Efficient Query Repair for Aggregate Constraints

(extended version)

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 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 job applicants. Thus, the query has to be repaired such that the result set of the fixed query satisfies all constraints. Prior work in this area, including query-based explanations [15, 37] and repairs [9] for missing answers, work on answering why-not questions [5, 15] as well as query refinement / relaxation approaches [26, 29, 38] 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.

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
```

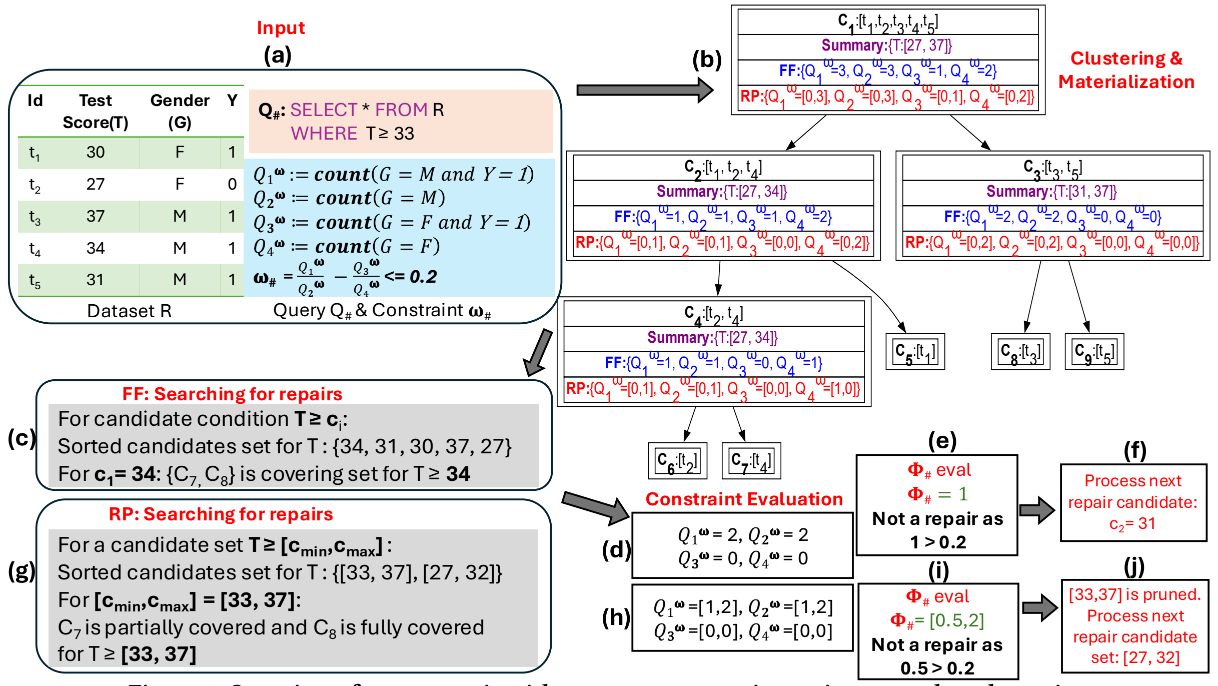
Aggregate Constraint*. The employer wants to ensure that interview decisions are not biased against a specific gender using statistical parity difference (SPD) [4, 27]. given two groups (e.g., male and female) and a binary outcome attribute Y where $Y = 1$ is assumed to be a positive outcome (*`OfferInterview`*=1 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* **count**(θ) *to denote the number of query results satisfying condition θ. For example,* **count**$(G = M \wedge Y = 1)$ *counts male applicants with a positive label.*

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

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.*

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 its selection conditions, similar to [24, 29]. 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. Moreover, instead of assuming a single best repair, we consider returning k repairs ranked by their distance to the original query to allow users

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



Input
(a)
Id | Test Score(T) | Gender (G) | Y
t1 | 30 | F | 1
t2 | 27 | F | 0
t3 | 37 | M | 1
t4 | 34 | M | 1
t5 | 31 | M | 1
Dataset R
Q#: SELECT * FROM R WHERE T ≥ 33
$Q_1^\omega := \textbf{count}(G = M \text{ and } Y = 1)$
$Q_2^\omega := \textbf{count}(G = M)$
$Q_3^\omega := \textbf{count}(G = F \text{ and } Y = 1)$
$Q_4^\omega := \textbf{count}(G = F)$
$\omega_\# = \frac{Q_1^\omega}{Q_2^\omega} - \frac{Q_3^\omega}{Q_4^\omega} <= 0.2$
Query Q# & Constraint ω#
(b)
Clustering & Materialization
C1:[t1,t2,t3,t4,t5] Summary:{T:[27, 37]} FF:{Q1ω=3, Q2ω=3, Q3ω=1, Q4ω=2} RP:{Q1ω=[0,3], Q2ω=[0,3], Q3ω=[0,1], Q4ω=[0,2]}
C2:[t1, t2, t4] Summary:{T:[27, 34]} FF:{Q1ω=1, Q2ω=1, Q3ω=1, Q4ω=2} RP:{Q1ω=[0,1], Q2ω=[0,1], Q3ω=[0,0], Q4ω=[0,2]}
C3:[t3, t5] Summary:{T:[31, 37]} FF:{Q1ω=2, Q2ω=2, Q3ω=0, Q4ω=0} RP:{Q1ω=[0,2], Q2ω=[0,2], Q3ω=[0,0], Q4ω=[0,0]}
C4:[t2, t4] Summary:{T:[27, 34]} FF:{Q1ω=1, Q2ω=1, Q3ω=0, Q4ω=1} RP:{Q1ω=[0,1], Q2ω=[0,1], Q3ω=[0,0], Q4ω=[1,0]}
C5:[t1]
C8:[t3]
C9:[t5]
C6:[t2]
C7:[t4]
(c)
FF: Searching for repairs
For candidate condition T ≥ ci:
Sorted candidates set for T : {34, 31, 30, 37, 27}
For c1= 34: {C7, C8} is covering set for T ≥ 34
(g)
RP: Searching for repairs
For a candidate set T ≥ [cmin,cmax] :
Sorted candidates set for T : {[33, 37], [27, 32]}
For [cmin,cmax] = [33, 37]:
C7 is partially covered and C8 is fully covered for T ≥ [33, 37]
Constraint Evaluation
(d)
Q1ω = 2, Q2ω = 2
Q3ω = 0, Q4ω = 0
(e)
Φ# eval
Φ# = 1
Not a repair as 1 > 0.2
(f)
Process next repair candidate: c2= 31
(h)
Q1ω =[1,2], Q2ω =[1,2]
Q3ω =[0,0], Q4ω =[0,0]
(i)
Φ# eval
Φ#= [0.5,2]
Not a repair as 0.5 > 0.2
(j)
[33,37] is pruned. Process next repair candidate set: [27, 32]

Figure 1: Overview of query repair with aggregate constraints using range-based pruning.

to choose the one that best matches their intent. ACs significantly generalize the cardinality constraints supported in prior work on query repair for fairness [13, 14, 26] and on query relaxation & refinement [25, 29]. By allowing arithmetic combinations of aggregation results we support common fairness measures such as SPD that cannot be expressed as cardinality constraints. Our work has applications beyond fairness, when applying uniform criteria to select a set of entities subject to additional constraints, e.g., a government agency has to solicit contractors, 20% of which should be local (See [3] for a detailed example). New challenges arise from the generality of ACs as ACs are typically not monotone, invalidating most optimizations proposed in related work.

Repairing queries based on aggregate constraints has additional applications beyond fairness including solving the *empty answer* problem where the conditions of a query that returns an empty result should be relaxed to return an answer [32, 33] and the converse where a query that returns to many answers should be refined [38]. This is the case, both types of use cases can be expressed as aggregate constraints: **count**() $>$ 0 for the empty answer problem and **count**() $< c$ for the query refinement problem. Other types of cardinality constraints can also be expressed naturally as aggregate constraints. In addition to SPD, other common fairness metrics can also be expressed as arithmetic combinations of aggregate results. Furthermore, the expressive power gained by allowing arithmetic operations is not limited to fairness as illustrated in the following example.

Example 2. *Consider a government agency responsible for contracting out supply of school meals to vendors. Due to anti-corruption legislation, the agency has to decide on a fixed set of criteria based on which vendors are prefiltered and can only hire vendors that fulfill the criteria. Such criteria can be naturally expressed as a query over a table storing vendors. Factors that could be used considered include the vendor's price per meal, the calories of the food they supply, their distance in miles to the school district, since when the vendor has been in business, and others. Based on these attributes, the agency may determine a set of criteria as shown below (low enough price, sufficient amount of calories, withing 100 miles of the school, and the vendor should have been in business for at least 5 years).*

