Cleaning data with Swipe

TOON BOECKLING, Telecommunications and Information Processing, Ghent University, Gent, Belgium
ANTOON BRONSELAER, Telecommunications and Information Processing, Ghent University, Gent, Belgium

The repair problem for functional dependencies is the problem where an input database needs to be modified such that all functional dependencies are satisfied and the difference with the original database is minimal. The output database is then called a *minimal-cost repair*. If the allowed modifications are value updates, then finding a minimal-cost repair is NP-hard. A well-known approach to find *approximations* of minimal-cost repairs builds a Chase tree in which each internal node resolves violations of one functional dependency and leaf nodes represent repairs. A key property of this approach is that controlling the branching factor of the Chase tree allows to control the tradeoff between repair quality and computational efficiency. In this article, we explore an extreme variant of this idea in which the Chase tree has only one path. To construct this path, we first create an *ordered partition* of attributes (i.e., a partition of which the classes are totally ordered) such that classes can be repaired sequentially. We repair each class only once and do so by fixing the order in which dependencies are repaired. This principle is called *priority repairing*, and we provide a simple heuristic to determine priority. The techniques for attribute partitioning and priority repair are combined in an algorithm called Swipe. An empirical study on four real-life datasets shows that Swipe is one to three orders of magnitude faster than Llunatic and HoloClean, whereas the quality of repairs is comparable or better. A scalability analysis shows that Swipe scales linearly for an increasing number of tuples and quadratically for an increasing number of FDs.

CCS Concepts: • **Information systems** → **Data cleaning;**

Additional Key Words and Phrases: Data quality, data cleaning, functional dependencies, error detection and repair

ACM Reference Format:
Toon Boeckling and Antoon Bronselaer. 2025. Cleaning data with Swipe. *ACM J. Data Inform. Quality* 17, 1, Article 2 (March 2025), 29 pages. https://doi.org/10.1145/3712205

1 Introduction

We study the problem of repairing an inconsistent database through value modifications in the case where constraints are **functional dependencies (FDs)**. More precisely, starting from a database that violates some FDs (i.e., a dirty database), we seek to generate a database that satisfies all FDs (i.e., a repair) and that originates from the dirty database by sequentially changing the value of a cell in a table. Beyond the scope of toy problems, there usually exist many possible repairs for

Toon Boeckling and Antoon Bronselaer contributed equally to this research.
Authors' Contact Information: Toon Boeckling, Telecommunications and Information Processing, Ghent University, Gent, Belgium; e-mail: toon.boeckling@ugent.be; Antoon Bronselaer, Telecommunications and Information Processing, Ghent University, Gent, Belgium; e-mail: antoon.bronselaer@ugent.be.

ACM 1936-1955/2025/03-ART2
https://doi.org/10.1145/3712205

tid	#provider	hospital name	city	county	zip	state	measure code	condition
1	10006	eliza coffee memorial hospital	florence	laudxrdalx	35631	al	ami-4	heart attack
2	10006	eliza coffee memorial hospital	florence	laudexdale	35631	al	hf-2	heart failure
3	10006	eliza coffee memorial hospital	florence	lauderdale	3563x	al	ami-2	heart attack
4	1000x	eliza coffee memorial hospital	florence	lauderdale	35631	al	pn-3b	pneumonia
5	1003x	huntsville hospital	huntsville	madison	35801	al	ami-4	heart attaxk
6	10039	huntsville hospital	huntsville	madison	35801	al	ami-5	heart attack

#provider ≻ hospital name | city | county | zip | state | measure code | condition

hospital name → city
hospital name → zip
hospital name → state
hospital name → #provider
city → county
#provider → hospital name
measure code → condition

tid	#provider	hospital name	city	county	zip	state	measure code	condition
1	10006	eliza coffee memorial hospital	florence	**lauderdale**	35631	al	ami-4	***heart attaxk***
2	10006	eliza coffee memorial hospital	florence	**lauderdale**	35631	al	hf-2	heart failure
3	10006	eliza coffee memorial hospital	florence	lauderdale	**35631**	al	ami-2	heart attack
4	**10006**	eliza coffee memorial hospital	florence	lauderdale	35631	al	pn-3b	pneumonia
5	1003x	huntsville hospital	huntsville	madison	35801	al	ami-4	heart attaxk
6	***1003x***	huntsville hospital	huntsville	madison	35801	al	ami-5	heart attack

Fig. 1. An example cleaning scenario with hospital data [15] (top) and seven FDs (middle right). Actual errors in the data are marked in grey. An ordered partition of attributes is shown (middle left) over which FDs are forward repairable. The first partition class shows a priority model over its attributes. For this ordered partition, a repair obtained by using majority voting with random tie breaking as repair function is shown (bottom). Correct changes are shown in green bold font. Incorrect changes are shown in red bold slanted font.

a given dirty database, and most of them are perceived as "not good," because they simply make too much changes to the dirty database. To deal with this, one of the most prevalent approaches is to define a *cost model* that assigns a positive cost to each change. Introducing this cost model gives rise to the *repair problem for FDs*, which is the problem of finding a repair that minimizes the accumulative cost of all changes one must make to produce that repair. Such a repair is called a *minimal-cost repair*. The following example illustrates the repair problem for FDs and will be used as a running example throughout the article:

Example 1. Figure 1 (top) shows a snippet of the Hospital dataset [15] for which the seven FDs shown must be satisfied. If we assume that any change to an attribute value has cost 1, then a minimal-cost repair can be obtained by changing the values marked in grey (which happen to be here the actual errors in the dataset). This repair has cost 6, and it can be verified by the reader that no repair of cost 5 exists. It can be seen from this example that minimal-cost repairs are not unique, since several other repairs of cost 6 exist. For example, the violation of measure code → condition by tuples 1 and 5 can also be resolved by changing the value of measure code for tuple 5.

The Llunatic Chase algorithm. Although we can easily find a minimal-cost repair in Example 1, it is known that the repair problem for FDs, as defined here, cannot be solved efficiently. Deciding whether there exists a repair with a cost lower than a constant C is NP-complete, and finding constant-factor approximations of minimal-cost repairs is NP-hard [37]. To deal with this,

algorithms have been proposed to search for *approximate* solutions [5, 8, 37, 39]. In the current article, we focus on one particular approach that relies on a variant of the Chase algorithm and has been implemented in the open-source framework Llunatic [29, 30]. We note and emphasize here that Llunatic can in fact deal with constraints that are more expressive than FDs. Yet, we limit the explanation of Llunatic here to FDs only, as these constraints are the focus of the current article.

Originally, the Chase algorithm was designed to reason about queries in the presence of FDs in the specific setting where only a few tuples are present and tuples contain variables [1, 3]. Informally, the algorithm takes as input (i) a set of tuples and (ii) a set of FDs and then builds a sequence of *Chase steps*. In each such Chase step, one FD that is currently violated on the given set of tuples becomes satisfied by equating the right-hand side attributes of the FD. Equating two variables with different labels is done by a consistent choice (e.g., if labels are integer-valued indices, then we can always choose the variable with the lower index). Note that this means that, in the given set of tuples, *all* occurrences of one variable are replaced with the other variable and that the cost of such a global replacement can be significant. Equating a variable and a constant works in a similar way: The variable is replaced with the constant value. In case two constants need to be equated, the algorithm fails and no output is generated. If no failure occurs, then the output of a Chase step is used as input for the next Chase step, and this process continues until either all FDs are satisfied or a failure occurs. Under these conditions, it is well-known that the Chase algorithm terminates, and when it does so without failure, the output satisfies all FDs. Moreover, the Chase algorithm satisfies the *Church Rosser* property, which means that the output of the Chase algorithm does not depend on the order in which FDs are selected [1].

To use the Chase algorithm as a repair tool in the presence of constants, a modification of the Chase algorithm called the *Llunatic Chase* has been proposed [30]. One important difference is that equating two constants no longer leads to failure but is handled by a conflict resolution strategy. More specifically, resolution of conflicts is based on a *partial order* that models preferences of different constants and variables (e.g., null values). If a conflict cannot be resolved by means of this partial order, then special labeled variables (i.e., lluns) are introduced, which can later be used to query for human input. While the Llunatic Chase has been shown to terminate, it does not satisfy the Church Rosser property, and the outcome of a sequence of Chase steps is thus dependent on the order in which FDs are selected for repair. To deal with this, Llunatic considers a *Chase tree* in which each path is a sequence of Chase steps and leaf nodes are repairs. From all repairs in a Chase tree, a repair can then be selected. For example, one could consider a cost model and choose the repair with minimal cost.

Because it is infeasible to generate the complete Chase tree, Llunatic uses a *cost manager* that allows the definition of a wide range of pruning strategies. Examples of these strategies include (i) limiting the number of outgoing branches of a node by means of a branching threshold, (ii) limiting the amount of leaf nodes by the potential solutions threshold, and (iii) limiting modifications to forward repairs only [30]. The cost manager thus offers a tradeoff between repair quality (i.e., generating more repairs implies a higher chance of a finding a high-quality repair) and computational cost (i.e., generating more repairs implies larger Chase trees).

Single-path Chase trees with Swipe. The key contribution of the current article is to investigate an extreme case of cost management. More specifically, we investigate a degenerate variant where a Chase tree has only one path and thus generates only one repair. In addition, we restrict to forward repairs only, which means that violations of an FD are always restored by equating values of the right-hand side attribute in that FD. The main problem we face in this scenario is how to best select the next FD to repair. To solve this problem, we first partition the attributes involved in a given set of FDs and then order the partition classes such that the partition classes are *forward*

repairable when following their order. This means we can iterate over the ordered partition classes sequentially and, for each class C, apply forward repairing for those FDs of which the right-hand side attribute is in C. The next step is to visit each partition class and consider a *priority model* for the FDs we must repair for that class. The combination of (i) the ordered partition of attributes and (ii) the priority models for each class gives us the ability to produce a (degenerate) Chase tree with a single sequence of Chase steps, where each Chase step uses forward repairing. We can now summarize the key contributions of this article as follows:

— We present the *Swipe* algorithm to repair FD violations. This algorithm is grounded on two key ideas: (i) the notion of an ordered partition of attributes that is forward repairable and (ii) the notion of a priority model for FDs. We show that for any set of FDs there always exists an ordered partition that is forward repairable, and we present a simple algorithm to construct one that is maximally refined. We provide a simple yet effective heuristic to create a priority model for the FDs that must be repaired when visiting a class from a forward repairable ordered partition.
— Similar to other FD repair algorithms, we use equivalence relations on the set of tuples to keep track of repair steps previously done. To do this efficiently, Swipe uses *disjoint set forests*. These data structures model equivalence classes in a tree-based manner and have the property that the asymptotic complexity of merging two classes is constant.
— We study the theoretical properties of Swipe. We prove that it always terminates, and when it does, it produces a repair. For unary FDs (i.e., FDs with a singleton left-hand side), we show that each FD must be repaired at most once if resolution of conflicts is based on choice.
— We empirically study the tradeoff between repair quality and computational cost. On one hand, we demonstrate that the generation of repairs with Swipe is one to three orders of magnitude faster than with Llunatic or HoloClean. On the other hand, the repair quality in terms of F-score of correctly repaired attribute values is shown to be comparable or better. This provides first evidence for the fact that the construction of a single sequence of Chase steps can lead to good repairs.

The remainder of this article is organized as follows: In Section 2, we revise the vast body of literature on FD repairing and highlight the most important connections to existing methods. In Section 3, we introduce the basic concepts and notations related to the relational model and functional dependencies used throughout the article. In Section 4, we first formalize the notion of an ordered partition of attributes that is (forward) repairable (Section 4.1) and then develop an algorithm to produce such an ordered partition for a given set of FDs (Section 4.2). Next, we show how repair of a partition class is done (Section 4.3) and finally combine our results to formulate the Swipe algorithm (Section 4.4). We provide an experimental analysis in Section 5 and summarize the main contributions in Section 6.

2 Related Work

In the context of data cleaning, FDs and relatives can be used for a wide variety of tasks [33], such as missing value imputation [10, 19, 50], data fusion [11], and pattern violation detection [46], among others. In this section, we mainly focus on existing solutions for repairing data that are inconsistent against a set of FDs and contrast them to the proposed algorithm in this article. For a more general overview, the interested reader is referred to overview papers [16, 33, 43] and monographs [26, 34]. Here, we make the common distinction between (i) traditional approaches where searching repairs is cast into an *(approximate) optimization problem* and (ii) more recent approaches where *learning* techniques are used to generate repairs.

The traditional approach towards repairing violations of FDs is to consider a cost model that encodes a positive cost to changes made to the original data. The goal is then to find repairs that can be produced with a minimal sum of costs. In the setting of **Consistent Query Answering (CQA)** [2] one usually considers the deletion of tuples as an elementary change [39, 55]. In the current article, we focus on the problem where each elementary change is the replacement of the value of one attribute in one row with some other value. Finding a minimal-cost repair in this setting has been shown to be max-SNP hard (Reference [37], Theorem 5). Several strategies have therefore been proposed to approximate minimal-cost repairs efficiently, including greedy and heuristic search algorithms [7, 8, 17, 20, 36, 37, 49], sampling approaches [5], usage of MAX-SAT solvers [21], and alignment with knowledge graphs [18]. Many of these approaches make use of the notion of equivalence classes over the set of tuples to keep track of which rows must receive the same value for some attribute. This technique has been found necessary to ensure termination of repair algorithms [8]. The way in which equivalence classes are used can differ between approaches. Equivalence classes could, for example, be gradually built as a repair is constructed row-by-row [8]. Another approach that has been used initializes equivalence classes in a greedy way prior to repair, thereby determining equivalences that already exist in the dirty relation [5]. More recent approaches [17, 36] propose to encode violations within equivalence classes in a so-called conflict hypergraph, which can be used to effectively find minimal-cost repairs. However, it is shown that these approaches might face scalability and memory issues when executed on moderate- to large-sized datasets. To overcome those issues, a more scalable approach is proposed in Reference [49], but, unlike our approach, it does not solely target value updates and it selects, among the potential repair candidates, the candidate with highest support instead of the one with minimal cost. The algorithm we present also uses equivalence relations on rows, although without the notion of greedy initialization. Moreover, we explicitly use the notion of disjoint set forests to manage equivalence relations on rows efficiently.

