Discovering Meaningful Certain Keys from Incomplete and Inconsistent Relations

Henning Köhler
Massey University, Palmerston North, New Zealand
h.koehler@massey.ac.nz

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

Xiaofang Zhou
The University of Queensland, Brisbane, Australia
Soochow University, Suzhou, China
zfx@itee.uq.edu.au

Abstract

Completeness and consistency are two important dimensions for the quality of data, in particular relational data. This is true because most data sets found in practice are both incomplete and inconsistent. The simplest yet arguably most important integrity constraint are keys. Recently, certain keys were introduced for incomplete relations. Certain keys can efficiently manage the integrity of entities while still permitting incompleteness in columns of the key. It is therefore an important task to discover the set of certain keys that hold in a given incomplete relation. However, if the given incomplete relation is also inconsistent with respect to some meaningful certain keys, algorithms that discover keys cannot succeed. As meaningful keys are likely to have a small number of violations, we propose an algorithm that discovers certain keys that do not exceed a given number of violations. We illustrate the effectiveness and efficiency of our algorithm in discovering meaningful certain keys from publicly available data sets.

1 Introduction

Data-driven decision making is a competitive imperative today, determining, for example, how businesses acquire, retain, and sell to their customers. Poor data quality is a main inhibitor to data-driven decision making [22, 23]. In fact, the better the quality of the data, the better the insight and value will be that we can derive from the data. This intuitive causal relationship has resulted in a lot of academic research and data quality tools [5]. In this article, we will address the integrity and completeness dimensions of data sets conforming to the relational model of data with missing information. More specifically, we are interested in the discovery of meaningful keys from inconsistent and incomplete data sets. Here, a key is considered to be meaningful when it models semantic properties of the underlying application domain. Furthermore, inconsistency refers to the violation of meaningful keys, and incompleteness refers to the presence of null marker occurrences in the given data set.

A very basic data quality principle is that of entity integrity: To represent each entity of an application domain uniquely in a data set. Entity integrity is one of Codd's three inherent integrity rules [3], and has been

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

enforced in database systems by primary keys over the last 40 years or so. Codd's rule of entity integrity states that every table must have a primary key and that the columns which form the primary key must be unique and not null [3]. Primary keys therefore address the consistency and completeness dimensions for the quality of data in relations. The benefits of enforcing entity integrity by keys are manifold: We can i) uniquely reference entities across data repositories, ii) minimize data redundancy at schema design time to process updates efficiently at run time, iii) provide better selectivity estimates in cost-based query optimization, iv) provide a query optimizer with new access paths that can lead to substantial speedups in query processing, v) allow the database administrator (DBA) to improve the efficiency of data access via physical design techniques such as data partitioning or the creation of indexes and materialized views, and vi) provide new insights into application data [24]. Modern applications raise the importance of keys even further. They can facilitate the data integration process and prune schema matches. Keys can further help with the detection of duplicates and anomalies [20], provide guidance in repairing and cleaning data [25], and provide consistent answers to queries over dirty data [13]. The discovery of keys, and other classes of dependencies, from data is one of the core activities in data profiling [1, 21, 19].

Consider the snippet of the PFAM (protein families) data set in Table 1. The whole data set is available at `http://pfam.sanger.ac.uk/`. The snippet fails to comply with the entity integrity rule as every potential primary key over the schema is violated. In particular, column *taxonomy* carries the null marker $\perp$. Nevertheless, every tuple in I can be uniquely identified by combining the two columns species and taxonomy. This is not possible with SQL `UNIQUE` constraints which cannot uniquely identify tuples in which null markers occur in the columns involved. The inability to insert important data into the database may force organizations to abandon key validation altogether, exposing their future database to less data quality, inefficiencies in data processing, waste of resources, and poor data-driven decision making.

species	*taxonomy*
cellular organisms	$\perp$
environmental samples	Bacteria;Proteobacteria;Gammaproteobacteria;Xanthomonadales;Xanthomonadaceae;Xylella;
environmental samples	Bacteria;Cyanobacteria;Prochlorales;Prochlorococcaceae;Prochlorococcus;

Table 1: Snippet of the ncbi_taxonomy data set

These observations had motivated us to investigate keys over relations from a well-founded semantic point of view. For this purpose, we interpret occurrences of $\perp$ as *no information* [15]. We use a possible world semantics, in which a possible world results from a table by replacing independently each occurrence of $\perp$, by a value from the corresponding domain, and where missing information is represented by the distinguished domain 'value' N/A for *not applicable*. In each possible world, occurrences of N/A are handled just as any domain value. In particular, the equality relation between two domain values extends to N/A. That is, N/A equals the domain value v if and only if $v = N/A$. In conclusion, each possible world can be handled as a relation without $\perp$ values in which duplicate tuples may occur. As usual, a possible world w satisfies a key X if and only if there are no two tuples in w that have distinct tuple identities and matching values on all the attributes in X. For example, the possible world W_1 in Table 2 satisfies the key $\{$*taxonomy*$\}$, and the possible world W_2 in Table 3 satisfies the key $\{$*species,taxonomy*$\}$ but violates the keys $\{$*taxonomy*$\}$ and $\{$*species*$\}$.

species	*taxonomy*
cellular organisms	N/A
environmental samples	Bacteria;Proteobacteria;Gammaproteobacteria;Xanthomonadales;Xanthomonadaceae;Xylella;
environmental samples	Bacteria;Cyanobacteria;Prochlorales;Prochlorococcaceae;Prochlorococcus;

Table 2: Possible World W_1 of snippet from Table 1

species	*taxonomy*
cellular organisms	Bacteria;Cyanobacteria;Prochlorales;Prochlorococcaceae;Prochlorococcus;
environmental samples	Bacteria;Proteobacteria;Gammaproteobacteria;Xanthomonadales;Xanthomonadaceae;Xylella;
environmental samples	Bacteria;Cyanobacteria;Prochlorales;Prochlorococcaceae;Prochlorococcus;

Table 3: Possible World W_2 of snippet from Table 1

In [12] we defined that a *certain key* $c\langle X\rangle$ is satisfied by a table I if and only if the key X is satisfied in every possible world of I. In particular, the semantics of certain keys does not prevent the entry of incomplete tuples that can still be identified uniquely, independently of which information any of the null marker occurrences represent. For example, the snippet in Table 1 satisfies the certain key $c\langle species, taxonomy\rangle$, but violates the certain key $c\langle taxonomy\rangle$ as witnessed by world W_2 in Table 3.

