Q-ACER: Query Aggregate Constraint Efficient Repair System

Vaishnavi Deshpande
University of Southampton
vpd1n24@soton.ac.uk

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

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

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

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

ABSTRACT

Selection processes, e.g., determining qualified job candidates or picking vendors, can naturally be modeled as relational queries. To ensure that such a query produces legally compliant and ethically sound results, the outputs of the query are often subject to additional constraints including fairness ratios, budget limits, and coverage requirements. When such constraints are violated, users are typically left without guidance on how to repair their queries. In this demonstration, we present *Q-ACER* (Query Aggregate Constraint Efficient Repair System), an efficient query repair system that enumerates candidate repairs of an input query such that the result set of the updated query fulfills user-provided constraints. In comparison to other query repair frameworks, Q-ACER covers a significantly larger class of constraints which is necessary for supporting fairness metrics such as statistical parity. However, this necessitates the development of novel techniques for sharing computations when evaluating constraints as well as evaluating multiple query repair candidates at once.

PVLDB Reference Format:
Vaishnavi Deshpande, Seokki Lee, Shatha Algarni, Boris Glavic, and Adriane Chapman. Q-ACER: Query Aggregate Constraint Efficient Repair System. PVLDB, 19(12): 4802 - 4805, 2026.
doi:10.14778/3827998.3828126

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/UCDBG/Q-ACER.

1 INTRODUCTION

Queries are often used to select a set of suitable objects / persons from a list of candidates, e.g., which job applicant should be interviewed or which vendor to order a product from. These use cases have in common that the same criteria, the filter conditions of the query, have to be used to judge every candidate. For instance, interview job candidate with a GPA of at least 3.9 and Java skills. The selected set of candidates as a whole typically has to fulfill additional constraints, e.g., fairness ratios, budget limits, or other policies. In [1], we demonstrated that such constraints can be expressed as so-called *aggregate constraints*, conditions on arithmetic combinations over the result of aggregate-filter queries evaluated over the input query's result.

In this paper, we present *Q-ACER* (*Query Aggregate Constraint Efficient Repair*), a system that addresses the problem of repairing a user-defined query so that its *entire* result set satisfies a complex aggregate constraint. The system is built upon the scientific contributions detailed in [1] which we briefly summarize in the following. Q-ACER empowers users to express aggregate constraints and queries through a easy-to-use interface. Based on the query and constraint, the system determines repairs, modified version of the input query that differ in selection conditions from the original query and fulfill the constraint. We support predicates on both numerical and categorical attributes. To preserve the intent of the original query as much as possible, Q-ACER only returns the top-k repairs with respect to their distance to the input query. The user can then browse the top-k repairs and select which repair to implement. To help the user compare repairs, Q-ACER highlights how the repair differs from the input query. Q-ACER is available on GitHub (https://github.com/UCDBG/Q-ACER).

Prior work in this area, either only produces explanations for why the constraint is violated (e.g., query-based explanations for missing answers [2, 7]) or only supports simpler constraints such as cardinality constraints [3, 5, 8], e.g., at least three selected job candidates should be female. However, real-world applications routinely demand richer constraints that goes well beyond cardinality and require the use of arithmetic operators. For example, standard fairness metrics [4] can be expressed as ratios of aggregation results.

Example 1. *A government agency is responsible for contracting out supply of school meals. 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. The agency may use a set of criteria as shown below: low enough price, sufficient amount of calories, and within 100 miles of the school.*

```
Q1: SELECT * FROM vendor
    WHERE price < 20
      AND calories > 500
      AND distance < 100
```

Using Q-ACER, the agency can specify additional requirements as aggregate constraints and can repair Q1 *such that the fixed query's result fulfills these constraints. For that, Q-ACER enumerates candidate repairs that fulfill a given constraint and differ only minimally from* Q1 *(preserve the original intent of* Q1 *to the degree possible).*

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828126

***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:*

$$\omega_{numvendors} := \textbf{count}() \geq low \wedge \textbf{count}() < high$$

***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:*

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

***Sufficient capacity**: Assume that 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:*

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

Q-ACER supports conjunctions[1] of aggregate constraints ω of the form:

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

Here, Φ is an arithmetic expression using operators $(+, -, *, /)$ over the result of filter-aggregation queries $\{Q_i^{\omega}\}$ which are evaluated over the output of a user query, *op* is a comparison operator, and τ is a constant threshold. A filter-aggregation query, is a query $\gamma_{f(A)}(\sigma_{\theta}(Q(D)))$ where f is an aggregation function and θ can be any Boolean condition. Aggregate constraints significantly generalize the cardinality constraints supported in prior work which are insufficient to express, e.g., the ω_{carbon} and $\omega_{capacity}$ constraints shown above. Q-ACER returns the top-k repairs of a query that fulfill ω and have the lowest distance to the user query. The system is guaranteed to return all such repairs, if they exist. Consider a 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 a weighted sum of differences between constants in the selection conditions used θ_i and θ_i'. The user can set the weights to penalize changing certain predicates in the query. Note that the system supports adding or removing query predicates by modeling this as refinement [1].

Challenges & Contributions. In contrast to prior work (e.g., ERICA [3]) which exploits the monotonicity of constraints for pruning the search space, the aggregate constraints supported by Q-ACER are not monotone in general, requiring novel pruning techniques that do not rely on monotonicity. Given that the problem is NP-hard in the number m of predicates of the query to be repaired, we cannot hope to avoid exponential runtime in m in all cases [1]. In Q-ACER, we focus on conservative heuristics for pruning the search space that do not incorrectly prune valid repair candidates and are efficient in practice. The brute force approach of generating top-k repairs is to enumerates all repair candidates in order of their distance from the user query, evaluate each by running the modified query and checking the aggregate constraint until k repairs are found. However, this is computationally prohibitive because the number of candidates grows exponentially with the number of predicates in the query, and evaluating each candidate requires executing the modified query and one or more aggregate queries on top of it.

Reuse of aggregate results. Similar repair candidates often share overlapping aggregate computations. To exploit this, Q-ACER partitions the dataset into clusters using a kd-tree, with each cluster storing both precomputed aggregation results and bounds on attribute values across tuples in the cluster. When evaluating a repair candidate, these bounds are used to classify each cluster into one of three cases: all tuples satisfy the selection condition of the repair candidate (include the cluster's aggregates), none do (skip the cluster entirely), or based on the bounds we cannot determine whether one of the two earlier cases apply (recurse into the cluster's children). Q-ACER evaluates the aggregate constraint for a repair candidate by combining the precomputed aggregates of a set of clusters whose attribute values are fully included in the candidate's result set. The system can further reduces cost by computing bounds on possible aggregation function results using partially covered clusters instead of recursing to their children.

Simultaneous evaluation of multiple candidate repairs. Q-ACER further extends this idea by reasoning about entire sets of repair candidates simultaneously, rather than evaluating each candidate individually. A set of candidates is compactly represented as an interval of values for each predicate, and the kd-tree bounds are used to derive valid lower and upper bounds on the aggregate constraint result for every candidate in the set. These bounds then allow Q-ACER to either confirm all candidates in the set as valid repairs or discard the entire set at once, significantly reducing the number of candidates that need to be evaluated individually.

The Q-ACER system offers a guided query & constraint builder (no raw SQL needed), which (i) computes top-k repairs candidates with visual diffs of predicates and constraint satisfaction bars, and (ii) adds a filtered-cluster caching layer that reduces search cost from ~75 minutes to tens of seconds, enabling live "repair–inspect–refine" cycles. This demo showcases how users can: (i) construct SPJ queries and aggregate constraints over real datasets, (ii) explore top-k repaired queries that minimally deviate from the original query, (iii) interactively vary thresholds (e.g., for fairness metrics), and (iv) examine kd-tree–based clustering.

2 DEMONSTRATION SCENARIO

We illustrate Q-ACER's capabilities using an example session of a user interacting with the system's UI shown in Figure 1 for the following scenario. A state Medicaid office is responsible for enrolling patients into a medical care management program. Due to healthcare equity legislation, the agency must declare a fixed, transparent set of eligibility criteria upfront. The criteria must balance medical need, financial vulnerability, and family burden. Only patients satisfying all criteria are pre-qualified for enrollment. Simultaneously, the office was advised that no more than 30% of the selected patients should be heavy smokers (`smoker = 2`) as smoking is considered a life-style choice that may boost a patients observed medical need.