When we evaluate search-based approaches, several experimental studies show that they either deal with scalability issues (even for moderate-sized datasets) [7, 8, 17, 21] or produce repairs of low quality [48]. This provides empirical evidence that it is challenging to find a good tradeoff between computational cost and repair quality. One particular method that facilitates this tradeof is the open-source framework Llunatic [29, 30], which uses a variant of the Chase algorithm. As mentioned in the introduction, this approach produces a Chase tree where each path is a sequence of repair steps and leaf nodes are actual repairs associated with a certain cost. To control the space and time complexity, the Chase tree can be pruned in several ways. The algorithm we present here can be thought of as a degenerate variant of such a Chase algorithm that considers a single sequence of Chase steps. By doing so, we minimize the computational cost and try to maximize quality of repairs as much as possible. Informally, the key idea is hereby to repair small groups of attributes of a dirty relation one after the other. This idea is similar to the Fellegi-Holt approach to repair tuple-level constraints [6, 9, 11, 27]. Interestingly, in the case of tuple-level constraints, it can be shown that if the set of constraints is satisfiable, then there always exists an ordered partition composed of *singleton* classes, such that we can repair sequentially. This result follows from Theorem 1 in Reference [27] for nominal edit rules and has been extended for other types of tuple-level constraints [6, 22]. As a consequence, it is possible for these constraints to repair one attribute at a time for each separate row. A similar property does *not* hold for FDs: There might not exist an ordered partition that is sequentially repairable as defined here and contains singleton classes only [31].

Over the past decade, there has been a growing interest in using learning approaches to deal with the repair problem [40, 47, 56]. The most well-known approach is HoloClean [47], which combines integrity constraints with statistical information about a given dataset to compose a probabilistic model that generates repairs. This learning-based technique has gained a lot of

attention [40] and inspired others to increase the role of learning. More specifically, learning approaches have also been used for error detection [32, 41, 45], where a sample of clean data is used to learn how errors should be characterized; or for error detection and correction, where non-expert users are involved to identify violations and corresponding repairs from which the model can learn [42]. Additionally, the increasing popularity of **Large Language Models (LLMs)** inspired several researchers to incorporate those LLMs into the repair process [14, 44]. In these approaches, there is no need anymore to use explicit integrity constraint formalisms. Rather, there is a model that is being trained to recognize errors. Although these approaches are promising for the future, their experimental evaluations have shown they face scalability issues as well, displaying run times of over 10 seconds for datasets with 1,000 tuples. We believe there is a requirement for tools that allow to produce repairs much faster than that. Evidence for that requirement can be found in a recent survey paper investigating the landscape of commercial tools for data quality monitoring [25]. Interestingly, none of those tools have adopted methods for automated error detection or repair, although we are convinced that these methods are key in the management of data quality.

3 Preliminaries

Let $\mathcal{A}$ be a countable set of attributes. For each $a \in \mathcal{A}$, let A denote the *domain* of a. We define a *(relational) schema* as a non-empty and finite set of attributes $\mathcal{R} = \{a_1, \ldots, a_k\}$ and a *relation* R with schema $\mathcal{R}$ as a finite set $R \subseteq A_1 \times \ldots \times A_k$. Elements of R are called *tuples*. We assume that $\mathcal{R}$ always has a special attribute tid that is unique for each $r \in R$. For a relation R with schema $\mathcal{R}$, a set of attributes $X \subseteq \mathcal{R}$, and a predicate P, we denote the *projection* of R over X by $R[X]$ and the *selection* over P by R_P. For convenience of notation, if X is a singleton set $\{a\}$, then we denote $R[\{a\}]$ by $R[a]$. In a similar way, if R contains only one tuple r, then we use the notation $r[X]$. Note that the projection $R[X]$ is a set and therefore contains no duplicate tuples. In some cases, it is useful to restrict the schema to attributes X but keep duplicates. We denote $R[\![X]\!]$ as the *multiset* of values obtained after restricting the schema to attributes X without removal of duplicates. An FD ϕ defined over a schema $\mathcal{R}$ is an expression of the form $X \rightarrow a$ such that $X \subseteq \mathcal{R}$ and $a \in \mathcal{R}$. In this expression, X is called the left-hand side and a is called the right-hand side of the FD.[1] For an FD ϕ, we will use the notation $\mathsf{LHS}(\phi)$ for the left-hand side of ϕ and $\mathsf{RHS}(\phi)$ for the right-hand side of ϕ. A relation R with schema $\mathcal{R}$ *satisfies* ϕ (denoted by $R \models \phi$) if:

$$\forall r_1 \in R : \forall r_2 \in R : r_1[X] = r_2[X] \Rightarrow r_1[a] = r_2[a].$$

Similarly, R *satisfies* a set of FDs Φ (denoted by $R \models \Phi$) if it satisfies all $\phi \in \Phi$. For a set of FDs Φ and a set $Z \subseteq \mathcal{R}$, the *projection* of Φ over Z is denoted by $\Phi[Z]$ and contains those FDs from Φ for which all attributes appear in Z. More formally:

$$\Phi[Z] = \{\phi \mid \phi \in \Phi \wedge \mathsf{LHS}(\phi) \subseteq Z \wedge \mathsf{RHS}(\phi) \in Z\}.$$

An FD ϕ defined over $\mathcal{R}$ is said to be *implied* by a set of FDs Φ, if for any relation R with schema $\mathcal{R}$, we have $R \models \Phi \Rightarrow R \models \phi$ (i.e., satisfaction of Φ implies satisfaction of ϕ). We use the notation $\Phi \models \phi$ to say that ϕ is logically implied by the set Φ. Two sets of FDs Φ and Φ' are said to be *equivalent*, denoted by $\Phi \equiv \Phi'$ if:

$$(\forall \phi \in \Phi : \Phi' \models \phi) \wedge (\forall \phi' \in \Phi' : \Phi \models \phi').$$

Finally, for any Φ, a *minimal cover* [1] of Φ is any set $\underline{\Phi}$ such that (i) $\Phi \equiv \underline{\Phi}$, (ii) no proper subset of $\underline{\Phi}$ is equivalent with Φ, and (iii) for each $X \rightarrow a$ in $\underline{\Phi}$, there exists no $X' \subset X$ such that $\Phi \models X' \rightarrow a$. An overview of all key concepts and the corresponding symbols is provided in Table 1.

[1] We can assume the right-hand side is a single attribute without loss of generality.

Table 1. Overview of Symbols and Key Concepts

Symbol	Meaning
$\mathcal{A}$	Countable set of attributes
$a \in \mathcal{A}$	An attribute with domain A
$\mathcal{R}$	Relational schema $\{a_1, \ldots, a_k\}$
tid	Tuple identifier attribute
R	Relation with schema $\mathcal{R}$
$r \in R$	A tuple in relation R
$R[X]$	Projection of R over $X \subseteq \mathcal{R}$
$R[\![X]\!]$	Projection of R over $X \subseteq \mathcal{R}$ with preservation of duplicates
ϕ	Functional dependency (FD) of the form $X \rightarrow a$
Φ	A set of FDs
$\Phi \models \phi$	FD ϕ is implied by Φ
$\underline{\Phi}$	A minimal cover of Φ
$\Phi[Z]$	Projection of Φ over Z
$\mathcal{P}$	An ordered partition of $\mathcal{R}$ of the form $[C_1 \ldots C_m]$
$\mathcal{R}_i$	For $\mathcal{P} = [C_1 \ldots C_m]$ and $1 \leq i \leq m$, $\mathcal{R}_i = \{C_1 \cup \ldots \cup C_i\}$
R_i^*	Partial repair: a relation with schema $\mathcal{R}$ that satisfies $\Phi[\mathcal{R}_i]$
$\mathcal{P} \rightsquigarrow \Phi$	Φ is sequentially repairable over $\mathcal{P}$
$\mathcal{P} \rightsquigarrow_F \Phi$	Φ is forward repairable over $\mathcal{P}$
P^+	The preorder obtained from a set of FDs Φ
DSF (a)	A disjoint set forest (DSF) for a
ρ_a	A repair function for a

4 Sequential Repairing

4.1 Basic Definitions

Consider a schema $\mathcal{R} = \{a_1, \ldots, a_k\}$ and a set of FDs Φ defined over $\mathcal{R}$. We define $\mathcal{P} = [C_1 \ldots C_m]$ with $m \leq k$ to be an *ordered partition* of $\mathcal{R}$, which is a partition of which the classes are totally ordered. In this article, we assume the total order on the partition classes is encoded by the *index* of the classes. In other words, we have that $C_i < C_j \Leftrightarrow i < j$ for any i and j in $\{1, \ldots, m\}$. If $\mathcal{P}$ and $\mathcal{P}'$ are two ordered partitions of $\mathcal{R}$, then we say that $\mathcal{P}$ is a *refinement* of $\mathcal{P}'$ if we can transform $\mathcal{P}$ into $\mathcal{P}'$ in a finite number of steps such that, in each step, we merge two *adjacent* classes from the previous step. For a given $\mathcal{P}$ and for any $i \in \{1, \ldots, m\}$, we denote by R_i^* a relation with schema $\mathcal{R}$ such that $R_i^* \models \Phi[\mathcal{R}_i]$ where $\mathcal{R}_i = \{C_1 \cup \ldots \cup C_i\}$. In the context of our repair approach, we will call such a relation a *partial repair*. We now say that Φ is *sequentially repairable* over $\mathcal{P}$, denoted by $\mathcal{P} \rightsquigarrow \Phi$, if:

$$\forall i \in \{1, \ldots, m\} : \forall R_{i-1}^* : \exists R_i^* : R_i^*[\mathcal{R}_{i-1}] = R_{i-1}^*[\mathcal{R}_{i-1}], \tag{1}$$

where we adopt the convention that $R_0^* = R$ and $\mathcal{R}_0 = \{\text{tid}\}$, which reflects the assumption that tid is unique for each $r \in R$. In words, if Φ is sequentially repairable over $\mathcal{P}$, then any partial repair R_{i-1}^* can be transformed into a partial repair R_i^* by modifying only the values for attributes in C_i. By repetition, we eventually obtain a relation R_m^* that satisfies Φ. The relevance of sequential repairability lies in the fact that if $\mathcal{P} \rightsquigarrow \Phi$, then classes from $\mathcal{P}$ can be repaired *one at a time* in order of increasing i. We now make two refinements to the notion of sequential repairability to make it a useful repair instrument.

First, it easy to see that, for any schema $\mathcal{R}$ and any set Φ defined over $\mathcal{R}$, there always exists at least one (trivial) ordered partition for which Φ is sequentially repairable. Indeed, since FDs are always satisfiable, we have that $[\mathcal{R}] \rightsquigarrow \Phi$. This ordered partition is not very useful, because it simply tells us we can repair violations of Φ. However, if we are able to build a refinement of this ordered partition by repeated splitting of classes into sub-classes and still satisfy the condition of sequential repairability, then we can separate the treatment of different FDs and exploit this property in a repair algorithm. For that reason, we are interested here in those ordered partitions $\mathcal{P}$ that are *maximally refined* without losing the property of sequential repairability for Φ.

Second, if $\mathcal{P} \rightsquigarrow \Phi$, then it is possible that FDs must be satisfied by either equating the values for the right-hand side attribute or differentiating the values for the left-hand side attributes. For example, if $\mathcal{R} = \{a, b\}$ and $\Phi = \{a \rightarrow b\}$, then we have $[\{a\}, \{b\}] \rightsquigarrow \Phi$ as well as $[\{b\}, \{a\}] \rightsquigarrow \Phi$. That is, for given values of a, we can choose values for b and make sure equal values for a imply equal values for b. Similarly, for given values of b, we can choose values for a that do not cause a violation of $a \rightarrow b$. In this article, we restrict repairing to the case where violations of FDs are resolved by changing the values of the right-hand side attribute. This operation is also known as *forward* repairing [30]. We provide the following definition:

Definition 1. For a schema $\mathcal{R}$ and a set of FDs Φ defined over $\mathcal{R}$, let $\mathcal{P}$ be an ordered partition of $\mathcal{R}$. Then, Φ is *forward repairable* over $\mathcal{P}$, denoted by $\mathcal{P} \rightsquigarrow_F \Phi$, if and only if:

$$\forall C_i \in \mathcal{P} : \forall \phi \in \Phi[\mathcal{R}_i] \setminus \Phi[\mathcal{R}_{i-1}] : \mathsf{RHS}(\phi) \in C_i. \tag{2}$$

In words, Φ is forward repairable if, for every class C_i, every FD not considered in previous classes has a right-hand side attribute that is an element of C_i. We now have the following result:

Proposition 1. *For a schema $\mathcal{R}$ and FDs Φ defined over $\mathcal{R}$, we have $(\mathcal{P} \rightsquigarrow_F \Phi) \Rightarrow (\mathcal{P} \rightsquigarrow \Phi)$.*

In this article, we seek to repair violations of FDs by first building an ordered partition of attributes $\mathcal{P}$ that allows forward repairing and then apply a repair algorithm to each class of $\mathcal{P}$, where we visit classes in the order. This repair algorithm uses a *priority model* that dictates the order in which FDs are repaired. We now demonstrate the main ideas in the context of the running example.

