Discovery and Ranking of Embedded Uniqueness Constraints

Ziheng Wei
School of Computer Science
The University of Auckland
Auckland, New Zealand
z.wei@auckland.ac.nz

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

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

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). These constraints represents unique column combinations embedded in complete fragments of incomplete data. We showcase their implementation as filtered indexes, and their application in integrity management and query optimization. We show that the decision variant of discovering a minimal eUC is NP-complete and W[2]-complete. We characterize the maximum possible solution size, and show which families of eUCs attain that size. Despite the challenges, experiments with real-world and synthetic benchmark data show that our column(row)-efficient algorithms perform well with a large number of columns(rows), and our hybrid algorithm combines ideas from both. We show how to rank eUCs to help identify relevant eUCs.

PVLDB Reference Format:
Ziheng Wei, Uwe Leck, Sebastian Link. Discovery and Ranking of Embedded Uniqueness Constraints. *PVLDB*, 12(13): 2339-2352, 2019.
DOI: https://doi.org/10.14778/3358701.3358703

1. INTRODUCTION

SQL UNIQUE is the response of the industry standard whenever entity integrity cannot be achieved by primary keys due to missing data values. A table over a column combination R satisfies the SQL UNIQUE constraint (UC) on $U \subseteq R$ whenever every pair of records with no missing values on all columns in U has non-matching values on some column in U. For example, if U consists of *v(oter_id)*, *r(egister_date)*, and *d(ownload_month)*, then the UC U is violated by the data snippet in Table 1. Indeed, t_1, t_2 is a pair of records with matching non-null values on all columns in U. However, if E consists of all the columns in U plus *f(ull_phone_num)*, then the UC E is satisfied. In particular, t_1 and t_2 carry the null marker symbol $\perp$ in column *f*. The record pairs t_3, t_4 and t_5, t_6 and t_7, t_8 show that E is a minimal UC, as none of the proper subsets of E forms a UC that holds on the data snippet. However, the UC E is unable to express the fact that $\{v, d, r\}$ is actually a key on the subset of records that have no missing values on E.

The reason is that SQL UNIQUE uses the same combination of columns to stipulate both completeness and uniqueness requirements. As a consequence, opportunities for better data management remain out of reach. Indeed, it is more natural to de-couple uniqueness from completeness requirements by stipulating that only records with no missing value in every column in E are considered, and the projection of every one of those records to a subset $U \subseteq E$ is unique. We call expressions of the form (E, U) *embedded uniqueness constraints* (eUCs). EUCs stipulate the uniqueness constraint U on the subset of records with no missing values on all the columns in E. SQL UNIQUE represents the special case (E, E) of a general eUC (E, U), that is, where $U = E$. In our example (and writing $E - U$ instead of E), the eUC $(\{f\}, \{d, r, v\})$ holds on the snippet r of Table 1. In fact, the records with no missing values in E are $r^E = r - \{t_1, t_2\}$, and no two different records of r^E have matching values on all the columns of U. Note that the eUC $(\{f\}, \{d, r, v\})$ holds on the entire real-world benchmark data set *ncvoter8m* that has over eight million records.

It is surprising that SQL has not brought forward a notion of a uniqueness constraint that separates completeness from uniqueness concerns. This shortcoming causes a common source of dirty data. This is illustrated by our example where the UC on $\{f, d, r, v\}$ would permit pairs of database records with matching non-null values on *d*, *r*, and *v*, and non-matching non-null values on *f*. This, however, is prevented by the eUC $(\{f\}, \{d, r, v\})$. It is also surprising since SQL queries naturally support such separation by using the attributes of a meaningful key as join attributes (uniqueness), and specifying attributes as `IS NOT NULL` in the `WHERE` clause (completeness). We will demonstrate in Section 2 how the new concept of eUCs overcomes serious limitations for the use of SQL UNIQUE in the optimization of queries that are based on the retrieval of unique records, mainly because an index for UCs contains missing data. Such queries include the arguably most common type of join query whose join attributes form key/foreign key relationships. Furthermore, eUCs (E, U) subsume both UCs and so-called unique column combinations (UCCs) as special cases, where $E = U$ for both UCs and UCCs, and different null marker occur-

Proceedings of the VLDB Endowment, Vol. 12, No. 13
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3358701.3358703

Table 1: Snippet of the *ncvoter8m* data set

id	*voter_id*	*first_name*	*middle_name*	*last_name*	*address*	*city*	*full_phone_num*	*register_date*	*download_month*
t_1	3885008	brandon	lee	jones	2120 ramsgate st	raleigh	$\perp$	2003-10-15	2011-12
t_2	3885008	brandon	lee	jones	1933 blackwolf run ln	raleigh	$\perp$	2003-10-15	2011-12
t_3	1014519	emilie	jackson	gaddy	55 summey rd	old fort	828 668 2036	1983-08-16	2011-10
t_4	1014519	emilie	jackson	gaddy	55 summey rd	old fort	828 668 2036	1983-08-16	2012-10
t_5	3492969	michael	martin	mullen	608 plum nearly ln #1	wilmington	910 425 4200	2003-12-15	2011-12
t_6	3492969	michael	martin	mullen	6837 wimbledon cir #201	fayetteville	910 425 4200	2001-03-21	2011-12
t_7	2047052	blanche	virginia	heath	141 happy ln	dobson	336 352 3682	1994-10-08	2011-10
t_8	2047054	carie	lynn	pressnell	141 happy ln	dobson	336 352 3682	1994-10-08	2011-10

rences are regarded as different values by UCs ($\perp \neq \perp$, while different null marker occurrences are regarded as the same value by UCCs ($\perp = \perp$). Hence, all optimization techniques applied to UCs and UCCs also apply to eUCs. Additional showcases for the use of eUCs in enforcing data integrity, referential integrity, optimizing the evaluation of other popular classes of queries, acquiring business rules and cleaning data have already been examined [38, 39]. In essence, the use of eUCs (E, U) results in a resource advantage whenever U is a proper subset of E, since the columns in $E - U$ are not required to separate E-complete records. In our example, the attributes v, r, and d suffice to identify uniquely every record that is complete on those attributes and f. Hence, both UCs and UCCs (E, E) utilize the columns in $E - U$ redundantly, in our example the column f. In summary, eUCs (E, U) work like primary keys on U for all records with no missing value on E.

