Solving Why Not Questions for Aggregate Constraints through Query Repair

Shatha Algarni
University of Southampton
Southampton, UK
University of Jeddah
Jeddah, Saudi Arabia
s.s.algarni@soton.ac.uk

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

Seokki Lee
University of Cincinnati
Ohio, U.S.
lee5sk@ucmail.uc.edu

Adriane Chapman
University of Southampton
Southampton, UK
Adriane.Chapman@soton.ac.uk

***Abstract*—Many real world problems require the result of a query to fulfill domain specific constraints, e.g., at least a certain fraction of applicants selected for an interview based on their qualifications should be female. In this work, we envision a framework that allows such requirements to be expressed as constraints on the result of a query that combine aggregation results through arithmetic expressions. When a user's query violates a constraint, then our framework repairs the query to satisfy the constraint. While our approach is related to prior work on query-based explanations for missing answers and enforcement of fairness constraints for query results, it supports more expressive constraints and is applicable to use cases other than fairness. We demonstrate the infeasibility of a brute force solution and discuss directions for future work to improve the performance of generating query repairs.**

***Index Terms*—query-based explanation for missing answers, query repair, query refinement**

1. Introduction

Large volumes of data are frequently examined to obtain information that is needed to make important decisions, such as examining genomic data for cancer prediction or examining customer data to detect trends in consumer behaviour. When data is retrieved from a database for analysis, it is essential to guarantee that query results adhere to legal and ethical regulations, such as fairness, and/or fulfill other constraints. Typically, it will be quite challenging to express such constraints as the conditions of a query. In this work, we model such constraints on query results as arithmetic expressions involving aggregate queries evaluated over the output of a *user query*. When the result of a query fails to adhere to such an *aggregate constraint*, we would like the system to fix the constraint violation by *repairing* the query, similar to [1], [2].

The problem we investigate is related to query-based explanations [3], [4] and repairs [5] for missing answers, why-not [4], [6] as well as query refinement / relaxation approaches [2], [7], [8]. We will use the term *"query repair"* to mean changes to query predicates through a combination of relaxation (weakening a predicate) and refinement (strengthening a predicate) and use the more specific terms *"query refinement / relaxation"* for approaches that do not allow the combination of both types of repairs. We allow for constraints that involve non-monotone arithmetic expressions over aggregation results. Li et al. [8] consider the problem of ensuring fairness for query results. In this work, fairness requirements are expressed as constraints on the cardinality of query results from a specified group. [8] introduces optimizations for such monotone conditions. However, many common fairness metrics (e.g., statistical parity [10]) are non-monotone in nature and cannot be expressed as a conjunction of cardinality constraints. Thus, the optimizations proposed in [8] and other existing work on query relaxation and refinement are not applicable to our problem. The main contributions of this work are:

- Identification of real-world problems that require query results to adhere to aggregate constraints in Section 2.
- Formulation of aggregation constraints and the corresponding query repair problem in Section 4.
- Identifying under which conditions an aggregate constraint is non-monotone and discussing the challenges arising from non-monotone constraints (Section 4.1)
- Implementation of a brute force algorithm for the query repair which is applied to both real-world and standard benchmark datasets in Section 5.
- We sketch potential solutions for an efficient query repair algorithm in Section 6.

2. Use Cases

We describe two example use cases to highlight the need for repairing user queries to fulfill aggregate constraints on the query's result.

2.1. Fairness Constraint

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

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

The Aggregate Constraint. The employer would like to ensure that their decision to interview a candidate is not biased against a specific gender. One way to measure such a bias is to measure the *statistical parity difference* (*SPD*) [10], [11] between demographic groups. Given a set of data points that belong to one of two groups (e.g., male and female) and a binary outcome attribute Y where $Y = 1$ is assumed to be a positive outcome (`OfferInterview`=1 in our case), the SPD is the difference between the probability for individuals from the two groups to receive a positive outcome. For instance, SPD is zero if the outcome is perfectly fair, i.e., the group membership of an individual does not affect the probability of a positive outcome. Such probabilities are typically computed using empirical data, e.g., in our example, the statistical parity difference can be computed as shown below (G is `Gender` and Y is `OfferInterview`).

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

The employer would like to ensure that the SPD 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. Using our framework, the employer can express their fairness requirement as an aggregate constraint SPD ≥ 0.2. Our framework will change the conditions of the prescreening query to ensure a fair outcome. Prior work on ensuring fairness by repairing queries [8] only consider cardinality constraints for a single group in the query result which cannot express statistical parity. In general, fulfilling such aggregate constraints requires changing multiple predicates in the query. The reader may expect that fairness can be ensured by using different conditions for each demographic group. However, this introduces a different type of bias as now individuals with the same credentials are treated differently.