```
Q3: SELECT * FROM vendor
     WHERE price < 20
       AND calories > 500
       AND distance < 100
       AND established < 2020
       ...
```

Using query repairs with aggregate constraints, the agency can refine their criteria to fulfill additional requirements such as:

- ***Cardinality constraints**: The agency needs to ensure that they have a sufficiently large pool of vendors that fulfill their criteria to choose from to ensure all schools are covered as not all contract negotiations with vendors will be successful. Furthermore, as the agency has to justify which vendors that match their criteria they select, returning too many vendors is also undesirable. This can be expressed as a conjunction of two aggregate constraints:*

$$\omega_{numvendors} := \textbf{count}() \geq k \wedge \textbf{count}() < l$$

for some $k < l$.
- ***Reducing carbon footprint**: A new regulation is proposed to reduce the carbon footprint of supplying meals requiring that at least 30% of vendors have to be within a distance of 15 miles of the*

school they supply. This can be expressed as an aggregate constraint:

$$\omega_{carbon} := \frac{\mathbf{count}(distance < 15)}{\mathbf{count}()} \geq 0.3$$

- ***Budget constraint****: Assume that the database used by the agency has a certain total number of meals M that need to be supplied and each vendor has a maximum capacity of meals they can supply. To ensure that vendors are selected such that at least the required capacity is met while ensuring that the total cost is below the budget B, the agency can use the following constraint:*

$$\omega_{budget} := \mathbf{sum}(capacity) \geq M \wedge \mathbf{sum}(price) \leq B$$

Example 3 (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
           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.

A brute force approach for solving the query repair problem is to enumerate all possible candidate repairs in order of their distance to the user query. Each candidate is evaluated by running the modified query and checking whether it fulfills the aggregate constraint. The algorithm terminates once k have been found. The main problem with this approach is that the number of repair candidates is exponential in the number of predicates in the user query. Furthermore, for each repair candidate we have to evaluate the modified user query and one or more aggregate queries on top of its result. Given that the repair problem is NP-hard we cannot hope to avoid this cost in general.

Reusing aggregation results. Nonetheless, we identify two opportunities for optimizing this process. 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. To exploit this observation, 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 attribute values within the cluster (as is done in, e.g., zonemaps [31, 42]). 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)*. In contrast to the brute force approach FF reuses aggregation results materialized for clusters. Continuing with 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. Here, we simplify Q_1 from Example 1 by considering only a single condition, $TestScore(T) \geq 33$, but use the same aggregate constraint $\omega_{\#}$. Consider cluster C_2 in Figure 1(b), where the values of attribute *TestScore* (T) are bounded by $[27, 34]$. For a repair candidate with a condition $T \geq 37$ the entire cluster can be skipped as no tuples in C_2 can fulfill the condition. In contrast, for condition $T \geq 30$, all tuples in C_3 satisfy the condition, since the values of attribute T are bounded by $[31, 37]$ (see Figure 1(b)).

Evaluating multiple candidate repairs at once. We extend this idea to bound the aggregation constraint result for sets of repair candidates at once. We refer to this approach as *Cluster Range Pruning (RP)*. A set of repair candidates is encoded as intervals of values for the constant c_i of each predicate a_i op c_i of the user query, e.g., $c_1 \in [33, 37]$ as shown in Figure 1(g). We again reason about whether all / none of the tuples in a cluster fulfill the condition for *every* repair candidate from the set. The result are valid bounds on the aggregation constraint result for any candidate repair within the candidate set. Using these bounds we validate or disqualify complete candidate sets at once.

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.
- We present an optimized algorithm for the aggregate constraint repair problem problem that reuses aggregation results when evaluating repair candidates (Section 3) and evaluates multiple repair candidates by exploiting sound bounds that hold for all repair candidates in a set (Section 4).
- A comprehensive experimental evaluation over multiple datasets, queries and constraints in Section 5. Compared to the state of the art [26], we cover significantly more complex constraints.

2 Problem Definition

We consider a dataset $D = R_1, \cdots, R_z$ consisting of one or more relations R_i, an input query Q that should be repaired, and an aggregate constraint. The goal is to find the k queries that fulfill the constraint and minimally differ from Q.

User Query. A user query Q is an select-project-join (SPJ) query, i.e., a relational algebra expression 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 $\theta = \theta_1 \wedge \ldots \wedge \theta_m$ of comparisons of the form a_i op$_i$ c_i. For numerical attributes a_i, we allow op$_i \in \{<, >, \leq, \geq, =, \neq\}$ and for categorical attributes a_i we only allow $\text{op}_i \in \{=, \neq\}$. 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:

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

Here, Φ is an arithmetic expression using operators $(+, -, *, /)$ over $\{Q_i^\omega\}$, *op* is a comparison operator, and τ is a threshold. Aggregate constraints are non-monotone in general due to (i) non-monotone arithmetic operators like division, (ii) non-monotone aggregation function, e.g., **sum** over the integers $\mathbb{Z}$, and (iii) combination of monotonically increasing and decreasing aggregation functions, e.g., $\textbf{max}(A) + \textbf{min}(B)$.

Query Repair. Given a user query Q, database D, and 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. We restrict repairs to changes of the selection condition θ of Q. For ease of presentation, we consider a single AC, but our algorithms can also handle a conjunction of multiple ACs, e.g., the cardinality for one group should be above a threshold τ_1 and for another group below a threshold τ_2. Given the user query Q with a condition $\theta = := \bigwedge_{i=1}^m a_i \text{ op}_i\, c_i$, 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: $\theta' := \bigwedge_{i=1}^m a_i \text{ op}_i\, c_i'$. For convenience, we will often use the vector of constants $\vec{c} = [c_1', \ldots, c_m']$ to denote a repair candidate and use $\textsc{Cand}_Q$ to denote the set of all candidates. A candidate is 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 to preserve the intent of the user's query as much as possible. We measure similarity using a linear weighted combination of distances between the constants used in selection conditions of the user query and the repair, similar to [13, 25].[1] 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:

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

where w_i is a weight in $[0, 1]$ such that $\sum_i w_i = 1$ and the distance between two predicates $\theta_i = a_i \text{ op}_i\, c_i$ and $\theta_i' = a_i \text{ op}_i\, c_i'$ 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. For example, for Example 1, the repair candidate with conditions `Major` = EE, `Testscore` ≥ 33, and `GPA` ≥ 3.9 has a distance of $1 + \frac{33-33}{33} + \frac{3.9-3.8}{3.8} = 1.026$.

We are now ready to formulate the problem studied in this work, computing the k repairs with the smallest distance to the user query. Among these k repairs, the user can then select the repair that best aligns with their preferences. Here $\text{top-k}_{x \in X} f(x)$ returns the k elements from set X with the smallest $f(x)$ values.

Aggregate constraint repair problem:

- **Input**: user query Q, database D, constraint ω, threshold k
- **Output**:

$$\underset{Q_{fix} \in \textsc{Cand}_Q:\ Q_{fix}(D) \models \omega}{\text{top-k}} \quad d(Q, Q_{fix})$$

Hardness. To generate a repair Q_{fix} of Q, we must explore the combinatorially large search space of possible candidate repairs. For a single predicate over an attribute a_i with N_i distinct values there are $O(N_i)$ possible repairs. Thus, the size of the candidate set $\textsc{Cand}_Q$ is in $O(\prod_{i=1}^m N_i)$, exponential in m, the number of conditions in the user query. Unsurprisingly, the aggregate constraint repair problem is NP-hard in the schema size.

Consider a query with a conjunction of conditions of the form $a_i \text{ op}_i\, c_i$ for $i \in [1, m]$. Again N_i denote the number of values in the active domain of a_i. Each candidate repair corresponds to choosing constants $[c_1', \ldots, c_m']$. 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.

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.

[1] We discuss other possible optimality criteria used in prior work in Section 6. Other options include returning all repairs that are pareto optimal regarding predicate-level distances or to minimize the change to the query's result. Our algorithms can be extended to optimize for any distance metric which can be interval-bounded based on bounds for attribute values of a set of tuples.

Algorithm 1 FullCoverClusterSet

```
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:     $C_{cur} \leftarrow$ pop$(stack)$1
 5:     $in \leftarrow$ true, $notin \leftarrow$ false
 6:     for all $\theta'_i = (a_i \text{ op}_i\ c'_i) \in \theta'$ do
 7:         $in \leftarrow in \wedge \text{eval}_\forall(\theta'_i, \text{BOUNDS}_{a_i}(C_{cur}))$   ▷ All tuples fulfill $\theta'_i$?
 8:         $notin \leftarrow notin \vee \text{eval}_\forall(\neg\theta'_i, \text{BOUNDS}_{a_i}(C_{cur}))$
 9:     if $in$ then   ▷ All tuple in C fulfill $\theta'$
10:         $\mathbf{C} \leftarrow \mathbf{C} \cup \{C_{cur}\}$
11:     else if $\neg notin$ then   ▷ Some tuples in C may fulfill $\theta'$
12:         for all $C \in$ children$(C_{cur})$ do   ▷ Process children
13:             $stack \leftarrow stack \cup \{C\}$
14: return $\mathbf{C}$
```

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 $\mathbf{count}(C) := |C|$ 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 $Q_2^\omega = \mathbf{count}(Gender(G) = M)$ 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 $\mathcal{S}$ (i.e. $|C| \leq \mathcal{S}$). 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 a set of disjoint clusters $\mathbf{C}$ such that the union of these clusters is

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

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 **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 **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 **eval**$_\forall$ returns true for all θ'_i and *not covered at all* (case (ii)) if **eval**$_\forall$ returns true for at least one comparison $\neg\theta'_i$.

3.2.2 Determining Coverage In Table 1, we define the function **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 **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 $[\underline{a}, \overline{a}]$.

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 $\textsc{bounds}_T(C_1) = [27, 37]$. 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 $1 \leq 0.2 = \mathbf{false}$ 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

Algorithm 2 ParCoverClusterSet

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:  $stack \leftarrow [C_{root}]$
2:  $\mathbf{C}_{full} \leftarrow \emptyset, \mathbf{C}_{partial} \leftarrow \emptyset$   ▷ Initialize cluster sets
3:  while $stack \neq \emptyset$ do
4:      $C_{cur} \leftarrow \mathbf{pop}(stack)$
5:      $in \leftarrow \mathbf{true}, pin \leftarrow \mathbf{true}$
6:      for all $\theta_i = (a_i \text{ op}_i\, c_i) \in \theta$ do   ▷ $C_{cur}$ fully / part. covered?
7:          $in \leftarrow in \wedge \mathbf{reval}_\forall(\theta_i, [\underline{c_i}, \overline{c_i}], \textsc{bounds}_{a_i}(C_{cur}))$
8:          $pin \leftarrow pin \wedge \mathbf{reval}_\exists(\theta_i, [\underline{c_i}, \overline{c_i}], \textsc{bounds}_{a_i}(C_{cur}))$
9:      if $in$ then   ▷ Add fully covered cluster to the result
10:         $\mathbf{C}_{full} \leftarrow \mathbf{C}_{full} \cup \{C_{cur}\}$
11:     else if $pin$ then
12:         if $\mathbf{isleaf}(C_{cur})$ then   ▷ Partially covered leaf cluster
13:             $\mathbf{C}_{partial} \leftarrow \mathbf{C}_{partial} \cup \{C_{cur}\}$
14:         else   ▷ Process children of partial cluster
15:             for all $C \in \mathbf{children}(C_{cur})$ do
16:                 $stack \leftarrow stack \cup \{C\}$
17: return $(\mathbf{C}_{full}, \mathbf{C}_{partial})$
```

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. 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 bounds.

4.1 Computing Top-k Repairs

RP (Algorithm 3) 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

Algorithm 3 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 Q_{top-k}

```
1:  $Q_{top-k} \leftarrow \emptyset$   ▷ 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:      $\mathbb{Q}_{cur} \leftarrow$ POP($queue$)
6:      $\mathbb{Q}_{next} \leftarrow$ PEEK($queue$)   ▷ Peek at next item in queue
7:      $(\mathbf{C}_{full}, \mathbf{C}_{partial}) \leftarrow$ ParCoverClusterSet($\mathbb{Q}_{cur}, C_{root}, \theta$)
8:      if ACEVAL$_\forall(\omega, \mathbf{C}_{full}, \mathbf{C}_{partial})$ then   ▷ All $Q' \in \mathbb{Q}_{cur}$ are repairs?
9:          $rcand \leftarrow$ INSERT($rcand, \mathbb{Q}_{cur}$)
10:     else if ACEVAL$_\exists(\omega, \mathbf{C}_{full}, \mathbf{C}_{partial})$ then   ▷ Some repairs?
11:         for $\mathbb{Q}_{new} \in$ RangeDivide($\mathbb{Q}_{cur}$) do   ▷ divide ranges
12:             if hasCandidates($\mathbb{Q}_{new}$) then
13:                 $queue \leftarrow$ INSERT($queue, \mathbb{Q}_{new}$)
14:     $Q_{top-k} \leftarrow$ topkConcreteCand($rcand, k$)   ▷ Top k repairs
15:     if $|Q_{top-k}| \geq k$ then   ▷ Have k repairs?
16:         if $\underline{d(Q, \mathbb{Q}_{next})} > d(Q, Q_{top-k}[k])$ then   ▷ Rest inferior?
17:             break
18: return $Q_{top-k}$
```

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.

