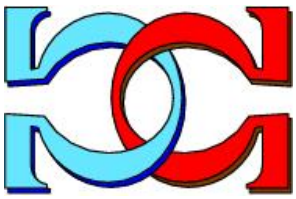


CDMTCS
Research
Report
Series

Discovery Algorithms for Embedded Uniqueness Constraints

Ziheng Wei
The University of Auckland

Uwe Leck
The University of Flensburg

Sebastian Link
The University of Auckland

CDMTCS-524
March 2018

Centre for Discrete Mathematics and
Theoretical Computer Science

Discovery Algorithms for Embedded Uniqueness Constraints

Ziheng Wei
The University of Auckland, New Zealand
z.wei@auchlanduni.ac.nz

Uwe Leck
The University of Flensburg, Germany
uwe.leck@uni-flensburg.de

Sebastian Link
The University of Auckland, New Zealand
s.link@auckland.ac.nz

March 15, 2018

Abstract

Data profiling is an enabler for efficient data management and effective analytics. The discovery of data dependencies is at the core of data profiling. We conduct the first study on the discovery of embedded uniqueness constraints (eUCs), a recently introduced class of data dependencies that represent unique column combinations embedded in complete fragments of incomplete data. We show that the decision variant of finding a minimal eUC is NP-complete and W[2]-complete in the input size. We also characterize the maximum possible solution size, and show which families of eUCs attain that size. The size is much larger than for the special case of minimal SQL uniques. Despite these challenges, our column-efficient, row-efficient, and hybrid discovery algorithms perform effectively and fast on real-world benchmark and synthetic data. We also propose the computation of small semantic samples of given data sets as a new direction in data profiling. These samples satisfy the same eUCs as the given data set and we showcase how discovery and sampling together provide a pathway towards effective data cleansing and business rule acquisition.

Keywords: Armstrong relation; Data cleaning; Data profiling; Discovery; Extremal combinatorics; Incomplete data; Sampling; Uniqueness; SQL

id	*voter_id*	*name_prefix*	*first_name*	*middle_name*	*last_name*	*address*	*city*	*phone_num*	*register_date*
t_0	702	⊥	nell	mrs	marshall	719 carter st	kernersville	⊥	5/11/1940
t_1	833	⊥	nell	⊥	marshall	1731 tredegar rd	kernersville	336 992 7811	5/11/1940
t_2	131	⊥	joseph	t	cox	9 casey rd	new bern	252 000 0000	3/06/1935
t_3	131	⊥	joseph	thomas	cox	1108 highland ave #22	new bern	252 288 4763	3/06/1935
t_4	320	⊥	robert	f	boone	213 s cumberland st	wallace	⊥	1/01/1940
t_5	720	⊥	robert	edward	boone	124 rolling rd	burlington	228 8872	5/11/1940
t_6	962	⊥	margaret	plonk	isley	1880 brookwood ave #102	burlington	336 226 3774	10/26/1940
t_7	937	⊥	margaret	marie	harper	7572 bullard rd	clemmons	⊥	10/26/1940
t_8	247	⊥	herbert	⊥	futrell	9802 us hwy 258	murfreesboro	252 398 3716	10/21/1938
t_9	244	⊥	sallie	b	futrell	9802 us hwy 258	murfreesboro	252 398 3716	10/21/1938

Table 1: Snippet of the NCVoter Data Set

1 Introduction

Keys provide efficient access to data in database systems. They are required to understand the structure and semantics of data. For a given collection of entities, a key refers to a set of column names whose values uniquely identify an entity in the collection. For example, a key for a relational table is a set of columns such that no two different rows have matching values in each of the key columns. Keys advance many classical areas of data management such as data modeling, database design, and query optimization. Knowledge about keys empowers us to 1) uniquely reference entities across data repositories, 2) reduce data redundancy at schema design time to process updates efficiently at run time, 3) improve selectivity estimates in query processing, 4) feed new access paths to query optimizers that can speed up the evaluation of queries, 5) access data more efficiently via physical optimization such as data partitioning or the creation of indexes and views, and 6) gain new insight into application data. Modern applications create even more demand for keys. Here, keys facilitate the data integration process, help detect duplicates and anomalies, guide the repair of data, and return consistent answers to queries over dirty data. The discovery of keys is a fundamental task in data profiling.

Recent years have seen tremendous progress on the discovery of keys [2, 14, 30] despite its computational difficulty. For example, on a data set with 50 columns, up to $126,410,606,437,752$ minimal keys may exist. Indeed, deciding if some key with at most n attributes holds on a given data set is not only NP-complete, but even W[2]-complete in the size of the key. That is, the problem is likely to be intractable, even when the size of the key is fixed [5]. It is remarkable that algorithms exist that can find all minimal keys for reasonably large numbers of rows or columns [29].

Incompleteness and inconsistency impose persistently hard challenges to the discovery of data dependencies. In practice, it is generally impossible to identify all entities uniquely. Over incomplete data, the default of discovery algorithms is to treat occurrences of the null marker ⊥ just like ordinary domain values. That is, null is considered to be equal to null [29, 30]. This leads to outputs with a questionable semantics, since the only interpretation of ⊥ where this makes sense is when a value does not exist. Worse, any fixed interpretation of ⊥ is questionable, in particular in data originating from various sources. The case where null is considered to be different from null renders the validity of a key constraint independent of null marker interpretations. While SQL evaluates comparisons of ⊥ as *unknown*, assigning *false* is consistent with the unique constraint (UC) of SQL. Indeed, a UC on a set U of columns evaluates to *true* on a given relation if there are no two different records that have matching non-null values on all

the columns in U. It is surprising that SQL's UC has not received any dedicated focus in previous studies of the discovery problem. However, even UCs are often not robust enough to accommodate peculiarities of modern day data. As an illustration, consider the data snippet of the NCVoter data set in Table 1.

Here, the primary key on *voter_id* is violated. The reason may be due to an attempt to manually clean the data by assigning the same *voter_id* to both t_2 and t_3. The violation could be prevented by assigning $\perp$ to the *voter_id* for either t_2 or t_3, say to t_2 in which information appears less reliable (see *middle name* and *full_phone_number*). Giving up completeness, the modified relation would satisfy the UC on *voter_id*. Nevertheless, the UC on *voter_id*, being just a surrogate unique, cannot prevent the problem that the same voter may have been assigned different *voter_ids*. For example, *Nell Marshall* may refer to the same voter. Data profiling can help identify business keys that allow us to identify voters based on stable, real-life properties. A reasonable UC, denoted by UC_1, may be specified on the combination of *first_name*, *last_name*, and *register_date*. Indeed, a phone number or address can change and information on the *middle_name* is unreliable. Nevertheless, UC_1 is violated in Table 1. The only solution that known discovery algorithms employ is to include additional columns in the constraint, such as *phone_num* and *voter_id*, resulting in the UC UC_2 that is satisfied. The additional columns a) reduce the scope of the unique value combinations to records with no missing values on the extended combination of columns, and b) provide more features for such records to be distinguishable. For example, UC_1 has scope $\{t_0, \ldots, t_9\}$ since all tuples have no missing values on *first_name*, *last_name*, and *register_date*, while UC_2 has only scope $\{t_1, t_3, t_5, t_6, t_8, t_9\}$ since t_0, t_2, t_4, t_7 have missing values on *phone_num* or *voter_id*. This solution has the disadvantage that actually consistent entities, which are already distinguishable by the original set of columns, are unnecessarily subjected to additional column checks. Furthermore, any algorithms that discover all minimal UCs must keep on adding all possible column combinations until uniqueness is achieved for all records that are complete on these column combinations. This will result in outputs with UCs that are inflated in terms of their size whenever consistent data is already unique on a smaller set of columns. Such inflation penalizes access to consistent data when the discovered constraints are used. In an effort to overcome this problem, we study the discovery problem for a recent generalization of the SQL unique constraint [31].

SQL unique uses the same combination of columns to stipulate completeness and uniqueness requirements. Instead, given a set E of columns on which rows should be complete, it is more natural to ask which *minimal subsets* of columns are sufficient to identify each of these rows uniquely. For this reason, *embedded uniqueness constraints* (eUCs) where introduced recently [31]. These consist of a pair (E, U) of column combinations with $U \subseteq E$. For an eUC (E, U), the combination E is called the *extension*, and U is called the *associated UC* of the eUC. Given some relation r, the extension of (E, U) defines the *scope* as the subset r^E of records in r which are complete on all the columns in E. The eUC (E, U) holds on a given relation r whenever the scope r^E satisfies the UC on U. As such, UCs are the special case of eUCs where $E = U$. For example, after modifying the value 131 to $\perp$ in row t_2 of Table 1, the relation satisfies the eUC $eUC_1 = (E, U)$ where U consists of *first_name*, *last_name*, and *register_date*, while E consists of the columns in U plus *voter_id*, *phone_num*. The scope of eUC_1 is

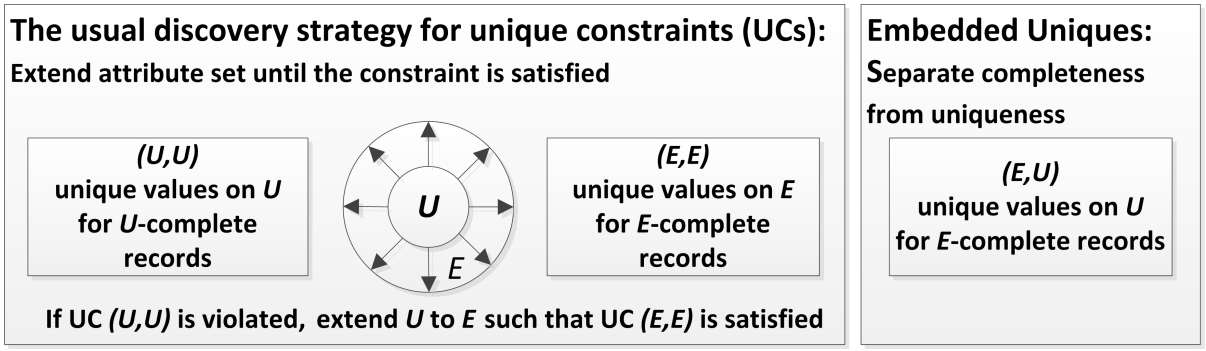


Figure 1: What makes Embedded Uniques better

$\{t_1, t_3, t_5, t_6, t_8, t_9\}$ and coincides with the set of rows that are complete on the columns of UC_2. The point is that UC_2 uses all five columns to identify each of the rows, while the eUC is able to uniquely identify each of these rows based on the proper subset U alone. As for SQL unique, the semantics of eUCs is independent of null marker interpretations, which is a huge advantage especially for modern applications like data integration. The concept of embedded uniqueness constraints is illustrated in Figure 1.

While eUCs subsume UCs as a special case, they empower users to separate completeness from uniqueness requirements while keeping their semantics independent of null marker interpretations. This is helpful for applications with specific completeness requirements. For example, voting campaigns may be conveniently targeted at voters that are registered with a phone number. Here, the phone numbers are not required to uniquely identify voters, but only used to contact them. Embedded uniques also help with the integration of incomplete data. Since inclusion dependencies reference data across tables, eUCs have been shown to provide the right notion of reference for incomplete data [20]. For example, we may want link new data to voters with a phone number. For such applications, it is useful to discover the many different combinations of extensions E and associated UCs U. Even eUCs that only hold accidentally are useful for query optimization, data access, and data linkage. It is therefore the task of data profiling to automate the process of discovering eUCs. Our main contributions are: **(1)** We distinguish the discovery of eUCs from previous work. **(2)** We show that the decision variant of the discovery problem for eUCs is NP- and $W[2]$-complete in the input size. **(3)** We characterize the largest possible number of minimal eUCs that an incomplete relation over n columns can have, and show which families of eUCs attain this number. The number is substantially larger than that of UCs. **(4)** We introduce a data structure for storing and looking up eUCs efficiently. **(5)** We establish the first column-efficient, row-efficient, and hybrid algorithms for the discovery of eUCs. Each of these is important and requires fundamentally new ideas over previous work. **(6)** As a special case of eUCs, we discover SQL UCs. **(7)** We conduct experiments on real-world benchmark data sets to demonstrate which algorithms perform well on which data sets, and that eUCs are effective in uniquely identifying most entities. **(8)** We propose the computation of Armstrong samples as a new direction in the profiling of incomplete data. Armstrong samples satisfy the same eUCs as the original data set, but can be substantially smaller. Users

may find this representation more helpful than an abstract set of constraints. **(9)** We conduct experiments on the computation and relative size of Armstrong samples. In particular, we generate relations with the maximum possible number of discoverable eUCs, and illustrate how quickly we can discover them. **(10)** We showcase how our discovery and sampling algorithms lead to effective data cleansing and business rule acquisition.
Organization. We discuss related work in Section 2. Basic definitions are given in Section 3. In Section 4, we investigate the computational complexity of discovering eUCs. In Section 6 we introduce an important data structure for our discovery algorithms. In Sections 7, 8, and 9, we present column-efficient, row-efficient and hybrid algorithms, respectively. In Section 10 we report our experimental results. In Section 11 we investigate Armstrong samples. Use cases are shown in Section 12. Conclusions and future work make up Section 13. More details, data sets, a prototype, and user guide are available[1].

2 Related Work

Embedded unique constraints were introduced recently [31]. Their implication problem and the generation of synthetic Armstrong relations from a given set of eUCs was studied, but neither the discovery problem nor the generation of Armstrong samples from a given data set were considered. In particular, eUCs are more expressive than unique constraints investigated in previous work. The current article is the first to investigate the discovery problem for embedded uniques. For any class of data dependency, we propose for the first time an approach to distinguish constraints that hold accidentally on a given data set from those that are meaningful (aka business rules). Our main idea is to combine sampling with discovery, which together unlock data cleansing and business rule acquisition. While one can combine the generation of Armstrong relations from [31] with our discovery algorithms, we also propose here the first algorithm that computes real-world Armstrong samples directly from the data. All our algorithms employ fundamentally new ideas, but they extend previous ideas for the discovery of unique constraints and functional dependencies (FDs). We discuss those ideas now. For a recent general survey on data profiling we refer to [1].

We strictly distinguish between *unique column combinations* (UCCs) and SQL unique constraints. UCCs evaluate $\perp == \perp$ to true, while SQL uniques evaluate it to false. SQL uniques form the special case of eUCs (E, U) where $E = U$. We also report results on this case because of its importance. By handling $\perp$ like any domain value, we also report on results for UCCs, but this is not our focus.

Column-based algorithms for the discovery of minimal keys [14] examine an attribute lattice bottom-up, top-down or in a hybrid manner. The bottom-up approach checks key candidates when all sets on the previous level are not satisfied. The top-down approach considers key candidates when some superset is satisfied. The hybrid approach combines bottom-up and top-down for faster pruning. The authors show upper bounds, but experiments consider only synthetic data. Larger column numbers cause efficiency problems.