The concept of certain keys was not available yet to the designers of the PFAM database. In an effort to meet Codd's rule of entity integrity, they introduced the surrogate primary key *ncbi_taxid*. We will briefly discuss the consequences of this decision. For this purpose, consider the extension I of the snippet from Table 1 which is shown in Table 4.

ncbi_taxid	*species*	*taxonomy*
39378	Catenula sp.	Eukaryota;Metazoa;Platyhelminthes;Turbellaria;Catenulida;Catenulidae;Catenula;
66404	Catenula sp.	Eukaryota;Metazoa;Platyhelminthes;Turbellaria;Catenulida;Catenulidae;Catenula;
131567	cellular organisms	$\perp$
329529	environmental samples	Bacteria;Proteobacteria;Gammaproteobacteria;Xanthomonadales;Xanthomonadaceae;Xylella;
379362	environmental samples	Bacteria;Cyanobacteria;Prochlorales;Prochlorococcaceae;Prochlorococcus;

Table 4: Snippet I of the ncbi_taxonomy data set

The first two rows of snippet I illustrate that some entities, a combination of a species and taxonomy, have been duplicated. The presence of the surrogate primary key does not help at all with preventing this. In particular, the relation violates the meaningful certain key $c\langle species, taxonomy\rangle$ and is therefore inconsistent. Unfortunately, such cases are not an exception in practice. In New Zealand, hospital patients are identified by their National Health Index (NHI) number. In a 12-month intensive duplicate-resolution programme in 2003, over 125,000 duplicate NHI numbers were identified and resolved[1]. This means there are some people who have more than one NHI number. The consequences can be severe, as the NHI numbers identify the health history of a patient. Being unaware of different NHI numbers for the same patient means that the health history is incomplete, which may lead to incorrect decisions.

In [12] we introduced a transversal-based method for the discovery of certain keys from incomplete relations. This method is agnostic to inconsistencies in the given incomplete relation, making it impossible to discover meaningful certain keys that are incorrectly violated. For example, an application of this method to the snippet I from Table 4 will only return the certain key $c\langle ncbi\_taxid\rangle$, giving the wrong impression that $c\langle species, taxonomy\rangle$ cannot uniquely identify entities.

Contribution and Organization. In this article we ask the challenging question how meaningful certain keys can be discovered from incomplete and inconsistent relations. After distinguishing our approach from previous work in Section 2, we briefly summarize our previous findings on certain keys and their discovery in Section 3. In Section 4, we will introduce certain near-keys of order n, stipulating that every possible world of a table must not have $n+1$ tuples with distinct tuple identities that all have matching values on all the attributes of the

[1] http://www.health.govt.nz/our-work/health-identity/national-health-index/nhi-information-health-consumers/national-health-index-questions-and-answers#duplicates

near-key. In particular, a certain near-key of order 1 is a certain key introduced in [12]. We will characterize this semantic definition syntactically, allowing us to decide satisfaction of a given near-key of order n just by looking at the given relation. The computational complexity of deciding satisfaction of certain near-keys is much more involved than deciding satisfaction of certain keys. While the latter can be done efficiently by checking weak agreement of tuple pairs, deciding satisfaction of near-keys is coNP-complete to decide, and even W[1]-hard in the order of near-keys. This can be interpreted as the overhead that inconsistency incurs on the problem of identifying meaningful certain key candidates. In Section 6, we will describe our algorithm for the discovery of certain near-keys of a given order, and illustrate the computation on our running example. As certain near-keys discovered can be accidental, we introduce an additional conflict-rate measure in Section 7 and discuss how it can be employed to quickly identify certain near-keys that are unlikely to constitute certain keys. In Section 8 we present some quantitative and qualitative results on applying our algorithm to a variety of publicly available data sets. The results suggest that our approach discovers good candidates for meaningful certain keys from inconsistent and incomplete tables. In practice, certain near-key mining does not seem to incur the large time penalties in comparison to certain key mining that may be expected due to the hardness of deciding certain near-key satisfaction. Finally, we conclude in Section 9.

2 Related Work

We briefly distinguish our approach from related work.

Certain keys were introduced recently in [12]. In this paper we introduce certain near-keys of order n, which subsume certain keys as the special case where $n = 1$. Apart from the discovery problem of certain keys, [12] addressed a variety of other problems which are beyond the scope of our contribution to certain near-keys. The discovery algorithm in [12] can only find certain keys that are satisfied by the given data set. This excludes any certain keys that are meaningful in the application domain but violated by the given data set.

The discovery of data dependencies from given data sets was introduced as the *dependency inference problem* in [17], and has received dedicated attention ever since. Recent surveys are given in [1, 21, 16]. In contrast to our work, most algorithms i) focus on complete data or treat null marker occurrences as any other domain value, ii) only return data dependencies that are satisfied by the given data set, and iii) do not attempt to distinguish between dependencies that are meaningful and those that are accidentally satisfied.

In the approximate dependency inference problem [8] not all the tuples of the given relation are considered. This is done in an effort to trade-in efficiency for the soundness of the results. In contrast, our algorithms report the exact degree of violation for the discovered certain keys, which is reported as the order of a near-key. Approximate functional dependencies [7] are those functional dependencies whose margin of violation in the given data set does not exceed a given threshold. The threshold is a global measure of approximation as it refers to a minimal number of tuples that need to be removed from the given data set to satisfy the functional dependency. In contrast, the order n of a certain near-key is a local measure of approximation as it refers to the maximum number of permissable duplicates (i.e. tuples that pairwise agree weakly with one another) for each combination of values on the attributes of the key.

The discovery of meaningful data dependencies from inconsistent tables has not received much attention. An exception is [27] in which good foreign key candidates are derived from inconsistent tables. In this work, null marker occurrences do not receive a special treatment. In contrast, the work in [18] studies the problem of foreign key discovery under different semantics of null marker occurrences recommended by SQL.

3 Certain Keys

We repeat the definition of certain keys from [12]. Beginning with basic terminology, let $\mathfrak{A} = \{A_1, A_2, \ldots\}$ be a (countably) infinite set of distinct symbols, called *attributes*. Attributes represent column names of tables. A

table schema is a finite non-empty subset T of $\mathfrak{A}$. Each attribute A of a table schema T is associated with an infinite domain *dom*(A) which represents the possible values that can occur in column A. In addition, we assume that each domain *dom*(A) contains the distinguished symbol 'N/A', which is short for *not applicable*. We include 'N/A' in each domain out of convenience, but stress that 'N/A' is not an actual domain value, but rather a marker that indicates that information does not exist, thereby representing Codd's null marker *inapplicable*. The marker 'N/A' will not feature in tables, but only in their possible worlds. This is in contrast to the *no information* null marker $\perp$ [15], representative of SQL's NULL value and also included in domains for syntactic convenience, which only features in tables but is replaced in possible worlds – either by 'N/A' or some other domain value.

Example 1: Consider a person's mobile phone number. This value may be missing, indicated by $\perp$, which can represent two different scenarios: (1) the person has a phone but its number in unknown, or (2) the person has no mobile phone, so no number exists. The latter case is represented by the value 'N/A'.

