Efficient Query Repair for Aggregate 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. For instance, a minimum percentage of interview candidates selected based on their qualifications should be female. These requirements can be expressed as constraints over 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 introduce a novel query repair technique that leverages bounds on sets of 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 Aggregate 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

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, a query result *in toto* often has to fulfill additional constraints, e.g. 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. Thus, the query has to be repaired such that the fixed query satisfies all constraints for the entire result set. Prior work in this area, including query-based explanations [10, 28] and repairs [7] for missing answers, work on answering why-not questions [5, 10] as well as query refinement / relaxation approaches [21, 24, 29] determine why specific tuples are not in the query's result or how to fix the 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 query results adhere to legal and ethical regulations, such as fairness. Typically, it is challenging to express such constraints through the selection conditions of a query. Consider the following example.

Example 1 (Fairness Motivating Example). *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 ML 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 ≥ 33 AND GPA ≥ 3.80
```

Aggregate Constraint. *The employer wants 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) [4, 22] 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`*). We use* $\mathbf{count}(\theta)$ *to denote the number of query result tuples satisfying a given condition θ. For example,* $\mathbf{count}(G = M \wedge Y = 1)$ *counts the number of tuples where the gender is male and the label is positive.*

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

The employer would like to ensure that the SPD between male and female is below 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 $SPD \leq 0.2$ 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 [21] only considers cardinality constraints which cannot express SPD.*

Example 2 (Company Product Management). *A retail company aims to support inventory planning by retrieving data on 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
```


Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

```
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND p_type = 'LARGE BRUSHED'
AND r_name = 'EUROPE'
```

Aggregate Constraint. *In order to minimize the impact of supply change disruption, the company wants only a certain amount of expected revenue to be from countries with import/export issues. The constraint requires that products from UK contribute less than 10% of the total revenue of the result set in order to minimize supply chain disruptions. Formally, the constraint is defined as follows:*

$$\frac{\sum Revenue_{ProductsSelectedFromUK}}{\sum Revenue_{Selected\ Products}} \leq 0.1$$

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.

In this work, we model constraints on the 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 (AC), we would like the system to fix the violation by *repairing* the query by adjusting the selection conditions of the query, similar to [19, 24]. 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 [21], we consider constraints that can be non-monotone which invalidates most of the optimizations considered in prior work.

Consider a user query with a conjunctive selection condition $\bigwedge_{i=1}^{m} a_i \text{ op } c_i$ where a_i is an attribute, c_i is a constant, and op is a comparison operator, e.g., $<$. 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 constants c'_1 to c'_m representing an updated condition $\bigwedge_{i=1}^{m} a_i \text{ op } c'_i$. The candidate repairs are then sorted based on their distance to the user query and 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 general.

Nonetheless, we identify two opportunities for optimizing this process: (i) when two repair candidates are similar (in terms of the constants they use in selection conditions), then typically there will be overlap between the aggregate constraint computations for 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 [12], 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 are repairs.

To exploit (i), we use a kd-tree [6] to partition the input dataset. For each cluster (node in the kd-tree) we materialize 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 [26, 33]). 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 kd-tree). We refer to this approach 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 the cluster fulfill the condition of the repair candidate and can skip any clusters that do not contain any tuples fulfilling the conditions.

For example, based on Example 1, consider the kd-tree in Figure 1(b) which partitions the input dataset R in Figure 1(a) into a set of clusters C_1 to C_9. Note that we have simplified Q_1 from Example 1 by considering only a single condition, $TestScore(T) \geq 33$. The user wants to modify this condition to satisfy the constraint $\omega_{\#} = SPD$ from Example 1. Consider cluster C_2 in Figure 1(b), where the attribute *TestScore* (T) has values bounded by the range $[27, 34]$. For a repair candidate with a condition $T \geq 37$ the entire cluster can be skipped as no tuple in the cluster can fulfill the condition. In contrast, with a repair candidate with the condition $T \geq 30$, all tuples in C_3 are guarenteed to satisfy the condition, since the values of the T attribute are bounded within $[31, 37]$, as illustrated in Figure 1(b). This allows us to use the materialized aggregation results of the cluster to compute the result for the repair candidate.

We extend this idea to bound the aggregation constraint result for sets of repair candidates at once to exploit observation (ii). We represent a set of candidates through intervals of values for each c_i for which the user query contains a predicate $a_i \text{ op } c_i$, e.g., $c_i \in [33, 37]$ as shown in Figure 1(g) for *Cluster Range Pruning (RP)*. We again reason about whether all / none of the tuples in a cluster will fulfill the condition for *every* repair candidate whose c_i values are within the bounds (e.g., $[33, 37]$ in our example). The result will be valid bounds on the aggregation constraint result for any candidate repair with constants within the bounds for the candidate set. We then exploit such bounds to determine all repair candidates within such bounds are repairs or none of them are. If the result is inconclusive, we can partition such a set of repair candidates into multiple smaller sets and apply the same approach to these sets. 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. In summary, (i) FF reuses materialized aggregation results to avoid unnecessary computations, and (ii) RP leverages interval arithmetic to evaluate sets of repair candidates at once, further reducing computational cost.

We make the following contributions in this work:

- A formal definition of query repair under constraints involving arithmetic combinations of aggregate functions in Section 2.

- In addition to the brute force method, we present two algorithms, FF in Section 3 and RP in Section 4, that solve the aggregate constraint repair problem.
- A extensive experimental evaluation over multiple datasets, queries and constraints in Section 5. We compare against the current state of the art [21], 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. Our results demonstrate the effectiveness of RP.

2 Problem Definition

In this work, we consider a dataset $D = R_1, \cdots, R_z$ consisting of one or more relations R_i.

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

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

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

Aggregate Constraints (AC). The user specifies requirements on the result of their query as an 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. These queries are evaluated over the user query's result $Q(D)$. An aggregate constraint ω is of the form:

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

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

The aggregate constraints we use in this work are not monotone in general. 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 [21] 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 [9, 21, 31]. A constraint $\omega := \tau \, \mathrm{op} \, \Phi$ may be non-monotone if (i) it contains a non-monotone arithmetic operator like division or subtraction, (ii) it uses a non-monotone aggregation function, e.g., **sum** over the integers $\mathbb{Z}$, or (iii) 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: $\theta = \theta_1 \wedge \ldots \wedge \theta_m$ where each θ_i is a comparison of the form $a_i \, \mathrm{op} \, c_i$. 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: $\theta' = \theta_1' \wedge \ldots \wedge \theta_m'$ where θ_i' is a condition $a_i \, \mathrm{op} \, c_i'$. 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 original query. 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. We define a distance metric between repair candidates based on their selection conditions. Consider the user query Q with selection condition $\theta_1 \wedge \ldots \wedge \theta_m$ and repair Q_{fix} with selection condition $\theta_1' \wedge \ldots \wedge \theta_m'$. Then the distance $d(Q, Q_{fix})$ is defined as:

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

where the distance between two predicates $\theta_i = a_i \, \mathrm{op} \, c_i$ and $\theta_i' = a_i \, \mathrm{op} \, c_i'$ 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 3. *For the use case in Example 1, the repair candidate with conditions* `Major` $=$ *EE,* `Testscore` ≥ 33, *and* `GPA` ≥ 3.9 *is more similar to the user query than the candidate with conditions* `Major` $=$ *EE,* $Testscore \geq 37$, *and* $GPA \geq 3.85$ *based on our distance metric. For the first candidate, the distance is* $1 + \frac{33-33}{33} + \frac{3.9-3.8}{3.8} = 1.026$ *while for the second candidate it is* 1.134.

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**: $\mathrm{topk}_{Q_{fix}} \, d(Q, Q_{fix})$

We focus on general solutions that also work for non-monotone constraints as they are more challenging. In a practical solution, we may first detect if a constraint is monotone and then apply existing optimizations [21] if it is.

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 $a_i \, \mathrm{op}_i \, c_i$ 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 $[c_1', \ldots, c_m']$. We will 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 number of candidate repairs depends on which comparison operators are used, e.g., for $\leq$ there are at most $N_i + 1$ possible values that lead to a different result in terms of which of the input tuples will fulfill the condition. To see why this is the case assume that the values in a_i sorted based on $\leq$ are $a_1, \ldots, a_p$. Then for any constant c, the condition $a_i \leq c$ includes tuples with values in $\{a_i \mid a_i \leq c\}$ and this filtered set of a_i values is always a prefix of $a_1, \ldots, a_p$. Thus, there are $N_i + 1 = p + 1$ for choosing the length of this sequence (0 to p). 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.

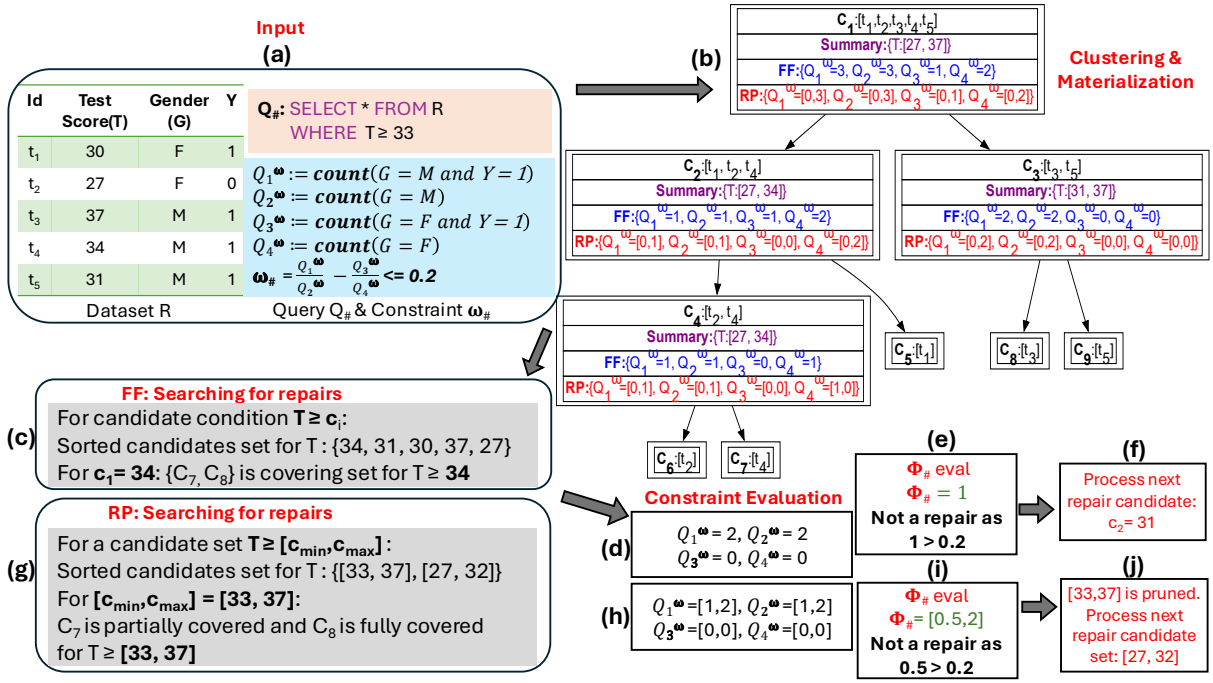


Figure 1: Overview of Query Repair with aggregate constraint approach, with example data.

