Efficient Query Repair for Arithmetic Expressions with Aggregation Constraints

Shatha Algarni*
University of Southampton
University of Jeddah
s.s.algarni@soton.ac.uk

Boris Glavic
University of Illinois, Chicago
bglavic@uic.edu

Seokki Lee
University of Cincinnati
lee5sk@ucmail.uc.edu

Adriane Chapman
University of Southampton
adriane.chapman@soton.ac.uk

Abstract

In many real-world scenarios, query results must satisfy domain-specific constraints, such as fairness or financial stability. For instance, a query may need to ensure that a minimum percentage of selected interview candidates are female based on their qualifications, or that a company's average purchase cost remains below its financial liability to suppliers. These requirements can be expressed as constraints on the result of an arithmetic combination of aggregates evaluated on the result of the query. In this work, we study how to repair a query to fulfill such constraints by modifying the filter predicates of the query. We prove the hardness of this problem and introduce a novel query repair technique that leverages bounds on candidate solutions and interval arithmetic to efficiently prune the search space. We demonstrate experimentally, that our technique significantly outperforms baselines that consider a single candidate at a time.

PVLDB Reference Format:
Shatha Algarni, Boris Glavic, Seokki Lee, and Adriane Chapman. Efficient Query Repair for Arithmetic Expressions with Aggregation Constraints. PVLDB, 14(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ShathaSaad/Query-Refinement-of-Complex-Constraints.

1 Introduction

Data is a commodity that allows organizations to build models and perform analysis. Analysts are typically well versed in writing queries that return data based on obvious conditions, e.g., only return applicants with a master's degree. However, query results often have to fulfill additional constraints such as fairness that do not naturally translate into conditions. While for some applications it is possible to filter the results of the query to fulfill such constraints this is not always viable, e.g., because the same selection criterion has to be used for all applicants for a job, more often the user requires the query to be repaired such that the fixed query satisfies all constraints. Previous work in this area, including query-based explanations [9, 26] and repairs [6] for missing answers, answering why-not questions [4, 9] as well as query refinement / relaxation approaches [19, 22, 27] look at why specific tuples are not in the result set or how to fix a query to return such tuples. In this work, we study a more general problem where the entire result set of the query has to fulfill some constraint. The constraints we study in this work are expressive enough to guarantee that query results adhere to legal and ethical regulations, such as fairness. Typically, it is challenging to express such constraints as the conditions of a query. Consider the following example:

Example 1 (Motivating Example). *A data marketplace allows a buyer to issue a query across data from multiple sellers. The buyer wishes to obtain UK standardized test scores from 2020 to analyze North-South trends. Unfortunately, due to Covid, not all schools gave exams. The buyer is willing to relax their query in order to get exams from* $> 80\%$ *of all schools but wants an even spread of schools between North-South to ensure that the results are representative.*

In this work, we model constraints on the entire query result as arithmetic expressions involving aggregate queries evaluated over the output of a *user query*. When the result of the user query fails to adhere to such an *aggregate constraint*, we would like the system to fix the violation by *repairing* the query by adjusting selection conditions of the query, similar to [17, 22]. Specifically, we are interested in computing the top-k repairs with respect to their distance to the user query. The rationale is that we would like to preserve the original semantics of the user's query as much as possible. Unlike previous work [19], we consider constraints that can be non-monotone which invalidates most of the optimizations considered in prior work.

A brute force approach for finding a solution to the query repair problem is to enumerate all possible candidate repairs. A candidate repair can be encoded as a combination of constant c_1 to c_n, one for each condition in the form of a_i op c_i of the user query where a_i is an attribute. The candidate repairs are then sorted based on their distance to the user query and then each candidate is evaluated by running the modified query and checking whether the candidate fulfills the aggregate constraint. The algorithm tests candidates until k repairs have been found that fulfill the aggregate constraint. The main problem with this approach is that the number of repair candidates is exponential in the number of predicates in the user query and for each repair candidate we have to evaluate the modified user query and one or more aggregate queries on top of the user query result. Given that the repair problem is NP-hard we cannot hope to avoid this cost in all cases.

Nonetheless, we identify two opportunities for optimizing this process: (i) when repair candidates are similar (in terms of the constants they use in selection conditions), then typically there will be large overlap between the aggregate constraint computations for

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

the two candidates and (ii) we can compute bounds on the aggregate constraint result for a set of similar candidate repairs at once using interval arithmetic [10], a technique for over-approximating the outputs of a computation over a set of values efficiently. If none or all values within the computed bounds fulfill the aggregate constraint, then we have shown for a set of candidate repairs that none of them is a repair or all of them is a repair.

Figure 1 shows an overview of the entire approach. To exploit (i), we use a k-d tree [5] to partition the input dataset and materialize for each cluster (node in the k-d tree) as shown in Figure 1(b). The result of evaluating the aggregation functions needed for a constraint on the set of tuples contained in the cluster as well as store bounds for the values of attributes within the cluster (as is done in zonemaps / small materialized aggregates [24, 31]). Then to calculate the result of an aggregation function for a repair candidate, we use the bounds for each cluster to determine whether all tuples in the cluster fulfill the selection conditions of the repair candidate (in this case the materialized aggregates for the cluster will be added to the result), none of the tuples in the cluster fulfill the condition (in this case the whole cluster will be skipped), or if some of the tuples in the cluster fulfill the condition (in this case we apply the same test to the children of the cluster in the k-d tree). See Figure 1(c)-(e). We refer to this algorithm as full cluster filtering (FF). The main advantage of this algorithm over the brute force approach is that it can reuse the aggregate query results materialized for a cluster if all tuples in cluster fulfill the conditions of the repair candidate and can skip any further computation for clusters that do not contain any tuples fulfilling the conditions. For instance, consider a cluster where the values of all tuples in the cluster in some attribute a are bounded by $[20, 30]$. For a repair candidate with a condition $a > 50$ the whole cluster can be skipped as no tuple in the cluster can fulfill the condition as shown in Figure 1(c). In contrast, all tuples in a cluster with a values bounded by $[60, 80]$ are guaranteed to fulfill the repair candidate's condition $a > 50$ and we can utilize the materialized aggregation results for the cluster to compute the aggregation result for the repair candidate. For instance, consider a cluster where the values of all tuples in the cluster in some attribute a are bounded by $[20, 30]$. For a repair candidate with a condition $a > 50$ the whole cluster can be skipped as no tuple in the cluster can fulfill the condition as shown in Figure 1(c). In contrast, all tuples in a cluster with a values bounded by $[60, 80]$ are guaranteed to fulfill the repair candidate's condition $a > 50$ and we can utilize the materialized aggregation results for the cluster to compute the aggregation result for the repair candidate.

We then extend this idea to bound the aggregation constraint result for sets of repair candidates to exploit observation (ii). In this case we represent set of repair candidate through intervals of possible values for each c_i for which the user query contains a predicate a_i op c_i, e.g., $a \in [50, 55]$. Then we can again reason about whether all / none of the tuples in a cluster will fulfill the condition for *every* repair candidate whose c_i value is within the bounds (e.g., $[50, 55]$ in our example). The result will be valid bounds on the aggregation constraint result for any candidate repair with constants in the bound. We then exploit such bounds to determine that all repair candidate within such bounds are repairs or none of them are. If the result is inconclusive, we can then partition such a set of repair candidates into multiple smaller sets and apply the same approach to these sets. We refer to this algorithm as cluster range pruning (RP). The advantage of this approach is that it often enables us to prune sets of repair candidates or confirm all of them to be repairs without individually evaluating them.

Contributions of this work include:

- A formal definition of query repair under aggregate constraints in Section 3, including a proof of hardness.
- In addition to the brute force method, we present two algorithms, FF in Section 4 and RP in Section 5, that solve the aggregate constraint repair problem.
- A full experimental evaluation over multiple datasets, queries and constraints in Section 6. We compare against the current state of the art [19], and show that we can cover more complex constraints. We investigate the impact of dataset size, parameter k, clustering parameters and fraction of the search space that needs to be explored on algorithm performance.

We ground the explanations and experimentation in two real-world examples described in Section 2.

2 Example Use Cases

2.1 Fairness Constraint

Consider a job applicant dataset D for a tech-company that contains six attributes: `ID, Gender, Field, GPA, TestScore,` and `OfferInterview`. The attribute `OfferInterview` was generated by an external AI model suggesting which candidates should receive an interview. The employer uses the query shown below to prescreen candidates: every candidate should be a CS graduate and should have a high GPA and test score.

```
Q1: SELECT * FROM D WHERE Major = 'CS'
          AND TestScore ≥ 30 AND GPA ≥ 3.80
```

The Aggregate Constraint. The employer would like to ensure that their decision to interview a candidate is not biased against a specific gender. One way to measure such a bias is to measure the statistical parity difference (SPD) [3, 20] between demographic groups. Given a set of data points that belong to one of two groups (e.g., male and female) and a binary outcome attribute Y where $Y = 1$ is assumed to be a positive outcome (`OfferInterview=1` in our case), the SPD is the difference between the probability for individuals from the two groups to receive a positive outcome. In our example, the SPD can be computed as shown below (G is `Gender` and Y is `OfferInterview`).

$$\text{SPD} = \frac{\textbf{count}(G = M \wedge Y = 1)}{\textbf{count}(G = M)} - \frac{\textbf{count}(G = F \wedge Y = 1)}{\textbf{count}(G = F)}$$

The employer would like to ensure that SPD is maximally 0.2. The model generating the `OfferInterview` attribute is trusted by the company, but is provided by an external service and, thus, cannot be fine-tuned to improve fairness. However, the employer is willing to change their prescreening criteria by expressing their fairness requirement as an aggregate constraint SPD ≤ 0.2 as long as the same criteria are applied to judge every applicant to ensure individual fairness. That is, the employer desires a repair of the query whose selection conditions are used to filter applicants. Prior work on ensuring fairness by repairing queries [19] only considers