The discovery of constraints such as UCs and UCCs helps with effective data access, cleaning, integration, linking, and processing [2]. In the same way, the discovery of eUCs facilitates these tasks more effectively when missing data is present. This applies particularly to modern applications. For example, missing values occur frequently in big data considering that veracity is one of its dimension. Similarly, the integration of data from heterogeneous sources in one schema typically requires the introduction of missing values on attributes that exist in some but not all sources.

Recent years have seen tremendous progress on the discovery of unique constraints [3, 14, 34] despite its computational difficulty. For example, on a data set with 50 columns, up to $126,410,606,437,752$ minimal UCs may exist. Figure 1 illustrates the search space for a schema with three columns, and the maximum solution space: the maximum number of minimal UCs is attained by either the set of all singleton subsets marked red, or by the set of all two attribute subsets marked purple. Deciding if some UC 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 UC [8]. Remarkably, algorithms exist that can find all minimal UCs for reasonably large numbers of rows or columns [32]. The discovery of eUCs is computationally even more challenging. Figure 1 illustrates the search space for a schema with three columns, and the unique set of minimal eUCs that attains maximum cardinality (marked red). We show for every schema with n attributes that the maximum number of minimal eUCs significantly exceeds that of minimal UCs. The following table shows the difference up to $n = 12$.

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

For data sets with 50 columns there can be over 390 million times more minimal eUCs than minimal UCs, namely

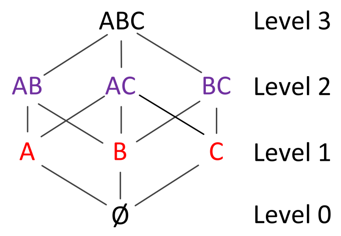


Figure 1: UC search/solution space over 3 columns

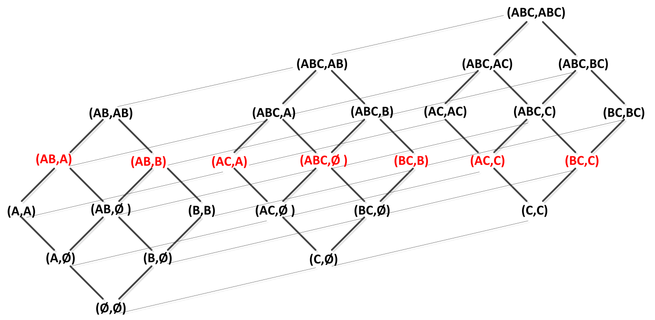


Figure 2: eUC search/solution space over 3 columns

up to $49,419,934,162,239,477,797,703$ minimal eUCs. Evidently, this growth requires new ideas and data structures for the discovery of eUCs.

This raises another challenge as it becomes even more important to provide computational support for identifying relevant eUCs from the large output of discovery algorithms. Relevant eUCs may include those that are meaningful for the underlying application domain, or those that result in performance increases for some application. Ultimately, a human is required to reasonably judge the relevance of eUCs, but automated rankings of eUCs can make this task easier.

Given the conceptual and practical advantages of eUCs, we will address their discovery from data. Our main contributions are: **(1)** We illustrate showcases for the use of eUCs in managing data integrity, improving schema quality, enabling schema design, and optimizing queries and updates. **(2)** We distinguish the discovery of eUCs from previous work. For example, eUCs subsume UCs and UCCs as idealized special cases. **(3)** We show that the decision variant of the discovery problem for eUCs is NP- and $W[2]$-complete in the input size. **(4)** We characterize the maximum number of minimal eUCs that an incomplete relation over n columns can have, and show which families of eUCs attain this number. **(5)** For handling the large search space, we introduce a new data structure that can store and look up eUCs efficiently, by effectively exploiting the separation between completeness and uniqueness requirements for pruning. **(6)** 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. Column(row)-efficient algorithms work efficiently with large numbers of columns (rows), while the hybrid algorithm performs best when the numbers of both columns and rows are larger. **(7)** As special cases of eUCs, we also discover UCs and UCCs. **(8)** We demonstrate which algorithms perform well on which benchmark data sets, and that eUCs are effective in uniquely identifying most entities. **(9)** We introduce rankings for eUCs, and illustrate how they help explore the output eUCs of discovery algorithms.

Organization. Section 2 presents showcases for the use of eUCs. We discuss related work in Section 3. Basic definitions are given in Section 4. In Section 5, we settle the computational complexity of discovering eUCs. Fundamental combinatorial results about eUCs are derived in Section 6. In Section 7 we introduce an important data structure for our discovery algorithms. In Sections 8, 9, and 10, we present column-efficient, row-efficient and hybrid algorithms, respectively. We report our experimental results in Section 11. We conclude in Section 12. More examples, proofs, data sets, a prototype, and user guide are available[1].

2. APPLICATION EXAMPLES

Previous work has already shown some of the benefits of eUCs [38, 39]. We will further show how they can be enforced in RDBMSs, improve schema quality, drive schema design, and advance query optimization.

2.1 Using trusted technology to enforce eUCs

EUCs can be implemented in trusted relational technology. They can be enforced directly on a database table by creating unique non-clustered indexes, or they can be used in views to reorganize the physical representation of data by creating unique clustered indexes. Such views are bound to the original table, so eUCs will be utilized by query optimizers. As a representative illustration, we create a schema R with columns f, d, r, and v in SQL Server, and define the following unique non-clustered index on R to enforce our example eUC $(\{f\},\{d,r,v\})$.