[1] Or equivalently, multiple constraints.

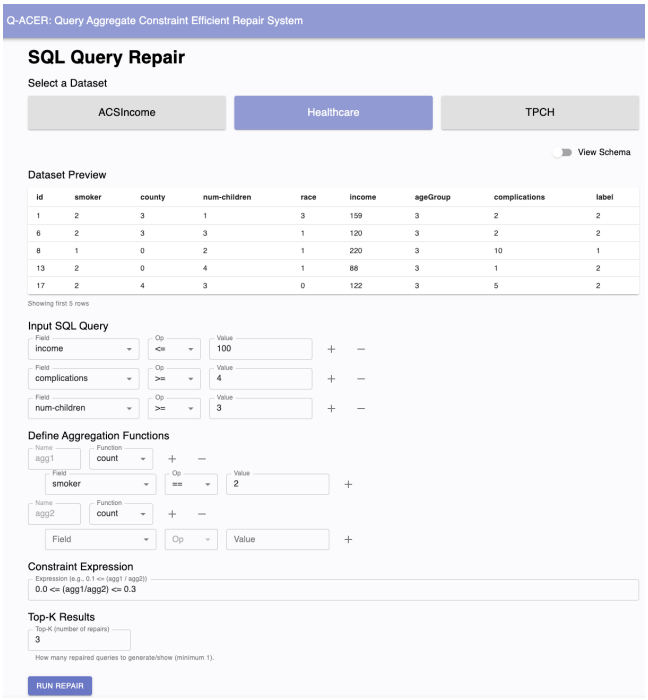


Figure 1: Q-ACER: specifying the query and constraint

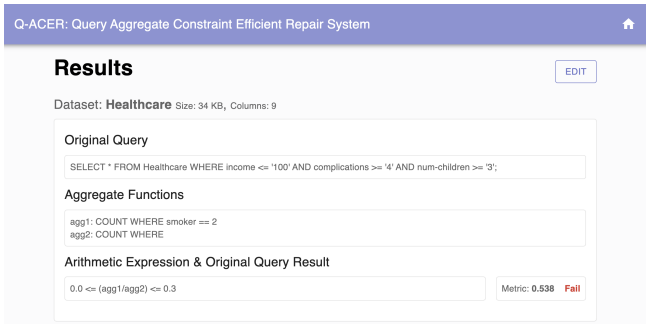
Dataset. The attributes of the dataset used by the Medicaid office include `smoker` status, `county`, `num-children`, `race`, `income`, `ageGroup`, number of recorded medical `complications`, and a `label` indicating program eligibility. Q-ACER shows five sample records as a preview for the dataset.

```
SELECT *
FROM Healthcare
WHERE income <= 100
  AND complications >= 4
  AND num-children >= 3;
```

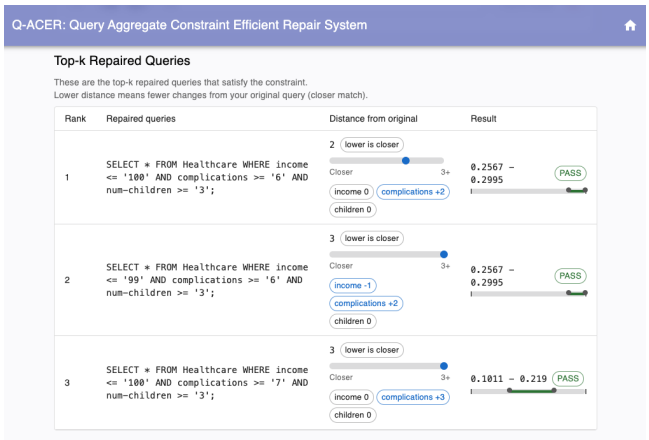
Input SQL Query. The user query filters the patient population using three predicates: financial vulnerability – income ≤ 100 (thousand dollars), medical need – complications ≥ 4, and family burden – number of children ≥ 3.

