Scalable Inclusion Dependency Discovery

Nuhad Shaabani(✉) and Christoph Meinel

Hasso-Plattner-Institut, University of Potsdam,
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
{nuhad.shaabani,christoph.meinel}@hpi.de
http://www.hpi.de

Abstract. Inclusion dependencies within and across databases are an important relationship for many applications in anomaly detection, schema (re-)design, query optimization or data integration. When such dependencies are not available as explicit metadata, scalable and efficient algorithms have to discover them from a given data instance.

We introduce a new idea for clustering the attributes of database relations. Based on this idea we have developed S-INDD, an efficient and scalable algorithm for discovering all unary inclusion dependencies in large datasets. S-INDD is scalable both in the number of attributes and in the number of rows. We show that previous approaches reveal themselves as special cases of S-INDD. We exhaustively evaluate S-INDD's scalability using many datasets with several thousands attributes and rows up to one million. The experiments show that S-INDD is up to 11x faster than previous approaches.

Keywords: Inclusion dependency · Data integration · Data profiling

1 Introduction

Dependencies are metadata that describe relationships between relational attributes. Dependencies play very important roles in database design, data quality management, and knowledge representation. In the case that they are modeled as part of the application requirements, they are then used in database normalization and are implemented in the designed database to ensure data quality. In contrast, dependencies in knowledge discovery are extracted from the existing data of the database. The extraction process is called dependency discovery and aims to find dependencies satisfied by existing data. A typical type of dependency is inclusion dependencies (INDs), which represent value reference relationships between two sets of attributes. Together with functional dependencies, they represent an important part of database semantics.

In the context of data integration, the discovery of inclusion dependencies can help to solve a very common and difficult problem: discovering foreign key constraints. There are many reasons for an absence of foreign key constraints in databases. These include a simple lack of domain knowledge within the development team during the design and development time, the worry that checking

M. Renz et al. (Eds.): DASFAA 2015, Part I, LNCS 9049, pp. 425–440, 2015.
DOI: 10.1007/978-3-319-18120-2_25

such constraints by the hosted system would hamper database performance, or the lack of support for checking foreign key constraints in the host system.

The manual search for INDs by domain experts is usually not feasible due to the large number of data sources, a widespread lack of reliable metadata about legacy databases, and the possibility of a high number of attributes in real-world relations. Therefore, efficient and scalable algorithms to detect INDs enable easy integration of new data sources that previously would not have been used, because their relationships with existing data was not known.

N-ary INDs cover pairs of n attributes, while unary INDs (uINDs) cover only pairs of single attributes (formal definitions are in Sec. 2). All known algorithms for detecting high-dimensional INDs require the discovery of all unary inclusion dependencies (single-column INDs) [10–13]. This is because any valid IND of a size greater than one implies that all unary INDs derivable from it have to be valid in the same database. This means that the reliability of the algorithms for detecting high-dimensional INDs is dependent on a scalable and efficient discovery of unary INDs.

There are three approaches in related work focused on exhaustive detecting single-column inclusion dependencies: Bell and Brockhausen [3], De Marchi et al. [11,13], and Bauckmann et al. [1,2] (see Sec. 6).

The algorithm proposed in [11,13] for discovering unary INDs uses an inverted index associating every value in the database with the set of all attributes having this value. Because for every attribute A the intersection of all attribute sets containing A is the set of all attributes including A, the algorithm runs through all values in order to compute such an intersection for every attribute. However, this approach is inefficient because an attribute set in the index can be associated with many different values. This means, the algorithm executes a lot of redundant intersection operations. These operations are very costly if the dataset has a large number of attributes sharing a lot of values.

The first research question addressed in this paper is how we can eliminate such redundant operations caused by using the inverted index. We tackle this problem by introducing the concept of *attribute clustering* (see Sec. 3).

Spider [1,2] is an external algorithm that writes the values of every attribute to a file after sorting them and removing duplicate values. Then it opens all files at once and starts comparing the values in parallel and in the same way in which the merge-sort algorithm does. During this process, Spider applies an efficient method for discarding unsatisfied unary INDs (see Sec. 6 for more details). Spider outperforms the approach proposed in [11] up to orders of magnitude [1,2]. However, the drawback in Spider's approach is its dependency on the number of attributes. This means, that by increasing the number of attributes, Spider's scalability decreases: the number of I/O- operations increases because the size of buffers allocated for the opened files becomes smaller.

The second research challenge addressed in this paper is how we can make Spider independent from the number of attributes in order to improve its scalability in two dimensions: in the number of attributes and the number of rows.

Table 1. Running example

A	B	C	D
1	1	5	1
2	2	5	1
2	3	6	3
4	4	7	3

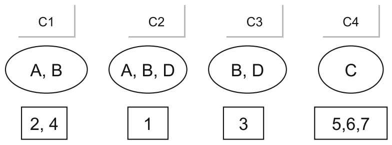

Fig. 1. Attribute clustering based on the data of table 1

We tackle this challenge by devising S-INDD, a scalable approach for computing the *attribute clustering* (see Sec. 4).

Every cluster in the *attribute clustering* is a subset of attributes sharing a subset of values that can not be shared by the attributes of any different cluster in the *attribute clustering*. E.g., $\{\{A, B\}, \{A, B, D\}, \{B, D\}, \{C\}\}$ shown in figure 1 is the *attribute clustering* over the values of table 1. Attributes A and B shape cluster $\mathcal{C}_1$ because both share the values $\{2, 4\}$ that can not be shared by the attributes of $\mathcal{C}_2$, $\mathcal{C}_3$, or $\mathcal{C}_4$. Every attribute of the attribute set, denoted by $\mathcal{A}$, must be contained in at least one cluster.

Clustering the attributes in this way allows us to derive the following inference rule[1]: Attribute X is included in attribute Y if and only if every cluster containing X contains Y. E.g., the set of D's values in table 1 is included in the set of B's values because the clusters $C2$ and $C3$ that both contain D also contain B, but B's values are not included in D's values because cluster $C1$ contains B and does not contain D.

For every attribute A, S-INDD stores all elements of the set $\mathcal{V}_A \times \{\{A\}\}$ ($\mathcal{V}_A$ denotes the value set of A) as a sorted list in an external repository. Then, for every value $v \in \mathcal{V}$ ($\mathcal{V}$ denotes the whole set of values in the dataset), S-INDD computes incrementally the set of attributes, denoted by $\mathcal{A}^v$, whose value sets contain v. The incremental computing of the sets $\mathcal{A}^v$ is achieved by executing a sequence of merging operations. Every merging operation merges simultaneously k lists from the repository ($k > 1$ is a given number) and replaces them with a new list. The new list contains the union of all sets $\mathcal{A}^v$ contained in the k lists read previously. In this way the sets $\mathcal{A}^v$ are incrementally computed. Such merging continues until the repository contains less than k lists. After finishing merging, S-INDD generates the clusters from the remaining lists by processing them in parallel. The possibility that S-INDD can control the number of lists to be merged makes its scalability independent from the number of attributes.