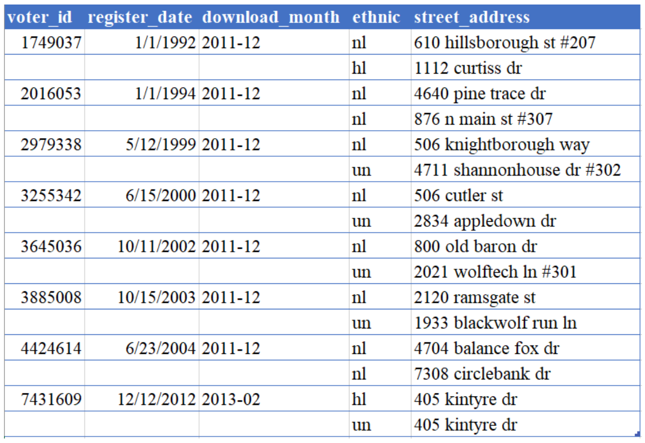
```
1:CREATE UNIQUE NONCLUSTERED INDEX EUC_fdrv_drv ON R (d, r, v)
2:WHERE f IS NOT NULL AND d IS NOT NULL AND r IS NOT NULL AND
        v is NOT NULL
```

The physical representation of data is important as non-unique columns can easily be retrieved after look-up or index search. The physical representation can be organized with eUCs by creating a unique clustered index on R as follows.

```
1:CREATE VIEW view_R WITH SCHEMABINDING AS
2:SELECT f, d, r, v
3:FROM R
4:WHERE f IS NOT NULL AND d IS NOT NULL AND r IS NOT NULL AND
        v is NOT NULL
5:GO
6:CREATE UNIQUE CLUSTERED INDEX EUC_fdrv_drv ON view_R (d, r, v)
```

Lines 1-5 create a view from table R which additionally stipulates the embedding of eUC $(\{f\},\{d,r,v\})$. Afterwards, a unique index is created on the view. With these implementations any updates that violate the eUC will raise

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



voter_id	register_date	download_month	ethnic	street_address
1749037	1/1/1992	2011-12	nl	610 hillsborough st #207
			hl	1112 curtiss dr
2016053	1/1/1994	2011-12	nl	4640 pine trace dr
			nl	876 n main st #307
2979338	5/12/1999	2011-12	nl	506 knightborough way
			un	4711 shannonhouse dr #302
3255342	6/15/2000	2011-12	nl	506 cutler st
			un	2834 appledown dr
3645036	10/11/2002	2011-12	nl	800 old baron dr
			un	2021 wolftech ln #301
3885008	10/15/2003	2011-12	nl	2120 ramsgate st
			un	1933 blackwolf run ln
4424614	6/23/2004	2011-12	nl	4704 balance fox dr
			nl	7308 circlebank dr
7431609	12/12/2012	2013-02	hl	405 kintyre dr
			un	405 kintyre dr

Figure 3: Records in *ncvoter8m* violating UC $\{d,r,v\}$

an error in SQL Server. The eUC will also be used by a query optimizer whenever the query only selects rows in the scope of the eUC, that is, whenever the query only selects $\{f,d,v,r\}$-complete rows in our example.

2.2 Schema quality and schema design

When analyzing the *ncvoter8m* data set we discovered the minimal eUC $(\{f\},\{d,r,v\})$. There are over 3 million records that are complete on $\{f,d,r,v\}$, and that subset of records satisfies the key $\{d,r,v\}$. Such a large data fragment suggests that $\{d,r,v\}$ should form a meaningful key. Figure 3 shows 8 pairs of records that have either non-matching values on attribute *ethnic* or *street address*. The changes in street address may result from updates within the same month the data is downloaded. Hence, the use of *download_month* as an attribute is too coarse. Instead, *download_date* should be used instead. Note that the eUC $(\{f,d,r,v\},\{d,r,v\})$ implies the SQL UNIQUE $\{f,d,r,v\}$, but not vice versa. In fact, the SQL UNIQUE $\{f,d,r,v\}$ cannot express the eUC $(\{f\},\{d,r,v\})$. As a consequence, $\{f,d,r,v\}$ cannot prevent violations of data integrity since it permits duplicate values on the meaningful key $\{d,r,v\}$.

Recently [41], eUCs and *embedded functional dependencies* (eFDs) were used to tailor relational schema design to data completeness requirements. While eFDs cause redundant data values on records that are E-complete, eUCs prohibit them. For illustration purposes we list the number *#complete* of data values that are complete, the number *#red* of those that are redundant, and the percentage *%red* of redundant data values in the data set.

data set	*#complete*	*#red*	*%red*
china	4,313,980	2,131,677	49.41
diabetic	1,017,738	543,935	53.45

The point is that eUCs remove all redundant data values during Boyce-Codd normalization or minimize them during Third normal form synthesis. For details we refer to [41]. Since eUCs cannot be expressed as eFDs [41] no eFD discovery algorithm can discover eUCs. Hence, our discovery algorithms for eUCs are fundamental for data-driven schema design approaches such as [31]. Schema design approaches customized to specific interpretations of null markers [21, 20] should be applied to records that do not meet the completeness requirements of the underlying application.

Table 2: Some performance measures in SQL Server

Read	Amount of page reads caused by query
Time	Query processing time in sec by CPU
Cost	Cost of query plan estimated by optimizer

2.3 Query optimization

We illustrate advantages of eUCs over SQL UNIQUE by an extension to our running example. For that purpose, we decompose *ncvoter8m* into tables over the following two schemata: `ncvoter8m_personal` contains attributes *voter_id*, *voter_reg_num*, *name_prefix*, *first_name*, *middle_name*, *last_name*, *name_suffix*, *age*, *gender*, *race*, *ethnic*, *full_phone_num*, *birth_place*, *register_date*, *download_month*, while `ncvoter8m_contact` has attributes *voter_id*, *street_address*, *city*, *zip_code*, *state* , *full_phone_num*, *register_date*, *download_month*. In the following join query we restore the original records in *ncvoter8m* that have no missing values on *voter_id*, *register_date*, *download_month*, and *full_phone_num*

```
SELECT * FROM ncvoter8_contact AS C
INNER JOIN ncvoter8_personal AS P
ON C.voter_id = P.voter_id AND
C.register_date = P.register_date AND
C.download_month = P.download_month
WHERE
C.full_phone_num IS NOT NULL AND
P.full_phone_num IS NOT NULL ;
```

To ease notation, we assume every attributes in the inner join is set to `IS NOT NULL` in the `WHERE` clause. Our SQL query makes natural use of the fact that $U = \{d, r, v\}$ forms a meaningful key on the records that are complete on $E = U \cup \{f\}$. In contrast, SQL UNIQUE offers neither effective nor efficient support for the evaluation of such common join queries that are based on key/foreign key relationships. In fact, in using SQL UNIQUE we are restricted to use either U or E. However, the SQL UNIQUE on U does not hold and the SQL UNIQUE on E would force users to formulate the above query as

```
SELECT * FROM ncvoter8_contact AS C
INNER JOIN ncvoter8_personal AS P
ON C.voter_id = P.voter_id AND
C.register_date = P.register_date AND
C.download_month = P.download_month AND
C.full_phone_num = P.full_phone_num ;
```