[1] `http://bit.ly/2gzDEYu`

GORDIAN discovers minimal keys [30] using *prefix trees*, which efficiently extract *non-keys* known to violate a key on the given data. The algorithm performs well on data with quite large numbers of rows and columns. Experiments for real world data are limited, and missing values not discussed.

The Histogram-Count-based Apriori algorithm (HCA) discovers UCCs [2] by generating different cardinalities from small to large. Validations of UCCs are done by counting the frequencies of distinct values. Missing values are not discussed and the real world data only exhibits few UCCs.

Scalable UCC discovery [15] is achieved by employing additional search strategies on the attribute lattice. A greedy strategy looks for new candidates if the given UCC is not satisfied; and a random-walk looks for supersets (subsets) and randomly switches to new candidates when the current UCC is satisfied (unsatisfied). Missing values do not conform to SQL unique semantics. The algorithms scale poorly on larger attribute numbers under SQL unique semantics.

Hybrid algorithms for UCC and FD discovery [29, 27] switch between column- and row-based algorithms. The column-based algorithm validates UCCs in an attribute lattice bottom-up and switches when too many invalid UCCs are found. The row-based algorithm finds counter-examples heuristically and switches when too few counter-examples are found. In contrast, our row-based algorithm prunes the search space of the column-based algorithm, and the latter eliminates redundancies in the row-based algorithm. These additional strategies are necessary to handle the larger search space that eUCs exhibit over UCs and UCCs.

Stripped partitions and prefix blocks [16] improve the run-time efficiency of column-based algorithms. A row-based algorithm refines FD sets by extracting counter-examples from data [13]. FD-trees manage FDs efficiently. These early algorithms only deal with complete data.

Discovering conditional FDs [12] has received attention, but completeness has not been considered as a condition for conditional FDs. This suggests future work.

Possible and certain SQL keys [18, 21] rely on the no information interpretation of null marker occurrences, and are different from eUCs. Among other problems, their discovery via hypergraph transversals was studied [18, 21, 22]. Probabilistic and possibilistic keys [6, 17] are targeted at data models where uncertainty is modeled using possible worlds, each of which is complete. Those keys are for different data models.

Overall, our article is the first to investigate the discovery problem for eUCs, and the first to propose the combination of discovery and sampling for data cleansing and business rule acquisition. All our algorithms require fundamentally new ideas, and the dedicated focus on the handling of nulls is the first of its kind.

3 Embedded Unique Constraints

We give the basic definitions and fix notation.

A relation schema is a finite, non-empty set of attributes (also called *column (names)*), often denoted by R. With each attribute A we associate a domain $dom(A)$ of possible values that can occur in column A. A *tuple* t over R, sometimes called *row* or *record*,

is a function that maps each $A \in R$ to a value in $dom(A)$. Two records are equal if they have matching values on all the attributes of the underlying schema, and distinct otherwise. A relation r over R is a finite set of distinct tuples over R. For a finite set $X = \{A_1, A_2, \cdots, A_m\}$ of attributes, we sometimes write X as $A_1A_2 \cdots A_m$, and XY instead of the union $X \cup Y$ of X and another attribute set Y. Attribute sets are sometimes called *column combinations*. For $X \subseteq R$ and a tuple t over R, we write $t(X)$ to denote the projection of t onto X, that is, the value $dom(A_1) \times \cdots \times dom(A_m)$.

Following previous research, we use the special symbol $\perp$ to denote a null marker. While $\perp$ is a marker but not a value, we abuse notation for convenience and assume that $\perp$ is a distinct element of each domain. That is, $\perp$ is different from each domain value. We say a tuple t over R is X-*total* whenever $t(A) \neq \perp$ for all $A \in X$. Furthermore, we use r^X to denote the set of all X-total tuples in a relation r, that is, $r^X = \{t \in r \mid t \text{ is } X\text{-total}\}$, and call r^X *the scope* of r with respect to X. A relation is *complete* when it has no null marker occurrence, that is, when the scope r^R coincides with r. Following [31] we will study the discovery and sampling of *embedded uniqueness constraints*, defined as follows.

An *embedded uniqueness constraint* (*embedded unique* or *eUC*) over a relation schema R is an expression of the form (E, U) where $U \subseteq E \subseteq R$. We call E the *extension*, and U the *unique constraint* of (E, U). A relation r over R *satisfies* the eUC (E, U), or the eUC is said to *hold* on r, denoted by $r \vDash (E, U)$, if and only if for all $t, t' \in r^E$, $t(U) = t'(U)$ implies $t_1 = t_2$. If r does not satisfy (E, U), then we also say that r *violates* (E, U). Note that the case where $E = U$ or $E - U = \emptyset$ captures the semantics of the SQL unique constraint (UC). Since $E = U$, it is sometimes easier to write just U instead of writing (U, U). Whenever we want to save space, we also write $(E - U, U)$ instead of (E, U). Identifying column names in our introductory example by their first letters, we write $(\{v, p\}, \{f, l, r\})$ instead of $(\{v, p, f, l, r\}, \{f, l, r\})$.

A *unique column combination* (UCC) over relation schema R is also a set $U \subseteq R$. However, the UCC U is *satisfied* by a relation r over R whenever for every pair of distinct tuples in r (not just those that are U-complete), there is some attribute in U on which the two tuples have different values. In particular, the comparison $\perp == \perp$ evaluates to true for UCCs. In the special case of complete relations, the notions of UCs, UCCs, and eUCs all coincide with the well-known notion of a key.

UCs U (and UCCs, respectively) that hold on a relation r are said to be *minimal* if there is no UC U' (UCC U', respectively) that also holds on r and where U' is a proper subset of U. We can restrict the discovery of uniqueness constraints to those that are minimal, since any supersets are also uniqueness constraints. This is true for UCs as well as UCCs. This begs the question, which eUCs are minimal. We say that (E', U') is *subsumed* by (E, U), denoted by $(E', U') \sqsubseteq (E, U)$, if and only if both $E' \subseteq E$ and $U' \subseteq U$ hold. Further, (E', U') is *properly subsumed* by (E, U), denoted by $(E', U') \sqsubset (E, U)$, if and only if E' is a proper subset of E or U' is a proper subset of U. Now we can say that an eUC (E, U) that holds on relation r is minimal if and only if there is no eUC (E', U') that holds on r and is properly subsumed by (E, U). If an eUC is not minimal, we sometimes say it is *implied* or *redundant*. Given a set Σ' of eUCs, the subset Σ of Σ' is a minimal cover of Σ' if it consists of all those eUCs in Σ' that do not properly subsume any other eUCs in Σ. That is, Σ contains those elements of Σ' that are minimal with respect to subsumption.

Consider our introductory example. After replacing 131 in row t_2 by $\perp$, some of the eUCs the modified snippet of Table 1 would satisfy are $(\{m\}, \{m\})$, $(\{v, p\}, \{f, l, r\})$, and $(\{v, p\}, \{f\})$, of which only $(\{m\}, \{m\})$ and $(\{v, p\}, \{f\})$ are minimal: neither $(\{m\}, \emptyset)$, nor $(\{v\}, \{f\})$, nor $(\{p\}, \{f\})$, nor $(\{v, p\}, \emptyset)$ are satisfied.

4 Computational Complexity

In this section, we establish the computational complexity for the decision variant of the discovery problem for eUCs. Its decision variant, eUC, is defined as follows.

Problem:	eUC
Input:	relation r over schema R positive integer k
Output:	yes, if there is some $U \subseteq E \subseteq R$ where $\|E\| \leq k$ and r satisfies (E, U) no, otherwise

Note that $U \subseteq E$, so the cardinality $|E|$ of the extension E is an appropriate definition for the size of an eUC (E, U). Our first observation is that eUC is at least as hard as the decision variant Key of the key discovery problem in complete relations, defined as follows.

Problem:	Key
Input:	complete relation r over schema R positive integer k
Output:	yes, if there is some $K \subseteq R$ where $\|K\| \leq k$ and r satisfies K no, otherwise

Indeed, complete relations satisfy the key K if and only if they satisfy the eUC (K, K). It is known that Key is NP-complete [4], and by reducing Key to eUC we can establish NP-completeness for eUC, too.

Theorem 1 *The problem* eUC *is* NP*-complete.*

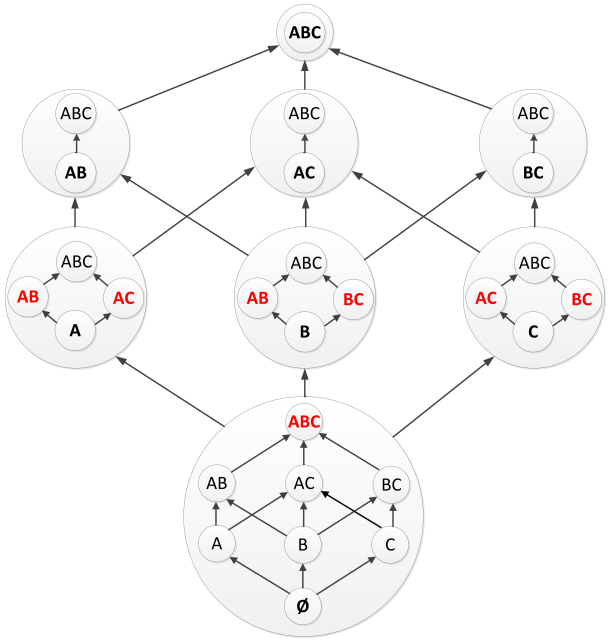
In recent research [5], Key was shown to be W[2]-complete in the size of the key. As we can show that Key and eUC are FPT-equivalent, it follows that the discovery of eUCs is likely to be an intractable problem even when the size of the eUCs is fixed. For the necessary definitions and proofs please see the appendix.

Theorem 2 *The problem* eUC *is* W[2]*-complete.*

Despite the likely intractability of the key discovery problem, even with a fixed input size, it is remarkable that recent algorithms can quickly find all minimal keys that hold on large real world data sets [30, 28]. The next section will show that the maximum number of eUCs that can hold on incomplete relations is much larger than the maximum possible number of minimal UCs. Despite the likely intractability, even with a fixed input size, and despite the large potential output, we will develop various efficient algorithms for the discovery of all minimal eUCs.

Table 2: Relation with maximum solution space over $A = id$, $B = name$, $C = phone$, together with its embedded lattice

id	*name*	*phone*
0	Adam	6756
0	⊥	⊥
⊥	Eve	7654
1	⊥	0023
⊥	⊥	6756
2	Dave	⊥
⊥	Adam	⊥



5 Maximum Solution Space

It is useful to know how large the solution space of the discovery problem can be. For most classes of constraints exact numbers are unknown. For example, only upper bounds are known for the maximum cardinality of a *non-redundant* family of FDs over a relation schema with n [9]. However, the maximum number of minimal keys over n attributes is $\binom{n}{\lfloor n/2 \rfloor}$ [8]. We will now establish a complete solution for the class of eUCs. While the result is interesting in its own right from a combinatorial perspective, it tells us precisely how large a solution space can be. The result shows that the solution space for eUCs is much larger than that for UCs. The result will also prove useful for experiments on synthetic data, as we can create data sets that attain the maximum number of minimal eUCs.

5.1 Extremal Families of Embedded Uniques

A family $\mathcal{F}$ of eUCs is non-redundant if and only if there there are no two eUCs (C, K) and (C', K') in $\mathcal{F}$ such that $C \subseteq C'$ and $K \subseteq K'$ hold. Our result will show that i) the maximum size of a non-redundant family of eUCs over a schema with n attributes is equal to the coefficient, denoted by $W(n)$, of x^n in the expansion of $(1 + x + x^2)^n$, and ii) the family that attains the maximum cardinality consist of all those eUCs (E, U) where $|E| + |U| = n$.

Table 2 exemplifies the case for $n = 3$ attributes. Here, the maximum family of minimal eUCs has seven elements, consisting of $(ABC, \emptyset)$, (AB, A), (AB, B), (AC, A), (AC, C), (BC, B), and (BC, C), as marked by red. In what follows, 2^X denotes the

power set of a set X.

Theorem 3 *Let R be a finite set, and let $\mathcal{F} \subseteq 2^R \times 2^R$ such that for all $(E,U) \in \mathcal{F}$: (i) $U \subseteq E$ and (ii) there is no $(E',U') \in \mathcal{F} - \{(E,U)\}$ with $(E',U') \sqsubseteq (E,U)$. Then $|\mathcal{F}| \leq W(|R|)$, where for $|R| \geq 2$ equality is attained if and only if*

$$\mathcal{F} = \left\{(E,U) \in 2^R \times 2^R : U \subseteq E \text{ and } |E| + |U| = |R|\right\}.$$

5.2 Comparison to UCs

We just want to emphasize that the maximum number of minimal eUCs is much larger than the maximum number of minimal UCs, which makes the solution space much larger. The following table illustrates the difference in concrete numbers up to $n = 12$ where these numbers are already separated by a factor higher than 79.

n	2	3	4	5	6	7	8	9	10	11	12
UCs	2	3	6	10	20	35	70	126	252	462	924
eUCs	3	7	19	51	141	393	1107	3139	8953	25653	73789

This is to illustrate the difficulty we face in developing efficient algorithms for the discovery of eUCs.

6 eUC-Trees As Data Structures

Facing big search and solution spaces, it is of utmost importance to provide a data structure that i) can represent a minimal cover of the set of eUCs found, and ii) can be used to decide whether some eUC is redundant. For this purpose, we will introduce the new data structure of eUC-trees, which we will employ in all our discovery algorithms. Our data structure generalizes the concept of antecedent trees from [13], which we recall here.

Given relation schema R with a total order of attributes, an *antecedent tree* over R is a tree such that: 1) Every node of the tree, except the root node, is an attribute of R, and 2) The children of a node are larger attributes.

In an antecedent tree, attribute sets are represented as paths, and different paths of the tree represent different attribute sets. An antecedent tree can effectively store a minimal cover of a set of keys. Antecedent trees cannot represent eUCs, since the latter involve both extensions and UCs. We therefore propose a new data structure, called eUC-*trees*, which serve the same purpose for eUCs that antecedent trees serve for keys. We say that an eUC-path *represents* an eUC (E,U) when E is the set of e-nodes of the path, and U is the set of u-nodes of the path.

