Horizon: Scalable Dependency-driven Data Cleaning

El Kindi Rezig
MIT CSAIL
elkindi@csail.mit.edu

Mourad Ouzzani
Qatar Computing Research Institute
mouzzani@hbku.edu.qa

Walid G. Aref
Purdue University
aref@cs.purdue.edu

Ahmed K. Elmagarmid
Qatar Computing Research Institute
aelmagarmid@hbku.edu.qa

Ahmed R. Mahmood
Purdue University
amahmoo@cs.purdue.edu

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

ABSTRACT

A large class of data repair algorithms rely on integrity constraints to detect and repair errors. A well-studied class of constraints is Functional Dependencies (FDs, for short). Although there has been an increased interest in developing general data cleaning systems for a myriad of data errors, scalability has been left behind. This is because current systems assume data cleaning is performed offline and in one iteration. However, developing data science pipelines is highly iterative and requires efficient cleaning techniques to scale to millions of records in seconds/minutes, not days. In our efforts to re-think the data cleaning stack and bring it to the era of data science, we introduce *Horizon*, an end-to-end FD repair system to address two key challenges: (1) Accuracy: Most existing FD repair techniques aim to produce repairs that minimize changes to the data that may lead to incorrect combinations of attribute values (or patterns). *Horizon* leverages the interaction between the data patterns induced by the various FDs, and subsequently selects repairs that preserve the most frequent patterns found in the original data, and hence leading to a better repair accuracy. (2) Scalability: Existing data cleaning systems struggle when dealing with large-scale real-world datasets. *Horizon* features a linear-time repair algorithm that scales to millions of records, and is orders-of-magnitude faster than state-of-the-art cleaning algorithms. A benchmark of *Horizon* against state-of-the-art cleaning systems on multiple datasets and metrics shows that *Horizon* consistently outperforms existing techniques in repair quality and scalability.

PVLDB Reference Format:
El Kindi Rezig, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, Ahmed R. Mahmood, and Michael Stonebraker. Horizon: Scalable Dependency-driven Data Cleaning. PVLDB, 14(11): 2546 - 2554, 2021.
doi:10.14778/3476249.3476301

1 INTRODUCTION

Current data cleaning systems do not scale well with large datasets. The reason is that they are designed to support one-shot and offline data cleaning. However, emerging data science applications pose new scalability requirements that current data cleaning systems do not deliver. In light of our collaborations with data scientists at several organizations, we observe that (1) Data preparation for large datasets takes the bulk of the human effort; (2) Specificity: Because it is important to know the input/output of each stage in the pipeline to facilitate debugging, various data preparation tools are loosely connected to address specific problems in the data (e.g., inputting missing values). Because it is hard to debug, data scientists rarely use a one-box data cleaning system that strives to clean all types of data errors [16, 26]; (3) Iterativeness: Developing data science pipelines is highly iterative and data scientists run their pipelines dozens of time to refine them [29].

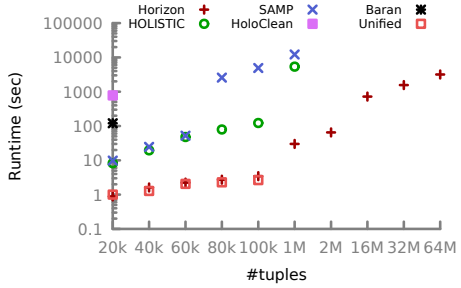


Figure 1: Runtime of state-of-the-art cleaning systems vs. *Horizon*

In response to the above observations, we envision that building data cleaning systems that are tailored to specific data errors, as opposed to general-purpose data cleaning systems, is more amenable to efficient implementations and is easier to debug and tune. As a result, our approach to data cleaning is to re-think common data quality problems and build scalable *by design* techniques to address them. The importance of scalability of data cleaning in a data science setting is twofold: (1) Support larger datasets, and (2) Allow data scientists to refine their pipelines by enabling faster workflow iterations. Following this direction, we develop a data cleaning system for a specific, yet common, data inconsistency problem, namely, functional dependency violations, that is efficient *by design*.

A functional dependency (FD) $X \rightarrow Y$ defined over a relation R that has attributes X and Y states that records sharing the same value in X must share the same value in Y (e.g., $zip \rightarrow state$). FDs are some of the most fundamental and well-studied integrity constraints. This is because their syntax is easy to understand and they can be expressed in a variety of languages (e.g., SQL). Additionally, they form the basis for more expressive rules, e.g., Conditional FDs (CFDs) [17] and Denial Constraints (DCs) [10].

Although extensive efforts have been proposed to clean dirty data, the focus has mainly been on supporting more error types rather than on developing scalable solutions to existing data quality problems [24, 25]. Case in point, it still takes hours/days to even

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476301

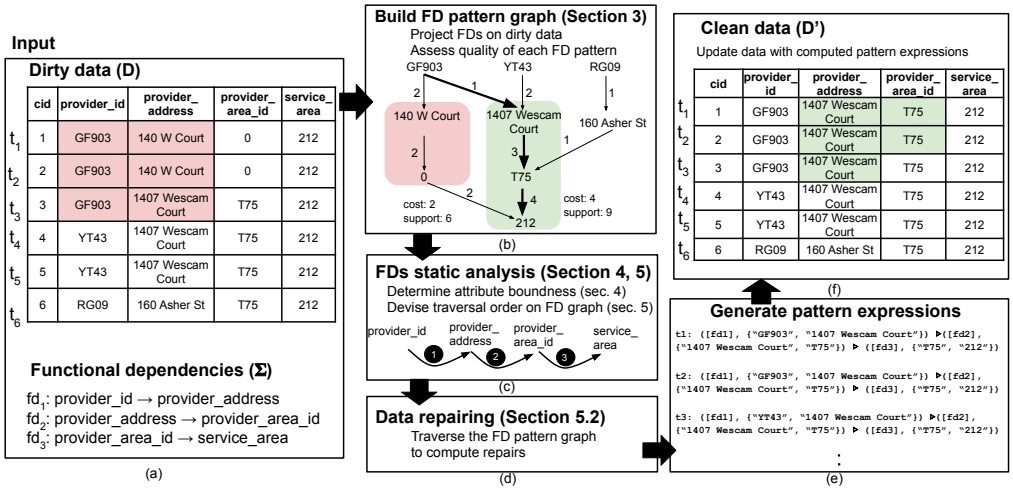


Figure 2: *Horizon* at a glance. ⓑ *Horizon* computes the FD pattern graph. ⓒ Performs static analysis on the FDs to devise a traversal order of the graph in Step ⓑ. ⓓ Repairs the input tuples and presents the results to the user in ⓕⓔ

enforce simple FD rules on small datasets. Figure 1 reports the runtime of various state-of-the-art data cleaning systems to repair FD violations on a real-world dataset (refer to Section 6 for further detail on baselines and setup). It is clear that current systems do not scale well with regard to dataset size. Even for moderately sized datasets (e.g., 1M), most of the baselines do not terminate within 24 hours on a 32GB memory machine with an 8-core CPU.