3 The Full Cluster Filtering Algorithm

We now present *Full Cluster Filtering (FF)*, our first algorithm for the aggregate constraint repair problem that materializes results of each aggregate-filter query Q_i^ω for subsets of the input database D and combines these aggregation results to compute the result of Q_i^ω for a repair candidate Q_{fix} and then use it to evaluate the aggregate constraint (AC) ω,for Q_{fix}. Figure 1 shows the example of applying this algorithm: (b) building a kd-tree and materializing statistics, (c) searching for candidate repairs, and (d)-(e) evaluating constraints for repair candidates.

3.1 Clustering and Materializing Aggregations

For ease of presentation, we consider a database consisting of a single table R from now on. However, our approach can be generalized to queries involving joins by materializing the join output and treating it as a single table. As repairs only change the selection conditions of the user query, there is no need to reevaluate joins when checking repairs. We use a kd-tree to partition R into subsets (*clusters*) based on attributes that appear in the selection condition (θ) of the user query. The rationale is that the selection conditions of a repair candidate filter data along these attributes.

To evaluate the AC ω for a candidate $Q_{fix} = [c'_1, \ldots, c'_m]$, we determine a set of clusters (nodes in the kd-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 results of the aggregation queries Q_i^ω used in ω for $Q_{fix}(D)$. 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 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^ω in constraint ω, we store $Q^\omega(C)$.

An example kd-tree is shown in Figure 1(b). The user query filters on attribute $TestScore(T)$. The root of the kd-tree represents the full dataset. At each level, the clusters from the previous level are split into a number of sub-clusters (this is a configuration parameter $\mathcal{B}$ called the branching factor), 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 T. For cluster C_2 containing three tuples t_1, t_2, and t_4, we have $\text{BOUNDS}_T = [27, 34]$ as the lowest T value is 27 (from tuple t_2) and the highest value is 34 (tuple t_4). The value of $Q_2^\omega = \textbf{count}(Gender(G) = M)$ for C_2 is 1 as there is one male in the cluster. Consider a repair candidate with the condition $T \geq 37$. Based on the bounds $\text{BOUNDS}_T = [27, 34]$, we know that none of the tuples satisfy this condition. Thus, this cluster and the whole subtree rooted at the cluster can be ignored for computing the AC $\omega_\#$ for the candidate.

For ease of presentation we assume that the leaf nodes of the kd-tree contain a single tuple each. As this would lead to very large trees, in our implementation we do not further divide clusters C if that contain less tuples than a threshold $\mathcal{S}$ ($|C| \leq \mathcal{S}$). We refer to this parameter as the *bucket size*.

3.2 Constraint Evaluation for Candidates

The FF algorithm (Algorithm 1) takes as input the condition θ' of a repair candidate, the root node of the kd-tree C_{root}, and returns

Algorithm 1 CoveringClusters

Input: kd-tree with root C_{root}, condition $\theta' = \theta'_1 \wedge \ldots \wedge \theta'_m$, relation R.
Output: Set of clusters $\mathbf{C}$ such that $\bigcup_{C \in \mathbf{C}} C = \sigma_{\theta'}(R)$.

1: $stack \leftarrow [C_{root}]$
2: $\mathbf{C} \leftarrow \emptyset$ ▷ *Initialize result set*
3: **while** $stack \neq \emptyset$ **do**
4: $\quad C_{cur} \leftarrow \mathbf{pop}(stack)1$
5: $\quad in \leftarrow \mathbf{true}, notin \leftarrow \mathbf{false}$
6: $\quad$ **for all** $\theta'_i = (a_i \text{ op}_i c'_i) \in \theta'$ **do**
7: $\quad\quad in \leftarrow in \wedge \mathbf{eval}_\forall(\theta'_i, \text{BOUNDS}_{a_i}(C_{cur}))$ ▷ *All tuples fulfill θ'_i?*
8: $\quad\quad notin \leftarrow notin \vee \mathbf{eval}_\forall(\neg\theta'_i, \text{BOUNDS}_{a_i}(C_{cur}))$
9: $\quad$ **if** in **then** ▷ *All tuple in C fulfill θ'*
10: $\quad\quad \mathbf{C} \leftarrow \mathbf{C} \cup \{C_{cur}\}$
11: $\quad$ **else if** $\neg\, notin$ **then** ▷ *Some tuples in C may fulfill θ'*
12: $\quad\quad$ **for all** $C \in \mathbf{children}(C_{cur})$ **do** ▷ *Process children*
13: $\quad\quad\quad stack \leftarrow stack \cup \{C\}$
14: **return** $\mathbf{C}$

a set of disjoint clusters $\mathbf{C}$ such that the union of these clusters is precisely the subset of the relation R that fulfills θ':

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

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

3.2.1 Determining a Covering Set of Clusters The algorithm maintains a *stack* of clusters to be examined that is initialized with the root cluster C_{root} (line 1). It then processes one cluster at a time until a set of clusters $\mathbf{C}$ fulfilling Equation (1) has been determined (lines 3-14). For each cluster C, we distinguish 3 cases (lines 6-8): (i) we can use the bounds on the selection attributes recorded for the cluster to show that all tuples in the cluster fulfill θ', i.e., $\sigma_{\theta'}(C) = C$ (line 7). In this case, the cluster will be added to $\mathbf{C}$ (lines 9-10); (ii) based on the bounds, we can determine that none of the tuples in the cluster fulfill the condition (line 8). Then this cluster can be ignored; (iii) either a non-empty subset of C fulfills θ' or based on the bounds $\text{BOUNDS}_{a_i}(C)$ we cannot demonstrate that $\sigma_{\theta'}(C) = \emptyset$ or $\sigma_{\theta'}(C) = C$ hold. In this case, we add the children of C to the stack to be evaluated in future iterations (lines 11-13). The algorithm uses the function $\mathbf{eval}_\forall$ 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) holds by applying $\mathbf{eval}_\forall$ 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 is *fully covered* (case (i)) if $\mathbf{eval}_\forall$ returns true for all θ'_i and *not covered at all* (case (ii)) if $\mathbf{eval}_\forall$ returns true for at least one comparison $\neg\theta'_i$.

3.2.2 Determining Coverage In Table 1, we define the function $\mathbf{eval}_\forall$ which takes a condition $a \text{ op } c$ and bounds $\text{BOUNDS}_a(C)$ for attribute a in cluster C and returns true if it is guaranteed that all tuples $t \in C$ fulfill the condition. Ignore $\mathbf{reval}_\forall$ for now, this function will be used in Section 4. An inequality $>$ (or $\geq$) is true for all tuples if the lower bound $\underline{a}$ of a is larger (larger equal) than the threshold c. The case for $<$ and $\leq$ is symmetric: the upper bound $\overline{a}$ has to be smaller (smaller equals) than c. For an equality, we can only guarantee that the condition is true if $\underline{a} = \overline{a} = c$. For $\neq$, all tuples fulfill the inequality if c does not belong to the interval $[\underline{a}, \overline{a}]$.

Table 1: Given the bounds $[\underline{a}, \overline{a}]$ for the attribute a of a condition $a \text{ op } c$ or $a \in [c_1, c_2]$, function $\text{eval}_\forall$ does return true if the condition evaluates to true for all values in $[\underline{a}, \overline{a}]$. For RP, we consider a range $[\underline{c}, \overline{c}]$ (corresponding to a set of candidates) or two ranges $[\underline{c_1}, \overline{c_1}]$ and $[\underline{c_2}, \overline{c_2}]$ for operator $\in$. $\text{reval}_\forall$ determines whether for every $c \in [\underline{c}, \overline{c}]$, the condition is guaranteed to evaluate to true for every $a \in [\underline{a}, \overline{a}]$ while $\text{reval}_\exists$ determines whether for some $c \in [\underline{c}, \overline{c}]$, the condition may evaluate to true for $a \in [\underline{a}, \overline{a}]$.

Op.	$\text{eval}_\forall$	$\text{reval}_\forall$	$\text{reval}_\exists$
$>, \geq$	$\underline{a} > c, \quad \underline{a} \geq c$	$\underline{a} > \overline{c}, \quad \underline{a} \geq \overline{c}$	$\overline{a} > \underline{c}, \quad \overline{a} \geq \underline{c}$
$<, \leq$	$\overline{a} < c, \quad \overline{a} \leq c$	$\overline{a} < \underline{c}, \quad \overline{a} \leq \underline{c}$	$\underline{a} < \overline{c}, \quad \underline{a} \leq \overline{c}$
$=$	$\underline{a} = \overline{a} = c$	$\underline{a} = \underline{c} = \overline{a} = \overline{c}$	$[\underline{a}, \overline{a}] \cap [\underline{c}, \overline{c}] \neq \emptyset$
$\neq$	$c \notin [\underline{a}, \overline{a}]$	$[\underline{a}, \overline{a}] \cap [\underline{c}, \overline{c}] = \emptyset$	$\neg(\underline{a} = \underline{c} = \overline{c} = \overline{a})$
$\in [c_1, c_2]$	$c_1 \leq \underline{a} \wedge \overline{a} \leq c_2$	$\overline{c_1} \leq \underline{a} \wedge \overline{a} \leq \underline{c_2}$	$[\underline{a}, \overline{a}] \cap [\underline{c_1}, \overline{c_2}] \neq \emptyset$

For the running example in Figure 1, consider a repair candidate with the condition $T \geq 34$, where $c_1 = 34$. The algorithm maintains a stack of clusters initialized to $[C_1]$, the root node of the kd-tree. In each iteration it takes on cluster form the stack. The root cluster C_1, has $\text{BOUNDS}_T(C_1) = [27, 37]$. The algorithm evaluates whether all or none of the tuples satisfy the condition. Since it neither is the case, we proceed to the children of C_1: C_2 and C_3. The same situation occurs for C_2 and C_3 leading to further exploration of their child $\{C_4$ and $C_5\}$ for C_2 and $\{C_8$ and $C_9\}$ for C_3. Since the coverage for C_4 cannot be determined, the algorithm proceeds to process C_6 and C_7. Clusters C_5, C_6 and C_9 are determined to not satisfy the condition while C_7 and C_8 are confirmed to meet the condition and are added to $\mathbf{C}$. In this example, we had to explore all of the leaf clusters, but often we will be able to prune or confirm clusters covering multiple tuples. For instance, for $T \geq 37$, C_2 with bounds $[27, 34]$ with all of its descendents can be skipped as $T \geq 37$ is false for any $T \in [27, 34]$.