Definition 1 (eUC-tree) *Let R be a relation schema with a total order on its attributes. An* eUC*-tree is a tree with nodes that are either the root, or labeled as either* e(xtension)-nodes *or* u(nique)-nodes *and satisfy the following properties:*

1. *Every node, except the root, is an attribute of R;*

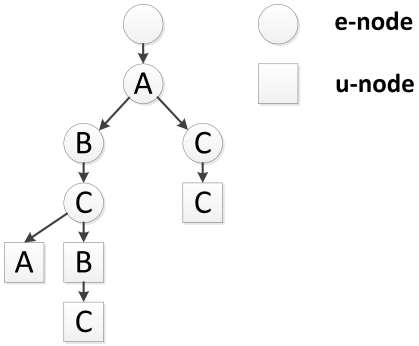


Figure 2: Example of an eUC-tree

2. *All children of the root are e-nodes;*
3. *E-nodes can have e-node or u-node children;*
4. *E-node children are larger than their e-node parent;*
5. *U-nodes only have u-node children;*
6. *U-node children are larger than their u-node parent;*
7. *For each path of the tree from the root to a leaf, the set of u-nodes is a subset of the set of e-nodes of the path.*
8. *The set of eUCs, represented by the different paths from the root to the leaves of the tree, is non-redundant.*

Example 1 (*eUC*-tree) *Figure 2 shows an example of an* eUC*-tree. The tree contains a set of non-redundant* eUC*s including* $(ABC, A), (ABC, BC)$ *and* (AC, C) *over the relation schema* $\{A, B, C\}$*.*

Algorithm 1 decides whether a given eUC (E, U) is redundant with respect to a given eUC-tree. For this we need to search for some path in the eUC-tree that represents an eUC $(E', U') \sqsubseteq (E, U)$. EUC-trees provide effective pruning mechanisms to support this search. The algorithm recursively traverses a chain of e-nodes and then u-nodes, starting at the root (line 23). A root node without children represents the eUC $(\emptyset, \emptyset)$, which is subsumed by every other eUC (line 5). Whenever an e-node is visited, the next step is to recursively traverse the u-node children, and then the e-node children. The algorithm only starts traversing u-nodes if the value of a u-node is not null (line 6). The search for a path can be limited to those with e-nodes (u-nodes, respectively) contained in E (in U, respectively), see lines 19 and 15.

We will employ Algorithm 1 for the *column-efficient*, *row-efficient*, and also the *hybrid* discovery algorithm of eUCs, in order to check for redundancies efficiently.

Algorithm 1

```
1: INPUT: Root node root of an eUC-tree, an eUC (E, U) over R
2: OUTPUT: true if (E, U) is redundant, false otherwise
3: function isRedundant(eNode, uNode)
4:     if eNode has no children then
5:         return true
6:     if uNode ≠ null then
7:         if uNode has no u-children then
8:             return true
9:         children ← the set of all u-children of uNode
10:        for child ∈ children ∩ U do
11:            if isRedundant(eNode, child) then
12:                return true
13:    else
14:        children ← the set of all u-children of eNode
15:        for child ∈ children ∩ U do
16:            if isRedundant(eNode, child) then
17:                return true
18:        children ← the set of all e-children of eNode
19:        for child ∈ children ∩ E do
20:            if isRedundant(child, uNode) then
21:                return true
22:    return false
23: return isRedundant(root, null)                    ▷ Invoke recursion here
```

7 Column-efficient Discovery

We first present a column-efficient, sometimes called row-based, algorithm for the discovery of eUCs.

The first step of the algorithm is to scan all pairs of distinct rows in the given relation. For each pair, we record the set E of columns on which both rows are total as well as the subset $U \subseteq E$ of columns on which both tuples have matching values. More formally, let r be a relation over R and $U \subseteq E \subseteq R$. The pair (E, U) is called an *embedded non-unique* (NU) of r if there are distinct $t_1, t_2 \in r^E$ such that i) for all $A \in R - E$, $t_1(A) = \perp$ or $t_2(A) = \perp$, and ii) for all $A \in E$, $t_1(A) = t_2(A)$ if and only if $A \in U$. A NU (E, U) of r is *maximal* if there is no NU (E', U') of r such that $(E, U) \sqsubset (E', U')$ holds. The set of maximal NUs (MNUs) of r is denoted by Σ^{-1}. The importance of Σ^{-1} for the discovery of eUCs is embodied in the following result. It says informally that an eUC holds in a relation if and only if the eUC is not subsumed by any MNU.

Theorem 4 *Let r be a relation over R. An* eUC *(E, U) is satisfied by r if and only if there is no $(E', U') \in \Sigma^{-1}$ such that $(E, U) \sqsubseteq (E', U')$.*

Theorem 4 forms the basis for the following iterative algorithm. Here, the minimal eUCs are represented by an eUC-tree from Section 6. If there is no maximal embedded non-unique, then every eUC holds and Algorithm 2 will simply return the root node,

representing the minimal cover $\{(\emptyset, \emptyset)\}$. Otherwise, we scan Σ^{-1} one by one element, and refine the current set of minimal eUCs that hold on r accordingly. Indeed, whenever a currently minimal eUC (E, U) is subsumed by the MNU (M, N) under inspection, then the algorithm removes (E, U) in line 7 (recursively removing the leaf of the path until the current node is a non-leaf of some other path), and adds the following eUCs: for all $A \in R - M$, (EA, U) is added, and for all $A \in M - N$, (EA, UA) is added, unless they contain some other minimal eUC.

Algorithm 2 Column-efficient algorithm

```
 1: INPUT: The set Σ^{-1} of r
 2: OUTPUT: The eUC-tree T_Σ representing a minimal cover Σ of those eUCs that hold on r
 3: T_Σ ← root                                          ▷ Start with just a root node
 4: for each (M, N) ∈ Σ^{-1} do
 5:     Ω ← {(E, U) ⊑ (M, N) | (E, U) is an eUC-path in T_Σ}
 6:     for (E, U) ∈ Ω do
 7:         Remove eUC-path (E, U) from T_Σ
 8:         for A ∈ R − M do
 9:             if (EA, U) non-redundant or T_Σ = ∅ then
10:                 if T_Σ = ∅ then
11:                     T_Σ ← root
12:                 Insert (EA, U) as a new eUC-path into T_Σ
13:         for A ∈ M − N do
14:             if (EA, UA) non-redundant or T_Σ = ∅ then
15:                 if T_Σ = ∅ then
16:                     T_Σ ← root
17:                 Insert (EA, UA) as a new eUC-path into T_Σ
18: Return T_Σ                          ▷ If Σ^{-1} = ∅, then T_Σ represents {(∅, ∅)}
```

Algorithm 2 works correctly, see the appendix.

Theorem 5 *Given the set of maximal embedded non-uniques of a relation, Algorithm 2 computes a minimal cover of the set of eUCs that are satisfied by the relation.*

8 Row-efficient Discovery

A row-efficient algorithm, sometimes called column-based, creates its search space from a given relation schema and verifies eUCs by traversing from the most general ones until all potentially valid eUCs in the search space have been examined. *Attribute lattices* have been widely used for row-efficient approaches to the discovery of data dependencies [16, 28]. As shown in Figure 3, *level* i of an attribute lattice contains all attribute sets of cardinality i. In particular, attribute sets of lower levels have smaller cardinalities and represent more general uniques. By traversing an attribute lattice from lower to higher levels, an algorithm can discover minimal uniques and prune redundant uniques in the search space. In the case of eUC discovery, the search space becomes significantly

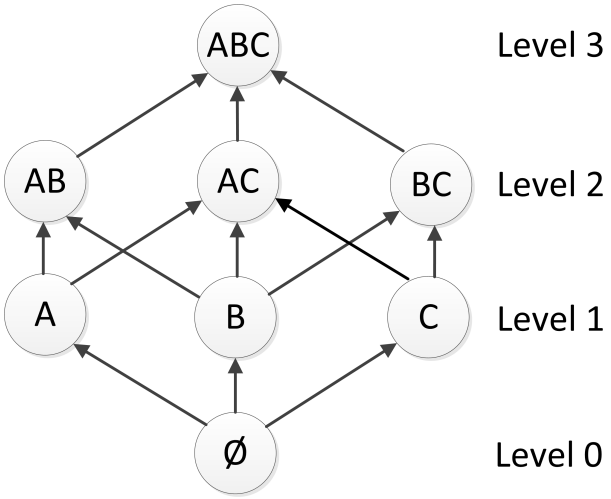


Figure 3: An Attribute Lattice and Its Levels

larger. For our row-efficient algorithm, we propose to use an attribute lattice, named *u(nique)-lattice*, to model the search space of the uniques associated with an eUC. While traversing a u-lattice, the algorithm employs another lattice, called *e(xtension)-lattice* for the discovery of all minimal extensions that apply to a given unique. We call traversals in the u-lattice *u-traversals*, and traversals in the e-lattice *e-traversals*.

Our algorithms for u- and e-traversals are based on characterizations that help us validate whether a given eUC holds on the given relation. In [16], the authors proposed to use the *stripped partitions* of a relation to validate FDs. We will now define the concept of stripped partitions for the purpose of validating eUCs.

Let r be a relation over R and $U \subseteq R$. The *U-equivalence class* of tuple $t \in r$ is the set $[t]_U = \{s \in r^U \mid t[U] = s[U]\}$. The *stripped partition* of a relation r over U is $\pi_U(r) = \{[t]_U \mid t \in r^U, |[t]_U| \geq 2\}$. The main use of stripped partitions in u-traversals is embodied in the following result. It provides an effective characterization to validate an eUC.

Proposition 1 (eUC validation)
An $\mathsf{eUC}(E, U)$ *over* R *is satisfied by a given relation* r *over* R *if and only if for all* $S \in \pi_U(r)$, $|r^E \cap S| \leq 1$.

However, the following result also shows how stripped partitions can be used in e-traversals. In effect, we can find an extension E for a given unique U such that the eUC (E, U) holds on r if and only if each stripped partition for U contains at most one total tuple. This helps us characterize effectively when we do not need to spend effort on finding an extension for a unique.

Proposition 2 (Existence of extensions)
Let $U \subseteq R$, *and* r *a relation over* R. *Then there is some* $E \subseteq R$ *with* $U \subseteq E$ *such that* r *satisfies* (E, U) *if and only if for all* $S \in \pi_K(r)$, $|r^R \cap S| \leq 1$.

Algorithm 3 Unique-traversal (row-efficient algorithm)

1: **INPUT:** A relation r over relation schema R
2: **OUTPUT:** The eUC-tree T_Σ representing a minimal cover Σ of those eUCs that hold on r
3: $T_\Sigma \leftarrow \emptyset$
4: $R' \leftarrow \{A \in R \mid \exists t \in r \text{ such that } t(A) = \perp\}$
5: extns $\leftarrow$ eTraversal$(R', \pi_\emptyset(r), \emptyset)$ ▷ $\pi_\emptyset(r) = \{r\}$
6: **if** $|\text{extns}| > 0$ **then**
7: $\quad T_\Sigma \leftarrow$ root
8: **for** $E \in$ extns **do**
9: $\quad$ insert $(E, \emptyset)$ as a new eUC-path into T_Σ
10: currentLevel $\leftarrow \{A \in R \mid (A, A)$ non-redundant in $T_\Sigma\}$
11: **while** $|\text{currentLevel}| > 0$ **do**
12: $\quad$ uGenNextLevel $\leftarrow \emptyset$
13: $\quad$ **for** $U \in$ currentLevel **do**
14: $\quad\quad$ **if** $r^U = \emptyset$ **then**
15: $\quad\quad\quad$ insert (U, U) as a new eUC-path into T_Σ
16: $\quad\quad\quad$ *continue* ▷ Goto line 13
17: $\quad\quad$ uGenNextLevel $\leftarrow$ uGenNextLevel $\cup \{U\}$
18: $\quad\quad$ **if** $|r^R \cap S| \leq 1$ for all $S \in \pi_U(r)$ **then**
19: $\quad\quad\quad$ $R' \leftarrow \{A \mid \exists S \in \pi_U(r), t \in S(t(A) = \perp)\}$
20: $\quad\quad\quad$ extns $\leftarrow$ eTraversal$(R', \pi_U(r), U)$
21: $\quad\quad\quad$ **for** $E \in$ extns **do**
22: $\quad\quad\quad\quad$ **if** (E, U) non-redundant or $T_\Sigma = \emptyset$ **then**
23: $\quad\quad\quad\quad\quad$ **if** $T_\Sigma = \emptyset$ **then**
24: $\quad\quad\quad\quad\quad\quad$ $T_\Sigma \leftarrow$ root
25: $\quad\quad\quad\quad\quad$ insert (E, U) as a new eUC-path into T_Σ
26: $\quad$ nextLevel $\leftarrow \emptyset$
27: $\quad$ **for all** $X, Y \in$ uGenNextLevel where $|XY| = |X| + 1$ **do**
28: $\quad\quad$ **if** (XY, XY) non-redundant or $T_\Sigma = \emptyset$ **then**
29: $\quad\quad\quad$ nextLevel $\leftarrow$ nextLevel $\cup \{XY\}$
30: $\quad$ currentLevel $\leftarrow$ nextLevel
31: **return** T_Σ

Next, we describe the u-traversal (row-efficient algorithm) as Algorithm 3, and the e-traversal as Algorithm 4. Algorithm 3 firstly computes the minimal extensions for an associated UC that is empty. Subsequently, a level-wise traversal on the u-lattice starts from the singleton attribute sets (those on Level 1). On each level, those UCs that are certain to have extensions will invoke an e-traversal as given by Algorithm 4. UCs for which no extensions exist, and UCs with larger extensions than themselves generate UCs for the next level. In a u-traversal, all discovered eUCs are stored in an eUC tree for fast redundancy checking. In e-traversal, instead of traversing an attribute lattice over an entire relation schema, only those attributes are used on which some tuple in some stripped partition holds a null marker. Finally, both Algorithm 3 and 4 (line 27 and 15, respectively), employ *prefix blocks* to generate candidates for the next level.

Prefix blocks were introduced in [3] and have been widely used for the discovery of data dependencies [24, 30, 2]. The blocks sort attribute sets in lexicographical order, and only form the union of two sets that have the same prefix on the first k attributes. This ensures that all attribute sets on the next level are generated exactly once. Otherwise, candidate attribute sets on the next level need to be generated by adding one attribute at a time to attributes sets of the current level, which will result in too many redundant new candidates.

Algorithm 4 Extension-traversal

```
 1: INPUT: Subset R' ⊆ R, stripped partition π_U(r), UC U
 2: OUTPUT: The set ℰ of all minimal extensions E such that (E, U) holds in r
 3: ℰ ← ∅
 4: currentLevel ← R'
 5: while |currentLevel| > 0 do
 6:     invalidExtns ← ∅
 7:     newValidExtns ← ∅
 8:     for E ∈ currentLevel do
 9:         if |r^E ∩ S| ≤ 1 for all S ∈ π_U(r) then
10:             ℰ ← ℰ ∪ {EU}
11:             newValidExtns ← newValidExtns ∪ {E}
12:             continue                                   ▷ Goto line 8
13:         invalidExtns ← invalidExtens ∪ {E}
14:     nextLevel ← ∅
15:     for all E, F ∈ invalidExtns where |EF| = |E| + 1 do
16:         if ¬∃E' ∈ newValidExtns where E' ⊆ EF then
17:             nextLevel ← nextLevel ∪ {EF}
18:     currentLevel ← nextLevel
19: return ℰ
```

The computation of stripped partitions for uniques affects the scalability of Algorithm 3 on relations with a large number of rows. This is because it is inefficient to recompute stripped partitions for the entire relation from scratch for each unique. As another novelty, we propose Algorithm 5, which computes stripped partitions iteratively. The algorithm verifies whether tuples in the same current partition have matching total values on the new attribute. In essence, each occurring total value on the new attribute represents a new partition. Consequently, tuples of the input stripped partition are directly mapped into new partitions according to their values on the new attribute, see line 5.

