two lines for each of them. The first line contains the string value, which is the entries key. Followed by the the attributes which formed the value combined with the occurrences. Thereby the second line persists the value of the entry. Figure 2 illustrates this workflow while sorting for unary pIND discovery. The Attributes are 1 :*id*, 2 :*postCode*, 3 :*landline*. For every spilled file, we retain the path of the sorted (sub) chunk, ultimately returning a list of these paths.

A minor optimization conduct here is spilling values based on the sum of occurrences in the connected attributes. We know that in a single chunk, there are at most *CHUNK_SIZE* unique values. In n-ary layers there might be relations where the number candidates of than relation is greater than the number of columns. Therefor we can calculate that the maximal possible number of distinct values generated from a chunk is $N = \underbrace{\lfloor \frac{CHUNK\_SIZE}{\#cloumns} \rfloor}_{=\#rows} \cdot \#candidates$, since every row can produce at most one unique value for each candidate. We can now say that $\frac{N}{SORTED\_SIZE}$ is the largest possible minimum of occurrences over all present values. The proposed heuristic suggests to keep all values with more than $\frac{\#candidates \cdot CHUNK\_SIZE}{SORTED\_SIZE}$ occurrences which is greater than $\frac{N}{SORTED\_SIZE}$ in main memory and only spill those values which where not seen often. The idea being, that values which have occurred often will be more likely to occur often. Spilling these last can save complexity during merging since less candidate occurrences need to be deserialized.

3.4 Merging (spilled) Chunks

We now have a bunch of files which are all sorted by themselves. The next step is to merge all files which originated from the same relation. Again, this is computed in parallel. To avoid too many files

[3]Provided by the open source library "Opencsv" https://opencsv.sourceforge.net/

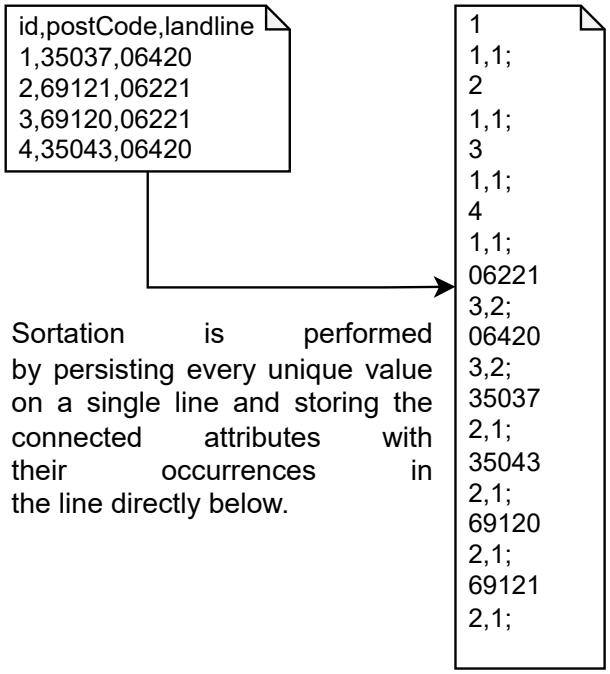

Figure 2: Simplified illustration of the sorting process.

being opened at the same time the constant *MERGE_SIZE* can be used to limit the number of files which are being sorted per thread. Due to multithreading the actual number of simultaneously opened files is $\#threads \cdot (MERGE\_SIZE + 1)$. The plus one originates since we always need to open the resulting output file of a merge as well. We attach a buffered reader to each file and use a priority queue to perform a k-way merge **??**. At the end of the merge phase, we have constructed a single sorted file for each relation.

3.5 Validation

The validation process expects a sorted file per relation. The validation process works analog to the validation performed by *SPIDER*. Given the sorted files we check which attributes (attribute combinations) share a value and move the readers forward bit by bit in a way that only those readers are updated which share the same smallest value. *SPIDER* uses a priority queue which stores the head value of each reader. Such a data structure enables us to calculate the smallest of the head values in $log(M)$ time, M being the number of attributes (attribute combinations). Assume the complexity of pruning the connected attributes to some value is $O(k)$. Validating all values N would cost $O(k \cdot N \cdot log(M))$. Pruning is a synchronous task, meaning the active candidates need to be synchronized between every prune. We will therefor try to create the attribute groups in parallel and conduct pruning in the main thread.