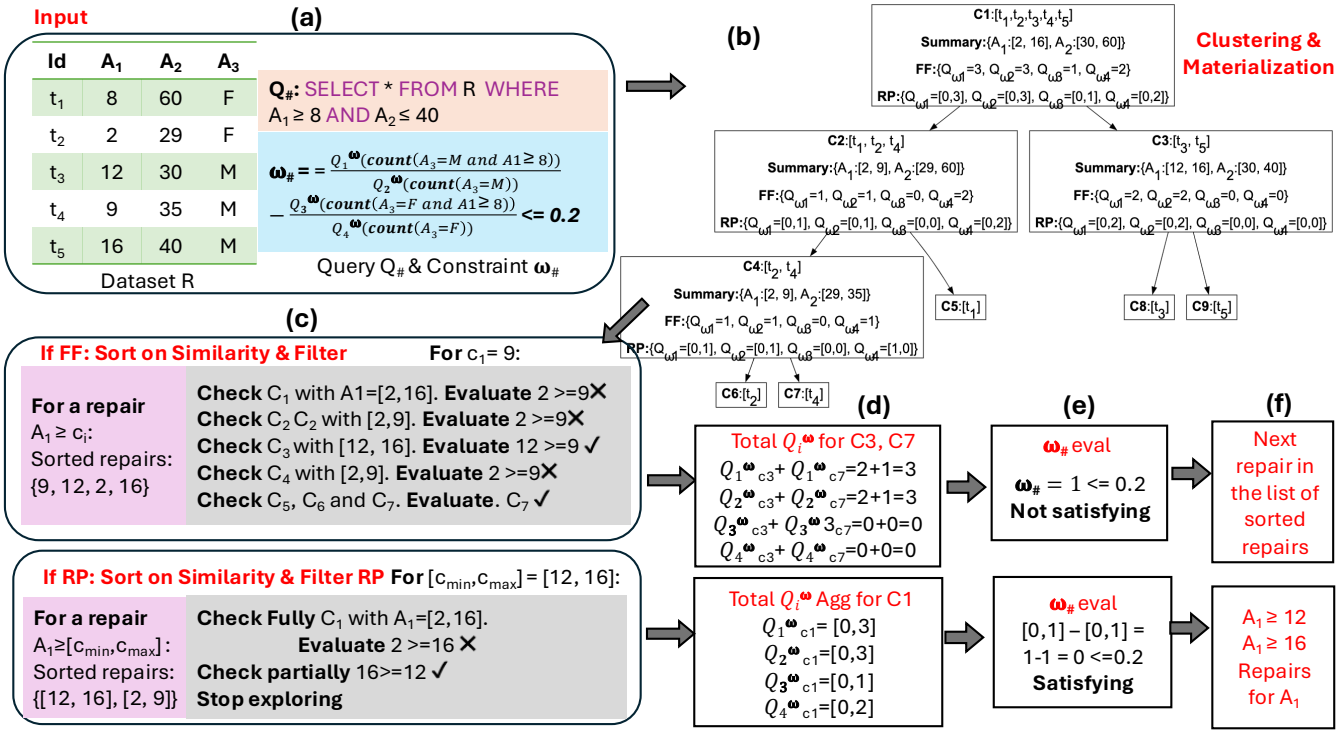


Figure 1: Overview of Query Repair with aggregate constraint approach, with example data.

cardinality constraints for a single group in the query result which cannot express SPD.

2.2 Company Product Management

A retail company aims to monitor product performance by retrieving information about the parts of type "Large Brushed" with a size greater than 10 that are supplied by suppliers located in Europe. The company uses the following query to retrieve this information:

```
Q2: SELECT *
    FROM part, supplier, partsupp, nation, region
    WHERE p_partkey = ps_partkey AND
        s_suppkey = ps_suppkey AND p_size >= 10
        AND s_nationkey = n_nationkey
        AND n_regionkey = r_regionkey
        AND p_type = 'LARGE␣BRUSHED'
        AND r_name = 'EUROPE'
```

The company, however, has set an aggregate constraint to manage its product selection and ensure effective inventory planning. They want to avoid overstocking under-performing products that contribute less to revenue. The constraint requires that products from Ukraine only contribute 10% to 30% of the total revenue of the result set in order to minimize supply chain disruptions. Formally, the constraint is defined as follows:

$$0.1 \leq \frac{\sum \text{Revenue}_{\text{ProductsSelectedFromUkraine}}}{\sum \text{Revenue}_{\text{Selected Products}}} \leq 0.3$$

Prior work on query repair [2] only supports constraints on a single aggregation result while the constraint shown above is an arithmetic combination of aggregation results as supported in our framework.

3 Problem Definition

In this work, we consider a dataset $D = R_1, \cdots, R_n$ consisting of one or more relations R_i. Let $A = a_1, \ldots, a_m$ represent the set of attributes in D. In this section, we formalize the query repair problem studied in this work.

User Query. We consider the class of select-project-join (SPJ) as a user query Q, i.e., relational algebra expressions of the form:

$$\pi_A(\sigma_\theta(R_1 \bowtie \ldots \bowtie R_n))$$

We assume that the selection predicate θ of such a query is a conjunction of comparisons of the form $a_i \, \text{op} \, c_i$. For numerical attributes a_i, we allow $\text{op} \in \{<, >, \leq, \geq\}$ and for categorical attributes a_i we only allow $\text{op} \in \{=, \neq\}$. We use $Q(D)$ to denote the result of evaluating Q over D.

Aggregate Constraints. The user specifies requirements on the result of their query as aggregate constraint (AC). An AC is a comparison between a threshold and an arithmetic expression over the result of filter-aggregation queries. Such queries are of the form $\gamma_{f(a)}(\sigma_\theta(Q(D)))$ where f is an aggregate function – one of **count**, **sum**, **min**, **max**, **avg** – and θ is a selection condition. We use Q^ω to denote such a filter-aggregation query and use $\textbf{count}(\theta)$ as a shorthand for $\gamma_{\textbf{count}(*)}(\sigma_\theta(Q(D)))$. Note that we already made

use of this notation in Section 2. These queries are evaluated over the user query's result $Q(D)$. An aggregate constraint ω is of the form:

$$\omega := \tau \text{ op } \Phi(Q_1^{\omega}, \ldots, Q_n^{\omega}).$$

Here, Φ is an arithmetic expression using operators $(+, -, *, /)$ over $\{Q_i^{\omega}\}$, op is a comparison operator, and τ is a threshold.

An aggregate function f is monotonically increasing (decreasing), if $f(S_1) \leq f(S_2)$ when $S_1 \subseteq S_2$ (if $f(S_1) \geq f(S_2)$ when $S_1 \supseteq S_2$) for any two bags of values S_1 and S_2. Many query refinement and relaxation techniques [19] exploit monotonicity to optimize search as relaxing (refining) a query Q's selection conditions is bound to increase (decrease) $f(Q(D))$ if f is monotone, e.g., by pruning unpromising candidates to find a refined query without enumerating all candidates in the search space [7, 19, 29].

The aggregate constraints we use in this work are not monotone in general. However, under certain restrictions such constraints are monotone and some of the optimizations proposed in past work are applicable. Consider a constraint $\omega := \tau \text{ op } \Phi$. The arithmetic expression Φ may be non-monotone if one of the following conditions holds:

(1) It uses a non-monotone arithmetic operator like division or subtraction.
(2) Is uses a non-monotone aggregation function, e.g., **sum** over the integers $\mathbb{Z}$.
(3) It uses both monotonically increasing and monotonically decreasing aggregation functions, e.g., **min+count** or **max+min**.

Query Repair. Given a user query Q, database D, and aggregate constraint ω that is violated on $Q(D)$, we want to generate a repaired version Q_{fix} of Q such that $Q_{fix}(D)$ fulfills ω. In this work, we restrict repairs to changes of the selection condition θ of Q. Recall that Q is an SPJ query with a conjunctive selection condition. That is, the user query condition is of the form: $\theta = \theta_1 \wedge \ldots \wedge \theta_n$ where each θ_i is a comparison of the form $a_i \text{ op } c_i$. A *repair candidate* is a query Q_{fix} that differs from Q only in the constants used in selection conditions, i.e., Q_{fix} uses a condition: $\theta' = \theta_1' \wedge \ldots \wedge \theta_n'$ where θ_i' is a condition $a_i \text{ op } c_i'$. A repair candidate is called a *repair* if $Q_{fix}(D) \models \omega$.

Repair Distance. Ideally, we would want to achieve a repair that minimizes the changes to the user's query. Additionally, we may be interested in minimizing changes to the result returned by the user's query. In fact, many different optimization criteria are reasonable and which criteria is the most important will depend on the application. In this paper, we focus on minimizing changes to the user's query. For that, we define a distance metric between repair candidates based on their selection conditions. Consider the user query Q with selection condition $\theta_1 \wedge \ldots \wedge \theta_n$ and repair Q_{fix} with selection condition $\theta_1' \wedge \ldots \wedge \theta_n'$. Then the distance $d(Q, Q_{fix})$ is defined as:

$$d(Q, Q_{fix}) = \sum_{i=1}^{n} d(\theta_i, \theta_i')$$

where the distance between two predicates $\theta_i = a_i \text{ op } c_i$ and $\theta_i' = a_i \text{ op } c_i'$ for numeric attributes a_i is: $\frac{|c_i' - c_i|}{|c_i|}$. For categorical attributes, the distance is 1 if $c_i \neq c_i'$ and 0 otherwise.

Example 2. *For the use case in Section 2.1, the repair candidate* `Major = 'EE', Testscore ≥ 33, and GPA ≥ 3.9` *is more similar to the user query than the candidate* `Major = 'EE', Testscore ≥ 37, and GPA ≥ 3.85` *based on our distance metric. For the first candidate, the distance is* $1 + \frac{33-30}{30} + \frac{3.9-3.8}{3.8} = 1.13$ *while for the second candidate it is* 1.24.

We are now ready to formulate the problem studied in this work, computing the k repairs with the smallest distance to the user query.

Aggregate constraint repair problem:

- **Input**: user query Q, database D, constraint ω, threshold k
- **Output**: $\text{topk}_{Q_{fix} \textbf{ is a repair}}\, d(Q, Q_{fix})$

We will focus here on non-monotone constraints as they are more challenging. In a practical solution, we may detect if a constraint is monotone and apply existing optimizations [19] for monotone constraints.

Search Space. To generate a repair Q_{fix} of Q, we must explore the search space of possible candidate repairs. Consider a query with a conjunction of conditions of the form $a_i \text{ op}_i c_i$ for $i \in [1, m]$. Let n_i denote the number of values in the active domain of a_i. Each candidate repair corresponds to choosing constants $[c_1', \ldots, c_m']$. In fact, we will often use $\vec{c} = [c_1', \ldots, c_m']$ to denote a repair and use $\mathbb{Q}_{fix}$ to denote the set of all candidate repairs. The exact number of candidate repairs depends on which comparison operators are used, e.g., for $\leq$ there are at most $n_i + 2$ possible values that lead to a different result in terms of which of the input tuples will fulfill the condition. In general, the size of the search space is $O(\prod_{i=1}^{m} n_i)$, exponential in m, the number of conditions in the user query. Unsurprisingly, the aggregate constraint repair problem is NP-hard in the schema size.

Seokki says: Should we explain why the "+2" possible values are needed?

Theorem 3.1. *Given a dataset D, an SPJ query Q, an aggregate constraint, the aggregate constraint repair problem is NP-hard.*

Proof. It has been established in [8] that the problem of repairing a query Q by selecting constants in selection condition such that the repaired query Q_{fix} satisfies a single cardinality constraint on $Q(D)$ is NP-hard in combined complexity. As ACs can express cardinality constraints, our problem is also NP-hard. □

4 The Full cluster filtering Algorithm

We now present the full cluster filtering (FF) algorithm for the aggregate constraint repair problem that materializes aggregation results for subsets of the input database D and combines these aggregation results to compute the aggregation functions to be able to compute the aggregate constraint ω for a repair candidate Q_{fix}. Figure 1 shows the overview of steps, following the FF path, including (a) input, (b) building cluster trees and materializing statistics, (c) Searching for candidate repairs, and (d)-(e) evaluating constraints.

4.1 Clustering and Materializing Statistics

We use k-d trees to partition each table in the database into subsets (*clusters*) based on attributes that appear in the selection condition (θ) of the user query. The rationale for that is that the selection conditions of a repair candidate filter data along these attributes. For ease of presentation, we will consider a database consisting

of a single table R. To evaluate the aggregation constraint ω for a candidate $Q_{fix} = [c_1, \ldots, c_m]$, find a set of clusters (nodes in the k-d tree) that cover exactly the subset of D that fulfills the selection condition of the candidate. We can then merge the materialized aggregation results for these clusters to compute the aggregation functions for $Q_{fix}(D)$. To be able to do that, we record the following information for each cluster $C \subseteq D$ that can be computed by a single scan over the tuples in the cluster, or by combining results from previously generated clusters if we generate clusters bottom up.