3.2.3 Constraint Evaluation After identifying the covering set of clusters $\mathbf{C}$ for a repair candidate Q_{fix}, our approach evaluates the AC ω over $\mathbf{C}$. Recall that for each cluster C we materialize the result of each filter aggregate query Q_i^ω used in ω. For aggregate function **avg** that is not decomposable, we apply the standard approach of storing **count** and **sum** instead. We then compute $Q_i^\omega(Q(D))$ over the materialized aggregation results for the clusters. Concretely, for such an aggregate query $Q^\omega := \gamma_{\mathbf{f}(a)}(\sigma_{\theta'}(Q(D)))$ we compute its result as follows using $\mathbf{C}$:

$$\gamma_{\mathbf{f}'(a)}\left(\bigcup_{C \in \mathbf{C}} \{Q^\omega(C)\}\right)$$

Here $\mathbf{f}'$ is the function we use to merge aggregation results for multiple subsets of the database. This function depends on $\mathbf{f}$, e.g., for both **count** and **sum** we have $\mathbf{f}' = \mathbf{sum}$, for **min** we use $\mathbf{f}' = \mathbf{min}$, and for **max** we use $\mathbf{f}' = \mathbf{max}$. We then substitute these aggregation results into ω and evaluate the resulting expression to determine whether Q_{fix} fulfills the constraints and is a repair or not.

In the example from Figure 1(c), the covering set of clusters for the repair candidate with $c_1 = 34$ is $\mathbf{C} = \{C_7, C_8\}$. Evaluating $Q_1^\omega = \mathbf{count}(Gender(G) = M \wedge Y = 1)$ over $\mathbf{C}$, we sum the counts:

$Q_1^\omega = Q_{1C_7}^\omega + Q_{1C_8}^\omega = 1+1 = 2$. Similarly, $Q_2^\omega = Q_{2C_7}^\omega + Q_{2C_8}^\omega = 1+1 = 2$, $Q_3^\omega = Q_{3C_7}^\omega + Q_{3C_8}^\omega = 0+0 = 0$, $Q_4^\omega = Q_{4C_7}^\omega + Q_{4C8}^\omega = 0+0 = 0$ as shown in Figure 1(d). Substituting these values into $\omega_\#$, we obtain $1 \leq 0.2 =$ **false** as shown in Figure 1(e). Since the candidate $T \geq 34$ does not satisfy the constraint it is not a valid repair.

3.3 Computing Top-k Repairs

To compute the top-k repairs, we enumerate all repair candidates in increasing order of their distance to the user query using the distance measure from Section 2. For each candidate Q_{fix} we apply the FF to determine a covering clusterset, evaluate the constraint ω, and output Q_{fix} if it fulfills the constraint. Once we have found k results, the algorithm terminates.

4 Cluster Range Pruning (RP)

While algorithm FF reduces the effort needed to evaluate aggregation constraints for repair candidates, it has the drawback that we still have to evaluate each repair candidate individually. We now present an enhanced approach that reasons about sets of repair candidates. For a user query condition $\theta_1 \wedge \ldots \wedge \theta_m$ where $\theta_i := a_i \text{ op}_i c_i$, we use ranges of constant values instead of constants to represent such a set of repairs $\mathbb{Q}$: $[[\underline{c_1}, \overline{c_1}], \ldots, [\underline{c_m}, \overline{c_m}]]$. Such a list of ranges $\mathbb{Q}$ represents a set of a repair candidates:

$$\{[c_1, \ldots, c_m] \mid \forall i \in [1, m] : c_i \in [\underline{c_i}, \overline{c_i}]\}$$

Consider an aggregation constraint $\omega := \tau \text{ op } \Phi(Q_1^\omega, \ldots, Q_n^\omega)$. Our enhanced approach RP uses a modified version of the kd-tree from FF to compute conservative bounds of the arithmetic expression $\underline{\Phi}$ and $\overline{\Phi}$ on the possible values for Φ that hold for all repair candidates in $\mathbb{Q}$. Based on such bounds, if (i) $\tau \text{ op } c$ holds for every $c \in [\underline{\Phi}, \overline{\Phi}]$, then every $Q_{fix} \in \mathbb{Q}$ is a valid repair, if (ii) $\tau \text{ op } c$ is violated for every $c \in [\underline{\Phi}, \overline{\Phi}]$, then no $Q_{fix} \in \mathbb{Q}$ is a valid repair and we can skip the whole set. Otherwise, (iii) there may or may not exist some candidates in $\mathbb{Q}$ that are repairs. In this case, our algorithm partitions $\mathbb{Q}$ into multiple subsets and applies the same test to each partition. In the following we introduce our algorithm that utilizes such repair candidate sets and bounds on the aggregate constraint results and then explain how to use the kd-tree to compute such bounds.

4.1 Computing Top-k Repairs

RP (Algorithm 2) takes as input a kd-tree with root C_{root}, a user query condition θ, a AC ω, a candidate set $\mathbb{Q} = [[\underline{c_1}, \overline{c_1}], \ldots, [\underline{c_m}, \overline{c_m}]]$, and user query Q and returns the set of top-k repairs $\mathcal{Q}_{top-k}$.

The algorithm maintains three priority queues: (i) $\mathcal{Q}_{top-k}$ is a queue of individual repairs that eventually will store the top-k repairs. This queue is sorted on $d(Q, Q_{fix})$ where Q_{fix} is a repair in the queue; (ii) *rcand* is a queue where each element is a repair candidate set $\mathbb{Q}$ encoded as ranges as shown above. For each $\mathbb{Q}$ we have established that for all $Q_{fix} \in \mathbb{Q}$, Q_{fix} is a repair. This query is sorted on the lower bound $\underline{d(Q, Q_{fix} \in \mathbb{Q})}$ of the distance of any repair in $\mathbb{Q}$ to the user query. Finally, (iii) *queue* is a queue where each element is a repair candidate set $\mathbb{Q}$. This queue is also sorted on $\underline{d(Q, Q_{fix} \in \mathbb{Q})}$. In each iteration of the main loop of the algorithm, one repair candidate set from *queue* is processed.

Algorithm 2 Top-k Repairs w. Range-based Pruning of Candidates

Input: kd-tree with root C_{root}, constraint AC ω, repair candidate set $\mathbb{Q} = [[\underline{c_1}, \overline{c_1}], \ldots, [\underline{c_m}, \overline{c_m}]]$, user query condition $\theta = \theta_1 \wedge \ldots \wedge \theta_m$, user query Q
Output: Top-k repairs $\mathcal{Q}_{top-k}$

1: $\mathcal{Q}_{top-k} \leftarrow \emptyset$ ▷ *Queue of repairs Q' sorted on $d(Q, Q')$*
2: $rcand \leftarrow \emptyset$ ▷ *Queue of repair sets $\mathbb{Q}'$ sorted on $\underline{d(Q, \mathbb{Q}')}$*
3: $queue \leftarrow [\mathbb{Q}]$ ▷ *Queue of repair candidate sets $\mathbb{Q}'$ sorted on $\underline{d(Q, \mathbb{Q}')}$*
4: **while** $queue \neq \emptyset$ **do**
5: $\quad \mathbb{Q}_{cur} \leftarrow \text{POP}(queue)$
6: $\quad \mathbb{Q}_{next} \leftarrow \text{PEEK}(queue)$ ▷ *Peek at next item in queue*
7: $\quad (\mathbf{C}_{full}, \mathbf{C}_{partial}) \leftarrow \text{CoveringClusterSet}(\mathbb{Q}_{cur}, C_{root}, \theta)$
8: $\quad$ **if** $\text{ACEVAL}_\forall(\omega, \mathbf{C}_{full}, \mathbf{C}_{partial})$ **then** ▷ *All $Q' \in \mathbb{Q}_{cur}$ are repairs?*
9: $\quad\quad rcand \leftarrow \text{INSERT}(rcand, \mathbb{Q}_{cur})$
10: $\quad$ **else if** $\text{ACEVAL}_\exists(\omega, \mathbf{C}_{full}, \mathbf{C}_{partial})$ **then** ▷ *Some repairs?*
11: $\quad\quad$ **for** $\mathbb{Q}_{new} \in \text{RangeDivide}(\mathbb{Q}_{cur})$ **do** ▷ *divide ranges*
12: $\quad\quad\quad$ **if** $\text{hasCandidates}(\mathbb{Q}_{new})$ **then**
13: $\quad\quad\quad\quad queue \leftarrow \text{INSERT}(queue, \mathbb{Q}_{new})$
14: $\quad \mathcal{Q}_{top-k} \leftarrow \text{topkConcreteCand}(rcand, k)$ ▷ *Top k repairs*
15: $\quad$ **if** $|\mathcal{Q}_{top-k}| \geq k$ **then** ▷ *Have k repairs?*
16: $\quad\quad$ **if** $\underline{d(Q, \mathbb{Q}_{next})} > d(Q, \mathcal{Q}_{top-k}[k])$ **then** ▷ *Rest inferior?*
17: $\quad\quad\quad$ **break**
18: **return** $\mathcal{Q}_{top-k}$

The algorithm initializes *queue* to the input parameter repair candidate set $\mathbb{Q}$. We call the algorithm with a repair candidate set that covers the whole search space (line 1-3). The algorithm's main loop processes one repair candidate $\mathbb{Q}_{cur}$ at a time (line 5) while keeping track of the next candidate $\mathbb{Q}_{next}$ (line 6) until a set of top-k repairs fulfilling AC ω has been determined (lines 4–17). For the current repair candidate set $\mathbb{Q}_{cur}$, we use function CoveringClusterSet (Algorithm 3) to determine two sets of clusters $\mathbf{C}_{full}$ and $\mathbf{C}_{partial}$ (line 7). For every cluster $C \in \mathbf{C}_{full}$, all tuples in C fulfill the condition of every repair candidate $Q_{fix} \in \mathbb{Q}_{cur}$ and for every cluster $C \in \mathbf{C}_{partial}$, there may exist some tuples in C such that for some repair candidates $Q_{fix} \in \mathbb{Q}_{cur}$, the tuples fulfill the condition of Q_{fix}. We use these two sets of clusters to determine bounds on the arithmetic expression $[\underline{\Phi}, \overline{\Phi}]$ of the AC ω. The algorithm then distinguishes between three cases (line 8-13): (i) function $\text{ACEVAL}_\forall$ uses $\mathbf{C}_{full}$ and $\mathbf{C}_{partial}$ to determine whether ω is guaranteed to hold for every $Q_{fix} \in \mathbb{Q}_{cur}$. For that we compute bounds $[\underline{\Phi}, \overline{\Phi}]$ on Φ that hold for every $Q_{fix} \in \mathbb{Q}_{cur}$. If this is the case then all $Q_{fix} \in \mathbb{Q}_{cur}$ are repairs and we add $\mathbb{Q}_{cur}$ to *rcand* (lines 8–9); (ii) function $\text{ACEVAL}_\exists$ determines $\mathbf{C}_{full}$ and $\mathbf{C}_{partial}$ to check whether some repair candidates $Q_{fix} \in \mathbb{Q}_{cur}$ may fulfill the AC and needs to be further examined (lines 10–13); (iii) if both $\text{ACEVAL}_\forall$ and $\text{ACEVAL}_\exists$ return false, then it is guaranteed that no $Q_{fix} \in \mathbb{Q}_{cur}$ is a repair and we can discard $\mathbb{Q}_{cur}$. We will discuss these functions in depth in Section 4.3.

