Incremental Discovery of Inclusion Dependencies

Nuhad Shaabani
Hasso-Plattner-Institut, University of Potsdam
Prof. Dr. Helmert-Str. 2-3
Potsdam, Germany 14482
nuhad.shaabani@hpi.de

Christoph Meinel
Hasso-Plattner-Institut, University of Potsdam
Prof. Dr. Helmert-Str. 2-3
Potsdam, Germany 14482
christoph.meinel@hpi.de

ABSTRACT

Inclusion dependencies form one of the most fundamental classes of integrity constraints. Their importance in classical data management is reinforced by modern applications such as data profiling, data cleaning, entity resolution and schema matching. Their discovery in an unknown dataset is at the core of any data analysis effort. Therefore, several research approaches have focused on their efficient discovery in a given, static dataset. However, none of these approaches are appropriate for applications on dynamic datasets, such as transactional datasets, scientific applications, and social network. In these cases, discovery techniques should be able to efficiently update the inclusion dependencies after an update in the dataset, without reprocessing the entire dataset.

We present the first approach for incrementally updating the unary inclusion dependencies. In particular, our approach is based on the concept of attribute clustering from which the unary inclusion dependencies are efficiently derivable. We incrementally update the clusters after each update of the dataset. Updating the clusters does not need to access the dataset because of special data structures designed to efficiently support the updating process. We perform an exhaustive analysis of our approach by applying it to large datasets with several hundred attributes and more than 116,200,000 million tuples. The results show that the incremental discovery significantly reduces the runtime needed by the static discovery. This reduction in the runtime is up to 99.9996 % for both the insert and the delete.

CCS CONCEPTS

•Information systems →Data management systems; Information integration;

KEYWORDS

Metadata, Clustering, Data discovery, Incremental discovery

ACM Reference format:
Nuhad Shaabani and Christoph Meinel. 2017. Incremental Discovery of Inclusion Dependencies. In *Proceedings of SSDBM '17, Chicago, IL, USA, June 27-29, 2017,* 12 pages.
DOI: http://dx.doi.org/10.1145/3085504.3085506

SSDBM '17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3085504.3085506

1 INTRODUCTION

Many emerging applications produce large datasets at a fast rate. Examples of such dynamic data include not only scientific measurements, social networks, but traditional transactions as well. As the volume of data generated by applications continues to increase the need for a better understanding increases too. Knowing the structure and properties of such datasets is important for data integration, data analytics, query optimization, and many further applications. For instance, when importing raw data (e.g., from scientific experiments or extracted from the Web) into a DBMS, it is often necessary to profile the data and then devise an adequate schema. A fundamental task of data profiling is to discover metadata about relationships between the attributes in the dataset [5, 18, 20, 23].

An important part of such metadata is inclusion dependencies (INDs). 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, making INDs important for many applications, such as data integration [17], integrity checking [3], query optimization [8], or schema redesign [13]. In particular, INDs are useful to discover foreign-primary key relationships, which are a necessity for suggesting join paths, data linkage, and data normalization [25]. According to their complexities, existing algorithms for exhaustively discovering INDs in a given static dataset can be divided into two categories: Algorithms for discovering unary INDs (i.e., INDs cover pairs of single attributes) [2, 15, 19, 21] and algorithms for discovering n-ary INDs (i.e., INDs cover pairs of n attributes) [11, 14, 15, 22]. The complexity of the algorithms in the former category is NP-hard [9], while the complexity of the algorithms in the latter category is quadratic in the number of the attributes. All algorithms for detecting n-ary INDs require the discovery of all unary INDs (uINDs) in the corresponding dataset 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 dataset.

However, a dataset is hardly ever fixed: Transactional data are appended to frequently, analytics-oriented datasets experience periodic updates (typically daily), and large datasets available on the Web are updated every few weeks or months. These data changes can cause the metadata to quickly become out-of-date [5, 18, 20, 23].

Example 1.1. Consider the dataset presented in Table 1. The set of valid unary INDs in this dataset is $\mathcal{I} = \{A \subseteq B, D \subseteq B\}$. Now assume the following two cases:

(1) Delete – The tuple (b, b, e, a) (i.e., the second tuple) is deleted from the dataset,which makes $A \subseteq B$ invalid because deleting (b, b, e, a) makes $b \in A$ and $b \notin B$. Thus, this deletion changes the current set of valid uINDs $\mathcal{I}$ to the set $\mathcal{I}_1 = \{D \subseteq B\}$.

Table 1: Running example

A	B	C	D
a	a	f	a
b	b	e	a
b	c	f	c
d	d	f	c

(2) Insert – The new tuple (c, c, e, e) is inserted into the dataset, which creates a new valid IND, namely $B \subseteq A$. Furthermore, it makes $D \subseteq B$ invalid. Thus, inserting the tuple (c, c, e, e) changes the previous set of valid uINDs $\mathcal{I}$ to the set $\mathcal{I}_2 = \{A \subseteq B, B \subseteq A\}$.

The previous example demonstrates that a data change can cause new INDs to appear or existing INDs to disappear, which means that the set of INDs may change and, therefore, may need to be updated after a change in the corresponding dataset. The current solution to keep the set of INDs up-to-date after arrival or deletion of data is to completely rediscover them. This rediscovering process requires applying one of the existing algorithms to the entire dataset because none of them are suitable for working on dynamic datasets.

However, this solution hurts performance significantly since (i) an initial dataset size is typically several orders of magnitude bigger than the size of a change in the dataset, and (ii) the performance of the IND discovering algorithms depends not only on the number of attributes but, for the most part, also on the number of tuples. Furthermore, rediscovering the set of INDs in the entire dataset does not take advantage of previously discovered INDs which might become only partly invalid after a data change. Therefore, an incremental approach is the most efficient way to keep the INDs up-to-date after the arrival or deletion of data [5, 7, 18, 20].

Contributions. This paper presents a new approach that can incrementally update the set of uINDs when new tuples are inserted and when existing tuples are deleted or changed (i.e., values are modified). In particular, we make the following contributions.

- We present a system architecture for modeling the workflow of our approach for the incremental detection of unary INDs in continuously changing datasets (Section 3).
- We realize the incremental update of uINDs through the incremental update of the attribute clustering. For incrementally updating the clusters, we define operations to be applied after each data change (Section 4).
- We develop algorithms and data structures that efficiently implement the incremental update of clusters (Section 5).
- We present the results of exhaustive experiments conducted on five large datasets with hundreds of attributes and more than 116,200,000 million tuples. The experiments show that our incremental approach can reduce the runtime of the static discovery by up to 99.9996 % (Section 6).

2 PROBLEM STATEMENT

Let $\mathcal{D}$ be database instance over a set $\mathcal{R}$ of relations. We denote the set of all attributes in $\mathcal{D}$ by $\mathcal{A}$, and an instance of a relation $R \in \mathcal{R}$ with r. The number of attributes in R is $|R|$ and the number of tuples in r is $|r|$. 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 tuple $r_i \in r$ $(1 \leq i \leq |r|)$ on X.

Definition 2.1. (IND) Let $R[A_1, \ldots, A_{|R|}]$ and $S[B_1, \ldots, B_{|S|}]$ be two relations in $\mathcal{R}$. 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 $r_i[X] = s_j[Y]$.

To simplify the formulation, 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 values occurring in the corresponding instance of the relation 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.

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

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 $\mathcal{D}$ is equivalent to the computation of the set:

$$\mathcal{I} = \{A \subseteq B \mid A, B \in \mathcal{A} \wedge \mathcal{V}_A \subseteq \mathcal{V}_B\}$$

We refer to any algorithm in [2, 15, 19, 21] for discovering the set $\mathcal{I}$ as a static discovery of $\mathcal{I}$. When the dataset $\mathcal{D}$ changes over time, the set $\mathcal{I}$ may also change. A new tuple may create a new valid uIND or may make a valid one invalid. Removing a tuple from a relation instance can also have the same effect on the set $\mathcal{I}$ (see Example 1.1). The current solution to keep the set $\mathcal{I}$ up-to-date is to rediscover the entire dataset $\mathcal{D}$ by applying one of the static discovery algorithms. However, the big disadvantage of this solution is the computation time required by these algorithms. Moreover, none of them take advantage of the previously discovered uINDs.

Therefore, the question that an incremental approach has to answer is how to efficiently update the set $\mathcal{I}$ within a short period of time without processing the entire dataset $\mathcal{D}$? Thus, the time needed by an incremental approach for updating the set $\mathcal{I}$ should be negligible in comparison to the time needed by a static approach. To define this requirement precisely, we denote the insert of a tuple t into any instance $r \in \mathcal{D}$ by $\mathcal{D} + \{t\}$, and the delete of t from r by $\mathcal{D} - \{t\}$. We refer to inserting or deleting a tuple as $\mathcal{D} \pm \{t\}$.

Definition 2.2. (Requirement for the incremental discovery) Let $T_{inc}(\{t\})$ be the time needed by an incremental approach for updating $\mathcal{I}$ after inserting or deleting t. Let $T_{st}(\mathcal{D} \pm \{t\})$ be the time needed by a static approach for discovering $\mathcal{I}$ in $\mathcal{D} \pm \{t\}$. Then

$$T_{inc}(t) + T_{st}(\mathcal{D} \pm \{t\}) \approx T_{st}(\mathcal{D} \pm \{t\})$$

must be fulfilled.

This definition was inspired by [7].