To handle a large dataset with a very large number of rows S-INDD partitions the whole dataset and computes the attribute clustering of every partition. The whole attribute clustering is then the union of all attribute clusterings of all partitions (see Sec. 4.2). This method makes the S-INDD's scalability independent from the number of rows.

[1] This rule is a generalization of property 1 formulated in [11]

Contributions. (1) We introduce the concept of *attribute clustering*, a new concept for inferencing all unary inclusion dependencies much more efficiently than using the inverted index introduced in [11].

(2) We devise S-INDD, a scalable algorithm for computing the *attribute clustering* in large datasets. Its scalability neither dependents on the number of attributes nor on the number of rows.

(3) We experimentally validate S-INDD on real and synthetic datasets and compare it with SPIDER [1,2]. The results show that S-INDD is up to 11x faster than Spider. Furthermore, we show that SPIDER is a special case of S-INDD.

2 Preliminaries

Let $\mathcal{A}$ be a finite set of attributes. Each attribute $A \in \mathcal{A}$ has an associated domain $dom(A)$, which defines the set of all its possible values. For $A_1, A_2, \dots A_n \in \mathcal{A}$ and for a symbol R, $R[A_1, A_2, \dots, A_n]$ is called a relational schema over $A_1, A_2, \dots, A_n$ and R is the relation name. A tuple t over R is an element from $dom(A_1) \times dom(A_2) \times \dots \times dom(A_n)$. For a tuple t over R and $X \subseteq \mathcal{A}$, we use $t[X]$ to denote the projection of t to X. A finite set r of tuples over R is called an instance of R. For an instance r of R and for a sequence X of attributes in R, the projection of r onto X, denoted by $\pi_X(r)$, is defined as $\pi_X(r) = \{t[X] \mid t \in r\}$.

A set $\mathcal{R}$ of relational schemata $R_i[A_{i,1}, \dots, A_{i,n_i}]$, where $A_{i,1}, \dots, A_{i,n_i} \in \mathcal{A}$, $1 \leq n_i \leq |\mathcal{A}|$ and $1 \leq i \leq m = |\mathcal{R}|$, is called a database schema. A relational database instance $\mathcal{D}$ over $\mathcal{R}$ is a set of instances r_i over each $R_i \in \mathcal{R}$.

Definition 1. *(Inclusion dependency) Let $R_i[A_{i,1}, \dots, A_{i,n_i}]$ and $R_j[A_{j,1}, \dots, A_{j,n_j}]$ be two relational schemata. Let X be a set of k distinct attributes from R_i and Y a set of k distinct attributes from R_j, with $1 \leq k \leq \min(n_i, n_j)$. An* inclusion dependency *(IND) is an assertion of the form $R_i[X] \subseteq R_j[Y]$ where k is the size of the IND. For $k = 1$ the inclusion dependency is called a unary inclusion dependency (uIND).*

Definition 2. *(IND satisfaction) Let $\mathcal{D}$ be a database over a database schema $\mathcal{R}$. An inclusion dependency $R_i[X] \subseteq R_j[Y]$ over $R_i, R_j \in \mathcal{R}$ is* satisfied *or* valid *in $\mathcal{D}$ iff $\forall u \in r_i, \exists v \in r_j$ such that $u[X] = v[Y]$.*

Thus, a satisfied IND $R_i[X] \subseteq R_j[Y]$ states that every value combination for attribute set X in relation R_i is also present as a value combination of attribute set Y in R_j. INDs are a prerequisite for foreign keys, and their discovery is particularly helpful to understand how records of two relations might be joined.

To simplify the formulation of the algorithm, we assume without loss of generality that attribute names are unique across all relations. Under this assumption, we can denote a unary inclusion dependency $R_i[A] \subseteq R_j[B]$ by $A \subseteq B$. We also define the two sets $\mathcal{V}_A$ and $\mathcal{V}$ to ease notation:

$\mathcal{V}_A$ is the set of A's values occurring in the corresponding instance of the relation schema in which A occurs:

$$\mathcal{V}_A = \{v \in dom(A) \mid \exists R \in \mathcal{R} : A \in R \wedge v \in \pi_A(r)\}$$

Then $\mathcal{V}$ is the set of all values of all attributes occurring in the database instance.

$$\mathcal{V} = \cup_{R_i \in \mathcal{R}} \cup_{A \in R_i} \mathcal{V}_A$$

It is now obvious that a unary inclusion dependency $A \subseteq B$ is valid if and only if $\mathcal{V}_A \subseteq \mathcal{V}_B$. Accordingly, the discovery of all valid unary inclusion dependencies in a database over a database schema $\mathcal{R}$ is equivalent to the computation of the following set: $\mathcal{I} = \{A \subseteq B \mid A, B \in \mathcal{A} \wedge \mathcal{V}_A \subseteq \mathcal{V}_B\}$

3 Attribute Clustering

We now formally introduce the concept of *attribute clustering.*

Definition 3. *(Attribute Clustering) The set $\mathcal{AC} \subseteq 2^{\mathcal{A}}$, where $\mathcal{AC} \neq \emptyset$ and $2^{\mathcal{A}}$ is the power set of $\mathcal{A}$, is an* attribute clustering *over $\mathcal{V}$ if there is a surjective function that maps every value $v \in \mathcal{V}$ to a $\mathcal{C} \in \mathcal{AC}$ that contains all attributes $A \in \mathcal{A}$ with $v \in \mathcal{V}_A$. In other words, $\mathcal{AC}$ is an* attribute clustering *if there is $f : \mathcal{V} \rightarrow \mathcal{AC}$ satisfying the following condition:*

1. *$(\forall \mathcal{C} \in \mathcal{AC})(\exists v \in \mathcal{V}) : f(v) = \mathcal{C}$ (i.e., f is surjective).*
2. *$(\forall v \in \mathcal{V})(\neg \exists A \in \mathcal{A}) : v \in \mathcal{V}_A \wedge A \notin f(v)$ (i.e., $f(v)$ is the maximal set of attributes $A \in \mathcal{A}$ with $v \in \mathcal{V}_A$).*

Each $\mathcal{C} \in \mathcal{AC}$ is called a cluster*. Clusters need not be mutually disjoint.*

The next lemma shows the relationship between the clusters and the values of the dataset.

