MonoTune: Analyzing Trend Deviations through Database Repair

Shunit Agmon
Technion
Haifa, Israel
shunita@campus.technion.ac.il

Itai Manor
Hebrew University
Jerusalem, Israel
itai.manor@mail.huji.ac.il

Brit Youngmann
Technion
Haifa, Israel
brity@technion.ac.il

Benny Kimelfeld
Technion
Haifa, Israel
bennyk@cs.technion.ac.il

Amir Gilad
Hebrew University
Jerusalem, Israel
amirg@cs.huji.ac.il

Abstract

Datasets often violate expected monotonic trends. For instance, average income is typically expected to increase with education level, and housing prices to rise over time. Recent work studied how to quantify such deviations by introducing *Aggregate Order Dependencies* (AODs), an aggregation-centric extension of order dependencies, and by measuring the extent of violations via minimal repairs. Building on the AOD framework, this paper presents a demonstration of MonoTune, an interactive system for detecting, quantifying, and explaining violations of monotonic trends. MonoTune enables users to upload a dataset, specify an expected trend using an AOD, visualize violations directly in the results, compute repairs, and summarize the changes induced by these repairs. The system supports both optimal repair algorithms with high computational cost and fast heuristic alternatives that trade accuracy for efficiency. To enable interactive analysis, MonoTune further provides an incremental algorithm that produces a sequence of progressively improving repairs, starting from a heuristic solution and converging to an optimal one. We demonstrate MonoTune on several real-world datasets through a collection of scenarios illustrating how the system helps analysts explore expected trends, quantify deviations in both directions, compare the effectiveness and efficiency of different repair algorithms, and gain insights into the nature of the proposed repairs.

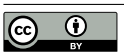
CCS Concepts

• **Information systems** → **Data management systems**; • **Human-centered computing** → *Visual analytics.*

Keywords

Database repair; aggregate order constraints; trend analysis

ACM Reference Format:
Shunit Agmon, Itai Manor, Brit Youngmann, Benny Kimelfeld, and Amir Gilad. 2026. MonoTune: Analyzing Trend Deviations through Database Repair. In *Companion of the International Conference on Management of Data (SIGMOD Companion '26), May 31–June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3788853.3801602



SIGMOD Companion '26, Bengaluru, India

ACM ISBN 979-8-4007-2450-3/2026/05
https://doi.org/10.1145/3788853.3801602

1 Introduction

Real-world datasets often deviate from expected monotonic trends. Typical examples include the assumption that median salary increases with education level, that newer homes are more expensive on average, or that the prevalence of diabetes rises with age. Detecting and analyzing monotonic trends is a well-studied problem in statistics, with applications spanning economics, medicine, and the social sciences. In recent work [1], we introduced *Aggregate Order Dependencies* (AODs) as a formal mechanism for specifying and reasoning about monotonic trends in the result of an aggregate query. An AOD captures the expectation that, for a given grouping attribute, the result of an aggregate function over another attribute should follow a monotonic order (non-decreasing or non-increasing) with respect to the grouping attribute. When a dataset violates an expected AOD, the deviation may be explained, in a *contrastive* manner, by a small subset of input tuples whose removal restores the expected trend. Identifying such subsets corresponds to computing a *cardinality-based repair* for the AOD and applying the "minimal repair" measure of inconsistency [2, 5, 6].

Yet, computing cardinality-based repairs is computationally challenging, similarly to repairs for classical integrity constraints such as functional dependencies [3, 4], as the number of possible repairs may be exponential in the input size. We therefore introduced several algorithms for this task, ranging from polynomial-time but costly optimal solutions to substantially faster heuristic methods that produce approximate repairs [1].

Building on the AOD framework, this work presents MonoTune, an interactive system for detecting, quantifying, and explaining violations of monotonic trends in user-provided datasets. While our prior work [1] focused on the formalization of AODs and the theoretical properties of the underlying repair problems and algorithms, MonoTune operationalizes these ideas in a user-facing system that supports exploratory analysis and explanation. The system allows users to define expected trends using AODs and to examine the nature of these violations. MonoTune visualizes violations, computes cardinality-based repairs, and summarizes the impact of each repair to aid interpretation. Recognizing the trade-off between accuracy and performance, the system integrates both optimal repair algorithms and substantially faster heuristic alternatives. To further support interactive analysis, MonoTune introduces a novel incremental repair procedure that produces a sequence of increasingly refined repairs, enabling users to progressively explore the space between fast approximations and optimal solutions.

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

Figure 1: Example relation r (left) and statistics grouped by the education level (edu).