2.2. Company Product Management

Consider a supply and demand scenario over the TPC-H Benchmark schema [12]. A company would like to have a list of the suppliers in Europe for the parts of type *"Large Brushed"* whose size is greater than 10.

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

The Company is concerned about cash flow problems and wants the average price of parts not to exceed the average amount owed to the selected suppliers (`s_acctbal`):

$$\mathbf{avg}(\texttt{p\_retailprice}) - \mathbf{avg}(\texttt{s\_acctbal}) \geq 2.0$$

Prior work on query repair for average [13] only supports constraints on a single aggregation result while the constraint shown above is an arithmetic combination of aggregation results as supported in our framework.

3. Related Work

Li et al. [8] determine all minimal refinements of a conjunctive query by changing constants in selection conditions such that the refined query fulfills a conjunction of cardinality constraints, e.g., the query should return at least 5 answers where `gender` = female. A refinement is minimal if it fulfills the constraints and there does not exist any refinement that is closer to the original query in terms of similarity of constants used in predicates. The cardinality constraints considered in that work involve filter-aggregation queries that we also consider in our work, but do not allow for arithmetic combinations of the results of such queries, e.g., as required by standard fairness conditions. Both our work and [8] face the challenge of an exponential search space (all possible combinations of changes to individual predicates in a query). For monotone constraints (either only relaxation of predicates or restriction of predicates is required to fix the query), the problem can be solved in PTIME by exploiting the ordering induced by monotonicity. We will discuss this in more detail in Section 4.

Query refinement. Another line of work related to our approach is query refinement [2], [3], [14]. Mishra et al. [2] refine a query to return a given number k of results. Koudas et al. [14] refine a query that returns an empty result to produce at least one answer. Another line of work uses refinement to repair a query to return missing results of interest provided by the user [3], [5]. For a query with n predicates, the number of possible refinements is exponential in n: for each predicate one can choose a constant from the domain of the attribute restricted by the predicate. Most work on query refinement has limited the scope to constraints that are monotone in the size of the query answer. Monotonicity is then exploited to prune the search space [2], [15]–[18]. However, real-world use cases are often inherently non-monotone.

How-to queries. Like our work and [8], the purpose of how-to queries [19] is to achieve a desired change to the result of a query. However, in how-to queries this is achieved by changing the input database rather than changing the query. Wang et al. [20] study the problem of deleting operations from an update history to fulfill a constraint over the current database expressed as tuple substitutions (replace t_i in the result with t_i'). However, this approach does not consider query repair (changing predicates) nor aggregate constraints.

Explanations for Missing Answers. Query-based explanations for missing answers [4], [9], [21] are operators or sets of operators that are responsible for the failure of a query to return a result of interest. However, this line of work does not generate query repairs.

4. Problem Definition

Consider a database $D = \{R_1, \cdots, R_n\}$ with one or more relations R_i. Let us assume for convenience that attribute names are unique within D. We use $A = \{a_1, \ldots, a_m\}$ to denote the set of all attributes in D.

User Query. A user query Q is an SPJ (Select-Project-Join) query. Such queries can be expressed as relational algebra expressions of the form $\pi_A(\sigma_\theta(R_1 \bowtie \ldots \bowtie R_n))$.

We consider selection predicates θ that are comparisons of the form $a_i \,\mathrm{op}\, c_i$ where $\mathrm{op} \in \{<, >, \leq, \geq\}$ for the numerical attributes a_i and constants c_i and $\mathrm{op} \in \{=, \neq\}$ for the categorical attributes a_i and constants c_i. We use $Q(D)$ to denote the result of Q over D.

Aggregate Constraints. The user specifies the desired requirements for a query result as **aggregate constraints**. That is, thresholds on the result of 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 aggregation function — one of $\mathbf{count}, \mathbf{sum}, \mathbf{min}, \mathbf{max}, \mathbf{avg}$ — and θ is a selection condition. We use Q_α to denote such a filter-aggregation query. These queries are evaluated over the user query's result $Q(D)$. An aggregate constraint α is of the form:

$$\alpha := \tau \;\mathrm{op}\; \Phi(Q_{\alpha_1}, \ldots, Q_{\alpha_n}).$$

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

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

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

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

where the distance between two predicates $\theta_i = a_i \,\mathrm{op}\, c_i$ and $\theta_i' = a_i \,\mathrm{op}\, c_i'$ on a numeric attribute a_i is:

$$\frac{|c_i' - c_i|}{|c_i|}$$

For categorical attributes, the distance is 1 if $c_i \neq c_i'$ and 0 otherwise.