Lemma 1. *An attribute clustering $\mathcal{AC} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_c\}$ divides the set $\mathcal{V}$ into $|\mathcal{AC}|$ disjoint partitions $\mathcal{P}_1, \mathcal{P}_2 \ldots, \mathcal{P}_c$ so that for every cluster $\mathcal{C}_i \in \mathcal{AC}$ there is a partition $\mathcal{P}_i$ with $\mathcal{P}_i \subseteq \cap_{A \in \mathcal{C}_i} \mathcal{V}_A$.*

Proof. According to definition 3, there is a surjective function $f : \mathcal{V} \rightarrow \mathcal{AC}$ where $f(v) = \mathcal{C}$ is the set of the all attributes A with $v \in \mathcal{V}_A$. For each cluster $\mathcal{C}_i (1 \leq i \leq c)$, we can define the set

$$\mathcal{P}_i = f^{-1}(\mathcal{C}_i) = \{v \in \mathcal{V} \mid f(v) = \mathcal{C}_i\} \subseteq \cap_{A \in \mathcal{C}_i} \mathcal{V}_A \tag{1}$$

because f is surjective.
Because any $v \in \mathcal{V}$ can not be mapped to two different clusters, we have

$$\mathcal{P}_i \cap \mathcal{P}_j = \emptyset \text{ for } i \neq j (1 \leq i, j \leq c) \tag{2}$$

Because there is $\mathcal{P}_i (1, \leq i \leq c)$ for any $v \in \mathcal{V}$, we have

$$\cup_{1 \leq i \leq c} \mathcal{P}_i = \mathcal{V} \tag{3}$$

According to (2) and (3), the sets $\mathcal{P}_1, \mathcal{P}_2 \ldots \mathcal{P}_c$ are disjoint partitions of $\mathcal{V}$. □

The next lemma states that for each two different attributes A, B, the set of A's values is included in the set of B's values if and only if the intersection of all clusters containing A contains B. In other words, we have the following inference rule: for any attribute A, the set of all attributes including A is the intersection of all clusters containing A.

Lemma 2. *Let $\mathcal{AC} = \{\mathcal{C}_1, \dots, \mathcal{C}_c\}$ be an attribute clustering over $\mathcal{V}$. Then the following holds:*

$$\forall A, B \in \mathcal{A} : \mathcal{V}_A \subseteq \mathcal{V}_B \Leftrightarrow B \in \cap_{\mathcal{C} \in \mathcal{AC}, A \in \mathcal{C}} \mathcal{C}$$

Proof. 1) "$\Rightarrow$": We assume $B \notin \cap_{A \in \mathcal{C}} \mathcal{C}$. This means, there is $\mathcal{C}$ with $A \in \mathcal{C}$ and $B \notin \mathcal{C}$. According to definition 3, there is at least $v \in \mathcal{V}$ mapped to $\mathcal{C}$ with $v \in \mathcal{V}_A$ and $v \notin \mathcal{V}_B$ because $A \in \mathcal{C}$ and $B \notin \mathcal{C}$. This means, $\mathcal{V}_A \nsubseteq \mathcal{V}_B$, contradicting $\mathcal{V}_A \subseteq \mathcal{V}_B$.

2) "$\Leftarrow$": We assume $\mathcal{V}_A \nsubseteq \mathcal{V}_B$. This means, there is at least $v \in \mathcal{V}$ with $v \in \mathcal{V}_A$ and $v \notin \mathcal{V}_B$. According to definition 3, v can only be mapped to a cluster $\mathcal{C}$ containing all attributes whose value sets contain v. This means, $A \in \mathcal{C}$ and $B \notin \mathcal{C}$ because $v \in \mathcal{V}_A$ and $v \notin \mathcal{V}_B$. This means, $B \notin \cap_{A \in \mathcal{C}} C$, contradicting $B \in \cap_{A \in \mathcal{C}} C$ □

We can now formulate the motivation for the introduction of the concept of *attribute clustering* as the answer of the following question.

Why is the deriving of all unary INDs from the *attribute clustering* much more efficient than deriving them from the inverted index?

Let $\mathcal{AC} = \{\mathcal{C}_1, \dots, \mathcal{C}_c\}$ be an *attribute clustering* and let $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_c\}$ be the partitions defined by its clusters (see lemma 1). The inverted index defined in [11] can now be formulated as $\mathbb{B} = \cup_{1 \le i \le c}(\mathcal{P}_i \times \{\mathcal{C}_i\})$. Furthermore, let $\mathcal{I}_A$ be the set[2] of all attributes including A. $\mathcal{I}_A$ is initially initialized with $\mathcal{A}$ in [11]. For every subset $\mathbb{B}_i = \mathcal{P}_i \times \{\mathcal{C}_i\} \subseteq \mathbb{B}$, the algorithm in [11] must run through $|\mathcal{P}_i|$ iterations in order to compute the set $\cap_{(v,\mathcal{C}_i) \in \mathbb{B}_i} \mathcal{C}_i \cap \mathcal{I}_A$. However, from all $|\mathcal{P}_i|$ intersections we need only to compute one intersection because the result of the remaining $|\mathcal{P}_i| - 1$ intersections is known, namely the set $\mathcal{C}_i$ itself. This means, using the clusters allows us to save $\Sigma_{1 \le i \le c}|\mathcal{P}_i| - |\mathcal{AC}| = |\mathcal{V}| - |\mathcal{AC}|$ redundant intersection operations compared to using the inverted index. Such intersection operations are very costly if we have a large dataset with a large number of attributes sharing a lot of values.

In fact, the runtime for computing the set $\mathcal{I}$ by using the inverted index is $\mathcal{O}(|\mathcal{V}| \times |\mathcal{A}|^2)$ while it is $\mathcal{O}(|\mathcal{AC}| \times |\mathcal{A}|^2)$ by using the *attribute clustering* (see line 5 in algorithm 1 in Sec. 4.1).

Furthermore, the way in which the inverted index has to be computed and presented has a big impact on the efficiency and the scalability of the algorithm in [11]. However, there is no explicit method suggested in [11] for computing the inverted index (one can only assume that it is computed in [11] as a kind of dictionary data structure presented in the main memory).

[2] This set is denoted as $rhs(A)$ in [11]

The scalable computing of the *attribute clustering* is the main objective of S-INDD's development.

The following lemma shows that the *attribute clustering* exists for every database instance $\mathcal{D}$. Its proof can be considered as the proof of S-INDD's correctness because S-INDD incrementally computes the sets $\mathcal{A}^v$ $(v \in \mathcal{V})$ defined in the proof and then generates the set $\mathcal{AC}$ (see Sec. 4.1).