For example, if $\omega := 0.7 \leq \Phi$ and we compute bounds $[\underline{\Phi}, \overline{\Phi}] = [0.5, 1]$ that hold for all $Q_{fix} \in \mathbb{Q}_{cur}$, then $\text{ACEVAL}_\forall$ returns false as some $Q_{fix} \in \mathbb{Q}_{cur}$ may not fulfill the constraint. However, $\text{ACEVAL}_\exists$ return true as some $Q_{fix} \in \mathbb{Q}_{cur}$ may fulfill the constraint. In this case, the algorithm partitions $\mathbb{Q}_{cur}$ into smaller sub-ranges $\mathbb{Q}_{new}$ using the function $\text{RangeDivide}(\mathbb{Q}_{cur})$ (line 11). Assume

that $\mathbb{Q}_{cur} = [[\underline{c_1}, \overline{c_1}], \ldots, [\underline{c_m}, \overline{c_m}]]$. RangeDivide splits each range $[\underline{c_i}, \overline{c_i}]$ into a fixed number of fragments $\{[\underline{c_{i_1}}, \overline{c_{i_1}}], \ldots, [\underline{c_{i_l}}, \overline{c_{i_l}}]\}$ such that each $[\underline{c_{i_j}}, \overline{c_{i_j}}]$ is roughly of the same size and returns the following set of repair candidate sets:

$$\{[[\underline{c_{1_{j_1}}}, \overline{c_{i_{j_1}}}], \ldots, [\underline{c_{m_{j_m}}}, \overline{c_{m_{j_m}}}]] \mid [j_1, \ldots, j_m] \in [1, l]^m\}$$

That is, each $\mathbb{Q}_{new}$ has one of the fragments for each $[\underline{c_i}, \overline{c_i}]$ and the union of all repair candidates in these repair candidate sets is $\mathbb{Q}_{cur}$. We use $l = 2$ in our implementation. The function hasCandidates (line 12-13) checks whether each range in $\mathbb{Q}_{new}$ contains at least one value that exists in the data. This restricts the search space to only include candidates that actually appear in the data. Recall from our discussion of the search space at the end of Section 2 that we only consider values from the active domain of an attribute as constants for repair candidates. That is, we can skip candidate repair sets $\mathbb{Q}_{new}$ that do not contain any such values. For example, if the dataset contains only values 8 and 10 for a given attribute, then applying a filter $a \leq 9$ would yield the same result as $a \leq 8$, since no data points lie between 8 and 10. If this condition is satisfied, $\mathbb{Q}_{new}$ is inserted into the priority queue *queue* to be processed in future iterations of the main loop. In each iteration we use function topkConcreteCand (line 14) to determine the k repairs Q_i across all $\mathbb{Q} \in rcand$ with the lowest distance to the user query Q. If we can find k such candidates (line 15), then we test whether no repair candidate from the next repair candidate set $\mathbb{Q}_{next}$ may be closer to Q then the kth candidate $\boldsymbol{Q}_{top-k}[k]$ from $\boldsymbol{Q}_{top-k}$ (line 16). This is the case if the lower bound on the distance of any candidate in $\mathbb{Q}_{next}$ is larger than the distance of $\boldsymbol{Q}_{top-k}[k]$. Furthermore, the same holds for all the remaining repair candidate sets in *rcand*, because *rcand* is sorted on the lower bound of the distance to the user query. That is, $\boldsymbol{Q}_{top-k}$ contains exactly the top-k repairs and the algorithm returns $\boldsymbol{Q}_{top-k}$.

4.2 Determining Covering Cluster Sets

Similar to FF, we can use the kd-tree to determine a covering cluster set $\mathbf{C}$. However, as we now deal with a set of candidate repairs $\mathbb{Q}$, we would have to find a $\mathbf{C}$ such that for all $Q_{fix} \in \mathbb{Q}$ we have: $Q_{fix}(D) = \bigcup_{C \in \mathbf{C}} C$. Such a covering cluster set is unlikely to exist as for any two $Q_{fix} \neq Q'_{fix} \in \mathbb{Q}$ it is likely that $Q_{fix}(D) \neq Q'_{fix}(D)$. Instead we relax the condition and allow clusters C that are *partially covered*, i.e., for which some tuples in C may be in the result of some candidates in $\mathbb{Q}$. We modify Algorithm 1 to take a repair candidate set as an input and to return two sets of clusters: $\mathbf{C}_{full}$ which contains clusters for which all tuples fulfill the selection condition of all $Q_{fix} \in \mathbb{Q}$ and $\mathbf{C}_{partial}$ which contains clusters that are only partially covered.

Analogous to Algorithm 1, the updated algorithm (Algorithm 3) maintains a stack of clusters to be processed that is initialized with the root node of the kd-tree (line 1). In each iteration of the main loop (line 3-16), the algorithm determines whether all tuples of the current cluster C_{cur} fulfill the conditions θ_i for all repair candidates $Q_{fix} \in \mathbb{Q}$. This is done using function $\mathbf{reval}_\forall$ (line 7). Additionally, we check whether it is possible that at least one tuple fulfills the condition of at least one repair candidate $Q_{fix} \in \mathbb{Q}$. This is done using a function $\mathbf{reval}_\exists$ (line 8). If the cluster is fully covered we add it to the result set $\mathbf{C}_{full}$ (line 10). If it is partially covered, then we distinguish between two cases (line 11- 16). Either the cluster is a leaf node (line 12-13) or it is an inner node (line 14-16). If the cluster is a leaf, then we cannot further divide the cluster and add it to $\mathbf{C}_{partial}$. If the cluster is an inner node, then we process its children as we may be able to determine that some of its children are fully covered or not covered at all.

Table 1 shows how conditions are evaluated by $\mathbf{reval}_\forall$ and $\mathbf{reval}_\exists$. For a condition $a > c$, if the lower bound of attribute $\underline{a}$ is larger than the upper bound $\overline{c}$, then all tuples in the cluster fulfill the condition for all $Q_{fix} \in \mathbb{Q}$. The cluster is partially covered if $\overline{a} > \underline{c}$ as then there exists at least one value in the range of a and constant c in $[\underline{c}, \overline{c}]$ for which the condition is true.

In the example from Figure 1, a repair candidate $[[33, 37]]$ is evaluated. Recall that the single condition in this example is $T \geq c$. C_{root} has $\text{bounds}_T = [27, 37]$. The algorithm first applies $\mathbf{reval}_\forall$ to check if all tuples in C_{root} satisfy the condition. Since $27 \not\geq 37$, the algorithm proceeds to evaluate the condition for partial coverage using $\mathbf{reval}_\exists$. Since C_1 is partially covered and not a leaf, the algorithm continues by processing C_1's children, C_2 and C_3. For C_3, a similar situation occurs: the lower bound of the attribute, $\underline{a} = 31$, is not greater than the upper bound of the constant, $\overline{c} = 37$ and we have to process additional clusters, C_8 and C_9. The same holds for C_2 and we process its children: C_4 and C_5. Additionally, C_4 fails $\mathbf{reval}_\forall$ but satisfies partial coverage with $\mathbf{reval}_\exists$, necessitating evaluation of its children, C_6 and C_7. Finally, the algorithm applies $\mathbf{reval}_\forall$ and $\mathbf{reval}_\exists$ if necessary to the clusters C_5, C_6, C_7, C_8, and C_9, confirming that $C_8 \in \mathbf{C}_{full}$ and $C_7 \in \mathbf{C}_{partial}$, as $t_3.T = 37 \geq c$ is true for all $c \in [33, 37]$ and $t_4.T = 34 \geq c$ is only true for some $c \in [33, 37]$.

4.3 Computing Bounds on Constraints

Given the cluster sets $(\mathbf{C}_{full}, \mathbf{C}_{partial})$ computed by Algorithm 3, we next (i) compute bounds on the results of the aggregation queries Q_i^ω used in the constraint, then (ii) use these bounds to compute bounds $[\underline{\Phi}, \overline{\Phi}]$ on the result of the arithmetic expression Φ of the AC ω over repair candidates in $\mathbb{Q}$. These bounds are conservative in the sense that all possible results are guaranteed to be included in these bounds. Then, finally, (iii) function $\text{aceval}_\forall$ uses the computed bounds to determine whether all candidates in $\mathbb{Q}$ fulfill the constraint by applying $\mathbf{reval}_\forall$ from Table 1. For a constraint $\omega := \tau \text{ op } \Phi$, $\text{aceval}_\forall$ calls $\mathbf{reval}_\forall$ with $[\underline{\Phi}, \overline{\Phi}]$ and τ. $\text{aceval}_\exists$ uses $\mathbf{reval}_\exists$ instead to determine whether some candidates in $\mathbb{Q}$ may fulfill the constraint. This requires techniques for computing bounds on the possible results of arithmetic expressions and aggregation functions when the values of each input of the computation are known to be bounded by some interval.

4.3.1 Bounding Aggregation Results We now discuss how to compute bounds for the results of the filter-aggregation queries Q_i^ω of an aggregate constraint ω based on the cluster sets $(\mathbf{C}_{full}, \mathbf{C}_{partial})$ returned by Algorithm 3. As every cluster C in $\mathbf{C}_{full}$ is fully covered for all repair candidates in $\mathbb{Q}$, i.e., all tuples in the cluster fulfill the conditions of each $Q_{fix} \in \mathbb{Q}$, the materialized aggregation results $Q_i^\omega(C)$ of C contribute to both the lower bound $\underline{Q_i^\omega}$ and upper bound $\overline{Q_i^\omega}$ as for FF. For partially covered clusters $(\mathbf{C}_{partial})$, we have to make worst case assumptions to derive valid lower and

upper bounds. For the lower bound, we have to consider the minimum across two options: (i) no tuples from the cluster will fulfill the condition of at least one Q_{fix} in $\mathbb{Q}$. In this case, the cluster is ignored for computing the lower bound e.g., in case for **max**; (ii) based on the bounds of the input attribute for the aggregation within the cluster, there are values in the cluster that if added to the current aggregation result further lowers the result, e.g., a negative number for **sum** or a value smaller than the current minimum for **min**. For example, for $\mathbf{min}(a)$ we have to reason about two cases: (i) we can add $\underline{a}$ to the aggregation in case of negative numbers; (ii) otherwise should ignore this cluster for computing lower bounds. For **sum** we have the two cases: (i) the attribute for the aggregation has negative numbers. In this case we sum the negative numbers for the lower bound. (ii) otherwise should ignore this cluster for computing lower bounds. For the upper bound we have the symmetric two cases: (i) if including no tuples from the cluster would result in a larger aggregation result, e.g., for **sum** when all values in attribute a in the cluster are negative then including any tuple from the cluster would lower the aggregation result and (ii) if the upper bound of values for the aggregation input attribute within the cluster increases the aggregation result, we include the aggregation bounds in the computation for the upper bound.