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

We are now ready to formulate the problem studied in this work:

- **Input**: user query Q, database D, and aggregate constraint α
- **Output**: $\mathrm{argmin}_{Q_{fix}\ \textbf{is a repair}}\, d(Q, Q_{fix})$

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

4.1. Non-monotonicity of Aggregate Constraints

An aggregation function f is monotonically increasing (decreasing), if $f(S_1) \leq f(S_2)$ when $S_1 \subseteq S_2$ ($S_1 \supseteq S_2$) for any two bags of values S_1 and S_2. As mentioned in Section 3, query refinement and relaxation techniques exploit the monotonicity of aggregation functions to optimize search as relaxing (refining) a query Q's selection conditions is bound to increase (decrease) $f(Q(D))$ if f is monotonically increasing, e.g., by pruning unpromising candidates to find the refined query faster without probing all candidates in the search space [8], [15], [18].

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

1) Using a non-monotone arithmetic operator like division or subtraction.
2) Using a non-monotone aggregation function ($\mathbf{sum}$ over the integers $\mathbb{Z}$).
3) At least one monotonically increasing and one monotonically decreasing aggregation function is used (e.g., $\mathbf{min} + \mathbf{count}$ or $\mathbf{max} + \mathbf{min}$)

5. Naive Algorithm and Preliminary Results

To gain a better understanding of the computational challenges of finding a repair, we evaluate a brute force algorithm that enumerates and tests all possible refinements. We measure runtime using multiple queries and datasets that yield various number of possible refinements. The algorithm was implemented in Python and the experiment was performed on a macOS Sonoma 14.2.1 machine with a 2.3 GHz Quad-Core Intel i7 and 32 GB memory.

Datasets. We have chosen two datasets, **Adult Census Income (Adult-CSI)** [22] and **Healthcare** [23], to evaluate the fairness use case. Adult-CSI has 1M rows and 14 attributes while for Healthcare we varied the dataset size between 100 and ∼800 rows (subsets of the original dataset). We utilize **TPC-H** [12] as a common benchmark which contains 61 attributes in total. We varied dataset size from a scale factor of 0.001 to 1 (roughly 1GB).

Queries. For Adult-CSI and Healthcare, we use queries from [8]: Q7 to Q10 (Adult-CSI) and Q3 to Q6 (Healthcare). These queries have between 3 and 4 selection predicates. For TPC-H, we generated two SPJ queries Q1 and Q2 with 3 predicates inspired by TPC-H's Q2.

Number of Possible Refinement Candidates. The number of possible refinements is calculated as the product of

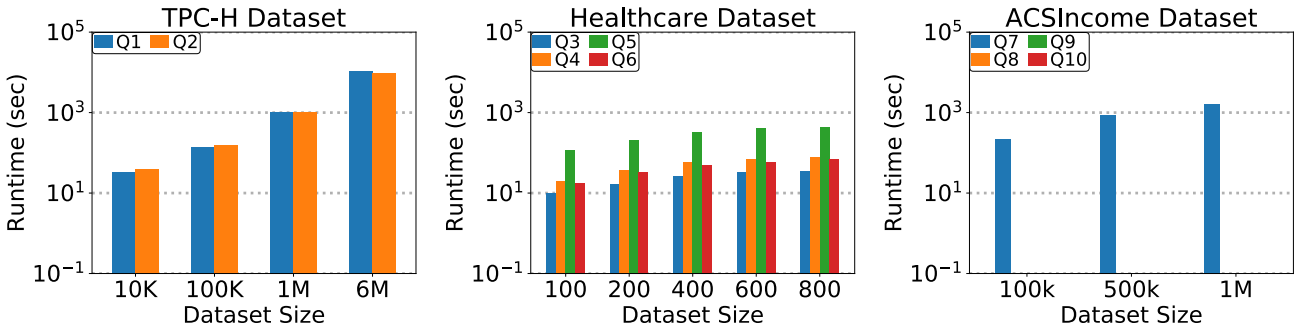

Figure 1: Comparison of runtime varying dataset, dataset size and query.