In Algorithm 3, extensions and their uniques are enumerated by cardinalities. If a unique with itself as an extension cannot form a valid eUC, the eUCs formed by supersets of the unique may be valid and non-redundant. Such eUCs are augmented by one attribute, exhausting all possibilities. While examining a unique, all its extensions are also enumerated by cardinalities so that only non-redundant ones are discovered. Similarly, if an extension cannot form a valid eUC with a given unique, it is augmented

by one attribute and validated on the next level. At the end, all minimal eUCs of a given relation have been computed.

Algorithm 5

```
1: INPUT: Stripped partition π of r over U, A ∈ R − U
2: OUTPUT: The stripped partition π′ of r over UA
3: π′ ← ∅
4: for S ∈ π do                          ▷ Create map M from r[A] to tuple sets
5:     for t ∈ S do
6:         if t(A) ≠⊥ then
7:             M[t(A)] ← M[t(A)] ∪ {t}
8:     for each set S in M do
9:         if |S| > 1 then
10:            π′ ← π′ ∪ {S}
11: return π′
```

Theorem 6 *Algorithm 3 computes a minimal cover of the set of eUCs that are satisfied by the given relation.* ∎

9 Hybrid Discovery

So far, our algorithms were targeted at relations with a large number of either columns or rows. Each algorithm suffers from defects that require new strategies to correct. The column-efficient algorithm has to compare all distinct rows, resulting in a quadratic growth of the running time in the number of rows. Moreover, redundant intermediate results are produced frequently. The row-efficient algorithm operates on a huge search space, which grows exponentially in the number of columns. Since stripped partitions are created at each level of the attribute lattice, the algorithm also duplicates a lot of information, which creates problems with the available memory. As a solution, we are now proposing a *hybrid algorithm* that utilizes good aspects of the column-efficient algorithm to compensate defects of the row-efficient algorithm, and vice versa. This amalgamation of ideas allows us to efficiently mine data sets that have a large number of both columns and rows.

Reducing search space. The column-efficient algorithm can help reduce the number of attribute sets that both u- and e-traversals consider on each level. Recall that NUs can be used to identify invalid eUCs and to derive new satisfiable eUCs. In a u- or e-traversal, an invalid attribute set is expanded by each remaining attribute. For example, if E is not an extension for U, then one checks if r satisfies (EA, U) for all $A \in R - E$. However, if an extension and its associated UC are subsumed by some NU (M, N), then one only needs to check if r satisfies (EA, U) for all $A \in R - M$. In fact, the row-efficient algorithm views invalid eUC as an NU, and then derives new eUCs. The use of NUs can thus reduce the search space in the row-efficient algorithm.

Reducing intermediate eUCs. The row-efficient algorithm can help the column-efficient algorithm reduce the number of eUCs generated at intermediate steps. By Theorem 4, the column-efficient algorithm cannot decide if an eUC is valid until the last MNU has been processed. When an eUC, such as (E, U), is subsumed by an NU, an extension of the eUC, such as (EA, U), is either redundant or not regarding some validated eUC. If it is redundant, then all eUCs that subsume (EA, U) are redundant, too. Hence, timely validation of eUCs reduces the number of intermediate eUCs generated by the column-efficient algorithm. In fact, one can validate eUCs of an eUC-tree in a level-wise manner, because levels of UCs and their extensions can be computed by traversing the eUC-tree. For this type of pruning, we define M_1 and M_2 as mappings that assign an attribute set to some eUCs. M_1, called *extension hints* (EH), is defined by $A \in M_1[E, U]$ iff (EA, U) is subsumed by some valid eUC, and M_2, called *unique hints* (UH), is defined by $A \in M_2[E, U]$ iff (EA, UA) is subsumed by some valid eUC.

Hybridization. Our hybrid algorithm runs the row-efficient algorithm as its core, but employs the column-efficient algorithm to update the search space whenever convenient. This results in an *hybrid e-traversal* algorithm and a *hybrid u-traversal* algorithm.

The hybrid e-traversal validates the extensions of a given UC level by level. Before a new level is used, hybrid e-traversal decides whether new NUs should update its search space. The decision is controlled by the ratio of the number of invalid extensions over the number of all extensions on a level. Similar to Algorithm 4, invalid extensions generate candidate extensions on the next level. Hence, the more invalid extensions are found on the current level, the more candidate extensions need to be validated on the next level. If the ratio exceeds a certain threshold, meaning that too many candidates would need to be validated, the search space is updated by a set of NUs sampled from stripped partitions. Otherwise, the algorithm only uses NUs composed by invalid extensions to update the search space. For example, if E is not an extension for U, (E, U) must be an NU. Eventually, e-traversal returns updates of the eUC-tree, EHs and UHs to the u-traversal algorithm.

Unlike the u-traversal algorithm in Algorithm 3, hybrid u-traversal does not only discover the extensions of a UC level by level, but also employs NUs returned by hybrid e-traversal to update the eUC-tree at the end of each iteration. Note that hybrid e-traversal will update the entire eUC-tree, so it is no longer necessary for hybrid u-traversal to explicitly compute UCs for the next level.

The pseudo-code and description of our hybrid algorithms can be found in Section B.

Theorem 7 *Our hybrid discovery Algorithm 9 computes a minimal cover of the set of eUCs that are satisfied by the given relation.* ∎

10 Experiments

We have conducted experiments on real world data sets to illustrate the performance and practicality of our algorithms. Theses data sets have emerged as benchmark data sets for testing the performance of discovery algorithms for classes such as functional dependencies[26, 28]. We implemented the proposed algorithms in Visual C++, and

Data set	#R	#C	#⊥	#IR	#IC	#eUC	#UC	Alg. 2	Alg. 3	Alg. 9
horse	300	28	1605	294	21	5040	31	**1.046**	ML	1.167
bridges	108	13	77	38	9	3	3	0.003	0.0039	**0.002**
hepatitis	155	20	167	75	15	446	102	**0.082**	17.991	0.154
breast-cancer	691	11	16	16	1	2	1	0.083	0.187	**0.009**
echocardiogram	132	13	132	71	12	45	27	0.006	0.018	**0.006**
plista	996	63	23317	996	32	2337	49	**3.369**	ML	4.177
flight	1000	109	51938	1000	69	26652	33672	**49.367**	ML	106.633
ncvoter	1000	19	2863	1000	5	147	69	0.346	1.376	**0.067**
uniprot	1000	223	179129	1000	212	3320220	664	4106.66	ML	**2742.15**
pm2.5china	262920	18	418580	157895	12	615	470	TL	ML	**77.365**

Table 3: Run time (in seconds) of the three algorithms to discover eUCs from incomplete data

Data set	#R	#C	#UC	Alg. 2	Alg. 3	Alg. 9
abalone	4177	9	29	2.8	0.18	**0.09**
adult	32537	15	2	205.99	ML	**0.64**
chess	28056	7	1	116.27	1.25	**0.21**
iris	147	5	1	0.004	**0.001**	**0.001**
letter	18668	17	1	78.99	ML	**0.55**
nursery	12960	9	1	34.65	2.19	**0.15**
balance-scale	625	5	1	0.06	0.005	**0.004**
fd-reduced	250000	30	3564	TL	**110**	313

Table 4: Discovery time (s) on complete data

carried out our experiments on an Intel i7-5820K, 3.3 GHz, 8 GB, Windows 10 PC. **Repeatability.** A prototype system and our data sets have been made available[2].

Next, we present our findings. For the experiments we set a time limit (TL) of 2 hours and a memory limit (ML) of 6 GB. The benchmarks include complete and incomplete data sets. For each data set, we report the number of rows (#R), columns (#C), missing values (#⊥), incomplete rows (#IR), incomplete columns (#IC), unique constraints (#UC), eUCs (#eUC), and the running time of each algorithm for the discovery of the eUCs. Since UCs just represent the special case of eUCs where the extension and associated UC coincide, we have simply indicated their total number. Over complete data sets, all three notions of UCCs, UCs, and eUCs coincide. We point out that our algorithms are designed for the discovery of eUCs from incomplete data, which covers a much larger search space than the discovery problem of UCCs or UCs.

Tables 3 and 4 show our results on the incomplete and complete data sets, respectively. Since most of the incomplete data sets only have a small number of rows, the column-efficient algorithm (Alg. 2) performs better on some of them, but has rarely a huge advantage over the hybrid algorithm (Alg. 9). Note that neither Alg. 2 nor Alg. 3 can process the data set *pm2.5china* [23] within the given time and memory limits. On the complete data sets, the hybrid algorithm usually wins. However, the row-efficient algorithm achieves typically a better running time on data sets with a large number of rows. In conclusion, the hybrid algorithm performs well overall but the column- and row-efficient algorithms usually perform better on data sets with an extreme number of columns or rows. This confirms our expectations based on the design of the algorithms.

We can also discover UCCs by not choosing a symbol that is interpreted as ⊥. Their

[2] `http://bit.ly/2gzDEYu`

Data set	#UCC	Alg. 2	Alg. 3	Alg. 9
horse	253	0.283	ML	0.128
bridges	5	0.003	0.047	0.003
hepatitis	348	0.06	19.318	0.161
breast-cancer	2	0.162	0.189	0.009
echocardiogram	72	0.008	0.026	0.011
plista	1	0.851	ML	0.308
flight	26652	7.8	ML	25.632
ncvoter	69	0.395	3.364	0.051
uniprot	?	ML	ML	ML
pm2.5china	2	TM	ML	12.997

Table 5: Discovery time (s) on incomplete data

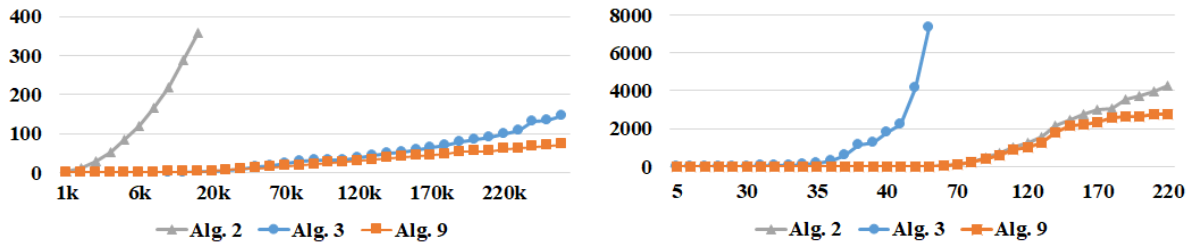


Figure 4: Row scalability on *uniprot* [left] and column scalability on *pm2.china* [right]

total numbers (#UCC) and corresponding running times of our algorithms for their discovery from the incomplete data sets are shown in Table 5.

To further analyze the row efficiency and column efficiency of our proposed algorithms, we analyze the discovery on projections on the data set *uniprot* with an increasing number of columns, and on subsets of the data set *pm2.5china_14c* with an increasing number of rows. Figure 4 shows how the run time of our algorithms scales when the number of rows or columns increase, respectively. Although the row- or column-efficient algorithm perform slightly better when the number of columns or rows is small, the hybrid algorithm eventually outperforms the other two algorithms when the number of columns or rows grows larger. Again, this meets the design expectations of all algorithms: Row-/column-efficient algorithms win when there are few enough columns/rows, respectively, while the hybrid algorithm wins when column and row numbers are large enough.

The more rows in the scope of an eUC, the more rows can be identified uniquely. We could use potentially different eUCs to distinguish different rows. Hence, we say that a row in a data set r is *covered* if there is some discovered eUC (E, U) such that the row belongs to the scope r^E. The eUC *coverage* of a data set is the ratio of rows in the data set that are covered.

Figure 5 shows the eUC coverage on each of the incomplete data sets. We distinguish between UCs and truly embedded UCs (where the extension properly contains the associated UC). Since UCs have the minimum extension amongst all eUCs with the same associated UC, their scope has maximum cardinality. However, the point is to discover

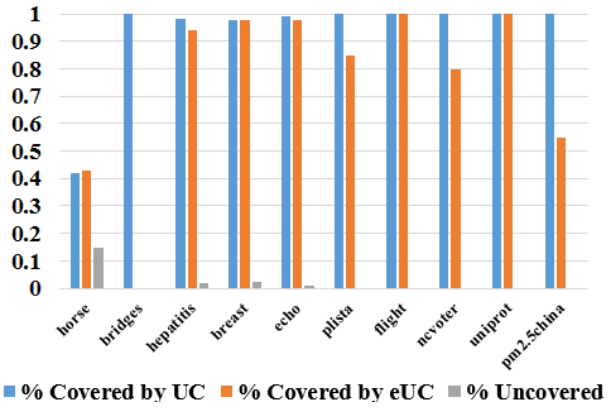


Figure 5: Coverage of incomplete data sets

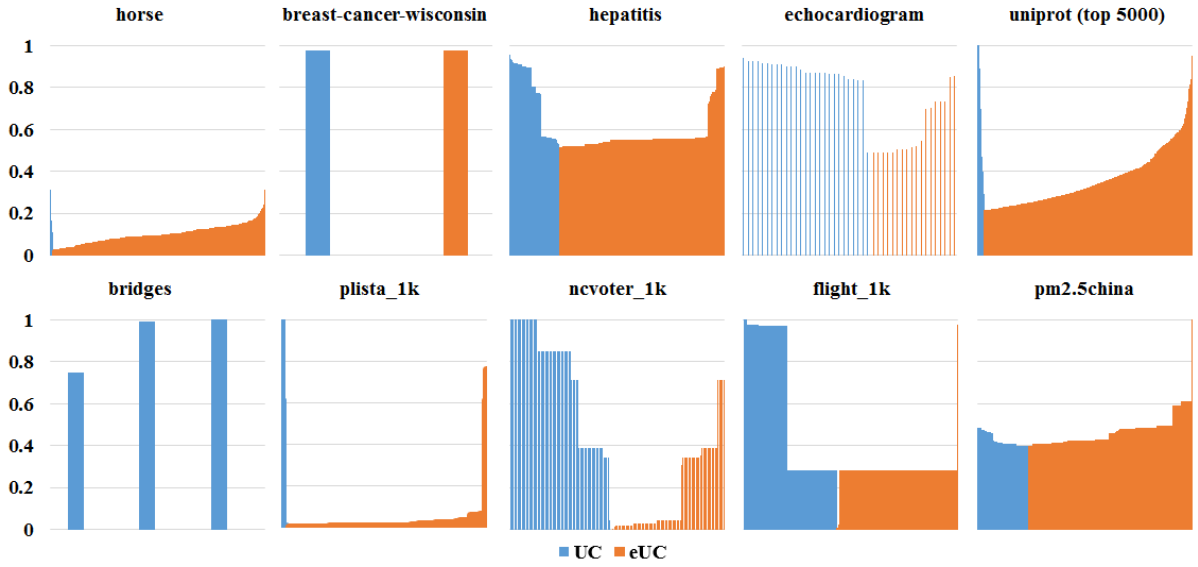


Figure 6: Relative scope of individual eUCs

which associated UCs are sufficient for the identification of which rows. Indeed, the high number of eUCs with a high coverage of truly embedded UCs is remarkable: It shows that there are many different ways by which a large proportion of rows can already be distinguished by a proper subset of the extension in eUCs. Recall that this was precisely our reason for studying them.

