Detecting Maximum Inclusion Dependencies without Candidate Generation

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

Abstract. Inclusion dependencies (INDs) within and across databases are an important relationship for many applications in data integration, schema (re-)design, integrity checking, or query optimization. Existing techniques for detecting all INDs need to generate IND candidates and test their validity in the given data instance. However, the major disadvantage of this approach is the exponentially growing number of data accesses in terms of the number of *SQL* queries as well as I/O operations. We introduce Mind2, a new approach for detecting n-ary INDs ($n > 1$) without any candidate generation. Mind2 implements a new characterization of the maximum INDs we developed in this paper. This characterization is based on set operations defined on certain metadata that Mind2 generates by accessing the database only 2 × the number of valid unary INDs. Thus, Mind2 eliminates the exponential number of data accesses needed by existing approaches. Furthermore, the experiments show that Mind2 is significantly more scalable than hypergraph-based approaches.

Keywords: Mind2 · Inclusion dependency · Data integration · Data profiling

1 Introduction

Inclusion dependencies (INDs) present an important part of metadata about relationships between attributes in relational datasets [2]. An IND states that all tuples of some attribute-combination in one relation are contained in the tuples of some other attribute-combination in the same or a different relation. This makes INDs important for many tasks, such as data integration [17], integrity checking [3], query optimization [4], or schema (re-)design [10].

However, in many real-life databases knowledge about INDs is often unknown, or is lost, or does not correspond any more to the dataset structure. Furthermore, a lot of production databases are constantly changing over time so that metadata quickly become out-of-date. Thus, there is a high demand for effective and scalable approaches for mining valid INDs from a given dataset.

The problem of n-ary IND discovery ($n > 1$) is NP-hard [5]. Existing algorithms in related work for exhaustively discovering all INDs in a dataset can

S. Hartmann and H. Ma (Eds.): DEXA 2016, Part II, LNCS 9828, pp. 118–133, 2016.
DOI: 10.1007/978-3-319-44406-2_10

be divided into two approaches: levelwise-based approaches such as MIND [13] and hypergraph-based approaches as FIND$_2$ [6,7] and ZIGZAG [14]. But what all these algorithms have in common is that they apply the projection invariance of INDs [2,11]: a n-ary valid IND implies sets of k-ary valid INDs ($1 \leq k \leq n$). Thus, the number of all valid INDs implied by a n-ary valid IND is 2^n.

For discovering a single valid IND σ of size n, the levelwise approach [12] has to discover $2^n - 1$ implied INDs before even considering σ. This means for MIND that it has to execute 2^n *SQL* queries for validation. Experiments conducted by [6,14] show that levelwise algorithms do not scale beyond a maximum IND size of 8.

Attempting to reduce the exponential number of database accesses needed by the Apriori-based approach, FIND$_2$ and ZIGZAG transform the IND discovery problem into a discovery problem in a hypergraph whose nodes are all valid unary INDs, respectively. FIND$_2$ maps the IND discovery problem to the hyperclique discovery problem while ZIGZAG maps it to the minimal traversal discovery problem. Both problems are polynomial in the number of edges, and therefore exponential in the number of nodes in the hypergraph because the number of edges in a hypergraph of n nodes is bounded by 2^n. In principle, both algorithms first discover unary and binary INDs by enumeration and validation. Then optimistically assume that all high-arity INDs constructed from validated unary and binary INDs (or in general, from validated INDs in the previous iteration) are likely to be valid. That assumption makes both algorithms extremely sensitive to an overestimation of valid unary and binary INDs. A high number of such small INDs can cause many invalid larger IND candidates to be generated and validated against the database. Furthermore, hypergraph-based algorithms have high complexity, and are scalable only for sparse hypergraphs [6,8].

Research Question. The research question we address in this paper is how we can find all n-ary valid INDs ($n > 1$) between two relations without generating candidates and testing them against the database.

Contributions. We answer the research question by developing MIND$_2$ (short for **M**aximum **IN**clusion **D**ependency **D**iscovery), a novel approach for mining all maximum INDs without any candidate generation, where a maximum IND is a valid IND that is not implied by any other valid IND.

Having the set of all valid unary INDs, denoted by $\mathcal{I}_u$, discovered, MIND$_2$ computes the unary IND coordinates C_u for every valid unary IND $u \in \mathcal{I}_u$. Unary IND coordinates is a new concept we introduce in this paper (see Definition 4). To compute all unary IND coordinates MIND$_2$ executes only $2 \times |\mathcal{I}_u|$ simple *SQL* *select* queries with an *order by* clause. After computing all unary IND coordinates MIND$_2$ does not access the database any more because MIND$_2$ computes the set of all maximum INDs, denoted by $\mathcal{I}_M$, by only applying set operations on the unary IND coordinates (see Sect. 3). We compare the performance of MIND$_2$ with that of FIND$_2$ using real and synthetic datasets. They experiments show that MIND$_2$ is much more faster than FIND$_2$. Furthermore, they show that MIND$_2$'s

scalability, on contrast to FIND$_2$'s scalability, is not influenced by a high number of small valid INDs.

2 Preliminaries

Let $\mathcal{A}$ be a finite set of attributes. For $A_1, \ldots, A_n \in \mathcal{A}$ and for a symbol R, $R[A_1, \ldots, A_n]$ is called a relational schema over $A_1, \ldots, A_n$ and R is the relation name. An instance of R, identified by r, is a finite set of tuples over R. A tuple over R is an element from $dom(A_1) \times \cdots \times dom(A_n)$, where $dom(A_i)$ defines the set of the possible values of attribute A_i $(1 \leq i \leq n)$. The number of attributes in R is $|R|$ and the number of tuples in r is $|r|$. We refer to a tuple in r as r_i, where i $(1 \leq i \leq |r|)$ is the tuple-ID in r. ID_R indicates the set of all tuple-IDs in r. For an attribute sequence $X = [A_{i_1}, \ldots, A_{i_m}]$, we define $\pi_X(R)$ as the projection of R on X. Accordingly, $r_i[X] = \pi_X(r_i)$ indicates the projection of the tuple r_i on X. Furthermore, we identify the selection of a tuple r_i from r with $\sigma_{ID_R=i}(R)$. That is, $\{r_i\}$ is the result of $\sigma_{ID_R=i}(R)$. Accordingly, $\sigma_{ID_R<i}(R)$ identifies the set of all tuples in r with an ID less than i. Thus, $\sigma_{ID_R<i}(R) = \{r_k \in r \mid k < i\}$.

Definition 1. *(IND). Let $R[A_1, \ldots, A_{|R|}]$ and $S[B_1, \ldots, B_{|S|}]$ be two relational schemata. For $n \geq 1$, let X be a sequence of n attributes from R and Y a sequence of n attributes from S. An **inclusion dependency** (IND) over R and S is an assertion of the form $R[X] \subseteq S[Y]$ where n is the size of the IND. For $n = 1$ the IND is called a unary IND (uIND).*

*Let r and s be instances of R and S, respectively. An IND $R[X] \subseteq S[Y]$ is **valid** according to r and s if and only if $\forall r_i \in r, \exists s_j \in s$ such that $r_i[X] = s_j[Y]$.*

In particular, INDs are a prerequisite for foreign keys, which are a necessity for suggesting join paths, data linkage, and data normalization.