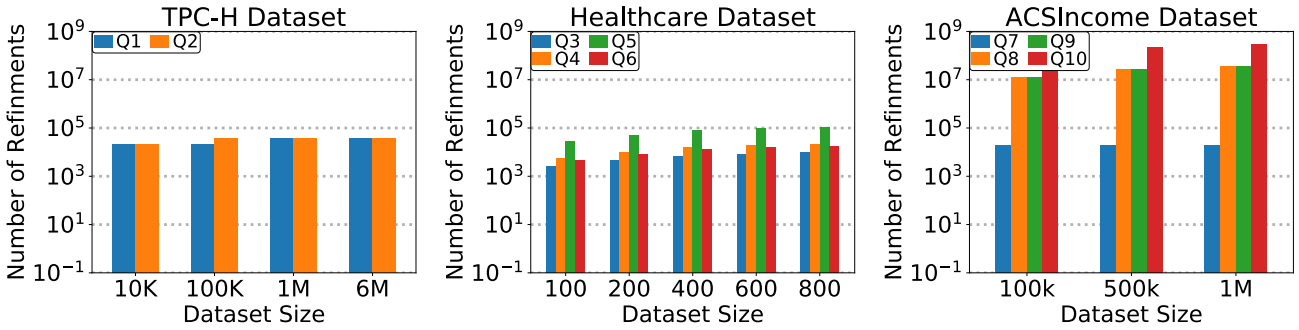

Figure 2: Comparison of number of possible refinements varying dataset, dataset size and query.

all predicates' possible values. For Adult-CSI, the number varies between ∼19K to ∼293M while, for Healthcare, it is between ∼2K and ∼105K. For the TPCH queries, we have up to ∼37K possible refinement candidates.

Aggregate Constraints. For Adult-CSI and Healthcare, we enforce that the statistical parity difference (SPD) between two demographic groups is within a certain range. For Adult-CSI, we determine groups based on gender and require that the SPD is within [-0.1,0.1]. For Healthcare, we determine demographic groups based on race and requiring a SPD in the range [-0.3,0.2]. For TPC-H, we use the aggregate constraint from Section 2.2 which is a difference between two aggregation functions.

Results. Figure 1 shows the runtime varying datasets, dataset sizes, and queries. Q8 to Q10 over Adult-CSI did not finish within the allocated time as the search space is too large. Q7 over the same size of the dataset completed due to having the smallest number of possible refinements (shown in Figure 2) among all considered queries for this dataset. As expected, larger datasets in general have longer runtime. Although Q8 to Q10 use a smaller dataset than Q1 and Q2, these queries did not finish due to a large number of possible refinements as shown in Figure 2. As expected, this confirms that the number of possible refinements significantly impacts runtime, supporting further investigation into query repair for aggregate constraints.

6. Challenges and Open Problems

The use of the arithmetic expression in this work invalidates past solutions that rely upon the monotonicity property for query refinement. We propose three possible optimizations that we believe to have potential for improving performance.

Calculating bounds. In order to improve the performance of evaluating refinement candidates, we propose to calculate lower and upper bounds on aggregation function results for subsets of the data and propagate these bounds to calculate bounds on the result of arithmetic expressions used in an aggregate constraints. The rationale for this approach is that it is sometimes possible to combine bounds for aggregation function results (or their arithmetic combination) over different subgroups without reevaluating the aggregation over the base tables. Such bounds can then be used to prune candidates. For example, for a constraint $\frac{\mathbf{sum}(a)}{\mathbf{count}(a)} < 10$, if a repair candidate has a bound $[0, 4]$, then we know it is a repair as it is bound to fulfill the aggregate constraint. As another example, if a candidate has bounds $[15, 20]$, then we can prune it as it cannot possibly fulfill the aggregate constraint.

Estimating bounds. If we are ok with excepting approximate results, then we could utilize approximate query processing (AQP) techniques to estimate bounds over samples or use statistics [24]. If we approach the problem using AQP, then we have to investigate how errors propagate for our setting.

Incremental aggregation. Note that evaluating the aggregations used in aggregate constraints for different candidate repairs is essentially evaluation of an aggregate query over different subsets of the data determined based on the selection conditions of the candidate query. Similar to techniques for the cube operator and other types of aggregation with multi-dimensional grouping criteria [25], [26], we may be able to reuse intermediate results and benefit from incrementally maintaining aggregation results when computing results for an aggregation over related subgroups. As a trivial example, consider computing the result of $\mathbf{sum}(a)$ for all settings of a single predicate $b < c$. Similar to aggregation by sorting, we can sort the data on b and then incrementally compute $\mathbf{sum}(a)$ and output the current result for each distinct value c' of b (corresponding to the refinement $b < c'$ of the predicate).

References

[1] D. V. Kalashnikov, L. V. Lakshmanan, and D. Srivastava, "Fastqre: Fast query reverse engineering," in *SIGMOD*, pp. 337–350, 2018.

[2] C. Mishra and N. Koudas, "Interactive query refinement," in *EDBT*, pp. 862–873, 2009.

