Analyzing Deviations from Monotonic Trends through Database Repair

SHUNIT AGMON, Technion, Israel
JONATHAN GAL, Technion, Israel
AMIR GILAD, Hebrew University, Israel
ESTER LIVSHITS, Technion, Israel
OR MUTAY, Technion, Israel
BRIT YOUNGMANN, Technion, Israel
BENNY KIMELFELD, Technion, Israel

Datasets often exhibit violations of expected monotonic trends—for example, higher education level correlating with higher average salary, newer homes being more expensive, or diabetes prevalence increasing with age. We address the problem of quantifying how far a dataset deviates from such trends. To this end, we introduce Aggregate Order Dependencies (AODs), an aggregation-centric extension of the previously studied order dependencies. An AOD specifies that the aggregated value of a target attribute (e.g., mean salary) should monotonically increase or decrease with the grouping attribute (e.g., education level).

We formulate the AOD repair problem as finding the smallest set of tuples to delete from a table so that the given AOD is satisfied. We analyze the computational complexity of this problem and propose a general algorithmic template for solving it. We instantiate the template for common aggregation functions, introduce optimization techniques that substantially improve the runtime of the template instances, and develop efficient heuristic alternatives. Our experimental study, carried out on both real-world and synthetic datasets, demonstrates the practical efficiency of the algorithms and provides insight into the performance of the heuristics. We also present case studies that uncover and explain unexpected AOD violations using our framework.

CCS Concepts: • **Information systems → Data management systems**.

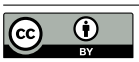
Additional Key Words and Phrases: Database repair; aggregate order constraints; trend analysis

ACM Reference Format:
Shunit Agmon, Jonathan Gal, Amir Gilad, Ester Livshits, Or Mutay, Brit Youngmann, and Benny Kimelfeld. 2026. Analyzing Deviations from Monotonic Trends through Database Repair. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 10 (February 2026), 27 pages. https://doi.org/10.1145/3786624

1 Introduction

Real-world datasets frequently violate anticipated monotonic relationships, such as the expectation that the median salary increases with the education level, that newer homes are generally more expensive, and that the prevalence of diabetes rises with age. The detection and analysis of monotonic trends in data is a fundamental problem in statistics [14, 57, 59], with wide-ranging applications in

Authors' Contact Information: Shunit Agmon, Technion, Haifa, Israel, shunit.agmon@gmail.com; Jonathan Gal, Technion, Haifa, Israel, jonathan.gal@campus.technion.ac.il; Amir Gilad, Hebrew University, Jerusalem, Israel, amirg@cs.huji.ac.il; Ester Livshits, Technion, Haifa, Israel, esterlivshits@gmail.com; Or Mutay, Technion, Haifa, Israel, or.mutay@campus.technion.ac.il; Brit Youngmann, Technion, Haifa, Israel, brity@technion.ac.il; Benny Kimelfeld, Technion, Haifa, Israel, bennyk@cs.technion.ac.il.



ACM 2836-6573/2026/2-ART10
https://doi.org/10.1145/3786624

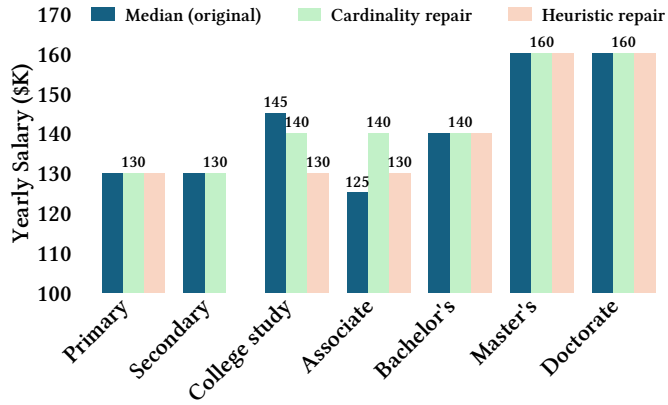


Fig. 1. Median salary versus education level in the USA, computed with the Stack Overflow dataset. "College study" refers to people who have studied in college or university without a degree. "Doctoral" includes professional doctoral degrees (e.g., MD, JD) and other doctoral degrees (e.g., PhD).

economics, medicine, and the social sciences. For example, monotonicity constraints incorporated into predictive models have been shown to increase predictive accuracy for housing prices, disease prediction, student success, and more [31, 53, 70]. A central task in this domain is to determine whether an underlying relationship—often modeled through noise-aware regression—is monotonic, either increasing or decreasing, with respect to an explanatory variable. The inherent challenge is distinguishing monotonic relationships from random fluctuations, especially in high-noise or sparsely sampled data; notable contributions include the work of Ghosal et al. [26] and of Hall and Heckman [32], who developed reliable tests for monotonicity. In this work, we propose a complementary database-centric view on the problem of analyzing monotonic trends and explaining their violation, namely *measuring the amount of intervention* required to exhibit monotonicity.

Example 1.1. The Stack Overflow dataset[1] consists of responses from developers to questions about their jobs, including demographics, education level, role, and salary. We focus on entries from the USA (8,682 tuples). One might expect that, as the education level grows, so does the median yearly salary. This, however, is not the case, as Figure 1 shows (left bars). For example, the median salary of people with some college or university studies but without a degree ($145K) is higher than that of people with an Associate's degree ($125K) or Bachelor's degree ($140K). Additionally, the median salary of people with primary and secondary school education ($130K each) is higher than that of people with an Associate degree ($125K). □

To analyze the violation of monotonic trends, we propose the *aggregate order dependency* (AOD), stating a monotonic trend exhibited by an aggregate query over an ordered grouping attribute. More precisely, an AOD has the form $G \nearrow \alpha(A)$ where G is a grouping attribute (column name) with a linearly ordered domain, A is an aggregated attribute, and α is an aggregate function. The AOD $G \nearrow \alpha(A)$ states that whenever one grouping variable is greater than another, its aggregated value cannot be lower. For example, "education$\nearrow$median(salary)" states that the median salary grows with the level of education. Put differently, the AOD $G \nearrow \alpha(A)$ states that the SQL query

$$\text{SELECT } G, \alpha(A) \quad \text{FROM } R \quad \text{GROUP BY } G$$

[1] https://survey.stackoverflow.co/2022 (accessed April 2025)

should return a table that defines a monotonically non-decreasing function. One can analogously define the AOD $G \searrow \alpha(A)$ that requires a non-increasing trend.

The computational challenge we study in this paper is that of finding a *cardinality-based repair* (C-repair for short), that is, the largest subset of the table that satisfies the AOD. While C-repairs have been traditionally studied as a *prescriptive* tool in data cleaning [21, 40, 45], our work is better motivated by the *descriptive* aspect of measuring how far the table is from satisfying the AOD and, consequently, the monotonic trend that it suggests. From another angle, our work can be seen as applying the "minimal repair" measure of inconsistency [8, 46, 50]; here, consistency is with respect to the AOD. Beyond their quantitative semantics, C-repairs can also serve as an explanatory tool of a qualitative nature: analyzing the removed tuples can shed light on the nature of the trend violation, as we demonstrate through several use cases in Section 7.

Example 1.2. The dataset discussed in Example 1.1 violates the AOD $\text{education} \nearrow \text{median}(\text{salary})$. A C-repair (Figure 1, central bars), removing the minimal number of tuples, requires deleting 77 tuples, constituting 0.88% of the data (17 people with college study and no degree and 60 with Associate degrees). In contrast, repairing for a *decreasing* trend $\text{education} \searrow \text{median}(\text{salary})$ requires the removal of a minimum of 566 tuples, which constitute 6.52% of the dataset. This suggests that the dataset is much closer to exhibiting an increasing trend than a decreasing one. We later discuss the *heuristic* repair that is referred to in the right bars of the figure. □

Computing a C-repair is challenging, both theoretically and pragmatically, since there can be exponentially many ways of restoring the AOD through tuple deletions (as is the case for functional dependencies [40, 45]). Hence, we begin with a complexity analysis of the problem, showing that while the general case is NP-hard, several practically relevant aggregates admit polynomial-time solutions. We present a general algorithmic template that uses dynamic programming to reduce the problem to ones that involve a single group, which are arguably simpler. We show the instantiation of the template on a list of common aggregate functions α, including max, min, count, count distinct, median, sum, and average. Hence, we establish a polynomial-time bound for each of these aggregates. However, in the case of sum and average, where the number of possible aggregation values can be exponential in the number of tuples, the resulting algorithm is only weakly polynomial, that is, it runs in polynomial time if we assume that the input represents the numbers in a unary encoding; we show that this weakening is necessary, as the problem is NP-hard otherwise.

Although the algorithmic template offers a polynomial-time guarantee, in practice, this time can be very long. For example, over an hour for 10K tuples with median, or for 1K tuples with avg. The algorithm is generic and applies to various aggregation functions under mild assumptions. While useful, this also means we are not using the specific properties of the aggregation functions, which could accelerate the computation. We address this limitation through two complementary approaches. First, we develop optimization techniques tailored to the specific aggregation functions, significantly reducing execution costs. Second, we introduce a heuristic method that computes a repair satisfying the AOD by repeatedly removing the tuple with the most impact on the violation, albeit potentially with more deletions than the C-repair. This heuristic serves two key purposes: (1) providing a practical alternative when computational resources are limited, and (2) yielding an upper bound on the number of deletions required. We demonstrate how this bound can be leveraged to substantially prune the search space of the dynamic program of the exact computation.

Example 1.3. As we demonstrate in Section 7, the heuristic algorithm performs well for several aggregate functions. However, in certain scenarios, it may yield substantially inferior repairs. For the use case described in Example 1.1, the heuristic algorithm computed a repair for the non-decreasing trend $\text{education} \nearrow \text{median}(\text{salary})$ in 0.52 seconds, removing 494 tuples—5.7% of the data. While

this is 30× faster than the exact algorithm (15.9 seconds), it removed 6.4 times more tuples than necessary. Notably, it eliminated the entire secondary-school group.

As for the opposite AOD, namely education$\searrow$median(salary), the heuristic algorithm removed 8,187 tuples (94.3% of the data) in 16.7 seconds. This number provides an upper bound on the number of tuples needed to be removed, but it does not give any information about the *minimum* number of deletions needed to fulfill the AOD; hence, it cannot support a claim such as the one closing Example 1.2. □

We then describe an extensive empirical study on our implementation of the algorithms. The datasets include the German credit [33], Stack Overflow,[1] H&M [61], Zillow,[2] and Diabetes,[3] as well as synthetic data that we generated. The empirical study demonstrates the effectiveness of the algorithms and optimizations and shows the quality of the heuristic solution, which is often highly competitive, even though it takes a small fraction of the execution time of finding a C-repair.

In summary, our contributions are as follows: (1) We introduce the notion of an AOD to express a monotonic trend in the result of an aggregate query over a table; (2) We provide a complexity analysis of C-repairs for AODs, and present a general algorithmic framework that reduces the global repair problem to local problems on individual groups; (3) We develop multiple optimization techniques to improve the practical performance of the algorithms; (4) We propose a heuristic algorithm that efficiently computes approximate repairs and supports the exact algorithm by providing an upper bound for pruning the search space; (5) We conduct an experimental study on real and synthetic datasets.

The remainder of the paper is organized as follows. After an overview of related work in Section 2, we present in Section 3 the formal framework and define AODs. Section 4 provides a complexity analysis of the C-repair problem. In Section 5, we describe our implementation and algorithmic optimizations, followed by heuristic repair strategies in Section 6. Section 7 reports on our empirical evaluation, and we conclude in Section 8.

2 Related Work

Database constraints. A large body of work in the data management community has studied enforcing constraints on relational data via data repair, traditionally focusing on integrity constraints [7, 21]. This includes functional dependencies [11, 15, 28, 40, 45, 49], conditional functional dependencies [10, 25], multi-valued dependencies [64], denial constraints [16, 18, 60], and inclusion dependencies [11, 17, 39, 47]. The most common repair operations include tuple deletion [15–17, 27, 45, 47, 49], tuple insertion [64], and value updates [11, 18, 25, 28, 39, 40, 60]. The AOD entails a fundamentally different repair problem, since it involves aggregation. Repairs of aggregation-centric constraints have been studied by Flesca et al. [22, 23, 24], but they have not considered monotonicity—a central aspect of our framework.