Example 2. In Figure 1, the ordered partition (middle left) is forward repairable for the FDs. It is also maximally refined w.r.t. forward repairability. That means we cannot split classes without breaking the condition of forward repairability. To construct a repair R^*, we visit the classes of $\mathcal{P}$ in order. The first class is $C_1 = \{\text{hospital name}, \#\text{provider}\}$ and we have:

$$\Phi[C_1] = \{\text{hospital name} \rightarrow \#\text{provider}, \#\text{provider} \rightarrow \text{hospital name}\}.$$

We then sort these FDs (this sorting is made more precise later in the article) and visit them in order. Suppose for now we first visit hospital name $\rightarrow$ #provider, then we find two violations: one for tuples $\{1, 2, 3, 4\}$ and one for tuples $\{5, 6\}$. We then do two things. First, we register that these two groups of tuples must have the same value for #provider in a special data structure. This data structure is used to keep track of decisions made during repair of visited FDs. Second, we resolve the violation by applying a function that maps the bag of values observed for #provider in the violating tuples to a single value. In this example, we do this by majority voting with random tie breaking. Concretely, this means we change #provider for tid $= 4$ into 10006. For the other violation, we have a random choice between the values so assume we choose 1003x. We then visit #provider $\rightarrow$ hospital name and find no violations. Since all FDs in $\Phi[C_1]$ are now satisfied, we have constructed the first partial repair R_1^*. We then proceed to $C_2 = \{\text{city}\}$. The set $\Phi[C_1 \cup C_2]$ now has one additional FD, which is hospital name $\rightarrow$ city and it has no violations. Moreover, the

FDs from the previous step are ensured to remain satisfied, so we now have a second partial repair R_2^* for which $R_2^* \models \Phi[C_1 \cup C_2]$. This procedure continues until we have visited $C_7 = \{\text{condition}\}$. After this last step, the original data has been iteratively transformed into a repair R^* that satisfies all FDs. An example repair under the assumption of majority voting with random tie breaking to fix violations is shown in Figure 1 (bottom).

From the example given above, two important questions arise. The first question deals with the generation of an ordered partition that is maximally refined and forward repairable for a given set of FDs. We will treat this question in Section 4.2. The second question deals with how violations of FDs are resolved for a single class. We will treat this question in Section 4.3. After that, we combine all these ideas in Section 4.4 to present the Swipe algorithm.

4.2 Ordered Partition Building

This section presents a method to construct, for given $\mathcal{R}$ and Φ over $\mathcal{R}$, an ordered partition $\mathcal{P}$ that (i) is maximally refined and (ii) satisfies $\mathcal{P} \rightsquigarrow_F \Phi$. To do so, note that Definition 1 (Equation (2)) implies that, for any FD $X \rightarrow a$ in Φ, attributes in X must be part of a class that does *not lie after* the class that contains a. We will encode this information in a preorder relation constructed from Φ. As a first step, we verify that Φ does not contain any trivial or implied FDs. This is to avoid that trivial FDs introduce unnecessary constraints on the order of attributes. In the scope of the running example in Figure 1, introducing the trivial FD $\{\text{county}, \text{zip}\} \rightarrow \text{county}$ would create a constraint on the order of the classes that contain county and zip. This constraint is unnecessary, since the FD is always satisfied. The same observation holds for FDs that are not minimal w.r.t. Φ (e.g., $\{\text{hospital name}, \text{state}\} \rightarrow \text{zip}$). To ensure that we encode only necessary requirements, we transform Φ into a *minimal cover* $\underline{\Phi}$. From such a minimal cover, we compose a binary relation $P \subseteq \mathcal{R} \times \mathcal{R}$ that satisfies the following criteria:

(1) $\forall a \in \mathcal{R} : (a, a) \in P$,
(2) $\forall (X \rightarrow a) \in \underline{\Phi} : \forall b \in X : (b, a) \in P$.

Here, $(b, a) \in P$ expresses the requirement that the class of b should not occur after the class of a in the ordered partition. Next, P is transformed into its *transitive closure* denoted by P^+ [54]. This is the smallest preorder (reflexive and transitive) that contains P. We say that P^+ is the preorder obtained from Φ. This preorder contains an equivalence relation $\equiv_{P^+}$ on $\mathcal{R}$. More specifically, we have $a \equiv_{P^+} a'$ whenever both $(a, a') \in P^+$ and $(a', a) \in P^+$. Consider now an ordered partition $\mathcal{P} = [C_1 \ldots C_m]$ such that:

$$\forall a \in \mathcal{R} : \forall a' \in \mathcal{R} : a \equiv_{P^+} a' \Leftrightarrow (\exists C_i \in \mathcal{P} : a \in C_i \wedge a' \in C_i) \tag{3}$$

and:

$$\forall C_i \in \mathcal{P} : \forall C_j \in \mathcal{P} : i < j \Rightarrow \left(\forall a \in C_i : \forall a' \in C_j : (a', a) \notin P^+\right). \tag{4}$$

An ordered partition $\mathcal{P}$ that satisfies these two constraints is said to be *induced* by P^+ and can be constructed as follows: First, we collect all equivalence classes from $\equiv_{P^+}$. Equation (3) implies that each such class must correspond to a class C_i from $\mathcal{P}$. Next, Equation (4) states that the total order we seek must be compatible with the *partial* order on the classes that is defined by: $[a] \leq [a'] \Leftrightarrow (a, a') \in P^+$, where $[x]$ is the equivalence class from $\equiv_{P^+}$ that contains x. The total order on classes that we seek is thus a linear extension of this partial order known as a *topological sort,* and several algorithms exist to construct it [35, 51]. We can now state the following result:

Theorem 1. *For a schema $\mathcal{R}$ and a set of FDs Φ defined over $\mathcal{R}$, let $\mathcal{P}$ be an ordered partition induced by P^+. Then, (i) $\mathcal{P} \rightsquigarrow_F \Phi$ and (ii) $\neg(\mathcal{P}' \rightsquigarrow_F \Phi)$ for any refinement $\mathcal{P}'$ of $\mathcal{P}$.*

P^+	hospital name	#provider	city	county	zip	state	measure code	condition
hospital name	x	x	x	*	x	x		
#provider	x	x	*	*	*	*		
city			x	x				
county				x				
zip					x			
state						x		
measure code							x	x
condition								x

Fig. 2. The construction of a preorder P^+ for the FDs from Figure 1 (middle right). Elements derived directly from the FDs are marked with "x," and elements added to compute the transitive closure are marked with "*." The preorder is the union of two preorders (marked by bold lines), each of them a preorder on a subset of $\mathcal{R}$.

Example 3. For the FDs shown in Figure 1 (middle right), Figure 2 shows the construction of the preorder P^+. First, we derive P from a minimal cover of the given FDs as explained. In our example, the available FDs are already a minimal cover and thus we proceed with the FDs as given. In Figure 2, the entries in P are marked by "x" symbols. We then compute the transitive closure P^+ and the additional entries to get P^+ are marked by "*" symbols. Because hospital name $\equiv_{P^+}$ #provider, both attributes must belong to the same partition class. Attributes determined by hospital name appear in singleton classes after the class that contains hospital name and #provider. Finally, condition should be after measure code. Note that the position of classes {condition} and {measure code} with respect to the other classes can be chosen freely and that this choice does not affect the further output of the repair process. More generally, when multiple topological sorts exist, any of them will lead to an ordered partition that behaves in an identical way.

4.3 Priority Repair

Assume an ordered partition $\mathcal{P} = [C_1 \dots C_m]$ for $\mathcal{R}$ such that $\mathcal{P} \rightsquigarrow_F \Phi$ for a given set of FDs Φ. Proposition 1 then ensures we can visit classes C_i in order and generate a partial repair R_i^* by changing only attributes from C_i. That is, we are given a partial repair R_{i-1}^*, and we must modify attributes C_i to fix violations of FDs $\Phi[\mathcal{R}_i]$ and obtain a new partial repair R_i^*. To do this, we propose a technique called *priority repairing*. The main principles of this technique are the following:

— FDs are inspected and repaired in a specific order dictated by a *priority* model that is customized for each class C_i. The key idea is to repair FDs that contain less reliable attributes in their right-hand side first and thereby try to maximize the accuracy of the repairs. We propose a simple heuristic to estimate the reliability of an attribute by means of the estimated number of changes that an attribute requires.

— To keep track of tuples that must receive the same value for $a \in C_i$, we use a ***disjoint set forest* (DSF)** [28] to represent equivalence classes on tuples. A DSF has the advantage that the necessary operations on equivalence classes have a near-constant time complexity [52]. This mitigates the complexity of FD *revision*, where one FD must be repaired again because of possible new violations caused by the repair of another FD.

— The value assigned to an equivalence class for $a \in C_i$ is determined by a *repair function* ρ_a. This is a function that maps the bag of values of a involved in a violation onto a new value. We have a particular interest in those repair functions that are *preservative*. These functions simply choose one of the values involved in a violation.

In the remainder of the section, we detail and formalize these main principles.

Priority model. To repair class C_i, we must ensure that the FDs over $\Phi[\mathcal{R}_i]$ are satisfied. Clearly, any FD from $\Phi[\mathcal{R}_{i-1}]$ contains no attributes from C_i and need not be considered. We can therefore restrict ourselves to the set $\Phi_i := \Phi[\mathcal{R}_i] \setminus \Phi[\mathcal{R}_{i-1}]$. In this set, there can be FDs ϕ for which:

$$\mathsf{LHS}(\phi) \cap C_i = \emptyset. \tag{5}$$

Those FDs have a left-hand side that contains only attributes that are clean at the time of repairing class C_i, because these attributes are all part of a class C_j with $j < i$. Such FDs are called *pilot* FDs for class C_i, and we repair them first, with two clear advantages. First, because we rely on *clean* data to make the first changes to attributes $a \in C_i$, we expect these changes to be accurate. Second, pilot FDs never require revision, as the values of their left-hand side attributes never change. Therefore, they need to be considered only once.

Example 4. In Example 2, there are no pilot FDs for class C_1. For all other classes, we have only pilot FDs, because the left-hand side of the FDs contains no attributes that are part of the class.

If an FD is not a pilot FD for class C_i, then it contains at least one attribute from C_i in the left-hand side. Those FDs are further sorted by means of a simple *priority model* that encodes the order in which attributes must be repaired. This priority model is a simple total order $\succ$ on attributes C_i, where $a \succ a'$ indicates that a must be repaired prior to a'. In other words, we require that $X \rightarrow a$ is repaired *before* $X' \rightarrow a'$ whenever $a \succ a'$. To construct a priority model, we propose to rank attributes within class C_i by increasing reliability. To make such a ranking, the most obvious method would be to ask for input of a human supervisor, which is not uncommon in repair approaches [29, 30]. However, such input might not be easy to collect, either because there is no supervisor or because a supervisor cannot easily rank attributes. In that case, we propose to use an *estimate* of the number of changes we will need to make to the values of a as follows.:

Suppose we have an FD $\phi := X \rightarrow a$ and a relation R, then for any $x \in R[X]$, the bag $R_{X=x}[\![a]\!]$ gives us the values for attribute a of those tuples for which $X = x$. If there are *different* elements in this bag, then we must change values for a to satisfy $X \rightarrow a$. If we denote the element with *highest* multiplicity[2] in this bag by $\mathsf{mv}(x, a)$ (i.e., the element obtained by majority voting), then changing the values of a for tuples $R_{X=x \wedge a \neq \mathsf{mv}(x,a)}$ allows to resolve the conflict with a *minimal* amount of changes. We can now collect these tuples for all values $x \in R[X]$ in a set that is denoted by ChangesFD and is defined by:

$$\mathsf{ChangesFD}(X \rightarrow a) := \bigcup_{x \in R[X]} R_{X=x \wedge a \neq \mathsf{mv}(x,a)}. \tag{6}$$

We can then aggregate over all FDs $\phi \in \underline{\Phi}$ where $\mathsf{RHS}(\phi) = a$ and collect the results in a set $\mathsf{Changes}(a)$ that is defined by:

$$\mathsf{Changes}(a) := \bigcup_{\phi \in \underline{\Phi} \wedge \mathsf{RHS}(\phi) = a} \mathsf{ChangesFD}(\phi). \tag{7}$$

The set $\mathsf{Changes}(a)$ basically gives us the tuples in R that require a change to attribute a if (i) we are interested in keeping the number of changes as small as possible and (ii) we would consider

[2]If there are multiple such values, then we choose one at random.

each FD independently. The more changes an attribute requires, the less reliable that attribute is. We therefore use $|\mathsf{Changes}(a)|$ to build our priority model such that:

$$a \succ a' \Leftrightarrow |\mathsf{Changes}(a)| \geq |\mathsf{Changes}(a')|. \tag{8}$$

The rationale of this heuristic is that, since we now must sort non-pilot FDs, there are attributes from C_i that appear in the left-hand side of these FDs. It is likely that some of these attributes will themselves contain errors (i.e., because they appear in the right-hand side of another FD). We thus seek to first repair an FD that has an attribute in the right-hand side that is *minimally* reliable. We now make two observations about this heuristic.