3 Principles of Mind$_2$

We consider two relational schemata $R[A_1, \ldots, A_{|R|}]$ and $S[B_1, \ldots, B_{|S|}]$ with corresponding instances r and s. To formulate the basic ideas of detecting all maximum INDs between R and S, we identify the set of all unary INDs with Σ_u and the set of all INDs with Σ. Furthermore, we introduce the following sets.

The set of all valid unary INDs between R and S according to r and s

$$\mathcal{I}_u = \{u \in \Sigma_u \mid u \text{ is valid according to } r \text{ and } s\}$$

The set of all valid INDs between R and S according to r and s

$$\mathcal{I} = \{I \in \Sigma \mid I \text{ is valid according to } r \text{ and } s\}$$

We represent every IND $\sigma = R[X] \subseteq S[Y]$ with $X = [A_{i_1}, \ldots, A_{i_n}]$ and $Y = [B_{i_1}, \ldots, B_{i_n}]$ as a set of all unary INDs implied by it. In other words, we present σ as the set $\{A_{i_1} \subseteq B_{i_1}, \ldots, A_{i_n} \subseteq B_{i_n}\}$. Furthermore, we identify the

set of all attributes occurring on the left hand side of σ with $LHS(\sigma)$ and the set of all attributes occurring on the right hand side of σ with $RHS(\sigma)$. Thus, we have $LHS(\sigma) = \{A_{i_1}, \ldots, A_{i_n}\}$ and $RHS(\sigma) = \{B_{i_1}, \ldots, B_{i_n}\}$. Representing an IND as a set allows us to characterize the computation of the set of all maximum INDs $\mathcal{I}_M$ as set operations.

Based on the set presentation, we introduce the concept of a maximum IND.

Definition 2 *(Maximum IND). Let $I \in \mathcal{I}$ be a valid IND between R and S. I is a maximum IND if and only if there is no $I' \in \mathcal{I}$ such that $I \subset I'$ holds. We denote the set of all maximum INDs between R and S with $\mathcal{I}_M$.*

Having $\mathcal{I}_M$ discovered, we can derive the set of all valid INDs $\mathcal{I}$ as

$$\mathcal{I} = \{I \mid \exists M \in \mathcal{I}_M : I \subseteq M\}$$

The set $\mathcal{I}_M$ can be considered as a concise representation of the set $\mathcal{I}$. Thus, our goal in this work is to directly compute $\mathcal{I}_M$ without any intermediate IND sets.

Table 1. Running Example

R

ID_R	A_1	A_2	A_3	A_4	A_5
1	a	b	c	d	e
2	f	g	i	j	k

S

ID_S	B_1	B_2	B_3	B_4	B_5
1	a	b	c	d	$\perp$
2	$\perp$	$\perp$	c	d	$\perp$
3	$\perp$	$\perp$	c	d	e
4	f	g	i	$\perp$	$\perp$
5	f	g	$\perp$	j	k

Example 1. According to the two relations presented in Table 1, we have

$\mathcal{I}_u = \{u_i = A_i \subseteq B_i \mid 1 \leq i \leq 5\}$
$\mathcal{I} = \{\{u_1, u_2\}, \{u_1, u_3\}, \{u_2, u_3\}, \{u_1, u_2, u_3\}, \{u_1, u_4\}, \{u_2, u_4\}, \{u_1, u_2, u_4\}, \{u_4, u_5\}\} \cup \mathcal{I}_u$
$\mathcal{I}_M = \{\{u_1, u_2, u_3\}, \{u_1, u_2, u_4\}, \{u_4, u_5\}\}$
E.g. $\sigma = \{u_1, u_5\} \notin \mathcal{I}$ (i.e. not valid) because $r_1[LHS(\sigma)] = r_1[\{A_1, A_5\}] = \{(a, e)\} \not\subseteq \pi_{RHS(\sigma)}(S)$.

The first principle of computing $\mathcal{I}_M$ is formulated as follows.

Principle 1. For every tuple pair $r_i \in r$ and $s_i \in s$, we compute M^{ij}, the maximum IND between $\sigma_{ID_R=i}(R)$ and $\sigma_{ID_S=j}(S)$ according to r_i and s_j ($1 \leq i \leq |r|$ and $1 \leq j \leq |s|$). To characterize the set M^{ij} we introduce two new concepts: attribute value-positions and unary valid IND coordinates.

Definition 3 *(Attribute Value-Positions). The value positions of an attribute $A \in U$, $U \in \{R, S\}$, is the set $P_A = \pi_{\{ID_U, A\}}(U)$*

Definition 4 *(Unary IND Coordinates). The coordinates of a valid unary IND* $u \in \mathcal{I}_u$ *is the set* $C_u = \{(i,j) \mid \exists (i,v) \in P_{LHS(u)}$ *and* $\exists (j,v') \in P_{RHS(u)} : v = v'\}$

The coordinates of a valid unary IND $u \in \mathcal{I}_u$ is the set of all tuple-ID pairs (i,j) where the value of attribute $LHS(u)$ in the tuple $r_i \in r$ is identical with the value of attribute $RHS(u)$ in the tuple $s_i \in s$. In other words, $(i,j) \in C_u$ if and only if $r_i[LHS(u)] = s_j[RHS(u)]$.

Having the coordinates of all unary INDs generated, we can compute the maximum IND M^{ij} between any tuple pair (r_i, s_j) without any database access based on the following lemma.

Lemma 1. M^{ij} *consists of all unary INDs* $u \in \mathcal{I}_u$ *with* $(i,j) \in C_u$*. In other words,* $M^{ij} = \{u \in \mathcal{I}_u \mid (i,j) \in C_u\}$.

Proof. Let $M^{ij} = \{u_1, \ldots, u_n\}$ be the set of all valid uINDs with $(i,j) \in C_{u_k}$ where $1 \leq k \leq n$. Based on Definition 4, there is $(i, v_k) \in P_{LHS(u_k)}$ and $(j, v'_k) \in P_{RHS(u_k)}$ with $v_k = v'_k$ for every $k \in \{1, \ldots, n\}$. This means that $(v_1, \ldots, v_n) = (v'_1, \ldots, v'_n)$. In other words, $r_i[LHS(M^{ij})] = s_j[RHS(M^{ij})]$. Based on Definition 1, M^{ij} is a valid IND between $\sigma_{ID_R=i}(R)$ and $\sigma_{ID_S=j}(S)$ according to r_i and s_j.