- **Selection attribute bounds:** for each attribute a_i used in the selection condition θ, we store $\text{BOUNDS}_{a_i} := [\min(\pi_{a_i}(C)), \max(\pi_{a_i}(C))]$.
- **Count**: The total number of tuples $\textbf{count}(C) := |C|$ in the cluster.
- **Aggregation results**: For each filter-aggregation query Q_Q^ω in constraint ω, we store $Q^\omega(C)$.

For example, for the running example from Figure 1 the user query filters on attributes A_1 and A_2. The root of the k-d tree represents the full dataset. At each level, the nodes from the previous level are split into a number of buckets (this is a configuration parameter), two in the example, along one of the attributes in θ. For instance, the root cluster C_1 is split into two clusters C_2 and C_3 by partitioning the rows in C_1 based on their values in attribute A_1. For cluster C_2 we have $\text{BOUNDS}_{A_1} = [2, 9]$ as the lowest A_1 value is 2 (from tuple t_2) and the highest value is 9 (tuple t_4). The value of $Q_2^\omega = \textbf{count}(A_3 = M)$ for C_2 is 1 as there is one male in the cluster. Now consider a repair candidate $[10, 40]$. Based on the bounds on $\text{BOUNDS}_{A_1} = [2, 9]$ for cluster C_2 we know that none of the rows can fulfill the condition $A_1 \geq 10 \wedge A_2 \leq 40$ of the candidate. Thus, this cluster and the whole subtree rooted at the cluster can be ignored for computing the AC $\omega_\#$ for the candidate.

4.2 Constraint Evaluation for Candidates

The FF algorithm (Algorithm 1) takes as input the condition θ of a repair candidate, the root node of the k-d tree C_{root} and returns a set of disjoint clusters $\mathcal{C}$ such that the union of these clusters is precisely the subset of the relation R that fulfills θ:

$$\bigcup_{C \in \mathcal{C}} = \sigma_\theta(R) \tag{1}$$

The statistics materialized for This cluster set $\mathcal{C}$ are then used to evaluate the AC for the repair candidate.

Boris says: Please add labels to the lines of the algorithm and add references to the text where appropriate

4.2.1 Determining a Covering Set of Clusters The algorithm maintains a stack *stack* of clusters to be examined that is initialized with the root cluster C_{root}. It then processes one cluster at a time until a set of clusters fulfilling Equation (1) has been determined. For each cluster C we distinguish 3 cases: (i) we can used the bounds on the selection attributes recorded for the cluster to show that all tuples in the cluster fulfill θ, i.e., $\sigma_\theta(C) = C$. In this case, the cluster will be added to $\mathcal{C}$; (ii) based on the bounds we can determine that none of the tuples in the cluster fulfill the condition. Then this cluster can be ignored; (iii) either a non-empty subset of C for fulfills θ or based on the bounds we cannot demonstrate that $\sigma_\theta(C) = \emptyset$ or $\sigma_\theta(C) = C$. In this case, we add the children of C to the stack to be evaluated in future iterations. The algorithm uses the a function $\textbf{eval}_\forall$ shown in Table 1 to determine based on the bounds of the cluster C, the comparison condition θ_i is guaranteed to be true for all $t \in C$. Additionally, it checks whether case (ii) applies by applying $\textbf{eval}_\forall$ to the negation θ_i. Note that to negate a comparison we simply push the negation to the comparison operator, e.g., $\neg(a < c) = (a \geq c)$. As the selection condition of any repair candidate is a conjunction of comparisons $\theta_1 \wedge \ldots \wedge \theta_m$, the cluster if *fully covered* (case (i)) if $\textbf{eval}_\forall$ returns true for all θ_i and *not covered at all* (case (ii)) if $\textbf{eval}_\forall$ return true for at least one comparison $\neg\theta_i$.

Algorithm 1 Fully Filtering to evaluate constraint candidates

Input: K-d tree with root C_{root}, condition $\theta = \theta_1 \wedge \ldots \wedge \theta_m$, relation R.
Output: Set of clusters $\mathcal{C}$ such that $\bigcup_{C \in \mathcal{C}} C = \sigma_\theta(R)$.

```
1:  stack ← [C_root]
2:  C ← ∅                                          ▷ Initialize result set
3:  while stack ≠ ∅ do
4:      C_cur ← pop(stack)
5:      in ← true, notin ← false
6:      for all θ_i = (a_i op_i c_i) ∈ θ do
7:          in ← in ∧ eval_∀(θ_i, BOUNDS_{a_i}(C_cur))
8:          notin ← notin ∨ eval_∀(¬θ_i, BOUNDS_{a_i}(C_cur))
9:      end for
10:     if in then
11:         C ← C ∪ {C_cur}
12:     else if ¬ notin then
13:         for all C ∈ children(C_cur) do
14:             stack ← stack ∪ {C}
15:         end for
16:     end if
17: end while
18: return C
```

4.2.2 Constraint Evaluation After identifying the covering set of clusters $\mathcal{C}$ for a repair candidate, our approach evaluates the AC ω over this set of cluster. Recall that for each cluster C we materialize the result of each filter aggregate query Q_i^ω used in ω. For aggregate functions that are not decomposable like **avg** we apply the standard trick of storing **count** and **sum**.

This evaluation is based on the results of the collected filter-aggregation queries Q_i^ω for the identified clusters. To determine whether the repair $a_i \text{ op } c_i'$ meets the constraint ω, the evaluation process follows these steps:

(1) Extract the filter-aggregation queries Q_i^ω involved in the user-defined aggregate constraint ω. For example, in the *SPD* constraint from Section 2.1, $\textbf{count}(G = M \wedge Y = 1)$ and $\textbf{count}(G = M)$ are filter-aggregation queries.
(2) Identify the threshold τ specified in ω, such as $[0.1, 0.2]$ in Section 2.1.
(3) Retrieve the precomputed results of Q_i^ω for each cluster satisfying the repair condition $a_i \text{ op } c_i'$. For instance, the values of $\textbf{count}(G = M \wedge Y = 1)$ and $\textbf{count}(G = M)$.
(4) Compute the total result of Q_i^ω by summing the values across all satisfying clusters. For example, if two clusters satisfy the repair condition with $\textbf{count}(G = M)$ values of 5 and 7, the total result is $5 + 7 = 12$.
(5) Evaluate the arithmetic operations $(+, -, *, /)$ in ω using the total results of the filter-aggregation queries. For instance, after computing $\textbf{count}(G = M) = 12$, we evaluate the $(-, /)$ operations in Section 2.1.

Op.	Fully Satisfied in FF	Full Range Satisfied in RP	Partial Range Satisfied in RP
$>, \geq$	$min >/\geq c_i'$	$min >/\geq max\ c_i'$	$max >/\geq min\ c_i'$
$<, \leq$	$max </\leq c_i'$	$max </\leq min\ c_i'$	$min </\leq max\ c_i'$
$=$	$min = c_i' \wedge max = c_i'$	$min = min\ c_i' \wedge max = max\ c_i'$	$min \leq max\ c_i' \wedge min\ c_i' \leq max$
$\neq$	$c_i' \notin [min, max]$	$[min, max] \cap [\min c_i', \max c_i'] = \emptyset$	$min \leq max\ c_i' \wedge min\ c_i' \leq max$

Table 1: A cluster is valid iff the fully or partially satisfied condition holds for a given operator.

(6) Compare the computed aggregate constraint result against the threshold τ of ω.
(7) Determine whether the repair condition a_i op $c_i{}'$ satisfies ω.

Adriane says: Enumerate all repairs. Go through them all, see whether they work.

4.3 Searching Candidates and Top-K Repairs

4.3.1 Most Similar Repairs The repair constants $c_1', \ldots, c_m'$ for a condition a_i op $c_i{}'$ are prioritized based on their similarity to the original constant c_i in the user query Q as described in Section 3. Starting with the most similar possible repair constant c_1' and progressing to less similar ones c_m'. This sorting ensures that the filtering process first evaluates the repair candidates that are closest to the original query. This approach not only reduces the computational effort but also guarantees that the solutions returned are the most relevant for the user's query.

Our algorithm allows users to specify k, the number of top-ranked repair conditions they wish to retrieve. The algorithm prioritizes repair conditions based on their similarity to the user's original query, as detailed in Section 3. By employing this similarity-driven traversal strategy, the algorithm ensures that the returned top-k repairs are those most closely with the user's original query Q.

5 Cluster Range Pruning (RP)

FF has the drawback of exhaustively evaluating all individual repair conditions within the search space of candidate repairs for each condition in the user's query. To overcome this limitation, we enhance FF by introducing bounds on the filter-aggregation queries and the aggregate constraint, leveraging interval arithmetic. This allows us to prune sets of repair candidates or identify valid repairs without evaluating each condition individually.

The Range-Based Pruning Query Repair RP technique evaluates each repair condition a_i op $c_i{}'$ by representing repairs as ranges of possible values for each c_i in the each user condition a_i op c_i contained in the query Q, avoiding exhaustive testing of all possible repairs.

5.1 Repair Initialization

The algorithm requires the similar initialization as described in Section 4.1. For the materialized statistics, the difference only in **Count** and **Aggregations**, where we materialized the lower and upper bound for each of them. The lower bound will be zero while the upper bound is the actual value for each aggregation in each cluster. As our repairs in this algorithm are ranges of constants $c_i{}'$, the algorithm prioritized ranges repairs in the same way as Section 4.3.1 based on the minimum actual c_i exits within each range. So, the most similar repair $c_1{}'$ is the range with a constant $c_1{}'$ that is the most relevant to the user's query constant c_1. We will walk through Algorithm 2 and Algorithm 3 as we break down the processes further below.

5.2 Evaluating repairs within a range

In [30], the challenge of computing iceberg cubes under non-antimonotone aggregation constraints is performed by estimating upper and lower bounds for aggregation functions. These estimated bounds narrow the search space, even when the constraints lack monotonicity. Their approach demonstrates how non-monotonic challenges can be managed by creating tight bounds for aggregation functions by using the Interval Arithmetic Expressions in Table 2.

To determine the bounds of each filter-aggregation queries Q_i^ω contained in the aggregate constraint ω defined by the user, the algorithm retrieves the materialized bounds of these aggregations Q_i^ω for each cluster. For each condition repair a_i op $c_i{}'$, a cluster is classified as **full** if all data points meet the repair condition, **partial** if at least one data point satisfies the repair conditions or **not satisfy** if all data points do not satisfy the repair conditions. This filtering step produces a set of clusters that either fully or partially meet the conditions. After that, the algorithm exploits the lower and upper bounds of each aggregation Q_i^ω for the set of satisfying clusters as follows:

- **Upper Bound of** f: The upper bound of the aggregations Q_i^ω over the satisfying clusters (both fully and partially) is computed as the sum of the *max* values materialized for Q_i^ω for each cluster:

$$\overline{E} = \sum_{j=1}^{i} f(\text{cluster}_j) \tag{2}$$

- **Lower Bound of** f: The lower bound of the aggregations Q_i^ω over the satisfying clusters is computed by summing the materialized values of Q_i^ω for clusters that fully satisfy the repair condition a_i op $c_i{}'$, while assigning a value of zero for clusters that partially satisfying. This ensures that only the fully compliant clusters contribute to the lower bound calculation:

$$\underline{E} = \sum_{j=1}^{i} \begin{cases} f(\text{cluster}_j), & \text{if cluster}_j \text{ is fully satisfied} \\ 0, & \text{if cluster}_j \text{ is partially satisfied} \end{cases} \tag{3}$$

We consider $\overline{E}$ as the upper bound and $\underline{E}$ as the lower bound of the aggregation functions over all satisfying clusters.