First, the assumption about FD independence is a naive one. When repairing two FDs with the same right-hand side attribute a, we must merge equivalence classes induced by both FDs (this is explained later) and our estimate neglects this. We accept this error in our estimate at the benefit of simple and efficient computation. Moreover, we will show empirically that classes C_i are in practical scenarios often singleton sets, in which case the estimate need not be computed, because all FDs are then by definition pilot FDs. If we do have multiple attributes in C_i, then empirical results show that the heuristic leads to sequences of FDs with repair quality superior to alternative sequences.

A second observation is that $|\mathsf{Changes}(a)|$ can be computed at different times with different results. One could, for example, compute $|\mathsf{Changes}(a)|$ on the original relation R before any modifications are done. The advantage of doing so would be that estimates are unbiased and not dependent on changes made by Swipe. An alternative strategy is to postpone the computation of $\mathsf{Changes}(a)$ up to the moment where the class that contains a is visited. In other words, we then compute $\mathsf{Changes}(a)$ from the partial repair R^*_{i-1}. Variations of this strategy are possible. We can, for example, estimate reliability before repair of C_i starts or postpone it to the point where pilot FDs have been repaired. Empirically, we have, however, not observed any significant differences between these variations on the datasets we tested. In the remainder of this article, we favor a strategy where estimation of reliability is done on *partial repairs*.

The two sorting principles that we explained (i.e., pilot FDs first and then least reliable right-hand side first) now allow us to rank the FDs that must be repaired in a class. Any possible ties in this ranking (e.g., two non-pilot FDs with the same right-hand side) are broken randomly *before* the repair of the class starts. The following example demonstrates the estimation of reliability of attributes and the implied order on FDs:

Example 5. In Example 2, we have $C_1 = \{\mathsf{hospital\ name}, \#\mathsf{provider}\}$. Both attributes appear once in the right-hand side of an FD. It can be verified that $\mathsf{Changes}(\mathsf{hospital\ name}) = \emptyset$ and $\mathsf{Changes}(\#\mathsf{provider})$ is either $\{4, 5\}$ or $\{4, 6\}$. It follows that $\#\mathsf{provider} \succ \mathsf{hospital\ name}$ and that we thus repair $\mathsf{hospital\ name} \rightarrow \#\mathsf{provider}$ first and then $\#\mathsf{provider} \rightarrow \mathsf{hospital\ name}$.

Tuple equivalence. Having determined the order in which FDs are treated, we turn our attention to the actual repair process. To repair attributes in C_i, we consider for each $a \in C_i$ an equivalence relation on the set of tuples from R^*_{i-1}. We use these equivalence classes to keep track of which tuples must receive the same value for a to satisfy FDs ϕ for which $\mathsf{RHS}(\phi) = a$. More specifically, whenever two tuples are in the same equivalence class for attribute a, we require they must receive the same value for a. Initially, each tuple is put in a separate singleton class. As we fix violations of FDs of the form $X \rightarrow a$, we require that tuples with the same value for X also have the same value for a. In other words, if we observe tuples in R^*_{i-1} with equal values for X, then we must *merge* the corresponding classes in which these tuples appear. To do this efficiently, we use a *disjoint set forest* [28] for each attribute a, denoted by $\mathsf{DSF}(a)$. With this data structure, each equivalence class is

ALGORITHM 1: Update DSF (a) for $X \rightarrow a$ and R^*_{i-1}

```
1:  procedure UPDATE(R*_{i-1}, X → a, DSF(a))
2:      H ← []                                          ▷ Initialize empty map
3:      for r ∈ R*_{i-1} do
4:          if H.notContains(r[X]) then
5:              H[r[X]] ← DSF(a).find(r[tid])           ▷ Register root
6:          else
7:              DSF(a).union(r[tid], H[r[X]])           ▷ Merge classes
8:              H[r[X]] ← DSF(a).find(r[tid])
9:          end if
10:     end for
11: end procedure
```

represented by a *tree* on tuple identifiers (i.e., values for tid). Each DSF (a) basically supports three operations:

- Operation makeset (id) adds a new singleton set (i.e., {id}) to the forest. Such a singleton set is represented by a one-node tree. This operation is used to initialize each DSF (a).
- Operation find (id) finds the root tid of the tree in which id resides and is used to determine if two values are in the same class or not.
- Operation union $(\mathsf{id}_1, \mathsf{id}_2)$ merges the classes in which id_1 and id_2 reside. This is done by finding the root nodes of the trees and in the case that the root nodes are distinct, attach one root as a child of the other. During the repair of an FD $X \rightarrow a$, this operation is used to merge equivalence classes for tuples that have equal values for X.

A key result is that the three elementary operations of a disjoint set forest can be implemented such that their *asymptotic* time complexity is very efficient. More specifically, one can provide an implementation such that the asymptotic time complexity of makeset (id) and union $(\mathsf{id}_1, \mathsf{id}_2)$ is $O(1)$, while that of find (id) is $O(\alpha(n))$, where $\alpha(.)$ is an inverse functional of the Ackermann function and n is the number of elements in the class where id is located [4, 52, 53]. The asymptotic efficiency of these operations is key, because it allows us to provide an efficient strategy for the update of DSF (a) during the repair of an FD $X \rightarrow a$. More specifically, if we want to repair $X \rightarrow a$, then we must update DSF (a) such that, after the update, we have:

$$\forall r \in R^*_{i-1} : \forall r' \in R^*_{i-1} : r[X] = r'[X] \Rightarrow \mathsf{DSF}(a).\mathsf{find}(r[\mathsf{tid}]) = \mathsf{DSF}(a).\mathsf{find}(r'[\mathsf{tid}]). \quad (9)$$

In words, after the update is done, we want that any two tuples from R^*_{i-1} with equal values for X are equivalent in DSF (a) and thus must receive the same value for a. The pseudo-code shown in Algorithm 1 provides a simple way to do so by using a map H with constant-time complexity for get and put operations. This map keeps, for each value in $R^*_{i-1}[X]$, the root node of the tree in DSF (a) in which tuples with this value reside. We iterate over tuples $r \in R^*_{i-1}$ and check if H currently contains $r[X]$. If this is not the case, then we register the root node of the tree where r[tid] is currently located (line 5). Else, we merge the current root node with the tree where r[tid] is currently located and update the map H (lines 7–8). The update procedure described here requires in each step one find operation and possibly one union operation on DSF (a), which implies an asymptotic complexity of $O\left(|R^*_{i-1}| \cdot \alpha(n)\right)$, with n the size of the largest equivalence class in DSF (a).

Example 6. In Example 5, it is shown that during the visit of C_1, we first repair hospital name → #provider. This is the first FD we visit, which means DSF (#provider) contains a singleton class for each tuple. Application of UPDATE $(R$, hospital name → #provider, DSF (#provider)) modifies

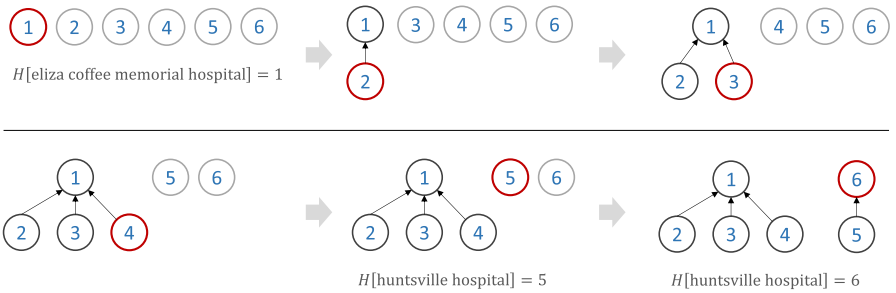

Fig. 3. Different update steps of DSF (#provider) during the repair of hospital name → #provider. In each step, the node with the thick red border contains the tid of the row we observe in that step. The nodes with dark grey border are visited in previous steps. Changes to H are shown below the DSF structure.

DSF (#provider) in such a way that tuples with equal values for hospital name are in the same class. As a result, DSF (#provider) has classes $\{\{1, 2, 3, 4\}, \{5, 6\}\}$ after the update. Figure 3 illustrates the update of DSF (#provider) step-by-step. Initially, each tid is in a separate tree, since it is the first FD we consider with #provider in the right-hand side. As Algorithm 1 iterates over the rows in the partial repair, the mapping H that maps values for hospital name to the roots of the different trees is updated and trees are merged when necessary. This example illustrates the "union by size" optimization, where $\text{union}(\text{id}_1, \text{id}_2)$ chooses the root of the largest tree as the new root. In case both trees have equal size, the root node of either of them can be chosen as the new root. This is one of the optimizations necessary to assure the amortized time complexity we mentioned above.

Fixing violations with repair functions. After updating $\text{DSF}(a)$ for some FD $X \rightarrow a$ with Algorithm 1, $\text{DSF}(a)$ represents classes of tuple identifiers from R^*_{i-1} that must have the same value for a. In other words, $\text{DSF}(a)$ corresponds with set $\{\mathcal{E}_1, \ldots, \mathcal{E}_\ell\}$ such that $\forall i \in \{1, \ldots, \ell\} : \mathcal{E}_i \subseteq R^*_{i-1}[\text{tid}]$. For each such set $\mathcal{E}$, we can get the corresponding tuples from R^*_{i-1} and verify whether or not these tuples have equal values for a. If not, then the tuples violate $X \rightarrow a$, and we fix this violation by application of a repair function defined by $\rho_a : \mathcal{B}(A) \rightarrow A$, where $\mathcal{B}(A)$ is the set of all *bags* (or *multisets*) with elements from A. A repair function maps a bag of values from A onto a single value from the domain A. Of particular interest here are repair functions that are *preservative*, which means that $\rho_a(B) \in B$ for any bag $B \in \mathcal{B}(A)$ [13]. Our interest in preservative repair functions is not coincidental. First, preservative repair functions possess important technical properties: They are *idempotent* ($\rho\{v, \ldots, v\} = v$), they ensure that for any repair R^*, we have $\forall a \in \mathcal{R} : R^*[a] \subseteq R[a]$ (i.e., they do not introduce new values in the repair), and they play an important role in the notion of cardinality-minimal repairs [5]. Second, preservative repair functions have some well-known representatives such as majority voting, weighted voting, and median or min/max selection for domains equipped with an order relation. Third, we will show in the following that they play a key role in the avoidance of revisions for unary FDs. Finally, from an application perspective, preservative functions are intuitive in the sense that we are assuming at least one of the values in the bag B is the correct one. In other words, we are assuming that not all values are wrong and the correct value can be retrieved by making a (smart) choice. This assumption makes sense especially when dealing with categorical data. Needless to say, there are applications where choice might not be the best option. For example, in a dataset where we have a location identifier (lid), latitude (lat), and longitude (long) and we have the FDs lid → lat and lid → long, then the arithmetic average might be a good repair function for both lat and long, and this is not a preservative repair function.

ALGORITHM 2: Fix violations of $X \longrightarrow a$

```
1:  procedure FIX(R*_{i-1}, X → a, DSF(a))
2:      UPDATE(R*_{i-1}, X → a, DSF(a))                    ▷ Algorithm 1
3:      fixes ← 0                                          ▷ Count violations
4:      for E ∈ DSF(a) do                                  ▷ For each class in DSF(a)
5:          R*_E ← {r | r ∈ R*_{i-1} ∧ r[tid] ∈ E}
6:          if |R*_E[a]| > 1 then                          ▷ Violation check
7:              v_fix ← ρ_a(R*_E⟦a⟧)                       ▷ Apply repair function
8:              exec update R*_{i-1} set a = v_fix where tid ∈ E   ▷ Fix violation
9:              fixes++
10:         end if
11:     end for
12:     return fixes
13: end procedure
```

Algorithm 2 provides the pseudo-code for the resolution of violations of $X \longrightarrow a$. It first updates the DSF (a) by using Algorithm 1. Then, for each class of tuple identifiers in the updated DSF (a), it gets the tuples from R^* with identifiers in this class and stores these tuples in a variable $R^*_{\mathcal{E}}$ (line 5). These tuples are inspected for their values for a (line 6). If a violation is observed, then the repair function for a is applied to the bag of values $R^*_{\mathcal{E}}[\![a]\!]$ and the result is stored in the variable v_{fix} (line 7). This value is assigned to attribute a for all tuples of R^*_{i-1} involved in the violation (line 8).

Example 7. In Example 6, we showed that DSF (#provider) has two equivalence classes. Suppose we choose as repair function majority voting with random tie breaking, then fixing hospital name $\longrightarrow$ #provider must change the value of #provider for tuple 4 into 10006. Moreover, the values of #provider for tuples 5 and 6 must be equated.

Priority repair. We now have all components in place to present an algorithm that performs priority repair on a class of attributes C_i. The pseudo-code for this algorithm is shown in Algorithm 3. The algorithm receives a class of attributes C_i and the corresponding set of FDs $\Phi_i := \Phi[\mathcal{R}_i] \setminus \Phi[\mathcal{R}_{i-1}]$, which contains only those FDs that are possibly still violated in R^*_{i-1}. The algorithm starts with sorting the FDs according to the principles explained before: We prioritize pilot FDs (see Equation (5)) followed by the other FDs, which are sorted by their right-hand side attribute based on the cardinality of the set Changes(.). FDs are added to a stack that maintains this order *at all time* (line 2).

Next, we initialize a DSF structure for each $a \in C_i$. This initialization creates a singleton class for each $r \in R^*_{i-1}$ and thus requires $|C_i| \cdot |R^*_{i-1}|$ makeset operations (line 3). We then start a loop that continues until the stack with FDs is empty. In each iteration, we poll the first FD from the stack and perform two steps: the fix step and the revision step. In the *fix* step, we retrieve the current DSF structure for RHS (ϕ), and we apply Algorithm 2 to fix any violations (line 6). In the *revision* step, we verify if the fix step made changes to R^*_{i-1}. If this is the case, then the values for RHS (ϕ) in R^*_{i-1} have been modified. We then check if there are FDs ϕ' that have RHS (ϕ) in the left-hand side and add those FDs back to the stack to check if new violations have been introduced (line 9). This continues until no new violations are found.