We introduce the last variable *VALIDATION_SIZE*. It is a fixed value which states how many values are loaded at the same time for each relation. Deserialization is computed in parallel for all readers and we calculate the smallest largest value for each relation queue after they have been refilled. This largest smallest queue value is calculate at most $\lceil \frac{N}{VALIDATION\_SIZE} \rceil$ times if all N values distributed over the different relations where distinct. The value groups are then also build in parallel using a stream architecture to efficiently include the synchronized pruning. The resulting complexity is $O(k \cdot N + \lceil M \cdot \frac{N}{VALIDATION\_SIZE} \rceil)$. Should M, the number of relations, be very large, the logarithm would yield a better theoretical complexity eventually. But since M can be expected to be much smaller than the *VALIDATION_SIZE* and the proposed version operates mostly in parallel, we find that this concept of loading groups in parallel decreases the compute time by a factor of up to five and especially helps when processing large datasets.

4 PARTIAL N-ARY CANDIDATE GENERATION

To discover n-ary pINDs, we need to generate candidates for the next layer using the already collected information of the current layer. An iterative process wherein progressively larger candidate sets are generated and verified follows the idea of an apriori-gen algorithm [1]. In related research we find proposals that only use the most recent layer to generate the candidate set for the following layer [21]. We will propose a refined version, which significantly decreases the number of false candidates that need to be validated.

As proven (partial) INDs respect projection. For any pIND of size n, all subsets of that pIND must be also valid. This property has not been used to its full potential by past research. An adjusted version for improved candidate generation needs to follow the three rule below.

Given two not necessarily different relational schema R_a and R_b with valid (n-1)-ary attributes $X \subset R_a$ and $Y \subset R_b$ and valid u-nary attributes $A \in R_a$ and $B \in R_b$, a valid n-ary candidate is formed if

1) The column index of A is greater than the column index of all $x_i \in X$.
2) B is not contained in Y.
3) All (n-1)-ary pINDs $R_a[(X \setminus X_i)A] \subseteq_\rho R_b[(Y \setminus Y_i)B] : i \in (1, \ldots, n-1)$ must be valid.
4) If R_a and R_b are the same scheme, then X, A, Y and B need to be pairwise disjoint.

Originally, B was also required to be non-empty, which makes sense when treating null as a subset. In that setting an all-null attribute will create a valid n-ary candidate with all (n-1)-ary pINDs and is therefore trivial. The proposed algorithm however offer different null interpretations under which all-empty columns might become relevant. This is why we will not require B to be non-empty.

We will further apply another step to the candidate generation which improves the original method by pruning candidates that are not possible during the generation without the need of the actual relation instances. We have showed that transitivity requires all subsets of a candidate to be valid for that candidate to be valid (Section **??**). In the original generation, this is only applied in a weak setting by ensuring $r_1[X] \subseteq_\rho r_2[Y]$ and $r_1[A] \subseteq_\rho r_2[B]$. Assume $|X| = 3$, we know that if $r_1[X_1, A] \subseteq_\rho r_2[Y_1, B]$ or $r_1[X_2, A] \subseteq_\rho r_2[Y_2, B]$ are not present in the current layer, then $r_1[X_1, X_2, A] \subseteq_\rho r_2[Y_1, Y_2, B]$ can not be valid. Therefor the proposed expansion is to ensure the subsets of the generated candidates are also valid.