5.2.1 Bounding Aggregate Constraint We use Interval Arithmetic in Table 2 to compute the lower and upper bounds of the aggregate constraint ω. After determining $\overline{E}$ and $\underline{E}$ for each aggregation Q_i^ω of each repair condition, the aggregate constraint ω is evaluated using the rules of Interval Arithmetic, as in Table 2. In this context,

E_1 and E_2 represent the bounds of two different aggregations Q_1^ω and Q_2^ω.

5.2.2 Evaluating Operators The range operators in our RP technique evaluate possible conditions against the materialized boundaries (*Min* and *Max*) for each cluster. Unlike the FF technique, this approach evaluate ranges (*Min Condition*, *Max Condition*) of each range against cluster boundaries (*Min* and *Max*). It ensures that clusters are categorized as either fully satisfying or partially satisfying a condition. Our technique evaluates different types of comparison operators such as $\geq$, $\leq$, $>$, $<$, == and $\neq$ for two scenarios:

- **Fully Satisfied Clusters**: Ensures that the entire cluster satisfies the given condition's range.
- **Partial Satisfied Clusters**: Ensures that at least part of the cluster satisfies the given condition's range.

Given:

- **Min** and **Max**: The minimum and maximum bounds for a cluster in a particular dimension.
- **Min Condition** and **Max Condition**: Min and Max conditions of the range of a possible condition repair.
- **Operator**: The comparison operator (e.g., $>, \geq, <, \leq, ==, \neq$).

5.3 Searching for repairs with ranges

The algorithm processes ranges as follows:

(1) **Pop the Most Similar Range:** The algorithm dequeues the most similar range from the queue (Lines 3-4 in Algorithm 2).
(2) **Filtering:** The algorithm filters the cluster tree based on this range (Lines 9-10 in Algorithm 2), following the approach detailed in Table 1 where *min* c_i is the minimum constant and *max* c_i is the maximum constant in each range repair. While *min* and *max* refers to the statistics within each cluster.
(3) **Constraint Evaluation:** The algorithm evaluates the constraints over the set of clusters that satisfy a given range, as described in 5.2 (Line 10 in Algorithm 2). Based on the evaluation results, the algorithm proceeds with the subsequent steps:
   - *Satisfying the Constraint:* If the constraint is fully satisfied within the range (Lines 11-12 in Algorithm 2), the algorithm identifies the range as a valid solution. It then extracts the concrete repair values from the range and the top-k results as detailed in Table 1.
   - *Partially Satisfying the Constraint:* If the constraint partially satisfies the range (Lines 13-14 in Algorithm 2), (e.g., the constraint bounds are $[0.1, 0.2]$, and the result of the constraint evaluation over the range is $[0.0, 0.15]$), the algorithm divides the range further (Line 14 in Algorithm 2), as explained in Section 5.3.1. The divided subranges are then pushed back into the queue, prioritized by their similarity to the user query if they contain potential repairs. The algorithm subsequently returns to Step 1.
(4) **Non-Divisible Ranges:** If the popped range cannot be divided further (Lines 5-7 in Algorithm 2) (e.g., the range is $[30, 30]$), the algorithm filters the tree based on the single repair value (in this case, 30). The returned set of clusters is then evaluated against the constraints using the methods outlined in Section 5.2.

Table 2: Bounds for expressions involving E_1 and E_2 under different operators [30].

op	bounds for the expression (E_1 op E_2)
+	$(\overline{E_1} + \overline{E_2}) = \overline{E_1} + \overline{E_2}, \quad (\underline{E_1} + \underline{E_2}) = \underline{E_1} + \underline{E_2}$
−	$(\overline{E_1} - \underline{E_2}) = \overline{E_1} - \underline{E_2}, \quad (\underline{E_1} - \overline{E_2}) = \underline{E_1} - \overline{E_2}$
×	$(E_1 \times E_2) = \max(\underline{E_1} \times \underline{E_2}, \underline{E_1} \times \overline{E_2}, \overline{E_1} \times \underline{E_2}, \overline{E_1} \times \overline{E_2})$ $(E_1 \times E_2) = \min(\underline{E_1} \times \underline{E_2}, \underline{E_1} \times \overline{E_2}, \overline{E_1} \times \underline{E_2}, \overline{E_1} \times \overline{E_2})$
/	$(E_1/E_2) = \max(\underline{E_1}/\underline{E_2}, \underline{E_1}/\overline{E_2}, \overline{E_1}/\underline{E_2}, \overline{E_1}/\overline{E_2})$ $(E_1/E_2) = \min(\underline{E_1}/\underline{E_2}, \underline{E_1}/\overline{E_2}, \overline{E_1}/\underline{E_2}, \overline{E_1}/\overline{E_2})$

5.3.1 Ranges Division The input to RP technique consists of two initial ranges, representing a default division of the possible repair conditions. Where each range contains half of the possible repairs.

These ranges are ordered based on their similarity to the user query. During the filtering process, our approach handles partially satisfying ranges by splitting each condition's range further into two halves. This incremental exploration of the search space allows the algorithm to focus on promising regions, avoiding unnecessary computations. This strategy is particularly advantageous for large datasets, as it reduces computational overhead while ensuring efficient identification of relevant solutions. The ranges division is based on the following:

$$\text{Divide:} \quad [a, b] \rightarrow [a, \text{mid}] \cup [\text{mid} + 1, b] \tag{4}$$

This systematic division allows the algorithm to explore the repair space incrementally.

Adriane says: c) it begs teh question of why evenly.

Adriane says: You never reference this algorithm in the text. Although I have big questions about how you filter, so manybe you should. You should reference Algorithm and Line to help your explanation.

5.3.2 Concrete Values Extraction and Top-k Repairs This step generates concrete solutions within the specified ranges that satisfy the given repairs from the list of all possible repair conditions. For fully satisfied clusters, concrete values are directly identified as the algorithm only considers repairs that are guaranteed to meet the constraints. In the case of partially satisfied clusters, the algorithm focuses only on those ranges that satisfy the constraints. From these ranges, potential repairs are dynamically filtered based on the list of possible repairs. This ensures that only relevant ranges contribute to the extraction of concrete values. For each concrete repair identified, the similarity is computed as the sum of the absolute differences between the repair values and the corresponding user query values. This approach ensures computational efficiency by avoiding unnecessary evaluations of irrelevant ranges or repairs. Once the concrete values for each repair are obtained, our technique do the following for each satisfying repair to generate the top-k

Algorithm 2 Range-Based Filtering

Input: Cluster Tree T, Conditions C, Operators O, Ranges R, Repair Tuples RT
Output: Top-K solutions satisfying user-defined predicates

Initialize an empty list for solutions: `ConcreteValuesList`
Initialize a priority queue `PriorityQueue` with ranges based on similarity
while `PriorityQueue` is not empty **do**
    `CurrentRanges` ← Pop most similar range from `PriorityQueue`
    if `CurrentRanges` cannot be divided further **then**
        Perform **full filtering** using exact matches for ranges
        Evaluate constraints and append results to `ConcreteValuesList`
    else
        `FilteredClusters` ← Filter based on `CurrentRanges`
        Evaluate constraints
        if Result within constraint boundaries **then**
            Append to `ConcreteValuesList` and extract concrete values and top-k
        else if Result partially within constraint boundaries **then**
            Divide `CurrentRanges` into subranges
            for all Subranges in `CurrentRanges` **do**
                if The range contains potential solutions **then**
                    Add to `PriorityQueue` with updated similarity scores
                end if
            end for
        end if
    end if
end while
Sort `ConcreteValuesList` by similarity and select Top-K solutions
return `ConcreteValuesList`

solutions and determine whether the process should continue or stop:

(1) **Sort Concrete Repairs**: Sort all concrete repairs based on their similarity to the user query.
(2) **Select Top-k Repairs**: Identify the top-k repairs using the user-defined parameter k.
(3) **Determine Maximum Similarity**: Calculate the maximum similarity value within the top-k repairs.
(4) **Compare with Next Range in the processing queue**: Retrieve the minimum similarity value of the next range in the priority queue.
(5) **Evaluate Stopping Condition**: If the minimum similarity of the next range is greater than the maximum similarity of the top-k repairs: stop the process and return the top-k repairs.
(6) **Continue Processing**: If the stopping condition is not met: pop the next range from the queue and continue processing the new

Adriane says: This level of detail is not needed. It's basic and non-interesting.

Algorithm 3 Range-Based Pruning Query Repair

Input: Cluster Tree T, Conditions C, Operators O
Output: Filtered Clusters List F

$stack$ ← [Root Clusters from T]
F ← {} ▷ Initialize filtered clusters
while $stack$ is not empty **do**
    $current\_key$ ← Pop a cluster from T
    Min ← retrieve current cluster Min
    Max ← retrieve current cluster Max
    $cluster\_fully\_satisfies$ ← True
    $cluster\_partially\_satisfies$ ← True
    for all $(condition, operator) \in (C, O)$ **do**
        if Check if the cluster is NOT fully included within the range **then**
            $fully\_satisfies$ ← False
        end if
        if Check if the cluster is NOT Partially included within the range **then**
            $partially\_satisfies$ ← False
            break ▷ No need to check further
        end if
    end for
    if $fully\_satisfies$ **then**
        Mark cluster as **Fully**
        Append cluster to F
    else if $partially\_satisfies$ **then**
        if current_key has child **then**
            for all child in current cluster **do**
                Min ← retrieve child cluster Min
                Max ← retrieve child cluster Max
                $child\_fully\_satisfies$ ← True
                $child\_partially\_satisfies$ ← True
                for all $(condition, operator) \in (C, O)$ **do**
                    if Check if the cluster is NOT fully included within the range **then**
                        $child\_fully\_satisfies$ ← False
                    end if
                    if Check if the cluster is NOT Partially included within the range **then**
                        $child\_partially\_satisfies$ ← False
                        break
                    end if
                end for
                if $child\_fully\_satisfies$ **then**
                    Mark child as **Full**
                    Append child to F
                else if $child\_partially\_satisfies$ **then**
                    Push child onto $stack$
                end if
            end for
        else
            Mark cluster as **Partial**
            Append cluster to F
        end if
    else
        Push children of current cluster onto $stack$
    end if
end while
return F

This stepwise process ensures that the most promising repairs are evaluated first while minimizing unnecessary computations.

6 Experiments

We evaluate the performance of our algorithms using real-world and TPCH benchmark datasets, various queries, and numerous arithmetic constraints. It begins with demonstrating the performance of Range-Based Pruning Query Repair (RP) algorithm (Section 5) and the baseline Fully Filtering Query Repair (FF) technique (Section 4.2) in Section 6.2. We then investigate the impact of the factors affecting the performance such as dataset size and the similarity distance of the repairs to the user query in Section 6.3. Finally, we compare our approach with Erica [19], which focuses on addressing cardinality constraints over groups in the query output by generating all minimal refinements, in Section 6.4.