Lemma 3. *For any database instance $\mathcal{D}$ over a database schema $\mathcal{R}$, there always exists an attribute clustering to satisfy Definition 3.*

Proof. For every value $v \in \mathcal{V}$, let $\mathcal{A}^v$ be the set of all attributes A whose values sets contain v. I.e.,

$$\forall A \in \mathcal{A}^v : v \in \mathcal{V}_A \text{ and } \neg\exists A' \in \mathcal{A} : v \in \mathcal{V}_{A'} \wedge A' \notin \mathcal{A}^v \tag{4}$$

For all values $v_{i_1}, v_{i_2}, \ldots, v_{i_j} (1 \le i, j \le |\mathcal{V}|)$ with $\mathcal{A}^{v_{i_1}} = \mathcal{A}^{v_{i_2}} = \cdots = \mathcal{A}^{v_{i_j}}$, we replace the sets $\mathcal{A}^{v_{i_1}}, \mathcal{A}^{v_{i_2}}, \ldots, \mathcal{A}^{v_{i_j}}$ with a set $\mathcal{C}_i$, i.e. $\mathcal{C}_i = \mathcal{A}^{v_{i_1}} = \cdots = \mathcal{A}^{v_{i_j}}$. We show now that the set

$$\mathcal{AC} = \{\mathcal{C}_1, \ldots, \mathcal{C}_c\} = \{\mathcal{C} \mid \exists v \in \mathcal{V} : \mathcal{C} = \mathcal{A}^v\}$$

is an *attribute clustering*:
Assuming, for a $v \in \mathcal{V}$, there are two different sets $\mathcal{C}_i$ and $\mathcal{C}_j$ with at least a common attribute A satisfying $v \in \mathcal{V}_A$. That contradicts (4) and consequently, the construction of the sets $\mathcal{C}_i (1 \le i \le c)$. This means, our assumption is wrong. This means, the function

$$f : \mathcal{V} \rightarrow \{\mathcal{C}_1, \ldots, \mathcal{C}_c\} \text{ with } f(v) = \mathcal{C} \text{ where } \mathcal{C} = \mathcal{A}^v$$

satisfies definition 3. □

The next lemma allows us to increase the scalability of S-INDD in the case of having datasets with a large number of rows (see Sec. 4.2).

Lemma 4. *Let $\mathcal{V}_1, \ldots, \mathcal{V}_n$ be disjoint partitions of the set $\mathcal{V}$ and let $\mathcal{AC}_1, \ldots, \mathcal{AC}_n$ be the corresponding attribute clusterings. Then $\cup_{1 \le i \le n} \mathcal{AC}_i$ is an attribute clustering over $\mathcal{V}$.*

Proof. For any $\mathcal{AC}_i$ $(1 \le i \le n)$ we can define a function $f_i : \mathcal{V}_i \rightarrow \mathcal{AC}_i$ satisfying definition 3 because $\mathcal{AC}_i$ is an attribute clustering over $\mathcal{V}_i$. Based on these functions and on the fact that the sets $\mathcal{V}_i$ $(1 \le i \le n)$ are disjoint partitions of $\mathcal{V}$, we define the function:

$$f : \mathcal{V} \rightarrow \cup_{1 \le i \le n} \mathcal{AC}_i \text{ with } \forall v \in \mathcal{V} : f(v) = f_i(v) \text{ iff } v \in \mathcal{V}_i$$

Obviously, f satisfies definition 3. This means, $\mathcal{AC} = \cup_{1 \le i \le n} \mathcal{AC}_i$ is an attribute clustering over $\mathcal{V}$. □

4 Algorithm

4.1 S-indd

Overall Idea. As an external algorithm (see algorithm 1), S-INDD uses a repository on a hard drive (as an external memory) in order to store temporary computation results. The input parameter $\mathcal{L}$ denotes the name of the repository. $\mathcal{L}$ contains initially the lists $L_1, L_2, \ldots, L_{|\mathcal{A}|}$ where every list $L \in \mathcal{L}$ relates to a different attribute $A \in \mathcal{A}$ and its elements are all elements of the set $\mathcal{V}_A \times \{\{A\}\}$ sorted according to the values in $\mathcal{V}_A$. Example 1 illustrates these data structures.

```
   Input   : $\mathcal{L}$, $\mathcal{A}$, $k$
   Output  : $\mathcal{I}$
1  while ($\mathcal{L}$ contains $k$ or more than $k$ lists) do
       mergeLists($\mathcal{L}$, $k$)
3  $\mathcal{AC} \leftarrow$ computeAttClustering($\mathcal{L}$)
   $\mathcal{I} \leftarrow \emptyset$
5  foreach $A \in \mathcal{A}$ do
       $\mathcal{I}_A \leftarrow \bigcap_{\mathcal{C} \in \mathcal{AC} \wedge A \in \mathcal{C}} \mathcal{C}$
7  foreach $A \in \mathcal{A}$ do
       foreach $B \in \mathcal{I}_A$ do
           $\mathcal{I} \leftarrow \mathcal{I} \cup \{A \subseteq B\}$
```

Algorithm 1. S-indd

```
    Input   : $\mathcal{L}$, $k$
1   $L_1, L_2, \ldots, L_k \leftarrow$ selectLists($\mathcal{L}, k$)
2   $Queue \leftarrow$ createPriorityQueue(
                    $L_1, L_2, \ldots, L_k, \mathcal{L}$)
    $L \leftarrow [\,]$
5   while $Queue$·size()$\neq 0$ do
        $(v, \mathcal{AS}) \leftarrow$ readNextAttSets()
        $\mathcal{C} \leftarrow \bigcup_{\mathcal{A}^v \in \mathcal{AS}} \mathcal{A}^v$
        $L \leftarrow L + [(v, \mathcal{C})]$
9   remove($L_1, L2, \ldots, L_k, \mathcal{L}$)
10  write($L, \mathcal{L}$)
```

Algorithm 2. mergeLists

Example 1. Using the data of table 1, repository $\mathcal{L}$ will be initialized with the following four lists:

$$L_1 = [(1, \{A\}), (2, \{A\}), (4, \{A\})], \quad L_2 = [(1, \{B\}), (2, \{B\}), (3, \{B\}), (4, \{B\})]$$
$$L_3 = [(5, \{C\}), (6, \{C\}), (7, \{C\})], \quad L_4 = [(1, \{D\}), (3, \{D\})]$$

The purpose of these data structures is to compute the sets $\mathcal{A}^v$ ($v \in \mathcal{V}$) incrementally, where $\mathcal{A}^v$ is the set of all attributes $A \in \mathcal{A}$ whose values sets contain v (i.e., $v \in \mathcal{V}_A$). After computing the sets $\mathcal{A}^v$, S-INDD generates the set $\{\mathcal{C} \mid \exists v \in \mathcal{V} : \mathcal{C} = \mathcal{A}^v\}$ which is, according to the constructive proof of lemma 3, an *attribute clustering*. Having the *attribute clustering*, S-INDD computes for every attribute the intersection of all clusters containing it (line 5). The set $\mathcal{I}$, the set of all uINDs, is then computed based on lemma 2 (line 7).