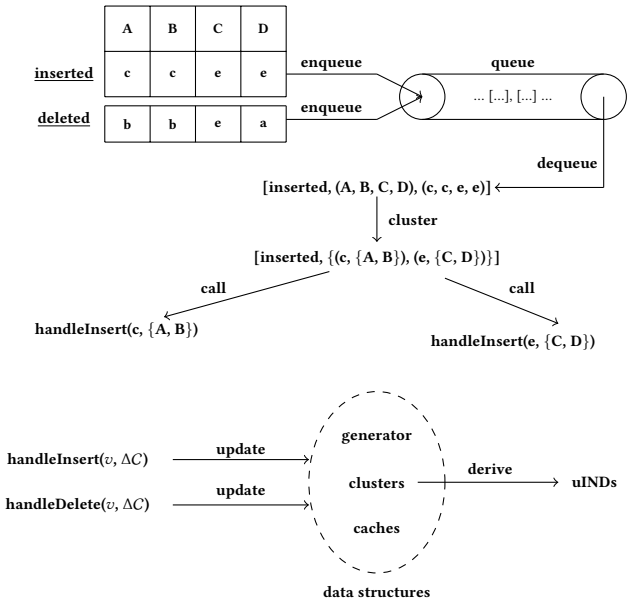

Figure 1: Workflow overview

3 WORKFLOW OVERVIEW

Figure 1 gives an overview of the workflow of our system for incrementally detecting uINDs. The system enqueues every tuple immediately after its insertion or deletion from the dataset, and also checks the queue periodically. If the queue is not empty, a tuple will be dequeued to be processed. In the first step, the processing of a tuple consists of mapping each value v in the tuple to the set of all tuple attributes to which this value belongs. We denote such a set of tuple attributes as ΔC. This mapping is a special case of attribute clustering introduced in Section 4. For instance, after dequeuing the tuple $t = (c, c, e, e)$, it is mapped to the set $\{(c, \{A, B\}), (e, \{C, D\})\}$.

Then for each $(v, \Delta C)$, the system calls Algorithm 1 to handle the insertion if the dequeued tuple has been inserted. Otherwise it calls Algorithm 6 to handle the deletion. Both algorithms are presented in Section 5. Algorithms 1 and 6 work on a set of data structures presented in Subsection 5.1 to keep the attribute clustering of the entire dataset up-to-date after an insertion or deletion of a tuple.

Keeping the attribute clustering up-to-date is one of the key points in our design because all valid uINDs are efficiently derivable from the attribute clustering as explained in Section 4.

Updating the attribute clustering $\mathcal{AC}$ of a dataset $\mathcal{D}$ is based on two operators applied on $\mathcal{AC}$ and the attribute clustering $\Delta\mathcal{AC}$ of the dequeued tuple referred to as t. If t is inserted, then we apply the merge operator (see Definition 4.4). Otherwise, we apply the extract operator (see Definition 4.8). The result of the merge operator is the attribute clustering of $\mathcal{D} + \{t\}$, while the result of extract operator is the attribute clustering of $\mathcal{D} - \{t\}$.

Updating the attribute clustering after changing an existing tuple can be achieved by a composite operation consisting of the merge operation and extract operation as illustrated in Subsection 4.2.

The queue will be almost empty if the time between two consecutive dataset operations is less than the time T_{inc} needed by the system to update the attribute clustering after the data change (i.e., in this case, updating the attribute clustering will be in real time). The specific implementation of the queue can be vary from a simple queue in the main memory to a dedicated queuing server. In our implementation, the queue is based on the file system.

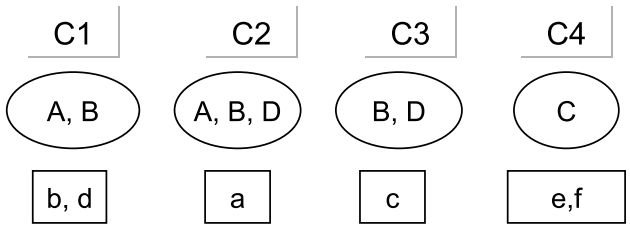

Figure 2: Attribute clustering based on the data of Table 1

4 ATTRIBUTE CLUSTERING

Attribute clustering is a concept introduced in [21]. From *attribute clustering* we can derive all uINDs efficiently because deriving uINDs from the *attribute clustering* eliminates all redundant intersection operations resulting from deriving them from the inverted index applied in [2, 15, 19]. In fact, as shown in [21] the number of the redundant intersection operations is $|\mathcal{V}| - |\mathcal{AC}|$ where $\mathcal{AC}$ is the set of the clusters.

Because of the efficiency of deriving uINDs from the attribute clustering, we reduce the problem of incrementally updating uINDs to incrementally updating the clusters. Therefore, we define new operations on the clusters to be applied after each data change, which keeps the attribute clustering of the dataset up-to-date.

4.1 Background

Definition 4.1. (Attribute clustering) Let $f : \mathcal{V} \rightarrow 2^{\mathcal{A}}$ be a function with the property:

$$(\forall v \in \mathcal{V})(\neg\exists A \in \mathcal{A}) : v \in \mathcal{V}_A \wedge A \notin f(v)$$

That is, $f(v)$ is the maximum set of attributes $A \in \mathcal{A}$ with $v \in \mathcal{V}_A$. We call the image of f:

$$\mathcal{AC} = f(\mathcal{V}) = \{C \subseteq \mathcal{A} \mid \exists v \in \mathcal{V} : f(v) = C\}$$

attribute clustering over the set $\mathcal{V}$. We call the function f the generator of $\mathcal{AC}$.

Thus, every cluster in the *attribute clustering* is a maximum subset of attributes that has a subset of values in common. For example, $\mathcal{AC} = \{\{A, B\}, \{A, B, D\}, \{B, D\}, \{C\}\}$ (shown in Figure 2) is the *attribute clustering* over the values of the dataset in Table 1. Attributes A and B shape cluster C_1 because both share the values $\{b, d\}$ that can not be shared by any superset of C_1. Every attribute $A \in \mathcal{A}$, must be contained in at least one cluster, and every value $v \in \mathcal{V}$ must belong to only one cluster. [21] has shown that for any dataset $\mathcal{D}$, there is always only one attribute clustering to satisfy Definition 4.1.

Based on the attribute clustering, we decide whether or not $A \subseteq B$ is a valid IND for two attributes $A, B \in \mathcal{A}$ as follows [21].

Lemma 4.2. *Let $\mathcal{AC} = \{C_1, \ldots, C_c\}$ be the attribute clustering over $\mathcal{V}$. Then the following holds:*

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

Thus, attribute A is included in attribute B if and only if every cluster containing A contains B. For example, the set of D values in Table 1 is included in the set of B values because the clusters $C2$ and $C3$ that both contain D also contain B (see Figure 2), but B values are not included in D values because cluster $C1$ contains B and does not contain D.

4.2 Attribute clustering operations

Our purpose is to update the attribute clustering $\mathcal{AC}$ of $\mathcal{D}$ after inserting or deleting a tuple t, meaning that our goal is two folds: (i) to compute the attribute clustering of $\mathcal{D} + \{t\}$ from $\mathcal{AC}$ and $\{t\}$ after inserting t into $\mathcal{D}$, and (ii) to compute the attribute clustering of $\mathcal{D} - \{t\}$ also from $\mathcal{AC}$ and $\{t\}$ but after deleting t from $\mathcal{D}$. To achieve these goals, we compute the attribute clustering of $\{t\}$, referred to as $\Delta\mathcal{AC}$, then (i) we define the merge operator on $\mathcal{AC}$ and $\Delta\mathcal{AC}$ to obtain the attribute clustering of $\mathcal{D} + \{t\}$, and (ii) we define the extract operator on $\mathcal{AC}$ and $\Delta\mathcal{AC}$ to obtain the attribute clustering of the dataset $\mathcal{D} - \{t\}$.

Attribute clustering of a tuple. Let t be a one-tuple instance of a relation $R \in \mathcal{R}$. We denote the set of all values occurring in t by $\Delta\mathcal{V}$. Accordingly, the *attribute clustering* over $\Delta\mathcal{V}$ is $\Delta\mathcal{AC}$ and the generator of $\Delta\mathcal{AC}$ is Δf.

Example 4.3. We consider Figure 1. For $\{t\} = \{(c, c, e, e)\}$, we have $\Delta\mathcal{V} = \{c, e\}$, $\Delta f(c) = \{A, B\}$, and $\Delta f(e) = \{C, D\}$. Thus, $\Delta\mathcal{AC} = \{\{A, B\}, \{C, D\}\}$.

Merge operator. The merge operator merges the *attribute clustering* $\mathcal{AC}$ of $\mathcal{D}$ and the *attribute clustering* $\Delta\mathcal{AC}$ of an inserted tuple t to produce the *attribute clustering* $\mathcal{AC} + \Delta\mathcal{AC}$ of $\mathcal{D} + \{t\}$.