In this paper, we introduce *Horizon*, an end-to-end FD repair system that efficiently cleans data using a novel cost model that preserves the frequent patterns found in the input data. *Horizon* outperforms state-of-the-art cleaning systems on both effectiveness of FD repairs and efficiency. In the remainder of this section, we outline the intuition behind *Horizon*'s repair strategy, and discuss limitations of existing methods through a motivating example. In the subsequent sections, we explore *Horizon*'s repair model in more detail, and how it is amenable to an efficient implementation.

Modeling Value Combinations. One way to capture value combinations that bind semantically-related attributes is through FDs. When instantiated on the data, these FDs form data patterns that bind together semantically-related data values. For instance, in Figures 2a-b, the pattern [provider_address= "1407 Wescam Court", provider_area_id = "T75"] is a binding of data values [provider_address= "1407 Wescam Court"] and [provider_area_id = "T75"] through fd_2. Consequently, every FD generates a set of patterns. We refer to these patterns as *FD patterns*. We propose to extract FD patterns from the dirty data and reason about their quality and interactions to compute data repairs. Since correct data is a genuine representation of reality, correct values will usually maintain some patterns based on their distribution and relationships [25, 32]. Therefore, we want to pick repairs that result in patterns that are well-supported in the data (e.g., [provider_address= "1407 Wescam Court", provider_area_id = "T75"] is more frequent than [provider_address= "140 W Court", provider_area_id = "0"].

Example 1. *Table D (Figure 2a) is a data snippet based on a collaboration we have with an organization (Company X) that connects customers to providers for various services. D has five attributes: (1) cid: Customer identifier; (2) provider_id: Service provider identifier; (3) provider_address: Address of the provider; (4) provider_area_id: Identifier of a provider's area; (4) service_area: Customer's area code. We use the FDs: fd_1 : provider_id → provider_address (Records with the same provider id must have the same provider address), fd_2 : provider_address → provider_area_id (Records with the same provider address must have the same provider area id) and fd_3 : provider_area_id → service_area (Records with the same provider area id must have the same service area code). In Figure 2a, the cells involved in the violation of fd_1 are highlighted.*

Given a dirty table and FDs (Figure 2a), *Horizon* creates a graph, termed FD Pattern Graph (FDG) (Figure 2b), that combines all the FD patterns in the data, and then selects the repairs with maximal support. In Figure 2b, each edge is an FD pattern and its weight corresponds to the number of records that support the pattern in table D. Most existing repair algorithms rely on the *minimality* cost model, i.e., picking the repairs that result in the least changes to the data. However, minimality falls short when dealing with patterns (instead of individual cells). For instance, in Figure 2b, the pattern [provider_address = "1407 Wescam Court", provider_area_id = "T75"] is more frequent than the pattern [provider_address = "140 W Court", provider_area_id = "0"]. However, the repair (or path) that selects the frequent pattern costs more (in terms of updates) than the alternative path ($4 > 2$), but has higher support ($9 > 6$). While there is no guarantee that the path with the highest support is always the correct one, our empirical evaluation clearly shows that the pattern granularity offers more context for selecting high-quality repairs than cells seen in isolation.

Repair Side Effects. When a repair algorithm is faced with conflicting values, oftentimes every choice will affect the underlying patterns in the data. For instance, in Example 1, choosing "140 W Court" over "1407 Wescam Court" (for cell $t_3[provider\_address]$) to repair the fd_1 violation will result in incorrect patterns, e.g., [provider_address = "140 W Court", provider_area_id = "0"]. *Horizon* considers how a repair choice for one FD affects the patterns of subsequent FDs (Figure 2b).

Typically, FD repair algorithms work in two phases: (1) Error detection which detects all the violating tuple pairs, and (2) Repairing that typically focuses on minimizing the changes to the violating cells so as to satisfy all the FDs, which constitutes a hard optimization problem [21]. *Horizon* disrupts this traditional workflow by (1) Using FDs as generative rules to produce a graph that encodes the FD-induced data patterns and their interactions; (2) Static analysis on the FDs to dissect the interactions among them (Figure 2c); and (3) Traversing the graph in linear time to select patterns that are "most supported" in the data (Figure 2d).

Representation. To ease their readability, *Horizon* presents repair results in an intermediate representation that we refer to as *pattern expressions* that shows the lineage of a given repair, instead of showing isolated cell updates generated by repairs. Example pattern expressions are given in Figure 2(e), where each output record is a "composition" of several FD patterns. The composition operator ▷ "joins" two FD patterns sharing the same attribute value.

Contributions.

1. We propose FD patterns to model value combinations and their interactions through *FDs*. We compile FD patterns into a graph data structure (FDG) within *Horizon* to clean the input data (Section 3.2).

2. We present measures for *Horizon* to reason about the quality of the FD patterns in the FDG that captures the intrinsic quality of the FD patterns and the ones they lead to (Section 3.3).

3. We transform the cleaning problem into an FD pattern mapping problem and develop algorithms in *Horizon* to generate repairs in linear time in the size of the data and FDs (Sections 4, 5).

4. We conduct a thorough experimental study to assess the performance of *Horizon* against a variety of state-of-the-art data repairing algorithms (Section 6). We evaluate *Horizon*'s scalability using various datasets including a real-world dataset with 64M records.

2 RELATED WORK

For simplicity, we classify the repairing methods into two categories: (1) **Rule-based [18, 21]**: These are the most related to our work. They produce an instance that is consistent with a set of constraints (e.g., FDs, CFDs). In this category, we have methods that focus solely on FDs (FD-centric), and those that address other types of rules (e.g., CFDs, DCs), which might encompass FDs (non-FD-centric). (2) **General**: These are general data cleaning methods [15, 25, 30] that were not designed to address a particular type of data errors, but strive to repair any cell that might contain an error.

Rule-based Data Cleaning. Existing rule-based data repairing techniques focus on computing repairs that minimally change the data instance to satisfy a set of rules, e.g., FDs [6, 8, 9, 14, 20, 23], Conditional FDs [6, 12, 13, 19], Fixing Rules [31], Order Dependencies [27], and Denial Constraints [10]). For instance, *SAMP* [6] produces repairs of FDs and CFDs by sampling from the space of possible repairs. *Holistic* repairs violations of denial constraints by leveraging the overlap between violating cells. Most of these methods use the subset of violating cells to find repairs.

Horizon provides a significant addition to this family of algorithms from the way *Horizon* models the data (the FD patterns) to the way it maps patterns to each other to produce a repaired instance efficiently. Furthermore, unlike existing rule-based solutions, *Horizon* benefits from evidence from all the data values in the dirty instance, including those that are not involved in violations.

[7, 9] address the problem of repairing the FDs in addition to data. For instance, *Unified* [9] decides whether it is best to repair the data, or repair the FDs by computing support measures for data patterns. However, in addition to their inherent inefficiency, the repairing cost model is still minimality.