The incremental computing of the sets $\mathcal{A}^v$ is achieved in two stages. The first stage (line 1) consists of a sequence of merging operations. The second stage (line 3) implicitly completes the computation of the sets $\mathcal{A}^v$ and generates the *attribute clustering*.

Merging. The merging operation reads k ($2 \leq k \leq |\mathcal{A}|$) lists

$$L_1 = [(v_{11}, \mathcal{A}^{v_{11}}), \ldots, (v_{1l_1}, \mathcal{A}^{v_{1l_1}})], \ldots, L_k = [(v_{k1}, \mathcal{A}^{v_{k1}}), \ldots, (v_{kl_k}, \mathcal{A}^{v_{kl_k}})]$$

from $\mathcal{L}$ and then replaces them with the new list

$$L = [(v_1, \mathcal{A}^{v_1}), (v_2, \mathcal{A}^{v_2}), \ldots, (v_n, \mathcal{A}^{v_n})]$$

that satisfies the following condition:

$$v_1 = \min_{\substack{1 \le i \le k \\ 1 \le j \le l_i}} \{v_{il_j}\}, \qquad \mathcal{A}^{v_1} = \bigcup_{\substack{v_{il_j} = v_1 \\ 1 \le i \le k \\ 1 \le j \le l_i}} \mathcal{A}^{v_{il_j}}$$

$$\vdots$$

$$v_s = \min_{\substack{1 \le i \le k \\ 1 \le j \le l_i}} \{v_{il_j}\} \setminus \{v_1, \ldots, v_{s-1}\}, \qquad \mathcal{A}^{v_s} = \bigcup_{\substack{v_{il_j} = v_s \\ 1 \le i \le k \\ 1 \le j \le l_i}} \mathcal{A}^{v_{il_j}}$$

with $s = 2, \ldots, n$

In other words, the new list L is sorted according to the values $v_s \in \{v_{il_j} \mid 1 \le i \le k, 1 \le j \le l_i\}$ $(1 \le s \le n)$ and every set $\mathcal{A}^s$ is the union of all sets $\mathcal{A}^{v_{il_j}}$ identified by the value v_s in the k lists.

S-INDD repeats the merging operation (line 1) until the repository $\mathcal{L}$ has less than k lists where every new list generated by the merging operation has to be stored as a temporary result in the repository $\mathcal{L}$ (line 10 in algorithm 2). Example 2 illustrates the merging operation.

Example 2. According to example 1 and for $\underline{k = 3}$, S-INDD has to execute only one merging operation.
If the first three lists $L_1, L_2,$ and L_3 (see line 1 in algorithm 2) are selected for merging, the following list

$$L_{1,2,3} = [(1, \{A, B\}), (2, \{A, B\}), (3, \{B\}), (4, \{A, B\}), (5, \{C\}), (6, \{C\}), (7, \{C\})]$$

will be generated and the repository $\mathcal{L}$ will be changed to contain only the lists: $L_{1,2,3}$ and L_4.

For an efficient implementation of the merging operation and for managing a simultaneous reading of k lists (files) from the repository $\mathcal{L}$, a priority queue is used by algorithm 2 (and also by algorithm 3 - see below). The queue manages k readers (sequential file readers). Every reader is associated with a list and points to the entry that can currently be read from the list. For every two readers r, r', reader r has a higher priority than r' if and only if the value v in $(v, \mathcal{A}^v)$ is smaller than or equal to the value v' in $(v', \mathcal{A}^{v'})$ where $(v, \mathcal{A}^v)$ is the entry that r can currently read and $(v', \mathcal{A}^{v'})$ is the entry that r' can currently read.

The purpose of using a priority queue is to enable an efficient collecting of all sets $\mathcal{A}^v_1, \ldots, \mathcal{A}^v_{l_v}$ $(1 \le l_v \le k)$ by a simultaneous and sequential reading of k lists where v is the smallest value among all values that have not been read from the k lists in the queue yet. That is possible in a simultaneous sequential reading because the lists are sorted according to the values $v \in \mathcal{V}$ and the priority in

```
Input   : $\mathcal{L}$
Output  : $\mathcal{AC}$
$Queue$ ← createPriorityQueue($\mathcal{L}$)
$\mathcal{AC} \leftarrow \emptyset$
while $Queue$·size()$\neq 0$ do
    $(v, \mathcal{AS})$ ← readNextAttSets()
    $\mathcal{C} \leftarrow \bigcup_{\mathcal{A}^v \in \mathcal{AS}} \mathcal{A}^v$
    $\mathcal{AC} \leftarrow \mathcal{AC} \cup \{\mathcal{C}\}$
```

Algorithm 3. computeAttClustering

```
Input   : $Queue$
Output  : $(v, \mathcal{AS})$
$\mathcal{AS} \leftarrow \emptyset$
repeat
    $r \leftarrow Queue$ ·pull()
    $(v, \mathcal{A}^v)$ ← $r$·readCurrent()
    $(v, \mathcal{AS}) \leftarrow (v, \mathcal{AS} \cup \{\mathcal{A}^v\})$
    if $r$·hasNext() then
        $r$·readNext()
        $Queue$·add($r$)
    $r' \leftarrow Queue$ ·peek()
    $(v', \mathcal{A}^{v'})$ ← $r'$·readCurrent()
until $(Queue$·size()$= 0) \vee (v' \neq v)$
```

Algorithm 4. readNextAttSets

the queue is defined according to the ascending order of the values. This kind of applying the priority queue is well-known by external merge-sort algorithms.

Clusters Computing. After finishing the merging, algorithm 3 will generate the clusters of the *attribute clustering* $\mathcal{AC}$ by processing all remaining k' $(1 \leq k' < k)$ lists simultaneously. For every value v, there are still l_v $(1 \leq l_v < k')$ lists containing entries of the form $(v, \mathcal{A}_i^v)$ $(1 \leq i \leq l_v)$. Algorithm 3 collects all these entries, computes the set $\mathcal{C} = \cup_{1 \leq i \leq l_v} \mathcal{A}_i^v$, and adds $\mathcal{C}$ as a cluster to the set $\mathcal{AC}$. Example 3 illustrates the computing of the clusters.