The latter query is unnatural as it uses *full_phone_num* as a join attribute. Hence, support for evaluating the query is ineffective. Even if users can deduce the SQL UNIQUE on E, its use in query optimization is inefficient. This is illustrated next by executing the two queries with and without index support. The data set *ncvoter8m* has $8,060,059$ rows and 19 columns. Measuring the pure runtime of a query is not enough to explain its performance, especially when the data is growing. Hence, our performance analysis will inspect both runtime cost and estimated cost of queries. We obtain these measures using *Microsoft SQL Sever Management Studio*, as in Table 2. The estimated costs may not be accurate to approximate the runtime of queries in general. However, our examples only aim at enhancing plain query plans, that will scan an entire table, with a more efficient search capability such as an index search. In this sense, the estimated costs actually provide a good measure on how well a query scales to larger tables.

Table 3: JOIN performance of example UC and eUC

Index	*Join by*	*Read*	*Time*	*Cost*
None	eUC	247227	120s	474
None	UC	247227	129s	493
UC	UC	16640034	161s	2251
eUC	eUC	111216	28s	103

Table 3 shows the performance of our four queries. The join query performs best when the eUC is enforced. Indeed, the inner join takes effective advantage of the eUC index (by applying a merge join, for example). The query performs worst if the UC is used. This is because the UC index is ineffective since significant processing time is spent on reading records with missing values. In general, the lack of separation between completeness and uniqueness concerns renders SQL UNIQUE useless for indexing purposes.

3. RELATED WORK

Surprisingly, no notion of uniqueness constraints has separated completeness from uniqueness requirements, until recently when eUCs were introduced [42]. In general, eUCs are more expressive than unique constraints from previous work. Their applications have been highlighted in recent work [38, 39]. The current article is the first to investigate the discovery problem for eUCs. Due to the separation principle, all our algorithms employ new techniques, but naturally take advantage of previous work on unique constraints and functional dependencies (FDs). We discuss those ideas now. For a recent survey on data profiling we refer to [1, 2].

We distinguish *unique column combinations* (UCCs) from SQL UNIQUE. The only but important difference in their semantics is that UCCs regard different null marker occurrences as matching values, while SQL UNIQUE regards them as non-matching values. SQL UNIQUE is the special case of eUCs (E, U) where $E = U$. We report results on this case because of its importance. By handling $\perp$ like any domain value, we also report on UCCs.

Core novelties. Column-based (row-efficient) 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. Stripped partitions and prefix blocks [17] improve the run-time efficiency of column-based algorithms. For the row-efficient discovery of eUCs (E, U) we employ similar ideas but tailor them to the discovery of uniques U and embeddings E. Another new idea is to prune the search space based on failure to meet the completeness requirements of E.

A row-based (column-efficient) algorithm refines FD sets by extracting counter-examples from data [13]. FD-trees manage FDs efficiently. This early algorithm only dealt with complete data. For the column-efficient discovery of eUCs (E, U) we generalize the notion of maximal non-uniques to maximal embedded non-uniques, and introduce rules for the induction of new candidates for eUC validation. Scalable UCC discovery [16] 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 [32, 30, 40] 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. A major shortcoming is the redundant validation of candidates based on the inability to compute stripped partitions dynamically. This is overcome by our hybrid eUC discovery algorithm where stripped partitions are generated dynamically during the traversal of candidate uniques, and used for efficient validation during the traversal of candidate embeddings. Specifically, the insight that completeness requirements are easy to validate is used for the design of eUC-trees. Similarly to previous hybrid algorithms our row-based part helps our column-based part reduce the search space, but does that for uniques and embeddings. Another new technique is that the discovered eUCs during our column-based part help our row-based part prune attributes of the search space. These strategies enable us to handle the larger search space that eUCs exhibit over UCs and UCCs.

Our combinatorial results are interesting from mathematical and practical points of view. They provide deep insight into the search and maximum solution space, showing that a potentially large number of eUCs must be discovered within a large product space, in contrast to an attribute lattice for UCs and UCCs. Our new data structure of an eUC tree was designed to meet these challenges: In contrast to the *antecedent trees* [13] that handle UC and UCC discovery, eUC trees accommodate further nestings at the leaf level to represent the product space. The growth in space is dealt with again by the separation of completeness and uniqueness requirements. Previous work on UC and UCC discovery simply regarded different occurrences of $\perp$ as matching (UCCs) or non-matching (UCs), and thus missed the opportunity to prune the search space when completeness requirements are not met. This, however, is done by eUC trees.

Other work. GORDIAN discovers minimal keys [34] using *prefix trees*, which efficiently extract *non-keys* known to violate a key. 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 [3] 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.

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 [5, 19, 22] rely on the no information interpretation of nulls, and are different from eUCs. Key sets can distinguish different tuple pairs by different sets of attributes [15, 24, 26, 35]. Probabilistic and possibilistic keys [6, 9, 18, 27, 33] are for data models that use possible worlds with no missing values.

Approximate keys [23] permit duplicate values up to a given threshold. They cannot express completeness requirements, and are therefore different from eUCs. For example, $\{f,d,r,v\}$ may be discovered as an approximate key on *ncvoter8m*, but does not give us any hint whether the eUC $(\{f\},\{d,r,v\})$ is satisfied. Vice versa, realizing that the eUC (E,U) holds on the relation r, tells us that the key U holds at least with the approximation ratio $|r^E|/|r|$. In this sense, eUCs tell us something about approximate keys, while approximate keys cannot tell us anything about eUCs. This provides more motivation for studying eUCs.

Overall, our article is the first to investigate the discovery problem for eUCs.

4. 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, $r^X = \{t \in r \mid t \textit{ is } X\text{-total}\}$, 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 [42] 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 $r \models (E,U)$, 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 $(\{f\},\{d,r,v\})$ instead of $(\{f,d,r,v\},\{d,r,v\})$.