General Data Cleaning. *HoloClean* [25] uses probabilistic inference to produce repairs based on different signals (e.g., constraint violations). Repairs are associated with marginal probabilities that reflect their accuracy. Another recent effort is *Baran* [24] that combines various data correction models to clean data cells. *Horizon* is different from *HoloClean* and *Baran* in three ways: (1) They both have to know which cells are erroneous, and this could be hard to get whereas *Horizon* has to find out automatically which cells might contain errors through the FDs; (2) They were both designed for generality, i.e., to clean a large class of data errors. This generality puts them at a disadvantage compared to *Horizon* which is tailored to deal with FD errors, allowing it to benefit from their interactions to produce repairs. (3) They do not produce repairs that are necessarily consistent w.r.t. a set of constraints.

SCARE [32], a probabilistic approach that relies on predicting attribute values given the data distribution. However, the user's feedback is needed to assess the quality of the repairs. In addition, *SCARE* is not bound by any data quality rules. *KATARA* [11] is a cleaning system that employs external, curated knowledge bases in addition to crowdsourcing to derive repairs. Both methods are different in scope in contrast to *Horizon*.

3 COMPUTING THE FD PATTERN GRAPH

3.1 Background

Let R be a relational schema of a data instance I. Let $A = \{A_1, A_2, ..., A_n\}$ be the set of attributes in R with active domains $dom(A_1), dom(A_2), ..., dom(A_n)$ respectively. Let Σ be the set of functional dependencies (FDs) defined over R. We assume that Σ is minimal and in canonical form [2]. An FD fd_i in Σ ($i < |\Sigma|$) has the format $X \rightarrow Y$, where $X, Y \in A$. X and Y are referred to as the antecedent and consequent attributes, respectively. Let $Left(fd_i)$ and $Right(fd_i)$ be the left- and right-hand sides of fd_i, respectively. The set of attributes involved in fd_i and Σ are referred to as $attr(fd_i)$ and $attr(\Sigma)$ respectively. When fd_i is projected on a tuple t, we refer to $t[X]$ and $t[Y]$ as *LHS* and *RHS* values of fd_i on t. An instance $I = \{t_1, t_2, ..., t_n\}$ of R satisfies Σ, denoted by $I \models \Sigma$, if I has no violations (i.e., every pair of tuples with the same *LHS* value must have the same *RHS* value) of any of the FDs in Σ. A cell $t[A]$ denotes the value of attribute A in tuple t. Every tuple has a unique identifier. The set of tuple identifiers in I is denoted $TID(I)$.

Definition 1. *Repair Instance [23] Given an instance I of schema R violating FDs Σ, an instance I' is a repair of I iff $I' \models \Sigma$ and $TID(I) = TID(I')$*

According to Definition 1, a repair is achievable only by modifying attribute values of tuples. Insertion or deletion of tuples or attributes are not allowed. Unlike [23], our space of repairs only contains constants from the active domain.

3.2 Encoding FD Patterns

We encode the FD patterns by projecting the FD graph on the instance. Refer to Figure 2b for illustration. Every FD pattern

$(X_1, X_2, ...X_n \rightarrow Y, [x_1, x_2, ..., x_n, y])$ $(n \leq |I|)$ is encoded with a directed hyperedge $(\{x_1, x_2, ..., x_n\}, \{y\})$. We refer to $x_1, x_2, ..., x_n$ as the *LHS* nodes and y as the *RHS* node.

FD Pattern Graph (*FDG*): The FDG of instance I is a directed hypergraph [5] $G(V, E)$, where: (1) Each node $v \in V$ has two attributes $v.attribute$ and $v.val$ encoding an attribute $a \in A$ and a data value $d \in dom(a)$, respectively; (2) A directed hyperedge $e(W, Z) \in E$ that (a) connects nodes in W (tail) to a node in Z (head) such that $|Z| = 1$ and $|W| \geq 1$; and (b) encodes an FD pattern $p(X_1, X_2, ..., X_n \rightarrow Y, [x_1, x_2, ..., x_i, y]) \in I$ $(n \leq |I|)$ such that there is a node $w \in W$ where $w.attribute = X_i$ and $w.val = x_i$ for all $i \leq |I|$, and $Z.attribute = Y$, $Z.val = y$.

For example, the graph in Figure 2b is the FD pattern graph for table D (Figure 2a). Each edge represents an FD pattern. In the sequel, since they represent the same thing, we employ the terms "FDG edge" and "FD pattern" interchangeably. Additionally, to ease the readability of the graph figures, we label the nodes with their values and omit the attribute names.

3.3 Pattern Quality

Our target is to select "good" FD patterns in the FD pattern graph to compute instance repairs. Therefore, it is crucial to characterize the quality of FD patterns in the FD pattern graph. This step is required by the repair algorithm (Section 5.2) to reason about the quality of various candidate FD patterns. We now present a general model to characterize the quality of FD patterns that also captures their interactions. By looking at an FD pattern $P : (X \rightarrow Y, [x, y])$ as an association rule [3] $(P[x] \rightarrow P[y])$, we can compute its *Support* (*Sup*) as the number of tuples with $X = x$ and $Y = y$ in I over the number of tuples in I.

$$Sup(P) = \frac{|P|}{|(X \rightarrow Y, *, *)|} \tag{1}$$

In the above equations, * denotes "any value".

As illustrated in Example 1, greedily selecting FD patterns based on their frequencies is not a good strategy for selecting the best FD patterns. That is, in Figure 2b, the edges ["YT43", "1407 Wescam Court"] and ["GF903", "140 W Court"] have the same weight. Therefore, it would be better if the score of an FD pattern not only includes its own support, but also the support of the FD patterns it can lead to. Thus, we extend *Sup* in Equations 1 to capture the quality of an FD pattern P by the set of FD patterns it can lead to (denoted $P^{\rightarrow}$) as follows:

$$Quality(P) = \frac{Sup(P) + \sum_{Q \in P^{\rightarrow}} Sup(Q)}{|P^{\rightarrow}| + 1} \tag{2}$$

The enumerator of $Quality(P)$ (Equation 2) is the sum of: (1) the *Support* of P, and (2) the *Support* of all the FD patterns that can be reached from P. We normalize the quality of a pattern using the average over the number of edges in $|P^{\rightarrow}|$.

Horizon performs a Depth-First Search (DFS) traversal over the FDG and computes the quality of each visited edge using Equation 2. To guarantee termination, back-edges (corresponding to cyclic FDs) are processed when the DFS traversal is complete. Specifically, *Horizon* performs the following steps: (1) Build a DFS tree from the input root vertex v; (2) For every edge $e = (v, w)$, if e is a back-edge add it to a set *BackEdges*. If not, compute the edge quality; (3) Assign the quality of the root vertex (The quality of a vertex v is the average quality of all the edges that can be reached from v). After the DFS step is completed, all back-edges are processed. The quality of a back-edge $e = (v, w)$ is the quality assigned to vertex w in the DFS step. Since it amounts to a DFS, the time and space complexities of computing and propagating scores in an FD pattern graph $FDG(V, E)$ are both $O(|V|+|E|)$.

