University of Southampton

Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

Efficient Query Repair for Aggregate Constraints

by

Shatha Saad Algarni

ORCiD: 0009-0004-9831-6034

A thesis for the degree of
Doctor of Philosophy

December 2025

University of Southampton

Abstract

Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

Doctor of Philosophy

Efficient Query Repair for Aggregate Constraints

by Shatha Saad Algarni

In many real-world scenarios, query results must satisfy domain-specific constraints such as fairness or financial stability. For example, selecting interview candidates based on their qualifications may require that at least a given percentage be female, or a report on purchase costs may need to ensure the average cost stays below a liability threshold. These requirements can be expressed as constraints over an arithmetic combination of aggregates evaluated on the result of the query.

This thesis studies how to repair a query to fulfill such constraints by modifying the filter predicates of the query. These constraints are non monotone and more complex than those considered in prior work, such as query based explanations for missing answers or fairness enforcement in query results. The constraints considered in this thesis invalidate many existing optimizations considered in prior work. The work in this thesis introduces a novel query repair technique that computes the top-k candidate repairs with respect to their distance to the user query. These techniques leverage materialization and data clustering to avoid unnecessary computation. It also exploits bounds on sets of candidate solutions and interval arithmetic to efficiently prune the search space. Experimental evaluation on real-world and benchmark datasets shows that the proposed pruning technique significantly outperforms baselines that consider a single candidate at a time.

Contents

List of Figures

List of Tables

Declaration of Authorship

Acknowledgements

First and foremost, I offer my deepest gratitude and praise to Allah. It is through His guidance, mercy, and countless blessings that I have found the strength and determination to complete this journey. For this, I am eternally thankful.

I would like to express my sincere appreciation to my supervisors. I am grateful to Professor Steffen Staab for his guidance and encouragement throughout this work. I am especially thankful to Professor Age Chapman for her steadfast mentorship and close supervision during every stage of this journey. Her continuous support, insightful feedback, and unwavering belief in my potential have been central to the development of this research and my growth as a scholar.

To my dear parents, your unwavering love, support, and prayers have been a constant source of strength. The values you taught me, your endless encouragement, and your faith in my journey have carried me through every challenge. I also extend my love and thanks to my siblings, whose support and have made this achievement even more meaningful.

To my beloved husband, Badr Algarni, thank you for your incredible patience, understanding, and steadfast support. Your belief in me and your sacrifices for our family have been an anchor throughout this journey. I am deeply grateful for your love. To my precious children, Aleen and Saleh, you are my greatest inspiration. Your laughter, love, and light give meaning to everything I do. You have been my motivation through the most demanding moments and the joy behind every success.

I am also deeply thankful to my friends, especially those who supported me during my time abroad. Your kindness and companionship have meant more than words can express.

I am sincerely grateful to the Saudi Arabian government and Jeddah University for their generous support, which made this research possible and reflects their commitment to the development of future scholars.

This acknowledgment is more than words, it is a reflection of the gratitude I hold in my heart for everyone who has been part of this journey. With the highest praise to Allah, I thank each and every one of you.

To my parents, husband, and children

Definitions and Abbreviations

D	The input database, consisting of one or more relations
R_i	A relation in the database
Q	The user's original SPJ query
A	The set of projected attributes in the query
θ	The selection predicate of the query
θ_i	An individual predicate in the query
a_i	An attribute used in a predicate
op	A comparison operator, e.g., $<, \leq, >, \geq, =, \neq$
c_i	A constant value in a selection predicate
$Q(D)$	The result of evaluating query Q on database D
f	An aggregate function, e.g., `count`, `sum`, `avg`, `min`, `max`
Q^ω	A filter-aggregation query
ω	An aggregate constraint
τ	A threshold used in an aggregate constraint
$\mathbf{\Phi}$	An arithmetic expression over aggregation results
Q_{fix}	A repair candidate query that modifies constants in θ
$Q_{fix}(D)$	The result of evaluating the repaired query Q_{fix} on D
$\models$	Logical satisfaction (i.e., $Q_{fix}(D) \models \omega$ means the result satisfies the constraint)
$d(Q, Q_{fix})$	The distance between the original and repaired query
$\mathbb{Q}_{fix}$	The space of all possible query repairs
$\text{topk}_{Q_{fix}}$	The set of top-k query repairs with smallest distance
C	A node (cluster) in the kd-tree representing a subset of the database
C_{root}	The root node of the kd-tree (covers the full dataset)
$\mathbf{C}$	A set of clusters whose union exactly matches tuples satisfying a candidate query
$\mathcal{B}$	The number of child nodes per internal node in the kd-tree
$\mathcal{S}$	The minimum number of tuples per leaf node (stopping condition for tree splitting)
θ'	Selection condition of a repair candidate query
$\textsc{bounds}_{a_i}$	The $[\min, \max]$ bounds of attribute a_i in a cluster
$\overline{a}, \underline{a}$	Upper and lower bounds of attribute a in a cluster
c'_i	A constant used in predicate θ'_i of a repair candidate
f'	Combination function used to merge aggregates across clusters (e.g., `sum` or `max`)