Closer to our work is the line of research on dependencies that involve ordering, specifically the *order dependencies* (ODs) [19, 20, 29, 30, 36, 41, 58], also known as *ordered functional dependencies* [51, 52]. An OD of the form $X \rightarrow_{\succeq} Y$ asserts that X functionally determines Y, and that the relationship is monotonic with respect to a specified order $\succeq$ over the domain. Of similar flavor are the *trend dependencies* (TD) [65], capturing changes over time (e.g., "salaries do not decrease in time"). These formalisms strengthen functional dependencies by capturing monotonic dependencies between attributes. In contrast, our AODs consider an aggregation level of the data and do not require or entail any underlying functional dependency. In addition, past work in this category focused on

[2] https://www.zillow.com/research/data/
[3] https://www.kaggle.com/datasets/iammustafatz/diabetes-prediction-dataset

logical analysis (satisfiability and implication) and constraint mining, differently from the focus of this work—quantifying violation via repairs.

Statistical perspectives on monotonicity. Monotonic relationships between variables, such as the expectation that an outcome variable consistently increases or decreases with an explanatory factor, have been extensively studied in statistics and econometrics. Much of that work focuses on detecting violations of monotonicity using nonparametric or regression-based methods, often under linearity assumptions [26, 32, 54]. These approaches are common in applications such as environmental trend analysis [34], economics [42, 54], and financial forecasting [54]. Other research takes a more theoretical view, using property testing to assess monotonicity over general domains, including partial orders [9]. While this field focuses on identifying monotonic models with noise, our work offers a complementary perspective: we quantify the deviation from the trend by the amount of required database intervention that creates the expected behavior. Moreover, our framework accommodates general aggregation functions and is not bound to any specific one.

Cherry-picking. Another relevant line of research focuses on assessing the robustness of claims based on the results of aggregate queries [3, 4, 37, 43, 67, 68]. Wu et al. [67, 68] defined parameterized queries and analyzed how perturbations in input parameters affect the resulting claims, using measures of relevance and naturalness. Closer to our work, Asudeh et al. [4, 5] investigate the robustness of trendlines to selective endpoints or item selection, aiming to expose cherry-picking. While these approaches highlight the vulnerability of trend-based claims to manipulation, our work focuses on quantifying and repairing violations of expected trends in the data. The challenge of identifying misleading trends is, again, different from the focus of this work on the repair challenge.

3 Formal Framework

We now present the formal framework of this work, starting with preliminary concepts on databases.

Database concepts. A relation schema $\mathcal{S}$ is a sequence $(A_1, \ldots, A_k)$ of *attributes*. We denote by $Att(S)$ the set $\{A_1, \ldots, A_k\}$. We assume, without loss of generality, that all attributes are numeric. In particular, a relation r over $\mathcal{S} = (A_1, \ldots, A_k)$ is a finite set of tuples in $\mathbb{R}^k$. If $t \in r$ is a tuple and $G \in Att(\mathcal{S})$, then we denote by $t[A]$ the value of t for the attribute A. We denote by $r[A]$ the projection of r to A, that is, the set $\{t[A] \mid t \in r\}$. We also write $r[\![A]\!]$ to denote the bag-semantics projection of r to A, that is, the bag of values that occur under A without removing duplicates.

In this work, an *aggregate function* α maps a bag V of numbers to a single value $\alpha(V)$. We focus on the common aggregate functions avg, sum, count, countd, min, max, and median.

Let G and A be two attributes of $\mathcal{S}$, let r be a relation, let α be an aggregate function, and let $g \in \mathsf{dom}(G)$ be a constant. We denote by $\alpha_r(A \mid G = g)$ the value $\alpha((\sigma_{G=g}r)[\![A]\!])$, where $\sigma_{G=g}r$ is the relation that consists of all tuples $t \in r$ with $t[G] = g$.

Aggregate Order Dependencies. Let $\mathcal{S}$ be a relation schema. An *aggregate order dependency* (AOD) is an expression of the form $G \nearrow \alpha(A)$ where G and A are attributes in $Att(\mathcal{S})$. We refer to G as the *grouping* attribute and to A as the *aggregate* attribute. A relation r *satisfies* the AOD $G \nearrow \alpha(A)$, denoted $r \models G \nearrow \alpha(A)$, if the following holds: for all values g and g' in $r[G]$, if $g \leq g'$ then $\alpha_r(A \mid G = g) \leq \alpha_r(A \mid G = g')$. (We focus on *increasing* trends, but all our results are applicable to *decreasing* trends as well.)

Example 3.1. Consider the relation r on the left of Figure 2. The tables on the right give the aggregate values $\alpha_r(\text{income} \mid \text{edu} = a)$ for $\alpha = \text{sum}$ (top) and $\alpha = \text{avg}$ (bottom). We can see that r satisfies the AOD edu $\nearrow$ sum(income), since the sum column is monotonically increasing. On the

person	edu	income
Ashley	1	1
Brandon	1	2
Chloe	2	2
Daniel	2	5
Emily	2	6
Faith	2	5
Gavin	2	2
Hanna	3	8
Isaac	3	4
Jerry	3	3
Katie	3	2
Larry	3	2
Marie	3	1
Nathan	3	1

edu	sum(income)
1	3
2	20
3	21

edu	avg(income)
1	1.5
2	4
3	3

Fig. 2. Example relation r (left) and statistics grouped by the education level (edu).

other hand, r violates the AOD edu $\nearrow$ avg(income) since the average salary for education level 2 is higher than the average salary for education level 3. □

Cardinality-based repair. The main problem we study is that of finding a *cardinality-based repair* (C-repair for short) for an AOD. The formal definition is as follows.

Definition 3.2. A *monotonic subset* of r is a subset r' of r such that $r' \models G \nearrow \alpha(A)$. A *C-repair* of r is a monotonic subset of maximum size. Hence, r' is obtained from r by deleting the minimum possible number of tuples needed to satisfy $G \nearrow \alpha(A)$.

We illustrate the definition with an example.

Example 3.3. Continuing Example 3.1, let us consider the AOD edu $\nearrow$ avg(income). We can repair r by removing the tuples of Larry, Nathan, and Marie (highlighted in green), which would result in $\alpha_{r'}(\text{income} \mid \text{edu} = 3) = 4.25$ and, therefore, satisfy the AOD. Nevertheless, we can also repair r by removing the tuples of Daniel and Emily (highlighted in pink), which would result in $\alpha_{r'}(\text{income} \mid \text{edu} = 2) = 3$. In this example, removing any single tuple will not satisfy the AOD, so the repair requires removing at least two tuples. Hence, we conclude that removing the tuples of Daniel and Emily results in a C-repair of size 12. □

4 The Complexity of C-repairs

In this section, we investigate the computational complexity of finding a C-repair for an AOD. The main result is the following.

Theorem 4.1. *Let $G \nearrow \alpha(A)$ be an AOD.*

(1) For $\alpha \in \{\max, \min, \text{count}, \text{countd}, \text{median}\}$, a C-repair can be found in polynomial time.
(2) For $\alpha \in \{\text{sum}, \text{avg}\}$, a C-repair can be found in pseudo-polynomial time.

Recall that *pseudo-polynomial time* means that the running time is polynomial if we assume that the numbers (here, in the column A) are integers given in a unary representation. In Section 4.3, we will show that this weakening is necessary, since the problem is NP-hard if we assume a binary representation.

To prove the theorem, we present algorithms for computing C-repairs for different aggregate functions α. We begin with a general procedure that applies to all aggregate functions (and requires

Algorithm 1: Dynamic programming algorithm for C-repair: CardRepair.

Input : Relation r, AOD $G \nearrow \alpha(A)$
Output: Maximum-size subset $r' \subseteq r$ such that $r' \models G \nearrow \alpha(A)$.

```
for x ∈ V_α(r) do
    H_0[x] ← ∅
for i ∈ {1, ..., m} do
    for x ∈ V_α(r_i) do
        if O_α(r_i, x) = ∅ then
            H_i[x] ← H_{i−1}[x]
        else
            H_i[x] ← O_α(r_i, x) ∪ H_{i−1} [arg max_{y≤x} |H_{i−1}[y]|] .
return H_m[max(V_α(r))]
```

some sub-procedures to be implemented) and then proceed to specific ones. Throughout this section, we assume the AOD is $G \nearrow \alpha(A)$ and refer to the input relation as r.

4.1 General Procedure

We first show a dynamic programming algorithm for finding a C-repair (described in Algorithm 1). The algorithm requires specific procedures that depend on the aggregation function α. In the next sections, we discuss the realization of these procedures for specific aggregation functions. The procedures are as follows.

(1) The first procedure takes a relation r as input, and produces a set $V_\alpha(r)$ that contains all possible values $\alpha_{r'}(A)$ that any nonempty subset $r' \subseteq r$ can take. For example, if α is max or min, then $V_\alpha(r)$ can be $r[A]$.
(2) The second procedure takes as input a relation r with a single group (hence, it ignores the grouping attribute G) and a value $x \in V_\alpha(r)$, and it computes an optimal (maximum-size) relation $r' \subseteq r$ such that $\alpha(r'[\![A]\!]) = x$, that is, the aggregate value is exactly x. We denote r' by $O_\alpha(r, x)$. If no such r' exists, then we define $O_\alpha(r, x) = \emptyset$. (Later on, we will refer to this problem as *aggregation packing*.)

Example 4.2. Consider again the relation r of Figure 2, and let α be avg. Then $V_\alpha(r)$ should include (at least) every mean of every possible bag of values from the income column. If r consists of the tuples of Daniel, Emily, and Faith, and α is avg, then $O_\alpha(r, 5)$ consists of the tuples of Daniel and Faith. □

We describe how to compute $V_\alpha(r)$ and $O_\alpha(r, x)$ efficiently for different aggregation functions in Section 4.2. Before we do so, we show how $V_\alpha(r)$ and $O_\alpha(r, x)$ can be used for computing a C-repair.

For convenience, we use the following definitions. Let $r[G] = \{g_1, \ldots, g_m\}$ where $g_i < g_j$ whenever $i < j$. We denote by r_i the relation $\{t \in r \mid t[G] = g_i\}$, which we refer to as a *group*. For $i \in \{1, \ldots, m\}$, we denote by $r_{\leq i}$ the relation $r_1 \cup \cdots \cup r_i$. As an example, referring to the relation r of Example 3.1, the relation r_2 consists of the tuples of Chloe, Daniel, Emily, Faith, and Gavin, and the relation $r_{\leq 2}$ includes, in addition to the five, the tuples of Ashley and Brandon.

Algorithm 1 applies dynamic programming by increasing $i = 1, \ldots, m$. For each i and $x \in V_\alpha(r_i)$, the algorithm computes $H_i[x]$, which is a largest subset $r' \subseteq r_{\leq i}$ such that r' satisfies the AOD

and, in addition, the value of the rightmost bar is x; that is, if g is the maximal value in $r'[G]$, then $\alpha_{r'}(A \mid G = g) = x$. For each i and $x \in V_\alpha(r_i)$, the algorithm considers (in lines 4–8) two cases.

(1) In the first case, $O_\alpha(r_i, x)$ is empty; that is, no subset of r_i has the α value x. In this case, $H_i[x]$ is simply $H_{i-1}[x]$ (ignoring the group r_i).
(2) Otherwise, if $O_\alpha(r_i, x)$ is nonempty, then $H_i[x]$ is the union of $O_\alpha(r_i, x)$ and the best $H_{i-1}[y]$ over all $y \le x$.

4.2 Instantiations

Next, we present instantiations of Algorithm 1 to specific aggregate functions α by showing how to compute $V_\alpha(r)$ and $O_\alpha(r, x)$ in polynomial time.

4.2.1 Polynomial-Time Instantiations. We begin with polynomial-time instantiations that give rise to the first part of Theorem 4.1.

Max and min. When α is max, the set $V_\alpha(r)$ of possible values over r is simply the set of values in the column A. For $x \in V_\alpha(r)$, the set $O_\alpha(r, x)$ is simply the set $\{t \in r \mid t[A] \le x\}$ of all tuples where the A attribute is at most x. The case of min is similar, except that $O_\alpha(r, x)$ is the set $\{t \in r \mid t[A] \ge x\}$.