We demonstrate MonoTune via the Zillow, Diabetes, H&M, and Stack Overflow datasets. Each of the datasets allows us to investigate an expected, yet violated, trend phrased as an AOD. For example, we may expect that higher degrees lead to higher salaries (Stack Overflow) or that newer houses should be priced higher than old ones (Zillow). Realizing that these expectations are not met, participants will run the repair algorithms that remove tuples to restore the AOD. The repairs are visibly shown in bar charts and precisely quantified in the UI. They will also be provided with an explanation summarizing the characteristics of the removed tuples during the repair process, providing further insight into the repair.

2 Aggregate Order Dependencies

Consider a group-by aggregation query over a single relation r:

$$\text{SELECT } \alpha(A) \text{ FROM } r \text{ GROUP BY } G;$$

The *aggregate function* α maps a bag V of numbers to a single value $\alpha(V)$. We use common aggregate functions: avg, sum, max, and median. We refer to A as the *aggregate* attribute, and to G as the *grouping attribute*, and assume an order over the G values. We denote $r[G] = \{g_1, \ldots, g_m\}$, 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 $r_1 \cup \cdots \cup r_i$.

For example, consider the relation r on the left of Figure 1. The relation r_2 consists of the tuples of Chloe, Daniel, Emily, Faith, and Gavin, and $r_{\leq 2}$ includes Ashley and Brandon in addition to the five.

An *aggregate order dependency* (AOD) is an expression of the form $G \nearrow \alpha(A)$, stating that, as the grouping attribute increases, the corresponding aggregate values should not decrease. More formally, if $g_i < g_j$ are two values of G, then $\alpha(r_i[A]) \leq \alpha(r_j[A])$. We describe the methods in the context of *non-decreasing* trends, but they are applicable to non-increasing trends as well.

For illustration, consider again the relation r in Figure 1. The tables on the right give the aggregate values $\alpha(r_i[\text{income}])$ 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. Yet, r violates the AOD edu $\nearrow$ avg(income) since the average salary for education level 2 ($\alpha(r_2[\text{income}])$ is higher than the average salary for education level 3 ($\alpha(r_3[\text{income}])$.

Given an AOD over a dataset, MonoTune finds a repair, that is, a subset of the data that satisfies the AOD. Specifically, it searches for a cardinality-based repair (C-repair), which removes the fewest possible tuples. The corresponding optimization problem and its computational complexity were studied in [1].

3 System Overview

We provide an overview of the implementation of MonoTune [1]. Full details and algorithms can be found in [1]. We implemented MonoTune using Python, with Streamlit for the frontend [2].

We developed a fast heuristic solution that is used by the system to accelerate the computation of an optimal solution, based on dynamic programming. In addition, we incorporated both aggregation-function-specific and general-purpose optimizations. The system workflow is as follows: the heuristic algorithm is first executed, and its output is provided as input to the DP algorithm, where the relevant optimizations are triggered. The resulting solution is then presented to the user along with supporting information, including intermediate repairs, the number of tuples removed in each group by each solution, runtimes of each step, and data summaries providing a deep dive into the removed tuples.

3.1 Computing Repairs

MonoTune builds on the algorithms developed in [1], which include an optimal repair algorithm (CardRepair) and a heuristic algorithm (HeurRepair). We present the high-level ideas here next.

Since HeurRepair is usually faster than CardRepair, we further implement a series of intermediate combined repairs, which improve over time, to enable an interactive experience. This feature was developed uniquely for the demo.

Heuristic repair. The heuristic algorithm iteratively removes tuples until the AOD is satisfied. In each iteration, it selects a tuple with the largest impact on the trend violation, and removes it.

The algorithm first computes the initial sum of trend violations. The violation, for each pair of consecutive groups, is defined as $max(0, \alpha(r_i[A]) - \alpha(r_{i+1}[A]))$. As long as the sum of violations ($S_{\text{MVI}}(r)$) remains positive, the algorithm calculates the impact of each tuple: $S_{\text{MVI}}(r) - S_{\text{MVI}}(r \setminus \{t\})$, and removes the tuple with the maximum impact. The dataset is then updated accordingly, and the measures of violations recomputed. Finally, the algorithm returns the modified relation r', which satisfies the AOD.

Optimal repair. This algorithm (CardRepair) uses dynamic programming for computing a C-repair. The algorithm is based on decomposing the global problem of finding a subset of the database that satisfies the trend into local, group-level problems, of finding a subset of the group that yields a specific aggregation value.