Algorithm 3 can now be attributed with the following properties: First, for each attribute a, the DSF structure is initialized only once, and during each update (Algorithm 1), we change DSF (a) only by merging classes. In particular, this means that DSF (a) *before* update is a partition refinement of DSF (a) *after* update. As such, any FDs that were previously repaired and have a in the

ALGORITHM 3: Repair FDs Φ_i by changing attributes C_i to obtain partial repair R_i^*

Require: C_i is a class of $\mathcal{P}$ such that $\mathcal{P} \rightsquigarrow_F \Phi$
1: **procedure** PRIORITYREPAIR(R_{i-1}^*, Φ_i, C_i)
2: **init** ($\mathbb{S}, \Phi_i$) ▷ Sorted stack of FDs
3: $\forall a \in C_i$: **init** (DSF (a)) ▷ Initialize DSF with singleton classes
4: **while** $\mathbb{S} \neq \emptyset$ **do**
5: $\phi \leftarrow$ **poll** ($\mathbb{S}$)
6: fixes $\leftarrow$ FIX(R_{i-1}^*, ϕ, DSF (RHS (ϕ))) ▷ Fix step
7: **if** fixes $\neq 0$ **then** ▷ Revision step
8: **for** $\phi' \in (\Phi_i \setminus \mathbb{S})$ **do**
9: **if** RHS (ϕ) $\in$ LHS (ϕ') **then**
10: **add** ($\mathbb{S}, \phi'$)
11: **end if**
12: **end for**
13: **end if**
14: **end while**
15: **return** R_{i-1}^*
16: **end procedure**

right-hand side remain satisfied. For FDs that have a in the left-hand side, this might not be the case, and these FDs are added to the stack for revision (only if fixes > 0). During such a revision, DSF (a) can again only change by merging classes, which means that, in consecutive revisions, the number of classes in DSF (a) is monotonically decreasing. This leads to the following theorem:

THEOREM 2. *Let $\mathcal{P} = [C_1 \dots C_m]$ be an ordered partition of $\mathcal{R}$ such that $\mathcal{P} \rightsquigarrow_F \Phi$ for a set of FDs Φ. Then, for any class C_i from $\mathcal{P}$ and any partial repair R_{i-1}^*, we have (i) execution of PRIORITYREPAIR$\left(R_{i-1}^*, \Phi_i, C_i\right)$ (Algorithm 3) terminates, and (ii) after termination R_{i-1}^* is modified into a partial repair R_i^*.*

Theorem 2 states that even in the case of revisions, we eventually end up with a partial repair R_i^*. Nevertheless, the presence of revisions comes with a cost, as it requires a new update of the DSF structure and potential modifications to the repair. It is therefore clear that avoiding revisions has a computational advantage. In that regard, we note that whenever $|C_i| = 1$, all FDs are necessarily pilot FDs and thus never require revision. Moreover, in the case of unary FDs, revision is not needed if the repair function for the single left-hand side attribute is preservative.

PROPOSITION 2. *Let $\mathcal{P} = [C_1 \dots C_m]$ be an ordered partition of $\mathcal{R}$ such that $\mathcal{P} \rightsquigarrow_F \Phi$ for a set of FDs Φ. Then, for any class C_i from $\mathcal{P}$ and any partial repair R_{i-1}^*, a unary FD $a' \rightarrow a$ can be ignored in the revision step of PRIORITYREPAIR$\left(R_{i-1}^*, \Phi_i, C_i\right)$ if $\rho_{a'}$ is preservative.*

The proposition above shows that if we use preservative repair functions, then repair of unary FDs is extremely efficient, as it is revision-free.

4.4 The Swipe Algorithm

The ability to (i) construct a maximally refined ordered partition that is forward repairable and (ii) repair classes of such an ordered partition by means of a priority model can now be used to construct a repair algorithm for FDs that we call Swipe. The pseudo-code to repair a dirty relation R in this way is shown in Algorithm 4. The algorithm starts by computing a minimal cover of FDs to remove any redundant information that is present (line 2). Based on such a minimal cover, we

ALGORITHM 4: Swipe

```
1: procedure SWIPE(R, Φ)
2:     Φ̲ ← MINIMALCOVER(Φ)                                  ▷ Minimal cover
3:     P ← BUILDFROMPREORDER(Φ̲)                              ▷ Construct P from preorder P+ on Φ
4:     R*_0 ← R                                              ▷ Initialize repair
5:     for i ∈ {1, ..., |P|} do
6:         R*_i ← PRIORITYREPAIR(R*_{i−1}, Φ̲[R_i] \ Φ̲[R_{i−1}], C_i)   ▷ Priority repair on C_i for new FDs
7:     end for
8:     return R*_m
9: end procedure
```

construct an ordered partition $\mathcal{P} = [C_1 \dots C_m]$ induced by the preorder P^+ (line 3), as explained in Section 4.2. We initialize the repair with the original dirty relation R. This repair is then modified step-by-step. More precisely, we iterate over the classes C_i of $\mathcal{P}$ in order and in each step apply the priority repair algorithm, where we give the current partial repair R^*_{i-1} as input, together with those FDs that involve at least one attribute from C_i (line 6). During the execution of the priority repair algorithm, R^*_{i-1} is modified by assigning values to attributes in C_i in such a way we get a new partial repair R^*_i. Because $\mathcal{P}$ is forward repairable, we know that such a modification is possible in each iteration. Eventually, we obtain a repair $R^* := R^*_m$ that satisfies all FDs.

As a final remark, we note that, so far, we have assumed that all values of an attribute a are constants from its domain A. In the presence of NULL values, the proposed techniques remain valid, but the procedures explained previously require the following minor modifications: First, any NULL value in the dirty relation R is replaced with a unique indexed variable x_i, as is custom in applications of the Chase algorithm. Second, verification that $r[X]$ is present in H (Algorithm 1, line 4) must account for the presence of variables. More specifically, $r[X] = r'[X]$ if for any attribute in X, r and r' either have the same constant or the same variable. A similar adaptation of tuple equality is needed in Algorithm 2 on line 6. Third, the application of a repair function (Algorithm 2, line 7) must account for variables as follows: If $R^*_{\mathcal{E}}[\![a]\!]$ contains only variables, then we do not apply ρ_a but instead assign the variable with the lowest index to v_{fix}. Else, we remove all variables from $R^*_{\mathcal{E}}[\![a]\!]$ prior to application of ρ_a. This ensures that (i) we maintain the correct equivalence classes in $\mathsf{DSF}(a)$ even in the presence of NULL values, and (ii) we apply repair functions only to bags of constant values.

5 Experimental Evaluation

In this section, we report the results of an empirical study that scrutinizes the efficiency and accuracy of Swipe. Because the intent of the article is to optimize the tradeoff between repair quality and computational cost in a Chase-like procedure, we are primarily interested in comparing Swipe (a single-sequence Chase procedure) with Llunatic (a multi-sequence Chase procedure). We also compare Swipe with HoloClean, as this method combines the notion of constraints with principles from machine learning. The goal of our experiments is to seek evidence for the hypothesis that Swipe allows to find good repairs of FD violations in an efficient manner and that exploring many repairs does not necessarily improve the quality of the final repair. To that end, the study assesses both the quality of repairs in terms of precision, recall, and F-score (Section 5.3) as well as the scalability of Swipe in terms of run time (Section 5.4). All experiments reported below were executed on a machine running Ubuntu 20.04.1 with 64 GB of RAM memory, 12 cores (i9-10920X/3.50 GHz), and two 500 GB Solid State Disks in mirror (WDS500G3XHC-00SJG0).

Table 2. Summary of Datasets used in the Experiments. Repairable Attributes are Attributes that Occur in at Least One FD. Cells in Error are Cells for which the Value is Different in R_{gold} Compared to R

	Hospital	Allergen	Eudract	Flight
# rows	1,000	1,160	86,670	776,067
# rows w. ground truth	1,000	206	3,133	70,951
# attributes	19	22	9	5
# repairable attributes	15	22	9	5
# FDs	15	21	11	4
# cells in error	509 (2.68%)	358 (8.28%)	2,962 (11.82%)	123,799 (43.62%)
# cells with NULL	0	0	1,722 (6.87%)	59,837 (21.08%)

5.1 Datasets

Real-life datasets. The study involves four real-life datasets, which differ in size, number of errors, and number of considered constraints. Two of them (Hospital [21, 30, 47] and Flight [11, 30, 38]) are commonly used in experiments on data quality, while the others have been composed and used in previous work by the authors. Each dataset is accompanied by a gold standard that contains the correct tuples for a sample of the dataset.

Table 2 provides a summary of the dataset descriptions, including the size of the gold standard and the number of constraints considered in our experiments. More details on the origin of these datasets and their gold standard, as well as full downloads, are openly available.[3] A short description for each dataset is included below.

— **Hospital.** A benchmark dataset often used in the literature on data cleaning [21, 47]. The original data [21] stems from the U.S. Department of Health and Human Services.[4] We use the version consisting of 1,000 tuples that has been used before in data-repairing experiments [47]. The running example from Figure 1 is a sample from this dataset.
— **Allergen.** A dataset containing data on food product allergens as found on two different websites [11]. The attributes indicate the presence ("2"), traces ("1"), or absence ("0") of allergens in a product, which can be identified by its barcode. A gold standard was compiled by manual consultation of pictures of a sample of products [11].
— **Eudract.** A dataset with data on clinical trials conducted in Europe [11, 12] and obtained from the Eudract register.[5] A gold standard was compiled based on the information of the same studies in other registries.
— **Flight.** A dataset that describes flights annotated with the departing and arrival airports as well as their expected and actual time of departure and arrival. The original dataset was used in the context of data fusion [23, 24, 38] and is available online.[6] In the current study, we use the version and gold standard proposed in recent work [11].

Artificial datasets. To assess scalability, artificial datasets (i.e., an artificial relation R paired with a set of constraints Φ) are used in Section 5.4. The process of generating these datasets takes two parameters: the number of FDs (i.e., $|\Phi|$) and the number of tuples (i.e., $|R|$). We assume that the

[3] https://gitlab.com/antoonbronselaer/swipe-reproducibility
[4] http://www.hospitalcompare.hhs.gov
[5] https://eudract.ema.europa.eu/
[6] http://lunadong.com/fusionDataSets.htm

number of attributes is equal to the number of FDs and thus that $|\mathcal{R}| = |\Phi|$. The reason for doing so is two-fold. First, increasing only $|\mathcal{R}|$ while keeping $|R|$ and Φ fixed would simply add singleton partition classes C_i to $\mathcal{P}$ for which $\underline{\Phi}[\mathcal{R}_i] \setminus \underline{\Phi}[\mathcal{R}_{i-1}]$ is empty, and this would not affect the run time for Swipe. Second, when increasing the number of FDs, we want to sample from a larger set of attributes to reduce the probability of generating *redundant* FDs that are removed when generating $\underline{\Phi}$. We then generate tuples by randomly assigning values to attributes under the condition that $|A| = 10$ for each attribute a. By choosing the domain size sufficiently small, we ensure that there are many violations of FDs that need repairing. Clearly, if we would increase domain sizes $|A|$, then we would obtain less violations and observe faster run times. To generate FDs, we first choose the size of the left-hand side by sampling from a uniform distribution $\mathsf{unif}\{1, \lceil|\mathcal{R}|/10\rceil\}$. This reflects that for larger schemata, it becomes more likely to have FDs with multiple attributes in the left-hand side. In the case that we want to generate unary FDs only, we skip this step and fix the left-hand side size to 1. We then sample for that FD the required number of attributes uniformly from $\mathcal{R}$ to compose the left-hand side. The right-hand side attribute is chosen from the remaining attributes. When a relation R and a set of FDs have been generated, we apply Swipe to obtain R^*.

5.2 Repair Techniques

We compare Swipe with Llunatic [30] and HoloClean [47]. We emphasize here that both Llunatic and HoloClean are repair tools that can deal with constraints more general than FDs. In that sense, the comparison we make here and the conclusions we draw from it apply to the specific scenario of FD repairing with updates, whereas both Llunatic and HoloClean have a wider scope than this single scenario. To use Llunatic and HoloClean, the datasets are stored in a PostgreSQL database in separate schemas. We used PostgreSQL version 9.5, because Llunatic relies on the `with oids` option during table creation, which is abandoned as of version 12.[7]

Llunatic. Llunatic is a general-purpose Chase engine that builds a Chase tree where nodes correspond to (sequences of) repair steps, and leaf nodes represent repairs. The implementation allows configuration of (i) pruning strategies to keep the search efficient and (ii) a partial order that indicates preferences of values for repair. We use the most recent version (2.0) available at GitHub.[8] We made a local build of the Llunatic code to be able to run it from the command line on an `lxc` container. We encoded the available FDs for each dataset in separate configuration files. As Llunatic allows for a wide range of configuration options, we created two configurations for each dataset.

The first configuration is the standard one (S) and is the same for each dataset. It searches for forward repairs only, and we set both the branching threshold and the potential solutions threshold to 2. This means that the Chase tree is binary, and we stop searching after two repairs have been found. The partial order to resolve conflicts is based on the frequency of values. The main idea of this standard configuration is that it resembles the search strategy of Swipe. More specifically, it resolves conflicts in the same way as the default method in Swipe (i.e., majority voting); it uses forward repairing only and it constructs few sequences. The main difference is that Swipe constructs a specific order on the FDs and uses no variables to resolve conflicts unless all values are NULL.