4 RULES COMPILATION

4.1 Interactions among FD Patterns

Figure 3a enumerates four cases in which FD patterns interact with each other. FD patterns $P_i : (fd_i, V_i)$ and $P_j : (fd_j, V_j)$ $(fd_i, fd_j \in \Sigma$ and V_i, V_j are values assigned to $attr(fd_i)$ and $attr(fd_j)$ respectively with $i \neq j)$ interact with each other iff: (1) fd_i and fd_j share at least one attribute, and (2) the value of the shared attribute(s) is the same in V_i and V_j. Note that different cases of interactions have different semantics. Consider a dirty tuple t containing two FD patterns P_i and P_j corresponding to two different FDs fd_i and fd_j. Without loss of generality, we discuss interaction cases with FDs that have one attribute in their antecedent. P_i and P_j can exhibit the following four cases of interaction depending on the FDs they embed (Figure 3a):

Case 1 ($fd_i = A \rightarrow B$, $fd_j = A \rightarrow C$): $t[A] = a_1$ can be mapped to any RHS value in B and C, i.e., the choice of values of B is independent of the choice of the value of C. In other words, choosing the RHS of a_1 to satisfy $A \rightarrow B$ does not affect the choice of the RHS of a_1 to satisfy $A \rightarrow C$.

Case 2 ($fd_i = A \rightarrow C$, $fd_j = B \rightarrow C$): $t[A] = a_1$ and $t[B] = b_1$ must be mapped to the same RHS value C. In other words, Patterns P_i and P_j have to share the C value. Thus, the choice of the C value for A affects the choice of the C value for B, and vice-versa.

Case 3 ($fd_i = A \rightarrow B$, $fd_j = B \rightarrow C$): In this case, the consequent of P_i is the antecedent of P_j. In this case, the choice of the value of B affects the C value. That is, choosing a value $B = b_x$ in P_i would make b_x the antecedent of P_j.

Case 4 ($fd_i = A \rightarrow B$, $fd_j = B \rightarrow A$): This is the case of circular FDs; the choice of the value of A affects the choice of the value of B and vice-versa.

In the above cases, depending on the interaction case of the FDs, selecting an FD pattern for one FD in a tuple t may affect the choice of the FD patterns for the subsequent FDs that interact with it. Next, we formalize this observation.

4.2 Determining Bound and Free Attributes

FDs impose a "many-to-one" relationship between LHS and RHS values. That is, for the instance to be consistent, a LHS value is mapped to a single RHS value. An attribute A that does not appear as a RHS of an FD is said to be a *bound* attribute. Bound attributes have two properties: (1) They appear as part of a LHS in Σ and are thus used to *determine* the value of RHS attributes, and (2) Since they do not appear as RHS attributes in Σ, we cannot use other attributes to determine their values (because of the many-to-one relationship, we can only determine attribute values from LHS to RHS and not the other way around). If an attribute is not *bound*, then, it is a *free* attribute, i.e., its values are determined from other (LHS) attributes. Obviously, an attribute cannot be *bound* and $free$

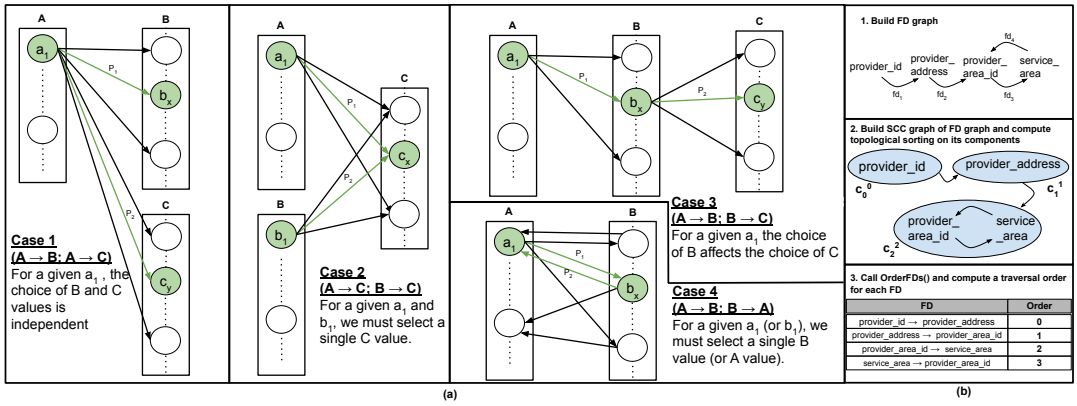


Figure 3: FD Patterns interaction cases

at the same time. Therefore, all *free* attributes must appear as RHS attributes in Σ (we discuss cyclic FDs next).

Proposition 1. *For every free attribute A in Σ, there must exist at least an attribute B such that: (1) There is an FD $fd_i(B \rightarrow A) \in \Sigma$; (2) If there is an FD $fd_j(A \rightarrow B) \in \Sigma$, then, there must exist at least an FD $fd_k \in \Sigma$ where $fd_k(C \rightarrow A)$ or $fd_k(C \rightarrow B)$. If $fd_k \notin \Sigma$, we designate either A or B to be a bound attribute.*

Proposition 1 states that every *RHS* attribute (free attribute) has to have at least one set of *LHS* attributes that determines it in Σ. This is trivial when there are no cyclic FDs. However, if Σ has cyclic FDs, some attributes could be *free* but would not have an *LHS* attribute that determines them outside the cycle. For instance, consider the FDs in Figure 4a where *provider_area_id* and *service_area* are *free* attributes because they both appear as *RHS* in Σ.

5 TRAVERSING THE FD PATTERN GRAPH

FDG node values from bound attributes are assigned from the input tuples. For example, consider table *D* and its FDs in Figure 4a (shaded cells correspond to fixed values and cell values "*" correspond to cells that we can change) and its corresponding FD pattern graph in Figure 4b (edge weights are quality scores as presented in Section 3.3). The set of *bound* attributes contains *provider_id* while all the other attributes involved in the FDs are free.

Proposition 2. *For an assignment β of bound attribute nodes A in the FDG(V, E), there exists a subgraph G(K, B) such that: (1) $K \subset V$ and $B \subset E$ and $A \subset K$; (2) $\forall fd(X \rightarrow Y) \in \Sigma : \exists e(U, W) \in B : U.attribute = X \wedge W.attribute = Y$.*

Proposition 2 states that assigning values to the *bound* attribute nodes in the FD pattern graph produces a subgraph (referred to as the chase graph) that covers all the FDs in Σ. In other words, the set of bound attribute values is all we need to determine the value of all the other attributes in Σ. Figure 4b illustrates the chase graph generated with *provider_id* as the bound attribute. For example, given Tuple t_1 in *D* (Figure 4a), the assignment of the bound attribute is $\beta = \{t1[provider\_id] = \text{"}GF903\text{"}\}$, all the other attributes in Σ can be modified. Then, we start the chase to get the FD patterns of all the FDs from the FD pattern graph (highlighted path in Figure 4b). Then, the resulting chase graph is translated to an FD pattern expression and used to repair tuple t_1 (Figure 4c).