4.3.2 Bounding Results of Arithmetic Expressions Given the bounds on aggregate-filter queries, we use *interval arithmetic* [12, 27] which computes sound bounds for the result of arithmetic operations when the inputs are bound by intervals. In our case, the bounds on the results of aggregate queries Q_i^ω are the input and bounds $[\underline{\Phi}, \overline{\Phi}]$ on Φ are the result. The notation we use is similar to [32]. Table 2 shows the definitions for arithmetic operators we support in aggregate constraints. Here, $\underline{E}$ and $\overline{E}$ denote the lower and upper bound on the values of expression E, respectively. For example, for addition the lower bound for the result of addition $\underline{E_1 + E_2}$ of two expressions E_1 and E_2 is $\underline{E_1} + \underline{E_2}$.

4.3.3 Bounding Aggregate Constraint Results Consider a constraint $\omega := \tau \,\mathrm{op}\, \Phi$. There are three possible outcomes for a repair candidate set: (i) $\tau \,\mathrm{op}\, \Phi$ is true for all $[\underline{\Phi}, \overline{\Phi}]$ which $\textsc{aceval}_\forall$ determines using $\mathbf{reval}_\forall$ and bounds $[\tau, \tau]$; (ii) some of the candidate in $\mathbb{Q}$ may fulfill the condition, which $\textsc{aceval}_\exists$ determines using $\mathbf{reval}_\exists$; (iii) none of the candidates in $\mathbb{Q}$ fulfill the condition (both (i) and (ii) are false).

In the running example from Figure 1(g), the covering set of clusters for repair candidate set $\mathbb{Q} := [[33, 37]]$ are $\mathbf{C}_{full} = \{C_8\}$ and $\mathbf{C}_{partial} = \{C_7\}$. To evaluate $Q_1^\omega = \mathbf{count}(G = M \wedge Y = 1)$ over these clusters, the algorithm include the materialized aggregation results for C_8 for both the lower bound $\underline{Q_i^\omega}$ and upper bound $\overline{Q_i^\omega}$. For the partially covered C_7, the lower bound of $Q^\omega_{1C_7}$ is 0 for this cluster (the lowest count is achieved by excluding all tuples from the cluster), while the upper bound is 1, as there exists a male in the cluster satisfying $Y = 1$. Thus, we get the following bounds for $Q^\omega_{1C_7} = [0, 1]$. Similarly, we compute the remaining aggregation bounds: $Q^\omega_{1C_8} = [1, 1]$, $Q^\omega_{2C_7} = [0, 1]$, $Q^\omega_{2C_8} = [1, 1]$, $Q^\omega_{3C_7} = [0, 0]$, $Q^\omega_{3C_8} = [0, 0]$, $Q^\omega_{4C_7} = [0, 0]$, $Q^\omega_{4C_8} = [0, 0]$.

Next, in Figure 1(h) we sum the lower and upper bounds for each aggregation Q_i^ω across all clusters in $\mathbf{C}$: $Q_1^\omega = Q^\omega_{1C_7} + Q^\omega_{1C_8} = [1, 2]$, $Q_2^\omega = Q^\omega_{2C_7} + Q^\omega_{2C_8} = [1, 2]$, $Q_3^\omega = Q^\omega_{3C_7} + Q^\omega_{3C_8} = [0, 0]$, $Q_4^\omega = Q^\omega_{4C_7} + Q^\omega_{4C_8} = [0, 0]$. We then substitute the computed values $\{Q_1^\omega, Q_2^\omega, Q_3^\omega, Q_4^\omega\}$ into $\omega_\#$ and evaluate the resulting expression using interval arithmetic (Table 2). Given: $\omega_\# = {}^{Q_1^\omega}/_{Q_2^\omega} - {}^{Q_3^\omega}/_{Q_4^\omega}$ the lower and upper bounds for the first term ${}^{Q_1^\omega}/_{Q_2^\omega}$ are computed as: $[\underline{{}^{E_1}/_{E_2}}, \quad \overline{{}^{E_1}/_{E_2}}] = [{}^1/_2, 2]$. Similarly, for the second term: ${}^{Q_3^\omega}/_{Q_4^\omega} = [0, 0]$. Applying interval arithmetic to compute the subtraction we get: $\underline{\mathbf{E_1} - \mathbf{E_2}}$, $\overline{\mathbf{E_1} - \mathbf{E_2}}$. Thus, we obtain bounds $[\underline{\Phi_\#}, \overline{\Phi_\#}] = [{}^1/_2, 2]$ (Figure 1(i)). Since $\underline{\Phi_\#} = {}^1/_2 > 0.2$, none of the candidates in $\mathbb{Q} = [[33, 37]]$ can be repairs and we can prune $\mathbb{Q}$.

Table 2: Bounds on applying an operator to the result of expressions E_1 and E_2 with interval bounds [32].

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

Algorithm 3 CoveringClusterSet

Input: kd-tree with root C_{root}, repair candidate set $\mathbb{Q} = [[\underline{c_1}, \overline{c_1}], \ldots, [\underline{c_m}, \overline{c_m}]]$, condition θ
Output: Partially covering cluster set $(\mathbf{C}_{full}, \mathbf{C}_{partial})$

```
1:  stack ← [C_root]
2:  C_full ← ∅, C_partial ← ∅   ▷ Initialize cluster sets
3:  while stack ≠ ∅ do
4:      C_cur ← pop(stack)
5:      in ← true, pin ← true
6:      for all θ_i = (a_i op_i c_i) ∈ θ do   ▷ C_cur fully / part. covered?
7:          in ← in ∧ reval_∀(θ_i, [c_i, c̄_i], BOUNDS_{a_i}(C_cur))
8:          pin ← pin ∧ reval_∃(θ_i, [c_i, c̄_i], BOUNDS_{a_i}(C_cur))
9:      if in then   ▷ Add fully covered cluster to the result
10:         C_full ← C_full ∪ {C_cur}
11:     else if pin then
12:         if isleaf(C_cur) then   ▷ Partially covered leaf cluster
13:             C_partial ← C_partial ∪ {C_cur}
14:         else   ▷ Process children of partial cluster
15:             for all C ∈ children(C_cur) do
16:                 stack ← stack ∪ {C}
17: return (C_full, C_partial)
```

5 Experiments

We evaluate the performance of our algorithms using real-world and TPC-H benchmark datasets. We start by comparing the baseline FF technique (Section 3.2) against our RP algorithm (Section 4) in Section 5.2. We then investigate the impact of several factors, such as dataset size and how similar the top-k repairs are to the user query, on performance in Section 5.3. Finally, we compare our approach with *Erica* [21], which only supports cardinality constraints over groups in the query output by generating all minimal refinements, in Section 5.4. Note that Erica uses a different optimization criterion than we do: as in skyline queries, *minimal refinements* in

Erica are repairs that where none of the predicates can be further refined and still yield a repair. In contrast we compare repairs based on a distance metric. We will explain in Section 5.4 how we achieve a fair comparison.

5.1 Experimental Setup

Datasets. We choose two real-world datasets of size of 500K, *Adult Census Income (ACSIncome)* [16] and *Healthcare dataset (Healthcare)* [17], that are commonly used to evaluate fairness. We also utilize a standard benchmark, *TPC-H* [1], to varying dataset size from 25K to 500K. We converted categorical columns into numerical data as the algorithms are designed for numerical data.

Queries. Table 3 shows the queries used in our experiments. For Healthcare, we use queries Q_1 and Q_2 from [21] and a new query Q_3. For ACSIncome, we use Q_4 from [21] and new queries 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 SPD between two demographic groups to be within a certain bound. Table 4 shows the details of the constraints used. In some experiments, we vary the bounds B_l and B_u. For a constraint ω_i we use $\omega_i^{d=p}$ to denote a variant of ω_i where the bounds have been set such that the top-k repairs are within the first p% of the repair candidates ordered by distance to the user query. In other words, an algorithm that explores the individual repair candidates in this order would have to explore the first p% of the candidate search space to find the top-k repairs. For the detailed settings see [3]. For ACSIncome, we determine the groups for SPD based on gender and race. For Healthcare, we determine demographic groups based on race and age group. For TPC-H, we enforce the constraint ω_5 as described in Example 2. We use Ω to denote a set of ACs. Ω_6 through Ω_8 are sets of cardinality constraints for comparison with Erica. While we have presented our repair methods for single ACs, the methods can be trivially extended to find repairs for a set of constraints, i.e., the repair fulfills $\bigwedge_{\omega\in\Omega}\omega$. For RP (Algorithm 2), it is sufficient to replace the condition in Line 8 with $\bigwedge_{\omega\in\Omega}\textsc{aceval}_\forall(\omega, \mathbf{C}_{full}, \mathbf{C}_{partial})$ and the condition in Line 10 with $\bigwedge_{\omega\in\Omega}\textsc{aceval}_\exists(\omega, \mathbf{C}_{full}, \mathbf{C}_{partial})$.

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 3.1. We consider two tuning parameters for the tree:

- **Branching Factor**: Each node has $\mathcal{B}$ children.
- **Bucket Size**: Parameter $\mathcal{S}$ determines the minimum number of tuples in a cluster. We do not split nodes with less than or equal to $\mathcal{S}$ tuples. When one of our algorithms reaches such a leaf node we just evaluate computations on each tuple in the cluster, e.g., to determine which tuples fulfill a condition.

We also control k, the number of repairs returned by our methods. The default setting for these parameters is as follows: $\mathcal{B} = 5$, $k = 7$, and $\mathcal{S} = 15$. The default dataset size is 50K tuples.

All 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 experiment was repeated five times and report the median runtime as the variance is low (~ 3%).