The second configuration is a **fine-tuned (FT)** configuration that is tailored to each dataset individually. First, we allowed backward repairing for Hospital, Eudract, and Flight to see if this helps in finding better repairs. We did not allow this for Allergen, as there we wanted to focus on mirroring the behavior of Swipe when using the max repair function. Second, to account for

[7] See https://www.postgresql.org/docs/12/release-12.html, section E.13.2.
[8] https://github.com/donatellosantoro/Llunatic

Table 3. Configuration Details of Llunatic (FT) for Each Dataset

Strategy	Hospital	Allergen	Eudract	Flight
Similarity cost manager	✓			✓
Higher branching threshold	✓		✓	✓
More potential solutions	✓		✓	✓
Backward repairs allowed	✓		✓	✓
Mimic max repair function		✓		

typographical errors that are present in the attributes of Hospital and Flight, we used a cost manager that uses Smith-Waterman string similarity between values to guide resolution. For Hospital, Eudract, and Flight, we also increased the branching threshold such that a larger part of the search space is explored and more repairs are searched for. A summary of the main differences of the FT configurations is given for each dataset in Table 3. The full configuration files are available online and can be used for reproduction purposes.[3]

HoloClean. HoloClean is a two-step learning-based approach that first constructs a probabilistic model of the data (learning phase) and then uses that model to compute repairs (repair phase). The learning phase utilizes different aspects of the data such as violations of integrity constraints and quantitative statistics such as co-occurrence statistics of attribute values. The repair phase uses probabilistic inference to generate the most probable repairs for a given dirty dataset. We selected HoloClean as a comparator because (i) it accepts constraints like FDs as input and can thus be used as an FD repair algorithm, and (ii) it outperforms other approaches that can deal with constraints such as Holistic and NADEEF [11, 43, 47]. We use the most recent version (1.0) available at GitHub.[9] For each dataset, the available FDs are given as input. We configure HoloClean to account for NULL values as well as violations of FDs during the learning phase. Any other parameters were set to their default as mentioned on the GitHub page. This approach is consistent with recent experiments regarding automated data repair that include HoloClean as a comparator [40, 43]. Attributes that are not relevant to repair (e.g., tuple identifiers, source information) are removed from the data prior to the learning phase to avoid that irrelevant co-occurrence statistics negatively influence the performance of HoloClean.

Swipe. Swipe was implemented in Java as part of the ledc framework,[10] which is an open-source framework that bundles various techniques for data quality under one umbrella project. The implementation of Swipe contains a set of basic repair functions. We consider three of them here:

— The first one is ***majority* voting (**mv**)** with random tie breaking. This repair function was used in the examples throughout this article and is considered a default choice.
— The second one is ***weighted* voting (**wv**)** with random tie breaking. The weight of a value in the bag is based on the number of NULL values that occur in the row where that value is taken. More specifically, if a value from the bag is taken from a tuple with N NULL values, then the weight for that value is equal to $(|\mathcal{R}| - N)^4$. We include this repair function, as it has been shown effective in previous work [11].
— The third repair function simply chooses the *maximal* value (max) and can be used in a scenario where attribute domains are equipped with an order relation that is relevant to the

[9] https://github.com/HoloClean/holoclean
[10] https://gitlab.com/ledc/ledc-fundy

repair. This is true for all datasets included in the experiment, except for the Hospital dataset, where using the natural order simply sorts the attribute values alphabetically.

For each dataset, we run Swipe three times and in each run use a different repair function. That repair function is then used for *all* attributes in the dataset. Note that each of these three functions is *preservative*, which means Proposition 2 applies in each scenario we test here.

5.3 Precision and Recall

Results. To examine the quality of repairs, we compare, for each dataset, the given relation R with a repair R^* and a gold standard R_{gold} and compute the following measures:

— **Precision (P)**: the number of correctly repaired cells divided by the number of repaired cells.
— **Recall (R)**: the number of correctly repaired cells divided by the number of erroneous cells.
— **F-score (F)**: harmonic mean of precision and recall.

Hereby, a *repaired* cell is a cell for which the value is different in R^* as compared to R. A *correctly repaired* cell is a repaired cell where in addition the values are the same in R^* and R_{gold}. An *erroneous* cell is a cell for which the value is different in R_{gold} compared to R. As Llunatic uses a multi-sequence Chase method, it provides multiple repairs. We report here the results for the repair that has the lowest cost. Moreover, in the case of Llunatic, R^* may contain named variables (llun-values) that indicate that the cell should be assigned some constant value, but the value remains unspecified. Llunatic offers the option to call on human intervention to fill in the correct constant. To report repair quality in the presence of variables, we propose to consider two extreme cases: one in which all variables are assigned the *correct* constant and one in which all variables are assigned an *erroneous* constant. These cases provide a best and worst outcome of precision, recall, and F-value whenever all variables would be assigned with constant values at random. As such, for Llunatic, we will report *intervals* of repair quality rather than single values. Recent studies on Llunatic have used measures that correspond to our proposed upper bound [30]. However, the inclusion of the lower bound introduces an important dimension to the analysis of results in the sense that a large amount of variables implies a big gap between lower and upper bound. This is important to recognize, as many variables imply many human interventions after the repair has been produced. Table 4 provides an overview of the repair quality obtained by the two configurations of Llunatic, HoloClean, and the three configurations of Swipe on all datasets. For each combination of dataset and measure of quality, we indicate the best-performing approach in bold font. For recall and F-score on the Allergen dataset, we did not indicate a best approach, as it cannot be indicated with certainty.

Discussion. Table 4 reveals several interesting observations that we discuss below. A first observation is that results are sensitive to the exact configurations. This is especially true for the Allergen dataset, which provides allergen information for each product coming from only two sources. This low number of sources makes voting procedures less suitable. The resolution of conflicts by means of the max repair function is a superior choice here, as it accounts for the *semantics* of the data. More specifically, if one source explicitly mentions the presence of an allergen (value 2) or a trace of it (value 1) and the other source does not (value 0), then we are inclined to believe the source with the higher value. This behavior is also encoded in the FT scenario of Llunatic, and it was confirmed that, in this case, Llunatic and Swipe produced exactly the same results. For other datasets, the voting scenarios of Swipe provide robust and good results. When a dataset has no NULL values (Hospital and Allergen), both voting scenarios are equivalent. When NULL values are present (Eudract and Flight), weighted voting produces better results and turns out to be a good baseline choice. From these results, we can say that if there are sufficient values to choose from, then voting strategies are the recommended choice for a repair function, with a preference

Table 4. Overview of Precision (P), Recall (R), and F-score Obtained for Llunatic (Two Configurations), HoloClean, and Swipe (Three Repair Functions). Best Results are Marked in Bold Font. n/a Indicates the Repair Function was not Applicable on that Dataset

	Dataset	Llunatic (S)	Llunatic (FT)	HoloClean	Swipe (mv)	Swipe (wv)	Swipe (max)
P	Hospital	[0.93,0.96]	[0.11,0.12]	**1.00**	0.96	0.96	n/a
	Allergen	[0.00,0.57]	**[0.64,0.64]**	0.08	0.21	0.21	**0.64**
	Eudract	[0.80,0.84]	[0.80,0.84]	**0.92**	0.83	0.84	0.23
	Flight	[0.67,0.72]	[0.80,0.80]	**0.81**	0.71	0.79	0.56
R	Hospital	[0.83,0.85]	[0.29,0.31]	0.47	**0.89**	**0.89**	n/a
	Allergen	[0.00,0.54]	[0.30,0.30]	0.01	0.10	0.10	0.30
	Eudract	[0.31,0.32]	[0.31,0.32]	0.34	0.36	**0.37**	0.32
	Flight	[0.66,0.71]	[0.54,0.54]	0.44	0.68	**0.79**	0.58
F	Hospital	[0.88,0.90]	[0.16,0.17]	0.64	**0.92**	**0.92**	n/a
	Allergen	[0.00,0.55]	[0.41,0.41]	0.01	0.13	0.13	0.41
	Eudract	[0.44,0.47]	[0.44,0.47]	0.50	0.51	**0.52**	0.27
	Flight	[0.67,0.72]	[0.65,0.65]	0.57	0.69	**0.79**	0.57

for weighted voting. In the cases where there are few values involved in violations or where data have particular semantics, a repair function should be used that is customized to the data. For Llunatic, the sensitivity to configuration choices seems to be higher than for Swipe. For the Hospital dataset, there is a large difference in repair quality between the two configurations, most likely due to the usage of backward repairing in the fine-tuned configurations. In general, we found that giving Llunatic the possibility to include backward repairs seems to decrease repair quality on the datasets considered here. Generalizing this observation should be done with caution, but the results reported here make a strong case for at least questioning the usefulness of backward repair on real-life datasets.

A second observation is that when we compare the repair quality of Swipe with that of Llunatic, it can be seen that Swipe produces repairs that are sometimes comparable and usually better than the best possible outcomes of Llunatic. The improvement in F-score is hereby mostly attributed to an improvement in recall, whereas precision behaves comparably. In this regard, it is interesting to see that Llunatic allows the usage of variables (i.e., lluns) when generating repairs. Opposed to that, repair functions used in Swipe always produce constants (unless no constants are involved in a violation). The results in Table 4 show that the more aggressive strategy where we always choose from the available constants usually pays off. Moreover, it can be seen that the number of variables might become very high. This happens, for example, with the Allergen dataset when using the standard configuration of Llunatic. A high number of variables can be problematic, as it requires much human intervention in the repair process. This might, however, be a necessity to obtain good repairs. Indeed, we see that for the Allergen dataset, the upper bound of recall and F-score are much higher for the standard configuration of Llunatic than for the other approaches. When comparing Swipe to HoloClean, similar observations can be made. It is clear that HoloClean (with default parameter values) scores very high at precision. In three out of four cases, HoloClean is superior when it comes to precision. However, the recall is substantially lower and the F-scores of Swipe are better in all cases. It can be seen that HoloClean has difficulties with the Allergen dataset, which can be explained by the fact that absence of an allergen is, on average, far more common than the presence. For each attribute, the value 0 is thus far more frequent than values 1 or 2. However, we have already explained that the presence of an allergen should not be neglected,

Table 5. Mean Execution Times (in Seconds) of the Repair Methods

Dataset	Llunatic (S)	Llunatic (FT)	HoloClean	Swipe (best)
Hospital	20.31	167.09	46.46	**0.20**
Allergen	1.52	264.82	84.35	**0.28**
Eudract	19.38	98.09	4,463.58	**7.15**
Flight	910.05	3,987.90	6,807.54	**17.80**

even if it is mentioned only once. The results for Llunatic (FT) and Swipe (max) confirm this. HoloClean, however, has difficulties in dealing with the skew value distributions featured in the Allergen dataset. Another potential problem with HoloClean is that repairs are not guaranteed to satisfy all FDs, which is an observation also made in other experimental studies [11, 43].

A third observation is found for the Eudract dataset, which is the only dataset where we encounter a partition class with more than 2 attributes. For this class with 3 attributes, Swipe estimates the reliability of the involved attributes and uses that estimate to rank order FDs. Table 4 shows that using this order of FDs eventually produces a repair for which $P = 0.84$ and $R = 0.37$. To investigate how good this order is, we repeated the experiment but forced Swipe to use a different priority model for this class. More precisely, we considered every possible priority order for the involved attributes to compute a best and worst case for precision and recall. We then found that precision ranged between 0.69 and 0.84 and recall ranged between 0.31 and 0.37. These additional results show that the order in which we repair FDs can indeed make a large difference in the repair quality. However, the sequence based on estimated reliability leads to the best possible repair quality. This is true for the Eudract dataset and also for the Hospital dataset, where we have one partition class with 2 attributes. Also in the latter case, it is confirmed that the chosen order of FDs is the best one, although the difference in quality is less pronounced.

5.4 Scalability

Run-time comparison. In this section, we investigate the run-time efficiency and scalability of Swipe. First, we measure the mean run time over five runs of Llunatic, HoloClean, and Swipe on the different datasets and report the mean execution times (in seconds). We do not report other scalability parameters such as CPU usage or main memory consumption, although we have investigated these parameters and came to similar conclusions. In the case of Llunatic, we differentiate between the standard configuration and the configuration that was fine-tuned for each dataset (Table 3). For Swipe, we took for each dataset the best-performing repair function. For HoloClean, we report the total execution time (learning phase + repair phase). The results from Table 5 show that Swipe outperforms both Llunatic and HoloClean on each dataset in terms of execution time. This comes as no surprise, as Swipe is designed explicitly as a simplification of the multi-sequence Chase algorithm that underlies Llunatic. Yet, we observe that the gain in execution time is considerable. Swipe is two to three orders of magnitude faster than the fine-tuned configuration of Llunatic. When we compare to the standard Llunatic configuration, the difference reduces to one order of magnitude. This provides us with some interesting insights. First, the cost manager of Llunatic does what it is supposed to do: Restricting repair operations to forward repair only and limiting the branching factor significantly reduces the execution time of Llunatic. However, Table 4 shows that this increase in computational efficiency also leads to a lower quality of the repairs. In that sense, Swipe makes more considerate choices in reducing computational cost with better repair quality as a result. Compared to HoloClean, Swipe is two orders of magnitude faster, indicating that there is a considerable gain in efficiency when using Swipe.