For attribute sets X and Y we may write XY for their set union $X \cup Y$. If $X = \{A_1, \ldots, A_m\}$, then we may write $A_1 \cdots A_m$ for X. A *tuple* over T is a function $t : T \to \bigcup_{A \in T} dom(A)$ with $t(A) \in dom(A)$ for all $A \in X$. For $X \subseteq T$ let $t[X]$ denote the restriction of the tuple t over T to X. Let t, t' be tuples over T. We define *weak similarity* of t, t' on $X \subseteq T$ as follows:

$$t[X] \sim_w t'[X] :\Leftrightarrow \forall A \in X.\, \big(t[A] = t'[A] \vee t[A] = \perp \vee t'[A] = \perp\big)$$

We will use the phrase t, t' *agree* interchangeably for t, t' *are similar*.

We say that $X \subseteq T$ is a *key* for the $\perp$-free table I over T, denoted by $I \vdash X$, if there are no two tuples $t, t' \in I$ that have distinct tuple identities and agree on X.

3.1 Definition

We will now define the notion of a certain key over general tables, using a possible world semantics. Given a table I on T, a *possible world* of I is obtained by independently replacing every occurrence of $\perp$ in I with a domain value different from $\perp$. We say that $X \subseteq T$ is a *certain key* for I, denoted by $c\langle X \rangle$, if the following holds:

$$I \vdash c\langle X \rangle :\Leftrightarrow X \text{ is a key for every possible world of } I\ .$$

We illustrate this semantics on our running example.

Example 2: The snippet I' in Table 1 violates the certain key $c\langle species \rangle$ because the value *environmental samples* is a duplicate in every possible world. The certain key $c\langle taxonomy \rangle$ does also not hold in I' as witnessed by the possible world W_2 in Table 3. However, the certain key $c\langle species, taxonomy \rangle$ does hold in I' because copies of the second and third tuples have different values on *taxonomy*, and the value *cellular organisms* is unique in *species*. The snippet I in Table 4 satisfies the certain key $c\langle ncbi\_taxid \rangle$, but violates the certain key $c\langle species, taxonomy \rangle$.

3.2 Syntactic Characterization

While the semantics of certain keys is well-founded in our possible worlds model, it is infeasible to explore infinitely many possible worlds in order to verify whether a given certain key holds on a given table. The following result from [12] characterizes the semantics of certain keys syntactically, using our notion of weak similarity. This means that we can directly use the given table to verify whether a given certain key holds, without having to look into any of its possible worlds. Theorem 1 provides a foundation for developing efficient algorithms that effectively exploit our keys in data processing.

Theorem 1: $X \subseteq T$ is a certain key for I iff no two tuples in I with distinct tuple identities are weakly similar on X.

Note that the validity of Theorem 1 does not rely on the availability of infinite domains, but also holds for finite domains of at least size two.

4 Certain Near-Keys and Their Satisfiability Problem

In incomplete tables the notion of *certain key* ensures entity integrity while permitting null values in key columns [12]. This helps us to deal with uncertainty with respect to unknown or missing values. We now introduce the notion of a certain near-key that handles inconsistent data in incomplete tables.

Definition 2 (certain near-key): Let T be a table schema and I an instance over T. A *certain near-key of order* n is an expression of the form $C_n\langle X\rangle$ where n is a positive integer and $X \subseteq T$ is a set of attributes. We say that $C_n\langle X\rangle$ holds on I iff for every possible world W of I there do not exist $n+1$ tuples $t_1, \ldots, t_{n+1} \in W$ with pairwise distinct tuple identities such that $t_1[X] = \ldots = t_{n+1}[X]$. We will also refer to X as a near-key of I, if $C_n\langle X\rangle$ holds on I, or as an *anti-key* otherwise.

According to this definition the semantics of a certain near-key forbids in all possible worlds the occurrence of too many tuples with distinct identities that have matching values on all of the key columns.

Example 3: We have seen before that the snippet I in Table 4 satisfies the certain key $c\langle ncbi\_taxid\rangle$, but violates the certain key $c\langle species, taxonomy\rangle$. In addition, I satisfies the certain near-key $C_1\langle ncbi\_taxid\rangle$ of order 1 and the certain near-key $C_2\langle species, taxonomy\rangle$.

As we experienced with certain keys already, the question arises how it can be decided whether a given certain near-key of order n is satisfied by a given table. We will see now that deciding satisfaction is a lot more complex for certain near-keys.

4.1 Satisfaction of Certain Near-keys

The first issue is how to deal with the "all possible worlds" requirement. Violation of a certain key can be tested by comparing tuples with respect to weak similiarity. This can be done efficiently (at least when the number of key-columns that permit null marker occurrences is small) by creating an index with respect to every null-columns combination [12].

For certain keys $c\langle X\rangle$ we could then iterate over each tuple t and check (using the index) whether t is weakly similar (on X) to any other tuple. For certain near-keys $C_n\langle X\rangle$ one may try to continue this check until n tuples weakly similar to t are found. If no such n tuples exist, the certain near-key $C_n\langle X\rangle$ is indeed satisfied. However, even if we find n tuples all weakly similar (on X) to our reference tuple t, there is no guarantee that a single possible world exists in which they are all identical (on X).

Example 4: Consider the following table:

$I =$

A	B	C
1	1	$\perp$
2	2	$\perp$
3	$\perp$	1
4	$\perp$	2

Here, each tuple is weakly similar on BC to two other tuples in I, but the certain near-key $C_2\langle BC\rangle$ still holds on I as no subset of 3 tuples displays pair-wise similarity.

Note that in the example the certain key holds on every possible world since the third and fourth tuple in I are *not* weakly similar (on BC), and thus the first tuple cannot be identical (on BC) to both of them in any possible world. Thus we require at least pairwise weak similarity (which is not transitive!) between tuples to ensure the existence of a possible world in which they are all identical. As it turns out, this necessary criterium is also sufficient.

Lemma 3: Given a set $\mathcal{S}$ of tuples, there exists a possible world in which all tuples in $\mathcal{S}$ have matching values on all the attributes in X iff the tuples in $\mathcal{S}$ are pairwise weakly similar on X.

Lemma 3 lets us decide whether a given certain near-key holds on a given instance.

4.2 Intractability of Deciding Satisfaction

Unfortunately, the problem of deciding satisfaction for certain near-keys is likely to be computationally hard.

Theorem 4: Given an instance I over table schema T and a certain near-key $C_n\langle X\rangle$ over T, deciding whether I satisfies $C_n\langle X\rangle$ is coNP-complete. □

The NP-hardness in Theorem 4 can be established by a reduction from the Maximum Clique problem, which is known to be NP-hard: Given an undirected graph G and integer n, does there exist a subset of n vertices which are pairwise adjacent (a clique) in G?