The results are even more encouraging when we look at the relative scope for each individual eUC discovered for a given data set. This is defined as the cardinality of the scope relative to the number of rows in the given data set. The relative scopes are shown in Figure 6. Evidently, there are quite a few eUCs in each data set which can uniquely identify most of the rows, except for the data set *horse*. Data sets have low coverage when several rows are ignored by eUCs, since such rows would create violations of the eUCs otherwise. In fact, rows outside the scope may represent less reliable data, because some desired eUCs would be violated if the missing information in these rows was updated.

We conclude that our experimental results confirm i) the practicality of eUCs as an effective mechanism to uniquely identify entities in incomplete data sets, independent of the interpretation of null markers, and ii) the effectiveness of our algorithm designs in efficiently discovering eUCs. The following section will introduce a novel direction in data profiling, which is to compute a user-friendly representation of the constraints that have been discovered.

11 Data Profiles by Example

Current data profiling tools return the set of constraints in a given class that hold on the given data set. Here we propose the use of data samples that perfectly represent the set of discovered constraints. The data samples are subsets of the original data set and are perfect because they satisfy the same set of constraints as the original data set. Such perfect samples are attractive for a number of reasons: They are accessible to a broader audience than abstract sets of constraints, and provide a good foundation for understanding why some constraints are satisfied or violated. Hence, data analysts gain insights into large data sets by looking at the right samples. While these perfect samples are already known as *informative Armstrong databases* [7], they have not been proposed as a useful tool in data profiling. Apart from [7], informative Armstrong databases have not been studied, and not at all for incomplete data.

We recall the definition of an informative Armstrong sample [7]. Given a relation r and a class $\mathcal{C}$ of data dependencies, an *Armstrong sample* for r with respect to $\mathcal{C}$ is a subset $r' \subseteq r$ such that r' and r satisfy the same dependencies in $\mathcal{C}$. Armstrong samples always exist since every relation is an Armstrong sample of itself. The idea is to find Armstrong samples of small size if possible.

As mentioned under related work, we proposed an algorithm to compute an *Armstrong relation* with synthetic data from a given set of eUCs [31]. For computing an informative Armstrong sample of the given data set, we could use any of our discovery algorithms to compute Σ from the data set, then compute the set Σ^{-1} of maximal non-uniques from Σ, and then choose the right tuples from the data set to get the sample. Instead, we propose here to compute Σ^{-1} directly from the data, and then construct an Armstrong sample by picking for each MNU (E, U) in Σ^{-1} a pair of rows from the stripped partition $\pi_U(r^E)$, see Algorithm 6.

Algorithm 6 Armstrong sample

```
1: INPUT: A set Σ^{-1} of maximal non-uniques over R
2: OUTPUT: An Armstrong sample r' of r
3: r' ← ∅
4: for each (E, U) ∈ Σ^{-1} do
5:     Select t_1, t_2 ∈ S where t_1 ≠ t_2 and S ∈ π_U(r^E)
6:     r' ← r' ∪ {t_1, t_2}
7: return r'
```

Table 8 shows the sizes of the Armstrong samples for our benchmark data sets. Typi-

Data set	#R	#⊥	#IR	#IC	#MNU	%
horse	91	248	85	21	178	30.33
bridges	4	0	0	0	2	3.70
hepatitis	73	21	15	6	73	47.10
breast-cancer	12	1	1	1	6	1.74
echocardiogram	28	19	13	5	16	21.21
plista	109	2689	109	34	214	10.94
ncvoter	114	255	114	5	65	11.40
flight	589	30131	589	69	739	58.90
uniprot	907	162048	907	212	14403	90.70
pm2.5china	509	78	51	5	255	0.19
abalone	56	0	0	0	30	1.34
adult	20	0	0	0	10	0.06
chess	10	0	0	0	6	0.04
iris	8	0	0	0	4	5.44
letter	32	0	0	0	16	0.17
nursery	11	0	0	0	8	0.08
balance-scale	5	0	0	0	4	0.80
fd-reduced	461	0	0	0	231	0.18

Table 6: Traits of real-world Armstrong samples

cally, the samples are much smaller than the original data set. There are several samples whose size exceeds 20%, but this is to be expected since the size of Σ^{-1} can be huge, in fact similar to the number of minimal eUCs. For example, *uniprot* has only 1000 rows but more than three million eUCs are valid. Armstrong samples can be presented to end users in different ways. Instead of presenting them as one table, one could present one pair of records for each maximal non-unique at a time.

We also computed Armstrong relations for the extreme eUC families from Section 5, and applied our discovery algorithms subsequently. The Armstrong relations were populated with synthetic data, and represent relations that exhibit the maximum possible search space for our discovery algorithms. Table 7 summarizes our results: For $n = 2, \ldots, 12$, it shows the maximum cardinality $\#\mathcal{F}_n$ of eUCs on relations with n columns, the number #arm of rows in the Armstrong samples, the time #time to compute them in seconds, the number $\#\perp$ of missing values in these relations, and the times to discover the families from the samples based on the three algorithms. Note that every column contains $\perp$, and only one row in each relation does not contain $\perp$. Since the maximum search space grows so quickly, so does the number of rows in an Armstrong relation and the time to compute it. For 12 columns, the generation took over 9.6 hours, but the row-efficient algorithm took less than 2 minutes to discover all 73,789 eUCs. While the hybrid algorithm is beaten due to the small number of columns, one can still appreciate its impact by the dramatic performance improvement over the column-efficient algorithm (which is not designed to perform well on data sets with 12 columns).

Lastly, we illustrate the use of Armstrong samples on a real-world sample of the data

n	$\#\mathcal{F}_n$	#arm	#time	$\#\perp$	Alg. 9	Alg. 3	Alg. 2
2	3	3	>0	2	>0	>0	0.001
3	7	7	>0	9	>0	>0	0.001
4	19	17	0.002	28	>0	0.001	0.001
5	51	46	0.008	95	0.002	0.001	0.002
6	141	127	0.044	306	0.012	0.003	0.018
7	393	358	0.291	987	0.064	0.014	0.17
8	1107	1017	3.019	3144	0.415	0.07	3.326
9	3139	2908	22.25	9963	2.94	0.404	52.68
10	8953	8351	222.7	31390	23.3	2.64	797
11	25653	24069	2116	98483	187.1	17.1	18256
12	73789	69577	34779	307836	1576	112	-

Table 7: Traits of synthetic Armstrong relations

0	1	2	3	4	5	6	7	8	9	10	11	12
E19	A	29	1866	HIGHWAY	1000	2	N	THROUGH	WOOD	MEDIUM	S	WOOD
E11	A	29	1851	HIGHWAY	1000	2	N	THROUGH	WOOD	MEDIUM	S	WOOD
E48	A	38	1900	HIGHWAY	2000	2	G	THROUGH	STEEL	MEDIUM	F	SIMPLE-T
E58	A	33	1900	HIGHWAY	1200	2	G	THROUGH	STEEL	MEDIUM	F	SIMPLE-T

Table 8: Armstrong sample of *bridges* with eUCs: $(\emptyset, \{0\}), (\emptyset, \{2, 3\})$ and $(\emptyset, \{3, 5\})$. Attributes are: identifier (0), river (1), location (2), erected year (3), purpose (4), length (5), lanes (6), clear-g (7), t-or-d (8), material (9), span (10), rel-l (11), and type (12).

set *bridges*, shown in Tables 6, 9 for eUCs and UCCs, respectively. For UCCs, the missing values on column 5 are interpreted as equal, which means we need an additional column to ensure the two rows can be distinguished, resulting in the UCCs {3,4,5}, {3,5,11}, and {3,5,12}. Considering the two missing values on column 5 (length) as equal is difficult to justify since both lengths exist and are likely different. One may say the UCCs are inflated since the more reliable rows can already be distinguished by {3,5}. This inflation is necessary because of the intent to distinguish less reliable tuples. Indeed, under SQL unique semantics we discover that {3,5} is already unique, with no intention to distinguish rows with missing values on the columns that are involved.

12 Semantic Data Profiling

We illustrate the use of our discovery and sampling algorithms for iterative data cleansing and business rule acquisition on the showcase of *ncvoter*. We also briefly mention use cases for eUCs that do not represent business rules (ie. only hold accidentally on the given data set).

Uncovering Surrogate Keys. Surrogate keys can speed up data management by accessing records via a single integer. By briefly inspecting the data set, one would expect that *voter_id* forms a surrogate key. However, the discovery of all eUCs (shown Table 10) reveals that no eUC $(\emptyset, \{voter\_id\})$ with 100% coverage exists. Data stewards wonder why the key is violated. Targeted inspection of the generated Armstrong sample

0	1	2	3	4	5	6	7	8	9	10	11	12
E19	A	29	1866	HIGHWAY	1000	2	N	THROUGH	WOOD	MEDIUM	S	WOOD
E11	A	29	1851	HIGHWAY	1000	2	N	THROUGH	WOOD	MEDIUM	S	WOOD
E48	A	38	1900	HIGHWAY	2000	2	G	THROUGH	STEEL	MEDIUM	F	SIMPLE-T
E58	A	33	1900	HIGHWAY	1200	2	G	THROUGH	STEEL	MEDIUM	F	SIMPLE-T
E96	Y	51	1945	RR	⊥	⊥	G	THROUGH	STEEL	MEDIUM	F	SIMPLE-T
E97	Y	52	1945	HIGHWAY	⊥	⊥	G	THROUGH	STEEL	MEDIUM	S	ARCH

Table 9: Armstrong sample of *bridges* data set considering UCCs: $\{0\}, \{2,3\}$, $\{3,4,5\}$, $\{3,5,11\}$ and $\{3,5,12\}$

extension	unique	coverage
name_prefix	voter_id	2.7%
	voter_id, middle_name	85.1%
name_suffix	voter_id	4.5%
	voter_id, street_address	100%
	voter_id, phone_number	38.8%
	voter_id, download_month	100%

Table 10: EUCs using *voter_id*

reveals the two records shown in Table 11. They only differ on *address*, *phone_number*, and *download_month*. So, these two records show us why *voter_id* is not a surrogate key. Indeed, the older record contains less reliable information, in particular it is unlikely to have "252 000 0000" as a phone number. In an attempt to recover the validity of the expected surrogate key, we replace the value 131 of the older record by the null marker. This preserves all the information but also uncovers other meaningful eUCs. For example, the eUCs $(\emptyset, \{voter\_id\})$ and $(\{voter\_id\}, \{registration\_number\})$ become valid on the modified data set. In particular, old and new record are still linked via the same value on *registration_number*.

Uncovering Futile Attributes. Many of the discovered eUCs contain *name_prefix* or *name_suffix* in their extension E. However, the coverage of these eUCs is only 2.7%. As shown in Fig. 7, most values of these attributes are missing. For existing values such as "Jr", "Sr", or "Mrs", only limited information is available. Indeed, all voters with an existing value on *name_suffix* can already be distinguished by *first_name* and *last_name*. In addition, no eUC contains *name_prefix* or *name_suffix* in their set U. One may strongly consider the removal of these attributes from the data set.

Uncovering Implicitly Missing Data. Continuing on, the following eUC validates after another round of discovery: $(\{middle\_name\}, \{first\_name, zip\_code, phone\_number\})$ means there are two different records with matching values on *first_name*, *zip_code*, and *phone_number*, and one of them should not have a middle name. This eUC represents the unusual scenario that different people with the same phone number have the same first name. In particular, situations like James Bond Sr and James Bond Jr do not occur. After re-sampling the data, we found the two records shown in Table 12. These voters have the same first name and live in the same area, but both have phone number "000 0000". In fact, 51 out of 388 voters have the phone number "000 0000". This suggests to replace the occurrences of this value by the null marker. In fact, doing so will result

voter_id	*address*	*phone_number*	*download_month*
131	1108 highland ave #22	252 288 4763	2011-12
131	9 Casey rd	252 000 0000	2011-10

Table 11: Snapshot of Armstrong sample for *ncvoter*

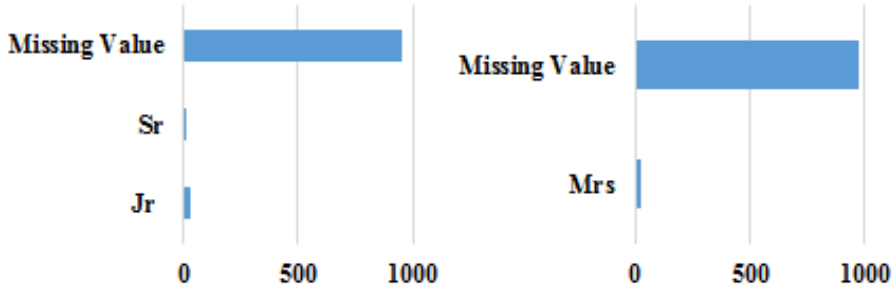


Figure 7: Values of *name_prefix* (left) and *name_suffix* (right)

in the validity of the business rule $(\emptyset, \{first\_name, zip\_code, phone\_number\})$.

Discovery and sampling together answer the true calling of data profiling. Indeed, the intrinsic links between data cleansing and business rule acquisition are unlocked. Business rules provide an instrument for data cleansing, and data cleansing leads to the discovery of business rules (and not just constraints that hold accidentally).

Even eUCs that hold accidentally can be used to speed up query processing (eg detect unnecessary duplicate removal from query results that are already unique due to some eUCs), data access (eg via creating indexes), and data linkage (referencing unique records in the given data set).