The algorithm requires two specific procedures that depend on the aggregation function α. The realization of these procedures for specific aggregation functions, as well as practical optimizations, are fully described in [1]. The procedures are as follows. (1) The first 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 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

[1] System demonstration video: https://youtu.be/mgFxGjMqhus.
[2] The implementation is available at https://github.com/shunita/aod/tree/demo-final.

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 $O_\alpha(r, x) = \emptyset$.

As an example, consider again the relation r of Figure 1, 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.

The dynamic programming table has the following semantics: for $i \in \{1, \ldots, m\}$ and $x \in V_\alpha(r)$, $H_i[x]$ holds the largest subset r' of $r_{\leq i}$ so that r' satisfies the AOD and, in addition, the value of the rightmost bar is x. For increasing i values and for $x \in V_\alpha(r_i)$, the algorithm considers two cases. 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). Otherwise, if $O_\alpha(r_i, x)$ is nonempty, then $H_i[x]$ is the union of $O_\alpha(r_i, x)$ and the maximum-size $H_{i-1}[y]$ over all $y \leq x$. The C-repair is given by $H_m[\max(V_\alpha(r))]$.

3.1.1 Intermediate repairs. CardRepair's quality is optimal, but its computation is time consuming. HeurRepair is faster in many cases, but has no optimality guarantees. MonoTune includes a unique feature, developed for this system, and designed to bridge the gap between the two solutions. We compute a series of increasing quality repairs, composed of combinations of HeurRepair and CardRepair, that can be presented to the user along the computation process.

CardRepair computes $H_i[x]$ for each possible x, in increasing order of i. Given a heuristic repair $r' \subseteq r$, and a computation of $H_j[x]$ for all $j \leq i$, we define an intermediate repair, composed of a fixed suffix of the heuristic repair (for groups $i+1$ to m) along with the optimal solution for groups 1 to i (given the fixed suffix). Formally, the intermediate repair is defined as $H_i[x] \cup (\cup_{i+1 \leq j \leq m} r'_j)$ where x is the minimal aggregation value in the suffix $r'_{\geq i+1}$.

Note that the quality of the intermediate repairs improves per iteration: the i-th repair is guaranteed to remove no more than the previous ones, and no more than the heuristic repair (as it is optimal for groups 1 to i). Additionally, the i-th repair may yield different aggregation values for groups 1 to $i-1$, as more groups are included in the optimal solution, and larger subsets may be found.

3.2 UI Overview

The user of MonoTune is first asked to upload a data file, or select a preloaded dataset. Then, the UI (Figure 2) consists of two key elements: a control sidebar on the left (①), and a result graph in the center. On the control sidebar, the user specifies the parameters defining the AOD: a grouping attribute, an aggregate attribute, an aggregation function, and a trend direction. For example, for the Diabetes dataset shown in Figure 2, we can select age as the grouping attribute, diabetes as the aggregate attribute, and avg as the aggregation function, which defines an AOD meaning that the prevalence of diabetes should increase with age. A textual description of the trend is shown at the top (②). Then the user can execute the original query (③). A bar chart (④) representing the query result is rendered on the right, along with arrows emphasizing the trend violations. The user can then choose to generate repairs for the trend: a heuristic repair (⑤) and an optimal repair (⑥). Since computing the optimal repair can be time consuming, intermediate repairs are shown as the computation progresses. Only the most recent intermediate repair (or the optimal repair) is shown; the previous steps can be added by clicking "show additional steps" below the bar chart (⑦) and selecting the specific steps.

When hovering over a column in the chart (⑧), repair statistics are shown: the aggregation value $\alpha(r_i[A])$, the number of tuples removed from r_i in the repair, and the number of remaining tuples.

Under the chart, a table specifies the runtime of each repair and the number of removed tuples. Below each row in the table, which corresponds to a repair, an explanation button (⑩) is available. The user can click it to get a summary of differences between the set of removed tuples and the set of remaining tuples. The summary contains a list of the dataset attributes where the difference in distributions (measured by KL-divergence) between the two subsets is largest. Under each attribute, we display the top values in each attribute that contributed to the difference in distributions.

4 Demonstration

We demonstrate MonoTune over several datasets, including Stack Overflow, Zillow, Diabetes (Kaggle), and H&M (RelBench [7]).

Navigating MonoTune. We first demonstrate basic analyses of intuitive trends across several datasets, each corresponding to a preloaded example. We will specify the expected trend and generate repairs (a heuristic repair and a C-repair) with the goal of understanding the trade-offs between them.