Constraint. As shown in Figure 1, the constraint on the proportion of smokers is expressed by defining two aggregate-filter queries, `agg1` and `agg2`: `agg1` counts the number of heavy smokers (`smoker = 2`) in the query result, while `agg2` counts the total number of patients in the query result. The constraint $\omega_{smokers} := 0.0 \leq$ `(agg1/agg2)` ≤ 0.3 requires that the ratio of heavy smokers to the total enrolled population is in $[0.0, 0.3]$, which ensures that heavy smokers do not constitute more than 30% of the enrolled patients. Finally, the top $k = 3$ repairs are requested, i.e., the three query repairs closest to the original query that satisfy the constraint.

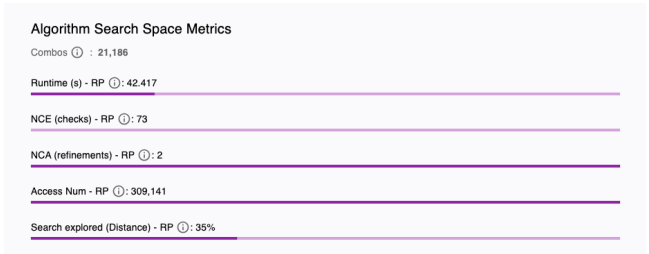
Constraint Violation (Figure 2a). Q-ACER displays a summary of the original query and the defined constraint. Evaluating the constraint expression $0.0 \leq$ `(agg1/agg2)` ≤ 0.3 on the query's result yields a fraction of heavy smokers of 0.538. The result is marked as Fail, confirming that the original prescreening criteria



(a) Constraint violation summary for the original query



(b) Top-k repairs ranked by distance from the original query



(c) Algorithm search efficiency metrics

Figure 2: Q-ACER results for the Healthcare scenario

enroll an unfairly high fraction of heavy smokers — 53.8% is well above the 30% threshold required by the Medicaid office.

Top-k Repaired Queries (Figure 2b). Q-ACER returns the top-3 repaired queries ranked by their distance from the original query, where a lower distance indicates fewer changes to the original predicates. The closest repair (Rank 1) tightens the `complications` threshold from ≥ 4 to ≥ 6 while leaving predicates on `income` and `num-children` unchanged, achieving a range $[0.2567, 0.2995]$ and earning a PASS. The second repair additionally relaxes the income threshold (from ≤ 100 to ≤ 99) while also raising the complications threshold to ≥ 6, yielding the same passing metric range. The third repair applies a stricter tightening of the complications threshold to ≥ 7, achieving a metric range of $[0.1011, 0.219]$, which satisfies the constraint with a wider margin. All three repairs modify the

complications predicate — the dominant driver of the constraint violation — while preserving the num-children condition.

Algorithm Search Efficiency Statistics (Figure 2c). The panel that appears below of the top-3 repairs reports efficiency statistics over this query repair instance. Out of 21,186 total candidate repair combinations, Q-ACER evaluates only 73 candidates (NCE), performing 2 candidate set refinements (NCA), and accessing 309,141 cluster entries. The algorithm explores only 35% of the search space before finding the top-3 repairs, completing the entire search in 42.4 seconds. These metrics illustrate the effectiveness of pruning of whole sets of candidates: rather than exhaustively checking all 21,186 combinations, Q-ACER identifies the top repairs by evaluating less than 0.4% of the total candidate set.

3 OVERVIEW OF Q-ACER

Q-ACER implements query repair under aggregate constraints as an interactive application using the algorithms from [1] with additional caching extensions. Related work such as [3] has focused on simpler monotone cardinality constraints. Dealing with aggregate constraints efficiently is the main contribution of [1].

3.1 Materializing Partial Aggregates

To determine whether the input aggregate constraint holds for a specific repair candidate Q_{fix}, we have to evaluate $Q_{fix}(D)$ and then evaluate all aggregate queries from the constraint over $Q_{fix}(D)$. As repair candidates only differ in selection conditions and many candidates will have similar conditions, there is a great potential for reuse. Q-ACER clusters the input tuples along the dimensions of the query's predicates, materializes partial aggregation results and bounds on attribute values for each cluster, and indexes these clusters in a kd-tree. To evaluate a candidate, a set of clusters is determined based on the cluster attribute bounds that covers precisely the input tuples that match the candidate's condition. The aggregation query results for the candidates can then be computed by combining the materialized cluster aggregation results.