Count and count-distinct. For count, the set $V_\alpha(r)$ is $\{0, \ldots, |r|\}$, and $O_\alpha(r, x)$ can be any subset of size x. For countd, the set $V_\alpha(r)$ is $\{0, \ldots, |r[A]|\}$; for $x \in V_\alpha(r)$, the set $O_\alpha(r, x)$ contains the tuples with the x most frequent A values in r. More precisely, for each $a \in r[A]$, denote $n_a = |\{t \in r \mid t[A] = a\}$. Let $a_1, \ldots, a_m$ be the values of $r[A]$ sorted by decreasing n_{a_i}. Then

$$O_{\text{countd}}(r, x) = \{t \in r \mid t[A] \in \{a_1, \ldots, a_x\}\}.$$

Median. The set $V_\alpha(r)$ of possible median values over r is

$$\{t[A] \mid t \in r\} \cup \{(t_1[A] + t_2[A])/2 \mid t_1, t_2 \in r\}.$$

For $x \in V_\alpha(r)$, we denote by $n_x^>$, $n_x^=$, and $n_x^<$ the number of tuples $t \in r$ with $t[A] > x$, $t[A] = x$, and $t[A] < x$, respectively. Let $t_1, \ldots, t_n$ be the tuples of r sorted by increasing $t_i[A]$. Then, the set $O_\alpha(r, x)$ is the largest of the two candidate sets r_{include} and r_{exclude}. The set r_{include} is the largest subset of r where x is the median by being the single middle element. Hence, the number of values higher than and lower than x should be equal (and the size of the set is odd):

$$r_{\text{include}} = \begin{cases} \emptyset & \text{if } n_x^= = 0, \\ r & \text{if } n_x^= \neq 0 \text{ and } n_x^> = n_x^<, \\ \{t_1, \ldots, t_{2n_x^< + n_x^=}\} & \text{if } n_x^= \neq 0 \text{ and } n_x^> > n_x^<, \\ \{t_{n_x^< - n_x^> + 1}, \ldots, t_n\} & \text{otherwise.} \end{cases}$$

The set r_{exclude} is the largest subset where x is the median because it is the average of two middle elements, which is only possible for an even-sized set. Let $S = \{(i, j) \mid t_i[A] + t_j[A] = 2x, i < j\}$ and let $(i^*, j^*) = \arg\max_{(i,j) \in S} \{\min\{i - 1, n - j\}\}$. Then,

$$r_{\text{exclude}} = \begin{cases} \emptyset & \text{if } S = \emptyset; \\ \{t_1, \ldots, t_{i^*-1}\} \cup \{t_{i^*}, t_{j^*}\} \cup \{t_{j^*+1}, \ldots, t_{j^*+i^*-1}\} & \text{if } i^* - 1 \le n - j^*; \\ \{t_{i^*-n+j^*}, \ldots, t_{i^*-1}\} \cup \{t_{i^*}, t_{j^*}\} \cup \{t_{j^*+1}, \ldots, t_n\} & \text{otherwise.} \end{cases}$$

Note that the goal here is to construct the largest possible balanced subset around a pair (t_i, t_j). The number of tuples that we can place on either side of this pair is at most $\min\{i-1, n-j\}$. The total size of the subset is $2 + 2 \cdot \min\{i-1, n-j\}$; hence, to maximize this size, we must choose the pair maximizing the value $\min(i-1, n-j)\}$.

4.2.2 Weakly polynomial-time instantiations. For sum and average, we assume that values in $r[A]$ are integers given in a unary representation (hence, establishing pseudo-polynomial time as stated in Theorem 4.1). Throughout this section, we denote $r = \{t_1, \ldots, t_n\}$.

Sum. We begin with $\alpha = \mathsf{sum}$. For $V_\alpha(r)$ we can use $V_{\mathsf{sum}}(r) = \{\mathsf{sum}(r^-[\![A]\!]), \ldots, \mathsf{sum}(r^+[\![A]\!])\}$ where r^- (resp., r^+) is the subset of r consisting of the tuples with a negative (resp., positive) value in the B attribute.

For $x \in V_\alpha(r)$, the computation of $O_\alpha(r, x)$ can be done via a standard dynamic program for the knapsack problem, as follows. The program computes the value $M[j, s]$, for $j = 0, \ldots, n$ and $s \in V_\alpha(r)$, that holds the maximal size of any subset of $\{t_1, \ldots, t_j\}$ with the sum s of A values. As an example, considering the relation r of Figure 2, the value $M[5, 8] = 3$, as witnessed by the subset that corresponds to Ashley, Brandon, and Daniel.

With the $M[j, s]$ computed, $O_\alpha(r, x)$ is simply $M[n, x]$. Beginning with $j = 0$, we set

$$M[j, s] = 0 \text{ if } s = 0, \text{ and } M[j, s] = -\infty \text{ otherwise}.$$

For $j > 0$ and any $s \in V_\alpha(r)$, we check whether $s - t_j[A] \in V_\alpha(r)$. If so, we set

$$M[j, s] = \max\left(M[j-1, s], 1 + M[j-1, s - t_j[A]]\right) .$$

If $s - t_j[A] \notin V_\alpha(r)$, then we set $M[j, s] = M[j-1, s]$.

Average. Consider now $\alpha = \mathsf{avg}$. For the set $V_\alpha(r)$ we can choose

$$V_{\mathsf{avg}}(r) = \{s/\ell \mid s \in V_{\mathsf{sum}}(r) \wedge 1 \leq \ell \leq r\} .$$

For $O_\alpha(r, x)$, we construct the data structure M' that has the three arguments j, ℓ, and s for $j = 0, \ldots, n$, $\ell = 0, \ldots, j$, and $s \in V_{\mathsf{sum}}(r)$. The value $M'[j, \ell, s]$ is Boolean, with $M'[j, \ell, s]$ being true if and only if $\{t_1, \ldots, t_j\}$ has a subset with precisely ℓ tuples, where the A values sum up to s. As an example, in Figure 2, both $M'[5, 8, 2]$ is true (due to Brandon and Emily) and $M'[5, 8, 3]$ is true (due to Ashley, Brandon, and Daniel). To set $O_\alpha(r, x)$, we take the maximal value ℓ such that $M'[j, \ell, s]$ is true and $s = \ell \cdot x$. (If no such s and ℓ exist, then we set $O_\alpha(r, x) = \emptyset$ as per our convention.) We build M' as follows.

- For $j = 0$, we set $M'[j, \ell, s]$ to true if $\ell = 0$ and $s = 0$, and false otherwise.
- For $\ell = 0$ and every j, we set $M'[j, \ell, s]$ to true if $s = 0$, and false otherwise.
- For $j > 0$ and $\ell > 0$, we check whether $s - t_j[A] \in V_\alpha(r)$; if so:

$$M'[j, \ell, s] = M'[j-1, \ell, s] \vee M'[j-1, \ell-1, s - t_j[A]] ,$$

  otherwise, if $s - t_j[A] \notin V_\alpha(r)$, then $M'[j, \ell, s] = M'[j-1, \ell, s]$.

4.3 Hardness of Sum and Average

The following theorem implies that the pseudo-polynomial weakening of Theorem 4.1 is required, since finding a C-repair is intractable for $\alpha = \mathsf{sum}$ and $\alpha = \mathsf{avg}$ under the conventional binary representation of numbers. The proof is available in the full version [2].

Theorem 4.3. *Let $G \nearrow \alpha(A)$ be an AOD where α is either* sum *or* avg. *It is NP-hard to find a C-repair (under the conventional binary representation of numbers).*

5 Implementation and Optimizations

Next, we discuss the implementation of CardRepair (Algorithm 1), along with its instantiations. We also discuss practical optimizations of the implementation.

The implementation uses a single data structure (dictionary) H instead of the m different H_is of Algorithm 1. To do so, in line 4 we traverse the xs in decreasing order, thereby assuring that we do not run over values from the previous iteration (i.e., those belonging to H_{i-1}) that are needed for computing $H[x]$ at iteration i; this holds because the computation of $H_i[x]$ requires only values $H_i[y]$ such that $y \leq x$. By doing so, we can restrict line 4 to iterate over only *feasible* values x of r_i (i.e., the ones satisfying $O(r_i, x) \neq \emptyset$), which we compute in advance. The rest of the values are set in a previous iteration (hence, due to line 6, need not be changed). We use lazy initialization of H, so we avoid lines 1–2 altogether.

5.1 Pruning based on Upper Bounds

CardRepair computes a solution (subset) for every feasible aggregation value and subset size. We can prune the search space to ignore subsets that are too small. For example, consider an AOD over a database with 1K tuples, and assume we know of a monotonic subset satisfying the AOD by removing 100 tuples. CardRepair will compute a solution for every subset size, but solutions removing more than 100 tuples will never be used, as they cannot be completed into a C-repair, and therefore, can be avoided altogether.

In this work, we propose two methods of achieving a bound on the number of tuples to remove. The first is to compute an *approximate* repair using a fast algorithm that makes the best effort to build a large (but possibly not the largest) monotonic subset. Such a heuristic algorithm is described in Section 6.

We utilize this pruning for the computation of $O(r_i, x)$ in the cases of sum and avg, which are the most computationally intensive. For that, we replace the computation of $M[j, s]$ (or, $M'[j, \ell, s]$ for avg) with a new (yet conceptually equivalent) data structure $P[j, s]$ ($P'[j, \ell, s]$ for avg), which holds the minimal number of tuples needed to be deleted from $\{t_1, \ldots, t_j\}$ in order to satisfy the AOD, starting with $j = 0$ (i.e., no tuples are removed from r) and $s = \alpha_r(A)$, where $P[j, s] = 0$. This allows us to ignore the entries $P[j, s]$ and avoid their storage in the case where $P[j, s] > h$, where h is the number of tuples removed in a given approximate repair. We elaborate on this optimization in the full version [2].

The second method exploits a dominance relation over partial solutions saved in H: a solution is said to dominate another if it retains more tuples and induces weaker constraints on the remaining groups. Dominated solutions can be pruned from the search space without affecting optimality (see [2]). In general, while the pruning strategies for H we describe there substantially reduce H, their current overhead diminishes a runtime improvement.

In the next section, we show how to compute $O(r_i, x)$ in a holistic way for all relevant x, and there we also show how the bound h can be incorporated for avg, sum, and median.

5.2 Holistic Packing

5.2.1 Naive WholePack$\langle\alpha\rangle$. Algorithm 1 requires a packing procedure to compute the set $O_\alpha(r_i, x)$ for each value $x \in V_\alpha(r_i)$. This computation is costly, especially in the cases of sum, avg and median. For sum and avg, recall that we populate data structures containing the optimal subset size for each aggregation value (Section 4.2.2). In Algorithm 1, these data structures need to be populated from scratch for every $x \in V_\alpha(r_i)$. We can optimize the algorithm by populating these data structures once, before the iteration in line 3 and without repeating their shared computation. For median,

instead of searching for a tuple (or pair) that yields a specific median x for each $x \in V_\alpha(r_i)$, we can iterate once over all tuples and tuple pairs and store the resulting medians.

Computing the $O_\alpha(r_i, x)$ in a holistic manner for all $x \in V_\alpha(r_i)$ enables further optimizations for specific aggregation functions. Next, we devise such optimizations for each of the three aggregates α; we refer to the optimized procedures uniformly as WholePack$\langle\alpha\rangle$. As we show in Section 7, the holistic computation gives considerable acceleration of the computation of a repair.

5.2.2 Optimizing WholePack$\langle$median$\rangle$. We next describe an algorithm to compute $O_{\text{median}}(r_i, x)$ holistically for all feasible medians x. This allows us to avoid the search for the specific tuples that yield each median. Furthermore, we can also iterate over unique values in r_i instead of a costly iteration over tuples. Due to its length, the pseudocode is omitted and can be found in the full version [2].

As described in Section 4.2.1, a median x is either the A value of a single middle tuple (if the number of tuples if odd) or the average of the A values of two middle tuples (if it is even). In what follows, we refer to the middle tuple(s) as pivot(s).