Using the subset check is only sensible when generating candidates for the third or higher layer. Unary candidates are trivially generated using all available columns and the second layer is a

Input: $rDep, nDep, uDep, rRef, nRef, uRef$
Output: Whether the next level candidate is possible

```
for skip in 1, ..., |nDep| do
    dependant = rDep[(nDep \ nDep_skip), uDep]
    referred = rRef[(nRef \ nRef_skip), uRef]
    if dependant ⊈_ρ referred then
        return False
return True
```

Algorithm 1: Subset candidate check

combination of two unary pINDs which directly implies that all subsets have to be valid.

4.1 Serializing n-ary tuples.

Our unary discovery is based on strategies which relie on hashing and value comparison (sorting). By default, a list of values (tuple) is a non-hashable type. We could define a custom function and implement comparison functions for tuples. It would require us to use different functions for unary and n-ary discovery, even if they practically do the same. Using a string representation of tuples instead, enables us to use the unary procedures for n-ary pIND discovery, creating a simpler algorithmic structure. *BINDER* also uses this strategy, where the values contained in some multi-dimensional tuple are concatenated using "#" as a separator. The tuple ("*Marburg*", "35037", "06421") gets transformed to the string "*Marburg*#35037#06421". Concatenating values in such a way can produce incorrect results since there can be multiple tuples which create the same string. We search for for a bijective transformation f which accepts a tuple and creates a string. The function used by *BINDER* is not bijective since the tuples $(1\#1, 1)$ and $(1, 1\#1)$ will both produce the same string. While this proves to be highly unlikely in real datasets, we should still eliminate this point of failure. Or proposal function will take a given k dimensional tuple and first conduct a length encoding for the first $k-1$ values. These lengths are concatenated using ":", but any non-numerical character would work here. We mark the end of the length encoding using "|", again the precise choice of that character is not important. After the length encoding all values are concatenated without using a delimiter. The two tuples from before would be transformed to("1#1", "1") $\rightarrow$ "3|1#11" and ("1", "1#1") $\rightarrow$ "1|11#1". To prove that our proposal is indeed bijective, we need to make sure that every string that can be produced by our function originates form exactly one input. We will construct the inverse function f^{-1} and show that the chain $fcirclejoinf^{-}1$ is an identity function for all tuples. The inverse function will first search for the first occurrence of "|" which marks the end of the length encoding. It will then split the second part based on the length information to extract the inputs. Given a k dimensional tuple of strings $(s_1, \ldots, s_k)$, f produces the string "$len(s_1) : len(s_2) : \cdots : len(s_{k-1})|s_1 s_2 \ldots s_k$". The inverse function f^{-1} can now cut the second part at the correct indices reproducing the tuple $(s_1, \ldots, s_k)$. This function is independent of the characters present in the strings, since we only care about the first occurrence of "|" which is guaranteed to be the end of the length encoding due to the explained serialization procedures. Algorithm 2 shows how the transformation and inverse transformation can be implemented.

Input: *attributeValues*: The list of string values
Output: A String representation of all attribute values

```
if attributeValues.length == 1 then
    return attributeValues[1]
representation = ""
/* Create the length encoding                    */
for index in 1, ..., attributeValues.length - 1 do
    representation.append(attributeValues[index].length)
    representation.append(":")
representation.append("|")
/* Append the actual values                      */
for value in attributeValues do
    representation.append(value)
return representation
```

Algorithm 2: N-ary Attribute String Creation

5 HYPERPARAMETER OPTIMIZATION

There are five hyperparameters which influence the execution of *SPIND*. The size of the initially created chunks (*CHUNK*), the maximal number of values which are kept in main memory during the sorting phase per thread (*SORT*), the maximal number of files which are merged at once per thread (*MERGE*), the queue size for each relation during candidate validation (*VALIDATION*) and the degree of parallelization (*PARALLEL*). Leveraging Bayesian optimization [22], we seek to efficiently and iteratively identify parameter configurations that minimize the execution time of *SPIND* of various datasets. We lower bound *CHUNK* by 10,000 to avoid creating a massive amount of files the upper bound is set to 100 million. We limit *SORT* between 10,000 and 3.5 million to find a balance between a large number of files being written and a safe number under the main memory threshold. *MERGE* is lower bounded by two and upper bounded by 1,000 to not put too much pressure on the file system. *VALIDATION* is upper bounded by 1 million. Finlay, *PARALLEL* is upper bounded by the number of virtual threads of the executing machine.