6.1 Experimental Setup

Datasets. We choose two real-world datasets of size of 500K, **Adult Census Income (ACSIncome)** [14] and **Healthcare** [15], that are commonly used to evaluate fairness. We also utilize a standard benchmark, **TPC-H** [1], to evaluate our methods while varying the data size from 25K to 500K. We converted categorical columns to numerical columns as the algorithms are designed for that.

Queries. Table 7 presents the queries used in this experiment. For Healthcare dataset, we adaopt queries Q_1 and Q_2 from [19] and generate Q_3. For ACSIncome dataset, we take query Q_4 from [19] and we generate Q_5 and Q_6. We generated Q_7 with 3 predicates inspired by TPC-H's Q_2.

Constraints. For Healthcare and ACSIncome, we enforce the statistical parity difference (SPD) between two demographic groups to be within a certain bound. Table 4 shows the details of the constraints used. B_l and B_u represent a specific lower and upper bound, respectively. C_1 and C_2 are the constraints for the Healthcare dataset with various bounds between 0.2 and 0.6. Similarly, C_3 and C_4 are the constraints for the ACSIncome dataset with multiple bounds between 0.2 and 0.5. For ACSIncome, we determine groups based on gender and race requiring that the SPD is in certain ranges while, for Healthcare, we determine demographic groups based on race and age group requiring that the SPD fall in the defined ranges. For TPC-H, we enforce the aggregate constraint C_5 from the use case in Section 2.2 to be larger than or equal to a specific value. C_6 through C_8 are defined for comparison with Erica.

Number of Candidate Repairs. The number of candidate repairs is calculated as the product of the unique possible values for all predicates. For ACSIncome dataset, this number ranges from approximately 14K to 18K. In Healthcare dataset, it varies between approximately 10K and 22k. For TPCH queries, the number of possible repair candidates is approximately 38K.

Parameters. There are three key tuning parameters that could impact the performance of our methods. Recall that we use a kd-tree to perform the clustering described in Section 4.1. Two parameters in the tree are as follows:

- **Multi-clusters Partitioning (Branching)**: Unlike the traditional binary kd-tree, where each cluster partitions data into two clusters, each cluster is split into multiple branches.

ID	Query
	`SELECT * FROM Healthcare`
Q_1	`WHERE income >= 200K AND num-children >= 3 AND county <= 3`
Q_2	`WHERE income <= 100K AND complications >= 5 AND num-children >= 4`
Q_3	`WHERE income >= 300K AND complications >= 5 AND county == 1`
	`SELECT * FROM ACSIncome`
Q_4	`WHERE working_hours >= 40 AND Educational_attainment >= 19 AND Class_of_worker >= 3`
Q_5	`WHERE working_hours <= 40 AND Educational_attainment <= 19 AND Class_of_worker <= 4`
Q_6	`WHERE Age >= 35 AND Class_of_worker >= 2 AND Educational_attainment <= 15`
	`SELECT * FROM part, supplier, partsupp, nation, region`
Q_7	`WHERE p_partkey = ps_partkey AND s_suppkey = ps_suppkey AND s_nationkey = n_nationkey AND n_regionkey=r_regionkey AND p_size >= 10 AND p_type in ('LARGE BRUSHED') AND r_name in ('EUROPE')`

Table 3: Queries for Experimentation

ID	Constraint
C_1	$B_l \leq \frac{\text{count(race=1}\wedge\text{label=1)}}{\text{count(race=1)}} - \frac{\text{count(race=2}\wedge\text{label=1)}}{\text{count(race=2)}} \leq B_u$
C_2	$B_l \leq \frac{\text{count(ageGroup=1}\wedge\text{label=1)}}{\text{count(ageGroup=1)}} - \frac{\text{count(ageGroup=2}\wedge\text{label=1)}}{\text{count(ageGroup=2)}} \leq B_u$
C_3	$B_l \leq \frac{\text{count(sex=1}\wedge\text{PINCP}\geq 20k\text{)}}{\text{count(sex=1)}} - \frac{\text{count(sex=2}\wedge\text{PINCP}\geq 20k\text{)}}{\text{count(sex=2)}} \leq B_u$
C_4	$B_l \leq \frac{\text{count(race=1}\wedge\text{PINCP}\geq 15k\text{)}}{\text{count(race=1)}} - \frac{\text{count(race=2}\wedge\text{PINCP}\geq 15k\text{)}}{\text{count(race=2)}} \leq B_u$
C_5	$B_l \leq \frac{\text{sum(p\_retailprice)}}{\text{count(p\_retailprice)}} - \frac{\text{sum(s\_acctbal)}}{\text{count(s\_acctbal)}}$
C_6	$\{\text{count(race)} = \text{race1}\} \leq B_u, \{\text{count(age)} = \text{group1}\} \leq B_u$
C_7	$\{\text{count(race)} = \text{race1}\} \leq B_u, \{\text{count(age)} = \text{group1}\} \leq B_u,$ $\{\text{count(age)} = \text{group3}\} \leq B_u$
C_8	$\{\text{count(Sex)} = \text{Female}\} \leq B_u, \{\text{count(Race)} = \text{Black}\} \leq B_u,$ $\{\text{count(Marital)} = \text{Divorced}\} \leq B_u$

Table 4: Constraints for Experimentation

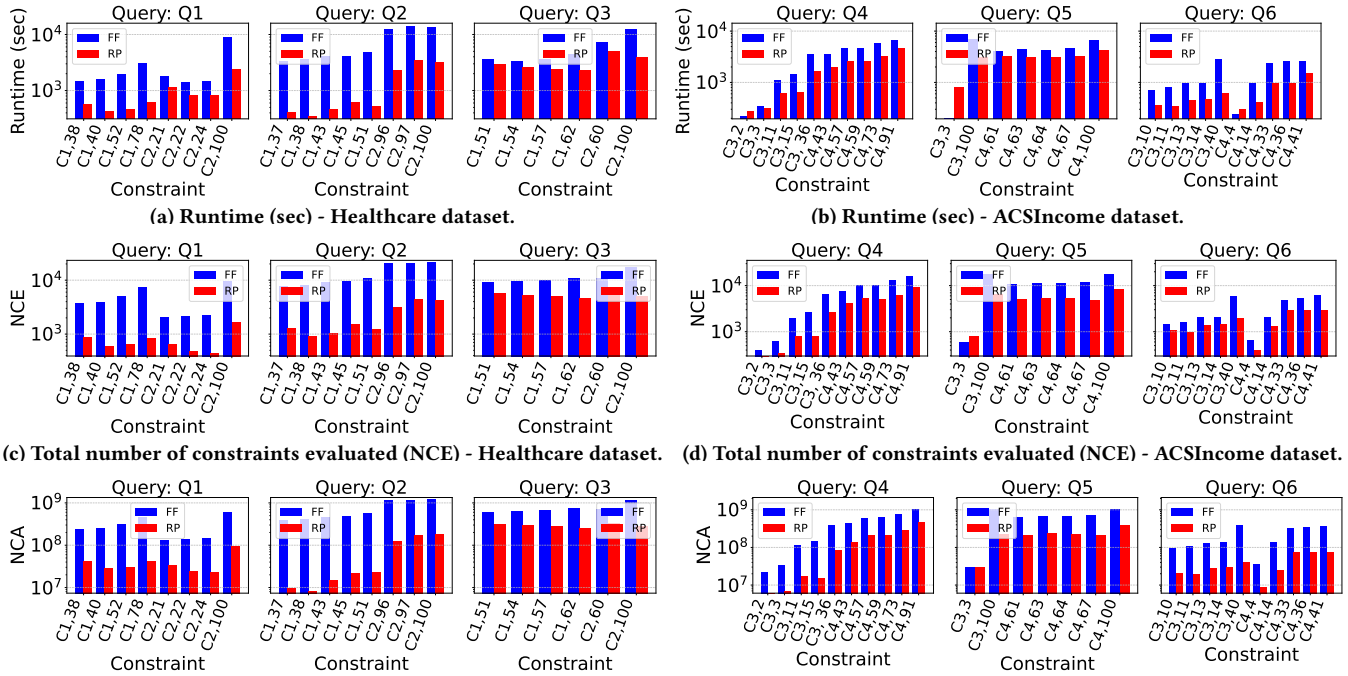
- **Bucket Size**: This parameter determines the minimum number of points a cluster can hold before further partitioning. If the number of points in a cluster is below this threshold, the algorithm stops splitting and assigns the points to individual leaf clusters.

In addition, we evaluate our methods over, top-k, a user-defined parameter that reflects the user's preference for the number of repairs to be generated by our techniques. The default setting for these parameters is as follows: 5 branches, top-7 solutions, and a bucket size of 15 while fixing a dataset size at 50K.

Platform. The algorithms were implemented in Python, and the experiments were conducted on a machine with 2 x 3.3Ghz AMD Opteron CPUs (12 cores) and 128GB RAM. Each evaluation was repeated five times and we report the median runtime as the variance is low ($\sim$ 3%).

6.2 Performance of FF and RP

We conduct experiments to measure the performance of FF and RP using the Healthcare and ACSIncome datasets with queries in Table 7, constraints in Table 4, and default settings in Section 6.1. For the Healthcare, the constraints C_1 and C_2 are used while C_3 and C_4 are considered for the ACSIncome.



(a) Runtime (sec) - Healthcare dataset.

(b) Runtime (sec) - ACSIncome dataset.

(c) Total number of constraints evaluated (NCE) - Healthcare dataset.

(d) Total number of constraints evaluated (NCE) - ACSIncome dataset.

(e) Total number of cluster accessed (NCA) - Healthcare dataset.

(f) Total number of cluster accessed (NCA) - ACSIncome dataset.

Figure 2: Performance result of FF and RP over the Healthcare and ACSIncome datasets using queries in Table 7

Comparison with Brute Force. We first compare our proposed methods, FF and RP, with the brute force (BF) method using the Healthcare dataset, queries Q_1 and Q_2, and the constraint C_1 with various bounds. As expected, both FF and RP outperform BF by at least one order of magnitude in terms of runtime. The RP algorithm significantly reduces both the total number of constraints evaluated (NCE) and the total number of clusters accessed (NCA), while the FF method maintains the same NCE as BF but decreases the NCA compared to BF. The detailed evaluation result is available in [].

Seokki says: Cite the extended version of the paper

Runtime. Figures 2a and 2b illustrate the runtime of the FF and RP algorithms for the Healthcare and ACSIncome datasets, respectively. The x-axis shows pairs of a constraint used and the proximity of solutions (exploration distance). The exploration distance is defined as the ratio of the actual number of candidate solutions checked by the algorithm for the given Top-k to the total number of candidate solutions available. For example, (C_1,38) in Figure 2a for Q_1 presents the constraint C_1 is taken in Table 4 with a bound where solutions can be found after exploring 38% of the search space. The RP algorithm (red bars) generally outperforms FF (blue bars) in most constraints. In Figure 2b, two algorithms exhibit similar performance for Q_4 with C_3, where solutions are found after exploring only 2% and 3% of the search space. A similar pattern is observed in the evaluation for Q_5 with (C_3,3) and Q_6 with (C_4,4). In general, RP significantly outperforms FF, demonstrating an improvement of about an order of magnitude. This performance gain is attributed to RP's ability to efficiently explore the search space by leveraging range-based pruning rather than exhaustively evaluating all possible candidates.

A key insight from these findings is that the efficiency of RP is strongly influenced by the proximity of solutions within the search space. When solutions are closer to the start, fewer explorations are required, allowing FF to quickly locate the answer. Conversely, RP needs to examine multiple ranges, leading to increased execution time.