Chapter 1

Introduction

Data is a commodity that allows organizations to build models, perform analyses, and make strategic decisions. Across different fields such as healthcare, retail, finance, and education, large volumes of structured data are stored and accessed via a variety of data management systems. These include traditional relational databases (Minker and Sable, 1970; Yang et al., 2009), online analytical processing (OLAP) systems (Alkharouf et al., 2005; Joglekar et al., 2017; Patel and Sharma, 2020), cloud data warehouses (Rehman et al., 2018; Li et al., 2024), and more recently, data marketplaces where structured datasets are traded between entities (Abbas et al., 2021; Azcoitia and Laoutaris, 2022).

Analysts, data engineers, and other practitioners interact with these systems on a daily basis. Their workflows often involve querying data to extract insightful information that informs dashboards, trains machine learning algorithms, or supports reporting. Yet querying data is not just a technical task but it is also embedded within broader organizational objectives, that includes maintaining fairness, achieving representativeness, meeting compliance standards, and optimizing operational performance. However, these professionals often face a disconnect between what a query technically specifies and what an organization needs from the dataset as a whole. For example, a machine learning engineer may need a dataset of training examples that is both high quality and demographically balanced, or a retail analyst may require sales data that represents different geographic regions proportionally. These real-world needs translate into constraints over query outputs that are not easily expressible using standard query predicates nor simple constraints. Consequently, ensuring that query results align with such constraints remains a significant challenge in modern data analysis.

In practice, analysts are typically well versed in writing queries that return data based on obvious conditions, e.g., only return applicants with a master's degree. However, a query result *in toto* often has to fulfil 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 fulfil such constraints this is not always viable, e.g., because the same selection criterion has to be used for all applicants for a job. Thus, the query has to be repaired such that the fixed query satisfies all constraints for the entire result set.

Prior work in this area, including query-based explanations (Tran and Chan, 2010; Chapman and Jagadish, 2009) and repairs (Bidoit et al., 2016) for missing answers, work on answering why-not questions (Chapman and Jagadish, 2009; Benjelloun et al., 2006) as well as query refinement / relaxation approaches (Vélez et al., 1997; Mishra and Koudas, 2009; Li et al., 2023) determine why specific tuples are not in the query's result or how to fix the query to return such tuples. This thesis studies a more general problem where the entire result set of the query has to fulfil some constraint. The constraints studied in this work are expressive enough to guarantee query results adhere to legal and ethical regulations, such as fairness. Typically, it is challenging to express such constraints through the selection conditions of a query.

This chapter provides an overview of the thesis. Section 1.1 presents the motivations underlying this work, while Section 1.2 summarizes the overall structure of the thesis. Section 1.3 discusses the key challenges involved in repairing queries to satisfy constraints defined by arithmetic combinations of aggregate functions. The scope and boundaries of the research are detailed in Section 1.4.Section 1.5 outlines the core research questions and summarizes the main contributions of the study.

1.1 Motivation

To illustrate the practical importance of the query repair problem under such constraints, this section presents two motivating use cases: one focusing on fairness in recruitment and the other on supply chain management. These examples demonstrate real-world scenarios where non-monotone aggregate constraints arise naturally and cannot be addressed by existing techniques.

1.1.1 Fairness Constraint

Fairness in algorithmic decision-making seeks to prevent systematic bias against demographic groups defined by protected attributes such as race, gender, or socioeconomic status. This example uses the group-fairness notion of *statistical parity difference* (SPD). SPD is a commonly used group fairness metric that measures the difference in the probability of receiving a positive outcome between a protected group and an unprotected group. Formally,

$$\text{SPD} \;=\; \Pr\Big(\hat{Y}=1 \mid A=0\Big) \;-\; \Pr\Big(\hat{Y}=1 \mid A=1\Big).$$

Here $\hat{Y}$ denotes the algorithm's binary outcome and $A \in \{0,1\}$ the protected attribute (e.g., gender, race). An SPD of 0 indicates perfect statistical parity (i.e., both groups receive positive outcomes at the same rate). Values that deviate from 0 indicate the extent of disparity: negative values imply the protected group is disadvantaged, while positive values imply the opposite (Kleinberg et al., 2018; Verma and Rubin, 2018; Mehrabi et al., 2021). There is no universally agreed-upon threshold for what constitutes an acceptable SPD. Instead, organizations and policymakers determine thresholds based on context, ethical considerations, and legal frameworks. For instance, the U.S. Equal Employment Opportunity Commission (EEOC) applies the four-fifths rule, where a disparity greater than 0.2 (20 percentage points) may indicate potential discrimination[1]. A stricter threshold (e.g., 0.01) might be appropriate in high-stakes domains such as healthcare or criminal justice, where even small disparities can have serious consequences, while more lenient thresholds may be chosen in exploratory or low-risk contexts. In practice, policymakers and organizations often calibrate such thresholds through empirical analysis of domain-specific data or fairness audits to ensure that the chosen level of disparity aligns with acceptable risk or compliance standards.