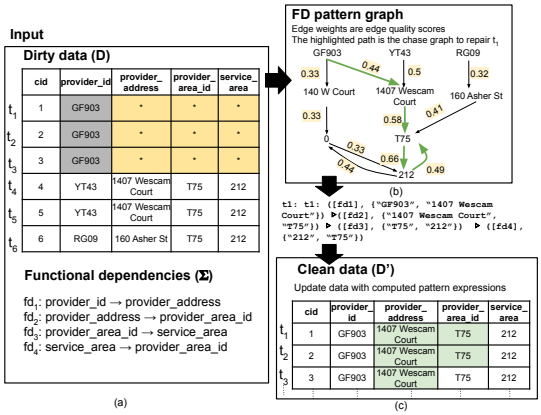


Figure 4: Example of repairing a tuple

5.1 Traversal order

Following the boundedness of attributes, we devise a traversal order of the FDG. Since the FDs can have cycles, we cannot directly apply standard topological ordering of the nodes in the FD graph. Instead, we first apply topological sorting on the Strongly Connected Component Graph (SCCG) induced by the FD graph (SCCG is guaranteed to be a DAG). We obtain the SCCG using the *Tarjan* algorithm [28] that runs in $O(|A|+|E|)$, where *A* and *E* are the vertices and edges in the FD graph, respectively.

Figure 3b illustrates how we go from the FD graph (top canvas) to the SCCG (middle canvas), and finally to ordering the FDs. Note how each SCC c_i has an order *o* assigned to it (denoted c_i^o)).

5.2 Pattern-Preserving Repairs

We present a linear-time repair algorithm that computes a repair instance in the form of pattern expressions. Notice that a pattern expression that covers all the FDs in Σ corresponds to a *chase graph* in the FDG. Our final goal is to choose chase graphs that have high edge weights without resorting to an exponential solution.

Algorithm 1 takes as input a dirty table *D* and the set of FDs Σ, and produces pattern expressions that correspond to clean tuples. Repair tables (*Rtable* in Algorithm 1) collect all the *LHS* to *RHS* mappings done so far and are used to update the input tuples accordingly. First, we build the FD pattern graph and compute its edge weights (Lines 1-2), and compute the order of FDs (Lines 3-5). We process the input data one tuple at a time (line 7), and

Algorithm 1: GeneratePatternPreservingRepairs(Σ, D)

output : For every tuple in D, return a pattern expression

```
1  FDG ← BuildFDPatternGraph(D, Σ)
2  FDG ← ComputePatternsQuality(IG)
3  SCCG ← BuildSCCGraph(Σ)
4  OC ← TopologicalSorting(SCCG)
5  Ordered_FDs ← OrderFDs(Σ, OC)
6  pattern_expressions ← ∅
7  forall Tuple t ∈ D do
8      for i ← 0 to |Ordered_FDs| do
9          forall FD f ∈ Ordered_FDs[i] do
10             Lval ← t[f.LHS]
11             if Rtable(f).contains(Lval) then
12                 FDPattern p ← New FDPattern(f, Lval →
                     Rtable(f).get(Lval))
13                 P^exp(t) ← P^exp(t) ⊳ p
14             else if f.RHS ∈ P^exp(t) then
15                 FDPattern p ← New FDPattern(f, Lval →
                     GetAttributeValue(P^exp(t), f.RHS))
16                 P^exp(t) ← P^exp(t) ⊳ p
17                 Rtable(f).Add(Lval, GetAttributeValue(P^exp(t), f.RHS))
18             else
19                 FDPattern p ← Edge_Selection(FDG)
20                 P^exp(t) ← P^exp(t) ⊳ p
21                 Rtable(f).Add(Lval, p.RHS)
22     pattern_expressions = pattern_expressions ∪ P^exp(t)
```

create a pattern expression $P^{exp}(t)$ for each Tuple t by building the chase graph from the FDG (lines 18-21). Then, the (LHS, RHS) mappings are written into the repair tables of each corresponding FD (lines 17 and 21). We handle the cases of a LHS that is (1) already mapped in a previous iteration (lines 11- 13); (2) already mapped to a *RHS* from another FD (lines 15- 17); or (3) not mapped yet, in which case we add a new pattern from the *FDG* (lines 19- 21).

6 EXPERIMENTAL STUDY

We present an experimental evaluation to answer the following questions: (1) How does *Horizon* perform under different error types and rates? (2) How does *Horizon* compare to state-of-the-art rule-based and non-rule-based cleaning techniques in terms of repair quality and runtime? (3) How scalable is *Horizon*?

6.1 Setup

Datasets. We use the following four datasets: (i) *DataX* is a private dataset from an active collaboration with *Company X* to clean their data. *DataX* integrates data from over 1,600 data sources and contains information about customers, their personal information and service providers that serve those customers. It contains 64M records, 43 attributes, over 2B and 750 million cells and 10 FDs. We have a sample of 470 correct cells that we use as the ground truth. (ii) *Parking* is a real-world dataset of parking ticket information for New York City [1] with 9M records and 9 FDs. We used a labeled random sample of 100 cells as the ground truth. Because the competing baselines we evaluate cannot handle larger datasets, we used a 100K and 20k records for *Parking* and *DataX* respectively in the effectiveness experiments and the whole 9M and 64M records (for *Parking* and *DataX* respectively) records for the runtime experiments. (iii) *Hospital* is a real-world dataset on health-care providers and hospitals [10]. It contains 100K records and 13 FDs. (iv) *Tax* is a synthetic dataset [17] with 6 FDs that contains records on tax information for individuals, e.g., *first name, last name*, and *whether the person has a child*. For measuring effectiveness, we use 100K records (*Tax*) while we generate 5M records for the scalability experiment (*Tax_Extended*).

Table 1: Data and FD properties of the datasets

Dataset	Tax	Hospital	Parking	DataX
Avg. Redundancy	8915.89	6274.03	540.46	4431.59
Atts w. AvgRed. ≤ 5	3	0	2	0
Avg. $\|val\|$	3.79	18.87	4.62	5.04
Attribute overlap	0.77	0.71	0.85	0.58

Datasets properties. Table 1 reports key dataset and FD properties which may affect the performance of *Horizon*. (1) The average redundancy is the average frequency of each attribute value. The number of attributes whose average redundancy is less or equal than 5 is reported in $Atts\ w.AvgRed \leq 5$. (2) Attribute overlap measures the overlap of attributes across the FDs.

Errors. We divide our experimental study into two parts: (1) Controlled errors (CE): We conduct a thorough experimental study to benchmark *Horizon* and its competing baselines under various error types and rates. Similar to existing data cleaning literature, e.g., [10, 18, 22] to cite a few, we use the state-of-the-art data cleaning benchmarking system BART [4] to control the injected error rate and type of errors. BART introduces synthetic errors to the *Tax* and *Hospital* datasets that would trigger violations of their corresponding FDs. More specifically, we generate errors for all their FDs with varying noise levels and using different data sizes. We introduce two types of errors to the *Hospital* and *Tax* datasets:

- **Error-1 (E1):** BART injects the input datasets with FD-detectable errors that include values from the active domain (e.g., replace "NY" with "CA") making it harder for the repair algorithms to find the correct repair if the candidate repair values are all well supported in the data.
- **Error-2 (E2):** In order to experiment with all kinds of errors, BART allows generating errors that may or may not be FD-detectable. These errors include outliers.

(2) Uncontrolled errors (UE): In this part, we do not inject errors and instead correct errors that are naturally occurring in the data. We evaluate *Parking* and *DataX* for these experiments.