The close relationship to Maximum Clique becomes evident in Section 5.1. In fact, maximum clique is even W[1]-hard [4], and the reduction in our proof preserves the parameter n. Therefore, fixed parameter-tractability with respect to the order n also appears to be impossible for certain near-key satisfaction.

Theorem 5: Certain near-key satisfaction, parameterized by the order of the certain near-key, is W[1]-hard. □

However, it seems likely that for genuine (near-)key candidates weak similarity of tuples is relatively rare, and hence the corresponding graph for which we need to solve the maximum clique problem is sparse, thus allowing effective kernelization.

5 Identifying Certain Near-keys

In [12] a *row-based* approach for mining certain keys was presented. However, we find that it requires pairwise comparison of tuples to identify agree sets, which becomes expensive for tables with a large number of rows, but scales well for tables with a large number of columns.

An alternative approach is to consider subsets of attributes of a table, and to test for each of the corresponding certain keys whether it holds on the relation. This requires traversal of (part of) the lattice of attribute subsets in some fashion, and while it scales well in the number of rows, the number of attribute subsets grows exponentially with the number of table columns. We shall refer to such an approach as *column-based*. For a comparison of row-based and column-based approaches in the related context of functional dependency mining see e.g. [1, 19].

While both row-based and column-based approaches can work well for the discovery of certain keys, depending on the characteristics of the dataset, we find that discovery of certain *near*-keys strongly favours column-based approaches. Since a (certain) near-key of order n is only violated if $n + 1$ tuples (weakly) agree on it, a row-based approach requires examination of all subsets of rows of size $n + 1$, of which there are $O(|I|^{n+1})$ many. As this makes row-based approaches quickly infeasible even for fairly small data-sets, we will consider a column-based approach instead.

Here we are faced with the NP-hard problem of deciding whether a given certain near-key holds on I, but it turns out that this can be resolved fairly efficiently in practice. In the following, we will present an algorithm which solves the certain key satisfaction problem by finding cliques on the corresponding conflict graph, while exploiting the way it is constructed to achieve an effective decomposition into subgraphs. Afterwards we present a column-based approach which utilizes this near-key satisfaction test.

5.1 Conflict Graph

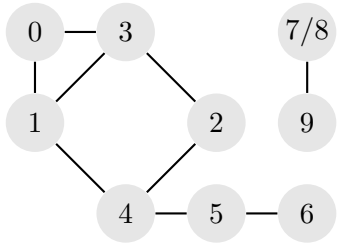
A *conflict graph* $\mathcal{G}$ corresponding to a certain near-key candidate $C_n\langle X\rangle$ consist of a vertex for every row in I and an edge between two vertices iff the corresponding rows are weakly similar on X. By Lemma 3 we have that $C_n\langle X\rangle$ is violated iff $\mathcal{G}$ contains a clique of size $n+1$.

When constructing the conflict graph, we only maintain tuples with conflicts, and combine tuples with identical key-values. Conflicts are identified by constructing indices similar to those used for certain keys [12]. We note that conflict graphs constructed in this fashion tend to be small when compared to the number of rows in I, and contain few edges. This holds true even for candidate sets X with a large number of conflicts, as clusters of weak-similarity tend to occur where multiple tuples are identical on X (rather than just weakly similar) and are thus merged into a single vertex.

Example 5: Consider the certain near-key $C_2\langle BCD\rangle$ on the table I below. Vertices in the corresponding conflict graph are labeled with the value of the matching tuple in column A.

A	B	C	D
0	1	1	1
1	1	1	$\perp$
2	1	2	$\perp$
3	1	$\perp$	1
4	$\perp$	$\perp$	2
5	2	1	$\perp$
6	2	1	1
7	3	1	1
8	3	1	1
9	3	$\perp$	1
10	4	1	1

$\Rightarrow$

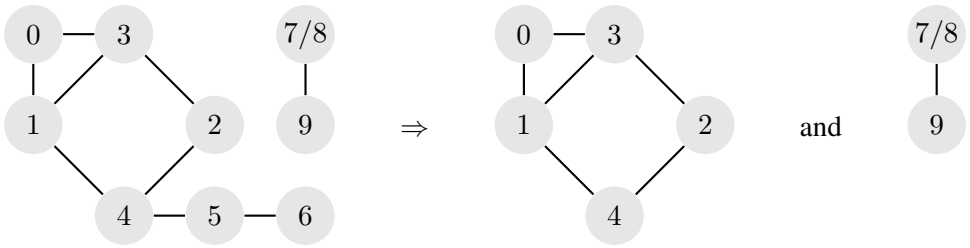


Note that tuple 10 is not weakly similar to any other tuple and thus omitted, while tuples 7 and 8 are merged.

5.2 Kernelization

Our kernelization phase is fairly standard for the maximum clique problem. We first keep removing vertices of degree $< n$, where n is the order of the near-key we want to check. Afterwards we partition the graph into connected components and process each component one-by-one. As conflicts between tuples are often isolated, this initial kernelization phase tends to produce many small components.

Example 6: Consider again the conflict-graph from Example 5. Initially the only vertex of degree < 2 is 6, as the vertices $7/8$ and 9 both have an "effective" degree of 2, as $7/8$ represents two tuples. After pruning 6 vertex 5 has degree 1, causing it to be pruned as well. Afterwards the remaining graph is partitioned into its connected components with vertices 0–4 and 7–9.



5.3 Decomposition

To check a connected component for cliques of size $n+1$, we pick one of the key columns (call it A). As edges are induced by weak similarity of tuples, no clique can contain tuples with distinct not-null values on A. This leads to two cases:

1. If the tuples associated with the vertices in the component are identical on A we simply drop the column.
2. Otherwise we partition the tuples with not-null values on A by their value on A, and add all remaining tuples (with $\perp$ on A) to each partition. We then drop columns A, and any vertices whose associated tuples have identical values on the remaining columns are merged (within each partition).

Note that in the second case, tuples within a component are weakly similar on the key iff they are weakly similar on the remaining columns, and any clique must fully lie within (at least) one component.

We continue this process until no columns remain. If at any point we encounter a vertex of multiplicity $> n$, we can abort[2] and report existence of a clique of size $n+1$.