The algorithm uses a histogram to keep track of the number of tuples on each side of the pivot(s). It maintains two data structures: a mapping $M[x]$, representing the size of the largest subset of r_i with median x, and a mapping $D[x]$ for reconstructing this subset, containing the (one or two) pivots and the number of tuples on each side to include in the subset. It addresses three cases:

(1) A single middle tuple: the algorithm iterates over the tuples t_i sorted by $t_i[A]$ within a pruned range $i \in [n/2 - h/2, n/2 + h/2]$ determined by the repair size bound h. For each tuple, it calculates the size of the largest balanced subset that can be formed around it as the single median and updates M and D if a new maximum is found. (2) Two adjacent middle tuples: A similar computation is performed for medians formed by pairs of consecutive tuples. (3) Two nonadjacent middle tuples: the algorithm iterates over pairs of unique values from the histogram that stores, for each A value, its number of appearances in r_i. For each pair (a_1, a_2), the histogram is used to compute k_{left} (the number of tuples with A value at most a_1) and k_{right} (the number of tuples with A value at least a_2). The maximum size of a balanced subset around this pair is $2 \cdot \min\{k_{\text{left}}, k_{\text{right}}\}$. The structures M and D are updated if a new maximum is found. This method avoids the costly process of checking all pairs of nonadjacent tuples.

5.2.3 Optimizing WholePack$\langle$sum$\rangle$. We next describe an algorithm to compute $O_{\text{sum}}(r_i, x)$ holistically for all feasible sums x, to avoid the repeated population of $M[j, s]$ (described in Section 4.2.2). We additionally apply a strategy of iterating over unique values instead of individual tuples (as in Section 5.2.2). We next describe an additional optimization, based on existing optimized solutions for the knapsack problem [48, 56]. For the latter optimization, we assume that all A values are positive.

Let $v_1, \ldots, v_n$ denote the unique A values in ascending order and stored in the histogram. Algorithm 2 maintains two data structures: *(1)* an array M where an index s represent a possible sum, and $M[s]$ is the maximum size of a subset with sum s, and *(2)* a matrix D, where $D[j, s]$ represents the number of instances of the unique value v_j, used in the maximum subset for sum s. D can also be used to reconstruct the subset for a given feasible sum s.

The algorithm builds these data structures by iteratively processing each unique value v_j from the histogram in ascending order. The algorithm considers two distinct cases for each sum s:

(1) No instances of v_j left (lines 12-17): This is the case if the solution for the sum $s - v_j$ has already used every available copy of v_j, i.e., $D[j, s - v_j] = \text{Hist}[v_j]$. The algorithm must therefore reevaluate how to form the sum s by checking different combinations involving v_j and the previous unique values.

Algorithm 2: WholePack⟨sum⟩

Input : A bag of values $r_i[\![A]\!]$.
Output: Array M mapping sums to maximum subset sizes, and an array D mapping unique value ids j and sums s to the number of usages of v_j in the construction of s.

```
1  Hist ← a histogram of r_i[[A]], in ascending order by values v_j
2  T ← sum(r_i[[A]]) M ← Array of size T + 1 containing −1 in each cell and 0 in M[0]
3  D ← Empty array of size |Hist| × (T + 1)
4  Z ← 0 // Maximal sum reached so far.
5  for v_j ∈ Hist do
6      Z ← Z + v_j · Hist[v_j]
7      M' ← M // Avoid changing M directly.
8      for s ← v_j to Z do
9          u ← D[j, s − v_j]
10         s_prev ← s − (u + 1) · v_j
11         // All instances of v_j are used.
12         if u = Hist[v_j] then
13             for u' ← 1 to Hist[v_j] do
14                 size ← M[s − u' · v_j] + u'
15                 if M[s − u' · v_j] ≥ 0 and size > M'[s] then
16                     M'[s] ← size
17                     D[j, s] ← u'
18         // Additional instances of v_j are available.
19         if u < Hist[v_j] and M[s_prev] ≥ 0 and M[s_prev] + u + 1 > M[s] then
20             M'[s] ← M[s_prev] + u + 1
21             D[j, s] ← u + 1
22     M ← M'
23 return (M, D)
```

(2) There are available instances of v_j left (lines 19-21): This is the case if the solution for the sum $s - v_j$ did not require all copies of v_j. Meaning, $D[j, s - v_j] < \mathsf{Hist}[v_j]$. Here, we rely on an insight (stated in Lemma 5.1), that the maximum subset for a sum s can be found by comparing just two scenarios: the best solution for s without using v_j, and the best solution formed by adding v_j to the best subset for sum $s - v_j$.

Lemma 5.1. *If* $D[j, s - v_j] \neq \mathsf{Hist}[V_j]$, *then either* $D[j, s] = 0$ *or* $D[j, s] = D[j, s - v_j] + 1$.

The proof of the lemma as well as a detailed running example can be found in the full version [2]. As the algorithm builds the set of feasible sums from smallest to largest, it cannot be combined with the dictionary pruning optimization from Section 5.1.

5.2.4 Optimizing WholePack⟨avg⟩. As in the previous section, a holistic computation of $O_{\text{avg}}(r_i, x)$ for all feasible averages x enables skipping repeated computations. In the following, we describe several additional optimizations.

First, we modify the data structure to a value-based semantic (instead of tuple-based), which greatly reduces the size of the data structure. Recall that for avg, we used a dynamic programming table $M'[j, \ell, s]$ that stores true if there is a subset of $\{t_1, \ldots, t_j\}$ of size ℓ and sum s of A values. We modify the algorithm to maintain a table $P[s, \ell]$ of sums and sizes.

To enable the pruning optimization (described in Section 5.1), we use the semantics that $P[s, \ell]$ stores true if there is a subset $r' \subseteq r$ of size ℓ such that the sum of A values of $r \setminus r'$ is s.

Algorithm 3: WholePack⟨avg⟩

Input: A bag of values $r_i[\![A]\!]$, and an upper bound h on the number of tuples to remove.
Output: An array F mapping average values to maximum subset sizes.

```
1  Hist ← a histogram of r_i[[A]], in ascending order by values v_j
2  T ← sum(r_i[[A]])
3  P ← an empty mapping with default false
4  P[T][0] ← true // Removing 0 tuples yields the sum T.
5  D ← an empty mapping
6  for v_j ∈ Hist do
7      P' ← an empty mapping with default false
8      D' ← D
9      for s ∈ P do
10         for ℓ ∈ P[s] do
11             for k ← 0 to Hist[v_j] do
12                 // Add v_j to each previous (removed) subset.
13                 s' ← s − k · v_j
14                 ℓ' ← ℓ + k
15                 if ℓ' > h then
16                     break // Prune using bound h.
17                 P'[s'][ℓ'] ← true
18                 if k > 0 and (s' ∉ D' or ℓ' ∉ D'[s']) then
19                     D'[s'][ℓ'] ← (s, ℓ, v_j, k)
20     P ← P'
21     for s ∈ D' do
22         for ℓ ∈ D'[s] do
23             D[s][ℓ] ← D'[s][ℓ]
24 // F Maps avg values to maximal subset sizes.
25 F ← an empty mapping
26 for s ∈ P do
27     for ℓ ∈ P[s] do
28         c ← |r_i| − ℓ
29         x ← s/c
30         if x ∉ F or F[x] < c then
31             F[x] ← c
32 return F
```

Furthermore, we apply a similar approach to WholePack⟨median⟩ and WholePack⟨sum⟩, and avoid iterating on individual tuples. Instead, we iterate on unique $r_i[A]$ values using a histogram, to find possible sums and sizes (thereby, possible averages). We store backtracking information in a separate data structure $D[s, \ell]$, to reconstruct the removed subset of tuples.

In Algorithm 3, we iterate over $r_i[A]$ values v_j stored in the histogram (line 6). For each previously obtained sum s and removed subset size ℓ, and each possible number k of instances of v_j, we try to remove k instances of v_j from s (lines 13-14), to obtain a new sum s' with a new removed subset size $\ell' = \ell + k$. If a bound h is given on the number of removed tuples and $\ell' > h$, we prune this option (lines 15-16). If this is indeed a new combination of sum and subset size, we update the data structures (lines 17-19). We then update the original data structures with the sums and sizes achieved using v_j (lines 20-23). Note that unlike the algorithms for WholePack⟨median⟩ and WholePack⟨sum⟩, the data structure used in the algorithm (P) does not map from aggregation

Algorithm 4: Heuristic algorithm for repair: HeurRepair.

Input : Relation r, AOD $G \nearrow \alpha(A)$.
Output: Subset $r' \subseteq r$ such that $r' \models G \nearrow \alpha(A)$.

1 **Function** *ComputeMVI(r, A, G, α)*
2 **return** $[\max\left(0, \alpha(r_i[\![A]\!]) - \alpha(r_{i+1}[\![A]\!])\right)$ for $i \in \{1, \ldots, |r[G]| - 1\}]$
3 $\mathsf{MVI} \leftarrow \mathtt{ComputeMVI}(r, A, G, \alpha)$
4 **while** $\mathrm{sum}(\mathsf{MVI}) > 0$ **do**
5 **for** $t \in r$ **do**
6 $r' \leftarrow r \setminus \{t\}$
7 $\mathsf{MVI}' \leftarrow \mathtt{ComputeMVI}(r', A, G, \alpha)$
8 $\mathsf{impacts}[t] \leftarrow \mathrm{sum}(\mathsf{MVI}) - \mathrm{sum}(\mathsf{MVI}')$
9 $t^* \leftarrow \arg\max_t \{\mathsf{impacts}[t]\}$
10 $r' \leftarrow r \setminus \{t^*\}$
11 $\mathsf{MVI} \leftarrow \mathtt{ComputeMVI}(r', A, G, \alpha)$
12 **return** r'

values (averages) to maximal subset sizes. Therefore a new mapping is built (lines 25-31), based on the sums and sizes of removed subsets.

6 Heuristic Algorithm

CardRepair (Algorithm 1) guarantees a C-repair, but its runtime can be prohibitively long for some aggregation functions. This is the case especially for sum and avg that take pseudo-polynomial time and depend on the size of $V_\alpha(r)$ (the set of possible values that $\alpha_{r'}(A)$ can take for any $r' \subseteq r$). In this section, we present a polynomial-time greedy algorithm that serves two purposes. First, it provides a fast, inexact alternative to CardRepair. Second, it accelerates CardRepair via pruning, by providing an upper bound on the number of tuples to remove in both the generic CardRepair algorithm (as described in the full version [2]) and the implementation of WholePack$\langle\alpha\rangle$ (described in Section 5.1), without sacrificing the optimality guarantee. Using this bound, we can avoid the computations in CardRepair searching for subsets that are too small.

CardRepair iterates over each feasible aggregation value, which may correspond to every possible subset of tuples. HeurRepair avoids this heavy iteration. Instead, it greedily selects the tuple(s) with the largest impact on the violations of the AOD. This process is repeated until the violations are eliminated, and a monotonic subset is reached. We next define the notions of violation and impact.

Denote the *measure of violation* of the groups r_i and r_{i+1} by:

$$\mathsf{MVI}(r_i, r_{i+1}) = \max(0, \alpha(r_i[\![A]\!]) - \alpha(r_{i+1}[\![A]\!])) \,. \tag{1}$$

The *sum of violations* over all groups r_i is defined as $S_{\mathsf{MVI}}(r) = \sum_{i=1}^{|r[G]|-1} \mathsf{MVI}(r_i, r_{i+1})$. The *impact* of a tuple t on the sum of violations is defined as $S_{\mathsf{MVI}}(r) - S_{\mathsf{MVI}}(r \setminus \{t\})$; this is called the "leave-one-out" influence and is inspired by causal counterfactuals [55]. That is, a tuple t has a positive impact if removing it results in a decrease in the sum of violations.

HeurRepair (Algorithm 4) iteratively removes the tuple with the highest impact, until all violations are resolved ($S_{\mathsf{MVI}} = 0$). The algorithm begins by computing the initial MVI for every pair of consecutive groups (line 3), based on Equation (1). As long as the sum of violations remains positive (line 4), the algorithm calculates the impact of each tuple (lines 5-8). The tuple with the maximum impact is selected for removal (line 9). Once this tuple is removed, the dataset is updated accordingly (line 10), and the measures of violations are recomputed (line 11). Finally, the algorithm returns the modified relation r', which satisfies the AOD (line 12).

Remark 1. The algorithm may choose to remove a tuple with a non-positive impact. In such cases, it still selects the tuple with the highest impact, which would be the least negative one. Restricting the algorithm to remove only tuples with a positive impact might prevent it from guaranteeing a valid solution, as it could get stuck in a situation where no tuples with a positive impact exist, yet violations remain. This can happen if removing a tuple from r_i reduces the violation between r_i, r_{i+1} but causes another violation to increase. We next show an example of this phenomenon. □

Example 6.1. Consider the relation $r(G, A)$ composed of the following tuples: $(1, 3)$, $(1, 4)$, $(2, 2)$, $(2, 3)$, $(2, 4)$, $(3, 1)$, $(3, 2)$, and the AOD $G \nearrow \max(A)$.