We now have to show that M^{ij} is maximum. We assume that M^{ij} is not maximum. This means based on Definition 2 that there is a valid IND M_1^{ij} with $M^{ij} \subset M_1^{ij}$. Thus, M_1^{ij} contains some $u' \in \mathcal{I}_u$ with $(i,j) \notin C_{u'}$. This means that the value of attribute $LHS(u')$ in r_i is different from the value of attribute $RHS(u')$ in s_j. Thus, $r_i[LHS(M_1^{ij})] \neq s_j[RHS(M_1^{ij})]$ which means that M_1^{ij} is not valid. Thus, our assumption is wrong. □

Table 2. The coordinates of all valid uINDs between R and S in Table 1

i	P_{A_i}	P_{B_i}	$C_{A_i \subseteq B_i}$
1	$\{(1,a),(2,f)\}$	$\{(1,a),(2,\perp),(3,\perp),(4,f),(5,f)\}$	$\{(1,1),(2,4),(2,5)\}$
2	$\{(1,b),(2,g)\}$	$\{(1,b),(2,\perp),(3,\perp),(4,g),(5,g)\}$	$\{(1,1),(2,4),(2,5)\}$
3	$\{(1,c),(2,i)\}$	$\{(1,c),(2,c),(3,c),(4,i),(5,\perp)\}$	$\{(1,1),(1,2),(1,3),(2,4)\}$
4	$\{(1,d),(2,j)\}$	$\{(1,d),(2,d),(3,d),(4,\perp),(5,j)\}$	$\{(1,1),(1,2),(1,3),(2,5)\}$
5	$\{(1,e),(2,k)\}$	$\{(1,\perp),(2,\perp),(3,e),(4,\perp),(5,k)\}$	$\{(1,3),(2,5)\}$

Example 2. Based on our running example, the second column in Table 2 lists the value positions P_{A_i} of R's attributes while the value positions P_{B_i} of S's attributes are listed in the third column. The last column in this table shows the coordinates of all valid unary INDs between R and S (see Example 1). E.g. for $A_5 \subseteq B_5$, we have $(1,e) \in P_{A_5}$ and $(3,e) \in P_{B_5}$. Therefore, $C_{A_5 \subseteq B_5}$ contains the pair $(1,3)$. Also, $(2,5) \in C_{A_5 \subseteq B_5}$ because $(2,k) \in P_{A_5}$ and $(5,k) \in P_{B_5}$.

The maximum INDs M^{ij} between r_i and s_j ($1 \leq i \leq 2$ and $1 \leq j \leq 5$) are

$$M^{1,1} = \{u_1, u_2, u_3, u_4\}, \quad M^{1,2} = \{u_3, u_4\}, \quad M^{1,3} = \{u_3, u_4, u_5\}, \quad M^{1,4} = M^{1,5} = \emptyset$$
$$M^{2,1} = M^{2,2} = M^{2,3} = \emptyset, \quad M^{2,4} = \{u_1, u_2, u_3\}, \quad M^{2,5} = \{u_1, u_2, u_4, u_5\}$$

E.g. let us explain the content of the maximum IND $M^{1,2}$ between r_1 and s_2. We have $(1,2) \in C_{u_3}$. Therefore, $u_3 \in M^{1,2}$. Also, $u_4 \in M^{1,2}$ because $(1,2) \in C_{u_4}$. But $u_1, u_2, u_5 \notin M^{1,2}$ because $(1,2) \notin C_{u_1}$, $(1,2) \notin C_{u_2}$, and $(1,2) \notin C_{u_5}$.

In the next step, we compute the set of all maximum INDs between every tuple $r_i \in r$ and the relation s based on the following principle, respectively.

Principle 2. For every tuple $r_i \in r$, we compute $\mathcal{I}^i_M$, the set of all maximum INDs between $\sigma_{ID_R=i}(R)$ and S according to r_i and s. To characterize the set $\mathcal{I}^i_M$, we introduce the following operator.

Definition 5 *(ϕ-operator).* $\phi : 2^\Sigma \to 2^\Sigma$, $\phi(\mathcal{S}) = \{\sigma \mid \nexists \sigma' \in \mathcal{S} : \sigma \subset \sigma'\}$

Operator ϕ takes a set of INDs and returns each IND that is not included in any other IND in this set. Thus, we conclude: $\phi(\mathcal{S}) \subseteq \mathcal{S}$ for any $\mathcal{S} \in 2^\Sigma$.

Lemma 2. *$\mathcal{I}^i_M = \phi(\mathcal{I}^i)$, where $\mathcal{I}^i$ is the set of all non-empty M^{ij} ($1 \leq j \leq |s|$).*

Proof. Every $M^{ij} \in \mathcal{I}^i$ is a valid (but not necessary a maximum) IND between $\sigma_{ID_R=i}(R)$ and S. But what we want to have is all maximum INDs from $\mathcal{I}^i$. Based on Definition 5, ϕ-operator solves this task. Thus, $\mathcal{I}^i_M = \phi(\mathcal{I}^i)$ is the set of all maximum INDs between $\sigma_{ID_R=i}(R)$ and S. □

Example 3. Based on Example 2, we have

$\mathcal{I}^1 = \{M^{1,1}, M^{1,2}, M^{1,3}\}$, $\mathcal{I}^1_M = \phi(\mathcal{I}^1) = \{M^{1,1}, M^{1,3}\}$
$\mathcal{I}^2 = \{M^{2,4}, M^{2,5}\}$, $\mathcal{I}^2_M = \phi(\mathcal{I}^2) = \{M^{2,4}, M^{2,5}\}$

We can now compute $\mathcal{I}_M$, the set of all maximum INDs between R and S, from the sets $\mathcal{I}^i_M$ ($1 \leq i \leq |r|$) based on Principle 3.

Principle 3. To explain the main idea behind Principle 3, let us consider the two relations in Table 1. What are the maximum INDs between them if we know $\mathcal{I}^1_M$ and $\mathcal{I}^2_M$ computed in Example 3? First, the intersection between any two INDs $M_1 \in \mathcal{I}^1_M$ and $M_2 \in \mathcal{I}^2_M$ is a valid IND between R and S. E.g., $M^{1,1} \cap M^{2,4} = \{u_1, u_2, u_3\}$ is a valid IND between R and S. Second, after computing the intersection between each pair $(M_1, M_2) \in \mathcal{I}^1_M \times \mathcal{I}^2_M$, taking all maximum sets from the result gives us the set of all maximum INDs (see Example 4). We generalize these two ideas as follows.

Definition 6 *(ψ-operator).* $\psi : 2^\Sigma \times 2^\Sigma \to 2^\Sigma$, $\psi(\mathcal{S}_1, \mathcal{S}_2) = \{\sigma \mid \exists (\sigma_1, \sigma_2) \in \mathcal{S}_1 \times \mathcal{S}_2 : \sigma = \sigma_1 \cap \sigma_2$ *and* $\sigma \neq \emptyset\}$

In words, for two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ of INDs the ψ-operator takes every tuple (σ_1, σ_2) from $\mathcal{S}_1 \times \mathcal{S}_2$ and computes the intersection between σ_1 and σ_2. To characterize the computation of the set $\mathcal{I}_M$, we define the ρ-operator.