Baselines. We compare the following repair algorithms to *Horizon*:

- *Holistic* [10], *SAMP* [6], *Unified* [9], *HoloClean* [25] and *Baran* [24] have been introduced in Section 2. For *Unified*, since we assume the FDs are correct in *Horizon*, we evaluated the data repairing part of *Unified* only.
- *Min* [8]. *Min* first assigns groups of cells that need to have the same value to different equivalence classes, then, a value is chosen for each equivalence class to repair the FD violations.

We picked different flavors of repair algorithms to show how *Horizon* compares to (1) a variety of FD-centric baselines (*SAMP*, *Min*, and *Unified*); (2) one non-FD-centric baseline (*Holistic*) and (3) general repair techniques (*HoloClean* and *Baran*).

Metrics. (1) Precision (P): The number of correctly repaired cells over the total number of repaired cells; (2) Recall (R): The number of correctly repaired cells over the total number of dirty cells; and (3) F1 score computed as $2(P*R)/(P+R)$. Since *SAMP* may generate different results in each run, we took the average of five runs to compute these metrics.

Table 2: Rule-based baselines effectiveness results (CE on the left, and UE on the right). E = Error type, P = Precision, R = Recall, F1 = F1 score

Algorithm	E	Tax			Hospital		
		P	R	F1	P	R	F1
Horizon	E1	**0.81**	0.74	**0.76**	**0.93**	**0.77**	**0.84**
	E2	0.8	**0.27**	**0.38**	0.88	0.61	0.71
Holistic	E1	0.16	**0.87**	0.25	0.04	0.28	0.06
	E2	0.22	0.19	0.19	0.48	0.04	0.03
SAMP	E1	0.08	0.34	0.09	0.20	0.29	0.20
	E2	0.12	0.08	0	0.24	0.43	0.29
Unified	E1	0.12	0.01	0.01	0.82	0.66	0.73
	E2	**1.0**	0	0	0.61	0.7	0.65
Min	E1	0.32	0.7	0.43	0.42	0.6	0.49
	E2	0.29	0.26	0.27	**0.99**	**0.76**	**0.85**

Algorithm	Parking			DataX		
	P	R	F1	P	R	F1
Horizon	**0.98**	**0.56**	**0.7**	0.93	**0.93**	**0.93**
Holistic	0.43	0.12	0.18	1.0	0.08	0.14
SAMP	0.14	0.02	0	0.2	0.32	0.24
Unified	0.42	0.1	0.16	0.71	0.06	0.11
Min	0.1	0.04	0.05	**0.95**	0.66	0.77

Table 3: Non-rule-based baselines results (CE on the left, and UE on the right). E = Error type, P = Precision, R = Recall, F1 = F1 score.

Algorithm	E	Tax				Hospital			
		P	R	F1	Time	P	R	F1	Time
Horizon	E1	0.95	0.75	0.83	0.5 sec	0.93	**0.79**	**0.85**	**0.8 sec**
	E2	0.85	0.05	0.09	0.63 sec	0.98	**0.73**	**0.83**	**0.57 sec**
HoloClean	E1	0.85	0.01	0.01	13 min	**1.0**	0.09	0.16	13 min
	E2	**0.91**	0.01	0.01	8 min	**1.0**	0.61	0.75	14 min
Baran	E1	**0.97**	**0.94**	**0.96**	2 min	1.0	0.14	0.24	2 min
	E2	0.72	**0.67**	**0.7**	20 min	0.54	0.44	0.49	5.2 min

Algorithm	Parking				DataX			
	P	R	F1	Time	P	R	F1	Time
Horizon	**0.89**	**0.61**	**0.72**	**1 sec**	**0.93**	**0.93**	**0.93**	**1 sec**
HoloClean	0.7	0.1	0.17	18 min	0.91	0.72	0.8	33 min
Baran	0.14	0.02	0.0	11 sec	**1.0**	0.2	0.33	44 sec

Table 4: Interaction cases (IC) vs. precision (P), recall (R), F1 and runtime (T) in sec. We use a bit array to indicate which IC is present in the input FDs (e.g., 0101 indicates FDs involved in IC2 and IC4).

	Tax										
ICs	0011	0101	0110	0111	1001	1010	1011	1100	1101	1110	1111
P	0.81	0.29	0.95	0.95	0.81	0.80	0.81	0.81	0.81	0.81	0.81
R	0.72	0.01	0.66	0.66	0.71	0.71	0.71	0.72	0.72	0.71	0.71
F1	0.75	0.01	0.77	0.77	0.75	0.74	0.75	0.75	0.75	0.75	0.75
T	3.68	2.34	3.42	5.32	5.09	6.00	8.04	5.13	6.61	7.70	9.71
	Hospital										
ICs	0011	0101	0110	0111	1001	1010	1011	1100	1101	1110	1111
P	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93
R	0.81	0.75	0.75	0.78	0.78	0.81	0.80	0.75	0.76	0.78	0.78
F1	0.86	0.82	0.82	0.84	0.84	0.86	0.85	0.82	0.82	0.84	0.84
T	6.01	7.82	4.90	9.09	8.87	5.99	9.52	7.81	12.01	8.75	13.03

Implementation and Hardware Platform. *Horizon* is implemented in Java. Evaluation was done on a Linux machine with 8 Intel Xeon E5450 3.0GHz cores and 32GB main memory.

6.2 Effectiveness Results

In Table 2, we report effectiveness results for CE (left) and UE (right). In this section, we focus on rule-based baselines while we present non-rule-based baselines results in Section 6.5.

Table 2 shows that *Horizon* outperforms (F1 score) all other baselines on all datasets with one exception where it is slightly outperformed by *Min* on *Hospital* with E2 errors. It also outperforms these baselines on Precision and Recall values with very few exceptions. As expected, E1 errors are more amenable to repairs than E2 errors. This is because E2 errors are random and are not necessarily FD-detectable, this is why the F1 scores for E2 errors is generally low. *Horizon* performs well on E1 errors as they usually belong to FD patterns that are not frequent in the data, making alternative "frequent" patterns more likely to be correct. We also note that the repair quality of *Horizon* is more or less consistent across all datasets and characteristics (Table 1).