The relation consists of three groups r_1, r_2, and r_3, where $r_i = \{t \in r \mid t[G] = i\}$. Any tuple $t \in r_i$ with $t[A] < \max(r_i[\![A]\!])$ has impact 0, as removing it does not change the aggregation value $\max(r_i[\![A]\!])$. As for the tuples with the maximum A value in each group: in r_1, the tuple $(1, 4)$ has impact 0 since $\max(r_1[\![A]\!]) = \max(r_2[\![A]\!])$ (hence, $\mathsf{MVI}(r_1, r_2) = 0$), and removing this tuple only decreases the value $\max(r_1[\![A]\!])$ (hence, $\mathsf{MVI}(r_1, r_2)$ remains 0). The tuple $(2, 4)$ in r_2 has impact 0 since removing it will decrease $\mathsf{MVI}(r_2, r_3)$ by 1 but increase $\mathsf{MVI}(r_1, r_2)$ by 1.

Finally, the tuple $(3, 2)$ in r_3 has impact -1, as removing it increases $\mathsf{MVI}(r_2, r_3)$ by 1. If the algorithm was restricted to removing only tuples with a positive impact, it would stop at this point and fail to find a solution. In contrast, the algorithm will remove one of the tuples with a maximum impact (0). Following the tie-breaking rule (lowest group index, then highest value), the algorithm removes $(1, 4)$. Now the impact of $(1, 3)$ is 0, the tuple $(2, 4)$ has impact 1, and the tuple $(3, 2)$ has impact -1. The algorithm removes the tuple $(2, 4)$. The process continues with the removal of $(1, 3)$, followed by $(2, 3)$, and at this point, no violations remain. □

For every aggregation function α, the impact of a tuple t (computed in lines 6-8 of Algorithm 4) can be calculated by running the query "SELECT α(A) GROUP BY G" twice: once with t and once without it. Then, S_{MVI} is calculated for each of the queries, and the impact is the difference between them. This process can be optimized to avoid the need to run these two queries for every tuple. We describe such optimizations in the full version [2]. The approximation ratio of HeurRepair can be arbitrarily high, as stated in the following proposition.

Proposition 6.2. *For $\alpha \in \{\mathsf{avg}, \mathsf{sum}, \mathsf{median}\}$, there are classes of instances where the approximation ratio of* HeurRepair *is $\Omega(n)$.*

In the proof (included in [2]), for each of the three aggregations, we show a database with n tuples where HeurRepair removes $\Omega(n)$ tuples while $O(1)$ tuples suffice for a C-repair.

Despite this large worst-case approximation ratio, the use of HeurRepair to prune the search of CardRepair can still lead to a reduction in the runtime of finding a C-repair, as we show in Section 7.2. For example, pruning has improved the runtime of CardRepair by up to 426 times for avg over 1K tuples and by up to 15 times for median over 10K tuples.

7 Experimental Evaluation

In our experiments, we aim to answer the following questions: (1) How do the repairs perform, and what do they delete in case studies? (Section 7.4.1) (2) How do the optimizations of the DP algorithm (Section 5) affect the runtime across various aggregation functions? (Section 7.2) (3) How do the algorithms perform in terms of runtime, with varying data sizes and varying amounts of noise? (Section 7.3) (4) What is the quality of the heuristic algorithm (Section 6), compared to the C-repair? (Section 7.4) (5) Can outlier removal methods be used to find an AOD repair or to accelerate AOD repairing? (Section 7.5)

Result summary: We found that aggregation-specific optimizations of WholePack$\langle\alpha\rangle$ in combination with pruning by a heuristic algorithm whenever possible consistently provided the largest

runtime improvement. Over all real world datasets, the largest improvement was recorded for the stack overflow dataset with avg, with a speedup of over 3 orders of magnitude. When comparing the algorithm runtimes for various aggregations, we found that in line with Theorem 4.1, the fastest aggregations were max, count and countd, and that the runtime advantage of the heuristic algorithm depends on the number of tuple removals necessary for a C-repair (an optimal repair). In terms of quality, the heuristic solution found C-repairs in all tested datasets and AODs except for Zillow (median aggregation) and Diabetes (avg aggregation). In these specific case studies, we found that the diabetes dataset deviated from the AOD age↗avg(diabetes) by 0.84% of tuples, and the Zillow dataset deviated from the AOD construction year↗median(listing price) by 2.6% of the tuples.

While outlier removal as a preliminary step before CardRepair was sometimes useful in speeding up the computation, it also resulted in many unnecessary tuple removals (over 10 times more than the C-repair). This suggests that repairing for an AOD typically requires more than just applying an outlier removal algorithm.

7.1 Experimental Setup

The implementation is in Python 3.11 and publicly available.[4] The experiments were done on a 2.50GHz CPU PC with 512GB memory.

Datasets and AODs. We analyze several publicly available datasets that contain intuitive near-trends.

- *German credit [33]* (1K tuples): This dataset classifies people as low or high credit risks. We focused on the probability of being classified as low-risk over the time a person has been in their current job. We expect to observe an increasing trend, as longer job tenure may reflect greater financial stability, which is typically associated with a higher credit score.
- *Stack Overflow*[5] (31,301 tuples): This dataset consists of responses from developers worldwide to questions about their jobs, including demographics, education level, role, and salary. We focused on the average salary, grouped by the education level (from elementary school to PhD). We expect to see an increasing trend, suggesting that with higher education, the average salary increases. We binned the salary attribute using equi-width bins of size \$1000, and truncated it at \$2,250,000, which is larger than 99% of the salary values in the data.
- *H&M [61]* (1,877,674 tuples): This dataset contains transactions of customers in H&M in 2019-2020, with features like the purchased item, the price, the age of the customer, etc. When analyzing the purchase prices by customer age, we found that overall, there was a clear trend where customers aged 25 spent the most money in H&M, with a clear decrease until the age of 40, where the sum of transaction prices begins rising again. The trend is violated from May to July 2020. We focused on this time period. We experimented with the sum and max aggregations.
- *Zillow*[6] (629926 tuples): This dataset contains information on real estate sales in three California counties in 2016. We focus on listing prices, grouped by the year of construction, and expect listing prices to rise with construction year.
- *Diabetes*[7] (100K tuples): This dataset contains information about diabetes diagnosis, risk factors, and complications of individuals. We examine the trend of diabetes prevalence across age groups, anticipating higher rates among older individuals. To that end, we divided the age attribute into equal-width bins, 5 years each.

[4] https://github.com/shunita/aod
[5] https://survey.stackoverflow.co/2022 (accessed April 2025)
[6] https://www.zillow.com/research/data/
[7] https://www.kaggle.com/datasets/iammustafatz/diabetes-prediction-dataset

Synthetic datasets. The synthetic data generation process is controlled by the following parameters: number n of tuples, number of groups, fraction f of noise tuples, and the number of violating groups. In our experiments, we used 10 groups in total and 4 violating groups. We first generate $n \cdot (1 - f)$ tuples, by uniformly sampling values for the aggregated attribute between 1 and 100, and uniformly dividing them into groups. Next, $f \cdot n$ noise tuples are generated by uniformly sampling aggregate values from (100, 120) and uniformly dividing them into the specified number of violating groups. This process means that the number of tuples that need to be removed to find a repair will be correlated with $n \cdot f$.

Algorithms. We evaluate two algorithms: CardRepair (Section 4), which is guaranteed to return a C-repair, and HeurRepair (Section 6), a heuristic algorithm that returns a monotonic subset. We also use a third alternative, that combines CardRepair with outlier removal methods. We bring this alternative to understand the role of outliers in the violation of AODs, and in particular, to understand if they capture a significant part of the violation. Specifically, since outlier removal is not guaranteed to return a monotonic subset, we first apply outlier removal methods, followed by CardRepair. We measure their effect on the size of the repair and the runtime. For the combined algorithm (*Outlier removal* + CardRepair), we experimented with three methods of outlier removal, of different types: Z-score [38], Local outlier factor [13] (a K-Nearest-Neighbors based method), and Isolation forest [44] (based on decision trees).

7.2 Evaluation of CardRepair Optimizations

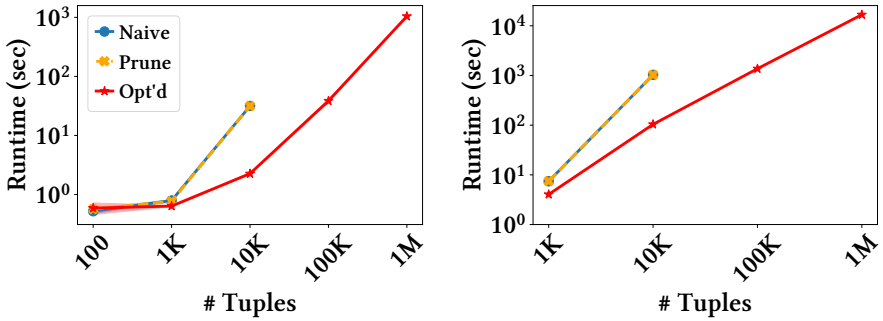
Next, we compare the WholePack$\langle\alpha\rangle$ variations with regard to the effect of optimizations on the runtime. We evaluate the following variations: *(1)* Naive WholePack$\langle\alpha\rangle$—computing $O_\alpha(r_i, x)$ for all $x \in V_\alpha(r_i)$ holistically, without optimizations (Section 5.2.1), *(2)* aggregation-specific optimization of WholePack$\langle\alpha\rangle$ (as described in Sections 5.2.2 to 5.2.4), *(3)* pruning WholePack$\langle\alpha\rangle$ using a bound from a heuristic algorithm (Section 5.1), and *(4)* a combination of pruning and aggregation-specific optimizations. Experiments with dictionary pruning (Section 5.1) are included in the full version [2].

For each aggregation, we compare the optimizations over increasing sample sizes from the datasets, as well as on synthetic datasets. For the scale experiments shown later, we created samples of the Stack Overflow, H&M and Zillow datasets. For Zillow and H&M we used sample sizes of different orders of magnitude. For SO, we used smaller samples (up to 1K) since the avg aggregation function used with this dataset is the hardest variation of the problem. For each sample size (and each synthetic dataset size), we performed the sampling (or generation) three times.

The most effective optimization for median was the combination of the value-based iteration and dictionary pruning (Section 5.2.2): for real data, the optimized algorithm was up to 860× faster than the naive algorithm (which timed out at 50K tuples). This combination of optimizations, adjusted for avg (Section 5.2.4) was 2282 times faster than the naive algorithm for 1K rows of real world data. For sum, the knapsack-based optimization of Section 5.2.3 led to the greatest improvement on real world data - up to 92× faster than the naive algorithm (which timed out at 100K tuples).

7.2.1 WholePack$\langle$sum$\rangle$ *optimizations.* For sum we compare variations (1)-(3), since a combination of the two optimizations is not possible. The three variations are compared over increasing sample sizes from the H&M dataset (Figure 3a), and increasing number of generated synthetic rows (Figure 3b).

The most effective optimization was the knapsack-based algorithm (red in Figure 3). Pruning the search by a bound on the number of removed tuples had almost no effect on the search time. This is likely because the main bottleneck of the WholePack$\langle\alpha\rangle$ procedure for sum is in combining each tuple with all of the previously reached subsets. Additionally, the C-repairs in the H&M samples required removing 30 tuples on average. Therefore, the bounds yielded by HeurRepair were not



(a) Samples of the H&M dataset. (b) Synthetic data.

Fig. 3. Comparison of runtime performance for WholePack⟨sum⟩ optimizations over increasing number of tuples. In Figure 3b, the timeout was 20 hours.

small enough to make a significant reduction to the runtime. Yet, for avg and median, we observed a runtime decrease when pruning by the HeurRepair result.

7.2.2 WholePack⟨median⟩ *optimizations.* For median, we compare four variations: the naive WholePack⟨median⟩, pruning by a bound from HeurRepair (Section 5.1), optimizing it using value-based iteration (Section 5.2.2), or combining the two optimizations. We compared them over increasing samples from the Zillow dataset (Figure 4a), and increasing number of rows generated by the synthetic dataset generator (Figure 4b).