Definition 4.4. (Merge operator) Let $f : \mathcal{V} \to 2^{\mathcal{A}}$ be the generator of $\mathcal{AC}$, and $\Delta f : \Delta\mathcal{V} \to 2^{\mathcal{A}}$ be the generator of $\Delta\mathcal{AC}$. We define $f + \Delta f : \mathcal{V} \cup \Delta\mathcal{V} \to 2^{\mathcal{A}}$ as

$$(f + \Delta f)(v) = \begin{cases} f(v) \cup \Delta f(v) & \text{if } v \in \mathcal{V} \cap \Delta\mathcal{V} \\ f(v) & \text{if } v \in \mathcal{V} \setminus \Delta\mathcal{V} \\ \Delta f(v) & \text{if } v \in \Delta\mathcal{V} \setminus \mathcal{V} \end{cases}$$

The merge of $\mathcal{AC}$ and $\Delta\mathcal{AC}$ is $\mathcal{AC} + \Delta\mathcal{AC} = (f + \Delta f)(\mathcal{V} \cup \Delta\mathcal{V})$.

Lemma 4.5. *$\mathcal{AC} + \Delta\mathcal{AC}$ is the attribute clustering over $\mathcal{V} \cup \Delta\mathcal{V}$ and $f + \Delta f$ its generator.*

Proof. For any $v \in \mathcal{V} \cup \Delta\mathcal{V}$, we have to show that $(f + \Delta f)(v)$ is the maximum set of attributes whose value sets contain v.

i) For $v \in \mathcal{V} \cap \Delta\mathcal{V}$, we assume that $(f + \Delta f)(v) = f(v) \cup \Delta f(v)$ is not the maximum set of attributes whose value sets contain v, which means that there is some attribute A with $v \in \mathcal{V} \cap \Delta\mathcal{V}$, $A \notin f(v)$ and $A \notin \Delta f(v)$. $v \in \mathcal{V}_A$ and $A \notin f(v)$ is a contradiction to the fact that $f(v) \in \mathcal{AC}$. $v \in \Delta\mathcal{V}_A$ and $A \notin \Delta f(v)$ is a contradiction to the fact that $\Delta f(v)$ is a cluster in $\Delta\mathcal{AC}$. Thus, our assumption is wrong.

ii) The case $v \in \mathcal{V} \setminus \Delta\mathcal{V}$ means that inserting t does not add v to the column of any attribute $A \in \mathcal{A}$, meaning that v has the same cluster in $\mathcal{D}$ and in $\mathcal{D} + \{t\}$. Thus, we have $(f + \Delta f)(v) = f(v)$.

iii) The case $v \in \Delta\mathcal{V} \setminus \mathcal{V}$ means that v does not occur in $\mathcal{D}$, and after inserting t, occurs in $\mathcal{D} + \{t\}$ in the columns of all attributes in $\Delta f(v)$. Thus, $\Delta f(v)$ is the cluster of v in $\mathcal{D} + \{t\}$. □

Example 4.6. What is the attribute clustering of the dataset in Table 1 after inserting (c, c, e, e)? According to Figure 2 and Example 4.3, we have $(f + \Delta f)(c) = \{B, D\} \cup \{A, B\} = \{A, B, D\}$, and $(f + \Delta f)(e) = \{C\} \cup \{C, D\} = \{D, C\}$, because $c, e \in \mathcal{V} \cap \Delta\mathcal{V}$. For $x \in \{a, b, d, f\}$, we have $(f + \Delta f)(x) = f(x)$ because $x \in \mathcal{V} \setminus \Delta\mathcal{V}$. Thus, $\mathcal{AC} + \Delta\mathcal{AC} = \{\{A, B, D\}, \{A, B\}, \{C, D\}, \{C\}\}$. Based on Lemma 4.2, the set of uINDs is now $\mathcal{I} = \{A \subseteq B, B \subseteq A\}$.

Extract operator. The extract operator extracts the *attribute clustering* $\Delta\mathcal{AC}$ of a deleted tuple t from *attribute clustering* $\mathcal{AC}$ of $\mathcal{D}$ to produce the *attribute clustering* $\mathcal{AC} - \Delta\mathcal{AC}$ of $\mathcal{D} - \{t\}$.

Before defining the extract operator, we have to identify the set of distinct values in $\mathcal{D} - \{t\}$. For this purpose, we define for every $v \in \mathcal{V}$ and every attribute $A \in \mathcal{A}$ the variable F_A^v indicating the frequency of occurrences of the value v in the column of A before deleting t. Then we introduce the following operator.

Definition 4.7. (Bag difference) We call the set

$$\mathcal{V} \setminus \{v \in \Delta\mathcal{V} \mid (\forall A \notin \Delta f(v) : F_A^v = 0) \wedge (\forall A \in \Delta f(v) : F_A^v = 1)\}$$

the bag difference between $\mathcal{V}$ and $\Delta\mathcal{V}$, and refer to it as $\mathcal{V} \setminus_b \Delta\mathcal{V}$.

Now, it is obvious that the set of values in $\mathcal{D} - \{t\}$ is $\mathcal{V} \setminus_b \Delta\mathcal{V}$.

Definition 4.8. (Extract operator) Let $f : \mathcal{V} \to 2^{\mathcal{A}}$ be the generator of $\mathcal{AC}$, and $\Delta f : \Delta\mathcal{V} \to 2^{\mathcal{A}}$ be the generator of $\Delta\mathcal{AC}$. We define $f - \Delta f : \mathcal{V} \setminus_b \Delta\mathcal{V} \to 2^{\mathcal{A}}$ as

$$(f - \Delta f)(v) = \begin{cases} f(v) & \text{if } v \notin \Delta\mathcal{V} \\ f(v) \setminus \{A \in \Delta f(v) \mid F_A^v = 1\} & \text{if } v \in \Delta\mathcal{V} \end{cases}$$

The extract of $\Delta\mathcal{AC}$ from $\mathcal{AC}$ is $\mathcal{AC} - \Delta\mathcal{AC} = (f - \Delta f)(\mathcal{V} \setminus_b \Delta\mathcal{V})$.

Lemma 4.9. *$\mathcal{AC} - \Delta\mathcal{AC}$ is the attribute clustering over $\mathcal{V} \setminus_b \Delta\mathcal{V}$ and $f - \Delta f$ its generator.*

Proof. For any $v \in \mathcal{V} \setminus_b \Delta\mathcal{V}$, we have to show that $(f - \Delta f)(v)$ is the maximum set of attributes whose value sets contains v.

i) For the case $v \notin \Delta\mathcal{V}$, there is no change in the cluster of v in $\mathcal{D}$. Thus, $(f - \Delta f)(v) = f(v)$ is the cluster of v in $\mathcal{D} - \{t\}$.

ii) For $v \in \Delta\mathcal{V}$, $(f - \Delta f)(v)$ is the cluster $f(v)$ after removing each attribute $A \in \Delta f(v)$ whose column does not contain v any more after deleting t. Thus, after deleting t from $\mathcal{D}$, $(f - \Delta f)(v)$ becomes the cluster of v in $\mathcal{D} - \{t\}$. □

Example 4.10. What is the attribute clustering of the dataset in Table 1 after deleting (b, b, e, a)? For this tuple, we have $\Delta\mathcal{V} = \{b, e, a\}$, $\Delta f(b) = \{A, B\}$, $\Delta f(e) = \{C\}$, $\Delta f(a) = \{D\}$. For $v \notin \Delta\mathcal{V}$, $(f - \Delta f)(v) = f(v)$. For $v = b$, $(f - \Delta f)(b) = \{A, B\} \setminus \{B\} = \{A\}$ because $F_A^b = 2$ and $F_B^b = 1$ (i.e., we do not remove $A \in \Delta f(b)$ from $f(b) = \{A, B\}$ because $F_A^b > 1$). For $v = e$, $(f - \Delta f)(e) = \{C\} \setminus \{C\} = \emptyset$ because $F_C^e = 1$. For $v = a$, $(f - \Delta f)(a) = f(a) = \{A, B, D\}$ because $F_D^a = 2$ (i.e., we do not remove $D \in \Delta f(a)$ from $f(a)$ because $F_D^a > 1$). Thus, $\mathcal{AC} - \Delta\mathcal{AC} = \{\{A, B, D\}, \{A, B\}, \{B, D\}, \{A\}, \{C\}\}$. Based on Lemma 4.2, the set of uINDs is now $\mathcal{I} = \{D \subseteq B\}$.

Updating $\mathcal{AC}$ after a change of an existing tuple. The following example shows how to update the attribute clustering after a change of an existing tuple.

Example 4.11. Assume that the tuple (b, c, f, c) (i.e., the third tuple in Table 1) has to be modified to be (b, g, f, g), meaning that (i) the value c has to be deleted once from the column of attribute B and once from the column of attribute D, and (ii) the value g has to be added once to the column of B and once to the column of D. We consider deleting c from B and D as deleting the tuple $t_1 = (B = c, D = c)$ from $\mathcal{D}$, while we consider adding g to B and D as inserting the tuple $t_2 = (B = g, D = g)$ into $\mathcal{D}$. The attribute clustering of t_1 is $\Delta\mathcal{AC}_1 = \{B, D\}$ with the generator $\Delta f_1(c) = \{B, D\}$, while the attribute clustering of t_2 is $\Delta\mathcal{AC}_2 = \{B, D\}$ with the generator $\Delta f_2(g) = \{B, D\}$. Thus, updating $\mathcal{AC}$ after changing the tuple (b, c, f, c) consists of the following two operations: (i) extracting $\Delta\mathcal{AC}_1$ from $\mathcal{AC}$, and (ii) merging $\mathcal{AC}$ with $\Delta\mathcal{AC}_2$.