A first observation is that *PARALLEL* has a clear negative correlation with the execution time. Regardless of how the other four parameters are set. Figure 3 shows a group of hex-bin plots, where the y-Axis is always the degree of parallelization and the x-Axis take on the other variables. After X?X iterations, our estimated multivariable function has a maximal 2σ-uncertainty of X?X. These observations lead to the decision of fixing *PARALLEL* to its upper bound.

The next iteration yields that *CHUNK* is the second most important variable. While smaller chunks strictly correlate with the total number of files being created and therefore also the I/O overhead, we find an optimum at a chunk size of five million which is robust over different datasets. A smaller chunk size enables more tasks to be processed in parallel which has been shown to overcome the I/O operations at some point.

Summarizing the last three iterations, we find that *SORT* and *VALIDATION* should be as large as possible under the given main memory constraints. The results for *MERGE* X?X

In Figure 4 we can observe the relative change in the number of files created under changing chunk sizes (left) and the relative

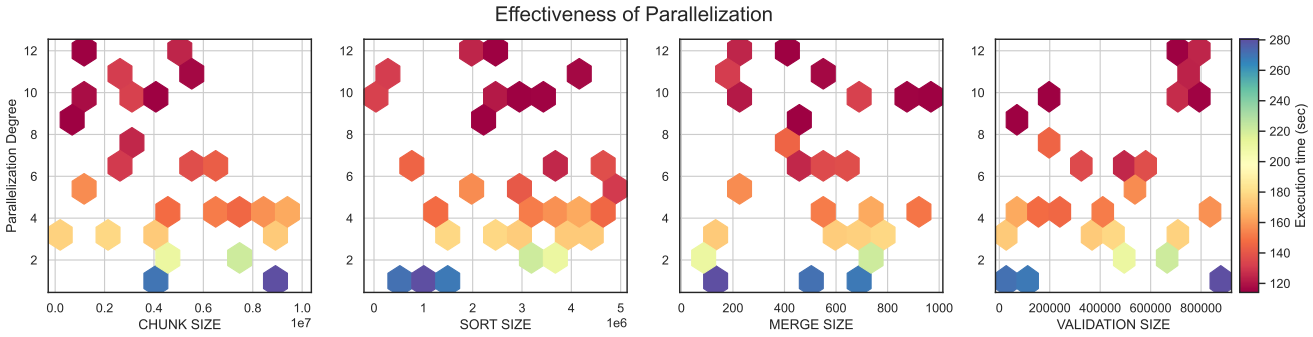

Figure 3: Effectiveness of the parallelization degree regardless of any other parameter.