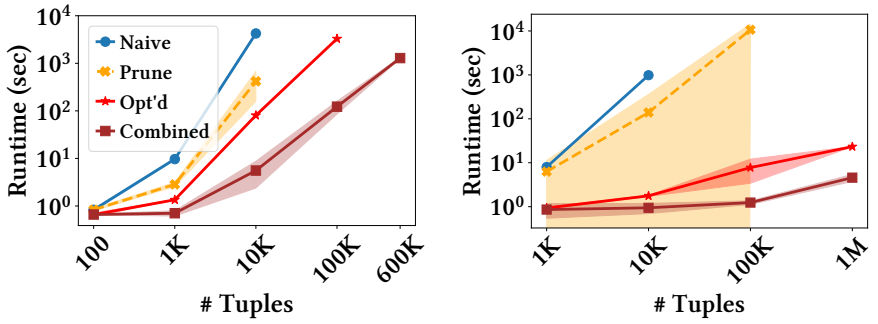
In both real and synthetic data, the most effective optimization was the combination of pruning and value-based iteration. Iterating over unique values rather than tuples reduced the number of iterations. Additionally, pruning allowed us to consider fewer options per value. For the heuristic pruning in the synthetic data experiment, we saw a high variance in the runtime measurements. The runtime of this optimization greatly depends on the number of tuples removed by HeurRepair,which is farthest from optimal for the median aggregation (more on this in the full version [2].

7.2.3 WholePack⟨avg⟩ *optimizations.* For avg (as for median), we compare four variations: the naive WholePack⟨avg⟩, pruning the procedure by a bound from HeurRepair (Section 5.1), optimizing it using value-based iteration (Section 5.2.4), or combining the two optimizations. We compared them over increasing samples from the SO dataset (Figure 5a), and increasing number of rows generated by the synthetic dataset generator (Figure 5b). In both cases we sampled up to 1K rows, since WholePack⟨avg⟩ still poses a computational challenge, even with the optimizations.

For avg, the combination of both optimizations worked best. Pruning by the heuristic bound had a greater effect on real data than it did on the generated synthetic data, due to the number of tuples that need to be removed (and therefore the size of the bound): in the real dataset, removing a small number of tuples (0.1%) yields a repair. In the synthetic dataset, the repair requires removing 10% of the data (10% of the generated tuples were noise tuples).

7.3 Comparing Between Aggregations

We compare the runtimes CardRepair and HeurRepair on different aggregation functions. CardRepair runs substantially longer for median, sum, and avg. For each aggregation function, we chose the version of CardRepair that was fastest according to the experiments in Section 7.2. To compare all aggregations on the same dataset, we generated synthetic datasets (as described in Section 7.1) ranging from 10K to 1M tuples, with 10% being noise tuples.



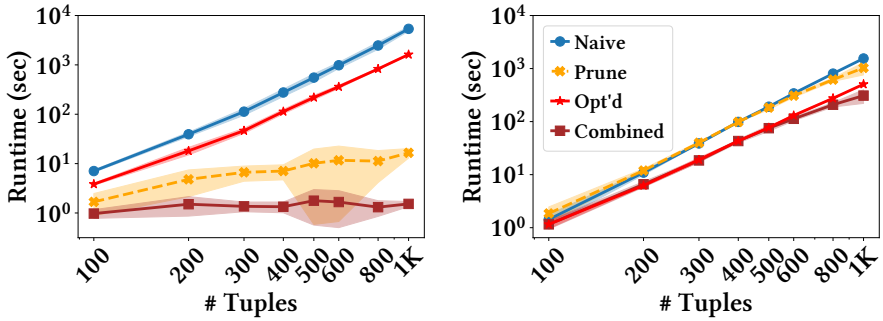
(a) Samples of the Zillow dataset. (b) Synthetic data.

Fig. 4. Runtime performance for WholePack⟨median⟩ optimizations for different numbers of tuples.

To evaluate the effect of the amount of violation on the algorithm runtimes, we generated 100K-tuple datasets with varying numbers of noise tuples (1K to 15K). In both experiments, for avg we included only HeurRepair, since CardRepair with avg remains a scalability challenge. For count and countd, we included only CardRepair results; since CardRepair is fast for these aggregations, we did not implement an instantiation of HeurRepair for them.

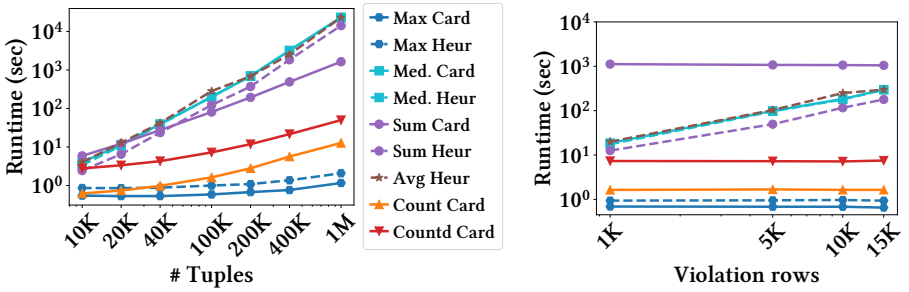
The runtimes versus the increasing number of tuples are shown in Figure 6a (A version of Figure 6 with confidence intervals is available in the full version [2].) As expected, CardRepair was fastest for max, count, and countd, where the computation of $O_\alpha(r_i, x)$ is linear in the number of tuples in r_i. CardRepair was slower for the non-linear aggregations: median, sum and avg.

Interestingly, for sum, HeurRepair was faster than CardRepair only for tables with fewer than 80K tuples, since HeurRepair removes one tuple at a time. As the number of tuples to remove increases with the size of the dataset, HeurRepair takes longer. This can also be observed in Figure 6b, showing the runtimes versus the number of noise tuples: the time of HeurRepair as well as CardRepair for median that uses greedy-based pruning, increase with the number of noise tuples, while other times are not affected.



(a) Samples of the SO dataset. (b) Synthetic data.

Fig. 5. Runtime performance for WholePack⟨avg⟩ optimizations over increasing number of tuples (log-scale).



(a) Increasing # tuples.

(b) Increasing # noise tuples.

Fig. 6. Run times of CardRepair and HeurRepair with various aggregations over synthetic data.

7.4 Evaluation of the Heuristic Algorithm

We next evaluate HeurRepair compared to CardRepair in terms of quality (number of tuples removed) and their runtime.

Result Summary: HeurRepair outperformed CardRepair in terms of runtime across all tested cases. For avg over Stack Overflow, it was more than 2000 times faster, while for max over H&M, the speedup was more modest at 2.5 times. In terms of solution quality, HeurRepair reached a C-repair for most cases, except median for Zillow and avg for the Diabetes dataset. For these two cases, we include an analysis of the differences in the repairs found by the two algorithms in Section 7.4.1.

We ran the best variation of CardRepair per each aggregation function as found in the previous experiments (Section 7.2). For HeurRepair, we applied the relevant optimizations described in the full version [2]. The results can be seen in Table 1 and Figure 7.

HeurRepair was faster than CardRepair in all tested cases. The largest difference was for edu↗avg(salary) in Stack Overflow—over 2000× faster; the smallest was for age↘max(price) in H&M (2.5× faster). For most of the tested AODs, HeurRepair found a C-repair. For age↗avg(diabetes) and year↗median(value), it removed more tuples than necessary (1.37× and 1.61× more tuples than CardRepair, respectively). A possible reason can be that for avg, and median, we lack monotonicity—the removal of a tuple can either increase or decrease the aggregate value of the group.

Another issue is that for median, the impact of the removed tuple on the aggregate value of the group (and on the sum of violations) depends on the other tuples in the group (specifically those in proximity to the current median). In these cases, HeurRepair that only sees a single step ahead is likely to make suboptimal choices. An example of this issue is provided in the full version [2].

Table 1. CardRepair and HeurRepair over real datasets.

Dataset	α	HeurRepair		CardRepair	
		time (s)	**#tuples removed**	**time (s)**	**#tuples removed**
Zillow	median	4912	22481	59904	16361
H&M	sum	57.53	1464	4441	1464
H&M	max	0.7522	43	1.91	43
German	avg	0.0126	16	0.19	16
SO	avg	0.61	37	1082	37
Diabetes	avg	4.95	834	306	518

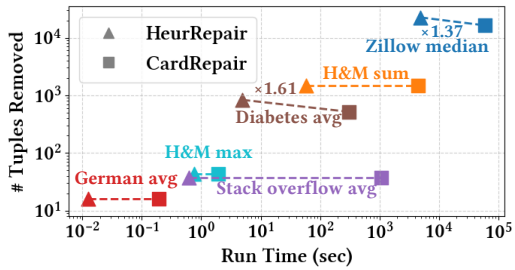
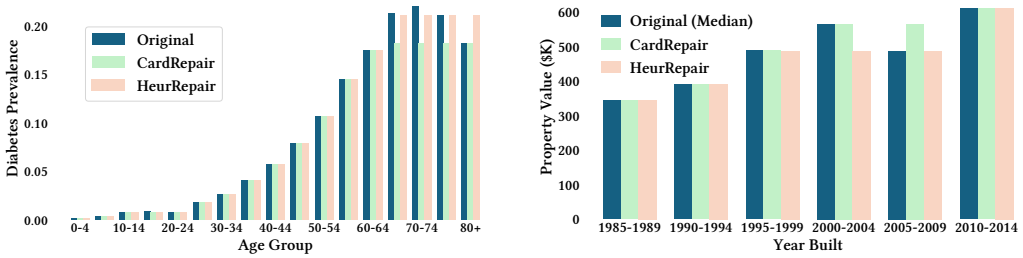


Fig. 7. Runtime and number of tuples removed (log scale) by HeurRepair and CardRepair. The number near the dashed line represents the removal overhead of HeurRepair (triangles) relative to CardRepair (squares).



(a) Diabetes prevalence by age.

(b) Zillow property value median vs. year built.

Fig. 8. Original trend and repair analyses from health and housing domains.

7.4.1 Drill Down on Removed Tuples. We next present two in-depth analyses, for the datasets where the HeurRepair reached a different solution size than CardRepair (analyses for additional datasets are included in the full version [2]). Our goal is to highlight the differences between CardRepair and HeurRepair in terms of the tuples they remove, and to demonstrate how our approach can help explain why a trend does not fully hold.

Diabetes. Result Summary: HeurRepair removed 834 (0.83%) tuples in 4.95 seconds, while CardRepair removed 316 fewer tuples (0.52% removed) but took 306.18 seconds - over 61 times more than HeurRepair, as shown in Table 1.

Initial AOD violations. Consider the average diabetes prevalence over age as depicted in Figure 8a (blue bars). There is a visible trend: as age increases, the prevalence of diabetes is higher. However, there are some violations to this AOD: $\text{avg}_r(\text{diabetes}|\text{age}{=}15{-}19) = 0.009$ while $\text{avg}_r(\text{diabetes}|\text{age} = 20{-}24) = 0.007$, and there is a decrease among the groups $70-74, 75-79$ and 80+.

Comparison of removed tuples. The data is close to satisfying the AOD. HeurRepair (Figure 8a, orange bars) executed in 4.95 seconds, removing 834 tuples (0.84%): 8 diabetes patients aged 15-19, to fix the first violation, as well as 60 diabetes patients aged 65-75, and 766 healthy individuals aged over 75, to fix the second violation. CardRepair (green bars) executed in 306.18 seconds and removed 518 diabetes patients (0.52%). Of these, it removed 8 patients aged 15-19, as HeurRepair did, and 510 patients aged 65-79.

The greedy nature of HeurRepair, which performs the most impactful single-tuple removal at each iteration, resulted in a sub-optimal repair; it removed 1.61 more tuples. However, it had a less drastic impact on the aggregation query result, as can be seen in Figure 8a. The aggregation values of the heuristic repair remained closer to the original values compared to those of CardRepair.

Zillow. Result Summary: HeurRepair removed 22,481 (3.57%) tuples in 39.8 minutes. In comparison, CardRepair removed 6120 fewer tuples (2.6%), but took 16.64 hours to run (12.2 times more than HeurRepair), as shown in Table 1.