Table 3: Queries for Experimentation

```
SELECT * FROM Healthcare
Q1: WHERE income >= 200K AND num-children >= 3
      AND county <= 3
Q2: WHERE income <= 100K AND complications >= 5
      AND num-children >= 4
Q3: WHERE income >= 300K AND complications >= 5
      AND county == 1
SELECT * FROM ACSIncome
Q4: WHERE working_hours >= 40 AND Educational_attainment >= 19
      AND Class_of_worker >= 3
Q5: WHERE working_hours <= 40 AND Educational_attainment <= 19
      AND Class_of_worker <= 4
Q6: WHERE Age >= 35  AND Class_of_worker >= 2
      AND Educational_attainment <= 15
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')
```

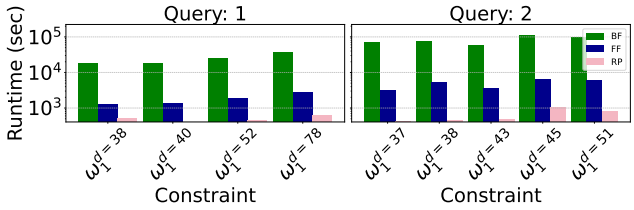
Table 4: Constraints for Experimentation

ID	Constraint
ω_1	$\frac{\textbf{count}(\text{race=1}\wedge\text{label=1})}{\textbf{count}(\text{race=1})} - \frac{\textbf{count}(\text{race=2}\wedge\text{label=1})}{\textbf{count}(\text{race=2})} \in [B_l, B_u]$
ω_2	$\frac{\textbf{count}(\text{ageGroup=1}\wedge\text{label=1})}{\textbf{count}(\text{ageGroup=1})} - \frac{\textbf{count}(\text{ageGroup=2}\wedge\text{label=1})}{\textbf{count}(\text{ageGroup=2})} \in [B_l, B_u]$
ω_3	$\frac{\textbf{count}(\text{sex=1}\wedge\text{PINCP}\geq 20k)}{\textbf{count}(\text{sex=1})} - \frac{\textbf{count}(\text{sex=2}\wedge\text{PINCP}\geq 20k)}{\textbf{count}(\text{sex=2})} \in [B_l, B_u]$
ω_4	$\frac{\textbf{count}(\text{race=1}\wedge\text{PINCP}\geq 15k)}{\textbf{count}(\text{race=1})} - \frac{\textbf{count}(\text{race=2}\wedge\text{PINCP}\geq 15k)}{\textbf{count}(\text{race=2})} \in [B_l, B_u]$
ω_5	$\frac{\sum \text{Revenue}_{\text{ProductsSelectedFromUK}}}{\sum \text{Revenue}_{\text{Selected Products}}} \in [B_l, B_u]$
Ω_6	$\omega_{61} := \textbf{count}(\text{race} = \text{race1}) \leq B_{u_1}$ $\omega_{62} := \textbf{count}(\text{age} = \text{group1}) \leq B_{u_2}$
Ω_7	$\omega_{71} := \textbf{count}(\text{race} = \text{race1}) \leq B_{u_1}$ $\omega_{72} := \textbf{count}(\text{age} = \text{group1}) \leq B_{u_2}$ $\omega_{73} := \textbf{count}(\text{age} = \text{group3}) \leq B_{u_3}$
Ω_8	$\omega_{81} := \textbf{count}(\text{Sex} = \text{Female}) \leq B_{u_1}$ $\omega_{82} := \textbf{count}(\text{Race} = \text{Black}) \leq B_{u_2}$ $\omega_{83} := \textbf{count}(\text{Marital} = \text{Divorced}) \leq B_{u_3}$

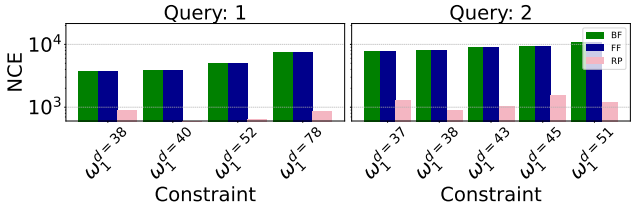
5.2 Performance of FF and RP

We conduct experiments to measure the performance of FF and RP using the Healthcare and ACSIncome datasets with queries in Table 3, constraints in Table 4, and default settings in Section 5.1. For the Healthcare, the constraints ω_1 and ω_2 are used while ω_3 and ω_4 are considered for the ACSIncome. In addition to runtime, we also measure number of candidates evaluated (NCE) which is the total of number of repair candidates for which we evaluate the AC and number of clusters accessed (NCA) which is the total number of clusters accessed by an algorithm.

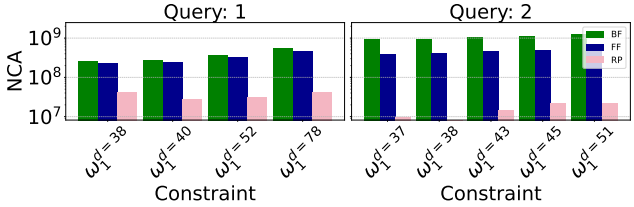
Comparison with Brute Force. We first compare our proposed methods, FF and RP, with the Brute Force (BF) method using the Healthcare, queries Q_1 and Q_2, the constraint $\omega_1$and default settings in Section 5.1. As expected, both FF and RP outperform BF by at least one order of magnitude in terms of runtime as shown in Figure 2a. The RP algorithm significantly reduces both the total NCE and the NCA, while the FF method maintains the same NCE as BF but decreases the NCA compared to BF (as BF does not use clusters we count the number of tuple accesses) as in Figure 2c and Figure 2b.



(a) Runtime (sec).



(b) Total number of constraints evaluated (NCE).



(c) Total number of cluster accessed (NCA).

Figure 2: Runtime, number of candidates evaluated (NCE), and number of clusters accessed (NCA) for FF, RP, and brute force over the Healthcare dataset.

Runtime. Figures 3a and 3b show the runtime of the FF and RP algorithms for Healthcare and ACSIncome, respectively. For given constraint ω_i we vary the bounds $[B_l, B_u]$ to control what percentage of repair candidates have to be processed by BF and FF to determine the top-k repairs as explained above. For example, $\omega_1^{d=38}$ in Figure 3a for Q_1 is the constraint ω_1 from Table 4 with the bounds set such that 38% of the candidate solutions have to be explored by these algorithms. We refer to this as the exploration distance (ED). RP (pink bars) generally outperforms FF (blue bars) for most settings. In Figure 3b, two algorithms exhibit similar performance for Q_4 with ω_3, where solutions are found after exploring only 2% and 3% of the search space. A similar pattern is observed for Q_5 with $\omega_3^{d=3}$ and Q_6 with $\omega_4^{d=4}$. We further investigate the relationship between the ED and the runtime of our algorithms in Section 5.3. In general, RP significantly outperforms FF, demonstrating an improvement of about an order of magnitude due to its capability of pruning and confirming sets of candidate repairs at once.

Total number of candidates evaluated (NCE). We further analyze how NCE affects the performance of our methods. Figures 3c and 3d shows the result of the NCE (on the y axis) for FF and RP on Healthcare and ACSIncome, respectively. RP consistently checks fewer candidates compared to FF. As observed in the runtime evaluation, the ED impacts the efficiency of our algorithms, producing comparable results for FF and RP when a small number of candidates has to be explored, e.g., as shown in Figure 3d for Q_4 with $\omega_3^{d=2}$ and Q_5 with $\omega_3^{d=3}$.

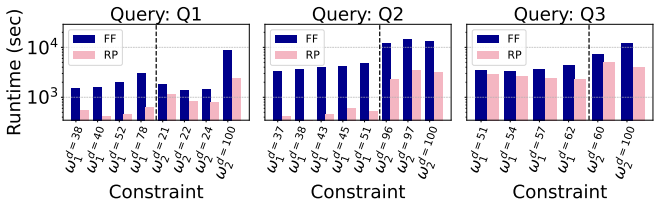
Total Number of Cluster Accessed (NCA). The results of the number of clusters accessed are shown in Figures 3e and 3f for Healthcare and ACSIncome, respectively. Similarly, RP accesses significantly fewer clusters than FF, highlighting its efficiency in limiting the exploration of the search space. Furthermore, 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.

5.3 Performance-Impacting Factors

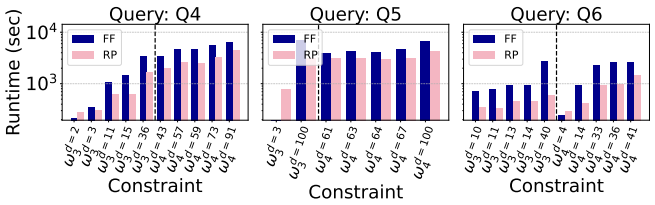
To gain deeper insights into the behavior observed in Section 5.2, we investigate the relationship between the exploration distance (ED) and performance. Additionally, we evaluate the performance of FF and RP in terms of the parameters from Section 5.1. We use the Healthcare, ACSIncome, and TPC-H datasets.

Effect of Exploration distance. We use queries Q_1–Q_3 and the constraint ω_1 on Healthcare and Q_4–Q_6 and the constraint ω_3 on ACSIncome and vary the bounds to control for ED. The result is shown in Figure 4a for Healthcare and in Figure 4b for ACSIncome. For Q_1 and Q_2, when ED 10% or less, FF and RP exhibit comparable performance. A similar pattern is seen for Q_3, where FF performs better than RP for lower ED, but RP outperforms FF for ED > 50% as shown in Figure 4a. The same trend holds for Q_4 and Q_5, while for Q_6, RP consistently outperforms FF for higher ED, as illustrated in Figure 4b. The NCE and NCA follow similar patterns to runtime. For ED > 50%, RP significantly reduces both NCE and NCA. However, when ED < 10%, the difference between the two algorithms diminishes, with both performing similarly. These trends are shown in Figure 4e and Figure 4f for NCA, and in Figure 4c and Figure 4d for NCE. The reason behind these trends is that when solutions are closed to the user query (smaller ED), then there is a lower chance that RP can prune larger sets of candidates at once.

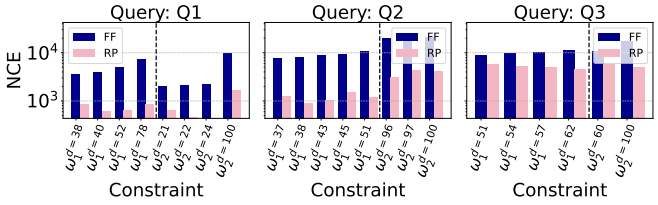
Effect of Bucket Size. We now evaluate the runtime of FF and RP varying the bucket size $\mathcal{S}$ using Q_1 with ω_1 using bounds $[0.44, 0.5]$ for the Healthcare dataset and Q_4 with ω_3 using bounds $[0.34, 0.39]$ for the ACSIncome dataset. We vary the $\mathcal{S}$ from 5 to 2500. Using the default branching factor $\mathcal{B}$ of 5, the structure of the kd-tree for this evaluation is as follows: (i) Level 1: 5 clusters, each with 10,000 data points; (ii) Level 2: 25 clusters, each with 2,000 data points; (iii) Level 3: 125 clusters, each with 400 data points; (iv) Level 4: 525 clusters, each with 80 data points; (v) Level 5: 3,125 clusters, each with 16 data points; (vi) Level 6: 15,625 clusters, each with 3 or 4 data points. Note that the algorithms will generate kd-tree up to the level where the capacity of each cluster at that level is less than or equal to $\mathcal{S}$. For example, for $\mathcal{S} = 200$, the tree will have 4 levels. The results of the runtime are shown in Figure 5a and Figure 5b. Similarly, the NCA as shown in Figure 5e and Figure 5f exhibit the same trend as the runtime. The advantage of smaller bucket sizes is that it is more likely that we can find a cluster that is fully covered / not covered at all. However, this comes at the cost of having to explore more clusters. For NCE, as shown in Figure 5c and Figure 5d, the number of constraints evaluated remains constant across different bucket sizes $\mathcal{S}$. This is because the underlying data remains the same, and varying $\mathcal{S}$ does not affect the set of constraints that need to be evaluated. In preliminary experiments, we have identified $\mathcal{S} = 15$ to yield robust performance for a wide variety of settings and use this as the default.