Definition 7 *(ρ-operator). Let $\mathcal{I}_\mathcal{M}$ be the set of all $\mathcal{I}_M^i$ $(1 \leq i \leq |r|)$.*

$$\rho(\mathcal{I}_\mathcal{M}) = \begin{cases} \mathcal{S} & \text{if } |\mathcal{I}_\mathcal{M}| = 1 \text{ and } \mathcal{S} \in \mathcal{I}_\mathcal{M} \\ \phi(\psi(\mathcal{S}, \rho(\mathcal{I}_\mathcal{M} \setminus \{\mathcal{S}\}))) & \text{if } |\mathcal{I}_\mathcal{M}| > 1 \text{ and } \mathcal{S} \in \mathcal{I}_\mathcal{M} \end{cases}$$

Now, we can compute $\mathcal{I}_M$ as follows.

Lemma 3. $\mathcal{I}_M = \rho(\mathcal{I}_\mathcal{M})$

Proof. We prove the lemma by induction on the number of tuples i in r.

Basis Step: For $i = 1$, we have $\mathcal{I}_\mathcal{M} = \{\mathcal{I}_M^1\}$. Thus, $\rho(\{\mathcal{I}_M^1\}) = \mathcal{I}_M^1 = \mathcal{I}_M$ based on the construction of the set $\mathcal{I}_M^1$.

Induction Assumption: For $1 \leq i < |r|$, let $\mathcal{I}'_\mathcal{M}$ be the set of all $\mathcal{I}_M^i$ and $\mathcal{I}'_M$ be the set of all maximum INDs between $\sigma_{ID_R<|r|}(R)$ and S. We assume

$$\mathcal{I}'_M = \rho(\mathcal{I}'_\mathcal{M}) \tag{1}$$

Inductive Step: Let $\mathcal{I}_M^{|r|}$ be the set of all maximum INDs between $\sigma_{ID_R=|r|}(R)$ and S. Thus, $\mathcal{I}_\mathcal{M} = \mathcal{I}'_\mathcal{M} \cup \mathcal{I}_M^{|r|}$. Based on assumption (1), we have to show

$$\mathcal{I}_M = \rho(\mathcal{I}_\mathcal{M}) = \phi(\psi(\mathcal{I}_M^{|r|}, \rho(\mathcal{I}'_\mathcal{M}))) = \phi(\psi(\mathcal{I}_M^{|r|}, \mathcal{I}'_M))$$

Every set in $\psi(\mathcal{I}_M^{|r|}, \mathcal{I}'_M)$ is a valid IND between R and S because the intersection of two valid INDs is a valid IND. We assume that there is a valid IND I with

$$I \notin \psi(\mathcal{I}_M^{|r|}, \mathcal{I}'_M) \tag{2}$$

Because I is a valid IND, there is $I_1 \in \mathcal{I}_M^{|r|}$ with $I \subseteq I_1$ and $I_2 \in \mathcal{I}'_M$ with $I \subseteq I_2$. Thus, $I \subseteq I_1 \cap I_2$, but $I_1 \cap I_2 \in \psi(\mathcal{I}_M^{|r|}, \mathcal{I}'_M)$. This means that assumption (2) is wrong. Consequently, $\psi(\mathcal{I}_M^{|r|}, \mathcal{I}'_M)$ contains all valid INDs between S and R. Based on Definition 5, $\phi(\psi(\mathcal{I}_M^{|r|}, \mathcal{I}'_M))$ is the set of all maximum INDs in $\psi(\mathcal{I}_M^{|r|}, \mathcal{I}'_M)$. Thus, $\mathcal{I}_M = \phi(\psi(\mathcal{I}_M^{|r|}, \mathcal{I}'_M))$. □

Example 4. Based on Example 3, we have $\mathcal{I}_\mathcal{M} = \{\mathcal{I}_M^1, \mathcal{I}_M^2\}$. Accordingly,
$\psi(\mathcal{I}_\mathcal{M}) = \{M^{1,1} \cap M^{2,4}, M^{1,1} \cap M^{2,5}, M^{1,3} \cap M^{2,4}, M^{1,3} \cap M^{2,5}\}$
$\psi(\mathcal{I}_\mathcal{M}) = \{\{u_1, u_2, u_3\}, \{u_1, u_2, u_4\}, \{u_3\}, \{u_4, u_5\}\}$
$\mathcal{I}_M = \rho(\mathcal{I}_\mathcal{M}) = \phi(\psi(\mathcal{I}_\mathcal{M})) = \{\{u_1, u_2, u_3\}, \{u_1, u_2, u_4\}, \{u_4, u_5\}\}$
(compare with Example 1).

In the following section, we formulate MIND$_2$ algorithmically. We also present its data structures. This formulation is the basis of our implementation of MIND$_2$.

4 Mind$_2$

Overall Mind$_2$. MIND$_2$ consists of three major components. Algorithm 1 as the first component, is responsible for computing the unary IND coordinates C_u of each valid unary IND $u \in \mathcal{I}_u$ based on Definition 4. It also stores each generated set C_u in a separate file in an external repository *Repo* on a hard drive.

Then Algorithm 2 reads the generated coordinates at once and computes the set of all maximum INDs $\mathcal{I}_M$ incrementally according to the ascending order of the tuple-IDs $i \in ID_R$ in the left relation r. In other words, it computes the ρ-operator (see Definition 7) iteratively. Before the iteration in which the set $\mathcal{I}_M^i$ (the set of all maximum INDs between $\sigma_{ID_R=i}(R)$ and S) can be generated, Algorithm 2 computes all maximum INDs between $\sigma_{ID_R<i}(R)$ and S and stores them in $\mathcal{I}_M$. In other words, before the computation of $\mathcal{I}_M^i$ starts, the set $\mathcal{I}_M$ contains the maximum INDs between the tuples $\{r_k \in r \mid 1 \leq k < i\}$ and s. Having $\mathcal{I}_M^i$ generated, Algorithm 2 replaces the current content of the set $\mathcal{I}_M$ with the result of the composite operation $\phi(\psi(\mathcal{I}_M, \mathcal{I}_M^i))$. This procedure continues until all tuple-IDs $i \in ID_R$ have been processed. At the end and based on Lemma 3, the set $\mathcal{I}_M$ contains all maximum INDs between R and S. At the beginning, we initialize $\mathcal{I}_M$ with $\{\mathcal{I}_u\}$ because $\{\{\mathcal{I}_u\}\}$ is an upper bound of $\mathcal{I}_M$.