13 Conclusion and Future Work

We presented the first discovery algorithms for embedded uniqueness constraints (eUCs). These constraints are expressions of the form (E, U), and separate completeness from uniqueness requirements. Applications can uniquely identify E-complete records by a minimal subset $U \subseteq E$ of columns. The validity of eUCs is independent of how nulls are interpreted. Known uniqueness constraints, such as SQL unique or unique column combinations, occur as simple special cases of eUCs. We showed that the problem of discovering an eUC with a given maximum size is both NP-complete and W[2]-complete in the input size. We further characterized the maximum possible solution size for the discovery problem over any given number of columns, and determined which families of eUCs attain this size. Despite these challenges, we established the first column-efficient, row-efficient, and hybrid algorithms for the discovery of all eUCs that hold on a given relation. Our hybrid algorithm performs well overall and is especially suited for data sets with a large number of columns and rows. For data sets with a large number of either columns or rows, the hybrid algorithm is outperformed by the other algorithms. Our experiments confirmed that the many eUCs discovered, in particular those with a

voter_id	*first_name*	*middle_name*	*zip_code*	*phone_number*
687	Margaret	Caudle	27215	000 0000
952	Margaret	$\perp$	27215	000 0000

Table 12: Records with suspicious phone numbers

high relative scope, offer opportunities for targeted and fast access to data by applications with completeness and uniqueness requirements. There were many eUCs that can uniquely identify every row in most of the data sets we analyzed. Finally, we proposed Armstrong sampling as a new direction in data profiling. We exemplified how end users can gain insight from looking at the samples, and how the combination of our profiling and sampling tools provides a pathway to effective data cleansing and business rule acquisition. In future work, we will investigate different hybrid and scalable approaches to the discovery of eUCs. Other classes of embedded data dependencies are appealing, foremost embedded functional dependencies as they can provide a foundation for developing a database design theory similar to that of relational databases [19].

References

[1] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: a survey. *VLDB J.*, 24(4):557–581, 2015.

[2] Z. Abedjan and F. Naumann. Advancing the discovery of unique column combinations. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1565–1570. ACM, 2011.

[3] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo, et al. Fast discovery of association rules. *Advances in knowledge discovery and data mining*, 12(1):307–328, 1996.

[4] C. Beeri, M. Dowd, R. Fagin, and R. Statman. On the structure of Armstrong relations for functional dependencies. *Journal of the ACM (JACM)*, 31(1):30–46, 1984.

[5] T. Bläsius, T. Friedrich, and M. Schirneck. The parameterized complexity of dependency detection in relational databases. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 63. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[6] P. Brown and S. Link. Probabilistic keys. *IEEE Trans. Knowl. Data Eng.*, 29(3):670–682, 2017.

[7] F. De Marchi and J.-M. Petit. Semantic sampling of existing databases through informative Armstrong databases. *Information Systems*, 32(3):446–457, 2007.

[8] J. Demetrovics. On the number of candidate keys. *Inf. Process. Lett.*, 7(6):266–269, 1978.

[9] J. Demetrovics and G. O. H. Katona. A survey of some combinatorial results concerning functional dependencies in database relations. *Ann. Math. Artif. Intell.*, 7(1-4):63–82, 1993.

[10] R. G. Downey and M. R. Fellows. *Fundamentals of parameterized complexity*, volume 4. Springer, 2013.

[11] K. Engel. *Sperner Theory*. Cambridge Univ. Press, 1997.

[12] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5):683–698, 2011.

[13] P. A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI communications*, 12(3):139–160, 1999.

[14] C. Giannella and C. Wyss. Finding minimal keys in a relation instance. `citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7086`, 1999.

[15] A. Heise, J. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. *PVLDB*, 7(4):301–312, 2013.

[16] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

[17] H. Köhler, U. Leck, S. Link, and H. Prade. Logical foundations of possibilistic keys. In *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, pages 181–195, 2014.

[18] H. Köhler, U. Leck, S. Link, and X. Zhou. Possible and certain keys for SQL. *VLDB J.*, 25(4):571–596, 2016.

[19] H. Köhler and S. Link. SQL schema design: Foundations, normal forms, and normalization. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 267–279. ACM, 2016.

[20] H. Köhler and S. Link. Inclusion dependencies and their interaction with functional dependencies in SQL. *J. Comput. Syst. Sci.*, 85:104–131, 2017.

[21] H. Köhler, S. Link, and X. Zhou. Possible and certain SQL key. *PVLDB*, 8(11):1118–1129, 2015.

[22] H. Köhler, S. Link, and X. Zhou. Discovering meaningful certain keys from incomplete and inconsistent relations. *IEEE Data Eng. Bull.*, 39(2):21–37, 2016.

[23] X. Liang, S. Li, S. Zhang, H. Huang, and S. X. Chen. Pm2. 5 data reliability, consistency, and air quality assessment in five chinese cities. *Journal of Geophysical Research: Atmospheres*, 121(17), 2016.

[24] H. Mannila, H. Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data mining and knowledge discovery*, 1(3):259–289, 1997.

[25] C. T. N. de Bruijn and D. Kruyswijk. On the set of divisors of a number. *Nieuw Arch. Wisk.*, 23:191–193, 1951.

[26] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment*, 8(10):1082–1093, 2015.

[27] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 821–833, 2016.

[28] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data*, pages 821–833. ACM, 2016.

[29] T. Papenbrock and F. Naumann. A hybrid approach for efficient unique column combination discovery. In *BTW*, pages 195–204, 2017.

[30] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. Gordian: efficient and scalable discovery of composite keys. In *Proceedings of the 32nd international conference on Very large data bases*, pages 691–702. VLDB Endowment, 2006.

[31] Z. Wei, S. Link, and J. Liu. Contextual keys. In *Conceptual Modeling - 36th International Conference, ER 2017, Valencia, Spain, November 6-9, 2017, Proceedings*, pages 266–279, 2017.

Algorithm 7

1: **INPUT:** eUC-tree T_Σ over R, NU (M, N), EH M_1, UH M_2, number l of validated levels
2: **OUTPUT:** An updated eUC-tree T_Σ
3: paths $\leftarrow$ {eUC-paths in T_Σ subsumed by (M, N)}
4: **for** each $(E, U) \in$ paths **do**
5: $\quad\mathcal{P} \leftarrow \emptyset$
6: $\quad$Remove eUC-path (E, U) from T_Σ
7: $\quad\mathcal{E} \leftarrow M_1[(E, U)], \mathcal{U} \leftarrow M_2[(E, U)]$
8: $\quad$Remove (E, U) from M_1, M_2
9: $\quad$**for** $A \in R - (\mathcal{E} \cup E)$ **do** ▷ $\mathcal{E}$-attributes no longer required
10: $\quad\quad$**if** $(EA, U) \sqsupseteq (E', U')$ in T_Σ with $|U'| \leq l$ **then**
11: $\quad\quad\quad\mathcal{E} \leftarrow \mathsf{E} \cup \{A\}, \mathcal{U} \leftarrow \mathsf{U} \cup \{A\}$
12: $\quad\quad\quad$*continue* ▷ Goto line 9
13: $\quad\quad$**if** $T_\Sigma = \emptyset$ **then**
14: $\quad\quad\quad T_\Sigma \leftarrow$ a root node of an eUC-tree
15: $\quad\quad$Insert (EA, U) as a new eUC-path into T_Σ
16: $\quad\quad\mathcal{P} \leftarrow \mathcal{P} \cup \{(EA, U)\}$
17: $\quad$**for** $A \in E - (\mathcal{U} \cup U)$ **do** ▷ $\mathcal{U}$-attributes no longer required
18: $\quad\quad$**if** $(EA, UA) \sqsupseteq (E', U')$ in T_Σ with $|U'| \leq l$ **then**
19: $\quad\quad\quad\mathcal{U} \leftarrow \mathsf{U} \cup \{A\}$
20: $\quad\quad\quad$*continue* ▷ Goto line 17
21: $\quad\quad$**if** $T_\Sigma = \emptyset$ **then**
22: $\quad\quad\quad T_\Sigma \leftarrow$ a root node of an eUC-tree
23: $\quad\quad$Insert (EA, UA) as a new eUC-path into T_Σ
24: $\quad\quad\mathcal{P} \leftarrow \mathcal{P} \cup \{(EA, UA)\}$
25: $\quad$**for** each $(E', U') \in \mathcal{P}$ **do**
26: $\quad\quad M_1[(E', U')] \leftarrow \mathcal{E}$, $M_2[(E', U')] \leftarrow \mathcal{U}$

A Examples

A.1 Column-efficient Algorithm

Example 2 *We illustrate how Algorithm 2 discovers the eUCs that hold in Table 13 over* $R = \{E, D, M\}$*. Comparing distinct pairs of tuples, we compute*

$$\Sigma^{-1} = \{(E, E), (EDM, DM), (E, \emptyset)\}.$$

In the first iteration (starting from line 4), $\Omega = \{(\emptyset, \emptyset)\}$ *since only an initial eUC* $(\emptyset, \emptyset)$ *is in* T_Σ*. For each eUC in* Ω *new eUCs are generated by adding extra attributes to its extension or UC, that is eUCs* $(\emptyset \cup \{D\}, \emptyset)$ *and* $(\emptyset \cup \{M\}, \emptyset)$ *are generated since* $M, D \in R - \{E\}$*. In the second iteration,* $\Omega = \{(D, \emptyset), (M, \emptyset)\}$*. To compute new eUCs from* $(D, \emptyset)$*, a new eUC* (ED, E) *is generated and added to* T_Σ*. From* $(M, \emptyset)$*, a new eUC* (EM, E) *is generated and added to* T_Σ*. For the last iteration,* $\Omega = \{\}$*, and the algorithm terminates. The discovered eUCs from Table 13 are* (ED, E) *and* (EM, E)*.*

Table 13: An example relation

E(mployee)	*D(epartment)*	*M(anager)*
Homer	Toys	Burns
Homer	$\perp$	$\perp$
Marge	Toys	Burns

Heuristics to improve performance of the column-efficient algorithm. In addition to using an eUC-tree to minimize redundancy among the set of discovered eUCs, we further propose a heuristic which drastically improves the running time of Algorithm 2.

The order in which MNUs are processed affects the running time of Algorithm 2. Some MNUs generate more eUCs that are not in the final result than other MNUs. For example, consider the set

$$\Sigma^{-1} = \{(A, A), (BC, B), (ABC, C)\}$$

over $R = \{A, B, C\}$. If Algorithm 2 processes Σ^{-1} in the order of (A, A), (BC, B), (ABC, C), the generated eUCs are $(B, \emptyset)$, $(C, \emptyset)$, (BC, C), $(AB, \emptyset)$, (C, C), $(AC, \emptyset)$, (AB, A), (AB, B), (AC, A), (ABC, B), (AC, AC), (BC, BC). If the processing order is $(ABC, C), (BC, B), (A, A)$ instead, the generated eUCs become (A, A), (B, B), (BC, BC), (AB, B), (AC, A), (AB, A). So, the first order generates 12 eUCs but the second only 6 eUCs. In this paper, we will prioritize the MNUs with larger extensions. The intuition behind this heuristic comes from two observations. Firstly, the cardinalities of extensions and uniques keep increasing as more MNUs are processed. This way, MNUs with lower priority generate less eUCs. Secondly, line 17 adds a new attribute to both extension and associated UC, but line 12 only adds a new attribute to the extension. So, an MNU with larger extension could identify more new attributes which can be added to an extension and associated UC at the same time. Such MNUs are more likely to generate final eUCs faster. The proposed solution has considerably improved the running time of Algorithm 2. More research on the order of processing MNUs is outside the scope for this paper.

A.2 Row-efficient Algorithm

We now show how the row-efficient algorithm discovers eUCs from the relation r in Table 13. Note that we denote the first, second, and third tuple in Table 13 as t_1, t_2, t_3 respectively. First of all, there is no eUC in r whose UC is an empty set since $\pi_\emptyset(r) = \{r\}$ and $|r^R \cap r| = 2$ (Proposition 2). In other words, there is no extension for $\emptyset$ to form an eUC satisfied by r. So, the initial set of UC candidates are $\{E, D, M\}$. Before starting a discovery, we pre-compute the stripped partitions for a given set of UC candidates, i.e., $\pi_E(r) = \{\{t_1, t_2\}\}$, $\pi_D(r) = \{\{t_1, t_3\}\}$, and $\pi_M(r) = \{\{t_1, t_3\}\}$. On level 1, all UC candidates need to have larger extensions so that they may become valid eUCs (tested by line 14). Moreover, only attribute E can have an extension with a valid eUC (tested by line 18). At this point, Algorithm 4 will start with E and $\pi_E(r)$ (line 20). Here, the algorithm only takes extra attributes which are possible to form extensions for E as extension candidates, i.e., currentLevel $= \{M, D\}$. This way, the attribute lattice of

extensions for E becomes smaller. Algorithm 4 stops at the first iteration because M and D form minimal extensions for E, respectively. Returning to Algorithm 3, newly discovered eUCs (EM, E) and (ED, E) are stored in T_Σ (line 25). At the end of examining level 1, attributes D and M will be used to generate UC candidates on level 2 (line 27), i.e., DM. In addition, Algorithm 5 can be used to compute $\pi_{DM}(r)$ given $\pi_D(r)$ and M, i.e., $\pi_{DM}(r) = \{\{t_1, t_3\}\}$. On level 2, the only UC candidate DM cannot form any valid eUC and the row-efficient algorithm terminates. Therefore, the discovered eUCs are (ED, E) and (EM, E).

B Hybrid Algorithms

B.1 Pruning the Candidate Generation

Our first hybrid, Algorithm 7, updates the current eUC-tree T_Σ based on some NU (M, N), EH M_1, UH M_2, and the number l of levels for which eUCs with associated UC $|U| \leq l$ have been validated. The main speed ups are achieved by line 9 and line 17, where attributes in EH and UH do no longer need to be considered when new candidate eUCs are generated for the next level. Moreover, Algorithm 7 also updates the given EH and UH for each eUC in the eUC-tree whenever a new candidate eUC is found to be implied by some valid eUC (line 11). Note that l indicates that any eUC (E, U) in the given eUC-tree where $|U| \leq l$ is already known to hold on the given relation.

B.2 Hybrid e-traversal

Algorithm 8 summarizes the techniques for our hybrid e-traversal. Based on an input UC U for which it has been validated that some extension E exists for which (E, U) is valid on the given input relation, and based on an antecedent tree $A_\mathcal{E}$ that represents all candidate extensions, Algorithm 8 validates all extensions in $A_\mathcal{E}$ level by level. However, the antecedent tree needs to be updated iteratively because it does not contain all the extensions after all. Instead of using prefix blocks like Algorithm 4, the hybrid e-traversal algorithm uses invalid extensions to construct NUs, such that Algorithm 8 can update the antecedent tree with a procedure that is similar to that in Algorithm 7. Meanwhile, the input eUC-tree T_Σ, EH and UH are also updated when an NU is processed. Note that EH and UH can be updated if and only if an invalid extension is implied by extensions on the previous levels, namely by the valid eUCs (line 20 and 29). If the ratio between the number of invalid extensions and number of possible extensions on the current level exceeds some threshold (0.01 found to be most suitable in our experiments), see line 11, then the invalid extensions would generate too many candidate extensions for the next level. In that case, the sampling of tuple pairs from our input stripped partition can generate fewer candidate extensions (resulting in larger embedded non-uniques that subsume multiple embedded non-uniques generated by the invalid extensions). We discuss details about our sampling method at the end of this section.