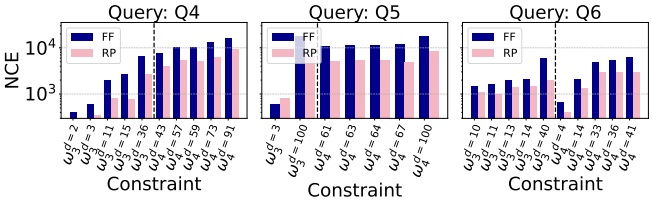
(a) Runtime (sec) - Healthcare dataset.



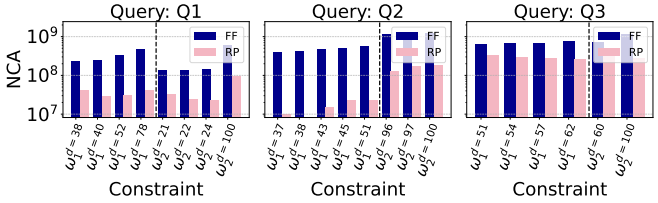
(b) Runtime (sec) - ACSIncome dataset.



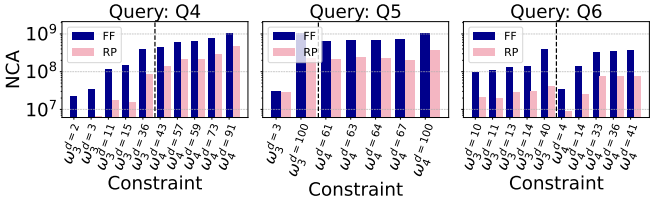
(c) Total number of constraints evaluated (NCE) - Healthcare dataset.



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

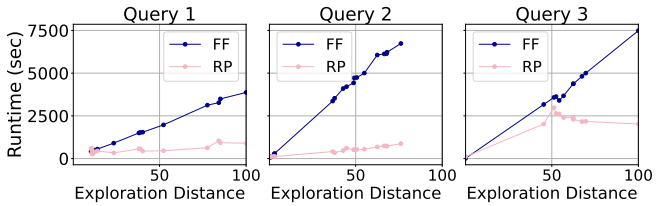


(e) Total number of cluster accessed (NCA) - Healthcare dataset.

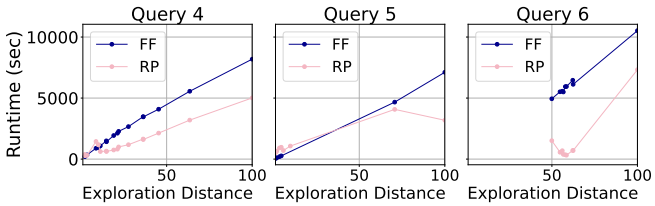


(f) Total number of cluster accessed (NCA) - ACSIncome dataset.

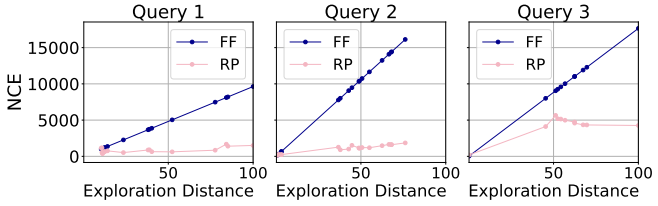
Figure 3: Runtime, NCE, and NCA for FF and RP over the Healthcare and ACSIncome datasets using the queries from Table 3.



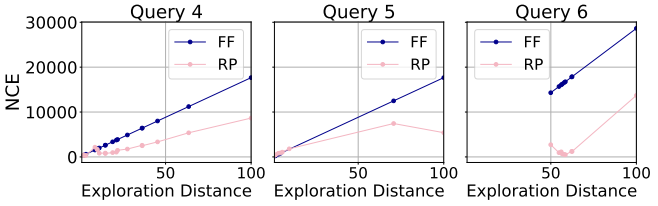
(a) Runtime of FF and RP, varying ED - Healthcare dataset.



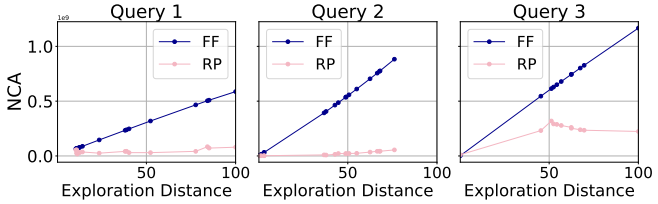
(b) Runtime of FF and RP, varying ED - ACSIncome dataset.



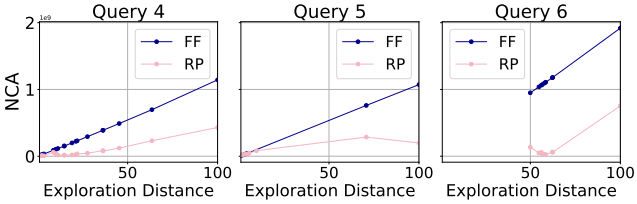
(c) Total number of constraints evaluated (NCE) - Healthcare dataset.



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

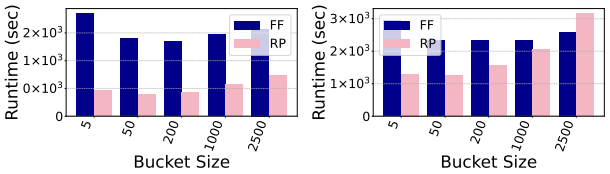


(e) Total number of cluster accessed (NCA) - Healthcare dataset.



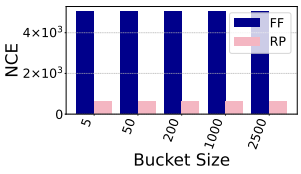
(f) Total number of cluster accessed (NCA) - ACSIncome dataset.

Figure 4: Runtime, NCE, and NCA for FF and RP over the Healthcare and ACSIncome datasets, varying ED.

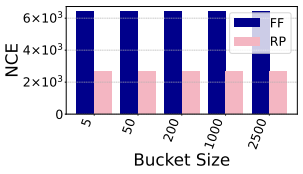


(a) Runtime - Healthcare dataset.

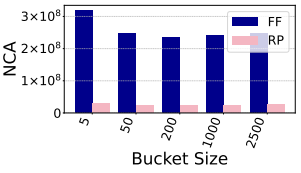
(b) Runtime - ACSIncome dataset.



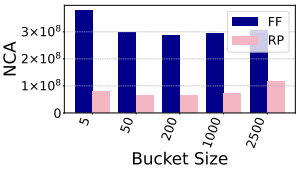
(c) Total number of constraints evaluated (NCE) - Healthcare dataset.



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



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



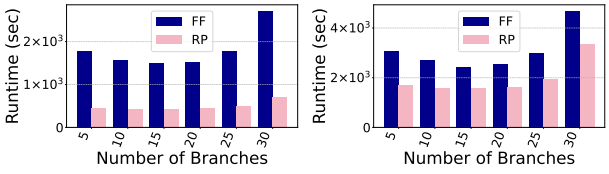
(f) Total number of cluster accessed (NCA) - ACSIncome dataset.

Figure 5: Runtime, NCE, and NCA for FF and RP over the Healthcare and ACSIncome datasets, varying bucket size $\mathcal{S}$.

Table 5: Branching Configuration and Data Distribution

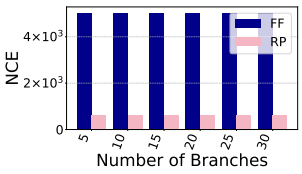
# of Branches	# of Leaves	# of Branches	# of Leaves
5	15625	20	8000
10	10000	25	15625
15	3375	30	27000

Effect of the Branching Factor. We now examines the relationship between the branching factor $\mathcal{B}$ and the runtime of the FF and RP. We use the same queries, constraints, bounds, and datasets as in the previous evaluation. In this experiment, we vary the $\mathcal{B}$ from 5 to 30. The corresponding number of leaf nodes in the kd-tree is shown in Table 5. As we use the default bucket size $\mathcal{S}$ = 15, the branching factor affects the depth of the tree. The result shown in Figure 6a and Figure 6b confirms that the performance of FF and RP correlates with the number of clusters at the leaf level. For the FF, 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 $\mathcal{B}$ = 10 and $\mathcal{B}$ = 20. At $\mathcal{B}$ = 15, FF achieves the lowest runtime, as it involves the smallest number of leaves (3,375). For $\mathcal{B}$ = 30, the number of leaf clusters significantly increases, leading to a substantial rise in the runtime of FF. These results also demonstrate that the tree depths only has negligible impact on the runtime. Similarly, the NCA as shown in Figure 6e and Figure 6f exhibit the same trend as the runtime. For NCE, as shown in Figure 6c and Figure 6d, the number of constraints evaluated remains constant across different branching factors $\mathcal{B}$. This is because the underlying data remains the same, and varying $\mathcal{B}$ does not affect the set of constraints that need to be evaluated. For RP, overall performance trends align with those of FF. However, RP is less influenced by the branching factor.

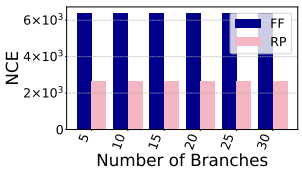


(a) Runtime - Healthcare dataset

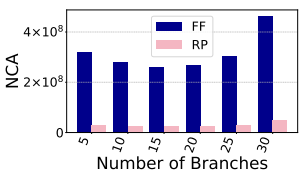
(b) Runtime - ACSIncome dataset



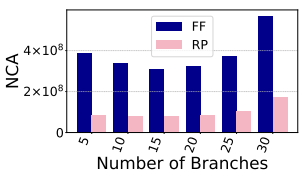
(c) Total number of constraints evaluated (NCE) - Healthcare dataset



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



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

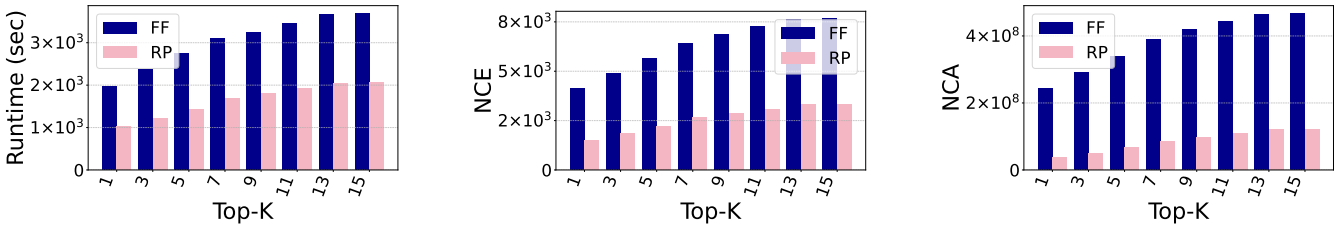


(f) Total number of cluster accessed (NCA) - ACSIncome dataset

Figure 6: Runtime, NCE, and NCA for FF and RP over the Healthcare and ACSIncome datasets, varying the number of branches $\mathcal{B}$.