5 ALGORITHMS

5.1 Data structures

Data structure for the clusters. For every cluster $C \in \mathcal{AC}$, we define a record $r_C = (cid, C)$ where, $cid \in \mathbb{N}$ identifies the cluster C uniquely in $\mathcal{AC}$. We denote the set of all these records r_C by *ACRecs*. To efficiently support retrieving and deleting a cluster by giving its identifier, we define an index on *ACRecs*. The keys of this index are the identifiers and the entries are the clusters. Furthermore, we define a second hash index to efficiently retrieve an identifier by giving the corresponding cluster. The keys in the second index are the clusters and the entries are the identifiers.

Data structure for the generator. For every value v in the dataset, we define a record $r_v = (v, F_1^v, \ldots, F_i^v, \ldots, F_{|\mathcal{A}|}^v), cid)$, where cid is the identifier of the cluster to which v belongs and F_i^v $(1 \leq i \leq |\mathcal{A}|)$ is the frequency of v occurrences in the column of A_i. We denote the set of all these records r_v by *GenRecs*. Every record $r_v \in GenRecs$ is uniquely identified by the corresponding value v because a value can belong to only one cluster (see Definition 4.1). We implement *GenRecs* as a relational table, which means that all operations defined on *GenRecs* are SQL queries. To efficiently retrieve, update, and delete a value record by giving the value, we define an index on the values. We also define an index on the identifiers of the clusters for efficiently retrieving and counting the values belonging to a certain cluster.

The reason why we implement the generator in terms of an external data structure is the number of values in $\mathcal{V}$. This number can be so large that *GenRecs* does not fit into the main memory, which is also the reason why we separate clusters from the data structure of the generator, meaning that we keep and manage the data structure of the clusters in the main memory. Table 2 presents the data structures of the generator for the dataset in Table 1 and the corresponding clusters in Figure 2.

Data structures for accessing the generator. During the process of updating *GenRecs* and *ACRecs*, a cluster C will be removed from *ACRecs* if the number of values belonging to it is zero, which requires that each time we have to know whether or not a cluster C is to be removed, we have to access *GenRecs* in order to compute the number of values belonging to the cluster. To reduce the number of required SQL queries for this case, we define a record $(cid, vcount)$ for every cluster. The variable $vcount$ is a counter of the values that have been added to, retrieved from, and/or deleted from *GenRecs* for the cluster identified by cid. We denote the set of all such records by *CountCache*. Thus, if $vcount \geq 1$ for a cluster C, then we can immediately decide that C is to not be removed from *ACRecs*. Otherwise, if there is no record in *CountCache* for C or if there is record with $vcount = 0$, then we have to retrieve the number of values belonging to C from *GenRecs*. To efficiently update and delete a counter in *CountCache* we define an index with the values as keys and the counters as entries.

For every value $v \in \Delta\mathcal{V}$, we have to determine whether or not it is new (i.e. $v \notin \mathcal{V}$). For the case that v is not new (i.e. $v \in \mathcal{V} \cap \Delta\mathcal{V}$), we have to know its cluster, which requires querying *GenRecs*. To reduce the database accesses in this case, we cache the record (v, cid) for every value that has been added to or retrieved from *GenRecs*. We refer to the set of all such records as *ValCache*. Thus, if there is a record in *ValCache* for an input value v, then we immediately know that v is not new. Otherwise, we have to query *GenRecs*. If the result of the query is empty, we conclude that there is no cluster to which v belongs and, therefore, v is new. For efficiently retrieving a record from *ValCache* we define an index with the values as keys and the cluster identifiers as entries.

We refer to *ValCache* and *CountCache* in the rest of the paper as *cache strategies*.

Table 2: Data structures for the generator and the clusters of the dataset in Table 1

v	F_A^v	F_B^v	F_C^v	F_D^v	cid
a	1	1	0	2	1
b	2	1	0	0	2
c	0	1	0	2	3
d	1	1	0	0	2
e	0	0	1	0	4
f	0	0	3	0	4

cid	cluster
1	$\{A, B, C\}$
2	$\{A, B\}$
3	$\{B, D\}$
4	$\{C\}$

Algorithm 1: handleInsert

```
Input  : v, ΔC
cid ← getClusterID(v)
if cid = null then
    handleInsertingNewValue(v, ΔC)
else
    handleInsertingExistingValue(v, ΔC, cid)
```

Algorithm 2: getClusterID

```
Input  : v
Output : cid
cid ← ValCache·getCID(v)
if cid = null then
    cid ← GenRecs·getCID(v)
    if cid ≠ null then
        ValCache·add(v, cid)
        CountCache·init(cid, 1)
```

Algorithm 3: handleInsertingNewValue

```
   Input  : v, ΔC
1  cid ← ACRecs.getCID(ΔC)
2  if cid = null then
3  |  cid ← ACRecs.add(ΔC)
4  if cid ∈ CountCache then
5  |  CountCache·.increase(cid)
6  else
7  |  CountCache·init(cid, 1)
8  ValCache·add(v, cid)
9  GenRecs·add(v, cid, ΔC)
```

Algorithm 4: handleInsertingExistingValue

```
   Input  : v, ΔC, cid
1  C ← ACRecs·getCluster(cid)
2  C′ ← C ∪ ΔC
3  if C′ = C then
4  |  GenRecs·iUpdate(v, ΔC)
5  else
6  |  cid′ ← ACRecs·getCID(C′)
7  |  if cid′ = null then
8  |  |  cid′ ← ACRecs.add(C′)
9  |  if cid′ ∈ CountCache then
10 |  |  CountCache·increase(cid′)
11 |  else
12 |  |  CountCache·init(cid′, 1)
13 |  GenRecs·iUpdate(v, cid′, ΔC)
14 |  ValCache·update(v, cid′)
15 |  CountCache·decrease(cid)
16 |  handleDeletingCluster(cid)
```

Algorithm 5: handleDeletingCluster

```
   Input  : cid
1  vc ← 0
2  if cid ∈ CountCache then
3  |  vc ← CountCache·getCount(cid)
4  |  if vc = 0 then
5  |  |  vc ← GenRecs·getCount(cid)
6  else
7  |  vc ← GenRecs·getCount(cid)
8  if vc = 0 then
9  |  ACRecs·remove(cid)
10 |  if cid ∈ CountCache then
11 |  |  CountCache·remove(cid)
12 else
13 |  if cid ∉ CountCache then
14 |  |  CountCache·init(cid, vc)
```

5.2 Handling Insert

Algorithm 1 implements the merge operator introduced in Definition 4.4. It receives a value v and the set ΔC containing all attributes into whose columns v has been inserted. If the value is new (i.e. $v \notin \mathcal{V}$), then it does not belong to any cluster in *ACRecs*. Consequently, there is no record for v in *GenRecs*. Otherwise, v already exists in the dataset and belongs to a cluster. Algorithm 1 calls Algorithm 2 to identify whether or not the value v is new.

Algorithm 2 returns the NULL-marker if the value v does not exist either in the cache *ValCahs* or in the set *GenRecs*. Otherwise, it returns the identifier of the cluster to which v belongs. At the beginning, Algorithm 2 tries to find the identifier of the cluster of v in the cache *ValCache* (Line 1). If the cache *ValCache* does not contain a record (v, cid) for the value v (Line 2), Algorithm 2 queries *GenRecs* (Line 3). If it has found a record for the value v in *GenRecs* (Line 4), then it adds v and the identifier of its cluster to the cache *ValCache* (Line 5) and creates a value counter for the cluster and initializes it with one, i.e., it adds the record $(cid, 1)$ to *CountCache* (Line 6).

Handling the insert of a new value. Algorithm 3 implements the merge operation for the case $v \in \Delta\mathcal{V} \setminus \mathcal{V}$. In this case, we have $(f + \Delta f)(v) = \Delta f(v) = \Delta C$ according to Definition 4.4 and Lemma 4.5, meaning that the attribute set ΔC is the cluster of the new value v. But ΔC is not necessarily a new cluster because ΔC can already exist in the set *ACRecs* if there is some value v' in *GenRecs* that is different from v and has ΔC as a cluster. To identify whether or not ΔC already exists in *ACRecs*, Algorithm 3 retrieves *ACRecs* for the existing of ΔC (Line 1). The operation *getCID(*ΔC*)* on *ACRecs* returns the identifier of ΔC if ΔC exists in *ACRecs*, otherwise, it returns the NULL-marker. If ΔC does not exist in *ACRecs*, Algorithm 3 adds ΔC to *ACRecs* (Lines 2-3). Now and after knowing the identifier of ΔC, Algorithm 3 has to update *CountCache*, *ValCache*, and *GenRecs*, increasing the number of values belonging to ΔC by one if the identifier of ΔC exists in *CountCache*. Otherwise, it creates a value counter for ΔC and initializes it with one (Line 7). Notice that the value counter for a cluster exits in *CountCache* only if a value belonging to the cluster has been handled. For *ValCache*, Algorithm 3 adds the new value v with the identifier of its cluster to *ValCache* (Line 8), adds a new record $(v, F_1^v, \ldots, F_{|\mathcal{A}|}^v, cid)$ to *GenRecs* (Line 9) where F_i^v $(1 \leq i \leq |\mathcal{A}|)$ is initialized as follows.

$$F_i^v = \begin{cases} 1 & \text{if } A_i \in \Delta C \\ 0 & \text{if } A_i \notin \Delta C \end{cases} \tag{1}$$

We initialize F_i^v with one for each $A_i \in \Delta C$ because v is new and has been added only once to each column of $A_i \in \Delta C$.