Algorithm 8 Hybrid e-traversal

1: **INPUT:** Associated UC U, eUC-tree T_Σ, antecedent tree $A_\mathcal{E}$, stripped partition $\pi_U(r)$ of relation r, EH M_1, UH M_2
2: **OUTPUT:** Antecedent tree $A_\mathcal{E}$ with all extensions of U
3: $l \leftarrow 1$
4: currentLevel $\leftarrow$ {paths of length l in $A_\mathcal{E}$}
5: **while** the length of maximal path in T_Σ is at least l **do**
6: nus $\leftarrow \emptyset$
7: **for** each $E \in$ currentLevel **do**
8: **if** $|r^E \cap S| \leq 1$ for all $S \in \pi_U(r)$ or $r^E = \emptyset$ **then**
9: *continue* ▷ Goto line 7
10: nus $\leftarrow$ nus $\cup \{E\}$ ▷ Invalid extensions
11: **if** |nus|/|currentLevel| > 0.01 **then** ▷ Reduce non-uniques
12: Sample pairs in S for all $S \in \pi_U(r)$
13: nus $\leftarrow$ max(nus $\cup$ {NUs from sampling})
14: **for** each $(M, N) \in$ nus **do**
15: paths $\leftarrow$ {paths in $A_\mathcal{E}$ subsumed by M}
16: **for** each $E \in$ paths **do**
17: Remove E from $A_\mathcal{E}$
18: $\mathcal{P} \leftarrow \emptyset$, $\mathcal{E} \leftarrow M_1[(E, U)]$, $\mathcal{U} \leftarrow M_2[(E, U)]$
19: Remove (E, U) from M_1, M_2
20: **for** each $A \in R - (\mathcal{E} \cup E)$ **do**
21: **if** EA subsumes any E' in $A_\mathcal{E}$ where $|E'| < l$ or
22: (EA, U) subsumes any (E', U') in T_Σ where $|E'| < |E|$ **then**
23: $\mathcal{E} \leftarrow$ E $\cup \{A\}$, $\mathcal{U} \leftarrow$ U $\cup \{A\}$
24: *continue* ▷ Goto line 20
25: **if** $A_\mathcal{E} = \emptyset$ **then**
26: $A_\mathcal{E} \leftarrow$ a root node of an antecedent tree
27: Insert EA as a new path into $A_\mathcal{E}$
28: $\mathcal{P} \leftarrow \mathcal{P} \cup \{(EA, U)\}$
29: **for** each $A \in E - (\mathcal{U} \cup U)$ **do**
30: **if** EA subsumes any E' in $A_\mathcal{E}$ where $|E'| < l$ or
31: (EA, UA) subsumes any (E', U') in T_Σ where $|U'| < |U|$ **then**
32: $\mathcal{U} \leftarrow$ U $\cup \{A\}$
33: *continue* ▷ Goto line 29
34: **if** $T_\Sigma = \emptyset$ **then**
35: $T_\Sigma \leftarrow$ a root node of an eUC-tree
36: Insert (EA, UA) as a new eUC-path into T_Σ
37: **for** each $(E', U') \in \mathcal{P}$ **do**
38: $M_1[(E', U')] \leftarrow \mathcal{E}$, $M_2[(E', U')] \leftarrow \mathcal{U}$
39: $l := l + 1$
40: currentLevel $\leftarrow$ {paths of length l in $A_\mathcal{E}$}
41: **return** $A_\mathcal{E}$

B.3 Hybrid u-traversal

We now combine Algorithm 7 and Algorithm 8 to obtain Algorithm 9, which is our hybrid u-traversal algorithm. Here, we examine associated UCs level by level, traversing our eUC-tree while counting the number of visited u-nodes. If all eUCs with a fixed associated UC have been validated, they must be non-redundant because both line 18 and 21 perform redundancy checks before inserting a new eUC. Otherwise, Algorithm 8 is used to discover all minimal extensions of the associated UC (line 15). Meanwhile, a set of NUs, an EH and a UH are updated during hybrid e-traversal. Since NUs acquired from hybrid e-traversals are used to update antecedent trees (Alg. 8, line 27) or the next level of the eUC-tree (Alg. 8, line 36), they also update the part of the eUC-tree which has not been validated so far.

Algorithm 9 Hybrid u-traversal

1: **INPUT:** A relation r over relation schema R
2: **OUTPUT:** The eUC-tree T_Σ representing a minimal cover Σ of those eUCs that hold on r
3: $T_\Sigma \leftarrow$ eUC-tree with e-node A for all $A \in R$
4: $M_1[(A, \emptyset)] = \emptyset$ for all $A \in R$ ▷ Initial EH
5: $M_2[(A, \emptyset)] = \emptyset$ for all $A \in R$ ▷ Initial UH
6: $l \leftarrow 0$
7: currentLevel $\leftarrow \{(E, U) \mid (E, U)$ is eUC-path with $|U| = l\}$
8: **while** there is eUC-path (E, U) in T_Σ where $|U| >= l$ **do**
9: nus $\leftarrow \emptyset$
10: **for** each U such that $(E, U) \in$ currentLevel **do**
11: $\mathcal{E} \leftarrow \{E \mid (E, U) \in$ currentLevel$\}$
12: **if** $\exists E \in \mathcal{E}$ where $|r^E \cap S| \leq 1$ for all $S \in \pi_U(r)$ **then**
13: $A_\mathcal{E} \leftarrow$ Antecedent tree with paths in $\mathcal{E}$
14: Remove eUC-path (E, U) from T_Σ for all $E \in \mathcal{E}$
15: $\mathcal{E}' \leftarrow \mathsf{hyE} - \mathsf{Traversal}(U, T_\Sigma, A_\mathcal{E}, \pi_U(r), M_1, M_2)$
16: Add NUs used in the hybrid e-traversal to nus
17: **for** each $E \in \mathcal{E}'$ **do**
18: **if** (E, U) non-redundant in T_Σ **then**
19: Insert (E, U) as a new eUC-path to T_Σ
20: **for** each $(M, N) \in$ nus **do**
21: updateEUCTree$(T_\Sigma, (M, N), M_1, M_2, l)$ ▷ Algorithm 7
22: $l \leftarrow l + 1$
23: currentLevel $\leftarrow \{(E, U) \mid (E, U)$eUC-path with $|U| = l\}$
24: **return** T_Σ

B.4 Correctness sketch

The main argument is similar to that of Theorem 5, because our hybrid algorithms only use NUs to update the eUC-tree. NUs are generated by either invalidating an eUC

or by extracting samples from a stripped partition. Nevertheless, the algorithm still follows the characterization of valid eUCs by MNUs, see Theorem 4. Our algorithms further use Proposition 1 to validate eUCs level by level. That way, EHs and UHs can be computed and employed by Algorithm 7, which is just a more efficient implementation of Algorithm 2. In Algorithm 7, whenever an eUC (E, U) is subsumed by an NU (M, N), new eUCs will be generated regarding the NU. Among the new eUCs, some of them may be implied by eUCs that have already been validated. So, for those new eUCs which are not implied but generated by the same eUC, they cannot be extended by the attributes which have been used to generate the invalid eUCs. For instance, if (EA, U) is implied, but (EB, U) is not implied by a set of valid eUCs, such that $A, B \in R - M$, then an EH will map (EB, U) to an attribute set containing A, indicating that (EAB, U) will result in an implied eUC since $(EA, U) \sqsubseteq (EAB, U)$. Lastly, since extensions and their associated UCs are enumerated by cardinalities, all minimal eUCs that hold on a given relation will be generated.

B.5 Sampling

We conclude our discussion of hybrid algorithms by giving details about the sampling of NUs from a stripped partition. In efficiently exploring the antecedent tree in Algorithm 8, we need to use NUs appropriately to identify invalid extensions of a given UC, so that fewer new extensions are generated at each level. Let $\pi_U(r)$ be a stripped partition where r is the input relation over R. Take any distinct $t_1, t_2 \in S$ where $S \in \pi_U(r)$. An NU (M, N) can be computed by comparing t_1 and t_2, i.e., $M = \{A \in R \mid t_1(A) \neq \perp \neq t_2(A)\}$ and $N = \{A \in R \mid t_1(A) = t_2(A) \neq \perp\}$. If there is some $E \subseteq R$ such that $(E, U) \sqsubseteq (M, N)$, t_1 and t_2 form a witness pair showing that (E, U) is invalid. Meanwhile, (EA, U) may be valid for all $A \in R - M$, because t_1 or t_2 may not belong to r^{EA}. Therefore, the antecedent tree of extensions can be updated by NUs; and NUs with larger extensions generate fewer new extensions in the tree. Note that it is unnecessary to compare tuples from different partitions since they do not have matching values on U. Eventually, we want to sample pairs of tuples by a small number of comparisons such that NUs with larger extensions can be found. Algorithm 10 summarizes our progressive sampling method.

Firstly, the algorithm ranks tuples in each partition by the decreasing number of complete values they hold. Secondly, each tuple is compared to all tuples of lower rank. Since only the maximal elements of the discovered NUs matter, we measure how frequent new NUs are discovered. If the frequency is below some certain threshold, the algorithm stops sampling and returns the sampled NUs. Thirdly, several variables are maintained by the algorithm, including the current sampling progress, the total number of discovered NUs, the number of comparisons, and a threshold which stops the sampling process. The frequency in line 3 denotes the ratio of the total number of discovered NUs over the number of comparisons. It indicates the likelihood that some new, not already subsumed NU can be found by additional comparisons. Such measurement cannot guarantee that there are no more new NUs, so we divide the current threshold by 2 in line 19, just before sampling is aborted. This is to ensure that sampling can carry on when the algorithm is invoked again. Every time sampling is aborted, it is assumed that there are no additional NUs. However, the sampling algorithm may be invoked again when there

are too many invalid extensions, which means the previous assumption was inaccurate and the threshold should be decreased such that additional NUs can be found. Lastly, we can further improve the efficiency of the sampling algorithm by one simple optimization. Let $S = \{t_1, t_2, t_3, \ldots, t_n\} \in \pi_U(r)$. Since the algorithm is only invoked after it has been verified that there is some valid extension for U, at most one tuple in S can be complete, and this tuple will be t_1 because S is sorted accordingly. Suppose (M_1, N_1) and (M_2, N_2) were generated from t_1, t_2 and t_2, t_3, respectively. Let $V_i = \{A \in R \mid t_i(A) \neq \perp\}$. So, $M_2 \subseteq V_2, V_3$ since N_2 consists of common attributes on which t_2 and t_3 are complete. Furthermore, $N_2 \subseteq N_1$ because $V_1 = R$ and $V_1 \cap V_2 = M_1$. Therefore, if there is a complete tuple in a partition, our sampling algorithm only needs to retrieve the NUs which are generated using the complete tuple, since others will be subsumed by these NUs.

Algorithm 10 NUs sampling

INPUT: Stripped partition $\pi_U(r)$, a set nus of NUs, the sampling progress $p \in \mathbb{N}$, the total number n of discovered NUs, the number c of comparisons, a threshold t
OUTPUT: A set of NUs
frequency $\leftarrow n/c$
while frequency $>= t$ **do**
  for each $S \in \pi_U(r)$ **do**
    if $p >= |S| - 1$ **then**
      continue
    if $p > 0$ and $S[0] \in r^R$ **then**
      continue
    $i \leftarrow p + 1$
    while $i < |S|$ **do**
      $c \leftarrow c + 1$
      $M = \{A \mid S[p](A) \neq \perp \neq S[i](A)\}$
      $N = \{A \mid S[p](A) = S[i](A) \neq \perp\}$
      if $\neg\exists(M', N') \in$ nus$((M', N') \sqsubseteq (M, N))$ **then**
        $n \leftarrow n + 1$
        nus $\leftarrow \max($nus $\cup \{(M, N)\})$
  $p \leftarrow p + 1$
$t \leftarrow t/2$
return cnks

C Armstrong Samples

Computing Maximal Non-Uniques. Computing maximal non-uniques can take two approaches. One approach is to compute a maximal set of non-uniques by comparing all pairs of tuples in a given relation. Considering data sets with a large number of rows, the other approach computes maximal non-uniques given the set of minimal eUCs of a given relation.

Algorithm 11 Computing maximal non-uniques

INPUT: A set Σ of eUCs over a relation schema R
OUTPUT: Σ^{-1}
$\Sigma \leftarrow \Sigma \cup \{(R, R)\}$, $\Sigma' = \emptyset$;
$\Sigma^{-1} \leftarrow \{(R, R)\}$;
for each $(E, U) \in \Sigma$ **do**
    $\Sigma' \leftarrow \Sigma' \cup \{(E, U)\}$
    for each $(M, N) \in \Sigma^{-1}$ **do**
        if $(E, U) \sqsubseteq (M, N)$ **then**
            $\Sigma^{-1} \leftarrow \Sigma^{-1} - \{(M, N)\}$
            for each $A \in E - U$ **do**
                $\Sigma^{-1} \leftarrow \Sigma^{-1} \cup \{(M - \{A\}, N - \{A\})\}$
            for each $A \in U$ **do**
                $\Sigma^{-1} \leftarrow \Sigma^{-1} \cup \{(M, N - \{A\})\}$
    for each $(M, N) \in \Sigma^{-1}$ **do**
        if $\exists A \in M - N\ \forall (E', U') \in \Sigma : (E', U') \not\sqsubseteq (M, NA)$ or
            $\exists A \in R - M\ \forall (E', U') \in \Sigma : (E', U') \not\sqsubseteq (MA, N)$ **then**
            $\Sigma^{-1} \leftarrow \Sigma^{-1} - \{(M, N)\}$;
return Σ^{-1};

Algorithm 11 demonstrates the second approach, which iteratively refines a set of non-uniques, given the set of minimal eUCs on a given relation. If any of the non-uniques are redundant, attributes in the intersection with the eUCs can be removed so that a non-unique will become non-redundant. At the end of an iteration, the algorithm checks the maximality of the non-uniques with respect to the examined eUCs.

Another Example. The Armstrong samples for *breast-cancer-wisconsin* in Table 14 further illustrate our original motivation for eUCs. The two rows with value 733,639 on column 0 are identical, except for column 6, where one row has value 1 and the other an occurrence of $\perp$. Since UCCs interpret $\perp$ to be different from other domain values, the UCC {0,1,4,6,7} is valid on the data set. It is doubtful that $\perp$ represents a value different from 1, but more likely that both rows are duplicates. Indeed, all other rows are already uniquely identifiable by {0,1,4,7}. This illustrates again the robustness of eUCs: Instead of inflating our UC by column 6, we add column 6 to the extension of the corresponding eUC giving $(\{0, 1, 4, 6, 7\}, \{0, 1, 4, 7\})$.