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 ParCoverClusterSet (Algorithm 2) 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 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 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 ACEVAL$_\forall$ and 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 ACEVAL$_\forall$ returns false as some $Q_{fix} \in \mathbb{Q}_{cur}$ may not fulfill the constraint. However, 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 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:

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

Each RangeDivide forces further evaluation down to the leaf clusters, making recursive partitioning the dominant factor in the overall runtime. In the worst-case scenario, where every split results in another partitions case, the algorithm must descend to the finest-grained leaf clusters, causing its runtime to approach that of the brute-force approach.

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 $Q_{top-k}[k]$ from 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 $Q_{top-k}[k]$. 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, Q_{top-k} contains exactly the top-k repairs and the algorithm returns 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 2) 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 2, 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 2. 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* [17, 35] 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 [41]. 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 $\omega := \tau \text{ op } \Phi$. There are three possible outcomes for a repair candidate set: (i) $\tau \text{ op } \Phi$ is true for all $[\underline{\Phi}, \overline{\Phi}]$ which $\text{ACEVAL}_\forall$ determines using $\mathbf{reval}_\forall$ and bounds $[\tau, \tau]$; (ii) some of the candidate in $\mathbb{Q}$ may fulfill the condition, which $\text{ACEVAL}_\exists$ 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 $Q_{1C_7}^\omega$ 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 $Q^{\omega}_{1\,C_7} = [0, 1]$. Similarly, we compute the remaining aggregation bounds: $Q^{\omega}_{1\,C_8} = [1, 1]$, $Q^{\omega}_{2\,C_7} = [0, 1]$, $Q^{\omega}_{2\,C_8} = [1, 1]$, $Q^{\omega}_{3\,C_7} = [0, 0]$, $Q^{\omega}_{3\,C_8} = [0, 0]$, $Q^{\omega}_{4\,C_7} = [0, 0]$, $Q^{\omega}_{4\,C_8} = [0, 0]$.

Next, in Figure 1(h) we sum the lower and upper bounds for each aggregation Q^{ω}_i across all clusters in $\mathbf{C}$: $Q^{\omega}_1 = Q^{\omega}_{1\,C_7} + Q^{\omega}_{1\,C_8} = [1, 2]$, $Q^{\omega}_2 = Q^{\omega}_{2\,C_7} + Q^{\omega}_{2\,C_8} = [1, 2]$, $Q^{\omega}_3 = Q^{\omega}_{3\,C_7} + Q^{\omega}_{3\,C_8} = [0, 0]$, $Q^{\omega}_4 = Q^{\omega}_{4\,C_7} + Q^{\omega}_{4\,C_8} = [0, 0]$. We then substitute the computed values $\{Q^{\omega}_1, Q^{\omega}_2, Q^{\omega}_3, Q^{\omega}_4\}$ into $\omega_{\#}$ and evaluate the resulting expression using interval arithmetic (Table 2). Given: $\omega_{\#} = {}^{Q^{\omega}_1}/_{Q^{\omega}_2} - {}^{Q^{\omega}_3}/_{Q^{\omega}_4}$ the lower and upper bounds for the first term ${}^{Q^{\omega}_1}/_{Q^{\omega}_2}$ are computed as: $[\underline{{}^{E_1}/_{E_2}}, \overline{{}^{E_1}/_{E_2}}] = [{}^1/_2, 2]$. Similarly, for the second term: ${}^{Q^{\omega}_3}/_{Q^{\omega}_4} = [0, 0]$. 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 $\mathbb{Q}$.

In practice, RP performs best when kd-tree clusters are homogeneous with respect to the predicate attributes a_i in θ i.e., when most cluster bounds, BOUNDS$_{a_i}$, lie entirely above or below the repair intervals $[\underline{c_i}, \overline{c_i}]$. This enables efficient pruning of infeasible or fully satisfying candidate sets. The effect is especially strong when large regions of the search space can be ruled out or accepted entirely based on the aggregation constraint ω. If predicate attributes are strongly correlated with those in the arithmetic expression Φ, cluster inclusion often predicts the outcome of Φ, allowing entire repair sets to be evaluated at once. Conversely, when many clusters partially overlap the predicate ranges, the algorithm must recursively partition $\mathbb{Q}$ and evaluate finer-grained cluster levels, eventually matching the cost of brute-force in the worst case.

Theorem 4.1 (Correctness of FF and RP). *Given an instance (Q, D, ω, k) of the* aggregate constraint repair problem, *FF and RP (Algorithm 3) compute the solution for this problem instance.*

Before presenting the proof of Theorem 4.1 we first establish several auxiliary results used in the proof. First we demonstrate that given the conditions of a repair candidate Q_{fix}, Algorithm 1 returns a *covering cluster set* $\mathbf{C}$ which is a set of clusters that cover exactly the tuples from the input database D that fulfill the selection condition of Q_{fix}. Then we proceed to show that evaluating any filter-aggregate query over the result of Q_{fix} using the materialized aggregation results for a covering $\mathbf{C}$ yields the same result as computing this aggregation on the result of Q_{fix} (Lemma 4.1). As aggregate constraints are evaluated over the results of filter-aggregate queries (Lemma 4.2), this then immediately implies that checking whether an aggregate constraint holds for repair candidate Q_{fix} can be determined using the pre-aggregated results stored for clusters using a fully covering cluster set for Q_{fix}. Thus, algorithm FF described in Section 3 solves the Aggregate Constraint Repair Problem.

Next we prove several auxiliary lemmas that will help us establish the correctness for algorithm RP. RP determines for a set of repair candidates $\mathbb{Q}$ (described through ranges for each predicate) at once whether all or none of the candidates in the set fulfill the given aggregate constraint. Recall that we use Algorithm 3 to compute a partially covering cluster set for $\mathbb{Q}$ which is a pair $(\mathbf{C}_{full}, \mathbf{C}_{partial})$ where $\mathbf{C}_{full}$ contains clusters for which all tuples fulfill the condition of all repair candidates in $\mathbb{Q}$ and $\mathbf{C}_{partial}$ contains clusters for which some tuples may fulfill the condition of some repair candidates in $\mathbb{Q}$. We show that for any $Q_{fix} \in \mathbb{Q}$, the set of tuples returned by Q_{fix} over the input database D is a superset of the tuples in $\mathbf{C}_{full}$ and a subset of the tuples in $\mathbf{C}_{full} \cup \mathbf{C}_{partial}$ (Lemma 4.3). Then we show that aggregation over partially covering cluster sets yields *sound* bounds for the result of a filter-aggregate query for any candidate $Q_{fix} \in \mathbb{Q}$. That is, for every $Q_{fix} \in \mathbb{Q}$ and filter-aggregate query $Q^{\omega}_i := \gamma(a)(\sigma_{\theta_i}(R))$ for $R = Q_{fix}(D)$ we have that the bounds computed for Q^{ω}_i contain the result of Q^{ω}_i evaluated over R (Lemma 4.4). Finally, we demonstrate that for a partially covering cluster set $(\mathbf{C}_{full}, \mathbf{C}_{partial})$ and constraint ω, ACEVAL$_{\forall}$ returns true (ACEVAL$_{\exists}$ returns false) returns true if all repair candidates in $\mathbb{Q}$ are guaranteed to fulfill (not fulfill) the constraint (Lemma 4.5). Using these lemmas we then prove Theorem 4.1.

Algorithm 1 takes as input the kd-tree and a condition θ' and returns a set of clusters $\mathbf{C}$. We first show that $\mathbf{C}$ cover precisely the set of tuples from D which fulfill the condition θ'.

Lemma 4.1 (Algorithm 1 Returns Covering Clustersets). *Consider a repair candidate Q_{fix} with selection condition θ' and input relation R.*[2] *Let $\mathbf{C}$ be the clusterset returned by Algorithm 1. We have*

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

Proof. The input condition θ' is a conjunction of the form:

$$\bigwedge_{i \in [1,m]} \theta'_i$$

where θ'_i is a comparison of the form $a_i \text{ op}_i\ c_i$ and op_i is one of the supported comparison operators. Recall that Algorithm 1 traverses the kd-tree from the root maintaining a list of clusters that are fully covered, i.e., all tuples in the cluster fulfill condition θ'. For each cluster C and attribute a we store BOUNDS$_a(C) = [\underline{a}, \overline{a}]$ where $\underline{a}$ is the smallest a value in C and $\overline{a}$ is the largest a value:

$$\underline{a} = \min\left(\{t.a \mid t \in C\}\right)$$
$$\overline{a} = \max\left(\{t.a \mid t \in C\}\right)$$

For each cluster the algorithm uses the attribute ranges stored for each cluster C to distinguish three cases: (i) all tuples in C the fulfill the condition θ': $\forall t \in C : t \models \theta'$, (ii) no tuples in C fulfill the condition (the cluster can be ignored): $\forall t \in C : t \not\models \theta'$, and (iii) some tuples in C may fulfill the condition. Clusters for which case (i) applies are added to the result, clusters for which case (ii) applies are ignored, and for case (iii) the algorithms proceeds to the children of the cluster in the kd-tree. Note that the algorithm is sound, but not complete, in the sense that it may fail to determine that case (i) or (ii) applies and classify a cluster as case (iii) instead.

[2]If the user query contains joins, we treat the join result as the input relation R.

This does not affect the correctness of the algorithm as for case (iii) the algorithm processes all children of the cluster. As the children of a cluster cover exactly the same tuples as the cluster itself, this approach is guaranteed to return a set of clusters that exactly cover the tuples in $\sigma_{\theta'}(R)$ as long as the algorithm can correctly determine for a given cluster which case applies. For that the algorithm uses function $\textbf{eval}_\forall$ that takes as input a comparison $a \,\text{op}\, c$ and bounds $\textsc{bounds}_a(C)$ for the values of a in the cluster C. Thus, what remains to be shown is that for each condition θ'_i if $\textbf{eval}_\forall(\theta'_i, \textsc{bounds}_a(C))$ returns true given the bounds $\textsc{bounds}_a(C)$ for the attribute a used in θ'_i, then $\forall t \in C : t \models \theta'_i$. We prove this through a case distinction over the supported comparison operators. Recall that the definition of $\textbf{eval}_\forall$ is shown in Table 1.

Order relations ($a <, \leq, \geq, > c$). We prove the claim for $<$. The remaining cases are symmetric or trivial extensions. $\textbf{eval}_\forall$ returns true if $\overline{a} < c$. In this case for every tuple $t \in C$ we have $t.a \leq \overline{a} < c$ which implies $t.a < c$ and in turn $t \models \theta'_i$.

Equality ($a = c$). $\textbf{eval}_\forall$ return true if $\underline{a} = \overline{a} = c$. In this case we have $t.a = c$ for every tuple $t \in C$. Thus, $t \models \theta'_i$.

Inequality ($a \neq c$). $\textbf{eval}_\forall$ return true if $c \notin [\underline{a}, \overline{a}]$. Consider a tuple $t \in C$. We know $t.a \in [\underline{a}, \overline{a}]$. Using $c \notin [\underline{a}, \overline{a}]$ it follows that $t.a \neq c$.