Handling the insert of an existing value. Algorithm 4 implements the merge operator for the case $v \in \mathcal{V} \cap \Delta\mathcal{V}$. Based on Definition 4.4 and Lemma 4.5, the new cluster of the value v is $C' = (f + \Delta f)(v) = f(v) \cup \Delta f(v) = C \cup \Delta C$, where C is the current v's cluster identified by *cid*. If $\Delta C \subseteq C$, we have $C' = C$. For this case there is no change in *ACRecs*. But what Algorithm 4 needs to do here is only to update the v's record $(v, F_1^v, \ldots, F_{|\mathcal{A}|}^v, cid)$ in

GenRecs as follows.

$$F_i^v = \begin{cases} F_i^v + 1 & \text{if } A_i \in \Delta C \\ F_i^v & \text{if } A_i \notin \Delta C \end{cases} \quad (2)$$

We increase F_i^v for each $A_i \in \Delta C$ by one because v has been added to the column of each $A_i \in \Delta C$.

For the case $C' \neq C$, Algorithm 4 has to find out i) whether or not C' already exists in *ACRecs*, and ii) whether or not C is to be deleted from *ACRecs*. If C' does not exist in *ACRecs*, then Algorithm 4 adds C' to *ACRecs* (Lines 7-8). After that Algorithm 4 has to take care of *CountCache*. In the case in which there is no value count for C', Algorithm 4 creates a new value count for C' and initializes it with one. In the other case it increases the value count by one. Notice that a value count for a cluster exists in *CountCache* only if a value belonging to this cluster has been inserted before.

Because the cluster of v has become C', the records of v in *GenRecs* and in *ValCache* have to be updated. The updating of v's record $r_v = (v, F_1^v, \ldots, F_{|\mathcal{A}|}^v, cid)$ consists of replacing identifier cid by the identifier cid' of the new cluster of v and of applying the Formula 2. The updating of the record (v, cid) in *ValCache* consists of only replacing cid by cid' (Lines 13-14).

Now the value v does not belong to C any more. Therefore, Algorithm 4 decreases the number of values belonging to the cluster C by one (Line 15) and calls Algorithm 5 which has to decide whether or not C has to be deleted.

Deleting a cluster. When a value has been deleted or assigned to a different cluster, we have to check whether or not its previous cluster has to be deleted from *ACRecs*. A cluster has to be deleted if the number of values belonging to it is zero. Algorithm 5 performs this check. If the cluster (i.e. its identifier) does not exist in *CountCache* or the associated number of values in *CountCache* is zero, then Algorithm 5 retrieves the number of values from *GenRecs* (Lines 2-7). If this number is zero, then the cluster will be removed from *ACRecs*, and will also be removed from *CountCache* in the case of its existence in *CountCache* (Lines 8-11).

In the case that the values' number retrieved from *GenRecs* is greater than zero and the cluster does not exist in *CountCache*, Algorithm 5 creates a value count for the cluster in *CountCache* and initializes it with the retrieved number (Lines 12-14). By doing so, Algorithm 5 will reduce the querying *GenRecs* in the future. Accessing and querying *ValRecs* is more expensive than querying *CountCache* because *ValRecs* is an external data structure in the form of relational table.

5.3 Handling delete

Algorithm 6 implements the extract operator defined in Definition 4.8, handling deleting a value v from the columns of the attributes in ΔC. Notice that ΔC can only be a subset of the current cluster C of v because C is the maximum set of attributes whose columns contain v (see Definition 4.1). Based on Definition 4.8 and Lemma 4.9, we obtain the new cluster C' of v from the current cluster C after removing some attributes in ΔC from C (Lines 1-7). Each attribute $A_i \in \Delta C$ is to be removed from C if the value v occurs only once in the column of A_i (i.e. $F_i^v = 1$).

If the deleted value v occurs more than once in the column of each $A_i \in \Delta C$ (i.e. $\forall A_i \in \Delta C : F_i^v > 1$), then the new cluster C' is identical to the current cluster C. In this case Algorithm 6 needs only to update the value record in *GenRecs* as follows (Lines 8-9).

Algorithm 6: handleDelete

```
   Input  : v, ΔC
1  cid ← getClusterID(v)
2  C ← ACRecs·getCluster(cid)
3  C' ← C
4  {F_i^v | A_i ∈ ΔC} ← GenRecs·getFreqs(v, ΔC)
5  for F_i^v ∈ {F_i^v | A_i ∈ ΔC} do
6      if F_i^v ≤ 1 then
7          C' ← C' \ {A_i}
8  if C = C' then
9      GenRecs·dUpdate(v, ΔC)
10 else
11     if C' = ∅ then
12         GenRecs·remove(v)
13         ValCache·remove(v)
14     else
15         cid' ← ACRecs·getCID(C')
16         if cid' = null then
17             cid' ← ACRecs·add(C')
18             CountCache·init(cid', 1)
19         else
20             CountCache·increase(cid')
21         ValCache·update(v, cid')
22         GenRecs·dUpdate(v, cid', ΔC)
23     CountCache·decrease(cid)
24     handleDeletingCluster(cid)
```

$$F_i^v = \begin{cases} F_i^v - 1 & \text{if } A_i \in \Delta C \\ F_i^v & \text{if } A_i \notin \Delta C \end{cases} \quad (3)$$

Algorithm 6 decreases F_i^v for each $A_i \in \Delta C$ by one because v has been deleted only once from the column of each $A_i \in \Delta C$.

If the new cluster C' is different from C, then the value v does not belong to C anymore. Therefore, Algorithm 6 reduces the number of values belonging to C (Line 23), and then calls Algorithm 5 (see Subsection 5.2) to decide whether or not C has to be deleted from *ACRecs* (Line 24).

The case in which the new cluster C' is empty occurs if $C = \Delta C$ and $F_i^v = 1$ for each $A_i \in \Delta C$. This case means that the value v has been completely deleted from the dataset $\mathcal{D}$. Therefore, Algorithm 6 deletes the value v from *GenRecs* and from *ValCache* (Lines 12-13).

If the new cluster C' is not empty and different from C, then Algorithm 6 has to know whether or not C' already exists in *ACRecs*. For the case that C' does not exist in *ACRecs*, Algorithm 6 adds it to *ACRecs*, creates a new value count for C' in *CountCache*, and initializes this counter with one (Lines 16-18). For the other case in which C' already exists, Algorithm 6 only increases the number of values belonging to C' (Line 20).

The last step Algorithm 6 has to execute for the case $C' \neq \emptyset$ and $C' \neq C$ is to update the records of the value v in *GenRecs* and

in *ValCache*, respectively (Lines 21-22). Updating the v' record in *ValCache* consists of only replacing the previous cluster identifier cid by cid'. While Updating the record $r_v = (v, F_1^v, \ldots, F_{|\mathcal{A}|}^v, cid)$ of v consists of decreasing F_i^v for each $A_i \in \Delta\mathcal{C}$ by one (see Formula 3), and replacing the previous cluster identifier cid by cid'.

5.4 Performance analysis

As the data structure of the generator is an external data structure, the performance of our algorithms depends mainly on querying and updating this data structure. Therefore, we analyze the performance of the algorithms in terms of the number of accesses needed for querying and updating *GenRecs* (i.e., the generator).

Identifying the cluster of an input value. To find the identifier of the cluster to which an input value v belongs, Algorithm 2 needs one query to retrieve the identifier from *GenRecs* (Line 3) if *ValCache* does not contain a record for v. Otherwise, Algorithm 2 does not need any access to *GenRecs*. Thus, in the worst case we need one generator access for identifying the cluster of a value.

Deleting a cluster. Algorithm 5 queries *GenRecs* to identify the number of values belonging to a cluster whose deletion comes into consideration (Line 5 or 7). This query is required if the value count in *CountCache* is zero or if *CountCache* does not contain an entry for the cluster. Thus, in the worst case we need one access to *GenRecs* to know whether or not a cluster is to be deleted.

Handling an insert. If the input value v is new, Algorithm 3 needs one generator access for inserting a new value record r_v for v into *GenRecs* (Line 9). If the input value v is not new, Algorithm 4 also needs one generator access to update the value record r_v in *GenRecs* (Line 4 or 13). Thus, in the best case Algorithm 1 needs one generator access for updating the attribute clustering, while it needs three accesses in the worst case.

Handling a delete. Algorithm 6 needs one access to *GenRecs* to compute the frequency of occurrences of the input value in each attribute from whose column the value has been deleted (Line 4). Furthermore, Algorithm 6 needs to access *GenRecs* either to remove the value record r_v from *GenRecs* or to update it (Line 12 or 22). Therefore, for handling the delete of a value we need two generator accesses to *GenRecs*. Thus, in the best case Algorithm 6 needs two accesses to external data structure for updating the attribute clustering after deleting a value from the dataset, while it needs four accesses to *GenRecs* in the worst case.

We now formulate the results of the previous analysis as follows.