A *unique column combination* (UCC) over relation schema R is a set $U \subseteq R$. The UCC U is *satisfied* by a relation r over R whenever for every pair of distinct tuples in r (not just the U-complete ones), there is some attribute in U on which the two tuples have different values. Recall that UCCs regard different occurrences of $\perp$ as matching values. In the special case of complete relations, 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. Some of the eUCs the snippet of Table 1 satisfies are $(\{f\}, \{d, r, v\})$, $(\emptyset, \{f, d, r, v\})$, and $(\{f, fi\}, \{d, r, v\})$, of which only $(\{f\}, \{d, r, v\})$ is minimal: neither $(\emptyset, \{d, r, v\})$, nor $(\{f\}, \{d, r\})$, nor $(\{f\}, \{d, v\})$, nor $(\{f\}, \{r, v\})$ are satisfied.

5. 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 $\lvert E\rvert \leq k$ and r satisfies (E, U)
	no, otherwise

As $U \subseteq E$ the cardinality $|E|$ of the extension E is an appropriate definition for the size of an eUC (E, U). We show that EUC is at least as hard as the decision variant KEY of the key discovery problem in complete relations, defined as follows.

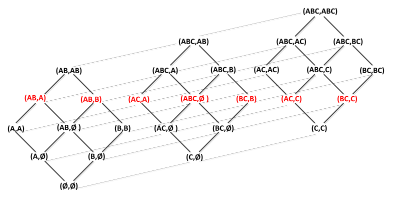
Problem:	KEY
Input:	complete relation r over schema R
	positive integer k
Output:	yes, if there is some $K \subseteq R$ where $\lvert K\rvert \leq k$ and r satisfies K
	no, otherwise

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 [7], and by reducing KEY to EUC we can establish NP-completeness for EUC, too. In recent research [8], 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, 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 `http://bit.ly/2gzDEYu`.

THEOREM 1. *Problem* EUC *is* NP- *and* W[2]*-complete.*

Table 4: Relation with max solution space over $A =$ *id*, $B =$ *name*, $C =$ *phone* and its embedded lattice

id	*name*	*phone*
0	Adam	6756
0	$\perp$	$\perp$
$\perp$	Eve	7654
1	$\perp$	0023
$\perp$	$\perp$	6756
2	Dave	$\perp$
$\perp$	Adam	$\perp$



Remarkably, recent algorithms can quickly solve large instances of KEY [30, 34]. The next section shows that the efficiency bar is raised even higher for eUCs.

6. 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 schema with n attributes [11, 36]. However, the maximum number of minimal keys over n attributes is $\binom{n}{\lfloor n/2 \rfloor}$ [10, 37]. 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.

A family $\mathcal{F}$ of eUCs is non-redundant if and only if there there are no two eUCs (E, U) and (E', U') in $\mathcal{F}$ such that $E \subseteq E'$ and $U \subseteq U'$ 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 4 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 2. *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 $\mathcal{F} = \{(E, U) \in 2^R \times 2^R : U \subseteq E$ and $|E| + |U| = |R|\}$.*

Despite the likely intractability, even with a fixed input size, and despite the large potential output, we will i) develop various efficient algorithms for the discovery of all minimal eUCs, and ii) define measures allowing us to effectively rank the many eUCs we discover.

7. EUC-TREES AS DATA STRUCTURES

Facing big search and solution spaces, it is necessary 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

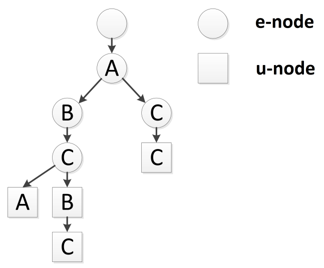


Figure 4: Example of an eUC-tree

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 E and UCs U. We therefore propose a new data structure, called eUC-*trees*. By separating the completeness requirements E from the uniqueness requirements U for eUCs (E, U) represented in eUC-trees, we can effectively prune the larger search space whenever one requirement is not met. 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* e(xtension)-nodes *or* u(nique)-nodes *and satisfy the following properties:*
1. Every node, except the root, is an attribute of R;
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 4 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.

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 $\neq$ $null$ then
7:         if uNode has no u-children then
8:             return true
9:         children $\leftarrow$ the set of all u-children of uNode
10:        for child $\in$ children $\cap U$ do
11:            if isRedundant(eNode, child) then
12:                return true
13:    else
14:        children $\leftarrow$ the set of all u-children of eNode
15:        for child $\in$ children $\cap U$ do
16:            if isRedundant(eNode, child) then
17:                return true
18:        children $\leftarrow$ the set of all e-children of eNode
19:        for child $\in$ children $\cap E$ do
20:            if isRedundant(child, uNode) then
21:                return true
22:    return false
23: return isRedundant(root, $null$)        ▷ Invoke recursion here
```

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.

8. 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 3. *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 3 forms the basis for the following iterative algorithm. Here, the minimal eUCs are represented by an eUC-tree from Section 7. 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 $\Sigma^{-1}$ of $r$
2: OUTPUT: The eUC-tree $T_\Sigma$ representing a minimal cover
   $\Sigma$ of those eUCs that hold on $r$
3: $T_\Sigma \leftarrow$ root                      ▷ Start with just a root node
4: for each $(M, N) \in \Sigma^{-1}$ do
5:     $\Omega \leftarrow \{(E, U) \sqsubseteq (M, N) \mid (E, U)$ is an eUC-path in $T_\Sigma\}$
6:     for $(E, U) \in \Omega$ do
7:         Remove eUC-path $(E, U)$ from $T_\Sigma$
8:         for $A \in R - M$ do
9:             if $(EA, U)$ non-redundant or $T_\Sigma = \emptyset$ then
10:                if $T_\Sigma = \emptyset$ then
11:                    $T_\Sigma \leftarrow$ root
12:                Insert $(EA, U)$ as a new eUC-path into $T_\Sigma$
13:        for $A \in M - N$ do
14:            if $(EA, UA)$ non-redundant or $T_\Sigma = \emptyset$ then
15:                if $T_\Sigma = \emptyset$ then
16:                    $T_\Sigma \leftarrow$ root
17:                Insert $(EA, UA)$ as a new eUC-path into $T_\Sigma$
18: Return $T_\Sigma$        ▷ If $\Sigma^{-1} = \emptyset$, then $T_\Sigma$ represents $\{(\emptyset, \emptyset)\}$
```

Algorithm 2 works correctly, please see[2].