Interval membership ($a \in [c_1, c_2]$). $\textbf{eval}_\forall$ return true if $c_1 \leq \underline{a} \wedge \overline{a} \leq c_2$. For every tuple $t \in C$ we have $t.a \in [\underline{a}, \overline{a}]$ and, thus, also $t.a \in [c_1, c_2]$. Thus, $t \models \theta'_i$. □

Using Lemma 4.1, we demonstrate that for a given repair candidate Q_{fix}, any filter-aggregate query evaluated using the materialized aggregation results for the clusters returned by Algorithm 1 for Q_{fix} are the same as aggregation function results computed over the result of evaluating Q_{fix} on the input database D. Recall that for **avg** we compute **sum** and **count** and then calculate the average as $\frac{\textbf{sum}}{\textbf{count}}$.

Lemma 4.2 (Aggregate Results on Fully Covering Clustersets). *Consider a repair candidate Q_{fix} with condition θ', database D, and filter-aggregate query $Q^\omega := \gamma_{(a)}(\sigma_{\theta''}(Q(D)))$. Let $\mathbf{C}$ be the result returned for Q_{fix} by Algorithm 1 as shown in Lemma 4.1. We have:*

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

where $\mathbf{f}'$ is chosen based on $\mathbf{f}$:

- $\mathbf{f}' = \textbf{sum}$ *for* $\mathbf{f} = \textbf{sum}$ *and* $\mathbf{f} = \textbf{count}$
- $\mathbf{f}' = \textbf{min}$ *for* $\mathbf{f} = \textbf{min}$
- $\mathbf{f}' = \textbf{max}$ *for* $\mathbf{f} = \textbf{max}$

Proof. We demonstrated in Lemma 4.1 that $\mathbf{C}$ contains exactly the set of tuples returned by $\sigma_{\theta'}(D)$. Recall that for each Q^ω we materialize the result $Q^\omega(C)$. It is well-known that aggregation functions **sum**, **min**, and **max** are associative and commutative. Thus, computing these aggregation functions over $\sigma_{\theta''}(Q(D))$ can be decomposed into computing the aggregation function results over the results for each cluster in the covering cluster set returned by Algorithm 1. Thus, the lemma holds for these aggregation functions. Aggregation function **count**() can be expressed alternatively as a sum over a constant value 1 per tuple. Thus, the same decomposition applies. As $\textbf{avg} = \frac{\textbf{sum}}{\textbf{count}}$, we compute the sum and count and use these to compute the average. □

Next, we demonstrate that Algorithm 2 returns a partially covering cluster set.

Lemma 4.3 (Partially Covering Cluster Sets). *Consider a set of repair candidates $\mathbb{Q}$ and let $(\mathbf{C}_{full}, \mathbf{C}_{partial})$ be the result returned by Algorithm 2 for $\mathbb{Q}$. The following holds for every $Q_{fix} \in \mathbb{Q}$ with condition θ':*

$$\bigcup_{C \in \mathbf{C}_{full}} C \;\subseteq\; \sigma_{\theta'}(D) \;\subseteq\; \bigcup_{C \in \mathbf{C}_{full} \cup \mathbf{C}_{partial}} C$$

Proof. Algorithm 2 takes as input a set of repair candidates $\mathbb{Q}$ encoded as a list of ranges, one for each condition θ_i in the selection condition θ of the input query Q. It keeps a stack of clusters to be processed that is initialized with the root of the kd-tree C_{root}. In each iteration, a single cluster C_{cur} is checked. The algorithm uses variables in and pin to distinguish between 3 cases for C_{cur}. Either (i) all tuples in C_{cur} are guaranteed to fulfill the selection conditions of every repair candidate $Q_{fix} \in \mathbb{Q}$; (ii) none of the tuples in C_{cur} fulfill the condition of any repair candidate $Q_{fix} \in \mathbb{Q}$; (iii) some tuples may fulfill the condition of some repair candidate $Q_{fix} \in \mathbb{Q}$.

Analog to determining fully covering cluster sets, the tests used to determine these cases are only sound, in that for some clusters for which case (i) or (ii) applies, the algorithm may fail to detect that and treat the cluster as case (iii). Again this is safe, as it is sufficient to compute lower and upper bound on aggregation results and, then, constraints.

In case (i) the cluster is added to C_{full} as all tuples from the cluster will be in $\sigma_{\theta'}(R)$ for every $Q_{fix} \in \mathbb{Q}$. For case (ii) the cluster can be discarded as it none of the tuples in C_{cur} are in $\sigma_{\theta'}(D)$ for any $Q_{fix} \in \mathbb{Q}$. Finally, in case (ii) the algorithm processes the children of C_{cur} to potentially identify smaller clusters covering parts of the tuples from C_{cur} that are fully included (case (i)), or excluded (case (ii)). The exception are partially covered (case (iii)) leaf clusters which are included in $\mathbf{C}_{partial}$. As long as the tests for case (i) and (ii) are sound, the claim trivially holds as $\sigma_{\theta'}(D)$ contains $\bigcup_{C \in \mathbf{C}_{full}} C$ for any repair candidate $Q_{fix} \in \mathbb{Q}$ where θ' is the condition of Q_{fix} and only tuple in $\bigcup_{C \in \mathbf{C}_{full} \cup \mathbf{C}_{partial}} C$ can be in $\sigma_{\theta'}(D)$.

The algorithm maintains two variables in and pin to determine which case applies for C_{cur}. We will prove that if $in = \textbf{true}$ and $pin = \textbf{true}$ then case (i) applies, if $in = \textbf{false}$ and $pin = \textbf{false}$ then case (ii) applies, and otherwise (which will be the case if $in\textbf{false}$ and $pin = \textbf{true}$) then case (iii) applies. The algorithm sets $in = \textbf{true}$ if for all conditions $\theta_i = a_i \,\text{op}_i\, c_i$ in the condition θ of the user query Q, function $\textbf{reval}_\forall$ returns true. Analog $pin = \textbf{true}$ if the same holds using function $\textbf{reval}_\exists$. Thus, it remains to be shown that given the bounds $\textsc{bounds}_{a_i}(C_{cur})$ for the values of a_i of all tuples in C_{cur} and ranges $[\underline{c_i}, \overline{c_i}]$ of constants for repair candidates, that (a) if $\textbf{reval}_\forall$ returns true, then for every $Q_{fix} \in \mathbb{Q}$ with constant $c_i \in [\underline{c_i}, \overline{c_i}]$ and tuple $t \in C_{cur}$ we have that $t.a_i \,\text{op}_i\, c_i$ evaluates to true and (b) if $\textbf{reval}_\exists$ return false then for every $Q_{fix} \in \mathbb{Q}$ and every $c \in [\underline{c_i}, \overline{c_i}]$ and tuple $t \in C_{cur}$ we have that $t.a_i \,\text{op}_i\, c_i$ evaluates to false. As clearly under these conditions $in \Rightarrow pin$ ($\neg pin \Rightarrow \neg in$ as (a) is a more restrictive condition than (b), the claim then immediately follows. Recall that Table 1 shows the definitions of $\mathbf{reval}_{\forall}$ and $\mathbf{reval}_{\exists}$.

Claim (a): $\mathbf{reval}_{\forall}$. We prove the claim for each of the supported comparison operators.

Order relations ($a <, \leq, \geq, > [\underline{c}, \overline{c}]$). We prove the claim for $<$. The remaining cases are symmetric or trivial extensions. $\mathbf{reval}_{\forall}$ returns true if $\overline{a} < \underline{c}$. In this case for every tuple $t \in C_{cur}$ and repair candidate with constant $c \in [\underline{c}, \overline{c}]$, i.e., condition $\theta'_i := a < c$, we have $t.a \leq \overline{a} < \underline{c} \leq c$ which implies $t.a < c$ and in turn $t \models \theta'_i$.

Equality ($a = c$). $\mathbf{reval}_{\forall}$ return true if $\underline{a} = \underline{c} = \overline{c} = \overline{a}$. This then implies $c = \underline{c} = \overline{c}$ as $c \in [\underline{c}, \overline{c}]$. We get $t.a = c$ for every tuple $t \in C$. Thus, $t \models \theta'_i$.

Inequality ($a \neq c$). $\mathbf{reval}_{\forall}$ returns true if $c \notin [\underline{a}, \overline{a}] \cap [\underline{c}, \overline{c}]$. Consider a tuple $t \in C$ and repair candidate with constant c. We know $t.a \in [\underline{a}, \overline{a}]$ and $c \in [\underline{c}, \overline{c}]$. Using $c \notin [\underline{a}, \overline{a}] \cap [\underline{c}, \overline{c}]$ it follows that $t.a \neq c$.

Interval membership ($a \in [[\underline{c_1}, \overline{c_1}], [\underline{c_2}, \overline{c_2}]]$). $\mathbf{reval}_{\forall}$ return true if

$$\overline{c_1} \leq \underline{a} \wedge \overline{a} \leq \underline{c_2} \tag{2}$$

For every tuple $t \in C$ and repair candidate with constants $[c_1, c_2]$ we have $t.a \in [\underline{a}, \overline{a}]$, $c_1 \in [\underline{c_1}, \overline{c_1}]$, and $c_2 \in [\underline{c_2}, \overline{c_2}]$. Then Equation (2) implies:

$$c_1 \leq \overline{c_1} \leq \underline{a} \leq t.a \leq \overline{a} \leq \underline{c_2} \leq c_2$$

Thus, also $t.a \in [c_1, c_2]$ and $t \models \theta'_i$.

Claim (b): $\mathbf{reval}_{\exists}$. The proof for $\mathbf{reval}_{\exists}$ is analog replacing universal with existential conditions, i.e., there may exist $t \in C_{cur}$ and $Q_{fix} \in \mathbb{Q}$ such that the condition θ'_i for Q_{fix} holds on t. □

Given such a partially covering cluster set, we can compute sound bounds on the results of a filter-aggregate query as shown below.

Lemma 4.4 (Sound Bounds on Aggregation Results). *Consider a set of repair candidates $\mathbb{Q}$ and filter-aggregate query Q_i^{ω}. Let $(\mathbf{C}_{full}, \mathbf{C}_{partial})$ be the result returned by Algorithm 2 for $\mathbb{Q}$. Furthermore, let $\underline{Q_i^{\omega}}$ and $\overline{Q_i^{\omega}}$ be the bounds computed for the result of Q_i^{ω}. For every $Q_{fix} \in \mathbb{Q}$ with condition θ' we have:*

$$\underline{Q_i^{\omega}} \leq Q_i^{\omega}(Q_{fix}(D)) \geq \overline{Q_i^{\omega}}$$

Proof. We proof the claim through a case distinction over the supported aggregation functions **sum**, **min**, **max**, **count** and **avg** using Lemma 4.3. For $\underline{Q_i^{\omega}}$ we have to reason about the smallest aggregation result that can be achieved by including all tuples from clusters in $\mathbf{C}_{full}$ that fulfill the condition θ' of the filter-aggregate query Q_i^{ω} and a subset (possibly empty) of tuples from cluster in $\mathbf{C}_{partial}$. Analog for $\overline{Q_i^{\omega}}$ we have to determine the maximal aggregation result achievable. We first determine $(\mathbf{C}_{full}, \mathbf{C}_{partial})$ using the filter condition θ'' of the repair candidate. Recall that we materialize aggregation results for each Q_i^{ω} and each cluster C.