Lemma 5.1. *Let $\Delta\mathcal{V}$ be the set of distinct values occurring in a tuple t inserted into $\mathcal{D}$. Updating the attribute clustering after inserting t needs $|\Delta\mathcal{V}|$ generator accesses in the best case, and $3\times|\Delta\mathcal{V}|$ generator accesses in the worst case.*

Lemma 5.2. *Let $\Delta\mathcal{V}$ be the set of distinct values occurring in a tuple t deleted from $\mathcal{D}$. Updating the attribute clustering after deleting t needs $2 \times |\Delta\mathcal{V}|$ generator accesses in the best case, and $4 \times |\Delta\mathcal{V}|$ generator accesses in the worst case.*

This means that in both cases, in the best case and in the worst case, updating the generator after deleting a tuple t needs $|\Delta\mathcal{V}|$ more database accesses than updating it after inserting t, where $\Delta\mathcal{V}$ is the set of distinct values occurring in t. Notice that updating the generator without the cache strategies always requires $3\times|\Delta\mathcal{V}|$ accesses after an insertion, and $4 \times |\Delta\mathcal{V}|$ after a deletion.

For every input value $v \in \Delta\mathcal{V}$, an access to the generator is either (i) a modification of v's record, (ii) removing the v's record from the generator, or (iii) inserting the v's record into the generator. Removing a record from the generator and inserting a record may require more runtime than the modification of a record because the former operation does not cause any reorganization either of the index defined on the values or of the index defined on the identifiers of the clusters, while the latter two operations may cause such kind of reorganizing these two indexes.

Furthermore, the cost of updating the generator depends on two variables: the number of its records and the number of attributes, which means that the performance of updating the generator increases if the number of distinct values grows or the number of attributes grows.

6 EXPERIMENTAL EVALUATION

We now evaluate our system in terms of Definition 2.2, which means we experimentally investigate the runtime T_{inc} needed for updating the data structures after inserting a tuple into the corresponding dataset and after deleting a tuple from it. In this section, we often refer to the expression "updating the data structures of the system" as "updating the attribute clustering". In particular, we carried out this evaluation to answer the following questions: What is the average runtime T_{inc} for updating the attribute clustering of different large datasets, how effective are the cache strategies of the system, how does the incremental runtime T_{inc} change in relation to an increase in the number of attributes, and most importantly, can we ignore the incremental runtime towards the runtime required by the static discovery of uINDs?

The function of the quantities F_A^v ($A \in \mathcal{A}$ and $v \in \mathcal{V}$) in the data structure of the generator is to handle the deletion. Thus, we do not need these quantities if we limit our system to support only the insert. Therefore, the following question arises: how does the updating time for the insert change if we limit our system to support only the insert? To answer this question we have implemented an extra version of Algorithm 1, where the generator does not contain the frequency of occurrences F_A^v of v in the column of A. We refer to this version of the implementation as `only-insert`.

6.1 Setup

Experimental conditions. We performed the experiments on a Windows 7 Enterprise system with an Intel Core i5-3470 (Quad Core, 3.20 GHz CPU) and 8 GB RAM. We installed Oracle 11g on the same machine as the database server and used an external disk for the storage of all used datasets and for the storage of the generators.

Datasets. Table 3 shows some characteristics of the datasets used in our experiments. The first column states the name of the dataset. The second column gives the number of relations in the corresponding dataset. The total number of attributes in each dataset is given in the third column. The fourth column states the total number of rows in each dataset. The number of distinct values in each dataset is given in the fifth column. The number of clusters in the attribute clustering in each dataset is stated in the sixth column. The last column gives the number of valid uINDs in each dataset.

Table 3: Characteristics of datasets used in the experiments

$\mathcal{D}$	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{A}\rvert$	$\sum_{r_i \in \mathcal{D}} \lvert r_i \rvert$	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{AC}\rvert$	$\lvert\mathcal{I}\rvert$
TPC-H	8	61	8 661 245	11 807 306	126	80
MB1	45	273	10 000 000	10 382 340	663 584	1844
MB2	18	100	24 000 000	20 552 799	294 059	178
Plista	4	140	33 364 151	46 882 120	185	408
H-Genome	43	387	116 227 014	72 559 365	287 738	4976

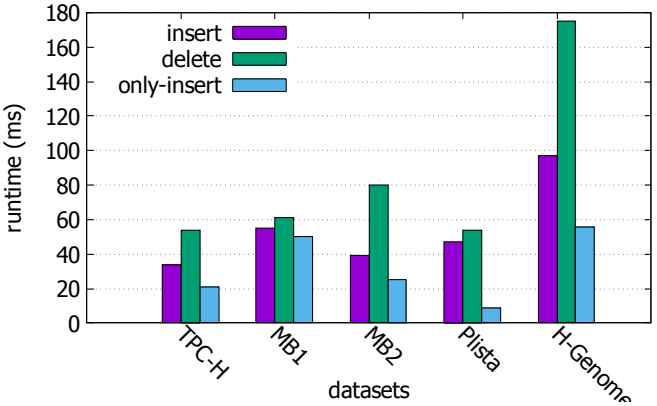

Figure 3: Comparing the runtime for an insert with the runtime for a delete

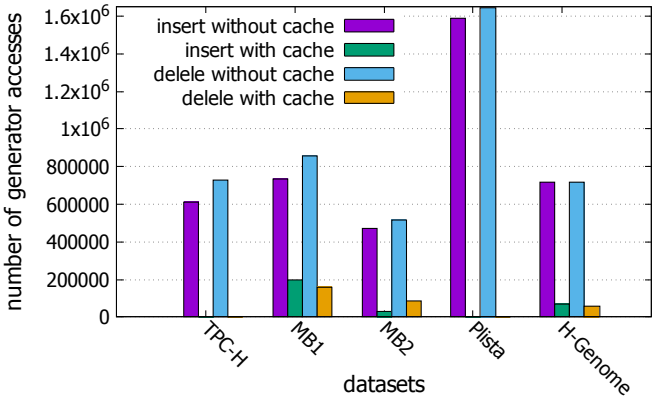

Figure 4: Comparing the number of generator accesses in two cases: using the cache strategies and not using the cache strategies.

TPC-H is a benchmark dataset available at http://tpc.org/tpch. All other datasets are real-world datasets. Both MB1 and MB2 are parts of MusicBrainz dataset available at https://musicbrainz.org. MusicBrainz is an open music encyclopedia that collects music metadata and makes them available to the public. Plista [10] contains anonymized web-log data provided by the advertisement company Plista. H-Genome is a genome dataset of homo sapiens available at http://ensembl.org.

Systems. We compare our system for incrementally discovering uINDs against the algorithm presented in [21], which is a scalable algorithm for discovering uINDs in large, static datasets. We refer to this algorithms as S-indd. We also implemented extra procedures for initializing the clusters and the generator of each dataset in Table 3. These procedures are based on an adaption S-indd. They are not necessarily a part of our system because the system can begin with empty datasets, which will constantly grow and shrink. We implemented all algorithms in 64-bit Java 7.

6.2 Varying the datasets

To estimate the time T_{inc} needed for updating the attribute clustering of different datasets, we designed the experiments as follows.

Design of experiments. From each dataset $\mathcal{D}$ in Table 3, we randomly selected two different sets of tuples $\mathcal{D}_{ins}$ and $\mathcal{D}_{del}$. Each selected set consists of approximately 100 000 tuples. The set $\mathcal{D}_{ins}$ is to conduct experiments for inserts, while $\mathcal{D}_{del}$ is to conduct experiments for deletions. Then we removed all tuples of $\mathcal{D}_{ins}$ from the original dataset $\mathcal{D}$ to reinsert them again in a later step. After that we created and initialized the data structures *ACRecs* and *GenRecs* for $\mathcal{D} \setminus \mathcal{D}_{ins}$. For initialization of the data structures we implemented a special procedure based on adaptation of the algorithm presented in [21]. For each dataset, the statistics presented in Table 3 are calculated after selecting $\mathcal{D}_{ins}$ and $\mathcal{D}_{del}$.

To estimate the time for updating the attribute clustering of $\mathcal{D}$ after inserting a tuple, we insert all the tuples of $\mathcal{D}_{ins}$ again into $\mathcal{D}$. After each insertion we updated the data structures. For each update we recorded the needed time, took the average of all runtimes. We repeated this procedure for the tuples $\mathcal{D}_{del}$, but instead of insertions, we removed all tuples of $\mathcal{D}_{del}$ from $\mathcal{D}$.

For the only-insert version we recreated and reinitialized the *GenRecs* without the quantities F_A^v for each dataset $\mathcal{D}$, and used the same set $\mathcal{D}_{ins}$ selected for the regular version.

Figure 3 shows the results of these experiments. In this figure there is a group of three bars for each dataset. In each group, the left bar presents the average time required for updating the attribute clustering after inserting a tuple, while the middle bar presents the average time needed for updating the attribute clustering after a deletion. The right bar in each group shows the average time needed for updating the attribute clustering after an insertion in the only-insert version.

Evaluation of the runtime. For each dataset, the average runtime for updating the attribute clustering after an insertion is shorter than the average runtime for updating the attribute clustering after a deletion. The main reason is that the update after a deletion always needs one more access to the external data structure *GenRecs* than the update after an insertion as discussed in Subsection 5.4. Furthermore, it is clear that the runtime for updating after an insertion in the only-insert version is always less than the runtime for

Table 4: The reduction of generator accesses gained by cache strategies

$\mathcal{D}$	reduction by insert(%)	reduction by delete(%)
TPC-H	99.987	99.984
MB1	73.440	81.213
MB2	93.177	83.896
Plista	99.997	99.995
H-Genome	90.076	91.811

Table 5: Comparing the runtime (in seconds) with the runtime of S-INDD [21] applied to the entire dataset

$\mathcal{D}$	S-INDD [21]	only-insert	insert	delete
TPC-H	424	0.021	0.034	0.054
MB1	176	0.050	0.055	0.061
MB2	484	0.025	0.039	0.080
Plista	13 580	0.009	0.047	0.054
H-Genome	3135	0.056	0.097	0.175

updating after an insertion in the regular version. The reason is that updating *GenRecs* in the only-insert version is faster than updating it in the regular version because *GenRecs* in the only-insert version does not contain the quantities F_A^v.