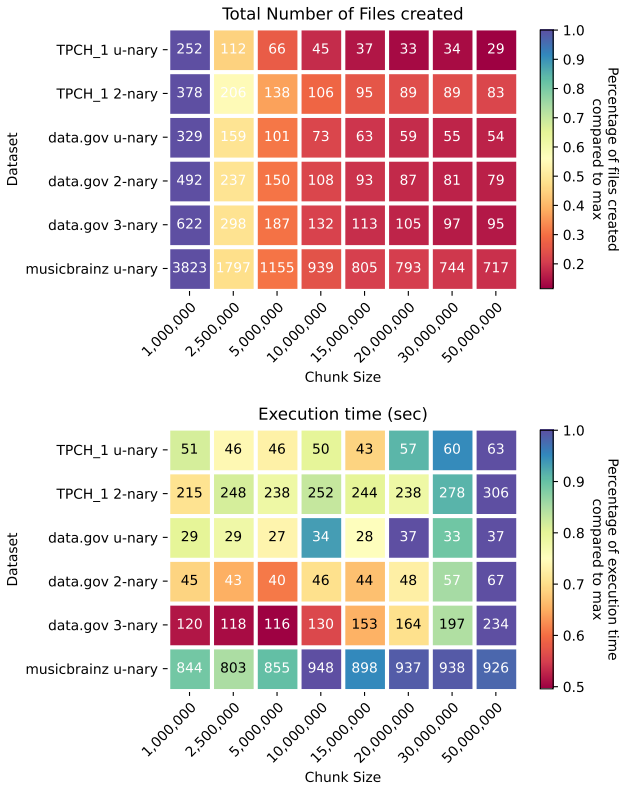

Figure 4: Files created (top) and execution time (bottom) under varying chunk sizes. Both relative to the maximum.

execution times compared to the longest run (right). For these experiments, we always average the execution of three runs.

6 DATASETS

There are many sources for csv or tsv files online spanning a range of domains. We aim to test the proposed algorithm on a collection of datasets which vary in size and the domain. Table 2 holds information regarding the characteristics of each dataset. The given numbers on INDs have be calculated by treating *NULL* as subset (see Section 2.2).

Cars is a small dataset regarding used cars and their retail prices. The different relations each focus on a single brand (e.g. Audi, Ford or Skoda). Each row of a relation contains information on a specific model of that brand.

The smallest dataset by required disk size, **ACNH**, contains data form the video game Animal Crossing: New Horizons[4]. This dataset contains information on all items, recipes and achievements in the game. It was selected size it contains a vast amount of deep (p)INDs a poses the challenge of efficient candidate handling.

T2D is a gold standard for matching web tables to DBpedia[5]. We use a subset of the 779 provided web table that span various domains. The subset includes all tables with at least five rows.

Similar to **T2D**, **WebTables** offers an even larger collection. WebDataCommons published a random sample of their Web Table Corpus[6]. We will use that Sample to evaluate the capabilities when a massive amount of relations and attributes are present. We will not try to compute n-ary pINDs for this dataset.

The **US** government and the European union (**EU**) both publicly share regional, national and international data. The two dataset that are build on these sources represent the governmental domain.

Population is a wide table containing world wide demographic data. It includes the population sizes for different ages in various countries from 1950, including predictions, up to 2025.

Musicbrainz is a repository of music knowledge. The dataset in relational form has a large number of attributes, relations and rows. It is the most computational complex of the chosen datasets.

The biological domin is covered by **UniProt**. This dataset contains vertebrate genomes aquired form Ensembl [7] in the UniProt[8] standard.

Lastly, we have the synthetic datasets **Tesma** and **TPC-H**. **Tesma** is a database generation tool developed by the Hasso Plattner Institut. Using a config file which relational constraints and the wanted size, it will create multiple csv files with the wanted structure. **TPC-H** is a benchmark for database performance. Using the generation

[4]https://animalcrossing.nintendo.com/new-horizons/
[5]https://www.dbpedia.org/
[6]https://webdatacommons.org/webtables/2015/downloadInstructions.html
[7]https://www.ensembl.org/index.html
[8]https://www.uniprot.org/

Name	Domain	Size on Disk	Relations	Attributes	Unaries	N-aries	n_{max}
Cars	Retail	6.1 MB	13	117	281	91	4
ACNH	Video-Games	3.5 MB	30	630	8,686	20,908,814	12
T2D	Benchmark	4 MB	669	3125	362,604	9,301,847	7
WebTables	Various	?	?	?	?	-	1
US	Governmental	1.2 GB	16	255	753	215,308	7
EU	Governmental	1.8 GB	37	624	18,752	54,634	6
Population	Demographics	1.7 GB	1	109	236	1	2
Musicbrainz	Entertainment	16.7 GB	171	301	1,843	?	?
UniProt	Biology	633 MB	263	2367	420,412	1,174,863	5
Tesma	Synthetic	80 MB	2	12	4	1	2
TPC-H 1	Synthetic	1 GB	7	61	96	8	2
TPC-H 10	Synthetic	?	7	61	?	?	?