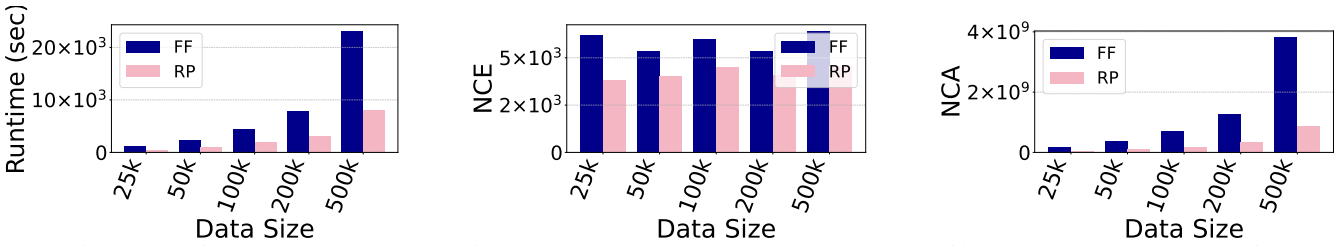
Effect of k. In this experiment, we vary the parameter k from 1 to 15. For both FF and RP, as k increases, the runtime also increases, as shown in Figure 7a. This behavior is expected since finding a single repair (k=1) requires less computation than identifying multiple repairs. When k is larger, the algorithm must explore a larger fraction of the search space to find additional repairs. Similarly, both the NCE as shown in Figure 7b and NCA as shown in Figure 7c 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. RP consistently outperforms FF.

Effect of Dataset Size. Next, we vary the dataset size and measure the runtime, NCE and the NCA for the TPC-H dataset, Q_7 from Table 3, ω_5 from Table 4. We use default settings for all parameters (Section 5.1). As shown in Figure 8a, as the data size increases, the runtime also increases. Dataset size impacts both the size of the search space and the size of the kd-tree. Nonetheless, our algorithm scale roughly linearly in dataset size demonstrating the effectiveness of using materialized aggregation results for clusters and range-based pruning of candidate repair sets. This is further supported by the NCA measurements shown in Figure 8b, which exhibit the same trend as the runtime. For NCE, as shown in Figure 8c, the number of constraints evaluated varies across different dataset sizes. This variation occurs because the underlying data itself changes with the dataset size. This contrasts with the observations in Figure 5 and Figure 6, where the number of evaluated constraints remained constant due to the data being fixed across configurations. These results confirm that the number of evaluated



(a) Runtime (sec). (b) Total number of constraints evaluated (NCE). (c) Total number of clusters accessed (NCA).

Figure 7: Runtime, NCE, and NCA for FF and RP over top-k.



(a) Runtime (sec). (b) Total number of constraints evaluated (NCE). (c) Number of clusters accessed (NCA).

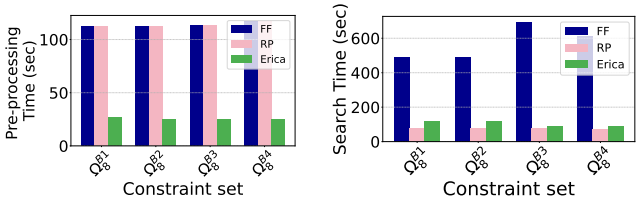
Figure 8: Runtime, NCE, and NCA for FF and RP over the TPC-H dataset, varying data size.

constraints is influenced by changes in the dataset content, rather than by variations in the branching factor $\mathcal{B}$ or bucket size $\mathcal{S}$ alone.

5.4 Comparison with Related Work

We compare our approach with Erica [21], which solves the related problem of finding all minimal refinements of a given query that satisfy a set of cardinality constraints for groups within the result set. Such constraints are special cases on the ACs we support. Erica returns all repairs that are not *dominated* by any other repair where a repair dominates another repair if it is at least as close to the user query for every condition θ_i and strictly closer in at least one condition. That is, Erica returns the skyline [8]. Thus, different from our approach, the number of returned repairs is not an input parameter in Erica. For a fair comparison, we determine the minimal repairs and then set k such that our methods returns a superset of the repairs returned by Erica. To conduct the evaluation for Erica, we used the available Python implementation (https://github.com/JinyangLi01/Query_refinement).[1] We adopt the queries, constraints, and the dataset from [21]. We compare the generated refinements and runtime of our techniques with Erica using Q_1 and Q_2 (Table 3) on the Healthcare dataset (size 50K) with constraints Ω_6 and Ω_7 (Table 4), respectively.

Comparison on the Generated Repairs. We first compare the generated repairs by our approach and Erica. For Q_1 with Ω_6, Erica generates 7 minimal repairs whereas our technique generates 356, including those produced by Erica. Similarly, for Q_2 with Ω_7, Erica generates 9 minimal repairs while our approaches generates



(a) Pre-processing (b) Search

Figure 9: Runtimes of FF, RP, and Erica.

1035, including those produced by Erica. In general, our techniques successfully generates all repairs produced by Erica. Note that the top-1 repair returned by our approach is guaranteed to be minimal (not dominated by any other repair). However, the remaining minimal repairs returned by Erica may have a significantly higher distance to the user query than the remaining top-k answers returned by our approach. For example, in Q_2, given the condition `num-children >= 4` of the user query, our solution includes a refined condition `num-children >= 3` whereas Erica provides a refinement `num-children >= 1` which is far from the user query. As mentioned above, to ensure that our approach returns a superset of the solutions returned by Erica's solutions, we did adjusted k per query and constraint set to ensure that.

Runtime Comparison. The experiment utilizes Q_4 with Ω_8 on the 50K ACSIncome dataset, which is derived from Erica's dataset, query, and constraint. We use the same bounds in the constraints for both Erica and our algorithms: $B1 := (B_{u_1} = 30, B_{u_2} = 150, B_{u_3} = 10)$ and $B2 := (B_{u_1} = 30, B_{u_2} = 300, B_{u_3} = 25)$, $B3 := (B_{u_1} = 10, B_{u_2} = 650, B_{u_3} = 50)$, and $B4 := (B_{u_1} = 15, B_{u_2} = 200, B_{u_3} = 15)$. 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. We set k=17 for constraintsets Ω_8^{B1}

[1]We did modify the the code for a more fair comparison of algorithms regarding the evaluation of constraints as this part of our implementation is implemented in pure Python while Erica used Pandas Dataframe operations that are implemented in C. We leave a full implementation of our methods in a lower-level language such as C++ to future work.

and Ω_8^{B2}, k=11 for Ω_8^{B3}, and k=13 for Ω_8^{B4}. Due to the different optimization criterions, variations in the generated repairs between our approach and Erica are expected. The results in Figure 9a reveal 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 9b, in pre-processing time which is the time of materializing aggregates and constructing the kd-tree for our methods and generating provenance expressions for Erica, Erica outperforms both RP and FF, indicating that Erica summarizes information needed for the search process more effectively. Overall the total runtime of RP and Erica are comparable, even through our approach does not apply any specialized optimizations that exploit the monotonicity of the constraints supported in Erica, while our approach supports a significantly broader class of constraints that include common fairness notions such as SPD. These results also highlight the need for our range-based optimizations in RP, as FF is significantly slower than Erica.

6 Related Work

Query refinement & relaxation. Li et al. [21] 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 [21] involve filter-aggregation queries that we also consider as compared in Section 5.4. However [21] do not allow for arithmetic combinations of the results of such queries, e.g., as required by standard fairness conditions. Mishra et al. [24] refine a query to return a given number k of results. Koudas et al. [20] refine a query that returns an empty result to produce at least one answer. In [7, 28] a query repairs return missing results of interest provided by the user. 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 [9, 18, 24, 25, 31]. However, real-world use cases are often inherently non-monotone.

How-to queries. Like our work and other query refinement techniques [21], the purpose of how-to queries [23] 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. [30] 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 [10, 13, 14] 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 interval arithmetic across various database applications [12, 15, 27, 32]. For instance, [15] explored uncertainty tracking using attribute-level bounds for complex queries, demonstrating the utility of bounds-based approaches in query processing. Similarly, the work in [32] introduced a bounding technique for efficiently computing iceberg cubes, establishing an early foundation for leveraging interval arithmetic to constrain aggregate functions. The use of intervals and interval arithmetic to bound the result of computations has been used extensively in *abstract interpretation* [11]. For example, see [12, 27] for introductions to interval arithmetic and more advanced numerical abstract domain.

7 Conclusion and Future Work

In this work, we study the problem of repairing a query to satisfy a constraint that evaluates a predicate over an arithmetic combination of the results of aggregation-filter queries evaluated on top of a user query result. Unlike previous work on repairing queries to satisfy cardinality constraints or constraints on monotone aggregation functions, our method in addition supports non-monotone constraints, including important fairness metrics such as SPD. We handle the exponential search space of candidate repairs through two related optimizations: (i) we materialize aggregation results for subsets of the input database in a kd-tree. Then these materialized results can be combined to determine the aggregation result for a repair candidate, effectively reusing computations across repair candidates; (ii) we represent sets of repair candidates using bounds on the constants for each condition of the user query and then determine bounds on the aggregation constraint result for every candidate in the set using interval arithmetic [12]. This enables us to confirm sets of candidates to be repairs or prune them at once. The results of our experiments demonstrate the efficiency of our techniques across different datasets, queries, and constraints.

Interesting directions for future work include (i) the study of more general types of repairs, e.g., repairs that add or remove joins or change the structure of the query by introducing / deleting operators, (ii) considering other optimization criteria, e.g., minimizing the changes to the user query result as in some work on query refinement, and (iii) employing more expressive domains than intervals for computing tighter bounds, e.g., zonotopes [12].

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] Shatha Algarni, Boris Glavic, Seokki Lee, and Adriane Chapman. 2025. Efficient Query Repair for Arithmetic Expressions with Aggregation Constraints (Extended Version). *techreport* (2025). https://github.com/ShathaSaad/Query-Refinement-of-Complex-Constraints/blob/main/query-repair-techreport.pdf

[4] 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.

[5] 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

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

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

[8] S. Borzsony, D. Kossmann, and K. Stocker. 2001. The skyline operator. In *ICDE*. 421–430.

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

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

[11] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*. 238–252.

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

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

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

[21] 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.

[22] 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.

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

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

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

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

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

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

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

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

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

[32] 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.

[33] 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.