Total Number of Constraints Evaluated (NCE). We further analyze how the number of candidate solutions checked affects the performance of our methods. Figures 2c and 2d show the result of the total number of constraints evaluated (on the y axis) for FF and RP over the Healthcare and ACSIncome datasets, respectively. As shown in the figures, RP consistently checks fewer constraints compared to FF which evaluates each possible repair individually, while RP efficiently reduces the number of evaluations through the range-based pruning algorithm (Section 5.3). As observed in the runtime evaluation, the solution proximity influences the efficiency of our algorithms, producing comparable performance between FF and RP when the proximity of solutions is close to the start, e.g., as shown in Figure 2d for Q_4 with (C_3,2) and Q_5 with (C_3,3).

Total Number of Cluster Accessed (NCA). The results of the number of clusters accessed are shown in Figures 2e and 2f for Healthcare and ACSIncome, respectively. Similarly, RP accesses significantly fewer clusters than FF, highlighting its efficiency in limiting the exploration of the search space. Furthermore, it follows the same trend of the previous evaluation results such that the benefit of RP becomes negligible and may even reverse when the proximity of solutions is low.

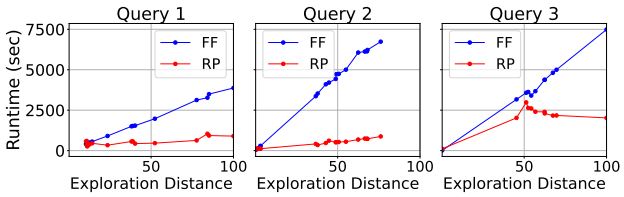
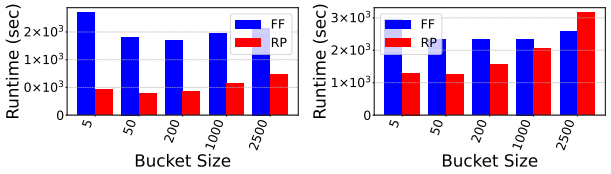


Figure 3: Runtime of FF and RP over exploration distance



(a) Healthcare dataset **(b) ACSIncome dataset**

Figure 4: Runtime for different bucket sizes.

6.3 Performance-Impacting Factors

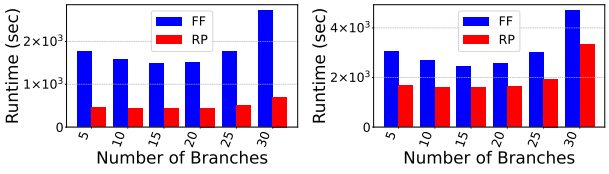
To gain deeper insights into the behavior observed in Section 6.2, we will further investigate the relationship between the proximity of solutions (search space exploration distance) and algorithm performance in this section. Additionally, we evaluate the performance of FF and RP in terms of the parameters in Section 6.1. We use the Healthcare, ACSIncome, and TPC-H datasets with queries in Table 7 and constraints in Table 4.

Effect of Solution Exploration Distance. We here focus on a deeper investigation by using queries Q_1–Q_3 and the constraint C_1 with various bounds on the Healthcare dataset. The detailed evaluation for the ACSIncome dataset is available in [].

Seokki says: cite the extended version

The result is shown in Figure 3. For Q_1 and Q_2, when the distance is approximately 10% or less, FF and RP exhibit comparable performance. A similar pattern is seen for Q_3, where FF performs better than RP at shorter distances, but RP overtakes FF at approximately 50% distance. In general, as the distance increases, we observe the performance gap widens, a clear advantage of RP. As highlighted in Section 6.2, constraints with solutions located farther in the search space favor RP, while those closer to the beginning may reduce its advantage. The reason behind these trends is that when solutions are closer to the start of the search space, FF requires fewer explorations, allowing it to quickly locate a solution that satisfies the constraint. Conversely, RP must examine multiple ranges from the beginning of the search space, leading to increased execution time.

Effect of Bucket Size. We now evaluate the runtime of FF and RP in terms of the number of bucket size using Q_1 with C_1 for the Healthcare dataset and Q_4 with C_3 for the ACSIncome dataset. While data size is set to 50K, branch size is at 5, and top-k = 7, we vary the bucket size from 5 to 2500. Selecting different size of buckets results in varying the number of leaf clusters and data points in each leaf. With a default branching factor of 5, the structure of the clustering tree for this evaluation is as follows: (i) Level 1: 5 nodes, each with 10,000 data points; (ii) Level 2: 25 nodes, each with 2,000 data points; (iii) Level 3: 125 nodes, each with 400 data points; (iv) Level 4: 525 nodes, each with 80 data points; (v) Level 5: 3,125 nodes, each with 16 data points; (vi) Level 6: 15,625 nodes, each



(a) Healthcare dataset **(b) ACSIncome dataset**

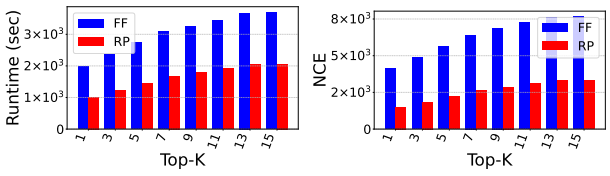
Figure 5: Runtime of varying the number of branches

with 3 or 4 data points. Note that the algorithms traverse the cluster tree up to the level where the capacity of each node at that level is less than equal to the defined bucket size. For example, for the bucket size = 200, the search proceeds up to Level 4. The result is shown in Figure 4 for both datasets. For the FF algorithm, when the bucket size is 5, the algorithm leads to the worst performance for FF where it traverses the largest number of leaves (15,625). For the smaller bucket size (up to bucket size ≃ 200), the runtime decreases as the bucket size increases. Upon the bucket size becoming larger, e.g., 1000 and 2000, the runtime increases. Although the algorithms check a smaller number of clusters in the tree, each node contains many data points, which reduces filtering opportunities. A similar trend can be seen in the results of ACSIncome. The results of the RP algorithm also align with this trend. RP tends to perform better with smaller bucket sizes because it can effectively prune more buckets. When the buckets are small, each leaf may contain only 3 or 4 data points, which reduces the need for further searching. However, as shown in Figure 4b, the performance of RP may decline compared to FF when the bucket size is large. Larger buckets contain more data points, which diminishes the filtering advantages of RP.

# of Branches	# of Leaves	# of Branches	# of Leaves
5	15625	20	8000
10	10000	25	15625
15	3375	30	27000

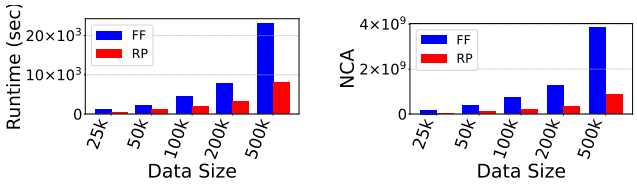
Table 5: Branching Configuration and Data Distribution

Effect of Number of Branches. The next analysis examines the relationship between the number of branches in the tree and the runtime of the FF and RP algorithms. The same queries, constraints, and datasets in the previous evaluation are used here. In this experiment, the number of branches ranges from 5 to 30. The corresponding number of leaf clusters in the tree is presented in Table 5. While varying the number of branches, the depth of the tree changes as each cluster at the parent level needs to be split until it reaches the desired bucket size which is default = 15 in this case. For example, 5 branches produces a cluster of depth of 6 whereas 30 branches yields a cluster of depth of 3. Interestingly, for each number of branches, the structure of the cluster remains the same across both datasets. The result shown in Figure 5 confirms that the performance of FF and RP correlates with the number of clusters at the leaf level. For the FF algorithm, the branching factors of 5 and 25 yield nearly identical runtime because both have the same number of leaves (15,625). A similar pattern can be observed for the branching factors of 10 and 20. At a branching factor of 15, FF achieves the lowest runtime, as it involves the smallest number of leaves (3,375). When the branching factor increases to 30, the number of leaf clusters significantly increases, leading to a substantial rise in



(a) Runtime(sec) (b) Number of constraints evaluated

Figure 6: Performance of FF and RP over top-k



(a) Runtime (sec) (b) Number of clusters accesses

Figure 7: Evaluation result over TPC-H dataset

the runtime of FF. It also infers that the depth of the tree does not affect the performance of our methods. For RP, overall performance trends align with those of FF. However, RP is less influenced by variations in the branching factor. As a comparison between the two algorithms, RP consistently outperforms FF, demonstrating greater efficiency across the datasets.

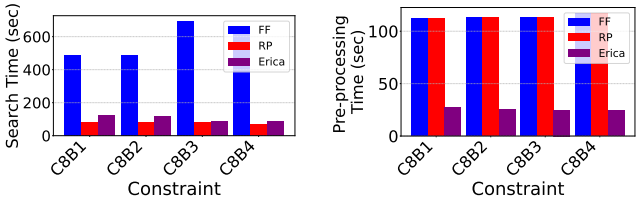
Effect of Top-k. In this experiment, we vary the parameter top-k from 1 to 15. For both FF and RP, as k increases, the runtime also increases, as shown in Figure 6a. This behavior is expected since finding a single repair (k=1) requires less computation than identifying multiple repairs. When k is larger, the algorithm must explore a broader search space to retrieve additional valid repairs. Similarly, both the number of constraints evaluated as shown in Figure 6b and the number of boxes accessed exhibit the same increasing trend. This pattern is observed consistently across both FF and RP, reinforcing the intuition that retrieving more solutions requires higher computational effort. Overall, across different top-k values, RP consistently outperforms FF repair technique, demonstrating greater efficiency in retrieving multiple solutions.

Effect of Dataset Size. This evaluation assesses runtime and the number of cluster accessed (NCA) of our algorithms using the TPC-H dataset, Q_7 in Table 7, C_5 in Table 4, and default settings in Section 6.1. As shown in Figure 7a, as the data size increases, the runtime also increases. While increasing data size, both FF and RP must explore a larger search space. This is further supported by the NCA evaluation results shown in Figure 7b, which indicate that more clusters are examined as the dataset grows.

6.4 Comparison with Related Work

We compare our approach with Erica [19], which focuses on addressing the problem of finding all minimal refinements of a given query that satisfy specific cardinality constraints for data groups within the result set. To conduct the evaluation for Erica, we used the available Python implementation[1]. We compare the generated refinements and runtime of our techniques with Erica using Q_1 and Q_2 (Table 7) on the Healthcare dataset (size of 50K) with constraints

[1]https://github.com/JinyangLi01/Query_refinement



(a) Search Time (b) Pre-processing Time

Figure 8: Comparison of FF and RP with Erica.

C_6 and C_7 (Table 4), respectively. For fair comparison, we adopt the queries, constraints, and the dataset from Erica. While Erica is limited to cardinality constraints, our approach focuses on more challenging constraints.

Comparison on the Generated Repairs. We first compare the generated repairs by our approach and Erica. For Q_1 with C_6, Erica generates 7 minimal repairs whereas our technique generates 356, including those produced by Erica. Similarly, for Q_2 with C_7, Erica generates 9 minimal repairs while our approaches generate 1035, including those produced by Erica. In general, our techniques successfully generate all repairs produced by Erica. Note that our approaches generate repairs that are closer to the original user queries compared to the repairs by Erica. For example, in Q_2, given the condition num-children >= 4 in the user query, our solution includes a refined condition num-children >= 3 whereas Erica provides a refinement like num-children >= 1 which is far from the user query. The differences in the results arise from the definitions of minimality defined by the two approaches. Erica considers user preferences, focusing on repairs that best satisfy these preferences. In contrast, our approach emphasizes similarity to the original user query by exploring the space of actual values for each predicate. In this evaluation, to include the Erica's solutions, we increase our Top-k parameter, demonstrating that our technique encompasses Erica's solutions while offering additional repairs.