Table 2: Datasets and their characteristics

tool, we have created a 1 GB version (**TPC-H 1**) and a 10 GB version (**TPC-H 10**)

To enable reproducibility, the connected GitHub repository contains a detailed technical documentation on how each dataset was acquired[9].

6.1 Synthetic Data Sets

To evaluate the proposed algorithms under detailed aspects, we will generate synthetic data sets. The strategies and claims are based on [14] synthetic data can be defined as *data that has been generated using a purpose-built mathematical model or algorithm, with the aim of solving a (set of) data science task(s).* While we will not try to train a model with the synthetic data, it is still of great use for us, since we have absolute knowledge about the underlying structures. The decision is based on the fact that there is a lot of real word data available, since open data is a growing market which expected to grow even further [9]. Synthetic data on the other hand enables us to evaluate the algorithm performances on edge cases, which we may not be able to find in the selection of real world data sets.

7 RELATED WORK

Inclusion dependencies (INDs) are a highly influential concept in both database research and practice, with a wide range of contributions and applications. The introduction already provided some insight into their diverse application areas of INDs. In this section, we will focus on the key achievements related to the implication problem of INDs. We will go over different algorithms and discuss their unique features.

In 1981 INDs started as a general notation of referential integrity, which was already a well established concept back then [8]. Casanova et al. presented a paper on the inference rules of INDs[6]. Three axioms where introduced: *reflexivity*, *transitivity* and *projection and permutation*. The application of these rules to partial INDs will later be discussed in detail (Section **??**). The paper further proofed that the discovery of INDs is PSPACE-complete if there is no limit on the size of inclusions. Publications typically fall into three groups of algorithms, foreign key discovery algorithms, unary IND discovery and n-nary IND discovery [20].

Bell and Brockhausen [4] propose a graph-based approach in 1995 to represent the relationships between attributes, allowing for a more efficient exploration of the search space. The algorithm is initiated with a directed graph, wherein all possible edges, which could not be pruned by statistical measures, are included. A directed edge in the graph represents an inclusion dependency, which is read as the edge from A_i to A_j ($A_i \rightarrow A_j$) represents the IND $A_i \subseteq A_j$. It then proceeds to remove those edges that failed the IND check. To determine the validity of an edge, the algorithm checks for transitivity, which enables it to answer whether a dependency could exist between two variables based on their relationships with a third variable, which was tested previously. If it is ascertained that a dependency is impossible, the algorithm skips the test and directly removes the given edge, thereby reducing the overall computational cost. The approach presented by Bell and Brockhausen for unary IND discovery has both reusable aspects and downsides. The algorithm's candidate generation technique, which uses data statistics such as data types and min-max values, can be reused in other discovery algorithms. This preprocessing step reduces the number of candidates that need to be validated and further reduce the storage overhead needed to store candidates. However, the validation technique used in the algorithm, which relies on SQL join-statements and requires accessing the data on disk, is infeasible for larger candidate sets. This limits the scalability of the approach and makes it less practical for large-scale data sets. Additionally, the need to store the data in a database and access it on disk for validation can add to the computational cost and time required for the discovery process.

The *SPIDER* algorithm is a disk-backed, all-column sort-merge join with early termination used for the discovery of inclusion dependencies [3]. It sorts the values of each attribute, removes duplicate values, and writes the results to disk in the first phase. In the second phase, it performs the actual inclusion dependency discovery by using a pointer for each file and validating all candidates at the same time. A major advantage is, that in this setting every value only needs to be read a single time from disk, which greatly reduces the I/O bottleneck. The Spider algorithm has been the subject of experimental evaluation and is considered one of the efficient techniques for unary IND discovery. Still, there are drawback if the data set is too big to be sorted in main memory

[9] https://github.com/Jakob-L-M/partial-inclusion-dependencies/tree/main/data

or if the number of simultaneously open files allowed by the OS system is reached [21].