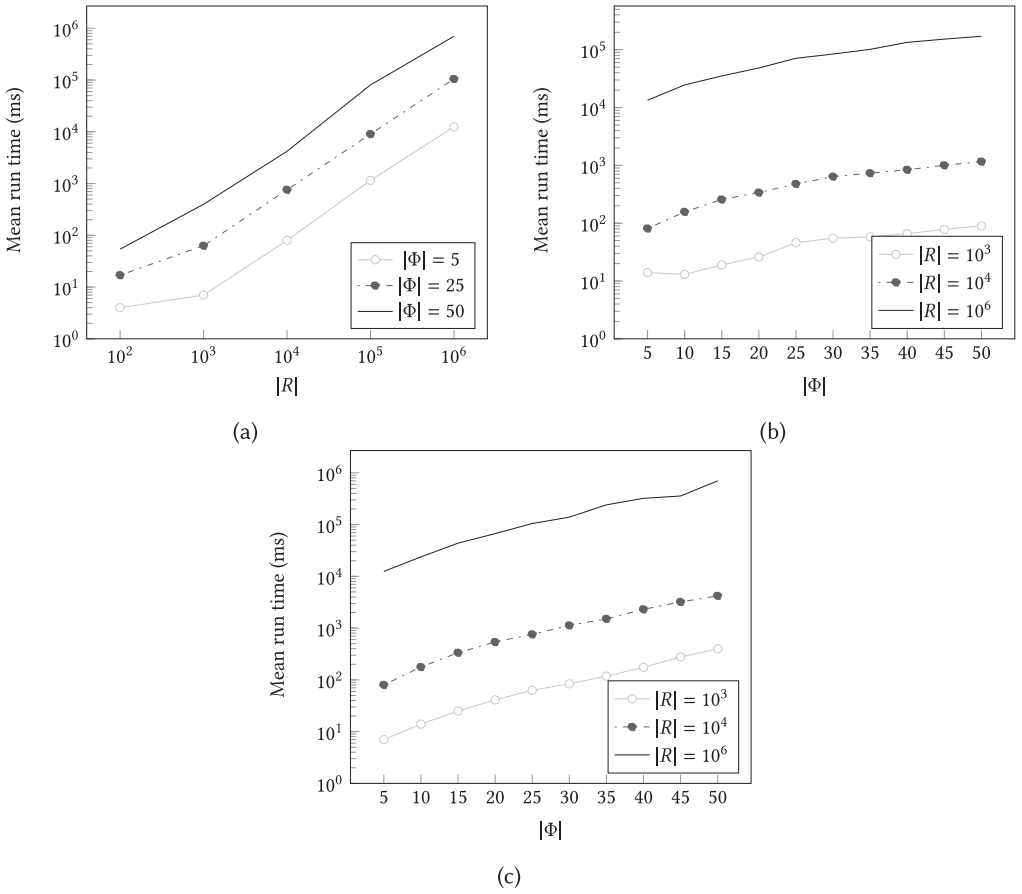

Fig. 4. Mean run time (ms) of 10 executions of Swipe in function of changing $|R|$ (a) and $|\Phi|$ for unary FDs only (b) and general FDs (c). When varying $|R|$ in panel (a), FDs are considered in the general case.

Size of classes in $\mathcal{P}$. The execution time of Swipe is strongly influenced by the size of classes in $\mathcal{P}$. For smaller partition classes, we usually require less revisions. To that extent, we computed the sizes of the partition classes for each of the four datasets and came to the conclusion that over all datasets, there are 47 classes of which 45 where singletons. One class had size 2 (Hospital) and one class had size 3 (Eudract). This observation confirms that, in real-life datasets, it is reasonable to apply the partitioning approach and to separate the repairs of different FDs.

Scalability in terms of $|R|$ and $|\Phi|$. To gain further understanding of the scalability of Swipe, we ran experiments with randomly generated data and constraints (Section 5.1). The repair functions are defaulted to majority voting with random tie breaking for all attributes. We generate repairs of the given dirty datasets with Swipe and then measure the time to generate a repair in milliseconds. For a fixed number of tuples and a fixed number of FDs, we repeat the procedure 10 times and report the mean of the measured run times. Figure 4 shows the mean run times for a varying number of tuples (a) and a varying number of FDs in the unary case (b) and the general case (c). More precisely, panel (a) shows the evolution of the mean run time as the number of tuples increases with powers of 10, starting at $|R| = 10^2$ and ending at $|R| = 10^6$. This trend is shown three times for different numbers of FDs. These results show that the run time of Swipe scales *linearly*

in terms of increasing $|R|$. This trend is confirmed in panels (b) and (c): An increase of $|R|$ with one order of magnitude yields an increase in run time of one order of magnitude. This behavior is not surprising. Increases of $|R|$ affect Swipe during (i) the initialization of DSF structures and (ii) the repair of an FD by means of Algorithm 2. On one hand, initializing a DSF will require $O(|R|)$ time. On the other hand, application of Algorithm 2 for an FD $X \rightarrow a$ will be affected by $|R|$ in two ways. First, the update of DSF (a) has an amortized time complexity of $O(|R| \cdot \alpha(|R|))$ (Section 4.3). Second, fixing violations requires applying a repair function to a bag of values. For simple preservative repair functions like majority voting, this requires linear time in terms of the size of the bag. In the worst case, each $r \in R$ is involved in a violation, and fixing violations requires $O(|R|)$ time. It follows that if $\mathcal{R}$ and Φ are fixed, then the time complexity of Swipe is $O(|R| \cdot \alpha(|R|))$, which is in practice approximately linear.

Figures 4(b) and 4(c) show the evolution of the mean run time as the number of FDs increases with increments of 5, starting at $|\Phi| = 5$ and ending at $|\Phi| = 50$. This trend is again shown three times, now for different sizes of $|R|$. These results show that, in the general case, the shift in run time is two orders of magnitude if we increase the number of FDs with one order of magnitude (i.e., from 5 to 50). This suggests that Swipe scales quadratically in terms of $|\Phi|$. This is confirmed by the results in panel (a) when looking at the differences in $|\Phi|$. We can explain this as follows: If we increase $|\Phi|$, then the dominating factor is again the number of calls to Algorithm 2. There are additional factors such as the construction of the ordered partition and the fact that the same DSF structure might be updated more, but these factors are less influential than the number of calls to Algorithm 2. In the general case, FDs are repaired more than once, leading to an increase in the number of times Algorithm 2 is executed. In the case of unary FDs only, Proposition 2 tells us that the usage of preservative repair functions ensures revisions of FDs are not needed. In other words, the number of times Algorithm 2 must be executed will then grow linearly in terms of $|\Phi|$. This is confirmed in Figure 4(b), where we indeed see a linear trend in function of increasing $|\Phi|$.

In summary, our results show that Swipe is very efficient in repairing large relations when the number of FDs is relatively small or when FDs are unary. For a growing number of FDs, Swipe still repairs relations relatively fast if we compare to other state-of-the art repair methods.

6 Conclusion

In this article, we have introduced the Swipe algorithm to repair violations of FDs. This algorithm is a degenerate variant of the Chase-based approach towards FD repairing. It hinges on two key principles. The first principle is to use an ordered partition of attributes that is forward repairable. We have provided an algorithm that, for a given set of FDs, constructs an ordered partition that is maximally refined and meets the requirement of forward repairability. The second principle is that of priority repairing, which fixes the order in which FDs are treated. We have shown a simple heuristic to build such a priority model based on the estimated reliability of attributes. From a theoretical point of view, we have shown that Swipe is ensured to terminate and that there are easy-to-meet conditions under which unary FDs are revision-free. Empirical results show that Swipe provides an excellent tradeoff between repair quality and computational efficiency. Future improvements of Swipe can focus on applying the principles developed here to more expressive constraints like conditional functional dependencies.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu (Eds.). 1995. *Foundations of Databases* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

[2] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. 1999. Consistent query answers in inconsistent databases. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'99)*. Association for Computing Machinery, New York, NY, USA, 68–79. DOI: https://doi.org/10.1145/303976.303983

[3] Catriel Beeri and Moshe Vardi. 1984. A proof procedure for data dependencies. *J. ACM* 31, 4 (1984), 718–741.
[4] Amir Ben-Amram and Simon Yoffe. 2011. A simple and efficient union-find-delete algorithm. *Theoret. Comput. Sci.* 412 (02 2011), 487–492. DOI: https://doi.org/10.1016/j.tcs.2010.11.005
[5] George Beskales, Ihab Ilyas, and Lukasz Golab. 2010. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB* 3 (09 2010), 197–207. DOI: https://doi.org/10.14778/1920841.1920870
[6] Toon Boeckling, Guy De Tré, and Antoon Bronselaer. 2022. Cleaning data with selection rules. *IEEE Access* 10 (2022), 125212–125229.
[7] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2007. Conditional functional dependencies for data cleaning. In *Proceedings of the IEEE International Conference on Data Engineering*. IEEE, 746–755.
[8] Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. 2005. A cost-based model and effective heuristic for repairing constraints by value modification. In *Proceedings of the SIGMOD Conference*. ACM, 143–154.
[9] Anges Boskovitz. 2008. *Data Editing and Logic: The Covering Set Method from the Perspective of Logic*. Ph. D. Dissertation. The Australian National University.
[10] Bernardo Breve, Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2022. RENUVER: A missing value imputation algorithm based on relaxed functional dependencies. In *Proceedings of the 25th International Conference on Extending Database Technology (EDBT'22)*. OpenProceedings.org, 52–64. DOI: https://doi.org/10.5441/002/edbt.2022.05
[11] Antoon Bronselaer and Maribel Acosta. 2023. Parker: Data fusion through consistent repairs using edit rules under partial keys. *Inf. Fusion* 100 (2023), 101942. DOI: https://doi.org/10.1016/j.inffus.2023.101942
[12] Antoon Bronselaer, Toon Boeckling, and Filip Pattyn. 2022. Dynamic repair of categorical data with edit rules. *Expert Syst. Applic.* 201 (2022), 15. DOI:http://dx.doi.org/10.1016/j.eswa.2022.117132
[13] Antoon Bronselaer and Guy De Tré. 2010. Aspects of object merging. In *Proceedings of the North American Fuzzy Information Processing Society Meeting*. IEEE, 27–32.
[14] Zhui Chen, Lei Cao, Sam Madden, Tim Kraska, Zeyuan Shang, Ju Fan, Nan Tang, Zihui Gu, Chunwei Liu, and Michael Cafarella. 2024. SEED: Domain-specific data curation with large language models. arXiv:2310.00749
[15] Xu Chu, Ihab Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *Proceedings of the International Conference on Data Engineering (ICDE'13)*. IEEE, 458–469.
[16] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data cleaning: Overview and emerging challenges. In *Proceedings of the International Conference on Management of Data (SIGMOD'16)*. Association for Computing Machinery, 2201–2206. DOI: https://doi.org/10.1145/2882903.2912574
[17] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *Proceedings of the IEEE 29th International Conference on Data Engineering (ICDE'13)*. IEEE, 458–469. DOI: https://doi.org/10.1109/ICDE.2013.6544847
[18] Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. 2015. KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. Association for Computing Machinery, 1247–1261. DOI: https://doi.org/10.1145/2723372.2749431
[19] Pinar Cihan and Zeynep Banu Ozger. 2019. A new heuristic approach for treating missing value: ABCimp. *Elektron. Elektrotech.* 25 (2019), 48–54. DOI: https://doi.org/10.5755/j01.eie.25.6.24826
[20] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. 2007. Improving data quality: Consistency and accuracy. In *Proceedings of the VLDB Conference*. ACM, 315–326.
[21] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. 2013. NADEEF: A commodity data cleaning system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, New York, NY, USA, 541–552. DOI: https://doi.org/10.1145/2463676.2465327
[22] Ton De Waal, Jeroen Pannekoek, and Sander Scholtus. 2011. *Handbook of Statistical Data Editing and Imputation*. Wiley.
[23] Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. 2009. Integrating conflicting data: The role of source dependence. In *Proceedings of the VLDB Conference*. ACM, 550–561.
[24] Xin Luna Dong, Barna Saha, and Divesh Srivastava. 2012. Less is more: Selecting sources wisely for integration. In *Proceedings of the VLDB Conference*. VLDB Endowment, 37–48.
[25] Lisa Ehrlinger and Wolfram Wöß. 2022. A survey of data quality measurement and monitoring tools. *Front. Big Data* 5 (2022), 28.
[26] Wenfei Fan and Floris Geerts. 2012. *Foundations of Data Quality Management*. Morgan & Claypool Publishers, Switzerland.
[27] Ivan Fellegi and David Holt. 1976. A systematic approach to automatic Edit and Imputation. *J. Amer. Statist. Assoc.* 71, 353 (1976), 17–35.
[28] Bernard Galler and Michael Fischer. 1964. An improved equivalence algorithm. *Commun. ACM* 7, 5 (1964), 301–303. DOI: https://doi.org/10.1145/364099.364331