Runtime Comparison. To ensure a fair comparison of execution time, we fix the number of generated repairs (i.e., Top-k) in our approach to equal to the number of repairs produced by Erica. For constraints $C8B1$ and $C8B2$ (k=17), for $C8B3$ (k=11) and for $C8B4$ (k=13). The experiment utilizes Q_4 with C_8 on the 50K ACSIncome dataset, which is derived from Erica's dataset, query, and constraint. We also use the same bounds for both Erica and our algorithms: B1 = {30, 150, 10} and B2 = {30,300,25}, B3 = {10, 650, 50}, and B4 = {15, 200, 15}. However, due to differing minimality definitions, variations in the generated repairs between our approach and Erica are expected. The result shown in Figure 8a reveals an advantage of the RP algorithm, which outperforms Erica in search time which is the time of exploring the search space to generate a repair. However, as shown in Figure 8, in pre-processing time which is the time of collecting statistical information, Erica outperforms both RP and FF, indicating that Erica summarizes information needed for the search process more effectively. Meanwhile, FF exhibits the worst performance across.

7 Related Work

Li et al. [19] determine all minimal refinements of a conjunctive query by changing constants in selection conditions such that the

refined query fulfills a conjunction of cardinality constraints, e.g., the query should return at least 5 answers where gender = female. A refinement is minimal if it fulfills the constraints and there does not exist any refinement that is closer to the original query in terms of similarity of constants used in predicates. The cardinality constraints in [19] involve filter-aggregation queries that we also consider as compared in Section 6.4. However [19] do not allow for arithmetic combinations of the results of such queries, e.g., as required by standard fairness conditions. Both our work and [19] face the challenge of an exponential search space (all possible combinations of changes to individual predicates in a query). For monotone constraints (either only relaxation of predicates or restriction of predicates is required to fix the query), the problem can be solved in PTIME by exploiting the ordering induced by monotonicity, as discussed in Section 3.

Query refinement. Another line of work related to our approach is query refinement [18, 22, 26]. Mishra et al. [22] refine a query to return a given number k of results. Koudas et al. [18] refine a query that returns an empty result to produce at least one answer. In [6, 26] a query repairs return missing results of interest provided by the user. For a query with n predicates, the number of possible refinements is exponential in n: for each predicate one can choose a constant from the domain of the attribute restricted by the predicate. Most work on query refinement has limited the scope to constraints that are monotone in the size of the query answer. Monotonicity is then exploited to prune the search space [7, 16, 22, 23, 29]. However, real-world use cases are often inherently non-monotone.

How-to queries. Like our work and [19], the purpose of how-to queries [21] is to achieve a desired change to the result of a query. However, in how-to queries this is achieved by changing the input database rather than changing the query. Wang et al. [28] study the problem of deleting operations from an update history to fulfill a constraint over the current database expressed as tuple substitutions (replace t_i in the result with t_i'). However, this approach does not consider query repair (changing predicates) nor aggregate constraints.

Explanations for Missing Answers. Query-based explanations for missing answers [9, 11, 12] are operators or sets of operators that are responsible for the failure of a query to return a result of interest. However, this line of work does not generate query repairs.

Bounds and Interval Arithmetic. Previous studies have highlighted the effectiveness of bounds and interval arithmetic across various database applications [10, 13, 25, 30]. For instance, [13] explored uncertainty tracking using attribute-level bounds for complex queries, demonstrating the utility of bounds-based approaches in query processing. Similarly, the work in [30] introduced a bounding technique for efficiently computing iceberg cubes, establishing an early foundation for leveraging interval arithmetic to constrain aggregate functions.

Building on these principles, our work addresses scenarios involving non-monotone constraints, where previous pruning techniques are not supporting these constraints. Our approach extends the bounding techniques proposed in [30], originally applied to aggregate functions in data cube computations. We adapt this methodology to the domain of query repair, utilizing interval arithmetic to evaluate constraints over defined ranges and efficiently prune unproductive search paths.

8 Conclusion and Future Work

In this work, we address the problem of repairing a query to satisfy a constraint that is a combination of arithmetic operators and aggregation functions. Unlike previous approaches that proposed query repairs to satisfy cardinality constraints or constraints of standard aggregation functions that follow the monotonicity property, our proposed technique supports an arithmetic constraint that breaks the monotonicity property. Our approach is capable of satisfying the constraints from previous works, as demonstrated in our experiments, while also supporting a wider range of constraints with arithmetic operators. To achieve this, we have proposed a technique to handle the large search space of possible solutions, along with a pruning technique to exclude unpromising candidate solutions. We exploit the use of ranges and arithmetic intervals, which allows for an efficient pruning process. The results of our experiments demonstrate the efficiency of our techniques across different datasets, queries, and constraints. For future work, this technique has been tested with datasets and queries containing a maximum of 38k candidate solutions. It would be beneficial to explore the applicability of our approach to scenarios involving a higher number of candidate solutions, which we leave for future research.

References

[1] 2024. TPC BENCHMARK H (Decision Support) Standard Specification Revision 3.0.1. https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp. Accessed on 2024-03-20.

[2] Abdullah M. Albarrak and Mohamed A. Sharaf. 2017. Efficient schemes for similarity-aware refinement of aggregation queries. *World Wide Web* 20, 6 (2017), 1237–1267.

[3] Rachel K. E. Bellamy, Kuntal Dey, Michael Hind, Samuel C. Hoffman, Stephanie Houde, Kalapriya Kannan, Pranay Lohia, Jacquelyn Martino, Sameep Mehta, Aleksandra Mojsilovic, Seema Nagar, Karthikeyan Natesan Ramamurthy, John T. Richards, Diptikalyan Saha, Prasanna Sattigeri, Moninder Singh, Kush R. Varshney, and Yunfeng Zhang. 2019. AI Fairness 360: An extensible toolkit for detecting and mitigating algorithmic bias. *IBM J. Res. Dev.* 63, 4/5 (2019), 4:1–4:15.

[4] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, and Jennifer Widom. 2006. ULDBs: databases with uncertainty and lineage. In *Very Large Data Bases Conference*. https://api.semanticscholar.org/CorpusID:1899963

[5] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

[6] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. 2016. Refining SQL Queries based on Why-Not Polynomials. In *TaPP*.

[7] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. 2006. Generating Queries With Cardinality Constraints for DBMS Testing. *TKDE* 18, 12 (2006), 1721–1725.

[8] N. Bruno, S. Chaudhuri, and D. Thomas. 2006. Generating Queries with Cardinality Constraints for DBMS Testing. *IEEE Transactions on Knowledge and Data Engineering* 18, 12 (2006), 1721–1725. https://doi.org/10.1109/TKDE.2006.190

[9] Adriane Chapman and H. V. Jagadish. 2009. Why not?. In *SIGMOD*. 523–534.

[10] Luiz Henrique De Figueiredo and Jorge Stolfi. 2004. Affine arithmetic: concepts and applications. *Numerical algorithms* 37 (2004), 147–158.

[11] Daniel Deutch, Nave Frost, Amir Gilad, and Tomer Haimovich. 2020. Explaining Missing Query Results in Natural Language. In *EDBT*. 427–430.

[12] Ralf Diestelkämper, Seokki Lee, Melanie Herschel, and Boris Glavic. 2021. To Not Miss the Forest for the Trees - A Holistic Approach for Explaining Missing Answers over Nested Data. In *SIGMOD*. 405–417.

[13] Su Feng, Boris Glavic, Aaron Huber, and Oliver A Kennedy. 2021. Efficient uncertainty tracking for complex queries with attribute-level bounds. In *Proceedings of the 2021 International Conference on Management of Data*. 528–540.

[14] Sorelle A. Friedler, Carlos Scheidegger, Suresh Venkatasubramanian, Sonam Choudhary, Evan P. Hamilton, and Derek Roth. 2019. A comparative study of fairness-enhancing interventions in machine learning. In *FAT*. 329–338.

[15] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. 2021. Mlinspect: A data distribution debugger for machine learning pipelines. In *SIGMOD*. 2736–2739.

[16] Abhijit Kadlag, Amol V. Wanjari, Juliana Freire, and Jayant R. Haritsa. 2004. Supporting Exploratory Queries in Databases. In *DASFAA*, Vol. 2973. 594–605.

[17] Dmitri V Kalashnikov, Laks VS Lakshmanan, and Divesh Srivastava. 2018. Fastqre: Fast query reverse engineering. In *SIGMOD*. 337–350.

[18] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. 2006. Relaxing Join and Selection Queries. In *VLDB*. 199–210.

[19] Jinyang Li, Yuval Moskovitch, Julia Stoyanovich, and HV Jagadish. 2023. Query Refinement for Diversity Constraint Satisfaction. *Proceedings of the VLDB Endowment* 17, 2 (2023), 106–118.

[20] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. 2022. A Survey on Bias and Fairness in Machine Learning. *ACM Comput. Surv.* 54, 6 (2022), 115:1–115:35.

[21] Alexandra Meliou and Dan Suciu. 2012. Tiresias: the database oracle for how-to queries. In *SIGMOD*. 337–348.

[22] Chaitanya Mishra and Nick Koudas. 2009. Interactive query refinement. In *EDBT*. 862–873.

[23] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating targeted queries for database testing. In *SIGMOD*. 499–510.

[24] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*. 476–487.

[25] Jorge Stolfi and Luiz Henrique de Figueiredo. 2003. An introduction to affine arithmetic. *Trends in Computational and Applied Mathematics* 4, 3 (2003), 297–312.

[26] Quoc Trung Tran and Chee-Yong Chan. 2010. How to conquer why-not questions. In *SIGMOD*. 15–26.

[27] Bienvenido Vélez, Ron Weiss, Mark A. Sheldon, and David K. Gifford. 1997. Fast and Effective Query Refinement. In *SIGIR*. 6–15.

[28] Xiaolan Wang, Alexandra Meliou, and Eugene Wu. 2017. QFix: Diagnosing Errors through Query Histories. In *SIGMOD*. 1369–1384.

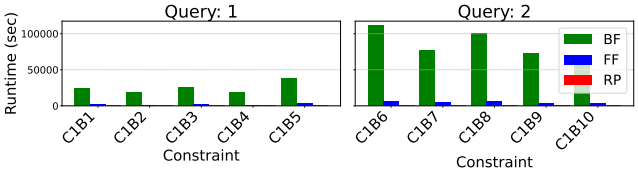
[29] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. *PVLDB* 6, 8 (2013), 553–564.

[30] Xiuzhen Zhang, Pauline Lienhua Chou, and Guozhu Dong. 2007. Efficient computation of iceberg cubes by bounding aggregate functions. *IEEE transactions on knowledge and data engineering* 19, 7 (2007), 903–918.