In 2009 Bauckmann et al. also proposed a partialized version of *SPIDER* in a section of the same paper. The authors found that there where surprisingly many partial inclusions (under a 5% threshold) in their test data sets. To find partial inclusion dependencies, they first count how many distinct violations are present and in a second step consider the amount of duplicates for not included values. This means their algorithm does not immediately stop once a single validation has been found but only after an added counter surpasses a given threshold. The paper is not particular clear on how the number of duplicates is stored/retrieved and additionally does not analyse the computational effect of these changes and with the original source code being lost, this a approach can only be verified using a best guess approach.

The *SAWFISH* algorithm [15], published in 2023, is designed for identifying similarity inclusion dependencies (sINDs) within datasets, introducing a novel perspective on inclusion dependencies (INDs). While traditional INDs assume error-free data, *SAWFISH* incorporates a similarity measure to accommodate minor errors like typos. Given a similarity threshold ω a sIND is valid if and only if for all tuples in the left hand side there exists a tuple in the right hand side which has at least a similarity of ω under a set similarity measure. The authors used the edit distance as well as the normalized edit distance. Through preprocessing, metadata generation, and a sliding-window approach, *SAWFISH* successfully identifies and validates sIND candidates using an inverted index, providing a valuable tool for database applications despite dirty data challenges.

The term *partial inclusion dependency* also appears in another research paper [16] under a different definition. The authors use the terms partial and simple IND to describe how NULL values are handled during IND discovery.

There is no proposed approach that purely focuses on partial inclusion dependencies which is the main motivation for this thesis.

8 FUTURE WORK

We are aware that there is still potential left to improve pIND discovery. A central observation in the complex datasets with large n-ary pINDs is, that the generated candidates eventually become equal or almost equal to the valid pINDs of some layer. An idea would be to jump a few layers ahead if such a thing happens since it strongly hint towards the existence of pINDs in a (much) deeper layer. In **ACNH** we find that from the sevens to the twelves layer the generated candidates where always all valid.

The published code behind the algorithm can also be improved further. The implementation mostly relies on standard java data structures. There are libraries, such as FastUtil[10], which offer high performance versions of these structures and may be able to improve the performance. Serialization is also performed in a human-readable style, which is not optimal. Encoding the attribute ids and occurrences (integers and longs) in one and four byte blocks could decrease the file size drastically which could help when I/O bottlenecks occur.

[10]https://fastutil.di.unimi.it/

REFERENCES

[1] 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.

[2] William W Armstrong, Yatsuka Nakamura, and Piotr Rudnicki. 2002. Armstrong's axioms. *Journal of formalized Mathematics* 14, 1 (2002).

[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] 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.

[7] Edgar F Codd. 1979. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems (TODS)* 4, 4 (1979), 397–434.

[8] CJ Date. 1981. Referential integrity. In *Proceedings of the seventh international conference on Very Large Data Bases-Volume 7*. 2–12.

[9] EUOpenData. 2022. Open Data Impact - European Union. https://data.europa.eu/en/publications/open-data-impact. Accessed: 2023-12-08.

[10] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 75–88.

[11] John Gantz and David Reinsel. 2012. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future* 2007, 2012 (2012), 1–16.

[12] Jarek Gryz. 1998. Query folding with inclusion dependencies. In *Proceedings 14th International Conference on Data Engineering*. IEEE, 126–133.

[13] Alon Halevy, Anand Rajaraman, and Joann Ordille. 2006. Data integration: The teenage years. In *Proceedings of the 32nd international conference on Very large data bases*. 9–16.

[14] James Jordon, Lukasz Szpruch, Florimond Houssiau, Mirko Bottarelli, Giovanni Cherubin, Carsten Maple, Samuel N Cohen, and Adrian Weller. 2022. Synthetic Data–what, why and how? *arXiv preprint arXiv:2205.03257* (2022).

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Thorsten Papenbrock. 2017. *Data profiling-efficient discovery of dependencies*. Ph.D. Dissertation. Universität Potsdam.

[21] 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.

[22] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. 2015. Taking the human out of the loop: A review of Bayesian optimization. *Proc. IEEE* 104, 1 (2015), 148–175.

[23] 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.

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