$\underline{\mathbf{min}(a)}$. For $\underline{Q_i^{\omega}}$, the smallest possible result is bound from below by including the minimum lower bound of a across $\mathbf{C}_{full}$ and $\mathbf{C}_{partial}$ as the value of any tuple that may contribute to the aggregation result is bound from below by

$$\underline{Q_i^{\omega}} = \mathbf{min}\left(\{Q_i^{\omega}(C) \mid C \in (\mathbf{C}_{full} \cup \mathbf{C}_{partial})\}\right)$$

For $\overline{Q_i^{\omega}}$, the maximum result is achieved by excluding all tuples from clusters in $\mathbf{C}_{partial}$ as **min** is antimonotone and, thus, including more tuples can only lower the minimum:

$$\overline{Q_i^{\omega}} = \mathbf{min}\left(\{Q_i^{\omega}(C) \mid C \in \mathbf{C}_{full}\}\right)$$

$\underline{\mathbf{max}(a)}$. Aggregation function **max** is monotone and, thus, the computation is symmetric to the one for **min**:

$$\underline{Q_i^{\omega}} = \mathbf{max}\left(\{Q_i^{\omega}(C) \mid C \in \mathbf{C}_{full}\}\right)$$
$$\overline{Q_i^{\omega}} = \mathbf{max}\left(\{Q_i^{\omega}(C) \mid C \in (\mathbf{C}_{full} \cup \mathbf{C}_{partial})\}\right)$$

$\underline{\mathbf{sum}(a)}$. For **sum** the lowest possible sum is bound from below by first computing the sum of the lower bound of each cluster multiplied by the number of tuples in the cluster over the clusters in $\mathbf{C}_{full}$. Including tuples from a cluster C in $\mathbf{C}_{partial}$ in the sum can only lower the sum if there may be tuples in the cluster that have negative values in attribute a which is the case if $\underline{a} < 0$ for this cluster. To be able to determine the greatest negative contribution we store for each cluster $Q_i^{\omega -}(C)$ and $Q_i^{\omega +}(C)$. Recall that within the context of this proof we use θ' to denote the condition of Q_i^{ω}. We use $\{\!\{\}\!\}$ to denote multisets and $\uplus$ to denote multiset union. Then we define:

$$Q_i^{\omega -}(C) = \mathbf{sum}(\{\!\{t.a \mid t \in Ct.a < 0 \wedge t \models \theta'\}\!\})$$
$$Q_i^{\omega +}(C) = \mathbf{sum}(\{\!\{t.a \mid t \in Ct.a > 0 \wedge t \models \theta'\}\!\})$$

Note that here we define $\mathbf{sum}(\emptyset) = 0$. Based on these observations we get:

$$\underline{Q_i^{\omega}} = \mathbf{sum}\left(\{\!\{Q_i^{\omega}(C) \mid C \in \mathbf{C}_{full}\}\!\} \uplus \{\!\{Q_i^{\omega -}(C) \mid C \in \mathbf{C}_{partial}\}\!\}\right)$$

To upper bound the maximal achievable sum, we again include aggregation results for clusters in $\mathbf{C}_{full}$ and add aggregation results over the subset of tuples in a partially covered cluster that have positive a values.

$$\overline{Q_i^{\omega}} = \mathbf{sum}\left(\{\!\{Q_i^{\omega}(C) \mid C \in \mathbf{C}_{full}\}\!\} \uplus \{\!\{Q_i^{\omega +}(C) \mid C \in \mathbf{C}_{partial}\}\!\}\right)$$

$\underline{\mathbf{count}}$. Function **count** is a special case of **sum** where each tuple contributes 1 to the sum. Hence for the lower bound no clusters from $\mathbf{C}_{partial}$ are included and for the upper bound all clusters from $\mathbf{C}_{partial}$ are included:

$$\underline{Q_i^{\omega}} = \mathbf{sum}\left(\{\!\{Q_i^{\omega}(C) \mid C \in \mathbf{C}_{full}\}\!\}\right)$$
$$\overline{Q_i^{\omega}} = \mathbf{sum}\left(\{\!\{Q_i^{\omega}(C) \mid C \in \mathbf{C}_{partial}\}\!\}\right)$$

$\underline{\mathbf{avg}(a)}$. For **avg**, we compute bounds on $\mathbf{sum}(a)$ and **count** and then compute bounds for $\frac{\mathbf{sum}(a)}{\mathbf{count}}$ using the rules for arithmetic over bounds that we prove correct in the proof of Lemma 4.5. □

Based on the sound bounds on the results of filter-aggregate queries, we can derive sound bounds for arithmetic expressions over the results of such queries. Based on such bounds, we then derive bounds on the results of aggregate constraints.

Lemma 4.5 (Universal and Existential Constraint Checking is Sound). *Let* $\mathbb{Q} = [[\underline{c_1}, \overline{c_1}], \ldots, [\underline{c_m}, \overline{c_m}]]$ *be a set of repair candidates,* ω *an aggregate constraint, and* $(\mathrm{C}_{full}, \mathrm{C}_{partial})$ *be a partially covering cluster set for* $\mathbb{Q}$*. Then the following holds:*

$$\text{ACEVAL}_{\forall}(\omega, \mathrm{C}_{full}, \mathrm{C}_{partial}) = \mathbf{true} \Rightarrow \forall Q \in \mathbb{Q} : Q \textit{ is a repair}$$

$$\text{ACEVAL}_{\exists}(\omega, \mathrm{C}_{full}, \mathrm{C}_{partial}) = \mathbf{false} \Rightarrow \forall Q \in \mathbb{Q} : Q \textit{ is not a repair}$$

Proof. Consider an aggregate constraints $\omega := \Phi \,\mathrm{op}\, \tau$ where Φ is an arithmetic expression over the results of the filter-aggregate queries used in the constraint. We have shown in Lemma 4.4 that the bounds we compute for the results of filter-aggregate queries are sound. Both $\text{ACEVAL}_{\forall}$ and $\text{ACEVAL}_{\exists}$ use interval arithmetic [17, 35] to compute ranges that bound all possible results of arithmetic expressions based on ranges bounding their inputs. The soundness of these rules has been proven in related work (e.g., [20, 41]). It is sufficient to demonstrate that the semantics for individual operators / functions is sound as the composition of two functions that are sound is guaranteed to be sound. In summary, given the bounds on filter-aggregate query results, we can compute sound bounds $[\underline{\Phi}, \overline{\Phi}]$ for the result of the arithmetic part Φ of the aggregate constraint ω.

$\underline{\textbf{ACEVAL}_{\forall}}$. Function $\text{ACEVAL}_{\forall}$ first computes $[\underline{\Phi}, \overline{\Phi}]$ and then uses $\mathbf{reval}_{\forall}$ with the condition $\Phi \,\mathrm{op}\, \tau$ of the aggregate constraint to check whether the constraint is guaranteed to evaluate to true for all repair candidates $Q_{fix} \in \mathbb{Q}$. As we have already shown that if $\mathbf{reval}_{\forall}$ returns true, then the condition holds for any constants within the input bounds, we know that if $\text{ACEVAL}_{\forall}$ returns true, then every $Q_{fix} \in \mathbb{Q}$ is a repair.

$\underline{\textbf{ACEVAL}_{\exists}}$. Function $\text{ACEVAL}_{\exists}$ also computes $[\underline{\Phi}, \overline{\Phi}]$ and then uses $\mathbf{reval}_{\exists}$ with the condition $\Phi \,\mathrm{op}\, \tau$ to check whether some repair candidate $Q_{fix} \in \mathbb{Q}$ may be a repair. As we have already shown that if $\mathbf{reval}_{\exists}$ returns false, then for all constants within the input bounds the condition evaluates to false, the claim follows immediately. □

Using the preceding lemmas, we are now ready to prove Theorem 4.1.

Proof of Theorem 4.1. **Correctness of Full Cluster Filtering (FF)**. Algorithm FF enumerates all repair candidate in increasing order of their distance from the user query (breaking ties arbitrarily) and for each candidate evaluates the aggregate constraint using a covering cluster set computed using Algorithm 1 and then returns the first k repair candidates that fulfill the aggregate constraint. As we have shown in Lemma 4.2 that Algorithm 1 returns a set of clusters that precisely covers the tuples returned by the repair candidate and that computing the results of filter-aggregate queries computed using the materialized aggregation results for each cluster yields the same result as evaluating the filter-aggregate query on the result of the repair candidate, FF correctly identifies repairs.

Correctness of Cluster Range Pruning (RP) (Algorithm 2). Algorithm 2 maintains a queue of repair candidate sets to be processed that is initialized with a single set covering all repair candidates which is sorted based on the lower bound on of the distance of repair candidates in the set to the user query.

In each iteration, the algorithm pops one repair candidate set $\mathbb{Q}_{cur}$ from the queue and computes a partially covering cluster set for $\mathbb{Q}_{cur}$. If $\text{ACEVAL}_{\forall}$ returns true on this partially covering cluster set then all candidates in $\mathbb{Q}_{cur}$ are guaranteed to be repairs (Lemma 4.5) and the set is added to list *rcand*. Otherwise, the algorithm checks whether some candidates in $\mathbb{Q}_{cur}$ may be repairs using $\text{ACEVAL}_{\exists}$. From Lemma 4.5 we know that if $\text{ACEVAL}_{\exists}$ returns false, then none of the candidates in $\mathbb{Q}_{cur}$ can be repairs and $\mathbb{Q}_{cur}$ can be discarded.

Otherwise, the algorithm splits $\mathbb{Q}_{cur}$ into multiple subsets such that the union of these subsets is $\mathbb{Q}_{cur}$ and pushes them onto the work queue. Note that hasCandidates is used to check whether at least one repair candidate in each new set $\mathbb{Q}_{new}$ that differs in terms of returned tuples from at least one repair candidate outside the set. This check is based on the observation that for certain comparison operators, different thresholds are guaranteed to yield the same result and for some thresholds we are guaranteed to get an empty result. For instance, $a = c$ is guaranteed to evaluate to false if c does not appear in attribute a.

In each iteration the algorithm updates the concrete set of repairs Q_{top-k} that are closest to the user query. Furthermore, in each iteration, the algorithm peeks at the next repair candidate set $\mathbb{Q}_{next}$ in *queue*. If we have found at least k repairs and the distance of the furthest repair in Q_{top-k} is smaller than the lower bound of the distance for repair candidates in $\mathbb{Q}_{next}$ then we are guaranteed to have found the top-k repairs and the algorithm terminates. The correctness of the algorithm follows based on the following observation: (i) the algorithm starts with the set of all possible repair candidates, (ii) in each iteration sets of repair candidates are only discarded if they are guaranteed to not contain any repair, (iii) in each iteration sets of repair candidates are only included in *rcand* if they are guaranteed to only contain repairs, and (iv) for sets that are neither added to *rcand* or discarded, the set $\mathbb{Q}_{cur}$ is split into multiple subsets such that their union covers exactly $\mathbb{Q}_{cur}$, i.e., no repair candidates that may be repairs are discarded. □

5 Experiments

4.3.4 Correctness We start by comparing the brute force approach and 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 on performance in Section 5.3, including dataset size and similarity of the top-k repairs to the user query. Finally, in Section 5.4, we compare with *Erica* [26] which targets group cardinality constraints.