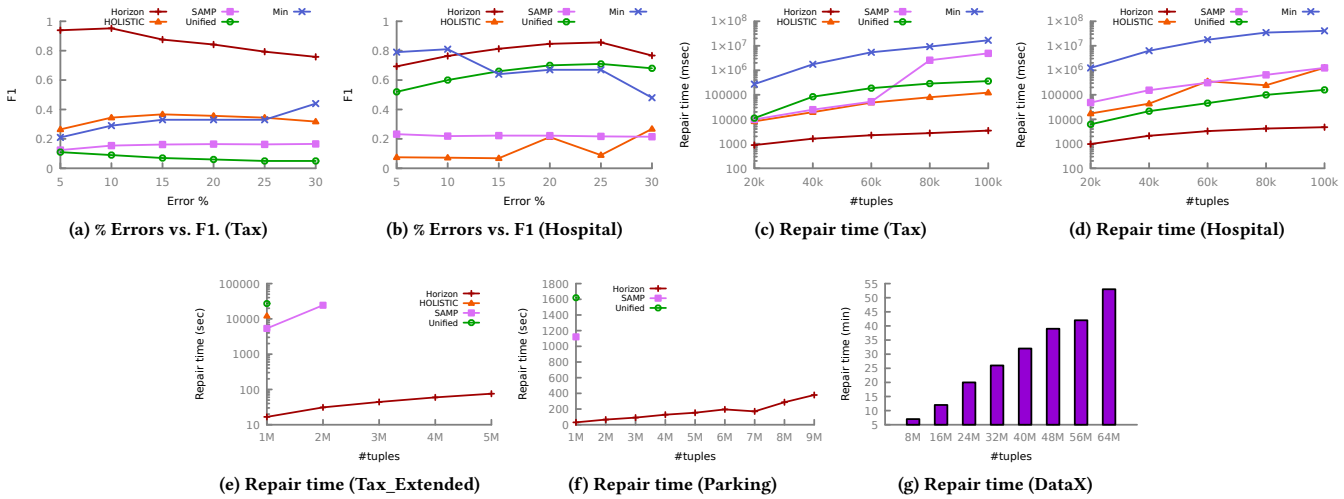
As for the competing baselines, For *Hospital* E2, *Min* has the highest F1; due to the high redundancy in the data, *Min* was able to find the repairs even with the added noise. However, it is worse than *Horizon* on all other datasets in addition to a high runtime. *Holistic* can reach a high precision (*DataX* and *Parking*), but misses a lot of repairs (hence the low recall). *Holistic*, *SAMP*, *Min* perform poorly because (1) they only focus on minimal changes to repair the data, which in most cases does not cover all the space of repairs; and (2) when they are undecided on a repair, they introduce special variables (outside the domain) to fix a rule violation. *Unified* performs poorly on *Tax* because: (1) The order of FDs in *Unified* was causing an incorrect fix to be performed for earlier FDs which limits the repair choices in the subsequent FDs. (2) We noticed that *Unified* starts with FDs that have columns with lower redundancies. From Table 1, we can see that *Tax* has the highest number of attributes with an average redundancy that is less than 5. These attributes appear in the FDs of *Tax* making it hard to select the correct fix for conflicting values in those columns, and making the wrong choices for FDs with those columns creates bad repairs in other FDs.

Repair quality vs. error rate. Figures 5a-b report the F1 score of repairs w.r.t. different data error rates. *Horizon* outperforms the other systems, especially as the error rate goes up. This is because adding more errors makes it harder for minimality-based algorithms to identify correct cell values, whereas *Horizon* selects values that lead to frequent FD patterns, and hence, a better repair quality.

Takeaways. ① Using FD patterns to repair an instance leads to high-quality FD repairs. ② Minimality does not produce high-quality repairs. ③ Because *Horizon* captures patterns across several attributes, the lack of redundancy on individual attributes does not significantly affect its performance.

6.3 Repair quality vs. interaction case

In this section, we examine the effect of the different pattern Interaction Cases (ICs) in the input FDs on repair quality and runtime. As discussed in Section 4, *Horizon* employs four pattern interaction cases. Table 4 reports all possible combinations among the four ICs. We have a total of 11 configurations (we exclude the case with 0 FDs and only one IC). We use a bit array notation to indicate which IC is present in the input FDs. For example 0101 indicates



(a) % Errors vs. F1. (Tax) (b) % Errors vs. F1 (Hospital) (c) Repair time (Tax) (d) Repair time (Hospital)

(e) Repair time (Tax_Extended) (f) Repair time (Parking) (g) Repair time (DataX)

Figure 5: Effectiveness and Runtime results

FDs participating in IC2 and IC4. As expected, the more interaction cases we have in the input FDs, the higher the repair time. We notice that IC4 (the cyclic FDs case) introduces the most overhead. With few ICs, the quality of repairs suffers (e.g., Tax 0101); *Horizon* is unable to benefit from the interactions among the FDs or the low data redundancy in *Tax*. The ICs that improve repair quality are IC3 and IC1. This is expected as IC3 forms longer chains among the FD patterns enabling the propagation of quality scores and IC1 allows choosing the *RHS* with the highest support.

6.4 Runtime Results

We report the runtime results in Figures 5c-g for all the datasets. *Horizon* significantly outperforms all the competing baselines by at least 3 orders of magnitude. All the competing baselines strive to generate minimal repairs, which boils down to solving an optimization problem to detect and then resolve the violating cells. All the algorithms were given a 24-hour deadline to finish running for each data size increment. In many cases, some of the competing baselines could not terminate (missing points in Figure 5e-g). Overall, it took *Horizon* about 75 seconds to clean the 5 million records in *Tax_Extended*. For *DataX*, *Holistic* and *SAMP* did not finish running even with a 1M-record partition. With *Parking*, *SAMP* and *Unified* could not finish for over 1M records while *Holistic* could not even handle 1M records. It took a total of around 300 seconds to clean the 9 million records in *Parking* (Figure 5f). In Figure 5g we report the runtime of *Horizon* using 8M increments of *DataX*. It took 53 minutes to clean the 64M records while none of the competing baselines were able to terminate even the smallest increment (8M) within 24 hours. The runtime of *Unified* is affected by the value length and average redundancy. For example, *Unified* takes the longest on *Tax* which happens to have the highest number of attributes with low redundancy (Table 1) which in turn increases the set of unique values, and hence repair time. Furthermore, even with the high redundancy of *Hospital*, *Unified*'s runtime is close to the one in *Tax* because *Hospital* has a high average value length. Overall, *Unified*'s runtime is unpredictable; it took 27 mins to repair *Parking* (1M) while it did not finish within 24 hours for DataX (1M). This is because *DataX* has a (1) slightly higher avg. $|val|$ and (2) a high set of candidate repairs, which leads to computing string similarity across a larger number of value pairs. In general, the runtime of *Unified* is relatively high compared to *Horizon*.

Takeaways. ① Thanks to its FDG traversal strategy, *Horizon* can scale to millions of records linearly. ② Data redundancy and value length directly affect the performance of rule-based baselines.

6.5 Comparison to non-rule-based baselines

Table 3 summarizes effectiveness and runtime results when *Horizon* is compared to *HoloClean* and *Baran*. In order for *HoloClean* and *Baran* to terminate within 24 hours, we had to evaluate them on smaller sizes of the datasets: Hospital (20k), Tax(10k), Parking(20k) and DataX(20k). We note the following: (1) *HoloClean* and *Baran* require a specification of the cells that have errors. If we consider all the cells involved in FD violations as potentially erroneous cells, *HoloClean* and *Baran* perform poorly on E1 errors while *Baran* performs better with E2 errors in *Tax*. (2) *HoloClean* performs well on *DataX*, which has a low redundancy, suggesting that *HoloClean* can generalize well even with a low redundancy. (3) Runtime of *HoloClean* is unpredictable. It took 13 min on 20k records of *Hospital*, while it took 33 min on *DataX*.