[3] Q. T. Tran and C.-Y. Chan, "How to conquer why-not questions," in *SIGMOD*, pp. 15–26, 2010.

[4] A. Chapman and H. V. Jagadish, "Why not?," in *SIGMOD*, pp. 523–534, 2009.

[5] N. Bidoit, M. Herschel, and K. Tzompanaki, "Refining sql queries based on why-not polynomials," in *TaPP*, 2016.

[6] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom, "Uldbs: databases with uncertainty and lineage," in *Very Large Data Bases Conference*, 2006.

[7] B. Vélez, R. Weiss, M. A. Sheldon, and D. K. Gifford, "Fast and effective query refinement," in *SIGIR*, pp. 6–15, 1997.

[8] J. Li, Y. Moskovitch, J. Stoyanovich, and H. Jagadish, "Query refinement for diversity constraint satisfaction," *Proceedings of the VLDB Endowment*, vol. 17, no. 2, pp. 106–118, 2023.

[9] R. Diestelkämper, S. Lee, M. Herschel, and B. Glavic, "To not miss the forest for the trees - a holistic approach for explaining missing answers over nested data," in *SIGMOD*, pp. 405–417, 2021.

[10] R. K. E. Bellamy, K. Dey, M. Hind, S. C. Hoffman, S. Houde, K. Kannan, P. Lohia, J. Martino, S. Mehta, A. Mojsilovic, S. Nagar, K. N. Ramamurthy, J. T. Richards, D. Saha, P. Sattigeri, M. Singh, K. R. Varshney, and Y. Zhang, "Ai fairness 360: An extensible toolkit for detecting and mitigating algorithmic bias," *IBM J. Res. Dev.*, vol. 63, no. 4/5, pp. 4:1–4:15, 2019.

[11] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan, "A survey on bias and fairness in machine learning," *ACM Comput. Surv.*, vol. 54, no. 6, pp. 115:1–115:35, 2022.

[12] "Tpc benchmark h (decision support) standard specification revision 3.0.1." https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp, 2024. Accessed on 2024-03-20.

[13] A. M. Albarrak and M. A. Sharaf, "Efficient schemes for similarity-aware refinement of aggregation queries," *World Wide Web*, vol. 20, no. 6, pp. 1237–1267, 2017.

[14] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica, "Relaxing join and selection queries," in *VLDB*, pp. 199–210, 2006.

[15] N. Bruno, S. Chaudhuri, and D. Thomas, "Generating queries with cardinality constraints for dbms testing," *TKDE*, vol. 18, no. 12, pp. 1721–1725, 2006.

[16] C. Mishra, N. Koudas, and C. Zuzarte, "Generating targeted queries for database testing," in *SIGMOD*, pp. 499–510, 2008.

[17] A. Kadlag, A. V. Wanjari, J. Freire, and J. R. Haritsa, "Supporting exploratory queries in databases," in *DASFAA*, vol. 2973, pp. 594–605, 2004.

[18] E. Wu and S. Madden, "Scorpion: Explaining away outliers in aggregate queries," *PVLDB*, vol. 6, no. 8, pp. 553–564, 2013.

[19] A. Meliou and D. Suciu, "Tiresias: the database oracle for how-to queries," in *SIGMOD*, pp. 337–348, 2012.

[20] X. Wang, A. Meliou, and E. Wu, "Qfix: Diagnosing errors through query histories," in *SIGMOD*, pp. 1369–1384, 2017.

[21] D. Deutch, N. Frost, A. Gilad, and T. Haimovich, "Explaining missing query results in natural language," in *EDBT*, pp. 427–430, 2020.

[22] S. A. Friedler, C. Scheidegger, S. Venkatasubramanian, S. Choudhary, E. P. Hamilton, and D. Roth, "A comparative study of fairness-enhancing interventions in machine learning," in *FAT*, pp. 329–338, 2019.

[23] S. Grafberger, S. Guha, J. Stoyanovich, and S. Schelter, "Mlinspect: A data distribution debugger for machine learning pipelines," in *SIGMOD*, pp. 2736–2739, 2021.

[24] X. Liang, S. Sintos, Z. Shang, and S. Krishnan, "Combining aggregation and sampling (nearly) optimally for approximate query processing," in *SIGMOD* (G. Li, Z. Li, S. Idreos, and D. Srivastava, eds.), pp. 1129–1141, 2021.

[25] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi, "On the computation of multidimensional aggregates," in *VLDB*, pp. 506–521, 1996.

[26] J. Han, "Review - an array-based algorithm for simultaneous multidimensional aggregates," *ACM SIGMOD Digit. Rev.*, vol. 1, 1999.