The third component of MIND$_2$ is Algorithm 3 called by Algorithm 2 to compute the sets $\mathcal{I}_M^i$ $(1 \leq i \leq |r|)$. It computes them based on Lemmas 1 and 2. Below, we explain these components in details.

```
   Input  : I_u, Repo
   Output : C_u for every u ∈ I_u
 1 foreach u ∈ I_u do
 2     i2jsMap ← createMap(Int, Set)
 3     A ← LHS(u); B ← RHS(u)
 4     Cur_A ← createCursor(A)
 5     Cur_B ← createCursor(B)
 6     (i, v) ← Cur_A.next()
 7     (j, v') ← Cur_B.next()
 8     while Cur_A.hasNext() and Cur_B.hasNext()
       do
 9         if v = v' then
10             ID_A ← {}; ID_B ← {}
11             (k, w) ← Cur_A.current()
12             while v = w do
13                 ID_A = ID_A ∪ {k};
                   (k, w) ← Cur_A.next()
14             (k, w) ← Cur_B.current()
15             while v = w do
16                 ID_B = ID_B ∪ {k};
                   (k, w) ← Cur_B.next()
17             if ID_A ≠ ∅ and ID_B ≠ ∅ then
18                 foreach i ∈ ID_A do
19                     i2jsMap.put(i, ID_B)
20         else if v > v' then
21             (j, v') ← Cur_B.next()
22         else
23             (i, v) ← Cur_A.next()
24     writer ← createWriter(u, Repo)
25     ID_A ← i2jsMap.keys(); sort(ID_A)
26     foreach i ∈ ID_A do
27         P_B ← i2jsMap.get(i); sort(ID_B)
28         writer.write(u, i, ID_B)
```

Algorithm 1. genCoordinates

$A_1 \subseteq B_1$	1, [1] 2, [4, 5]
$A_2 \subseteq B_2$	1, [1] 2, [4, 5]
$A_3 \subseteq B_3$	1, [1, 2, 3] 2, [4]
$A_4 \subseteq B_4$	1, [1, 2, 3] 2, [5]
$A_5 \subseteq B_5$	1, [3] 2, [5]

Fig. 1. The output of Algorithm 1 for the set of all valid unary INDs between R and S in the running example

Generating unary IND Coordinates. To compute the unary IND coordinates of a $u \in \mathcal{I}_u$, Algorithm 1 opens two cursors at once (lines 3–5): one for reading the sorted value positions of the attribute $A = LHS(u)$ and the other for reading the sorted value positions of the attribute $B = RHS(u)$ (see Definition 3 for the value positions of an attribute). The value positions of every attribute are sorted according to its values in the corresponding relation. In other words, for any $(i_1, v_1), (i_2, v_2) \in P_A$ $(\in P_B)$: the tuple (i_1, v_1) will be read by the corresponding cursor before the tuple (i_2, v_2) if the value v_1 occurs before the value v_2 in the sort sequence. Otherwise, (i_2, v_2) will be read before (i_1, v_1).

In the main *while*-loop (lines 8–23), Algorithm 1 moves the two cursors in such a way so that it can associate every tuple-ID $i \in ID_R$ with the set of all tuple-IDs $j \in ID_S$ for which both attributes A and B have the same value. In other words, the tuple-ID i is associated with the set $\{j \mid \exists (j, v) \in P_B : (i, v) \in P_A\}$. It saves this association temporary in the hash map *i2jsMap* (lines 17–19).

After finishing the reading of value positions of P_A and P_B, respectively, Algorithm 1 creates a file for the current unary IND u in the *for*-loop (lines 1–28) and saves every pair $(i, \{j \mid \exists (j, v) \in P_B : (i, v) \in P_A\})$ in a line in this file. The lines (records) are sorted in ascending order by the left tuple-IDs $i \in ID_R$ and in every line the IDs $j \in \{j \mid \exists (j, v) \in P_B : (i, v) \in P_A\}$ are also sorted in ascending order (lines 24–28). This policy of organizing the value positions is required by Algorithm 2.

MIND2 needs only $2 \times |\mathcal{I}_u|$ database accesses because every cursor needs a simple *SQL select* statement with an *order by* clause for reading the value position of an attribute.

Example 5. Based on the attribute value positions listed in Table 2, Fig. 1 illustrates the output of the Algorithm refalgo:coordinatesGen. Every row in this figure represents a file containing the coordinates of an unary IND.

Generating Maximum INDs between R and S. Algorithm 2, as implementation of Principle 3 (see Sect. 3), generates the set of all maximum INDs $\mathcal{I}_M$ by computing the ρ-operator (see Definition 7) incrementally. It opens all files of the unary INDs coordinates generated by Algorithm 1 and reads them at once (lines 3–4). Every $u \in \mathcal{I}_u$ is associated with a sequential file reader for reading its coordinates C_u. The file readers are managed by a priority queue. For any two readers fr, fr', reader fr has a higher priority than fr' in the queue if and only if the tuple-ID i in the file entry (u, i, L) is smaller than the tuple-ID i' in (u', i', L') where (u, i, L) is the entry that fr can currently read and (u', i', L') is the entry that fr' can currently read. Managing the readers in this way allows Algorithm 2 to collect all unary INDs $u \in \mathcal{I}_u$ that have the same tuple-ID i $(i \in ID_R)$ in their coordinates (lines 7–18).

In every pass through the main *while*-loop (lines 6–29) the algorithm collects the elements (u, L) in the set $\mathcal{L}$ where all unary INDs u in these elements have the same tuple-ID $i \in ID_R$. Every list L in (u, L) is (based on its construction by Algorithm 1) the list of all tuple-IDs $j \in ID_S$, where the values of attribute $RHS(u)$ in these tuples and the value of $LHS(u)$ in tuple i are identical.

```
   Input  : I_u, Repo
   Output : I_M
1  Queue ← createPriorityQueue()
2  foreach u ∈ I_u do
3      fr ← createFileReader(u, Repo)
4      Queue.add(fr)
5  I_M ← {I_u}
6  while Queue ≠ ∅ do
7      L ← ∅; Readers ← ∅
8      fr ← Queue.pull()
9      Readers ← Readers ∪ {fr}
       (u, i, L) ← fr.current()
10     L ← L ∪ {(u, L)}
11     while Queue ≠ ∅ do
12         fr' ← Queue.peek()
13         (u', i', L') ← fr'.current()
14         if i ≠ i' then break
15         fr ← Queue.pull()
16         Readers ← Readers ∪ {fr}
17         (u, i, L) ← fr.current()
18         L ← L ∪ {(u, L)}
19     I*_M ← genSubMaxINDs(L, I_M)
20     I_M ← φ(ψ(I_M, I*_M))
21     foreach u ∈ I_u : {u} ∈ I_M do
22         I_M ← I_M \ {{u}}
23     if I_M = ∅ then
24         I_M ← I_u; break
25     activeU ← ∪_{M∈I_M} M
26     foreach fr ∈ Readers do
27         if fr.hasNext() and
28           fr.u ∈ activeU then
29             fr.next();
               Queue.add(fr)
```