Table 2: Bounds on applying an operator to the result of expressions E_1 and E_2 with interval bounds [41].

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

5.1 Experimental Setup

Datasets. We choose two real-world datasets of size 500K, *Adult Census Income (Income)* [21] and *Healthcare dataset (Healthcare)* [22], that are commonly used to evaluate fairness. We also utilize the *TPC-H* [36] benchmark, 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 [26] and a new query Q_3. For Income, we use Q_4 from [26] and new queries Q_5 and Q_6. Q_7 is a query with 3 predicates inspired by TPC-H's Q_2.

Constraints. For Healthcare and Income, 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. An algorithm that explores the individual repair candidates in this order has to explore the first p% of the search space. For Income (Healthcare), we determine the groups for SPD based on gender and race (race and age group). For TPC-H, we enforce the constraint ω_5 as described in Example 3. We use Ω to denote a set of ACs. Ω_6 through Ω_8 are sets of cardinality constraints for comparison with Erica. As mentioned in section 2, we present our repair methods for a single AC. However, as we already mentioned in the original submission in the experimental section (top left of page 9 in the original submission), the methods can be trivially extended to find repairs for a set / conjunction of constraints, i.e., the repair fulfills $\bigwedge_{\omega \in \Omega} \omega$. For RP (Algorithm 3), it is sufficient to replace the condition in Line 8 with $\bigwedge_{\omega \in \Omega} \text{ACEVAL}_{\forall}(\omega, \mathrm{C}_{full}, \mathrm{C}_{partial})$ and in Line 10 with $\bigwedge_{\omega \in \Omega} \text{ACEVAL}_{\exists}(\omega, \mathrm{C}_{full}, \mathrm{C}_{partial})$.

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 $\leq \mathcal{S}$ 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 settings are follows: $\mathcal{B} = 5$, $k = 7$, and $\mathcal{S} = 15$. The default dataset size is 50K tuples.

Table 3: Queries for Experimentation

	`SELECT * FROM Healthcare`
Q_1	`WHERE income >= 200K AND num-children >= 3 AND county <= 3`
Q_2	`WHERE income <= 100K AND complications >= 5 AND num-children >= 4`
Q_3	`WHERE income >= 300K AND complications >= 5 AND county == 1`
	`SELECT * FROM ACSIncome`
Q_4	`WHERE working_hours >= 40 AND Educational_attainment >= 19 AND Class_of_worker >= 3`
Q_5	`WHERE working_hours <= 40 AND Educational_attainment <= 19 AND Class_of_worker <= 4`
Q_6	`WHERE Age >= 35 AND Class_of_worker >= 2 AND Educational_attainment <= 15`
Q_7	`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

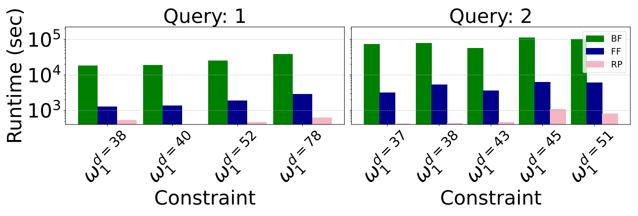
ID	Constraint
ω_1	$\frac{\textbf{count}(\text{race=1} \land \text{label=1})}{\textbf{count}(\text{race=1})} - \frac{\textbf{count}(\text{race=2} \land \text{label=1})}{\textbf{count}(\text{race=2})} \in [B_l, B_u]$
ω_2	$\frac{\textbf{count}(\text{ageGroup=1} \land \text{label=1})}{\textbf{count}(\text{ageGroup=1})} - \frac{\textbf{count}(\text{ageGroup=2} \land \text{label=1})}{\textbf{count}(\text{ageGroup=2})} \in [B_l, B_u]$
ω_3	$\frac{\textbf{count}(\text{sex=1} \land \text{PINCP} \geq 20k)}{\textbf{count}(\text{sex=1})} - \frac{\textbf{count}(\text{sex=2} \land \text{PINCP} \geq 20k)}{\textbf{count}(\text{sex=2})} \in [B_l, B_u]$
ω_4	$\frac{\textbf{count}(\text{race=1} \land \text{PINCP} \geq 15k)}{\textbf{count}(\text{race=1})} - \frac{\textbf{count}(\text{race=2} \land \text{PINCP} \geq 15k)}{\textbf{count}(\text{race=2})} \in [B_l, B_u]$
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	$\omega_{81} := \textbf{count}(\text{Sex} = \text{Female}) \leq B_{u_1}$
	$\omega_{82} := \textbf{count}(\text{Race} = \text{Black}) \leq B_{u_2}$
	$\omega_{83} := \textbf{count}(\text{Marital} = \text{Divorced}) \leq B_{u_3}$

All algorithms were implemented in Python. 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 we report median runtimes as the variance is low ($\sim$ 3%).

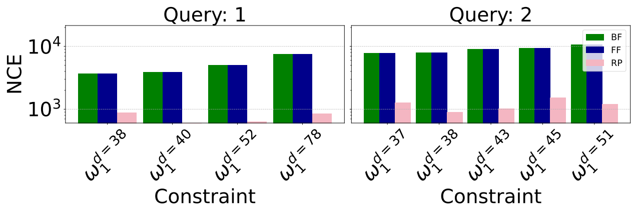
5.2 Performance of FF and RP

We compare FF and RP using datasets Healthcare (ACs ω_1 and ω_2 from Table 4) and Income (ACs ω_3 and ω_4) with the default parameter settings and queries Q_1, Q_2, and Q_3 (Table 3). 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.

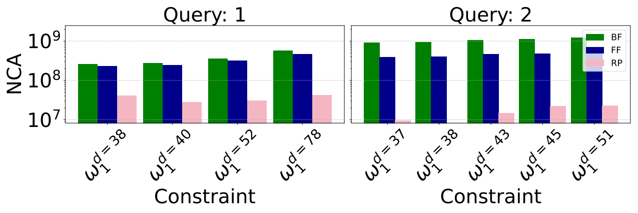
Comparison with Brute Force. We first compare FF and RP with the Brute Force (BF) method on the Healthcare dataset using queries Q_1 and Q_2, the constraint ω_1 and default settings in Section 5.1. As expected, both FF and RP outperform BF by at least one order of magnitude in terms of runtime as shown in Figure 2a. The RP algorithm significantly reduces both NCE and NCA, while the FF method maintains the same NCE as BF but decreases the NCA



(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.

compared to BF (as BF does not use clusters we count tuple accesses) as in Figure 2c and Figure 2b.

Runtime. Figures 3a and 3b show the runtime of the FF and RP algorithms for Healthcare and Income, respectively. For a subset of the experiments we also report results for the BF method. 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 the algorithms 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. We refer to this as the exploration distance (ED). As expected, both FF and RP outperform BF by at least one order of magnitude in terms of runtime as shown in Figure 3a. The RP algorithm significantly reduces both NCE and 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 tuple accesses). RP (pink bars) generally outperforms FF (blue bars) for most settings, demonstrating an additional improvement of up to an order of magnitude due to its capability of pruning and confirming sets of candidates at once. The only exception are settings where the top-k repairs are found by exploring a very small portion of the search space, e.g., Q_4 with ω_3. In Figure 3b, both 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 runtime in Section 5.3.

Total number of candidates evaluated (NCE). We further analyze how NCE affects the performance of our methods (Figures 3c and 3d) RP consistently checks fewer candidates compared to FF. As observed in the runtime evaluation, the difference between the two algorithms is more pronounced when larger parts of the search space have to be explored. For example, as shown in Figure 3d for Q_4 with $\omega_3^{d=2}$ and Q_5 with $\omega_3^{d=3}$.

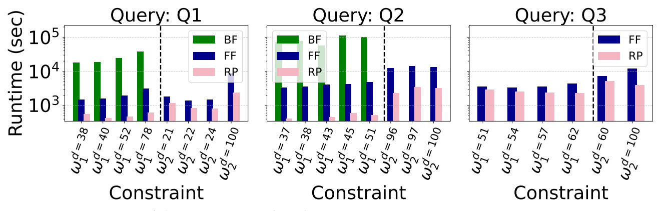
Total Number of Cluster Accessed (NCA). The results for the number of clusters accessed are shown in Figures 3e and 3f for Healthcare and Income, respectively. Similar to the result for NCE, RP accesses significantly fewer clusters than FF.

5.3 Performance-Impacting Factors

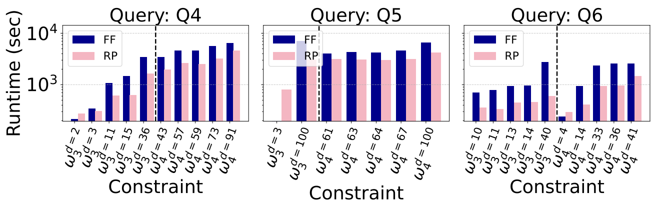
To gain deeper insights into the behavior observed in Section 5.2, we investigate the relationship between the exploration distance (ED) and performance. We also evaluate the performance of FF and RP in terms of the parameters from Section 5.1.

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 Income and vary the bounds to control for ED. The result is shown in Figure 4a for Healthcare and in Figure 4b for Income. 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 very low ED 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 ??. The reason behind this trend is that when solutions are closed to the user query (smaller ED), then there is less opportunity for pruning for RP.

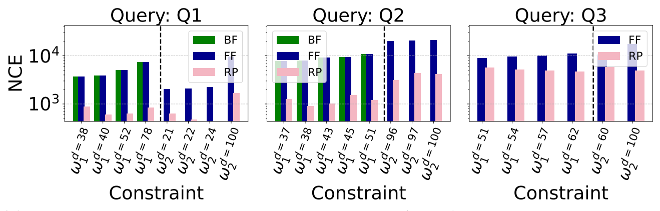
Effect of Bucket Size. We now evaluate the runtime of FF and RP varying the bucket size $\mathcal{S}$ using Q_1 with ω_1 with bounds $[0.44, 0.5]$ for the Healthcare dataset and Q_4 with ω_3 using bounds $[0.34, 0.39]$ for the Income dataset. We vary the $\mathcal{S}$ from 5 to 2500. We use the default branching factor $\mathcal{B} = 5$. For this branching factor the the structure of the kd-tree for 50k tuples 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. Our algorithm chooses the number of levels to ensure that the size of leaf clusters is $\leq \mathcal{S}$. 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.



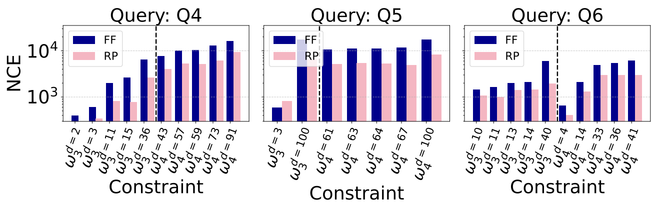
(a) Runtime (sec) - Healthcare dataset.