D Proofs

D.1 Computational Complexity

Let I be an instance of decision problems of class $\mathcal{D}$ and a natural number parameter k. (I, k) denotes the corresponding instance of *fixed-parameter* problem of $\mathcal{D}$. The fixed-parameter problem of class $\mathcal{D}$ is *fixed-parameter tractable* if a given instance (I, k) can be solved in time $\mathcal{O}(f(k) \cdot p(|I|))$ where p is a polynomial function. We use FPT to denote the

0	1	2	3	4	5	6	7	8	9	10
1067444	2	1	1	1	2	1	2	1	1	2
1036172	2	1	1	1	2	1	2	1	1	2
1212422	4	1	1	1	2	1	3	1	1	2
1212422	3	1	1	1	2	1	3	1	1	2
1182404	3	1	1	1	2	1	2	1	1	2
1182404	3	1	1	1	2	1	1	1	1	2
733639	3	1	1	1	2	1	3	1	1	2
733639	3	1	1	1	2	⊥	3	1	1	2
734111	1	1	1	1	2	2	1	1	1	2
734111	1	1	1	3	2	3	1	1	1	2
654546	1	1	1	3	2	1	1	1	1	2
654546	1	1	1	1	2	1	1	1	8	2

0	1	2	3	4	5	6	7	8	9	10
1067444	2	1	1	1	2	1	2	1	1	2
1036172	2	1	1	1	2	1	2	1	1	2
1212422	4	1	1	1	2	1	3	1	1	2
1212422	3	1	1	1	2	1	3	1	1	2
733639	3	1	1	1	2	1	3	1	1	2
733639	3	1	1	1	2	⊥	3	1	1	2
1182404	3	1	1	1	2	1	2	1	1	2
1182404	3	1	1	1	2	1	1	1	1	2
654546	1	1	1	3	2	1	1	1	1	2
654546	1	1	1	1	2	1	1	1	8	2

Table 14: Armstrong sample of *bcw* with eUCs (left): $(\{6\}, \{0, 1, 4, 7\})$, $(\emptyset, \{0, 1, 6, 7, 9\}$, and UCCs (right): $\{0, 1, 4, 6, 7\}$, $\{0, 1, 6, 7, 9\}$

class of fixed-parameter tractable problems. Let $\mathcal{D}$ and $\mathcal{D}'$ be two classes of parameterized problem. A *parameterized reduction* from $\mathcal{D}$ to $\mathcal{D}'$ is a fixed-parameter tractable algorithm that computes a corresponding instance of $\mathcal{D}'$ for any given instance of class $\mathcal{D}$ and the parameters of matching instances are only dependent on each other. We use $\mathcal{D} \leq_{FPT} \mathcal{D}'$ to denote if there is a fixed-parameter reduction from $\mathcal{D}$ to $\mathcal{D}'$. Moreover, we say $\mathcal{D}$ and $\mathcal{D}'$ are FPT-equivalent if $\mathcal{D} \leq_{FPT} \mathcal{D}'$ and $\mathcal{D}' \leq_{FPT} \mathcal{D}$. Parameterized problems are divided into a hierarchy of complete complexity classes by the problem WEIGHTED t-NORMALIZED SATISFIABILITY [10] i.e. $\mathsf{FPT} \subseteq \mathcal{W}[1] \subseteq \mathcal{W}[2] \ldots$.

Theorem 8 (Theorem 1 restated)
The problem EUC *is* NP*-complete.*

Proof EUC is in NP, because we can guess (E, U) with $|E| \leq k$ and verify in polynomial time in the input that r satisfies (E, U). For the NP-hardness, we reduce KEY to EUC. Take an instance (r, k) of KEY where r is a complete relation over relation schema R, and k is a positive integer. Let (r', k') be the instance of EUC where $r' = r$, and $k' = k$. Now it follows that a key $K \subseteq R$ with $|K| \leq k$ is satisfied by r if and only if the key K is satisfied by $r = r^K$ if and only if the eUC (K, K) is satisfied by $r' = r$.

Theorem 9 (Theorem 2 restated)
The problem EUC *is* W[2]*-complete.*

Proof We show that KEY and EUC are equivalent under FPT-reductions. The result then follows from the W[2]-completeness of KEY shown in [5].

For KEY $\leq_{FPT}$ EUC we observe that the PTIME reduction in the proof of Theorem 1 is actually an FPT-reduction, since the parameter k' only depends on k.

It remains to show that EUC $\leq_{FPT}$ KEY holds as well. Let (r, k) be an instance of EUC. We transform (r, k) into an instance (r', k') by defining r' as the result of replacing null marker occurrences in r with unique column values in r', and defining k' to be k. Clearly, this transformation is FPT. If there is some eUC (E, U) over R with $|E| \leq k$ that is satisfied by r, then for any two tuples $t, t' \in r'$ with $t \neq t'$ and $t[U] = t'[U]$ it would follow that $t, t' \in r^E$ - a contradiction to (E, U) being satisfied by r. Consequently, for any two tuples $t, t' \in r'$ with $t \neq t'$ we have $t[U] \neq t'[U]$, which means that r' satisfies the

key U with $U \subseteq E$ and thus $|U| \leq |E| \leq k = k'$. Vice versa, if U is a key with $|U| \leq k'$ that is satisfied by r', then r satisfies the eUC (U, U) with $|U| \leq k' = k$. Indeed, if there were $t, t' \in r^U$ with $t \neq t'$ and $t[U] = t'[U]$, then U would not be satisfied by r'. This concludes the proof.

D.2 Maximum Solution Space

Definitions of anti-chain, symmetric chain order, etc. can all be found in Engel's book [11].

Theorem 10 (Theorem 3 restated) *Let R be a finite set, and let $\mathcal{F} \subseteq 2^R \times 2^R$ such that for all $(E, U) \in \mathcal{F}$:*

(i) $U \subseteq E$ and

(ii) there is no $(E', U') \in \mathcal{F} - \{(E, U)\}$ with $(E', U') \sqsubseteq (E, U)$.

Then $|\mathcal{F}| \leq W(|R|)$, where for $|R| \geq 2$ equality is attained if and only if

$$\mathcal{F} = \left\{(E, U) \in 2^R \times 2^R : U \subseteq E \text{ and } |E| + |U| = |R|\right\}.$$

Proof Let P be the set of all $(E, U) \in 2^R \times 2^R$ that satisfy condition (i), and consider the partial order on P defined by:

$$(E', U') \leq (E, U) :\Longleftrightarrow E' \subseteq E \wedge U' \subseteq U.$$

Clearly, P is ranked, where the rank of an element (E, U) equals $|E| + |U|$. Let $n = |R|$. We have to show that any anti-chain $\mathcal{A}$ in P has size at most $W(n)$ and that equality is attained if and only if $\mathcal{A}$ is the n-th level of P.

Note that P is isomorphic to the n-th Cartesian power of the chain $0 < 1 < 2$. The corresponding isomorphism$\varphi : P \mapsto \{0, 1, 2\}^n$ is given by

$$\varphi(C, K) = (a_1, a_2, \ldots, a_n) \text{ with } a_i = \begin{cases} 0 & \text{if } i \notin E, \\ 1 & \text{if } i \in E - U, \\ 2 & \text{if } i \in U. \end{cases}.$$

It is well-known that chain products are symmetric chain orders which was first shown in [25]. It follows immediately that the unique largest anti-chain in P is its middle level (see also Theorem 5.1.4 in [11]), i.e. the n-level whose cardinality is equal to $W(n)$.

D.3 Column-efficient Discovery

Theorem 11 (Theorem 4 restated) *Let r be a relation over R. An eUC (E, U) is satisfied by r if and only if there is no $(E', U') \in \Sigma^{-1}$ such that $(E, U) \sqsubseteq (E', U')$.*

Proof Suppose there is some $(E', U') \in \Sigma^{-1}$ such that $(E, U) \sqsubseteq (E', U')$. Then there are distinct $t_1, t_2 \in r^{E'}$ such that $t_1(U') = t_2(U')$ holds. Since $E \subseteq E'$ holds, $r^{E'} \subseteq r^E$.

Since also $U \subseteq U'$ holds, there are distinct $t_1, t_2 \in r^E$ such that $t_1(U) = t_2(U)$ holds. Consequently, r does not satisfy (E, U).

Suppose r does not satisfy (E, U). Then there are distinct $t_1, t_2 \in r^E$ such that $t_1(U) = t_2(U)$ holds. Let

$$E'' := \{A \in R \mid t_1(A) \text{ and } t_2(A) \text{ are total}\},$$

and

$$U'' := \{A \in E'' \mid t_1(A) = t_2(A)\}.$$

It follows that $E \subseteq E''$ and $U \subseteq U''$. Hence, (E'', U'') is an embedded non-unique of r. Consequently, there must be some $(E', U') \in \Sigma^{-1}$ such that $E \subseteq E'' \subseteq E'$ and $U \subseteq U'' \subseteq U'$. Hence, there is some $(E', U') \in \Sigma^{-1}$ such that $(E, U) \sqsubseteq (E', U')$ holds.

Theorem 12 (Theorem 5 restated)
Given the set of maximal embedded non-uniques of a relation, Algorithm 2 computes a minimal cover of the set of eUCs that are satisfied by the relation.

Proof Let r be the given relation over the given relation schema R, and let Σ^{-1} denote the set of maximal embedded non-uniques of r. Let Σ denote the minimal cover of the set of eUCs that are represented by the eUC-tree T_Σ that Algorithm 2 returns. We prove the correctness by induction over the cardinality of Σ^{-1}.

Base case: Here, $|\Sigma^{-1}| = 0$. So, Algorithm 2 simply returns only a root node that represents the set $\Sigma = \{(\emptyset, \emptyset)\}$. Indeed, if there are no maximal embedded non-uniques, then there are no embedded non-uniques at all, which means that every eUC is satisfied by r. The unique minimal cover in this case is indeed the set Σ since $(\emptyset, \emptyset)$ is subsumed by every possible eUC. This concludes the base case.

Inductive steps: Here, $\Sigma^{-1} = \Gamma^{-1} \cup \{(M, N)\}$, where Γ^{-1} denotes the set of maximal embedded non-uniques for some relation $r' \subseteq r$. We denote by Γ a minimal cover of the set of eUCs that hold on r'. By induction hypothesis we know that, on input Γ^{-1}, Algorithm 2 returns a eUC-tree T_Γ that represents Γ. That is, a') r' satisfies all eUCs (E, U) represented by T_Γ, and b') if r' satisfies some eUC (E, U), then there is some eUC $(E', U') \in \Gamma$ such that $(E', U') \sqsubseteq (E, U)$ holds.

We will show now that a) r satisfies all eUCs (E, U) represented by T_Σ, and b) if r satisfies some eUC (E, U), then there is some eUC (E', U') in Σ such that $(E', U') \sqsubseteq (E, U)$ holds.

For a), we first look at the case where $(E, U) \in \Gamma$. Then $(E, U) \not\sqsubseteq (M, N)$ since the eUC-path (E, U) would not be represented by T_Σ because of line 7. Because of a') we also know that there is no $(E', U') \in \Gamma^{-1}$ such that $(E, U) \sqsubseteq (E', U')$ holds. Consequently, there is no $(E', U') \in \Sigma^{-1} = \Gamma^{-1} \cup \{(M, N)\}$ represented by T_Σ such that $(E, U) \sqsubseteq (E', U')$ holds. By Theorem 4, r satisfies (E, U). We will now look at the case where $(E, U) \notin \Gamma$. Consequently, (E, U) must be one of the eUCs that were added by line 12 or by line 17, so (E, U) results by the augmentation of some $(E', U') \in \Omega$ according to line 5, that is, $(E', U') \sqsubseteq (M, N)$. However, (E, U) can therefore not be

subsumed by (M, N), and cannot be subsumed by any maximal embedded non-unique in Γ^{-1}. Indeed, if (E, U) was subsumed by some maximal embedded non-unique in Γ^{-1}, then so would be (E', U'), which would be a contradiction to r' satisfying (E', U') is in Γ. Consequently, (E, U) is not subsumed by any maximal embedded non-unique in Σ^{-1}. By Theorem 4, r satisfies (E, U). This shows a).

For b), we first observe that whenever r satisfies (E, U), then $(E, U) \not\sqsubseteq (M, N)$ by Theorem 4, since $(M, N) \in \Sigma^{-1}$. Consequently, there is some attribute $A \in E - M$ or some attribute $A \in (U \cap E) - N$, which we denote as property (P) for easier reference later. Furthermore, whenever r satisfies (E, U), then r' satisfies (E, U) as well since $r' \subseteq r$. By b'), there is some $(E', U') \in \Gamma$ such that $(E', U') \sqsubseteq (E, U)$ holds. If $(E', U') \in \Sigma$, then the proof is completed. Otherwise, $(E', U') \in \Omega$ by line 5 and, in particular, $(E', U') \sqsubseteq (M, N)$. Now, Algorithm 2 adds new eUCs to Σ following line 12 to obtain $(E'A, U')$ for $A \in R - M$ and line 17 to obtain $(E'A, U'A)$ for $A \in M - N$. However, property (P) ensures that some attribute A can be picked from $A \in E - M$ or $A \in (U \cap E) - N$, thereby ensuring that the resulting new eUC is subsumed by (E, U). Consequently, there is some $(E', U') \in \Sigma$ such that $(E', U') \sqsubseteq (E, U)$ holds. This proves b) and completes the proof.

D.4 Row-efficient discovery

Proposition 3 (Proposition 1 restated)
An $\mathsf{eUC}(E, U)$ *over* R *is satisfied by a given relation* r *over* R *if and only if for all* $S \in \pi_U(r)$, $|r^E \cap S| \leq 1$.

Proof If there is some $S \in \pi_U(r)$ such that $|r^E \cap S| \geq 2$, then there are distinct $t, t' \in r^E$ such that $t[U] = t'[U]$. This means, r does not satisfy (E, U).

Vice versa, if there are distinct $t, t' \in r^E$ such that $t[U] = t'[U]$, then there is some $S \in \pi_U(r)$ such that $|r^E \cap S| \geq 2$.

However, the following result also shows how stripped partitions can be used in e-traversals. In effect, we can find an extension E for a given unique U such that the eUC (E, U) holds on r if and only if each stripped partition for U contains at most one total tuple.

Proposition 4 (Proposition 2 restated)
Let $U \subseteq R$, *and* r *a relation over* R. *Then there is some* $E \subseteq R$ *with* $U \subseteq E$ *such that* r *satisfies* (E, U) *if and only if for all* $S \in \pi_U(r)$, $|r^R \cap S| \leq 1$.

Proof Suppose $|r^R \cap S| \leq 1$ for all $S \in \pi_U(r)$. Let

$$E' := \{A \in R - U \mid \exists S \in \pi_U(r) \exists t \in S(t(A) = \perp)\},$$

and let $E := E' \cup U$. Consequently, for all $S \in \pi_U(r)$, $|r^E \cap S| \leq 1$. We conclude by Proposition 1 that r satisfies (E, U).

Vice versa, assume that there is some $S \in \pi_U(r)$ such that $|r^R \cap S| \geq 2$. Then for all $E \subseteq R$ with $U \subseteq E$, $|r^E \cap S| \geq 2$ and r does not satisfy (E, U) by Proposition 1.