Example 3. According to example 2 and for $\underline{k=3}$, $\mathcal{L}$ will contain the lists

$$L_{1,2,3} = [(1, \{A, B\}), (2, \{A, B\}), (3, \{B\}), (4, \{A, B\}), (5, \{C\}), (6, \{C\}), (7, \{C\})]$$
$$L_4 = [(1, \{D\}), (3, \{D\})]$$

after finishing the merging.
For the value $v = 1$ there are two entries: $(1, \{A, B\})$ in $L_{1,2,3}$ and $(1, \{D\})$ in L_4. Therefore, algorithm 3 collects the two sets $\{A, B\}$ and $\{D\}$ by calling algorithm 4 in the first run of the *while*-loop which delivers the tuple: $(1, \{\{A, B\}, \{D\}\})$. The first cluster is then $\mathcal{C}_1 = \{A, B\} \cup \{D\} = \{A, B, D\}$ and consequently $\mathcal{AC} = \{\{A, B, D\}\}$. After a second run of the *while*-loop we have $\mathcal{AC} = \{\{A, B, D\}, \{A, B\}\}$. Calling algorithm 4 in the third run of the *while*-loop delivers the tuple: $(3, \{\{B\}, \{D\}\})$. Consequently, $\mathcal{AC}$ will be extended to $\mathcal{AC} = \{\{A, B, D\}, \{A, B\}, \{B, D\}\}$. Computing $\mathcal{AC}$ will be finished after the seventh run of the *while*-loop resulting in $\mathcal{AC} = \{\{A, B, D\}, \{A, B\}, \{B, D\}, \{C\}\}$.

Repository Size. During the whole process of computing the *Attribute Clustering*, the repository size remains almost constant. This is because (i) the selected k lists in every merging operation will not be needed any more after merging them, which allows algorithm 2 to remove them from the repository after merging them (see line 9), and (ii) the size of the new list that results from merging the selected k lists can not exceed the total size of these k lists.

We can now answer the following question.

Why Spider [1] is a special case of S-indd? Spider can only process the whole set of the attributes at once. That means, Spider is only a form of algorithm 3. To let S-indd process all attributes at once we need only to put $k = |\mathcal{A}| + 1$.

```
    Input   : $\mathcal{L}_1, \ldots, \mathcal{L}_p, \mathcal{A}, k$
    Output  : $\mathcal{I}$
    $\mathcal{AC} \leftarrow \emptyset$
2   for $i \leftarrow 1$ to $p$ do
        while ($\mathcal{L}_i$ contains $k$ or more than $k$ lists) do
            mergeLists($\mathcal{L}_i$, $k$)

        $\mathcal{AC}_i \leftarrow$ computeAttClustering($\mathcal{L}_i$)
6       $\mathcal{AC} \leftarrow \mathcal{AC} \cup \mathcal{AC}_i$
    foreach $A \in \mathcal{A}$ do
        $\mathcal{I}_A \leftarrow \bigcap_{\mathcal{C} \in \mathcal{AC} \wedge A \in \mathcal{C}} \mathcal{C}$
    foreach $A \in \mathcal{A}$ do
        foreach $B \in \mathcal{I}_A$ do
            $\mathcal{I} \leftarrow \mathcal{I} \cup \{A \subseteq B\}$
```

Algorithm 5. Extended S-indd

```
    Input   : $L_A$ for every $A \in \mathcal{A}$
    Input   : $m_1, \ldots, m_{p-1}$
    Output  : $\mathcal{L}_1, \ldots, \mathcal{L}_p$
    for $i \leftarrow 1$ to $p - 1$ do
        $\mathcal{L}_i \leftarrow \emptyset$
        foreach $L_A$ with $L_A \neq \emptyset$ do
4           $L_A^i \leftarrow$ getSubList($L_A, m_i$)
            $\mathcal{L}_i \leftarrow \mathcal{L}_i \cup \{L_A^i\}$
            $L_A \leftarrow L_A \setminus L_A^i$

    $\mathcal{L}_p \leftarrow \emptyset$
    foreach $L_A$ with $L_A \neq \emptyset$ do
        $\mathcal{L}_p \leftarrow \mathcal{L}_p \cup \{L_A\}$
```

Algorithm 6. Partition

4.2 Extending S-indd

In the case that the dataset is very large and its values are shared among a lot of attributes, many temporary lists generated by the merging operation in subsequent iterations will have a relatively large size. Processing such large lists by algorithm 2 or algorithm 3 may demand more I/O-operations.

To avoid generating large temporary lists in this case, the dataset can be partitioned into disjoint partitions, and the *attribute clustering* will then be, according to lemma 4, the union of all clusters computed for all partitions.

Algorithm 5 is an implementation of this idea and consists of computing iterations whose number equals the number of the partitions of the dataset. Every iteration is an instance of S-indd applied for computing the *attribute clustering* over a different partition. The *attribute clustering* over the whole dataset is computed based on lemma 4 in line 6. The input of algorithm 5 contains the names of p repositories $\mathcal{L}_i$ $(1 \leq i \leq p)$ where every repository corresponds to a different partition and contains the initial data structures (lists) generated from the corresponding partition.

The disjunction of the partitions has an important computational advantage. It avoids redundant computation of the set $\mathcal{A}^v$ of any value $v \in \mathcal{V}$. However, the important question arising now is how can we partition a dataset to meet the requirement of the extended version of S-indd (algorithm 5)? The answer to this question is given in algorithm 6.

The main idea applied by algorithm 6 to partition the dataset is to choose p values $m_1, \ldots, m_p$ with $m_1 < m_2 < \cdots < m_p$(to ease notation and formulation, we put $m_p = \infty$) and then to divide every initial list L_A into disjoint sublists

L_A^i $(1 \leq i \leq p)$ where every sublist L_A^i has to satisfy the following condition:

$$\max\{v \mid (v, \{A\}) \in L_A^i\} \leq m_i$$

In other words, the maximal value in partition i does not exceed the value m_i.

The disjunction of the partitions is guaranteed by algorithm 6 because (i) the lists L_A are sorted, (ii) every sublist L_A^i is generated from L_A by processing L_A from the first element until all elements from it have been obtained that are less or equal to m_i (line 4 in algorithm 6), and (iii) after its generating and adding to the repository $\mathcal{L}_i$, L_A^i will be removed from L_A. Example 4 illustrates the extended version of S-INDD.