Theorem 4. *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.*

9. 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 [17, 30]. As shown in Figure 1, *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 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 [17], 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.

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

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 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 $S \in \pi_K(r)$, $|r^R \cap S| \leq 1$.

Next, we describe the u-traversal (row-efficient algorithm) as Algorithm 3, and the e-traversal as Algorithm 4.

Algorithm 3 Unique-traversal (row-efficient algorithm)

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

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 [4] and have been widely used for the discovery of data dependencies [28, 34, 3]. 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.

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.

THEOREM 5. *Algorithm 3 computes a minimal cover of the set of eUCs that are satisfied by the given relation.*

10. 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.

Algorithm 4 Extension-traversal

```
 1: INPUT: Subset $R' \subseteq R$, stripped partition $\pi_U(r)$, UC $U$
 2: OUTPUT: The set $\mathcal{E}$ of all minimal extensions $E$ such that $(E, U)$ holds in $r$
 3: $\mathcal{E} \leftarrow \emptyset$
 4: currentLevel $\leftarrow R'$
 5: while |currentLevel| $> 0$ do
 6:     invalidExtns $\leftarrow \emptyset$
 7:     newValidExtns $\leftarrow \emptyset$
 8:     for $E \in$ currentLevel do
 9:         if $|r^E \cap S| \leq 1$ for all $S \in \pi_U(r)$ then
10:             $\mathcal{E} \leftarrow \mathcal{E} \cup \{EU\}$
11:             newValidExtns $\leftarrow$ newValidExtns $\cup \{E\}$
12:             continue                                  ▷ Goto line 8
13:         invalidExtns $\leftarrow$ invalidExtens $\cup \{E\}$
14:     nextLevel $\leftarrow \emptyset$
15:     for all $E, F \in$ invalidExtns where $|EF| = |E| + 1$ do
16:         if $\neg\exists E' \in$ newValidExtns where $E' \subseteq EF$ then
17:             nextLevel $\leftarrow$ nextLevel $\cup \{EF\}$
18:     currentLevel $\leftarrow$ nextLevel
19: return $\mathcal{E}$
```

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.

Algorithm 5

```
 1: INPUT: Stripped partition $\pi$ of $r$ over $U$, $A \in R - U$
 2: OUTPUT: The stripped partition $\pi'$ of $r$ over $UA$
 3: $\pi' \leftarrow \emptyset$
 4: for $S \in \pi$ do   ▷ Create map $M$ from $r[A]$ to tuple sets
 5:     for $t \in S$ do
 6:         if $t(A) \neq \perp$ then
 7:             $M[t(A)] \leftarrow M[t(A)] \cup \{t\}$
 8:     for each set $S$ in $M$ do
 9:         if $|S| > 1$ then
10:             $\pi' \leftarrow \pi' \cup \{S\}$
11: return $\pi'$
```

Reducing intermediate eUCs. The row-efficient algorithm can help the column-efficient algorithm reduce the number of eUCs generated at intermediate steps. By Theorem 3, 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, as levels of UCs and their extensions are 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.

Table 5: Run time (in seconds) of the three algorithms to discover eUCs from incomplete data

Data set	#R	#C	#⊥	#IR	#IC	#eUC	#UC	Alg. 2	Alg. 3	Hyb
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**
diabetic	101766	30	192849	100723	7	20130	3632	TL	TL	**1239.25**
uniprot512k	512000	30	3759296	512000	19	9480	100	TL	TL	**529.478**
pdbx_poly_seq_scheme	17305799	13	2035242	683410	5	15	9	TL	TL	**512.492**
ncvoter8m	8060059	19	22368378	8060056	10	343	96	TL	TL	**7966.79**

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 has the row-efficient algorithm as its core, but employs the column-efficient algorithm to update the search space when convenient. This results in *hybrid e-traversal* and *hybrid u-traversal* algorithms.

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) is 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.

Theorem 6. *Our hybrid discovery algorithm* Hyb *computes a minimal cover of the set of eUCs that are satisfied by the given relation.*

11. 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[29, 30]. We implemented the proposed algorithms in Visual C++, and carried out our experiments on an Intel Xeon W-2123, 3.6 GHz, 256 GB, Windows 10 PC. **Repeatability**: A prototype system and our data sets have been made available[3].

Table 6: Discovery time (s) on complete data

Data set	#R	#C	**#UC**	Alg. 2	Alg. 3	Hyb
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	625	5	1	0.06	0.005	**0.004**
fd-reduced	250000	30	3564	TL	**110**	313

Table 7: Discovery time (s) on incomplete data

Data set	**#UCC**	Alg. 2	Alg. 3	Hyb
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
diabetic	52	TL	TL	22.3
uniprot512k	10	TL	TL	25.719
pdbx_poly_seq_scheme	5	TL	TL	475.54
ncvoter8m	96	TL	TL	950.93

Next, we present our findings. For the experiments we set a time limit (TL) of 3 hours and a memory limit (ML) of 64 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.

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

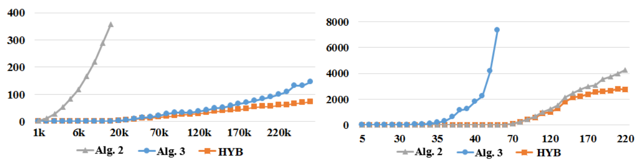


Figure 5: Row scalability on *uniprot* [left] and column scalability on *pm2.china* [right]

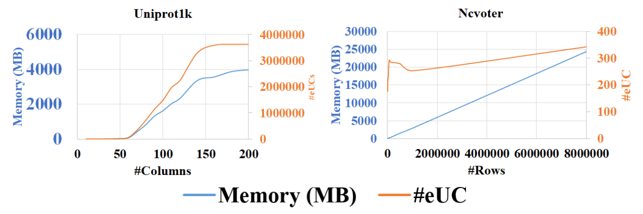


Figure 6: Row scalability on *uniprot* [left] and column scalability on *ncvoter* [right]

11.1 Benchmarks

Tables 5 and 6 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 (HYB). Note that neither Alg. 2 nor Alg. 3 can process the data set *pm2.5china* [25] 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 what is expected from the algorithmic design.