Example 7: Consider the reduced components from Example 6. As the following steps are based on the values of associated tuples and edges are not considered directly, we shall revert to the underlying tuples, projected onto the near-key columns BCD. This gives us the following components to investigate:

B	C	D
1	1	1
1	1	$\perp$
1	2	$\perp$
1	$\perp$	1
$\perp$	$\perp$	2

and

B	C	D	
3	1	1	$\times 2$
3	$\perp$	1	

We will begin with the component on the left. Starting with column B we find the values are not identical, so we partition by not-null values of which there is only one, and end up with a single component and no merges:

B	C	D
1	1	1
1	1	$\perp$
1	2	$\perp$
1	$\perp$	1
$\perp$	$\perp$	2

$\Rightarrow$

C	D
1	1
1	$\perp$
2	$\perp$
$\perp$	1
$\perp$	2

Proceeding with column C we find distinct not-null values, and thus obtain two partitions, where we merge tuples $(1,1)$ and $(\perp,1)$ which become identical after projection onto D:

[2] In fact multiplicity n and degree ≥ 1 is sufficient, so we could have aborted after encountering vertex $7/8$.

C	D
1	1
1	$\perp$
2	$\perp$
$\perp$	1
$\perp$	2

$\Rightarrow$

D
1
$\perp$
2

$\times 2$ and

D
$\perp$
1
2

In a last step we partition each of the two components again, this time merging tuples $(1) \times 2$ and $(\perp)$ which gives us a 3-clique, so we can abort and report that the certain near-key $C_2\langle BCD\rangle$ does not hold.

While it is possible to apply a kernelization step after each column-projection, we found that this actually increased computation time.

6 Algorithm for Discovering Certain Near-keys Up to a Given Order

We shall now employ our certain near-key check in a column-based algorithm for finding all minimal near-keys up to a given order. In this context we say that a column set $X \subset T$ is a *certain anti-near-key of order* n iff $C_n\langle X\rangle$ does not hold on I. Note that anti-near-keys correspond to the agree sets in the row-based algorithm of [12], and that their complements are the transversals of all certain near-keys of the same order n.

6.1 The Algorithm

The principle idea is to compute all minimal near-keys of each order in sequence, starting with near-keys of order 1 (i.e. keys), and utilizing the near-keys of the previous order subsequently. Consider two sets:

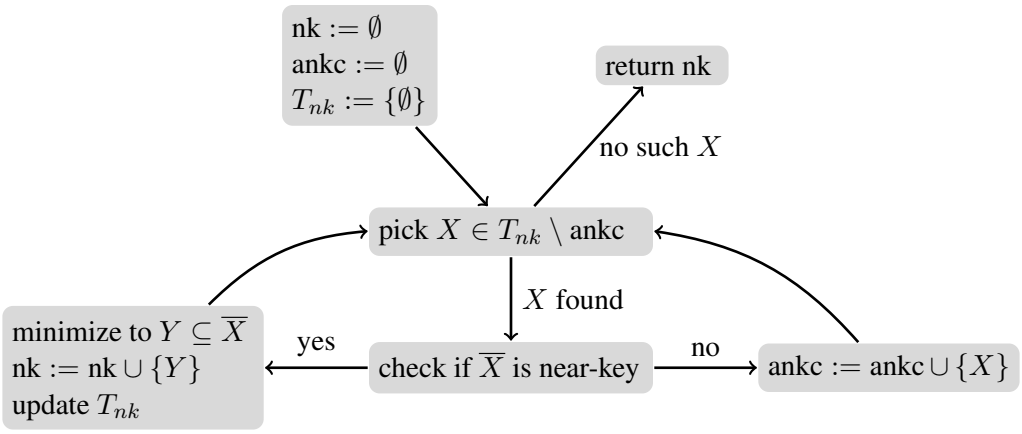
1. the set of all minimal near-keys (of a particular order), and
2. the set of all complements of maximal anti-near-keys (of the same order).

These two sets are transversal sets of each other, and we will compute them simultaneously. During computation, we maintain sets of all *currently known* minimal keys and complements of maximal anti-near-keys. Initially these sets are empty, and we extend at least one of them in each iteration until they become transversal sets of each other, at which point we must have found all of them.

To find a new minimal near-key or maximal anti-near-key, we track all minimal transversals T_{nk} of known near-keys, and pick one of these, say X, which is not the complement of a currently known anti-near-key. We then check whether its complement $\overline{X}$ is a near-key using the Algorithm sketched in Section 5, which can have two possible outcomes:

1. If $\overline{X}$ is not a near-key it must be a maximal anti-near-key which was not previously known.
2. If however $\overline{X}$ is a near-key then we minimize it to obtain a new minimal near-key. For this we attempt to remove each attribute from it in turn and check whether the resulting subset is still a near key.

The control flow of this algorithm is illustrated below.



Remarks. We can either search for near-keys and find anti-near-keys along the way, or vice versa. The final result is the same. The difference is that we minimize sets of the type we focus on, and track their transversals. Maximizing anti-near-keys may be slightly faster, as sometimes multiple attributes can be added with a single check using the counter-example $t_1, \ldots, t_{n+1}$ found, but the overall difference in speed turns out to be rather small. On the other hand, focusing on near-keys is not just more intuitive, but also allows us to 'carry over' results from the previous order, as discussed next. Hence we take the latter approach.

6.2 Utilizing near-keys of previous order

In the Algorithm described so far, near-keys of any particular order are computed independently of each other. The next step is to exploit the relationship between near-keys of different orders: A near-key of order n is also a near-key for any higher order.

This relationship can be utilized as follows: Instead of starting from scratch, we track all previous near-keys as known non-minimal near-keys. Note that using earlier orders other than the immediate predecessor would be redundant. In a first step we then minimize all known non-minimal near keys to convert them into minimal near-keys. Afterwards we compute the transversal set of known minimal near-keys and proceed as before.

6.3 An Example

We will now illustrate the computation for certain near-keys on the following real-world sample I from the `ncbi_taxonomy` data set of the pfam database:

ncbi_taxid	*species*	*taxonomy*
44482	Anopheles	Eukaryota;Metazoa;Arthropoda;Hexapoda;Insecta;Pterygota;Neoptera;Endopterygota;Diptera; Nematocera;Culicoidea;Culicidae;Anophelinae;Anopheles;
44484	Anopheles	Eukaryota;Metazoa;Arthropoda;Hexapoda;Insecta;Pterygota;Neoptera;Endopterygota;Diptera; Nematocera;Culicoidea;Culicidae;Anophelinae;Anopheles;
44534	Cellia	Eukaryota;Metazoa;Arthropoda;Hexapoda;Insecta;Pterygota;Neoptera;Endopterygota;Diptera; Nematocera;Culicoidea;Culicidae;Anophelinae;Anopheles;
59139	Cellia	Eukaryota;Metazoa;Arthropoda;Hexapoda;Insecta;Pterygota;Neoptera;Endopterygota;Diptera; Nematocera;Culicoidea;Culicidae;Anophelinae;Anopheles;
131567	cellular organisms	$\perp$
329529	environmental samples	Bacteria;Proteobacteria;Gammaproteobacteria;Xanthomonadales;Xanthomonadaceae;Xylella;
379362	environmental samples	Bacteria;Cyanobacteria;Prochlorales;Prochlorococcaceae;Prochlorococcus;
511133	environmental samples	Bacteria;Proteobacteria;Gammaproteobacteria;