Example 4. Using the lists in example 1 and for $m_1 = 3$, algorithm 6 produces two partitions. The first partition $\mathcal{L}_1$ contains the lists:

$$L_A^1 = [(1, \{A\}), (2, \{A\})], \quad L_B^1 = [(1, B), (2, B), (3, B)], \quad L_D^1 = [(1, \{D\}), (3, \{D\})]$$

The second partition $\mathcal{L}_2$ contains the lists:

$$L_A^2 = [(4, \{A\})], \quad L_B^2 = [(4, B)], \quad L_C^2 = [(5, \{C\}), (6, \{C\}), (7, \{C\})]$$

Based on these partitions algorithm 5 will be provided with two repositories $\mathcal{L}_1$ and $\mathcal{L}_2$. It generates $\mathcal{AC}_1 = \{\{A, B, D\}, \{A, B\}, \{B, D\}\}$ from the first repository and $\mathcal{AC}_2 = \{\{A, B\}, \{C\}\}$ from the second repository. The whole *attribute clustering* is then $\mathcal{AC} = \mathcal{AC}_1 \cup \mathcal{AC}_2 = \{\{A, B\}, \{A, B, D\}, \{B, D\}, \{C\}\}$.

5 Experiments

The main aim of our experiments is to compare the performance of S-INDD with that of SPIDER. This is our focus because SPIDER is reported to be the current leading algorithm for unary INDs discovery [1,2]. SPIDER already significantly outperforms other approaches, in particular [3] and [11].

Experimental Conditions. We implemented both algorithms in Java 7 and performed the experiments on the Windows 7 Enterprise system with an Intel Core i5-3470 (Quad Core, 3.20 GHz CPU) and 8 GB RAM. We used an external 500 GB hard drive as external memory. We set the minimum Java heap size to 4 GB and the maximum to 6 GB for all our experiments.

Datasets. Two groups of synthetic datasets are generated for conducting two different groups of experiments. The purpose of the first group is to evaluate and compare the scalability of both algorithms by varying the number of attributes and fixing the number of rows, while in the second group of datasets the number of rows is varied and the number of attributes is fixed.

Experiments with real-word datasets are conducted using datasets from the life science domain (see below).

Scaling the Number of Attributes. In these experiments, we generate thirteen synthetic datasets with the same number of rows, namely 200,000 rows.

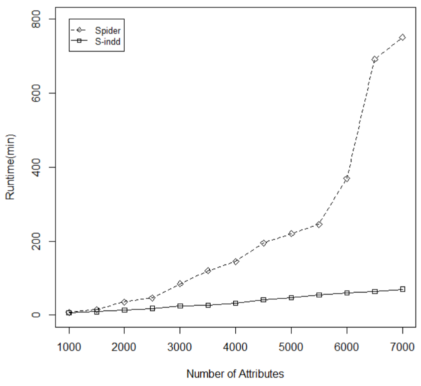

Fig. 2. Comparing scalability by scaling the number of attributes and fixing the number of rows to 200,000

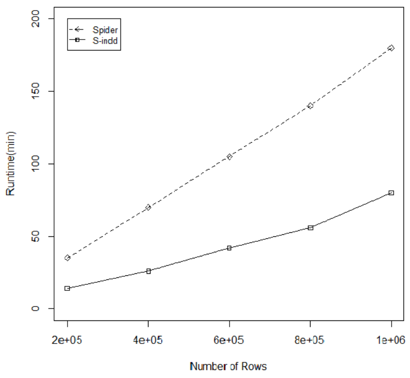

Fig. 3. Comparing scalability by scaling the number of rows and fixing the number of attributes to 2,000

Starting with 1,000 different attributes and ten unary INDs in the first dataset, the attributes set in the next dataset consists of the attributes set in the previous dataset plus 500 new different attributes and ten new different INDs so that the thirteenth dataset has 7,000 different attributes and 130 unary INDs. For all these datasets , S-INND is configured to merge 200 lists ($k = 200$) simultaneously.

Figure 2 shows the results of these experiments. (i) For every dataset S-INDD is faster than SPIDER. For example, for the dataset with 7,000 attributes and 36.2 GB size, S-INDD needs one hour and ten minutes while SPIDER needs twelve hours and thirty minutes. This means, S-INDD is about 11x faster than SPIDER. (ii) By increasing the number of attributes, SPIDER's runtime grows much faster than S-INDD's runtime. For example, by increasing the number of attributes from 6,000 to 7,000, SPIDER'runtime increases by 38 % while S-INDD'runtime increases by 1 % (for the dataset with 6,000 attributes and 31 GB SPIDER needs six hours and 10 minutes while S-INDD needs only about one hour).

Scaling the Number of Rows. In these experiments, we generate 5 synthetic datasets with the same number of attributes, namely 2,000 attributes. Starting with 200,000 rows in the first dataset, the next dataset contains all rows in the previous dataset plus 200,000 new different rows so that the fifth dataset has 1,000,000 rows and 48 GB size. Every dataset has the same number of INDs, namely 15 unary INDs. For all these datasets, we applied the extended version of S-INDD configured to merge 200 lists and to partition the datasets so that every partition had a maximum of 200,000 rows. For example, the dataset with 1,000,000 rows was divided into 5 disjoint partitions, i.e., algorithm 5 had to execute the *For*-loop 5 times (line 2).

Figure 3 shows the results of these experiments. These results also show that S-INDD is faster than SPIDER for every dataset. For example, for the dataset with 1,000,000 rows, S-INDD needs one hour and twenty-minutes while SPIDER needs about three hours. Furthermore, by increasing the number of rows, Spider's runtime grows faster than S-indd's runtime. For example, by increasing the number of rows from 200,000 to 1,000,000, Spider'runtime increases by 0.18 per thousand rows while S-indd'runtime increases by 0.08 per thousand rows (for the dataset with 200,000 rows and 10 GB Spider needs 35 minutes while S-indd needs only about 14 minutes).

INDs discovery in life science datasets. As real-word datasets we used SCOP[3], BIOSQL[4], CATH[5], and PDB[6] from the life science domain. To discover dependencies inside every dataset and between the datasets we processed the four datasets as a whole dataset. Their complete size is about 46 GB. Together they have total of 1,262 attributes. Life science databases are an example of the unreliability of the data type of the attributes. This means, we can not apply restriction on the data type of the attributes but rather, must assume that all attributes have the same data type (e.g. string). For this test, S-indd needed about 9 minutes while Spider needed about 17 minutes.

6 Related Work

Bell and Brockhausen [3] generate all unary IND candidates from previously collected statistics, such as min-max values and data types. Then they validate them using SQL join-statements. The transitivity of INDs is exploited to reduce the number of untested candidates. However, SQL-based validation is very costly because it is accesses the database for every candidate.

De Marchi et al. [11,13] propose an algorithm for unary INDs discovery that generates an inverted index associating every value to the attributes having the value. Because for every attribute A the intersection of all attribute sets containing A is the set of all attributes including A, the algorithm runs through all values in order to compute such an intersection for every attribute. However, this approach is inefficient because an attribute set in the index can be associated with many values. This means, the algorithm executes a lot of redundant intersection operations. The concept of *attribute clustering* we introduced in this paper solves this problem.