The cost of updating the data structure of the generator depends on two variables: the umber of distinct values $|\mathcal{V}|$ and the number of attributes $|\mathcal{A}|$, which means that the runtime for updating the generator increases if the number of distinct values or the number of attribute increase. This fact explains why the three bars of the dataset H-Genome are longer the corresponding bars of the other datasets, respectively (see also Table 3).

Evaluation of cache strategies. To evaluate the effectiveness of our cache strategies, we compare the number of generator accesses in the case of using both data structures *CountCache* and *ValCache* with the required number of generator accesses without using them. In both cases, we counted the number of accesses needed for inserting the set $\mathcal{D}_{ins}$ and the number of accesses needed for deleting the $\mathcal{D}_{del}$ of each dataset in Table 3. The results are presented in Figure 4. In this figure, there is a group of four bars for each dataset. Form left to right in each group, the first bar presents the number of accesses for inserting the $\mathcal{D}_{ins}$ without caches, while the second bar is for the same inserts but with caches. The third bar presents the number of accesses for the deleting the $\mathcal{D}_{del}$ without caches, while the far right bar is for the same deletions but with caches. Table 4 shows the percentage of reducing the number of accessing the generator in the case of using the caches.

Overall, we reduced the number of generator accesses by more than 73 % for the insert, and by more than 81 % for the delete. For TPC-H and Plista, the reduction is more than 99 % both in the insertion and in the deletion. This high reduction explains why the bars presenting the number of accesses in the case of using the caches do not appear in Figure 4 for both TPC-H and Plista.

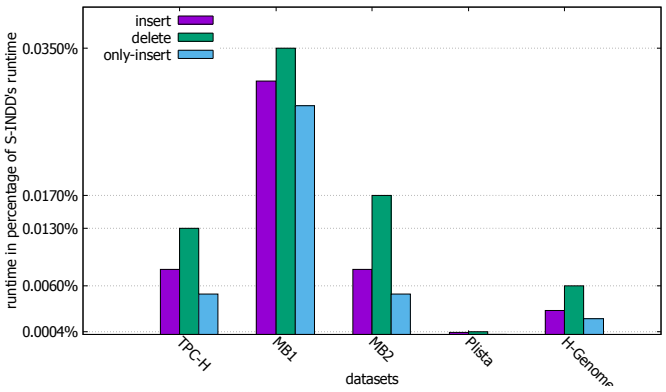

Figure 5: Runtime in percentage of S-INDD [21] runtime

6.3 Comparing with the static discovery

Comparing with S-INDD [21]. We now compare the incremental discovery with the static discovery of uINDs in terms of Definition 2.2. For this purpose we compare each runtime calculated in the previous subsection for each dataset in Table 3 with the corresponding runtime required by S-INDD [21]. These comparisons are presented in Table 5. As we can observe, the runtime needed by the incremental update of the attribute clustering is much smaller than the runtime needed by S-INDD. This observation is valid for all datasets. For instance, after inserting a tuple t into H-Genome S-INDD needs $T_{\text{S-INDD}}(\text{H-Genome} + \{t\}) = 3135$ seconds, while the incremental approach needed only $T_{inc}(t) = 0.097$ seconds (i.e., the incremental approach needed ca. 0.003 % of the time needed by S-INDD). Thus, we can neglect $T_{inc}(t) = 0.097$ seconds towards $T_{\text{S-INDD}}(\text{H-Genome} + \{t\}) = 3135$ seconds. For updating the attribute clustering of Plista after a deletion, the incremental approach requires only 0.0004 % of the runtime required by S-INDD (i.e., the reduction in the static runtime is here more than 99.9996 %). In fact, we can neglect each incremental runtime listed in Table 5 towards the corresponding statical runtime, which means that our system satisfies the requirement formulated in Definition 2.2 for incremental discovery of uINDs.

Figure 5 presents the incremental runtime in percentage of the corresponding static runtime. This presentation allows us to make an important observation, namely that there is a tendency for the percentages of larger datasets (Plista and H-Genome) to be smaller than the percentages of the smaller datasets, which means that by incrementally updating the uINDs, we avoid the performance suffering from static rediscovery after a simple change in the corresponding dataset.

Comparing with other static approaches. Other static approaches either explicitly [15] or implicitly [2, 19] derive the set of uINDs from the inverted index. [21] shows that the approaches presented in [2, 15] are special cases of S-INDD. The approach presented in [19] exports the values of every attribute in the dataset and divides them into a fixed number of buckets by hashing these values (i.e., it computes a set of buckets for every attribute). During the dividing process, it has to manage storing the buckets on the hard disk. After dividing the value sets of all attributes into buckets, the approach in [19] iteratively takes a set of buckets with the

Table 6: Comparing the number of intersections needed by inverted index (#i-idx-∩) with number of intersections needed by attribute clustering (#$\mathcal{AC}$-∩)

$\mathcal{D}$	#i-idx-∩	#$\mathcal{AC}$-∩	reduction in %
TPC-H	4.39×10^{10}	4.69×10^{5}	99.99
MB1	7.74×10^{11}	4.95×10^{10}	93.67
MB2	2.06×10^{11}	2.94×10^{9}	98.57
Plista	1.04×10^{12}	4.11×10^{6}	99.99
H-Genome	1.09×10^{13}	4.31×10^{10}	99.61

following properties: (i) the number of buckets equals the number of attributes, (ii) every bucket corresponds to only one attribute, and (iii) all values in these buckets have the same hash value, and then generates the inverted index from the selected set of buckets. Based on this inverted index it computes a subset of uINDs, and repeats the selecting process until all buckets are processed and the set of uINDs completely computed. Based on the experiments conducted by [19] on the dataset Plista (see [19]) we can conclude from Figure 7 in [19] that the runtime for discovering all uINDs in Plista is approximately 4983 seconds. Thus, we can neglect the incremental runtime required for updating the attribute clustering of Plista (see Table 5) towards the statical runtime needed by [19].

The effectiveness of the attribute clustering. To show the effectiveness of the attribute clustering concept, we now compare the number of intersection operations needed for deriving the valid uINDs from the clusters with the number of intersection operations needed for deriving them from the inverted index. Table 6 presents the results of these comparisons. The last column in this table shows the percentage of the reduction in the number of intersections gained by deriving the uINDs from the attribute clustering. As we can see, the gained reduction is more than 93% for all datasets and up to 99.99% for TPC-H and Plista.

Furthermore, deriving the set $\mathcal{I}$ from the clusters based on Lemma 4.2 has not required more than one second for every dataset listed in Table 3.

6.4 Scaling the number of attributes

We now evaluate the change of the runtime in relation to the increase in the number of attributes. For this purpose we designed the experiments in this case as follows.

Design of experiments. For these experiments, we created six datasets $\mathcal{D}^i$ $(1 \leq i \leq 6)$ from the dataset MusicBrainz, with the properties: (i) $\mathcal{D}^6$ = MB1, (ii) $\mathcal{D}^1$ has 15 attributes, (iii) the dataset $\mathcal{D}^{i+1}$ contains all relations of $\mathcal{D}^i$ and additional relations from MusicBrainz so that $\mathcal{D}^{i+1}$ has approximately 50 attributes more than $\mathcal{D}^i$ $(1 \leq i \leq 5)$, and (iv) each $\mathcal{D}^i$ $(1 \leq i \leq 6)$ has approximately 10,000,000 tuples. For each dataset $\mathcal{D}^i$ we estimated the runtime for updating the attribute clustering by applying the same process applied in Subsection 6.2. Figure 6 shows the results of these experiments.

Evaluation. Looking at this figure we can observe that there is a tendency for the incremental runtime to increase when increasing the number of attributes because (i) the size of the generator grows when the number of the attributes grows, and (ii) the runtime for the modification of the generator correlates with its size. The

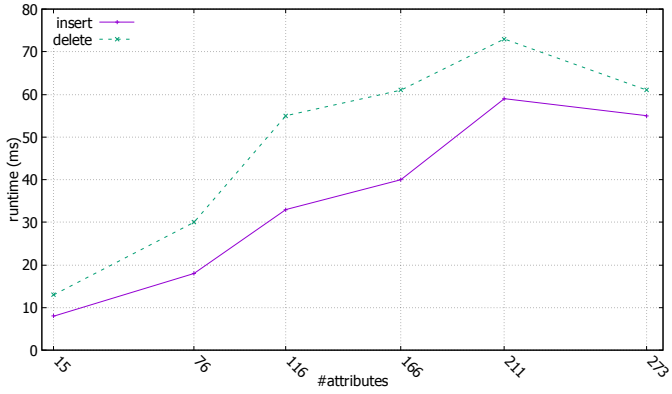

Figure 6: Scaling the number of attributes and fixing the number of rows to 10,000,000

reason for the correlation is that the time of creating and executing the SQL-Statements required for updating the generator generally increases when the number of the attribute increases. However, the incremental runtime declines slightly when the number of attributes increases from 211 to 273. This deviation from the general tendency may be because (i) the number of generator accesses for the dataset $\mathcal{D}^5$ (i.e. the dataset with 211 attributes) is higher than the number of generator accesses for $\mathcal{D}^6$ (i.e. the dataset with 273 attributes), and (ii) up to certain degree, the number of generator accesses has stronger influence on the runtime than the influence of the number of attributes on the runtime. In fact, the number of accesses for $\mathcal{D}^5$ is 199,808 for the insertions and 174,786 accesses for the deletions, while for $\mathcal{D}^6$ it is 195,340 for the insertions and 161,144 accesses for the deletions.