Takeaways. ① *Horizon*'s focus on FD errors allows it to generate higher-quality FD repairs than cleaning systems that target generality and may miss several errors that may appear as FD violations. ② The complexity of general cleaning systems makes them inefficient for iterative cleaning scenarios.

7 CONCLUSIONS

In this paper, we presented a novel technique that is a radical departure from existing repair approaches both in accuracy and scalability. Guided by the FDs, *Horizon* generates a set of modifications that exploit inherent FD-induced patterns found in the data to produce a repair instance. Moreover, we leverage the FD interactions to produce the repair instance in linear time.

REFERENCES

[1] New York City Open Data. https://opendata.cityofnewyork.us.

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

[3] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining Association Rules Between Sets of Items in Large Databases *(SIGMOD '93)*.

[4] Patricia C. Arocena, Boris Glavic, Giansalvatore Mecca, Renée J. Miller, Paolo Papotti, and Donatello Santoro. 2015. Messing Up with BART: Error Generation for Evaluating Data-cleaning Algorithms. *Proc. VLDB Endow.* 9, 2 (Oct. 2015), 36–47. https://doi.org/10.14778/2850578.2850579

[5] Giorgio Ausiello and Luigi Laura. 2017. Directed hypergraphs: Introduction and fundamental algorithms—A survey. *Theoretical Computer Science* 658 (2017), 293–306. https://doi.org/10.1016/j.tcs.2016.03.016 Horn formulas, directed hypergraphs, lattices and closure systems: related formalism and application.

[6] George Beskales, Ihab F. Ilyas, and Lukasz Golab. 2010. Sampling the Repairs of Functional Dependency Violations under Hard Constraints. *Proc. VLDB Endow.* 3, 1 (2010), 197–207. https://doi.org/10.14778/1920841.1920870

[7] George Beskales, Ihab F. Ilyas, Lukasz Golab, and Artur Galiullin. 2013. On the relative trust between inconsistent data and inaccurate constraints. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 541–552. https://doi.org/10.1109/ICDE.2013.6544854

[8] Philip Bohannon, Wenfei Fan, Michael Flaster, and Rajeev Rastogi. 2005. A Cost-based Model and Effective Heuristic for Repairing Constraints by Value Modification. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. 143–154. https://doi.org/10.1145/1066157.1066175

[9] Fei Chiang and Renée J. Miller. 2011. A unified model for data and constraint repair. *2011 IEEE 27th International Conference on Data Engineering* (2011), 446–457.

[10] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. 458–469. https://doi.org/10.1109/ICDE.2013.6544847

[11] Xu Chu, Mourad Ouzzani, John Morcos, Ihab F. Ilyas, Paolo Papotti, Nan Tang, and Yin Ye. 2015. KATARA: Reliable Data Cleaning with Knowledge Bases and Crowdsourcing. *Proc. VLDB Endow.* 8, 12 (2015), 1952–1955. https://doi.org/10.14778/2824032.2824109

[12] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. 2007. Improving Data Quality: Consistency and Accuracy. In *VLDB'07*. 315–326.

[13] Graham Cormode, Lukasz Golab, Flip Korn, Andrew McGregor, Divesh Srivastava, and Xi Zhang. 2009. Estimating the confidence of conditional functional dependencies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 469–482. https://doi.org/10.1145/1559845.1559895

[14] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed K. 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 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 541–552. https://doi.org/10.1145/2463676.2465327

[15] Sushovan De, Yuheng Hu, Venkata Vamsikrishna Meduri, Yi Chen, and Subbarao Kambhampati. 2016. BayesWipe: A Scalable Probabilistic Framework for Improving Data Quality. *ACM J. Data Inf. Qual.* 8, 1 (2016), 5:1–5:30. https://doi.org/10.1145/2992787

[16] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p44-deng-cidr17.pdf

[17] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional Functional Dependencies for Capturing Data Inconsistencies. *ACM Trans. Database Syst.* 33, 2, Article 6 (June 2008), 48 pages.

[18] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2013. The LLUNATIC Data-cleaning Framework. *Proc. VLDB Endow.* 6, 9 (July 2013), 625–636. https://doi.org/10.14778/2536360.2536363

[19] Lukasz Golab, Howard Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. 2008. On Generating Near-optimal Tableaux for Conditional Functional Dependencies. *Proc. VLDB Endow.* (Aug. 2008). https://doi.org/10.14778/1453856.1453900

[20] Shuang Hao, Nan Tang, Guoliang Li, Jian He, Na Ta, and Jianhua Feng. 2017. A Novel Cost-Based Model for Data Repairing. *IEEE Trans. on Knowl. and Data Eng.* 29, 4 (April 2017), 727–742. https://doi.org/10.1109/TKDE.2016.2637928

[21] Ihab F. Ilyas and Xu Chu. 2015. Trends in Cleaning Relational Data: Consistency and Deduplication. *Foundations and Trends in Databases* 5, 4 (2015), 281–393. https://doi.org/10.1561/1900000045

[22] Matteo Interlandi and Nan Tang. 2015. Proof positive and negative in data cleaning. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 18–29.

[23] Solmaz Kolahi and Laks V. S. Lakshmanan. 2009. On Approximating Optimum Repairs for Functional Dependency Violations *(ICDT '09)*.

[24] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: Effective Error Correction via a Unified Context Representation and Transfer Learning. *Proc. VLDB Endow.* 13, 11 (2020), 1948–1961. http://www.vldb.org/pvldb/vol13/p1948-mahdavi.pdf

[25] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1190–1201. https://doi.org/10.14778/3137628.3137631

[26] El Kindi Rezig, Lei Cao, Giovanni Simonini, Maxime Schoemans, Samuel Madden, Nan Tang, Mourad Ouzzani, and Michael Stonebraker. 2020. Dagger: A Data (not code) Debugger. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p35-rezig-cidr20.pdf

[27] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. 2018. Effective and Complete Discovery of Bidirectional Order Dependencies via Set-based Axioms. *The VLDB Journal* 27, 4 (Aug. 2018), 573–591. https://doi.org/10.1007/s00778-018-0510-0

[28] Robert Tarjan. 1972. Depth first search and linear graph algorithms. *SIAM JOURNAL ON COMPUTING* 1, 2 (1972).

[29] Manasi Vartak and Samuel Madden. 2018. MODELDB: Opportunities and Challenges in Managing Machine Learning Models. *IEEE Data Eng. Bull.* 41, 4 (2018), 16–25. http://sites.computer.org/debull/A18dec/p16.pdf

[30] Jiannan Wang, Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, Tim Kraska, and Tova Milo. 2014. A sample-and-clean framework for fast and accurate query processing on dirty data. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 469–480. https://doi.org/10.1145/2588555.2610505

[31] Jiannan Wang and Nan Tang. 2017. Dependable Data Repairing with Fixing Rules. *ACM J. Data Inf. Qual.* 8, 3-4 (2017), 16:1–16:34. https://doi.org/10.1145/3041761

[32] Mohamed Yakout, Laure Berti-Équille, and Ahmed K. Elmagarmid. 2013. Don'T Be SCAREd: Use SCalable Automatic REpairing with Maximal Likelihood and Bounded Changes *(SIGMOD '13)*.