Bauckmann et al. propose Spider [1,2]. Spider is an external algorithm that writes the sorted values of every attribute to a file. Then it opens all files at once and starts comparing the values in parallel and in the same way in which the merge-sort algorithm does. Spider prunes IND candidates as follows: for each two attributes A and B, A is not included in B (i) if there is an iteration i in which the current A's value is greater than the current B's value and in the subsequent iteration $i+1$, B does not have value equal to the A's value in iteration i, or (ii) if there is an iteration in which the current A's value is less

[3] http://scop.mrc-lmb.cam.ac.uk/scop
[4] http://obda.open-bio.org
[5] http://www.biochem.ucl.ac.uk/bsm/cath_new
[6] http://www.rcsb.org/pdb

than the current B's value and in the subsequent iteration $i + 1$, A does not have value equal to the B's value in iteration i. This technique makes Spider the most efficient algorithm for unary IND detection in related work. However, Spider's scalability decreases by increasing the number of attributes. To solve this problem we developed S-indd in this paper.

Dasu et al. [5] compute a summary of data from which they calculate a "rate of similitude" between attributes. Based on this "rate of similitude" unary INDs can be found approximately. This means, some discovered unary INDs aren't satisfied, but also satisfied unary INDs can be missed.

Mannila and Toivonen [9] suggest the first known approach for an exhaustive search of N-ary INDs. They point out that this problem can fit in the framework of level-wise algorithms and is representable as sets; algorithms and implementations are proposed in [7,10–13]

Rostin et al. [14] propose rule-based discovery technique based on machine learning to derive foreign keys from INDs.

Zhang et al. [15] propose an approximate techniques to discover foreign keys. They assume that the value sets of foreign keys and the value sets of corresponding primary keys obey the same probability distribution. They premise availability of primary keys. Furthermore, Their approach may produce unsatisfied references and may miss satisfied references. For this reason, they focus on precision and recall rather than on runtime. The specialization on foreign key discovery also makes their approach inapplicable to other IND use cases, such as schema matching [8], query optimization [6], or integrity checking [4].

7 Conclusion

We introduced a new idea for clustering the attributes of database relations. We showed that the inferencing of all unary inclusion dependencies from the *attribute clustering* is much more efficient than inferencing them from the inverted index introduced in [11,13]. We then devised S-indd for computing the *attribute clustering* in large datasets. S-indd computes such clusters incrementally by extending the idea of sort-merge-join approach. S-indd is a composite of configurable computing iterations. In each iteration, it can control the number of rows and the number of attributes having to be processed. This flexibility makes each iteration efficiently executable. We showed how to parametrize S-indd to present Spider [1,2] as a special case of this algorithm. Therefore, S-indd is much more faster and scalable than Spider.

Acknowledgments. Discussions and collaboration with Felix Naumann and Thorsten Papenbrock supported this paper.

References

1. Bauckmann, J.: Dependency Discovery for Data Integration. Ph.D. thesis, Hasso Plattner Institute at the University of Potsdam (2013). http://opus.kobv.de/ubp/volltexte/2013/6664/

2. Bauckmann, J., Leser, U., Naumann, F.: Efficiently computing inclusion dependencies for schema discovery. In: Proceedings of the International Workshop on Database Interoperability (InterDB) (2006)
3. Bell, S., Brockhausen, P.: Discovery of Data Dependencies in Relational Databases. Tech. rep., Universitat Dortmund (1995)
4. Casanova, M.A., Tucherman, L., Furtado, A.L.: Enforcing inclusion dependencies and referencial integrity. In: Proceedings of the 14th International Conference on Very Large Data Bases, VLDB 1988, pp. 38–49. Morgan Kaufmann Publishers Inc., San Francisco (1988). http://dl.acm.org/citation.cfm?id=645915.671795
5. Dasu, T., Johnson, T., Muthukrishnan, S., Shkapenyuk, V.: Mining database structure; or, how to build a data quality browser. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 240–251 (2002). http://doi.acm.org/10.1145/564691.564719
6. Gryz, J.: Query folding with inclusion dependencies. In: In Proc. of the 14th IEEE Int. Conf. on Data Engineering (ICDE 1998), pp. 126–133 (1998)
7. Koeller, A., Rundensteiner, E.: Discovery of high-dimensional inclusion dependencies. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 683–685 (2003)
8. Levene, M., Vincent, M.W.: Justification for inclusion dependency normal form. IEEE Transactions on Knowledge and Data Engineering **12** (2000)
9. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledgediscovery. Data Min. Knowl. Discov. **1**(3), 241–258 (1997). http://dx.doi.org/10.1023/A:1009796218281
10. De Marchi, F., Flouvat, F., Petit, J.-M.: Adaptive strategies for mining the positive border of interesting patterns: application to inclusion dependencies in databases. In: Boulicaut, J.-F., De Raedt, L., Mannila, H. (eds.) Constraint-Based Mining and Inductive Databases. LNCS (LNAI), vol. 3848, pp. 81–101. Springer, Heidelberg (2006). http://www.dx.doi.org/10.1007/11615576_5
11. De Marchi, F., Lopes, S., Petit, J.-M.: Efficient algorithms for mining inclusion dependencies. In: Jensen, C.S., Jeffery, K., Pokorný, J., Šaltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) EDBT 2002. LNCS, vol. 2287, pp. 464–476. Springer, Heidelberg (2002). http://www.dl.acm.org/citation.cfm?id=645340.650245
12. Marchi, F.D., Petit, J.M.: Zigzag: a new algorithm for mining large inclusion dependencies in databases. In: Proceedings of the Third IEEE International Conference on Data Mining, ICDM, pp. 27–34 (2003). http://dl.acm.org/citation.cfm?id=951949.952179
13. Marchi, F., Lopes, S., Petit, J.M.: Unary and n-ary inclusion dependency discovery in relational databases. Journal of Intelligent Information Systems **32**(1), 53–73 (2009). http://dx.doi.org/10.1007/s10844-007-0048-x
14. Rostin, A., Albrecht, O., Bauckmann, J., Naumann, F., Leser, U.: A machine learning approach to foreign key discovery. In: Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB), Providence, RI (2009)
15. Zhang, M., Hadjieleftheriou, M., Ooi, B.C., Procopiuc, C.M., Srivastava, D.: On multi-column foreign key discovery. Proc. VLDB Endow. **3**(1–2), 805–814 (2010). http://dx.doi.org/10.14778/1920841.1920944