3.2 Evaluating Sets of Candidates At Once

Instead of checking candidates one-by-one, Q-ACER evaluates sets of candidates represented as intervals for each predicate. Using bounds from the kd-tree entries we compute conservative bounds for each aggregate and then for the arithmetic constraint via interval arithmetic [6]; if a set must pass (or fail) we accept (or prune) it wholesale, otherwise we partition ranges recursively.

3.3 Caching kd-trees

Users will often explore multiple settings for predicates/thresholds. Based on the observation that the pre-materialized aggregates and bounds for clusters stored in the kd-tree are not affected by the threshold of an aggregate constraint, we reuse these data structures to significantly improve performance for interactive threshold exploration where the user only varies the threshold of a constraint or the number k of repairs to return.

4 DEMONSTRATION OVERVIEW

In general, for all of the demonstration scenarios described below, the user will:

(1) **Onboard & Select Dataset.** Users browse datasets and inspect previews or schemas.
(2) **Build Query & Constraint.** A guided builder collects predicates (attribute–operator–value), aggregate constraints (e.g., fairness/budget), and threshold k (Figure 2b).
(3) **Execute Repair.** The backend evaluates the query against constraints, computes repairs, and gathers statistics.
(4) **Explore Results.** The UI displays original/repaired queries, predicate diffs, and satisfaction bars (Figure 2b), alongside performance metrics (Figure 2c) for the top-k repairs.
(5) **Vary Thresholds.** Users can adjust thresholds to balance repair aggressiveness with constraint strength; Q-ACER achieves efficiency by reusing kd-trees and materialized aggregates.

In the demonstration, we will present several use cases including:

- **Medicaid Scenario**: This scenario is the use case discussed in Section 2. In addition to the constraint $\omega_{smokers}$, we will also explore fairness constraints based on `ageGroup` or `race`.
- **Fairness - ACS Income Dataset:** This dataset is commonly used in fairness research. Queries will identify successful individuals based on high working hours, education, and class of work. We want to ensure that the statistical parity difference between two groups is low (the aggregate constraint), considering several sensitive attributes including `age` and `gender`.
- **Vendor selection:** The vendor selection scenario from Section 1.
- **TPC-H:** We use a query that computes expected revenue. The constraint ensures robust supply chains, i.e., only a certain fraction of expected revenue comes from specific countries.

5 CONCLUSIONS

We present Q-ACER, a system that repairs a user-provided query to fulfill a given set of constraints on the query result by modifying the selection conditions of the query. In contrast to related work, Q-ACER supports aggregate constraints, a more general class of constraints that is sufficient to model a wider range of requirements including fairness metrics. This requires novel techniques for pruning the search space including sharing materialized aggregation results for subsets of the data indexed in a kd-tree and bounding the result for sets of repair candidates [1]. We demonstrate the system using a several use cases including standard fairness applications (e.g., ensure statistical parity based on gender or race), a medicaid enrollment use case, and several other scenarios.

REFERENCES

[1] Shatha Algarni, Boris Glavic, Seokki Lee, and Adriane Chapman. 2026. Efficient Query Repair for Aggregate Constraints. *PVLDB* 19, 2 (2026), 252–264. https://doi.org/10.14778/3773749.3773762
[2] Adriane Chapman and H. V. Jagadish. 2009. Why not?. In *SIGMOD*. 523–534.
[3] Jinyang Li, Yuval Moskovitch, Julia Stoyanovich, and HV Jagadish. 2023. Query Refinement for Diversity Constraint Satisfaction. *PVLDB* 17, 2 (2023), 106–118.
[4] 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.
[5] Chaitanya Mishra and Nick Koudas. 2009. Interactive query refinement. In *EDBT*. 862–873.
[6] Jorge Stolfi and Luiz Henrique de Figueiredo. 2003. An introduction to affine arithmetic. *Trends in Computational and Applied Mathematics* 4, 3 (2003), 297–312.
[7] Quoc Trung Tran and Chee-Yong Chan. 2010. How to conquer why-not questions. In *SIGMOD*. 15–26.
[8] Bienvenido Vélez, Ron Weiss, Mark A. Sheldon, and David K. Gifford. 1997. Fast and Effective Query Refinement. In *SIGIR*. 6–15.