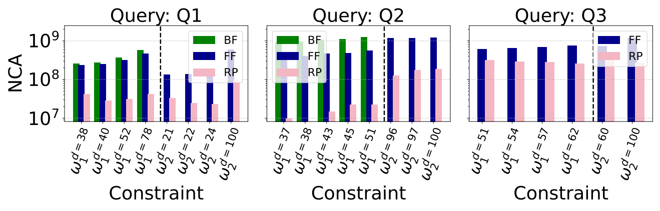
(b) Runtime (sec) - Income dataset.



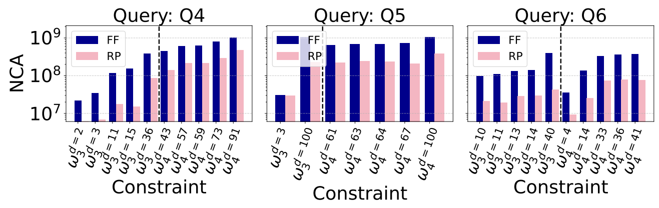
(c) Total number of constraints evaluated (NCE) - Healthcare dataset.



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

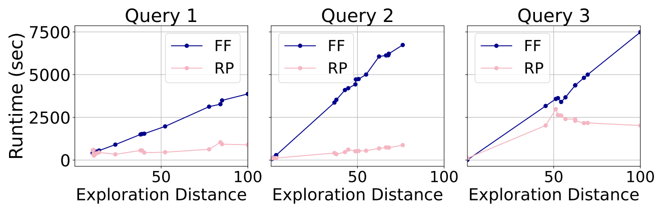


(e) Total number of cluster accessed (NCA) - Healthcare dataset.

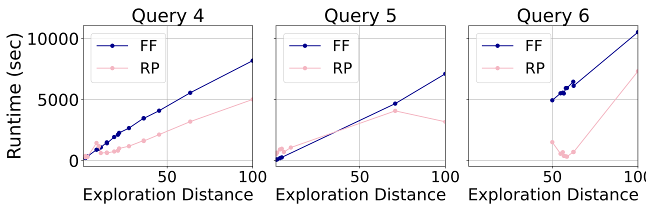


(f) Total number of cluster accessed (NCA) - Income dataset.

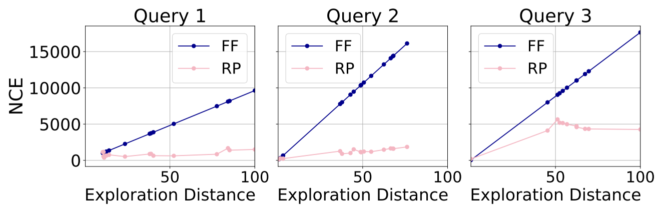
Figure 3: Runtime, NCE, and NCA for FF and RP over the Healthcare and Income datasets using the queries from Table 3.



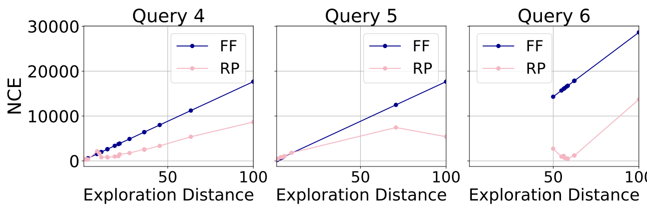
(a) Runtime of FF and RP, varying ED - Healthcare dataset.



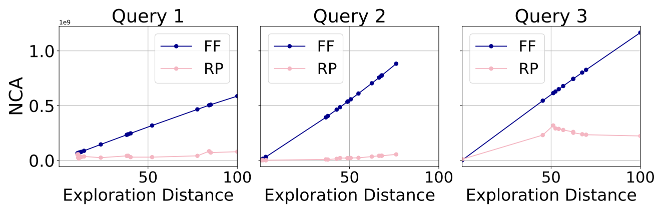
(b) Runtime of FF and RP, varying ED - Income dataset.



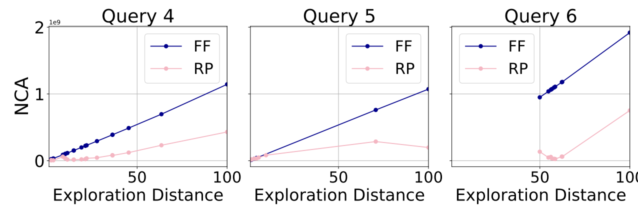
(c) Total number of constraints evaluated (NCE) - Healthcare dataset.



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

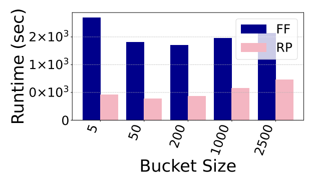


(e) Total number of cluster accessed (NCA) - Healthcare dataset.

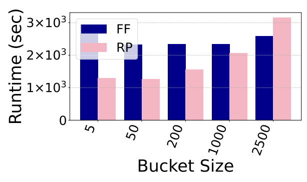


(f) Total number of cluster accessed (NCA) - Income dataset.

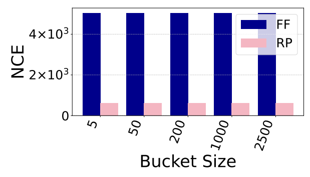
Figure 4: Runtime, NCE, and NCA for FF and RP over the Healthcare and Income datasets, varying ED.



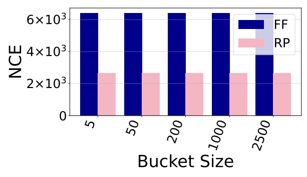
(a) Runtime - Healthcare dataset.



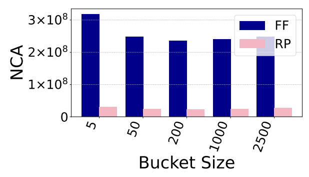
(b) Runtime - Income dataset.



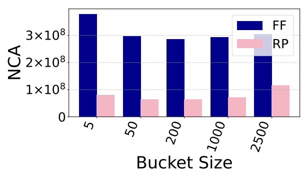
(c) Total number of constraints evaluated (NCE) - Healthcare dataset.



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

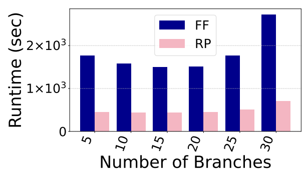


(e) Total number of cluster accessed (NCA) - Healthcare dataset.

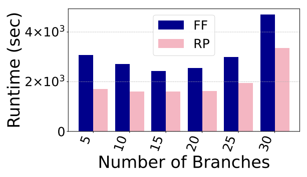


(f) Total number of cluster accessed (NCA) - Income dataset.

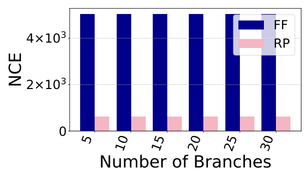
Figure 5: Runtime, NCE, and NCA for FF and RP over the Healthcare and Income datasets, varying bucket size $\mathcal{S}$.



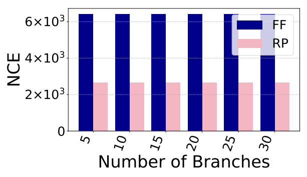
(a) Runtime - Healthcare dataset



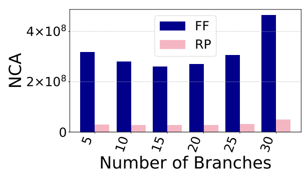
(b) Runtime - Income dataset



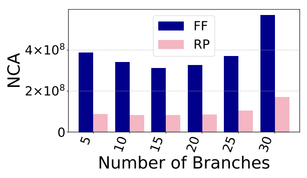
(c) Total number of constraints evaluated (NCE) - Healthcare dataset



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



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



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

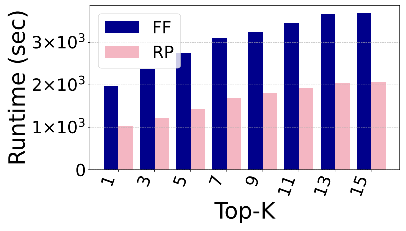
Figure 6: Runtime, NCE, and NCA for FF and RP over the Healthcare and Income datasets, varying the number of branches $\mathcal{B}$.

Table 5: Branching Configuration and Data Distribution

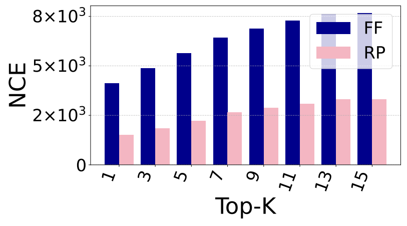
# of Branches	# of Leaves	# of Branches	# of Leaves
5	15625	20	8000
10	10000	25	15625
15	3375	30	27000

Effect of the Branching Factor. We now examines the relationship between the branching factor $\mathcal{B}$ and the runtime of FF and RP. We use the same queries, constraints, bounds, and datasets as in the previous evaluation and 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 determines the depth of the tree. The result shown in Figure 6a and Figure 6b confirms that, as expected, the performance of FF and RP correlates with the number of clusters at the leaf level. For FF, branching factors of 5 and 25 yield nearly identical runtime because both have the same number of leaf nodes (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. 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 as for smaller clusters it may be possible to prune / confirm larger candidate sets at once. Both bucket size S and branching factor B impact performance, and the optimal values depend on the characteristics of the dataset and queries. The intuition is as follows: when S is too small, the resulting tree becomes too deep, leading to an excessive number of leaf clusters; when S is too large, the ability to prune effectively diminishes because clusters encompass too many data points. Likewise, if B is too large, data is distributed across many child nodes, making it harder to prune entire sub-trees; if B is too small, the tree again becomes too deep with many leaf clusters. For new datasets, we suggest starting with moderate values for both S and B, then adjusting based on the number of leaf clusters observed: if the tree has too many leaf clusters, consider increasing S or decreasing B; if pruning is insufficient, consider decreasing S or increasing B. There are several additional factors that can affect optimal choices for these parameters: (i) strong correlations between attributes used in conditions lead to more homogeneous clusters which in turn means that larger clusters can be tolerated without significantly impacting pruning power, (ii) if attributes in user query conditions are correlated with attributes of filter-aggregate queries, then aggregation results vary widely for clusters potentially leading to a stronger separation between repairs and more pruning potential even with larger clusters. We leave automatic parameter tuning, e.g., based on measuring correlations between attributes over a sample, to future work.

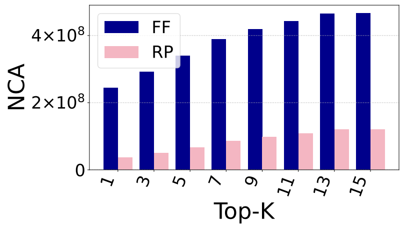
Effect of k. In this experiment, we vary the parameter k from 1 to 15. For both FF and RP, as k increases, the runtime also increases, as shown in Figure 7a. This behavior is expected since finding a single



(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.