We compute all certain near-keys that hold on I up to order 2. The three attribute names are abbreviated by the first letters, i.e., n, s, and t. At the start we do not know any certain near-keys or complements of anti-near-keys, i.e., $minKeys = \emptyset$ and $\overline{maxAnti} = \emptyset$. To find a minimal near-key or maximal anti-near-key, we

pick the maximum set $X = \{n, s, t\}$ and check, using the algorithm from the previous section, whether $C_2\langle X\rangle$ holds on I. For the minimization of this certain near-key of order 2 we determine that $C_2\langle s, t\rangle$ also holds on I, but further removal of attributes are not possible as neither $C_2\langle s\rangle$ nor $C_2\langle t\rangle$ hold on I. Consequently, we have $minKeys = \{\{s, t\}\}$, whose transversal set is $TR = \{\{s\}, \{t\}\}$. The complements of the elements in TR do not contain any known certain near-keys of order 2. A candidate for another certain near-key is therefore $X = \{n, t\}$, the complement of $\{s\} \in TR$. Indeed, our algorithm from the previous section confirms that $C_2\langle n, t\rangle$ holds on I. During minimization we can see that $C_2\langle t\rangle$ does not hold, but $C_2\langle n\rangle$ does hold on I. Consequently, we update $minKeys = \{\{s, t\}, \{n\}\}$, whose transversal set is $TR = \{\{n, s\}, \{n, t\}\}$. The complements of the elements in TR do not contain any known certain near-keys of order 2. A candidate for another certain near-key is therefore $X = \{t\}$, the complement of $\{n, s\} \in TR$. However, $C_2\langle t\rangle$ does not hold on I, which means that $\{t\}$ is a maximal anti-near-key and its complement $\{n, s\}$ in $\overline{maxAnti}$. The only remaining candidate for a certain near-key is $X = \{s\}$, the complement of $\{n, t\} \in TR$. However, $C_2\langle s\rangle$ does not hold on I, which means that $\{s\}$ is a maximal anti-near-key and its complement $\{n, t\}$ in $\overline{maxAnti}$. Hence, $\overline{maxAnti} = \{\{n, s\}, \{n, t\}\} = TR$, which means that $\overline{maxAnti}$ and $minKeys$ are transversal sets of each other. Consequently, the set of minimal certain near-keys of order 2 is

$$C_2\langle species, taxonomy\rangle, \text{ and } C_2\langle ncbi\_taxid\rangle \ .$$

Other near-keys that hold on this data set are $C_1\langle ncbi\_taxid\rangle$, $C_3\langle species\rangle$, and $C_4\langle taxonomy\rangle$.

7 Identifying Accidental Near-Keys

The notion of certain near-key was introduced to deal with dirty data issues which cause certain keys to be violated even though they *should* hold. However the opposite can happen as well, that is, certain keys should not hold in general but *happen* to hold for the instance examined. This occurs frequently for data sets with few rows, and likely more so for certain near-keys.

While the final decision on whether a mined certain near-key is sensible will typically rest with a domain expert, one may consider additional measures beyond the order of a certain near-key, to help decide whether it requires closer manual examination or can be discarded as accidental.

Two such measure quickly spring to mind: the number of conflicts (i.e., edges in the conflict graph) or the number of tuples participating in conflicts. While these tend to be closely related, the latter allows for a more sensible and intuitive normalization to the unit-interval $[0, 1]$. The point here is that the number of *potential* conflicts grows quadratically with the number of rows, while the number of actual conflicts hardly ever does, making a percentage-based measure behave very differently depending on the size of the table considered.

Example 8: Consider the following table *contacts*:

Name	Type	Phone
Ann Rosen	daytime	187629
Ann Rosen	mobile	762354
Bob Smith	daytime	655139
Bob Smith	mobile	334932

Here the certain near-key $C_2\langle$Name$\rangle$ holds, but the certain key $c\,\langle$Name$\rangle$ is not sensible. However, as the relation grows, the ratio of actual over potential conflicts converges towards 0. On the other hand, the percentage of tuples participating in conflicts remains at (or close to) 100% regardless of relation size.

This motivates the following.

Definition 6 (conflict rate): Let T be a table schema, I and instance over T and $C_n\langle X\rangle$ a certain near-key over T. The *conflict rate* of I w.r.t. $C_n\langle X\rangle$ is the percentage of rows in I that are weakly similar on X to at least one other row in I. We say that a *certain near key with conflict rate* r, written as $C_n^r\langle X\rangle$, holds on I if $C_n\langle X\rangle$ hold on I and the conflict rate of I w.r.t. $C_n\langle X\rangle$ is at most r.

Conflict rates can be used in at least two different ways:

1. After mining minimal certain near-keys, annotate them with their conflict rate. Optionally filter those that exceed a given threshold.

2. Reject attributes set $\overline{X}$ as near-key during the near-key test of the mining algorithm if the table's conflict rate w.r.t. $\overline{X}$ exceeds a given threshold.

Note that these options lead to different results, as in the latter case super-sets of $\overline{X}$ will be considered as minimal certain near-keys. This ensures that valid certain keys are less likely to be missed, but also increases the number of false positives and requires a threshold to be set a priori.

Finally we remark that searching for all certain near-keys that are minimal w.r.t. attribute set, order and conflict rate is not a promising approach. The problem here is that supersets of a certain near-key or order 2 or greater (but not including a certain key) are likely to meet this criterium of minimality, as any additional attribute is likely to reduce the conflict rate. The result is an explosion in the number of "minimal" near-keys.

8 Experiments with Real-World Data

We conducted experiments to get some insights about certain near-keys that occur in real-world data sets. For this purpose we mined a total of 145 tables from the following publicly available data sets:

- GO-termdb (Gene Ontology): `www.geneontology.org/`
- IPI (International Protein Index): `www.ebi.ac.uk/IPI`
- LMRP (Local Medical Review Policy): `www.cms.gov/medicare-coverage-database/`
- Naumann (benchmarks for FD mining): `https://hpi.de/naumann/projects/repeatability/data-profiling/fd.html`
- PFAM (protein families): `pfam.sanger.ac.uk/`
- RFAM (RNA families): `rfam.sanger.ac.uk/`
- UCI (Machine Learning Repository): `https://archive.ics.uci.edu/ml/datasets.html`

8.1 Quantitative Insights

As we are primarily interested in the discovery of meaningful certain keys, we restricted our attention to certain near-keys of order up to 5. In total, we found 642 minimal certain near-keys containing columns with null values, distributed across 22 tables. The following table shows how many certain near-keys of a particular order up to 5 we found and their distribution by table. In each column the number does not include certain near-keys that have already been found for a previous order. Numbers in brackets indicate how many of the near-keys found have a conflict rate of 5% or less. For order 1, conflict rates are always 0 by definition.