We can also discover UCCs by not choosing a symbol that is interpreted as $\perp$. Their total numbers (#UCC) and corresponding running times of our algorithms for their discovery from the incomplete data sets are shown in Table 7.

11.2 Scalability

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 5 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.

Additionally, Figure 6 demonstrates memory usages of the hybrid algorithm while discovering eUCs from data sets with different number of rows and columns. The *uniprot* data sets only contain 1000 rows. The memory consumption of the

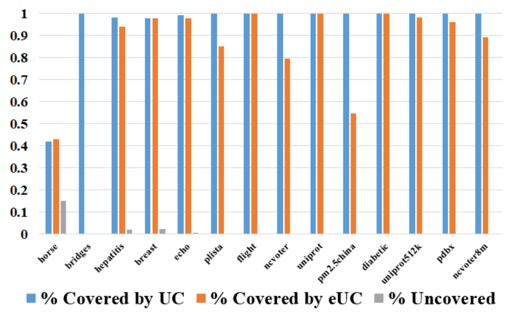


Figure 7: Coverage of incomplete data sets

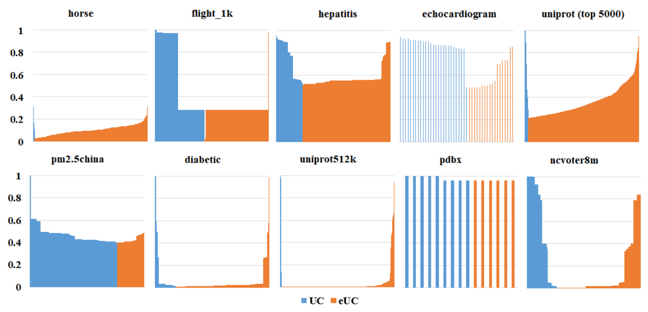


Figure 8: Relative scope of individual eUCs

hybrid algorithm slows down after most of the eUCs have been discovered. For the *ncvoter* data sets, the memory consumption of the hybrid algorithm increases faster than the number of eUCs. However, the memory consumption is almost linear to the number of rows in a data set and the increasing rate is around 0.003 MB/row.

11.3 Coverage

The main target of eUCs is the resource-conscious identification of entities in incomplete data. The more rows in the scope of an eUC, the more rows can be identified uniquely. Different eUCs can be used to distinguish different rows. Hence, we say a row in a data set r is *covered* if the row belongs to the scope of some eUC. The eUC *coverage* of a data set is the ratio of rows that are covered. Figure 7 shows the eUC coverage of all incomplete data sets. We distinguish between UCs and pure eUCs (where $E - U$ is non-empty). For a fixed U, UCs are of the form $(\emptyset, U)$ and have the minimum extension E among all eUCs (E, U). Hence, their scope r^U has maximum cardinality. However, the point is to discover for a given E which subsets U are sufficient for the identification of rows in r^E. Indeed, the high number of eUCs with a high coverage of pure eUCs is remarkable: There are many different ways by which many rows can already be distinguished by a proper subset of the extension in eUCs. This is precisely the reason for studying eUCs.

11.4 Ranking

In view of the large potential output of our discovery algorithms, we want to provide computational support for humans to judge the relevance of discovered constraints for a given application. Here, we propose the relative scope and the key length of eUCs as natural measures. The *relative*

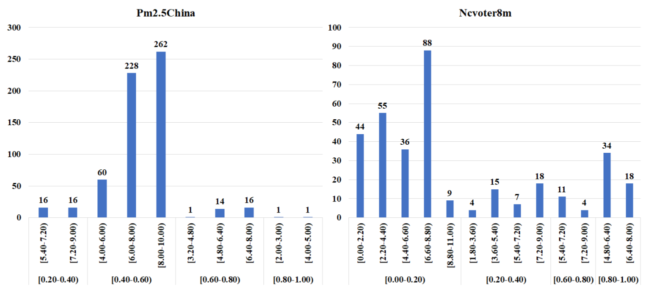


Figure 9: Rank eUCs by relative scope, then length

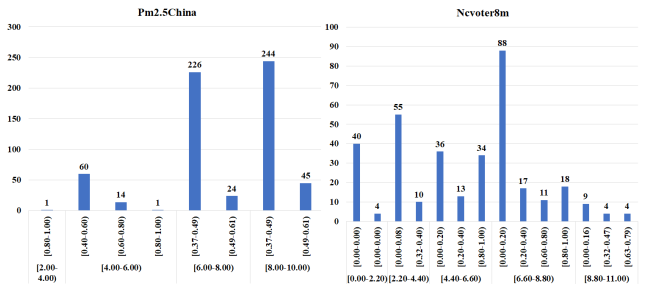


Figure 10: Rank eUCs by length, then relative scope

scope is defined as the cardinality of the eUC's scope relative to the number of rows in the given data set. The relative scopes are shown in Figure 8. Evidently, there are several eUCs in each data set which can uniquely identify most of the rows, except for *horse*. Indeed, the relative scope naturally ranks eUCs higher when their scope is larger, since more records can be identified uniquely. The *key length* of an eUC (E, U) is defined as the number $|U|$ of attributes in U. Different definitions may benefit different applications, but our definition is driven by the interest in fewer join attributes as they largely determine query performance. However, eUCs of larger key length may also be interesting, for example when the relative scope of eUCs with smaller key length is perceived as insufficient. The latter motivates the combination of our two measures: we can rank by relative scope first and then by length, or vice versa. The two combinations are illustrated in Figure 9 and Figure 10 for two of our larger benchmark data sets, respectively.

Note that the x-axis has been fixed here for our presentation, but in actual applications each of the measures is fully flexible to allow customization for the exploration needs of the humans who examine the eUCs. For instance, if too many eUCs fall within the same interval of some measure, then we can break up the interval into smaller fragments.