Algorithm 2. genMaxINDs

```
   Input  : L, I*^{-1}_M
   Output : I*_M
1  Queue ← createPriorityQueue()
2  foreach (u, L) ∈ L do
3      lr ← createListReader(u, L)
4      Queue.add(lr)
5  UB ← ∅
6  while Queue ≠ ∅ do
7      Readers ← ∅
8      lr ← Queue.pull()
9      Readers ← Readers ∪ {lr}
       (u, j) ← lr.current()
10     M*^j ← {u}
11     while Queue ≠ ∅ do
12         lr' ← Queue.peek()
13         (u', j') ← lr'.current()
14         if j ≠ j' then break
15         lr ← Queue.pull()
16         Readers ← Readers ∪ {lr}
           (u, j) ← lr.current()
17         M*^j ← M*^j ∪ {u}
18     if ∃M ∈ I*^{-1}_M : M ⊆ M*^j then
19         UB ← UB ∪ {M}
20     if UB = I*^{-1}_M then
21         I*_M ← I*^{-1}_M; break
22     I*_M ← I*_M ∪ {M*^j}
23     foreach lr ∈ Readers do
24         if lr.hasNext() then
25             lr.next(); Queue.add(lr)
26 I*_M ← φ(I*_M)
```

Algorithm 3. genSubMaxINDs

After creating the set $\mathcal{L}$ in the current pass of the main *while*-loop for a certain i, Algorithm 2 calls Algorithm 3 to compute the maximum INDs between $\sigma_{ID_R=i}(R)$ and S (line 19). We donate this set with $\mathcal{I}^*_M$ where the symbol $*$ is a placeholder for any $i \in ID_R$.

After computing maximum INDs $\mathcal{I}^*_M$ between $\sigma_{ID_R=i}(R)$ and S, the set of all maximum INDs $\mathcal{I}_M$ will be updated by applying the composite operation $\phi(\psi(\mathcal{I}_M, \mathcal{I}^*_M))$ in line 20 (see Definition 5 for ϕ-operator and Definition 6 for ψ-operator). The set $\mathcal{I}_M$ is initialized with the set $\{\mathcal{I}_u\}$ (line 5). If the updated set $\mathcal{I}_M$ contains only the unary INDs, the algorithm breaks the main *while*-loop and returns the set of all unary INDs as the maximum INDs (line 23–24). Otherwise, Algorithm 2 will update the queue only with readers of those unary INDs u which are contained at least in one set of $\mathcal{I}_M$ (lines 25–29).

Generating maximum INDs between $\sigma_{ID_R=i}(R)$ and S. Based on Principle 1 and Principle 2, Algorithm 3 computes the set of all maximum INDs between $\sigma_{ID_R=i}(R)$ and S from the set $\mathcal{L}$ while it exploits the set $\mathcal{I}^{*-1}_M$ to improve the performance. The set $\mathcal{L}$ generated by Algorithm 2 (lines 7–18) contains the elements (u, L): all unary INDs in these elements have the same left tuple-ID i in

their coordinates while every list L in (u, L) is the sorted list of all tuple-IDs $j \in ID_S$ in the coordinates $(i, j) \in C_u$. The algorithm reads all the lists in the set $\mathcal{L}$ at once and uses a priority queue to manage the list readers in the same way in which Algorithm 2 manages the file readers of the unary INDs coordinates.

In the main *while*-loop we collect all unary INDs u in the set M^{*j} that have the same tuple-ID j in their coordinates (lines 7–17). The symbol $*$ in M^{*j} is a placeholder for the corresponding i. Thus, based on the properties of the elements (u, L) of the set $\mathcal{L}$, the set M^{*j} contains all unary INDs u that have (i, j) in their coordinates C_u. This means, according to Lemma 1, M^{*j} is the maximum IND between $\sigma_{ID_R=i}(R)$ and $\sigma_{ID_S=j}(S)$.

Every computed set M^{*j} is collected in the set $\mathcal{I}^*_M$ (line 22). This means, updating $\mathcal{I}^*_M$ by applying the ϕ-operator on it gives us, according to Lemma 2, the maximum INDs between $\sigma_{ID_R=i}(R)$ and S (line 26).

The objective of the input set $\mathcal{I}^{*-1}_M$ is to improve the performance of computing $\mathcal{I}^*_M$. The set $\mathcal{I}^{*-1}_M$ is the set of all maximum INDs between $\sigma_{ID_R<i}(R)$ and S. For every generated set M^{*j} Algorithm 3 checks if there is a set M in $\mathcal{I}^{*-1}_M$ such that M is a subset of M^{*j} (lines 18–19). If such a set exists, it is added to the set UB. If the set UB contains all sets from $\mathcal{I}^{*-1}_M$, then the algorithm breaks the execution and returns $\mathcal{I}^{*-1}_M$ as the maximum INDs between $\sigma_{ID_R=i}(R)$ and S (lines 20–21). This rule does not have any affect on the correctness of Algorithm 2. This is because the result of the composite operation $\phi(\psi(\mathcal{I}_M, \mathcal{I}^*_M))$ in Algorithm 2 is the set $\mathcal{I}_M$ itself if every set in $\mathcal{I}^*_M$ is a superset of a set in $\mathcal{I}_M$.

5 Experiments

The main aim of our experiments is to compare the scalability of MIND$_2$ with that of FIND$_2$. This is our focus because FIND$_2$ is developed to reduce the number of IND candidates required by Apriori-based approaches. Although ZIGZAG is also designed to handle long INDs, we limited our experiments to FIND$_2$. That is because, as discussed in Sects. 1 and 6, FIND$_2$ and ZIGZAG approach the IND discovery problem from similar directions and have many properties in common.

The number of rows varies between 500,000 and 16,000,000 rows in these experiments. The other important variable that has a big impact on the scalability of discovering the n-ray INDs between two relations is the number of the unary INDs. The number of unary INDs in the experiments varies between 8 and 19 unary INDs in the corresponding table pairs.

Experimental Conditions. We performed the experiments on Windows 7 Enterprise system with an Intel Core i5-3470 (Quad Core, 3.20 GHz CPU) and 8 GB RAM. We used Oracle 11g as the database server installed on the same machine. We implemented both algorithms in 64-bit Java 7. We implemented FIND$_2$ based on [6]. For MIND$_2$, we set the minimum Java heap size to 4 GB and the maximum to 6 GB. While for FIND$_2$, we set the Java stack size to 4 GB. FIND$_2$ validates IND candidates by applying the *SQL* query proposed in [15].