Initial AOD violations. We focus on the trend of house prices by the year they were built. The years were binned to 5-year terms: 1985-1989 (inclusive) until 2010-2014 (inclusive). We observed (Figure 8b, blue bars) that the median of house prices, grouped by these 5-year terms, is increasing but with an outlier in the years 2005-2009 ($487K, versus $565K in 2000-2004). That is, the AOD year $\nearrow$ median(price) almost holds.

Comparison of removed tuples. HeurRepair (Figure 8b, orange bars) terminated in 39.8 minutes, removing 22,481 (3.6%) tuples, most (22,120) from the 2005-2009 group and the rest from 2000-2004. CardRepair (Figure 8b, green bars) took 16.64 hours and removed 16,361 tuples (2.6% of the dataset), all from the 2005-2009 group.

Comparing the tuples removed by CardRepair with those that remained, we observed notable differences in land use distribution. Among the removed tuples, 62.7% were condominium properties and 26.91% were single-family residences. In contrast, the remaining tuples showed an almost reverse pattern: only 34.36% were condominium properties, while 57.31% were single-family residences. This suggests a connection between condominium apartments built between 2005 and 2009 and lower property values, which may be related to the housing bubble in those years. We also analyzed the geographic distribution of the removed versus remaining properties. The most significant difference was observed in Los Angeles, which accounted for 29.2% of the removed tuples but only 13.17% of the remaining ones. A possible reason for this is that the Los Angeles housing market was one of the most rapidly deflating markets at the end of the housing bubble in 2008 [6].

Distance from trend. As mentioned previously, a C-repair for an AOD can serve as a measure of the distance of a dataset from satisfying a trend. When applying HeurRepair to find a repair for the opposite trend (a decrease in median house prices with increasing year), it removed 405K tuples (64.3%), 18.03 times more than for the increasing trend. Hence, the distance from the increasing trend is significantly smaller than the distance from the decreasing one. Note that HeurRepair does not provide any guarantees on the minimum number of tuples to be removed to satisfy each trend. For such guarantees, we need CardRepair.

When applying CardRepair to find a repair for the decreasing trend, the repair required removing $146.6K$ tuples (23.3%). Due to the optimality of the algorithm, it provides a lower bound on the number of tuples to be removed to satisfy the trend in each direction. Therefore, the dataset is 8.96 times closer to satisfying an increasing trend than to satisfying a decreasing trend.

From Figure 8, it may seem that CardRepair only equates the aggregation values of consecutive groups. Note, however, that in Figure 8a, the values are close but not identical, while in Figure 8b (with median aggregation) they became equal. This behavior (making two consecutive groups have close values) is an inherent property of the problem. When each tuple has a small effect on the aggregation value of the group, a C-repair will result in close aggregation values. The heuristic algorithm HeurRepair aims to utilize this property by trying to match the value of a group with its predecessor group. Yet, as we saw in the experiments, this approach may be overly simplistic and may be considerably suboptimal (see Section 7.4.1 and Figure 7).

7.5 Comparison to Outlier Detection Methods

We also compare our solutions to outlier removal—a common strategy in data cleaning where violations of expected trends are attributed to noise [12, 69]rather than legitimate data items. Like our algorithms, outlier removal methods work by deleting a subset of tuples, but they rely on

Table 2. Outlier removal on three settings. For each method, the results for the best parameter choice are shown for global and for per-group outlier removal (separated by "/").

Setting	Method	Best param	# Removed outliers	# Removed total
SO avg	Z score	6/6	369/373	369/373
SO avg	LOF	3/8	318/972	**356**/990
SO avg	Iso.Forest	0.05/0.05	1519/1556	1526/1561
H&M max	Z score	6/2	1356/83664	**1356**/84934
H&M max	Iso.Forest	0.005/0.01	8489/16657	8489/21258
H&M sum	Z score	1/1.5	462973/117349	465770/118930
H&M sum	Iso.Forest	0.2/0.01	367546/16657	369816/**18083**

statistical rarity rather than explicitly targeting monotonicity violations. This comparison allows us to quantify the benefit of using AOD-aware repair algorithms over generic noise-removal approaches. Therefore, we next aim to answer our 5th research question from the beginning of Section 7: can existing outlier removal methods be used to find a repair for an AOD?

We consider three settings: the SO dataset with the AOD edu $\nearrow$ avg(salary), and H&M with two AODs: age $\searrow$ max(price) and age $\searrow$ sum(price). Out of the real-world scenarios, these are most likely to be affected by outliers: for Diabetes and German credit, all values of the aggregated attribute are 0 or 1 (there are no outliers to be found); and the median function used with Zillow is less affected by outliers than avg, max and sum.

We experimented with three outlier removal methods using the scikit-learn implementations with hyper-parameter tuning. (1) *Z-score* [38] models $r[\![A]\!]$ as samples from a normal distribution, and removes points whose distance from the mean is over τ standard deviations; we used $1 \leq \tau \leq 6$. (2) *Local outlier factor (LOF)* [13] detects samples with substantially lower density compared to the k closest neighbors are considered outliers. We tried k values in $\{3, 5, 8, 10\}$. (3) *Isolation forest* [44] trains random decision trees to isolate observations; easily isolated samples (with shorter path lengths in the decision tree) are considered outliers. A contamination parameter controls the number of expected outliers. We tried contamination values between 0.001 and 0.2. For each method, we also examined a variation where the outlier removal is done per group r_i rather than the full dataset.

For each dataset setting, we applied all combinations of outlier removal method and hyper-parameters as a preliminary step, selected the combinations that were closest to a repair according to the sum of violations after removal, and applied CardRepair after the removal to find a C-repair.

The results of the best parameters for each combination of scenario and outlier removal method are shown in Table 2 (the full tables are available in the full version [2]). For each outlier removal method, we report the number of removed outliers and the total number of tuple removals including those removed by CardRepair. Note that due to the size of the dataset, the LOF method timed out at 2 hours for the H&M dataset (for comparison, CardRepair took under 1 minute for this dataset).

Most variations of outlier removal methods did not yield a monotonic subset (namely, a valid solution). For the sum scenario, none of the methods yielded a monotonic subset, all of them removed over 30 times more tuples than necessary, and CardRepair had to remove a similar number of tuples as without the preliminary outlier detection. For the avg and max scenarios, some of the outlier removal methods did return a monotonic subset satisfying the AOD, but they required removing 10 times more tuples than the C-repair.

Still, outlier removal can occasionally accelerate CardRepair. For example, for SO with avg, Isolation forest with 0.05 contamination (in both the group-wise and the non group-wise settings), there were only a few remaining tuples to remove. In both cases, HeurRepair yielded tight bounds, thus greatly shortening the runtimes of CardRepair (60.5 seconds and 115.9 seconds, respectively).

Overall, we conclude that *repairing for an AOD typically requires more than just applying outlier detection, as the tuples that obscure the trend are not necessarily statistical outliers*. They may appear structurally standard, but still interfere with the expected trend.

8 Concluding Remarks

We introduced the notion of an aggregate order dependency (AOD) and investigated the problem of computing a cardinality-based repair for databases that violate an AOD. This framework serves as a principled approach to analyzing violations of expected monotonic trends in data. We analyzed the computational complexity of the repair problem, proposed a general algorithmic framework (CardRepair), and provided tailored optimizations for common aggregate functions. We also presented a heuristic alternative (HeurRepair) that is often highly effective (that is, fast and close to optimal quality). Our experimental evaluation demonstrates the practical effectiveness of the proposed methods and highlights, through analysis of the removed tuples, the utility of our framework in quantifying and interpreting trend violations.

As for the next steps, a staggering open challenge lies in handling the *average* aggregate function, which remains computationally demanding and requires further optimization techniques; this difficulty is echoed in related contexts such as Shapley value computation [1]. Furthermore, CardRepair has theoretical guarantees but it is computationally heavy, while HeurRepair is usually faster but its quality is not guaranteed. To close this gap, it would be helpful to find effective algorithms with nontrivial guaranteed approximation.

Beyond that, several promising directions remain open for future work. First, our techniques can likely be extended to incorporate constraints on the *magnitude of change* between consecutive groups, such as a minimum absolute or relative difference, thereby capturing stronger forms of monotonicity. Second, a natural generalization is to accommodate *multiple* AODs, enabling the assessment of how closely a dataset conforms to multiple monotonic trends simultaneously. Third, we can extend the framework to support a richer class of AODs, like ones involving aggregation over joins of multiple relations, negation, etc. Another important extension is to support intervention models beyond tuple deletion, including updates of cell values and insertion of new tuples.

Finally, as an alternative to the number of removed tuples, we can explore different *measures of intervention*, such as a weighted sum of tuples, assuming that different tuples are associated with different weights (e.g., representing confidence scores). Another measure of intervention is the complexity of the description of removed tuples; in this context, we are currently exploring a variation of the framework where we delete batches of tuples using tuple templates, thereby seeking populations of deleted tuples that are characterized by easy-to-explain database queries (e.g., people of a certain age group), in the vein of previous work on pattern-based explanations [3, 35, 62, 63, 66]; yet, while previous work has focused on *abductive* explanations (what to keep in the database), our focus is on *contrastive* explanations (what to remove).

Acknowledgments

The work of Amir Gilad was funded by the Israel Science Foundation (ISF), Grant 1702/24, the Scharf-Ullman Endowment, and the Alon Scholarship. The work of Ester Livshits and Brit Youngman was partially funded by the United States-Israel Binational Science Foundation, Grant 2024101, and by the ISF, Grant 934/25. The work of Shunit Agmon, Ester Livshits, and Benny Kimelfeld was partially funded by the ISF, Grant 863/25. The work of Jonathan Gal was partially funded by the ISF, Grant 3001/24.

References

[1] Omer Abramovich, Daniel Deutch, Nave Frost, Ahmet Kara, and Dan Olteanu. 2025. Advancing Fact Attribution for Query Answering: Aggregate Queries and Novel Algorithms. arXiv:2506.16923 [cs.DB] https://arxiv.org/abs/2506.16923

[2] Shunit Agmon, Jonathan Gal, Amir Gilad, Ester Livshits, Or Mutay, Brit Youngmann, and Benny Kimelfeld. 2025. Analyzing Deviations from Monotonic Trends through Database Repair. arXiv:2512.08526 [cs.DB] https://arxiv.org/abs/2512.08526

[3] Shunit Agmon, Amir Gilad, Brit Youngmann, Shahar Zoarets, and Benny Kimelfeld. 2024. Finding Convincing Views to Endorse a Claim. *Proc. VLDB Endow.* 18, 2 (2024), 439–452. https://www.vldb.org/pvldb/vol18/p439-agmon.pdf

[4] Abolfazl Asudeh, Hosagrahar Visvesvaraya Jagadish, You Wu, and Cong Yu. 2020. On detecting cherry-picked trendlines. *VLDB* 13, 6 (2020), 939–952.

[5] Abolfazl Asudeh, You Will Wu, Cong Yu, and HV Jagadish. 2021. Perturbation-based Detection and Resolution of Cherry-picking. *A Quarterly bulletin of the Computer Soc. of the IEEE Technical Committee on Data Engineering* 45, 3 (2021), 39–51.

[6] Dean Baker. 2008. The housing bubble and the financial crisis. *Real-world economics review* 46, 20 (2008), 73–81.

[7] Leopoldo E. Bertossi. 2011. *Database Repairing and Consistent Query Answering.* Morgan & Claypool Publishers.

[8] Leopoldo E. Bertossi. 2019. Repair-Based Degrees of Database Inconsistency. In *LPNMR (Lecture Notes in Computer Science, Vol. 11481).* Springer, 195–209.

[9] Arnab Bhattacharyya, Eldar Fischer, Ronitt Rubinfeld, and Paul Valiant. 2011. Testing monotonicity of distributions over general partial orders.. In *ICS*. Citeseer, 239–252.

[10] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2006. Conditional functional dependencies for data cleaning. In *2007 IEEE 23rd international conference on data engineering*. IEEE, 746–755.

[11] Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. 2005. A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification. In *SIGMOD Conference*. ACM, 143–154.

[12] Sanae Borrohou, Rachida Fissoune, and Hassan Badir. 2023. Data cleaning survey and challenges–improving outlier detection algorithm in machine learning. *Journal of Smart Cities and Society* 2, 3 (2023), 125–140.

[13] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. 2000. LOF: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 93–104.