Data Source	Table	#rows	Order:	1	2	3	4	5
go_termdb	db	265		-	2(1)	-	2(0)	1(0)
ipi	protein_xref	156143		1(1)	-	-	-	-
lmrp	contractor	173		1(1)	-	-	-	-
lmrp	draft_contact_lookup	124		-	-	1(0)	-	7(0)
lmrp	lcd_related_documents	2311		-	1(1)	-	1(0)	1(0)
lmrp	lcd_x_advisory_committee	72		2(2)	1(0)	-	-	1(0)
lmrp	lcd_x_hcpc_code_group	2578		-	2(2)	-	-	-
lmrp	lcd_x_icd9_support_group	3429		1(1)	-	-	1(1)	1(1)
naumann	bridges	108		4(4)	30(0)	33(0)	16(0)	14(0)
naumann	echocardiogram	132		9(9)	36(1)	33(0)	32(0)	24(0)
pfam	clans	515		16(16)	8(5)	3(0)	1(0)	1(0)
pfam	dead_clans	38		-	1(1)	-	-	-
pfam	edits	8671		-	2(2)	-	-	-
pfam	literature_references	13784		-	-	1(0)	-	-
pfam	ncbi_taxonomy	822451		-	1(1)	-	-	-
pfam	pdb_pfamA_reg	246658		-	3(3)	2(2)	-	-
pfam	pdb_pfamB_reg	9624		-	6(6)	-	1(0)	-
rfam	features	21439852		2(2)	3(3)	-	-	1(1)
rfam	literature_references	1033		2(2)	1(0)	-	-	-
rfam	rfam	2208		20(20)	90(90)	50(37)	41(10)	45(8)
uci	adult	32561		-	-	37(37)	20(6)	13(1)
uci	auto_mpg	398		1(1)	4(3)	5(1)	3(0)	-
			Total:	59(59)	191(119)	165(77)	118(17)	109(11)

We find that a small number of tables tends to be responsible for the majority of near-keys. Such a skewed distribution suggests that many of the near-keys identified in tables with larger numbers of them are accidental. This is supported by the observation that, with the exception of table *adult*, such tables contain relatively few rows. However, the same skewed distribution occurs for both near-keys without columns containing nulls, and in particular ordinary keys, so a different behaviour for certain near-keys would have been surprising.

It is interesting to note though that there is no clear correlation between the number of minimal certain near-keys found in a table, and their conflict rates. We can however observer that (a) tables with few rows tend to have high conflict rates, and (b) conflict rates tend to increase with the order of certain near-keys. The distribution of conflict rates over all minimal certain keys with null columns, of orders 2-5, is shown on the left of Figure 1, where order 1 is omitted as conflict rates are necessarily 0. On the right, Figure 1 shows the relationship between conflict rates and "null-conflict rates", defined as the number of rows containing one or more null values in key columns over the total number of rows on the right (dot size indicates frequency).

We observe that for many (224 out of 583) of the minimal certain near-keys identified, the conflict rate lies at 5% or less, confirming their status as potential certain keys. However, it also highlights that many near-keys display a rather high conflict rate (218 have a conflict rate of 20% or higher), making it unlikely that they represent actual certain keys. A fair number of these (45) even have conflict rates of 100%, indicating cases similar to Example 8. For these cases, conflict rates can serve as an effective filter when searching for certain keys that may have been obscured by dirty data.

On the other hand, the relationship between conflict rates and null-conflict rates remains inconclusive. While null-conflict rates are by definition limited by conflict rates, they range anywhere from 0% to the conflict rate, with no discernable pattern. So far it appears that the role of null values in violating certain keys is highly specific to the data sets considered, and no general claims can be made.

In regards to the efficiency of our algorithms, the mining of all certain keys from all 145 tables took a total of 270 seconds, while the mining of all certain near-keys of up to order 5 from all 145 tables took a total of 1961 seconds. Hence, our algorithm required less than 7.3 times as much time to mine certain near-keys of up to order 5 in comparison to just certain keys. Note that these numbers report the averages over ten runs.

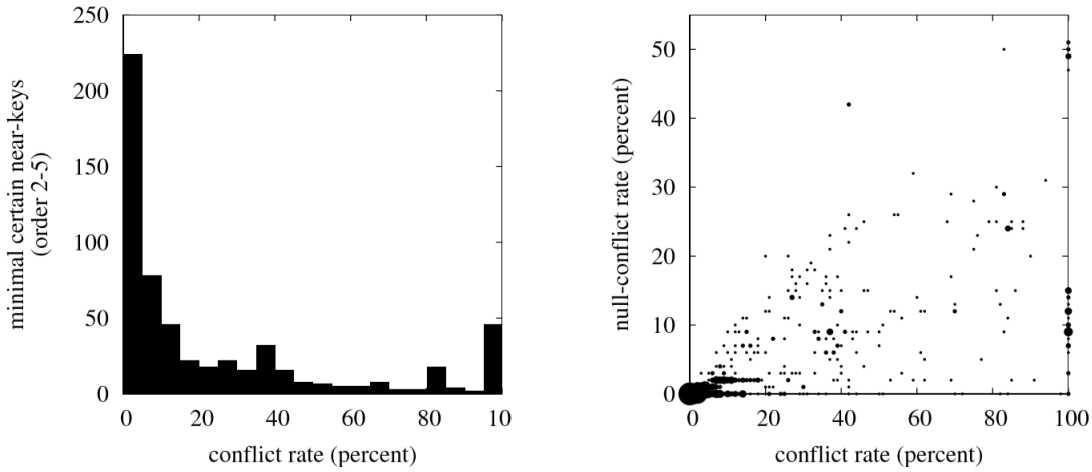


Figure 1: Distribution of conflict rates, and relationship between conflict and null-conflict rates

8.2 Qualitative Insights

Our running example for this paper was motivated by the `ncbi_taxonomy` table from the PFAM data set. The table contains three columns: *ncbi_taxid*, *species*, and *taxonomy*. Here, *ncbi_taxid* forms a surrogate primary key, and *taxonomy* features null marker occurrences. The table consists of 822,451 rows. Our algorithm found all certain keys (i.e. $c\langle ncbi\_taxid\rangle$) in an average 1.7 seconds over ten runs, and all certain near-keys up to order 5 (i.e. $C_1\langle ncbi\_taxid\rangle$ and $C_3\langle species, taxonomy\rangle$) in an average 13.9 seconds over ten runs.