[29] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2013. The LLUNATIC data-cleaning framework. *Proc. VLDB Endow.* 6, 9 (2013), 625–636.
[30] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2019. Cleaning data with Llunatic. *VLDB J.* 29 (2019), 867–892. DOI: https://doi.org/10.1007/s00778-019-00586-5
[31] Seymour Ginsburg and Sami Mohammed Zaiddan. 1982. Properties of functional-dependency families. *J. ACM* 29, 3 (1982), 678–698.
[32] Alireza Heidari, Joshua McGrath, Ihab F. Ilyas, and Theodoros Rekatsinas. 2019. HoloDetect: Few-shot learning for error detection. In *Proceedings of the International Conference on Management of Data (SIGMOD'19)*. Association for Computing Machinery, New York, NY, USA, 829–846. DOI: https://doi.org/10.1145/3299869.3319888
[33] Ihab F. Ilyas and Xu Chu. 2015. Trends in cleaning relational data: Consistency and deduplication. *Found. Trends Datab.* 5, 4 (2015), 281–393. DOI: https://doi.org/10.1561/1900000045
[34] Ihab F. Ilyas and Xu Chu. 2019. *Data Cleaning*. Association for Computing Machinery, New York, NY, USA.
[35] Arther Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (1962), 558–562.
[36] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2015. BigDansing: A system for big data cleansing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. Association for Computing Machinery, 1215–1230. DOI: https://doi.org/10.1145/2723372.2747646
[37] Solmaz Kolahi and Laks V. S. Lakshmanan. 2009. On approximating optimum repairs for functional dependency violations. In *Proceedings of the 12th International Conference on Database Theory (ICDT'09)*. Association for Computing Machinery, 53–62.
[38] Xian Li, Xin Luna Dong, Kenneth Lyons, Weiyi Meng, and Divesh Srivastava. 2013. Truth finding on the deep web: Is the problem solved? In *Proceedings of the VLDB Conference*. VLDB Endowment, 97–108.
[39] Ester Livshits, Benny Kimelfeld, and Sudeepa Roy. 2018. Computing optimal repairs for functional dependencies. In *Proceedings of the ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems*. ACM, 225–237.
[40] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: Effective error correction via a unified context representation and transfer learning. In *Proceedings of the VLDB Endowment*, Vol. 13. VLDB Endowment, 1948–1961.
[41] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2019. Raha: A configuration-free error detection system. In *Proceedings of the International Conference on Management of Data*. Association for Computing Machinery, 865–882.
[42] Giansalvatore Mecca, Paolo Papotti, Donatello Santoro, and Enzo Veltri. 2024. BUNNI: Learning repair actions in rule-driven data cleaning. *J. Data Inf. Qual.* 16, 2 (June 2024), 1–31. DOI: https://doi.org/10.1145/3665930
[43] Wei Ni, Xiaoye Miao, Xiangyu Zhao, Yangyang Wu, Shuwei Liang, and Jianwei Yin. 2024. Automatic data repair: Are we ready to deploy? *Proc. VLDB Endow.* 17, 10 (2024), 2617–2630. DOI: https://doi.org/10.14778/3675034.3675051
[44] Wei Ni, Kaihang Zhang, Xiaoye Miao, Xiangyu Zhao, Yangyang Wu, and Jianwei Yin. 2024. IterClean: An iterative data cleaning framework with large language models. In *Proceedings of the ACM Turing Award Celebration Conference*. Association for Computing Machinery, 100–105. DOI: https://doi.org/10.1145/3674399.3674436
[45] Minh Pham, Craig A. Knoblock, Muhao Chen, Binh Vu, and Jay Pujara. 2021. SPADE: A semi-supervised probabilistic approach for detecting errors in tables. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI'21)*. International Joint Conferences on Artificial Intelligence Organization, 3543–3551. DOI: https://doi.org/10.24963/ijcai.2021/488
[46] Abdulhakim Qahtan, Nan Tang, Mourad Ouzzani, Yang Cao, and Michael Stonebraker. 2020. Pattern functional dependencies for data cleaning. *Proc. VLDB Endow.* 13, 5 (2020), 684–697. DOI: https://doi.org/10.14778/3377369.3377377
[47] Theodoros Rekatsinas, Xu Chu, Ihab Ilyas, and Christopher Ré. 2017. Holoclean: Holistic data repairs with probabilistic inference. In *Proceedings of the VLDB Endowment* 10, 11 (2017), 1190–1201.
[48] El Kindi Rezig, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, and Ahmed R. Mahmood. 2017. Pattern-driven data cleaning. http://arxiv.org/abs/1712.09437
[49] El Kindi Rezig, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, Ahmed R. Mahmood, and Michael Stonebraker. 2021. Horizon: Scalable dependency-driven data cleaning. *Proc. VLDB Endow.* 14, 11 (2021), 2546–2554. DOI: https://doi.org/10.14778/3476249.3476301
[50] Shaoxu Song, Yu Sun, Aoqian Zhang, Lei Chen, and Jianmin Wang. 2020. Enriching data imputation under similarity rule constraints. *IEEE Trans. Knowl. Data Eng.* 32, 2 (2020), 275–287. DOI: https://doi.org/10.1109/TKDE.2018.2883103
[51] Robert Tarjan. 1976. Edge-disjoint spanning trees and depth-first search. *Acta Inform.* 6, 2 (1976), 171–185.
[52] Robert Endre Tarjan. 1975. Efficiency of a good but not linear set union algorithm. *J. ACM* 22, 2 (1975), 215–255. DOI: https://doi.org/10.1145/321879.321884
[53] Robert Endre Tarjan and Jan van Leeuwen. 1984. Worst-case analysis of set union algorithms. *J. ACM* 31, 2 (1984), 245–281. DOI: https://doi.org/10.1145/62.2160
[54] Henry Warren. 1975. A modification of Warshall's algorithm for the transitive closure of binary relations. *Commun. ACM* 18, 4 (1975), 218–220.

[55] Jef Wijsen. 2006. Project-join-repair: An approach to consistent query answering under functional dependencies. In *Flexible Query Answering Systems*, Henrik Legind Larsen, Gabriella Pasi, Daniel Ortiz-Arroyo, Troels Andreasen, and Henning Christiansen (Eds.). Springer Berlin, 1–12.

[56] Mohamed Yakout, Laure Berti-Équille, and Ahmed K. Elmagarmid. 2013. *Don't Be SCAREd: Use SCalable Automatic REpairing with Maximal Likelihood and Bounded Changes*. Association for Computing Machinery, New York, USA, 553–564. DOI:~https://doi.org/10.1145/2463676.2463706

Appendix

A Proofs

Proof of Proposition 1. If $\mathcal{P}$ satisfies Equation (2), then we can assign, for any $C_i \in \mathcal{P}$ with partial repair R^*_{i-1} and for any $\phi \in \Phi[\mathcal{R}_i] \setminus \Phi[\mathcal{R}_{i-1}]$, equal values to $\mathsf{RHS}(\phi)$ to satisfy ϕ. □

Proof of Theorem 1. We show first that $\mathcal{P}$ allows sequential forward repairability for Φ (i) and then that $\mathcal{P}$ cannot be refined without losing this property (ii).

(i) If $\mathcal{P}$ is induced by P^+ for a given set Φ, then consider an arbitrary class C_i from $\mathcal{P}$ and any FD $\phi \in \underline{\Phi}[\mathcal{R}_i] \setminus \underline{\Phi}[\mathcal{R}_{i-1}]$. If $\mathsf{RHS}(\phi) \notin C_i$, then $\mathcal{P}$ cannot be induced by P^+, because it would violate either Equation (3) or (4). It follows that we must have $\mathsf{RHS}(\phi) \in C_i$ and thus $\mathcal{P} \rightsquigarrow_F \Phi$.

(ii) Suppose there is a refinement $\mathcal{P}'$ of $\mathcal{P}$ for which $\mathcal{P}' \rightsquigarrow_F \Phi$. This would imply that there is a class from $\mathcal{P}$ that can be split into two disjoint subclasses without breaking forward repairability. In turn, this would imply that there is an equivalence class from $\equiv_{P^+}$ that can be split into two subclasses. Let us denote the equivalence relation in which this split is done by $\equiv_{P'}$. Now, if $\equiv_{P'}$ still contains P, then P^+ cannot be the transitive closure of P and we have a contradiction. Alternatively, if $\equiv_{P'}$ does not contain P, then there is at least one $(b, a) \in P$ that is not accounted for and $\mathcal{P}'$ is not forward repairable. In both cases, we obtain a conclusion that is in contradiction with the premise and it follows no refinement of $\mathcal{P}$ is forward repairable. □

Proof of Theorem 2. We first prove that Algorithm 3 terminates (i) and then that after termination we have obtained a partial repair R^*_i (ii).

(i) To see that priorityRepair $\left(R^*_{i-1}, \Phi_i, C_i\right)$ terminates, note that each time an FD ϕ is considered, either $\mathsf{DSF}(\mathsf{RHS}(\phi))$ does not change or its number of classes decreases. If for all $a \in C_i$, $\mathsf{DSF}(a)$ does not change anymore, then the algorithm terminates. Else, we must reach a point in which each $\mathsf{DSF}(a)$ has only one class. In that case, for each $a \in C_i$, each tuple in R^*_{i-1} gets the same value for a. It follows that in this case, there are no more violations, from which it follows that all $\mathsf{DSF}(a)$ remain unchanged and the algorithm stops.

(ii) Each time an FD $X \to a$ is polled from the stack, the fix step updates $\mathsf{DSF}(a)$ and potentially changes R^*_{i-1} in attribute a. In the revision step that follows, any FD $\phi' \in \Phi_i$ that is not on the stack was polled and repaired before, and there are three options. First, if $a \notin \mathsf{LHS}(\phi')$ and $a \neq \mathsf{RHS}(\phi')$, then clearly this FD is still satisfied. Second, if $a = \mathsf{RHS}(\phi')$, then because $\mathsf{DSF}(a)$ changes only by merging classes, any two tuples with the same value for $\mathsf{LHS}(\phi')$ are still in the same class in $\mathsf{DSF}(a)$ and thus will also have the same value after fixing $X \to a$. Hence, ϕ' remains satisfied. Third, if $a \in \mathsf{LHS}(\phi')$, then we put ϕ' back on the stack for revision. It follows that after application of fix $\left(R^*_{i-1}, X \to a, \mathsf{DSF}(a)\right)$, any FD from Φ_i not on the stack is currently satisfied by R^*_{i-1}. At the same time, at termination time, the stack is empty and thus all FDs are satisfied. □

Proof of Proposition 2. For an FD $a' \to a$ during a revision step, we have that:

(1) If $a' \notin C_i$, then we have $\forall \phi \in \Phi_i : \mathsf{RHS}(\phi) \neq a'$ and consequently $a' \to a$ is never considered during any revision step. We therefore assume that $a' \in C_i$.

(2) $a' \to a$ is considered in a revision step only if it is not an element of $\mathbb{S}$. This means it was fixed already and that two tuples with equal values for a' are in the same class of $\mathrm{DSF}(a)$.
(3) $a' \to a$ is considered in a revision step if the preceding fix step involves an FD $X \to a'$ and is therefore of the form $\textsc{fix}\left(R^*_{i-1}, X \to a', \mathrm{DSF}(a')\right)$. In this proof, we denote the partial repair *before* this fix step as R^*_{before} and *after* this fix step as R^*_{after}. Clearly, $R^*_{\text{after}} \models X \to a'$.
(4) From (1) and (2) we have that $X \to a'$ mentioned in (3) is not a pilot FD. Thus, FDs are polled from the stack in order induced by $\succ$.
(5) The fix step in (3) is preceded by a poll of $X \to a'$, so (2) implies $a \succ a'$.
(6) From (2) and (5), we have that $R^*_{\text{before}} \models a' \to a$. Transitivity of FDs implies $R^*_{\text{before}} \models X \to a$.

Suppose now R^*_{after} fails $a' \to a$, then there must exist two rows $r_1 \in R^*_{\text{before}}$ and $r_2 \in R^*_{\text{before}}$ that are transformed into $r^*_1 \in R^*_{\text{after}}$ and $r^*_2 \in R^*_{\text{after}}$, respectively, for which we have $r^*_1[a'] = r^*_2[a'] \wedge r^*_1[a] \neq r^*_2[a]$. In addition, we know that

$$r^*_1[a'] \neq r_1[a'] \vee r_2[a'] \neq r^*_2[a'],$$

because otherwise $r_1[a'] = r^*_1[a'] = r^*_2[a'] = r_2[a']$, which implies that $r^*_1[a] = r^*_2[a]$ (since $R^*_{\text{before}} \models a' \to a$), and this contradicts our construction of r_1 and r_2. We can then distinguish between two cases.

1 If $r^*_1[X] = r^*_2[X]$, then we have that $R^*_{\text{after}} \not\models X \to a$ and because $a' \notin X$, we have $R^*_{\text{before}}[X] = R^*_{\text{after}}[X]$. As such, it follows that $R^*_{\text{before}} \not\models X \to a$, which contradicts with (6).

2 If $r^*_1[X] \neq r^*_2[X]$, then because $\rho_{a'}$ is preservative, there must exist rows $r_3 \in R^*_{\text{before}}$ and $r_4 \in R^*_{\text{before}}$ such that, on one hand:

$$r_1[X] = r_3[X] \wedge r^*_1[a'] = r^*_3[a'] = r_3[a'],$$

and on the other hand:

$$r_2[X] = r_4[X] \wedge r^*_2[a'] = r^*_4[a'] = r_4[a'].$$

That is, r_1 and r_2 received their values for a' from, respectively, r_3 and r_4, and this is only possible if r_1 and r_3 have the same value for X and r_2 and r_4 have the same value for X. Since we assumed $r^*_1[a'] = r^*_2[a']$, it follows that $r_3[a'] = r_4[a']$ and because $R^*_{\text{before}} \models a' \to a$, we must also have $r_3[a] = r_4[a]$. Finally, since $R^*_{\text{before}} \models X \to a$, we also find that $r_1[a] = r_3[a]$ and $r_2[a] = r_4[a]$ by which we find that $r_1[a] = r_2[a]$. Again, this contradicts our construction of r_1 and r_2. □

Received 14 March 2023; revised 29 November 2024; accepted 22 December 2024