[14] Hugh D Brunk. 1955. Maximum likelihood estimates of monotone parameters. *The Annals of Mathematical Statistics* (1955), 607–616.

[15] Nofar Carmeli, Martin Grohe, Benny Kimelfeld, Ester Livshits, and Muhammad Tibi. 2021. Database Repairing with Soft Functional Dependencies. In *24th International Conference on Database Theory, ICDT 2021, March 23-26, 2021, Nicosia, Cyprus (LIPIcs, Vol. 186)*, Ke Yi and Zhewei Wei (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:17.

[16] Jan Chomicki and Jerzy Marcinkowski. 2005. Minimal-change integrity maintenance using tuple deletions. *Information and Computation* 197, 1-2 (2005), 90–121.

[17] Jan Chomicki and Jerzy Marcinkowski. 2005. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.* 197, 1-2 (2005), 90–121.

[18] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 458–469. doi:10.1109/ICDE.2013.6544847

[19] Cristian Consonni, Paolo Sottovia, Alberto Montresor, Yannis Velegrakis, et al. 2019. Discovering Order Dependencies through Order Compatibility.. In *EDBT*. 409–420.

[20] Jirun Dong and Richard Hull. 1982. Applying approximate order dependency to reduce indexing space. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data* (Orlando, Florida) *(SIGMOD '82)*. Association for Computing Machinery, New York, NY, USA, 119–127. doi:10.1145/582353.582375

[21] Wenfei Fan and Floris Geerts. 2012. *Foundations of Data Quality Management.* Morgan & Claypool Publishers.

[22] Sergio Flesca, Filippo Furfaro, and Francesco Parisi. 2007. Preferred database repairs under aggregate constraints. In *Scalable Uncertainty Management: First International Conference, SUM 2007, Washington, DC, USA, October 10-12, 2007. Proceedings 1*. Springer, 215–229.

[23] Sergio Flesca, Filippo Furfaro, and Francesco Parisi. 2010. Querying and repairing inconsistent numerical databases. *ACM Transactions on Database Systems (TODS)* 35, 2 (2010), 1–50.

[24] Sergio Flesca, Filippo Furfaro, and Francesco Parisi. 2011. *Repairing and querying databases under aggregate constraints.* Springer Science & Business Media.

[25] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2013. The LLUNATIC data-cleaning framework. *Proceedings of the VLDB Endowment* 6, 9 (2013), 625–636.

[26] Subhashis Ghosal, Arusharka Sen, and Aad W Van Der Vaart. 2000. Testing monotonicity of regression. *Annals of statistics* (2000), 1054–1082.

[27] Amir Gilad, Daniel Deutch, and Sudeepa Roy. 2020. On Multiple Semantics for Declarative Database Repairs. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 817–831. doi:10.1145/3318464.3389721

[28] Amir Gilad, Aviram Imber, and Benny Kimelfeld. 2023. The Consistency of Probabilistic Databases with Independent Cells. In *ICDT (LIPIcs, Vol. 255)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:19.

[29] Seymour Ginsburg and Richard Hull. 1983. Order Dependency in the Relational Model. *Theor. Comput. Sci.* 26 (1983), 149–195.

[30] Seymour Ginsburg and Richard Hull. 1986. Sort sets in the relational model. *J. ACM* 33, 3 (May 1986), 465–488. doi:10.1145/5925.5929

[31] Germán González-Almagro, Pablo Sánchez-Bermejo, Juan Luis Suarez, José-Ramón Cano, and Salvador García. 2024. Semi-supervised clustering with two types of background knowledge: Fusing pairwise constraints and monotonicity constraints. *Inf. Fusion* 102, C (Feb. 2024), 15 pages. doi:10.1016/j.inffus.2023.102064

[32] Peter Hall and Nancy E Heckman. 2000. Testing for monotonicity of a regression mean by calibrating for linear functions. *Annals of Statistics* (2000), 20–39.

[33] Hans Hofmann. 1994. Statlog (German Credit Data). UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5NC77.

[34] Mohamed Hussian, Anders Grimvall, Oleg Burdakov, and Oleg Sysoev. 2005. Monotonic regression for the detection of temporal trends in environmental quality data. *MATCH Commun. Math. Comput. Chem* 54 (2005), 535–550.

[35] Ibrahim A. Ibrahim, Xue Li, Xin Zhao, Sanad Al Maskari, Abdullah M. Albarrak, and Yanjun Zhang. 2018. Automated Explanations of User-Expected Trends for Aggregate Queries. In *Advances in Knowledge Discovery and Data Mining*, Dinh Phung, Vincent S. Tseng, Geoffrey I. Webb, Bao Ho, Mohadeseh Ganji, and Lida Rashidi (Eds.). Springer International Publishing, Cham, 602–614.

[36] Yifeng Jin, Lin Zhu, and Zijing Tan. 2020. Efficient Bidirectional Order Dependency Discovery. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 61–72. doi:10.1109/ICDE48307.2020.00013

[37] Saehan Jo, Immanuel Trummer, Weicheng Yu, Xuezhi Wang, Cong Yu, Daniel Liu, and Niyati Mehta. 2019. Aggchecker: A fact-checking system for text summaries of relational data sets. *VLDB* 12, 12 (2019), 1938–1941.

[38] Senthamarai Kannan Kaliyaperumal, Manoj Kuppusamy, and Arumugam Subbanna Gounder. 2015. Outlier detection and missing value in time series ozone data. *International Journal of Scientific Research in Knowledge* 3, 9 (2015), 220–226.

[39] Youri Kaminsky, Benny Kimelfeld, Ester Livshits, Felix Naumann, and David Wajc. 2025. Repairing Databases over Metric Spaces with Coincidence Constraints. In *ICDT (LIPIcs, Vol. 328)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:18.

[40] Solmaz Kolahi and Laks V. S. Lakshmanan. 2009. On approximating optimum repairs for functional dependency violations. In *ICDT (ACM International Conference Proceeding Series, Vol. 361)*. ACM, 53–62.

[41] Philipp Langer and Felix Naumann. 2016. Efficient order dependency detection. *The VLDB Journal* 25 (2016), 223–241.

[42] Sokbae Lee, Oliver Linton, and Yoon-Jae Whang. 2009. Testing for stochastic monotonicity. *Econometrica* 77, 2 (2009), 585–602.

[43] Yin Lin, Brit Youngmann, Yuval Moskovitch, HV Jagadish, and Tova Milo. 2021. On detecting cherry-picked generalizations. *Proceedings of the VLDB Endowment* 15, 1 (2021), 59–71.

[44] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *2008 eighth ieee international conference on data mining*. IEEE, 413–422.

[45] Ester Livshits, Benny Kimelfeld, and Sudeepa Roy. 2020. Computing Optimal Repairs for Functional Dependencies. *ACM Trans. Database Syst.* 45, 1 (2020), 4:1–4:46.

[46] Ester Livshits, Rina Kochirgan, Segev Tsur, Ihab F. Ilyas, Benny Kimelfeld, and Sudeepa Roy. 2021. Properties of Inconsistency Measures for Databases. In *SIGMOD Conference*. ACM, 1182–1194.

[47] Yasir Mahmood, Jonni Virtema, Timon Barlag, and Axel-Cyrille Ngonga Ngomo. 2024. Computing Repairs Under Functional and Inclusion Dependencies via Argumentation. In *FoIKS (Lecture Notes in Computer Science, Vol. 14589)*. Springer, 23–42.

[48] Silvano Martello and Paolo Toth. 1984. A mixture of dynamic programming and branch-and-bound for the subset-sum problem. *Management Science* 30, 6 (1984), 765–771.

[49] Dongjing Miao, Pengfei Zhang, Jianzhong Li, Ye Wang, and Zhipeng Cai. 2023. Approximation and inapproximability results on computing optimal repairs. *VLDB J.* 32, 1 (2023), 173–197.

[50] Shubhankar Mohapatra, Amir Gilad, Xi He, and Benny Kimelfeld. 2025. Computing Inconsistency Measures Under Differential Privacy. *Proc. ACM Manag. Data* 3, 3, Article 140 (June 2025), 27 pages. https://doi.org/10.1145/3725397

[51] Wilfred Ng. 1999. Ordered functional dependencies in relational databases. *Information Systems* 24, 7 (1999), 535–554. doi:10.1016/S0306-4379(99)00031-9

[52] Wilfred Ng. 2001. An extension of the relational data model to incorporate ordered domains. *ACM Trans. Database Syst.* 26, 3 (Sept. 2001), 344–383. doi:10.1145/502030.502033
[53] Sergey Ovchinnik, Fernando E. B. Otero, and Alex A. Freitas. 2019. Monotonicity Detection and Enforcement in Longitudinal Classification. In *Artificial Intelligence XXXVI*, Max Bramer and Miltos Petridis (Eds.). Springer International Publishing, Cham, 63–77.
[54] Andrew J Patton and Allan Timmermann. 2010. Monotonicity in asset returns: New tests with applications to the term structure, the CAPM, and portfolio sorts. *Journal of Financial Economics* 98, 3 (2010), 605–625.
[55] Judea Pearl. 2009. *Causality*. Cambridge university press.
[56] Vincent Poirriez, Nicola Yanev, and Rumen Andonov. 2009. A hybrid algorithm for the unbounded knapsack problem. *Discrete Optimization* 6, 1 (2009), 110–124. doi:10.1016/j.disopt.2008.09.004
[57] Prem S Puri and Harshinder Singh. 1990. On recursive formulas for isotonic regression useful for statistical inference under order restrictions. *Journal of statistical planning and inference* 24, 1 (1990), 1–11.
[58] Yu Qiu, Zijing Tan, Kejia Yang, Weidong Yang, Xiangdong Zhou, and Naiwang Guo. 2018. Repairing Data Violations with Order Dependencies. In *Database Systems for Advanced Applications*, Jian Pei, Yannis Manolopoulos, Shazia Sadiq, and Jianxin Li (Eds.). Springer International Publishing, Cham, 283–300.
[59] James O Ramsay. 1998. Estimating smooth monotone functions. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 60, 2 (1998), 365–375.
[60] 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. doi:10.14778/3137628.3137631
[61] Joshua Robinson, Rishabh Ranjan, Weihua Hu, Kexin Huang, Jiaqi Han, Alejandro Dobles, Matthias Fey, Jan E. Lenssen, Yiwen Yuan, Zecheng Zhang, Xinwei He, and Jure Leskovec. 2024. RelBench: A Benchmark for Deep Learning on Relational Databases. arXiv:2407.20060 [cs.LG] https://arxiv.org/abs/2407.20060
[62] Sudeepa Roy, Laurel Orr, and Dan Suciu. 2015. Explaining query answers with explanation-ready databases. *Proceedings of the VLDB Endowment* 9, 4 (2015), 348–359.
[63] Sudeepa Roy and Dan Suciu. 2014. A formal approach to finding explanations for database queries. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1579–1590.
[64] Babak Salimi, Luke Rodriguez, Bill Howe, and Dan Suciu. 2019. Interventional fairness: Causal database repair for algorithmic fairness. In *Proceedings of the 2019 International Conference on Management of Data*. 793–810.
[65] Jef Wijsen. 2001. Trends in databases: Reasoning and mining. *IEEE Transactions on Knowledge and Data Engineering* 13, 3 (2001), 426–438.
[66] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. *Proceedings of the VLDB Endowment* 6, 8 (2013), 553–564.
[67] You Wu, Pankaj K Agarwal, Chengkai Li, Jun Yang, and Cong Yu. 2014. Toward computational fact-checking. *Proceedings of the VLDB Endowment* 7, 7 (2014), 589–600.
[68] You Wu, Pankaj K Agarwal, Chengkai Li, Jun Yang, and Cong Yu. 2017. Computational fact checking through query perturbations. *ACM Transactions on Database Systems (TODS)* 42, 1 (2017), 1–41.
[69] Hui Xiong, Gaurav Pandey, Michael Steinbach, and Vipin Kumar. 2006. Enhancing data analysis with noise removal. *IEEE transactions on knowledge and data engineering* 18, 3 (2006), 304–319.
[70] Yun Zhou, Norman Fenton, and Cheng Zhu. 2016. An empirical study of Bayesian network parameter learning with monotonic influence constraints. *Decision Support Systems* 87 (2016), 69–79. doi:10.1016/j.dss.2016.05.001

Received July 2025; revised October 2025; accepted November 2025