As a second example we take a look at the `features` table from the RFAM (RNA families) data set. The table contains nine columns: *auto_features*, *auto_rfamseq*, *database_id*, *primary_id*, *secondary_id*, *feat_orient*, *feat_start*, *feat_end*, and *quaternary_id*. The columns *primary_id*, *secondary_id*, and *quaternary_id* each contain null marker occurrences. The table contains 21,439,852 rows, of which we mined the first million only. The following table shows the certain near-keys along with their order, number of conflicts and conflict rate.

certain near-key	*order*	*#conflicts*	*conflict rate*
auto_features	1	-	-
auto_rfamseq,database_id,feat_end,secondary_id	1	-	-
auto_rfamseq,database_id,feat_end,primary_id	1	-	-
auto_rfamseq,database_id,feat_end,feat_start	1	-	-
auto_rfamseq,feat_start,primary_id	2	10	0.002%
auto_rfamseq,feat_start,secondary_id	2	12	0.0024%
auto_rfamseq,feat_end	2	25	0.005%
auto_rfamseq,feat_start,quaternary_id	2	28	0.0056%
auto_rfamseq,feat_start	3	48	0.0093%
auto_rfamseq,database_id,primary_id	5	1473	0.2221%

Our algorithms found all certain keys in an average 31.6 seconds over ten runs, and all certain near-keys up to order 5 in an average 244 seconds over ten runs. Notably, the attribute *auto_features* serves as surrogate primary key, and all the certain near-keys of order 2 look very much like meaningful certain keys due to their small conflict rate. For example, the conflict rate for the certain key $c\langle auto\_rfamseq, feat\_start, primary\_id\rangle$ is only 0.002%, justifying the name *near*-key. Samples such as

auto_features	*auto_rfamseq*	*database_id*	*primary_id*	*secondary_id*	*feat_orient*	*feat_start*	*feat_end*	*quaternary_id*
102842	101318	EMBL-tRNA	tRNA-Ile	$\perp$	1	37601	37669	$\perp$
102841	101318	EMBL-tRNA	tRNA-Ile	$\perp$	1	37601	37678	$\perp$

indicate similar problems caused by the primary key as described for the `ncbi_taxonomy` table.

9 Conclusion and Future Work

We developed the first approach towards the discovery of meaningful keys from incomplete and inconsistent data sets. The new concept of a certain near-key of order n requires every possible world of a given data set to contain no more than n tuples that have matching values on all key attributes. Based on hypergraph transversals we developed an algorithm that discovers all certain near-keys up to a given order. The problem of deciding certain near-key satisfaction in a given data set is coNP-complete and W[1]-hard in the order of the certain near-key, i.e., it is unlikely that satisfaction can be decided in $O(P(|r|) \cdot f(n))$ time for some polynomial function P. Experiments with real-world data strongly indicate that our algorithm can discover meaningful certain keys which are violated in the given data set. Our conflict-rate measure can act as another filter to remove certain near-keys that are unlikely to constitute certain keys. Despite the theoretical hardness of certain near-key satisfaction, their discovery in practice appears to be not much more time-consuming than the discovery of certain keys.

There are many directions that warrant future research. It would be interesting to extend the work on certain keys and certain near-keys to conditional variants [5], which would also address the accuracy dimension of data quality. More generally, it is interesting to investigate the trade-offs between the expressivity of data dependencies and the efficiency of associated decision problems, such as deciding their satisfaction in given tables or discovering them from tables. One may think of suitable extensions to certain functional dependencies [11], differential dependencies [26], or inclusion dependencies [10], and conditional variants thereof, over relations or more sophisticated data formats [2, 6, 9]. A challenging and important problem is to determine to which degree affordable automation can help with establishing the meaningfulness of the discovered constraints [14]. The problems of discovering constraints from given data sets may become more effective or efficient when the application domain is more defined. In general we think that small numbers of violations are beneficial to the discovery of meaningful constraints. After all, exceptions prove the rule.

Acknowledgements

This research is supported by the Marsden Fund Council from New Zealand Government funding, by the Natural Science Foundation of China (Grant 61472263) and the Australian Research Council (Grants DP140103171).

References

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

[2] P. Brown and S. Link. Probabilistic keys for data quality management. In *CAiSE*, pages 118–132, 2015.

[3] E. F. Codd. *The Relational Model for Database Management, Version 2*. Addison-Wesley, 1990.

[4] R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013.

[5] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[6] S. Hartmann and S. Link. Efficient reasoning about a robust XML key fragment. *ACM Trans. Database Syst.*, 34(2), 2009.

[7] 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.

[8] J. Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. *Theor. Comput. Sci.*, 149(1):129–149, 1995.

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

[10] H. Köhler and S. Link. Inclusion dependencies reloaded. In *CIKM*, pages 1361–1370, 2015.

[11] H. Köhler and S. Link. SQL schema design: Foundations, normal forms, and normalization. In *SIGMOD*. http://dx.doi.org/10.1145/2882903.2915239, 2016.

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

[13] P. Koutris and J. Wijsen. The data complexity of consistent query answering for self-join-free conjunctive queries under primary key constraints. In *PODS*, pages 17–29, 2015.

[14] S. Kruse, T. Papenbrock, H. Harmouch, and F. Naumann. Data anamnesis: Admitting raw data into an organization. *Data Engineering Bulletin*, 39(2), 2016.

[15] Y. E. Lien. On the equivalence of database models. *J. ACM*, 29(2):333–362, 1982.

[16] J. Liu, J. Li, C. Liu, and Y. Chen. Discover dependencies from data - A review. *IEEE Trans. Knowl. Data Eng.*, 24(2):251–264, 2012.

[17] H. Mannila and K. Räihä. Dependency inference. In *VLDB*, pages 155–158, 1987.

[18] M. Memari, S. Link, and G. Dobbie. SQL data profiling of foreign keys. In *ER*, pages 229–243, 2015.

[19] F. Naumann. Data profiling revisited. *SIGMOD Record*, 42(4):40–49, 2013.

[20] F. Naumann and M. Herschel. *An Introduction to Duplicate Detection*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[21] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *SIGMOD*. http://dx.doi.org/10.1145/2882903.2915203, 2016.

[22] S. Sadiq. *Handbook of Data Quality*. Springer, 2013.

[23] B. Saha and D. Srivastava. Data quality: The other face of big data. In *ICDE*, pages 1294–1297, 2014.

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

[25] S. Song, L. Chen, and H. Cheng. On concise set of relative candidate keys. *PVLDB*, 7(12):1179–1190, 2014.

[26] M. W. Vincent, J. Liu, H. Liu, and S. Link. "Differential dependencies: Reasoning and discovery" revisited. *ACM Trans. Database Syst.*, 40(2):14, 2015.

[27] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. On multi-column foreign key discovery. *PVLDB*, 3(1):805–814, 2010.