Table 8: eUCs selected by length and relative scope

E	*U*	*relative scope*
	v, ethnic, street_address, r, d	100.00%
	v, ethnic, zip_code, r, d	99.99%
	v, first_name, street_address, r, d	99.99%
	v, first_name, zip_code, r, d	99.98%
	v, middle_name, ethnic, r, d	93.23%
f	v, r, d	39.75%
f	v, ethnic, street_address, d	39.75%
f	v, last_name, street_address, d	39.75%

Table 9: JOIN improvement for eUC over UC

Read in %			*Time* in %			*Cost* in %		
Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
15.5	87.7	99.9	1.6	75.0	95.3	84.1	95.5	99.8

For example, Table 8 shows the top 8 eUCs of the *ncvoter8m* data set that contain the attribute *v(oter_id)*, ranked by their relative scopes and key lengths. The first eUCs show that there exist voters who change their ethnicity, address, first name, middle name, zip code or register date after their information have been downloaded again. Due to the sixth eUC, for voters who have supplied their phone numbers, the information about their ethnicity, address and last name only changes at the time they re-register.

11.5 Query and Update Performance

Table 9 shows the minimum, average, and maximum performance improvements when comparing join queries based on the eUCs (E, U) to join queries based on the corresponding UCs E. Here, performance refers to reads, times, and costs as per Table 2. For each of the 247 discovered pure eUC we discovered on *ncvoter8m*, we project *ncvoter8m* into two tables: both share all attributes in E but split attributes in $R - E$ evenly. The first query joins the two tables with inner join attributes from U, and attributes in E are declared as `IS NOT NULL` in the `WHERE` clause. The second query is the same except for using inner join attributes from E. The first query uses a clustered index for the eUC, while the second query applies a standard index for the UC. The improvements are significant.

For updates we compared transaction times for each discovered pure eUC (E, U) from *ncvoter8m* and its corresponding UC E. Both insert all records from *ncvoter8m* into an empty table. The first table enforces the eUC (E, U) with a non-clustered index, while the second table enforces the corresponding UC E with a standard index. The minimum performance improvement was -4.5%, the average 21.8%, and the maximum 41.4%. Again, this quantifies the resource advantage we gain by using eUCs.

12. CONCLUSION AND FUTURE WORK

Embedded uniqueness constraints identify E-complete records by a minimal subset $U \subseteq E$ of columns. They can be implemented by filtered indexes, and offer advantages over their special case SQL UNIQUE in integrity management and query optimization. Hence, their discovery in data sets is important, but the solution size is large and the problem likely to be intractable even when the input size is fixed. 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 is particularly suited for data sets with large numbers of columns and rows. For data sets with a large number of either columns or rows, the hybrid algorithm is outperformed by the column(row)-efficient algorithm. Our experiments confirm that discovered eUCs can be ranked effectively according to their length or relative scope. This provides guidance in determining which eUCs facilitate targeted and fast access to data for applications. In the future, we will investigate different hybrid and scalable approaches to the discovery of eUCs. Other classes of embedded data dependencies are appealing, foremost embedded FDs [41].

13. REFERENCES

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

[2] Z. Abedjan, L. Golab, F. Naumann, and T. Papenbrock. *Data Profiling*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[3] Z. Abedjan and F. Naumann. Advancing the discovery of unique column combinations. In *CIKM*, pages 1565–1570. ACM, 2011.

[4] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI/MIT Press, 1996.

[5] M. Alattar and A. Sali. Keys in relational databases with nulls and bounded domains. In *ADBIS*, pages 33–50, 2019.

[6] N. Balamuralikrishna, Y. Jiang, H. Köhler, U. Leck, S. Link, and H. Prade. Possibilistic keys. Fuzzy Sets and Systems, `https://doi.org/10.1016/j.fss.2019.01.008`, 2019.

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

[8] 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.

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

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

[11] 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.

[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] M. Hannula and S. Link. Automated reasoning about key sets. In *IJCAR*, pages 47–63, 2018.

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

[17] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: an efficient algorithm for discovering functional and approximate dependencies. *Comput. J*, 42(2):100–111, 1999.

[18] H. Köhler, U. Leck, S. Link, and H. Prade. Logical foundations of possibilistic keys. In *JELIA*, pages 181–195, 2014.

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

[20] H. Köhler and S. Link. SQL schema design: Foundations, normal forms, and normalization. In *SIGMOD*, pages 267–279, 2016.

[21] H. Köhler and S. Link. SQL schema design: Foundations, normal forms, and normalization. *Inf. Syst.*, 76:88–113, 2018.

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

[23] S. Kruse and F. Naumann. Efficient discovery of approximate dependencies. *PVLDB*, 11(7):759–772, 2018.

[24] M. Levene and G. Loizou. A generalisation of entity and referential integrity in relational databases. *ITA*, 35(2):113–127, 2001.

[25] 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.

[26] S. Link. Old keys that open new doors. In *FoIKS*, pages 3–13, 2018.

[27] S. Link and H. Prade. Relational database schema design for uncertain data. *Inf. Syst.*, 84:88–110, 2019.

[28] H. Mannila, H. Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, 1997.

[29] 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. *PVLDB*, 8(10):1082–1093, 2015.

[30] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *SIGMOD*, pages 821–833, 2016.

[31] T. Papenbrock and F. Naumann. Data-driven schema normalization. In *EDBT*, pages 342–353, 2017.

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

[33] T. Roblot, M. Hannula, and S. Link. Probabilistic cardinality constraints. *VLDB J.*, 27(6):771–795, 2018.

[34] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. GORDIAN: efficient and scalable discovery of composite keys. In *VLDB*, pages 691–702, 2006.

[35] B. Thalheim. On semantic issues connected with keys in relational databases permitting null values. *J. Inform. Proc. and Cyber.*, 25(1/2):11–20, 1989.

[36] B. Thalheim. *Dependencies in relational databases.* Teubner, 1991.

[37] B. Thalheim. The number of keys in relational and nested relational databases. *Discrete Applied Mathematics*, 40(2):265–282, 1992.

[38] Z. Wei, U. Leck, and S. Link. Entity integrity, referential integrity, and query optimization with embedded uniqueness constraints. In *ICDE*, pages 1694–1697, 2019.

[39] Z. Wei and S. Link. DataProf: Semantic profiling for iterative data cleansing and business rule acquisition. In *SIGMOD*, pages 1793–1796, 2018.

[40] Z. Wei and S. Link. Discovery and ranking of functional dependencies. In *ICDE*, pages 1526–1537, 2019.

[41] Z. Wei and S. Link. Embedded functional dependencies and data-completeness tailored database design. *PVLDB*, 12(11):1458–1470, 2019.

[42] Z. Wei, S. Link, and J. Liu. Contextual keys. In *ER*, pages 266–279, 2017.