In the Zillow dataset, we expect that *the median price of the properties should increase with their building year*. We use 2-year bins for the building year and \$1K quantization for the listing price. Using MonoTune, the participants would see a general increasing trend except for a decrease in the median price between 2001 and 2007. The participants will then see the contribution of each solution: A heuristic repair (computed in 13.9 seconds) entirely removes listings built in those years (36.7% of the data), while the C-repair only removes 5.6% of the data, keeping a representation of each category. The C-repair's fast runtime in this case (6.7 seconds) was enabled by a pruning optimization that leverages the heuristic bound.

In Stack Overflow, we consider entries from USA (8,682 tuples), and the expected trend assumes that *the median salary increases with the education level*. Here again, the participants will see that, while this trend generally holds, there are some deviations: the median salary for people with a high school diploma (\$130K) or with some college study and no degree (\$145K) is higher than that of Associate (\$125K) and Bachelor's (\$140K) degree holders. Here as well, participants will be presented with two alternative repairs: A heuristic repair (computed in 1.9 seconds) requires removing 496 tuples (5.71%), while the C-repair (computed in 1.15 seconds, again, thanks to the pruning optimization) only removes 77 tuples (0.89%).

For H&M (10K out of 15.2M tuples sampled), we focus on the following alleged trend: *for customers aged 25-40, the total amount spent per age group generally decreases as age increases*. The audience will observe that, while it largely holds, there are notable deviations in the age ranges 25-26, 29-30, and 38-40. The participants will observe that both C-repair and the faster heuristic algorithm require removing only 37 tuples, corresponding to 0.37% of the data.

Trend Direction Analysis. Next, we question the initial trend selection by exploring the reverse trends to determine which direction

Figure 2: Screenshot of MonoTune.

requires less intervention (fewer tuples removed). This scenario highlights MonoTune's usability in helping analysts examine hypotheses and quantify the amount of intervention needed to enforce an expected trend compared to its opposite.

In the Zillow example, the participants would learn that the C-repair for the opposite trend requires removing only 389 (5.3%) listings (slightly less than the deviation from the original trend), suggesting that the decrease in median listing value in 2001-2007 is indicative of a real phenomenon. In the Stack Overflow example, the audience would discover that a C-repair for the opposite trend (median salary decreases with education) requires removing 564 tuples (6.49%), meaning that the data is closer to exhibiting an increasing trend. In the H&M example, the participants would see that enforcing the reversed trend (total sales increasing with age) requires removing 3,840 tuples (38.4%), indicating that the data more strongly supports the trend of total sales decreasing with age.

Analyzing the Deviation. Lastly, participants will use MonoTune to deep-dive into the reasons for deviation from the expected trend, by examining the removed tuples. By clicking "Explanation: Distribution Differences," MonoTune displays a summary of the differences in data characteristics of the tuples removed by the C-repair, versus the remaining tuples. For example, in the Stack Overflow dataset, analyzing the removed tuples shows differences in developer type, with a higher frequency of frontend and full-stack developers in the removed subset (66% vs. 53%), and more experienced developers: more people between 55 and 64 years old (12% vs. 6%). In the H&M dataset, removed tuples included, for example, more items from "Women's Trend" category (27% vs. 1%) and more "Outdoor Garments" (11% vs. 1%), which are usually more expensive.

Acknowledgments

The work of Gilad and Manor was funded by the Israel Science Foundation (ISF) grant 1702/24, the Scharf-Ullman Endowment, and the Alon Scholarship. The work of Youngmann was funded by the US-Israel Binational Science Foundation under grant 2024101, and by the ISF grant 934/25. The work of Agmon and Kimelfeld was funded by the ISF grant 863/25. We thank Elazar Gershuni for insightful discussions and assistance with the implementation.

References

[1] 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] to appear in SIGMOD 2026.
[2] Leopoldo E. Bertossi. 2019. Repair-Based Degrees of Database Inconsistency. In *LPNMR (Lecture Notes in Computer Science, Vol. 11481)*. Springer, 195–209.
[3] Wenfei Fan and Floris Geerts. 2012. *Foundations of Data Quality Management*. Morgan & Claypool Publishers.
[4] Ester Livshits, Benny Kimelfeld, and Sudeepa Roy. 2020. Computing Optimal Repairs for Functional Dependencies. *ACM TODS* 45, 1 (2020), 4:1–4:46.
[5] 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.
[6] 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
[7] Joshua Robinson, Rishabh Ranjan, Weihua Hu, Kexin Huang, Jiaqi Han, Alejandro Dobles, Matthias Fey, Jan Eric Lenssen, Yiwen Yuan, Zecheng Zhang, et al. 2024. Relbench: A benchmark for deep learning on relational databases. *Advances in Neural Information Processing Systems* 37 (2024), 21330–21341.