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. 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 using Q_7 with ω_5. Dataset size impacts both the size of the search space and the size of the kd-tree. Nonetheless, as shown in Figure 8a, our algorithms scale roughly linearly in dataset size demonstrating the effectiveness of reusing aggregation results for clusters and range-based pruning. 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 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

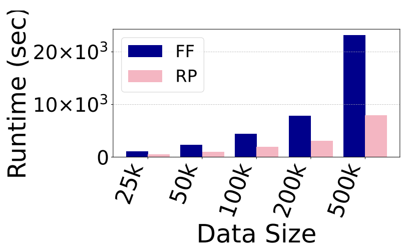
We compare our approach with Erica [26], which solves the related problem of finding all minimal refinements of a given query that satisfy a set of cardinality constraints for groups within the result set. Such constraints are special cases on the ACs we support. Erica returns all repairs that are not *dominated* (the skyline [10]) 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. Thus, different from our approach, the number of returned repairs is not an input parameter in Erica. For a fair comparison, we compute the minimal repairs and then set k such that our methods returns a superset of the repairs returned by Erica. Our algorithms, like Erica, operate by modifying constants in predicates on attributes already present in the query and do not introduce new predicates. A key difference is that Erica supports adding constants to set membership predicates for categorical attributes, e.g., replacing $A \in \{c_1\}$ with $A \in \{c_1, c_2, c_3\}$, while our approach maps categorical values to numeric codes and adjusts thresholds. As we will discuss in Section 6, both our approach and Erica can model addition of new predicates by refining dummy predicates that evaluate to true on all inputs. To conduct the evaluation for Erica, we used the available Python implementation [1][3].

We adopt the queries, constraints, and the dataset from [26]. We compare the generated refinements and runtime of our techniques with Erica using Q_1 and Q_2 (Table 3) on the Healthcare dataset (50K tuples) with constraints Ω_6 and Ω_7 (Table 4), respectively.

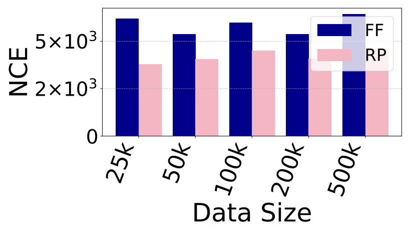
Generated Repairs Comparison. We first compare the generated repairs by our approach and Erica. As mentioned above, we did adjusted k per query and constraint set to ensure that our approach returns a superset of the repairs returned by Erica. For Q_1 with Ω_6 (Q_2 with Ω_7), Erica generates 7 (9) minimal repairs whereas our technique generates 356 (1035), including those produced by Erica. The top-1 repair returned by our approach is guaranteed to be minimal. 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 dissimilar to the user query.

Runtime Comparison. The experiment utilizes Q_4 with Ω_8 on the 50K Income 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 constraint sets Ω_8^{B1} 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 9b reveal an advantage of the RP algorithm, which outperforms Erica in the time exploring the search space to generate a repair. However, as shown in Figure 9a, 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. Erica's pre-processing is faster because it only computes provenance expressions and, for each predicate,

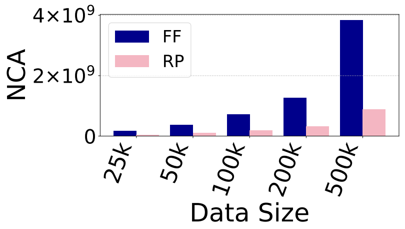
[3]We replaced Erica's DataFrame filter checks and constraints evaluation (which run in C) with equivalent pure-Python loops over lists just as in our own code, so that both implementations are using the same programming language. This change ensures our comparison highlights algorithmic differences rather than language speed.



(a) Runtime (sec).

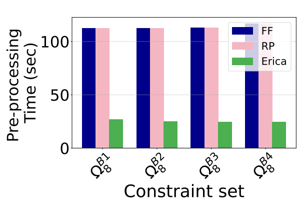


(b) Total number of constraints evaluated (NCE).

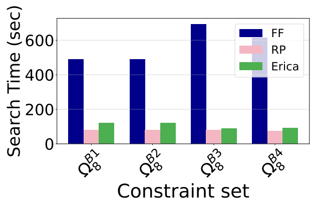


(c) Number of clusters accessed (NCA).

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



(a) Pre-processing



(b) Search

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

Table 6: Comparison of query repair techniques

Approach	**Supports sum, min, max, avg**	**Distance Metric**	**Constrains Result Sub-sets?**	**Repairs Joins?**	**Arithmetic Expressions Supported**
HC [11]	✗	✗	✗	✗	✗
TQGen [30]	✗	✗	✗	✗	✗
SnS [29]	✗	✗	✗	✗	✗
EAGER [2]	✓	*lin. comb.*	✗	✗	✗
SAUNA [23]	✗	*L2 (result)*	✗	✓	✗
ConQueR [37]	✗	*edit-distance*	✗	✓	✗
FixTed [9]	✗	*skyline*	✗	✓	✗
FARQ [34]	✗	*Jacc. (result)*	✓	✗	✗
Erica [26]	✗	*skyline*	✓	✗	✗
RP (ours)	✓	*lin. comb.*	✓	✗	✓

builds a list of candidate constants sorted by their distance to the original query. In contrast, our methods require clustering the data, indexing the clusters, and materializing summaries for each cluster, which is more computationally intensive. However, this extra work enables us to reason about complex, non-monotone constraints, which Erica's simpler list-based approach cannot. Furthermore, we argue that it can be beneficial to decrease search time at the cost of higher preprocessing time as some of the preprocessing results could be shared across user requests. Overall the total runtime of RP and Erica are comparable, even through our approach does not apply any specialized optimizations that exploit monotonicity as in Erica. 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. Table 6 summarizes several query refinement techniques for aggregate constraints, and compares their capabilities in terms of supported aggregates (only **count** or also other aggregates), distance metric use to compare repairs to the original query based on distances between predicates (e.g., *lin. comb.*: linear combination of predicate-level distances, *skyline*: skyline over predicate-level distances), whether the method allows constraints that apply only to a subset of the result (some methods only constraint the whole query result), whether join conditions can be repaired, and whether they support arithmetic expressions. Li et al. [26] 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 (*skyline*). However, [26] only supports cardinality constraints (**count**) and does not allow for arithmetic combinations of the results of such queries as shown in Table 6. Mishra et al. [29] refine a query to return a given number k of results with interactive user feedback. Koudas et al. [25] refine a query that returns an empty result to produce at least one answer. In [9, 37] a query is repaired to return missing results of interest provided by the user. Campbell et al. [13] repair top-k queries, supporting non-monotone constraints through the use of constraint solvers. [11, 30] refine queries for database testing such that subqueries of the repaired query approximately fulfill cardinality constraints. [11] demonstrated that the problem is NP-hard in the number of predicates. Both approaches do not optimize for similarity to the user query. [23] relaxes a query to return approximately N results preferring repairs based on the difference between the result of the user query and repair. 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 [12, 23, 29, 30, 40]. To the best of our knowledge, our approach is the only one that supports arithmetic constraints which is necessary to express complex real world constraints, e.g., standard fairness measures, but requires novel pruning techniques that can handle such non-monotone constraints. While some approaches explicitly support adding and deletion of predicates, any approach that can both relax or refine predicates and support deletion by relaxing a predicate until it evaluate to true on all inputs and addition by adding dummy predicates that evaluate to true on all inputs and then either refine them (adding a new predicate) or not (decide to not add this predicate).

How-to queries. Like in query repair [26], the goal of how-to queries [28] is to achieve a desired change to a query's result. However, how-to queries change the database to achieve this result instead of repairing the query. Wang et al. [39] study the problem of deleting operations from an update history to fulfill a constraint over the current database. However, this approach does not consider query repair (changing predicates) nor aggregate constraints.

Explanations for Missing Answers. Query-based explanations for missing answers [15, 18, 19] are 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 with Interval Arithmetic. Prior work has highlighted the effectiveness of interval arithmetic across various database applications [17, 20, 35, 41]. For instance, [20] determines bounds on query results over uncertain database. Similarly, the work in [41] introduced a bounding technique for iceberg cubes, establishing an early foundation for leveraging interval arithmetic to constrain aggregates. Interval arithmetic has been used extensively in *abstract interpretation* [16, 17, 35] to bound the result of computations. For example, see [17, 35] for introductions to interval arithmetic and more advanced numerical abstract domain.

7 Conclusions and Future Work

We introduce a novel approach for repairing a query to satisfy a constraint on the query's result. We support a significantly larger class of constraints than prior work, including common fairness metrics like SPD. We avoid redundant work by reusing aggregate results when evaluating repair candidates and present techniques for evaluating multiple repair candidates at once by bounding their results. Our approach works best if there is homogeneity among similar repair candidates that can be exploited. 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, (ii) considering other optimization criteria, e.g., computing a skyline as in some work on query refinement, (iii) employing more expressive domains than intervals for computing tighter bounds, e.g., zonotopes [17], and (iv) supporting dynamic settings where the table, predicates, constraints, or distance metrics may change. In this regard, we may exploit efficient incremental maintenance kd-trees and aggregate summaries [7, 8]. However, our setting is more challenging as small changes to aggregation results can affect the validity of large sets of repair candidates.

References

[1] 2024. Erica implementation. https://github.com/JinyangLi01/Query_refinement. 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 *VLDB*.

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

[7] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *CACM* 18, 9 (1975), 509–517.

[8] Jon Louis Bentley and James B Saxe. 1980. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms* 1, 4 (1980), 301–358.

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

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

[11] N. Bruno, S. Chaudhuri, and D. Thomas. 2006. Generating Queries with Cardinality Constraints for DBMS Testing. *TKDE* 18, 12 (2006), 1721–1725.

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

[13] Felix S. Campbell, Alon Silberstein, Julia Stoyanovich, and Yuval Moskovitch. 2024. Query Refinement for Diverse Top-k Selection. *SIGMOD* 2, 3 (2024), 166.

[14] Felix S. Campbell, Julia Stoyanovich, and Yuval Moskovitch. 2024. Rodeo: Making Refinements for Diverse Top-K Queries. *PVLDB* 17, 12 (2024), 4341–4344.

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

[16] 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.

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

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

[19] 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.

[20] Su Feng, Boris Glavic, Aaron Huber, and Oliver A Kennedy. 2021. Efficient uncertainty tracking for complex queries with attribute-level bounds. In *SIGMOD*. 528–540.

[21] 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.

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

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

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

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

[26] Jinyang Li, Yuval Moskovitch, Julia Stoyanovich, and HV Jagadish. 2023. Query Refinement for Diversity Constraint Satisfaction. *PVLDB* 17, 2 (2023), 106–118.

[27] 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.

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

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

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

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

[32] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. 2013. A probabilistic optimization framework for the empty-answer problem. *PVLDB* 6, 14 (2013), 1762–1773.

[33] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. 2016. A Holistic and Principled Approach for the Empty-Answer Problem. *VLDBJ* 25, 4 (2016), 597–622.

[34] Suraj Shetiya, Ian P Swift, Abolfazl Asudeh, and Gautam Das. 2022. Fairness-aware range queries for selecting unbiased data. In *ICDE*. 1423–1436.

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

[36] TPC. 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.

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

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

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

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

[41] Xiuzhen Zhang, Pauline Lienhua Chou, and Guozhu Dong. 2007. Efficient computation of iceberg cubes by bounding aggregate functions. *TKDE* 19, 7 (2007), 903–918.

[42] 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.