Experiment Groups 1. The purpose of these experiments is to compare the scalability of MIND$_2$ with that of FIND$_2$ by using a real-word dataset called

MusicBrainz[1]. MusicBrainz is an open music encyclopedia that collects music metadata and makes them available to the public. MusicBrainz contains 27 GB of data. It contains 206 tables (relations) with 1,165 non-empty columns (attributes). We found a total of 24,881 valid unary INDs by applying S-indd [18]. We detected pairs of tables where there is at least one valid n-ary IND with size greater than 2 between the tables of every pair. The number of tuples varies between 500,000 and 1,000,500 tuples. The results of these experiments are presented in Table 3. The acronym "TP." stands for table pair. The left part of Table 3 shows some statistics about detected INDs: the number of valid unary INDs ($|\mathcal{I}_u|$), the number of detected maximum INDs ($|\mathcal{I}_M|$), the size of the longest maximum INDs (n_{max}) accompanied by their number ((x Nr.)), and the size of shortest maximum INDs (n_{min}) accompanied by their number ((x Nr.)). The right part of Table 3 shows the needed time (in minutes) by Mind$_2$ and Find$_2$ for detecting the valid INDs, respectively. The acronym "o.o.M." refers to out of memory exception. In most of these experiments, Mind$_2$ outperforms Find$_2$ significantly. Furthermore, they show that Mind$_2$'s scalability, on the contrary to that of Find$_2$, is robust and not sensitive to the high number of small valid INDs. The reason why Find$_2$ terminates with an *out of memory* exception is the complexity of hypergraphs created by Find$_2$. If one of these hypergraphs is not sparse (irreducible), then the hyperclique-finding subroutine presented in [6] attempts to simplify the corresponding hypergraph by removing hyperedges from it. The removing of hyperedges performed by this subroutine of Find$_2$ is not defined deterministically. This behavior causes a lot of recursive calls and consumes a huge amount of memory. Find$_2$ needed less time than Mind$_2$ only for the table pair 5 and 7, respectively. This is because the created hypergraphs for these table pairs are sparse, respectively.

Table 3. Comparing Mind$_2$'s runtime with Find$_2$'s runtime using MusicBrainz database (o.o.M. = out of memory, m = minutes)

TP.	$\|\mathcal{I}_u\|$	$\|\mathcal{I}_M\|$	n_{max} (x Nr.)	n_{min} (x Nr.)
1	19	75	5 (x 2)	2 (x 4)
2	17	25	3 (x 13)	2 (x 12)
3	15	28	3 (x 17)	2 (x 11)
4	15	56	3 (x 56)	-
5	14	28	3 (x 20)	2 (x 8)
6	13	23	3 (x 6)	2 (x 17)
7	12	26	3 (x 19)	2 (x 7)
8	12	11	3 (x 11)	-

TP.	Mind$_2$	Find$_2$
1	184 m	o.o.M. after 250 m
2	4 m	o.o.M. after 40 m
3	2 m	o.o.M. after 33 m
4	1.5 m	o.o.M. after 322 m
5	15 m	4 m
6	15 m	o.o.M. after 33 m
7	22 m	6 m
8	11 m	30 m

Experiment Groups 2. The purpose of these experiments is to compare Mind$_2$'s performance with the performance of the best case for Find$_2$. The best case for Find$_2$ (also for Zigzag) is when Find$_2$ needs to build only the

[1] https://musicbrainz.org.

Table 4. Results of the experiments in groups 2 and 3 (# = number of, m = minutes)

DB	$\|\mathcal{I}_M\|$	n_{max} (x Nr.)
1	1	9 (x 1)
2	1	10 (x 1)
3	8	8 (x 8)
4	9	9 (x 9)

	#DB-Accesses		Runtime	
DB	FIND$_2$	MIND$_2$	FIND$_2$	MIND$_2$
1	37	18	57 m	11 m
2	46	20	100 m	12 m
3	509	18	263 m	9.5 m
4	1021	20	906 m	11.5 m

2-hypergraph and then finds only one clique representing a valid IND. This happens for example when the database contains only one valid IND σ of size $n > 2$. In this case, FIND$_2$ needs $n \times (n-1)/2$ database access to enumerate the valid binary INDs and one access to validate the clique. To demonstrate this case, we generated two synthetic databases DB 1 and DB 2. Both databases contain 16,000,000 tuples. DB 1 contains one valid maximum IND in size 9. While DB 2 contains one valid maximum IND in size 10. The results of these experiments are presented in Table 4 (rows 1 and 2 in each part of Table 4). FIND$_2$'s runtime is dominated by the runtime of the required *SQL* queries for enumerating the valid binary INDs. Therefore, MIND$_2$ is up to 8x faster than FIND$_2$.

Experiment Groups 3. The purpose of these experiments is to show that in some cases FIND$_2$ needs the same exponential number of database accesses as needed by the Apriori approach. Let $\sigma = \{u_1, \ldots, u_n\}$ be an invalid n-ary IND with the property that every $(n-1)$-ary IND contained in σ is a valid IND. In this case, FIND$_2$ builds $n-2$ k-hypergraphs $(2 \leq k \leq n-1)$ where every k-hypergraph has $\binom{n}{k}$ edges and contains only the same clique, namely $\{u_1, \ldots, u_n\}$. Thus, FIND$_2$ needs $\binom{n}{2} + \cdots + \binom{n}{n-1} + (n-1) = 2^n - 3$ *SQL* queries to discover the n valid $(n-1)$-ary INDs contained in σ. To illustrate this case, we also generated two synthetic databases DB 3 and DB 4, where every database has 10,000,000 tuples in average. DB 3 contains 8 valid INDs in size 8. While DB 4 contains 9 valid INDs in size 9. Table 4 (rows 3 and 4 in each part of this table) presents the results of these experiments. The FIND$_2$'s runtime is dominated by the exponential number of the database accesses needed for the validation of the IND candidates. Therefore, MIND$_2$ is much more (up to 82x) faster than FIND$_2$.

6 Related Work

Kantola et al. [5] give an upper bound for the complexity of the IND-detecting problem and proof of its NP-completeness. Casanova et al. [3] formulate the simple axiomatization for INDs and prove that the decision problem for INDs is PSPACE-complete. Köhler and Link [9] investigated INDs and NOT NULL constraints under simple and partial semantics from theoretical point of view.

N-ary INDs. FIND$_2$ proposed by Koeller and Rundensteiner [6,7] begins by exhaustively validating unary and binary INDs, forming a 2-uniform hyper-