As we can see, the maximum incremental runtime for the insert is 0.059 seconds, while the maximum incremental runtime for the deletion is 0.074 seconds, and the statical runtime for discovering uINDs in $\mathcal{D}^6$ = MB1 is 176 seconds (see Table 5). Thus, we can ignore both incremental runtimes towards the statical runtime.

7 RELATED WORK

Kantola et al. [9] 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 [12] investigated INDs and NOT NULL constraints under simple and partial semantics from theoretical point of view.

Shaabani and Meinel developed S-indd [21], 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 [2, 15, 19]. Furthermore, Shaabani and Meinel have shown that Spider presented by Bauckmann et al. [2] is a special case of S-indd. Spider [2] 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] which applies a divide and conquer technique for discovering unary INDs.

BINDER takes a further step to generate all *n*-ary INDs by applying string concatenations and the same Apriori strategy applied by MIND [15]. This approach results in an exponential number of I/O-operations and exponentially increases the original data size.

For discovering n-ary INDs (n > 1), MIND [15], FIND$_2$ [11], and ZIGZAG [14] apply the principle of candidate generation, which is the basic idea of Apriori algorithm for discovering frequent item sets. They start by computing all valid unary INDs satisfied by the input dataset, and then inductively continue to larger arities by selecting sets of new candidate INDs to be tested against the dataset. But the difference between MIND on one side, and FIND$_2$ and ZIGZAG on the other side is that MIND is a straightforward application of the Apriori strategy, while FIND$_2$ and ZIGZAG transform the IND discovery problem into a discovery problem in a hypergraph. FIND$_2$ maps the IND discovery problem to the hyperclique discovery problem while ZIGZAG maps it to the minimal traversal discovery problem. A common problem with the principle of candidate generation is poor scalability in the length of the longest valid IND in the dataset since validation of an n-ary IND might require first checking all (exponentially many in n) INDs implied by it. Shaabani and Meinel developed MIND$_2$, the first approach for detecting n-ary INDs without any candidate generation. MIND$_2$ characterizes the maximum INDs by set operations defined on certain metadata, which MIND$_2$ generates by accessing the database only 2 × the number of valid unary INDs.

Zhang et al. [25] applied approximation techniques for discovering foreign keys. Memari et al. [16] extended the techniques presented in [25] to handle different semantics for NULL markers of the SQL Standard.

Research has been done for detecting functional dependencies and unique column combinations in dynamic manner: Wang et al. [24] present an approach for maintaining discovered functional dependencies after data deletions. Cong et al. [4] suggest an approach for incremental data repairing with respect to functional dependencies and conditional functional dependencies. Fan et al. [6] present approaches for incremental detection of functional dependencies and conditional functional dependencies violation in distributed database. Abedjan et al. [1] present an approach to efficiently discover unique and non-unique column combinations on dynamic datasets.

It is worth noting that most commercial relational DBMS allow users to specify a set of inclusion dependencies in terms of foreign key constraints between relations. The DBMS validates all user-defined foreign key constraints after an insert, a delete and a change of a tuple and aborts the operation in the case it does not satisfy one of these constraints. However, the DBMS con not find new inclusion dependencies after inserting new tuples.

8 CONCLUSION

In this work we developed the first approach for incrementally discovering unary inclusion dependencies in continuously changing data. We reduced the problem of the incremental update of unary INDs to the incremental update of the attribute clustering, from which unary inclusion dependencies are efficiently derivable. We solved the problem of incrementally updating the attribute clustering by developing new cluster operations to be applied after every data change. Then we designed algorithms and data structures for efficient implementation of the cluster operations. The main goal of the incremental discovery is to avoid reprocessing the entire dataset by applying the static discovery after every change, which requires a long computation time. In this regard, we performed a comprehensive experimental evaluation showing that the computation time of the incremental discovery is negligible in comparison to computation time of the static discovery. In fact, the reduction in computation time is up to 99.9996 %.

The generator can be implemented in a different way than presented in this paper (e.g., in form of distributed hash table). Therefore, we work on compression strategies for reducing the size of its data structure. The reason is to increase the caching capacities and to reduce the number of generator accesses.

REFERENCES

[1] Z. Abedjan, J. A. Quian-Ruiz, and F. Naumann. 2014. Detecting unique column combinations on dynamic data. In *ICDE 2014*. 1036–1047.

[2] J. Bauckmann, U. Leser, and F. Naumann. Efficiently Computing Inclusion Dependencies for Schema Discovery. In *ICDE Workshops, 2006*.

[3] M. A. Casanova, L. Tucherman, and A. L. Furtado. 1988. Enforcing Inclusion Dependencies and Referencial Integrity. In *VLDB 1988 (VLDB '88)*.

[4] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. 2007. Improving Data Quality: Consistency and Accuracy. In *VLDB (VLDB '07)*. VLDB Endowment, 315–326.

[5] W. Fan. 2008. Dependencies Revisited for Improving Data Quality. In *PODS (PODS '08)*. ACM, New York, NY, USA, 159–170.

[6] W. Fan, J. Li, N. Tang, and W. Y. qa. 2014. Incremental Detection of Inconsistencies in Distributed Data. *TKDE* 26, 6 (June 2014), 1367–1383.

[7] A. Gruenheid, X. L. Dong, and D. Srivastava. 2014. Incremental Record Linkage. *VLDB* 7, 9 (May 2014), 697–708.

[8] J. Gryz. 1998. Query Folding with Inclusion Dependencies. In *ICDE 1998*. 126–133.

[9] M. Kantola, H. Mannila, K.-J. Räihä, and H. Siirtola. 1992. Discovering functional and inclusion dependencies in relational databases. *JIIS* 7, 7 (1992).

[10] B. Kille, F. Hopfgartner, T. Brodt, and T. Heintz. 2013. The Plista Dataset. In *NRS Workshops (NRS '13)*. 16–23.

[11] A. Koeller and E.A. Rundensteiner. 2003. Discovery of high-dimensional inclusion dependencies. In *ICDE 2003*. 683–685.

[12] H. Köhler and S. Link. 2015. Inclusion Dependencies Reloaded. In *CIKM (CIKM '15)*. 1361–1370.

[13] M. Levene and M. W. Vincent. 2000. Justification for inclusion dependency normal form. *TKDE* 12 (2000), 2000.

[14] F. D. Marchi and J.-M. Petit . Zigzag: A New Algorithm for Mining Large Inclusion Dependencies in Databases. In *ICDM, 2003*.

[15] F. D. Marchi, S. Lopes, and J.-M. Petit. 2009. Unary and n-ary inclusion dependency discovery in relational databases. *JIIS* 32, 1 (2009), 53–73.

[16] M. Memari, S. Link, and G. Dobbie. 2015. *Conceptual Modeling: ER 2015*. Springer International Publishing, Cham, Chapter SQL Data Profiling of Foreign Keys, 229–243.

[17] R. J. Miller, M. A. Hernández, L. M. Haas, L.-L. Yan, C. T. Howard Ho, R. Fagin, and L. Popa. 2001. The Clio Project: Managing Heterogeneity. *SIGMOD Rec., 2001* 30, 1 (2001), 78–83.

[18] F. Naumann. 2014. Data Profiling Revisited. *SIGMOD Rec.* 42, 4 (2014), 40–49.

[19] T. Papenbrock, S. Kruse, J.-A. Quiane-Ruiz, and F. Naumann. 2015. Divide & Conquer-based Inclusion Dependency Discovery. *VLDB, 2015* 8, 7 (0 2015), 774–785.

[20] B. Saha and D. Srivastava. 2014. Data quality: The other face of Big Data. In *ICDE 2014*. 1294–1297.

[21] N. Shaabani and C. Meinel. 2015. Scalable Inclusion Dependency Discovery. In *Database Systems for Advanced Applications: DASFAA 215*. LNCS, Vol. 9049.

[22] N. Shaabani and C. Meinel. 2016. Detecting Maximum Inclusion Dependencies without Candidate Generation. In *Database and Expert Systems Applications: DEXA 2016, Part II*. Springer International Publishing, Cham, 118–133.

[23] K. Smith, L. Seligman, A. Rosenthal, C. Kurcz, M. Greer, C. Macheret, M. Sexton, and A. Eckstein. 2014. "Big Metadata": The Need for Principled Metadata Management in Big Data Ecosystems. In *Proceedings of Workshop on Data Analytics in the Cloud (DanaC'14)*. Article 13, 4 pages.

[24] S.-L. Wang, W.-Ch. Tsou, J.-H. Lin, and T.-P. Hong. 2003. *Maintenance of Discovered Functional Dependencies: Incremental Deletion*. Springer Berlin Heidelberg, Berlin, Heidelberg, 579–588.

[25] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C M. Procopiuc, and D. Srivastava. 2010. On Multi-column Foreign Key Discovery. *VLDB* 3, 1-2 (sep 2010), 805–814.