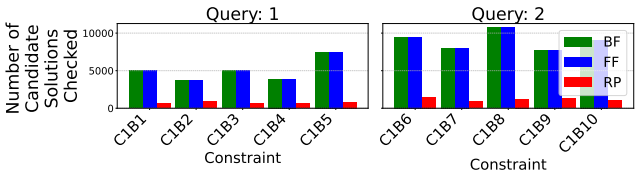
[31] Mohamed Ziauddin, Andrew Witkowski, You Jung Kim, Dmitry Potapov, Janaki Lahorani, and Murali Krishna. 2017. Dimensions based data clustering and zone maps. *PVLDB* 10, 12 (2017), 1622–1633.

A Comparison with Brute Force

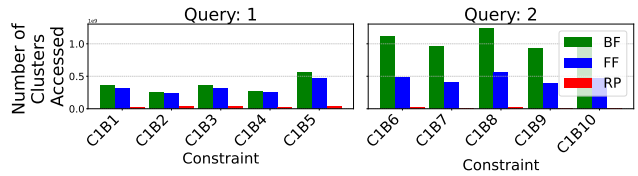
In this section, we compare the brute-force (BF) method with our proposed techniques, FF and RP. In all experiments, we use green for BF, shades of blue for FF, and shades of red for RP. We evaluate the performance of these techniques using the Healthcare dataset with queries Q1 and Q2, as well as constraint C1 with different bounds using our default settings: a dataset size of 50K, 5 branches, top-7 solutions, and a bucket size of 15.



(a) Comparison of Runtime.



(b) Comparison of the number of Candidate solutions evaluated.



(c) Comparision of number of clusters accessed.

Figure 9: Comparison of runtime, number candidate solutions evaluated, and number of clusters accessed for Healthcare dataset.

Performance. Figure 9a illustrates the runtime of the three algorithms **BF, FF, and RP** for queries Q1 and Q2 across constraint C1 with different bounds. The BF algorithm consistently exhibits the highest runtime, as it processes all data points for each repair and exhaustively evaluates all combinations, leading to significant computational overhead. Consequently, as shown in Figure 9c, BF also incurs the highest number of accesses, as it scans every individual data point in the dataset. The FF algorithm substantially reduces runtime by employing an optimized traversal technique that filters data based on the bounds of the clustering tree. Instead of accessing each data point, FF only accesses cluster boundaries, as demonstrated in Figure 9c, where it reduces the number of accesses. The RP algorithm further improves performance by efficiently pruning irrelevant repair combinations using range-based constraints. As observed in Figure 9c, RP significantly minimizes the number of accesses to clusters, making it the most efficient approach. In general, FF outperforms BF by at least an order of magnitude, while RP achieves an additional fivefold improvement, showcasing the efficiency of the proposed approaches.

Number of Candidate Solutions Checked. Figure 9b illustrates the number of candidate solutions evaluated by each algorithm for query Q1 and Q2 across different bounds of C1. The BBF and FF methods exhibit identical counts, as both exhaustively examine all possible repairs. In contrast, the RP method achieves the lowest number due to its efficient use of ranges. For example, under constraint C1B5, the BBF and FF methods evaluate over 7,400 constraints, as they individually assess each possible candidate repair. Meanwhile, the RP method evaluates approximately 850 candidate repair by leveraging ranges to skip sets of unpromising repairs. This reduction highlights the efficiency of the RP technique in eliminating redundant evaluations and minimizing computational overhead. Based on the results of the experiment, it is clear that the BF algorithm performs considerably worse than our proposed algorithms in terms of runtime. Given its poor performance and impracticality for real-world use cases, we have decided to exclude the BF algorithm from the remaining experiments. Instead, we will focus on comparing the FF and RP algorithms, as they represent more realistic and competitive solutions.

B Effect of Solution Exploration Distance

As highlighted in Section 6.2, constraints with solutions located farther in the search space favor the RP technique, while those closer to the beginning may reduce its advantage. We investigate the impact of solution distance on our proposed algorithms by running queries Q1–Q3 with constraint C1 on the Healthcare dataset using different bounds, and queries Q4–Q6 with constraint C1 on the ACSIncome dataset under varying bounds.

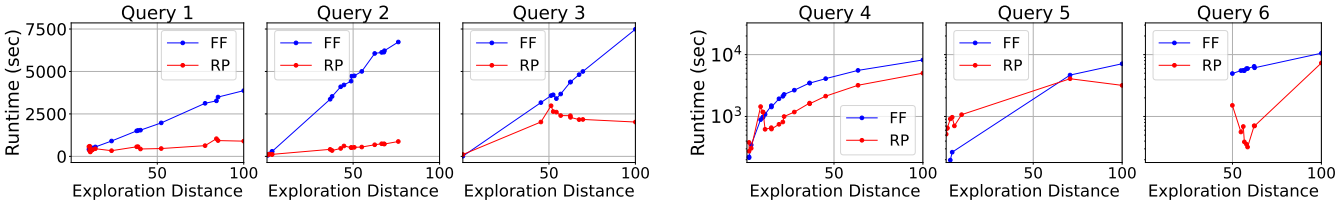
In the Healthcare dataset, we observe same trends across all three measures (Runtime, Candidate solutions checked and Clusters accessed) as Figure 10a; therefore, we do not present them here. For Query 1, when the distance is approximately 10% and less, FF and RP exhibit comparable performance. Similarly, in Queries 2 and 3, when the distance is between 0% and 3%, both algorithms perform similarly. However, as the distance increases, we observe a clear advantage for RP, the performance gap widens, indicating that the RP becomes increasingly efficient compared to Fully Filtering Repair. This confirms our previous claim that the RP benefits more when solutions are located farther in the search space.

Similarly, in the ACSIncome dataset, the trends across all three measures as Figure 10b. For Queries 4 and 5, when the distance is less than 10%, FF consistently outperforms RP. However, as the distance surpasses 10%, the performance dynamics shift, favoring RP. This change is reflected by a noticeable reduction in the number of clusters accessed and a decrease in runtime, illustrating that RP becomes increasingly effective as the search space expands.

The reason behind these trends is that when solutions are closer to the start of the search space, FF requires fewer explorations, allowing it to quickly locate a solution that satisfies the constraint. Conversely, RP must examine multiple ranges from the beginning of the search space, leading to increased execution time. These observations reinforce the assertion that the efficiency of the RP repair technique improves as the exploration distance increases, while FF maintains an advantage in scenarios where solutions are in closer proximity.

Table 6: Constraints for Experimentation

Constraint ID	Expression	Bounds
C1	$B_l \leq \frac{\text{count(race=1}\wedge\text{label=1)}}{\text{count(race=1)}} - \frac{\text{count(race=2}\wedge\text{label=1)}}{\text{count(race=2)}} \leq B_u$	B_1:[0.44, 0.5], B_2:[0.25, 0.5], B_3:[0.42, 0.5], B_4:[0.35, 0.5], B_5:[0.5, 0.6], B_6:[0.23, 0.25], B_7:[0.25, 0.4], B_8:[0.34, 0.39], B_9:[0.2, 0.25], B_{10}:[0.3, 0.5], B_{11}: [0.31, 0.36], B_{12}: [0.2, 0.5]
C2	$B_l \leq \frac{\text{count(ageGroup=1}\wedge\text{label=1)}}{\text{count(ageGroup=1)}} - \frac{\text{count(ageGroup=2}\wedge\text{label=1)}}{\text{count(ageGroup=2)}} \leq B_u$	B_1:[0.44, 0.5], B_2:[0.25, 0.5], B_3:[0.42, 0.5], B_4:[0.35, 0.5], B_5:[0.5, 0.6]
C3	$B_l \leq \frac{\text{count(sex=1}\wedge\text{PINCP}\geq 20k)}{\text{count(sex=1)}} - \frac{\text{count(sex=2}\wedge\text{PINCP}\geq 20k)}{\text{count(sex=2)}} \leq B_u$	B_1:[0.34, 0.39], B_2:[0.3, 0.5], B_3:[0.44, 0.5], B_4:[0.25, 0.5], B_5:[0.2, 0.25], B_6:[0.15, 0.4], B_7:[0.23, 0.25], B_8:[0.22, 0.4], B_9:[0.34, 0.36], B_{10}:[0.34, 0.35], B_{11}:[0.35, 0.36], B_{12}:[0.39, 0.4], B_{13}:[0.42, 0.44]
C4	$B_l \leq \frac{\text{count(RACE=1}\wedge\text{PINCP}\geq 15k)}{\text{count(RACE=1)}} - \frac{\text{count(RACE=2}\wedge\text{PINCP}\geq 15k)}{\text{count(RACE=2)}} \leq B_u$	B_1:[0.34, 0.39], B_2:[0.3, 0.5], B_3:[0.44, 0.5], B_4:[0.25, 0.5], B_5:[0.2, 0.25], B_6:[0.37, 0.39], B_7:[0.38, 0.39], B_8:[0.34, 0.35], B_9:[0.34, 0.39], B_{10}:[0.3, 0.5], B_{11}:[0.23, 0.25], B_{12}:[0.34, 0.36], B_{13}:[0.3, 0.33], B_{14}:[0.32, 0.35]
C5	$B_l \leq \frac{\text{sum(p\_retailprice)}}{\text{count(p\_retailprice)}} - \frac{\text{sum(s\_acctbal)}}{\text{count(s\_acctbal)}}$	B_1: 0.0014, B_2: 0.0025, B_3: 0.005, B_4: 0.01, B_5: 0.026
C6 Erica [19]	$\{\text{count(race)} = \text{race1}\} \leq B_u$, $\{\text{count(age)} = \text{group1}\} \leq B_u$	{300, 170}
C7 Erica [19]	$\{\text{count(race)} = \text{race1}\} \leq B_u$, $\{\text{count(age)} = \text{group1}\} \leq B_u$, $\{\text{count(age)} = \text{group3}\} \leq B_u$	{300, 170,250}
C8 Erica [19]	$\{\text{count(Sex)} = \text{Female}\} \leq B$, $\{\text{count(Race)} = \text{Black}\} \leq B$, $\{\text{count(Marital)} = \text{Divorced}\} \leq B$	B1 = {30, 150, 10}, B2 = {30, 300, 25}, B3 = {10, 650, 50}, B4 = {15, 200, 15}



(a) Healthcare dataset. **(b) ACSIncome dataset.**

Figure 10: Exploration distance comparison of our proposed algorithms.

Table 7: Queries for Experimentation

Dataset	ID	Query	Source
Healthcare	Q1	`SELECT * FROM Healthcare WHERE income >= 200K and num-children >= 3 and county <= 3`	Erica [19]
	Q2	`SELECT * FROM Healthcare WHERE income <= 100K and complications >= 5 and num-children >= 4`	Erica [19]
	Q3	`SELECT * FROM Healthcare WHERE income >= 300K and complications >= 5 and county == 1`	Generated
Adult-CI	Q4	`SELECT * FROM ACSIncome WHERE working_hours >= 40 and Educational_attainment >= 19 and Class_of_worker >= 3`	Erica [19]
	Q5	`SELECT * FROM ACSIncome WHERE working_hours <= 40 and Educational_attainment <= 19 and Class_of_worker <= 4`	Generated
	Q6	`SELECT * FROM ACSIncome WHERE Age >= 35 and Class_of_worker >= 2 and Educational_attainment <= 15`	Generated
TPC-H	Q7	`SELECT * FROM part, supplier, partsupp, nation, region WHERE p_partkey = ps_partkey and s_suppkey = ps_suppkey and s_nationkey = n_nationkey and n_regionkey=r_regionkey and p_size >= 10 and p_type in ('LARGE_BRUSHED') and r_name in ('EUROPE')`	Generated inspired by TPC-H's Q2