graph using unary INDs as nodes and binary INDs as edges. Then the algorithm proceeds in stages enumerated by a $k = 2, 3, \ldots$. In every stage k, all hypercliques are detected by HYPERCLIQUE algorithm [6] in the k-hypergraph, where every hyperclique represents an IND candidate. Then IND candidates are checked for validity in the database. Each invalid IND corresponding to hyperclique in the k-hypergraph is broken into $(k+1)$-ary INDs contained in it. Then the $(k+1)$-ary INDs form the edges of a $(k+1)$-hypergraph. Edges corresponding to invalid $(k+1)$-ary INDs are removed from the $(k+1)$-hypergraph. The process is repeated for increasing k until no new cliques are found. DeMarchi and Petit [14] developed ZIGZAG algorithm based on borders of theories [12]. Initially and for a k specified by the user, ZIGZAG initializes the positive border and the negative border by applying an adaptation of the level-wise algorithm MIND until the level k is reached. Furthermore, ZIGZAG introduces the optimistic positive border computed by finding minimal hypergraph traversals in a hypergraph generated from the negative border. The algorithm iteratively updates the three borders as long as the optimistic positive border contains INDs that are not contained in the positive border. Every updating process combines a pessimistic bottom-up with an optimistic top-down search. In the bottom-up search ZIGZAG validates IND candidates against the database. While in the top-down approach it estimates the distance between invalid INDs and the positive border by counting the number of tuples that do not satisfy these invalid INDs. MIND proposed by Marchi et al. [13] applies the level-wise approach to generate IND candidates. MIND generates all $(k+1)$-IND candidates from the valid k-INDs and the valid unary INDs. It is based on the view that the validity of $\sigma_1 = R[A_1, \ldots, A_k] \subseteq S[B_1, \ldots, B_k]$ and the validity of $\sigma_2 = R[A_{k+1}] \subseteq S[B_{k+1}]$ are necessary but not sufficient conditions for $\sigma = R[A_1, \ldots, A_k, A_{k+1}] \subseteq S[B_1, \ldots, B_k, B_{k+1}]$ to be valid. That is, if σ_1 or σ_2 is invalid, then it is impossible for σ to be valid. In this case, σ is pruned and no testing for its validity is necessary. In the other case, if both of σ_1 and σ_2 are valid, then σ has a chance to be valid and therefore becomes a candidate of size $k+1$. This candidate is then validated against the database. After all the $(k+1)$-ary IND candidates are generated and tested, the algorithm generates and tests $(k+2)$-ary IND candidates.

Unary INDs. Shaabani and Meinel developed S-INDD [18], a scalable algorithm for discovering unary INDs in large datasets. S-INDD introduces the concept of attribute clustering. Deriving unary INDs from the attribute clustering eliminates the redundant intersection operations resulting from deriving them from the inverted index applied in [13]. Furthermore, Shaabani and Meinel have shown that SPIDER [1] is a special case of S-INDD and that S-INDD is much more scalable than SPIDER. SPIDER [1] is presented by Bauckmann et al. The algorithm first sorts the distinct values in all columns and then uses a parallel merge-sort like algorithm to compute all unary INDs simultaneously. Papenbrock et al. presented BINDER [19]. BINDER applies a divide and conquer technique for discovering unary INDs. The main goal of BINDER's approach was to improve SPIDER's performance. BINDER takes a further step to generate

all n-ary INDs by applying string concatenations and the same Apriori strategy applied by MIND [13]. This approach results in an exponential number of I/O-operations and exponentially increases the original data size.

Foreign Key Discovery. Zhang et al. [20] applied approximation techniques for profiling foreign keys. Memari et al. [16] proposed algorithms for profiling foreign keys under the different semantics for NULL markers of the SQL Standard.

7 Conclusion and Future Work

We developed MIND$_2$, a new approach for mining maximum inclusion dependency between two relations. MIND$_2$ is based on a new characterization of maximum INDs. We achieved this characterization by only defining set operations on unary IND coordinates, a new concept we also introduced in this paper. Applying these set operations on unary IND coordinates enables discovering maximum INDs without any candidate generation, which has a big impact on a scalable discovery of long n-ary INDs. This work is the main milestone for our further works: as MIND$_2$'s performance is quadratically bounded by the number of tuples, we work in a distributed version of MIND$_2$ in order to parallelize both the computation of unary IND coordinates and the computation of maximum INDs.

References

1. Bauckmann, J., Leser, U., Naumann, F.: Efficiently computing inclusion dependencies for schema discovery. In: ICDE Workshops (2006)
2. Casanova, M.A., Fagin, R., Papadimitriou, C.H.: Inclusion dependencies and their interaction with functional dependencies. In: PODS (1982)
3. Casanova, M.A., Tucherman, L., Furtado, A.L.: Enforcing inclusion dependencies and referencial integrity. In: VLDB (1988)
4. Gryz, J.: Query folding with inclusion dependencies. In: Proceedings of the 14th IEEE Internation Conference on Data Engineering (ICDE 1998), pp. 126–133 (1998)
5. Kantola, M., Mannila, H., Räihä, K.J., Siirtola, H.: Discovering functional and inclusion dependencies in relational databases. JIIS **7**(7), 591–607 (1992)
6. Koeller, A., Rundensteiner, E.: Discovery of high-dimensional inclusion dependencies. Technical Reports WPI-CS-TR-02-15, Worcester Polytechnic Institute (2002)
7. Koeller, A., Rundensteiner, E.: Discovery of high-dimensional inclusion dependencies. In: ICDE, pp. 683–685 (2003)
8. Koeller, A., Rundensteiner, E.A.: Heuristic strategies for inclusion dependency discovery. In: Meersman, R. (ed.) OTM 2004. LNCS, vol. 3291, pp. 891–908. Springer, Heidelberg (2004)
9. Köhler, H., Link, S.: Inclusion dependencies reloaded. In: CIKM 2015 (2015)
10. Levene, M., Vincent, M.W.: Justification for inclusion dependency normal form. IEEE Trans. Knowl. Data Eng. **12**, 2000 (2000)
11. Liu, J., Li, J., Liu, C., Chen, Y.: Discover dependencies from data- a review. IEEE Trans. Knowl. Data Eng. **24**(2), 251–264 (2012)

12. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledgediscovery. Data Min. Knowl. Discov. **1**(3), 241–258 (1997)
13. Marchi, F.D., Lopes, S., Petit, J.M.: Efficient algorithms for mining inclusion dependencies. In: EDBT 2002, pp. 464–476 (2002)
14. Marchi, F.D., Petit, J.M.: Zigzag: A new algorithm for mining large inclusion dependencies in databases. In: ICDM (2003)
15. Marchi, F., Lopes, S., Petit, J.M.: Unary and n-ary inclusion dependency discovery in relational databases. JIIS **32**, 53–73 (2009)
16. Memari, M., Link, S., Dobbie, G.: SQL Data Profiling of Foreign Keys. In: Johannesson, P., Lee, M.L., Liddle, S.W., Opdahl, A.L., López, O.P. (eds.) Conceptual Modeling. LNCS, vol. 9381, pp. 229–243. Springer, Heidelberg (2015)
17. Miller, R.J., Hernández, M.A., Haas, L.M., Yan, L., Howard Ho, C.T., Fagin, R., Popa, L.: The clio project: Managing heterogeneity. SIGMOD Rec. **30**, 78–83 (2001)
18. Shaabani, N., Meinel, C.: Scalable inclusion dependency discovery. In: Renz, M., Shahabi, C., Zhou, X., Cheema, M.A. (eds.) DASFAA 2015. LNCS, vol. 9049, pp. 425–440. Springer, Heidelberg (2015)
19. Papenbrock, T., Sebastian Kruse, J.: Divide & conquer-based inclusion dependency discovery. VLDB **8**, 774–785 (2015)
20. Zhang, M., Hadjieleftheriou, M., Ooi, B.C., Procopiuc, C.M., Srivastava, D.: On multi-column foreign key discovery. VLDB **3**, 805–814 (2010)