Importantly, SPD is a difference of probabilities, so it cannot be directly interpreted as a probability itself. Instead, it should be viewed as a measure of deviation from parity. The SPD formulation originates from fairness literature on algorithmic bias detection (Feldman et al., 2015; Calders et al., 2009; Mehrabi et al., 2021) and is widely adopted due to its simplicity and intuitive interpretation.

Example 1.1 (Fairness Motivating Example)**.** *Consider a job applicant dataset D for a tech-company that contains six attributes: `ID`, `Gender`, `Major`, `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 ≥ 32 AND GPA ≥ 3.80
```

Aggregate Constraint*. The employer wants to ensure that their decision to interview a candidate is not biased against a specific gender. One way to measure such a bias is to measure the statistical parity difference (SPD) (Bellamy et al., 2019; Mehrabi et al., 2022) between demographic groups. Given a set of data points that belong to one*

[1] https://www.uniformguidelines.com/uniformguidelines.html#18

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 this case), the SPD is the difference between the probability for individuals from the two groups to receive a positive outcome. In this example, the SPD can be computed as shown below (G is `Gender` and Y is `OfferInterview`). $\mathbf{count}(\theta)$ *is used to denote the number of query result tuples satisfying a given condition θ. For example,* $\mathbf{count}(G = M \wedge Y = 1)$ *counts the number of tuples where the gender is male and the label is positive.*

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

The employer would like to ensure that the SPD between male and female is below 0.2. The model generating the `OfferInterview` attribute is trusted by the company, but is provided by an external service and, thus, cannot be fine-tuned to improve fairness. However, the employer is willing to change their prescreening criteria by expressing their fairness requirement as an aggregate constraint $SPD \leq 0.2$ as long as the same criteria are applied to judge every applicant to ensure individual fairness. That is, the employer desires a repair of the query whose selection conditions are used to filter applicants. Prior work on ensuring fairness by repairing queries (Li et al., 2023) only considers cardinality constraints which cannot express SPD.

The approach proposed in this thesis (as well as other query repair approaches such as the work by Li et al. (2023)) is suited to scenarios where, due to fairness considerations or regulatory requirements, the same selection criteria must be uniformly applied to all candidate items or individuals. These criteria are typically expressed using SPJ (Select-Project-Join) queries, a standard formalism in relational databases. However, when additional constraints such as group fairness must be met, users the challenge of identifying query selection conditions that simultaneously fulfill legal or ethical requirements and remain aligned to their original selection intent. For instance, in Example 1.1, an employer aims to select job candidates based on *testScore* and *GPA* criteria that can be easily formulated as a query. Due to regulations, the same criteria must apply to all applicants. If the employer wishes to ensure fairness with respect to *Gender*, they must adjust the selection conditions to achieve fairness while remaining close to their original goal of choosing candidates with high *GPA* and *testScore*. The algorithms developed in this thesis enable such repairs, automatically producing revised selection conditions that satisfy constraints while preserving the user's intent.

1.1.2 Company Product Management

Example 1.2 (Company Product Management)**.** *A retail company aims to support inventory planning by retrieving data on parts of type "Large Brushed" with a size* *greater than 10 that are supplied by suppliers located in Europe. The company uses the following query to retrieve this information:*

```
Q2: SELECT *
    FROM part, supplier, partsupp, nation, region
    WHERE p_partkey = ps_partkey AND
        s_suppkey = ps_suppkey AND p_size >= 10
        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 (Albarrak and Sharaf, 2017) only supports constraints on a single aggregation result while the constraint shown above is an arithmetic combination of aggregation results as supported in this framework.

The example in Example 1.2 demonstrates that the approach proposed in this thesis is not limited to traditional fairness applications. It highlights the relevance of this work in supply chain and product management contexts. In this scenario, a retail company formulates a query to retrieve details about parts and suppliers in order to support inventory planning. The selection criteria such as part size, type, and supplier region are uniformly applied through a structured query that reflects operational policy. However, the company must also satisfy a business constraint aimed at mitigating supply chain risk: ensuring that products sourced from the UK account for no more than 10% of the total expected revenue. This constraint is expressed as a ratio between two aggregate values and cannot be handled by existing techniques that support only simple aggregate constraints.

1.2 Overview of This Thesis

In this work, constraints are modelled 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), the system will fix the violation by *repairing* the query by adjusting the selection conditions of the query, similar to (Kalashnikov et al., 2018; Mishra and Koudas, 2009).

For clarity, throughout this thesis the following terms are used. The term best repair refers to a modified query that most closely preserves the intent of the user's original query according to a defined distance function between the two queries. The phrases **similarity to** and **closest to** both refer to this same notion of distance which is the quantitative measure of how small a modification is required to make the repaired query satisfy the given constraint while preserving the user's intent. Formally, the distance may be defined over the values of constants used in the query predicates, and the top-k repairs are those that minimize this distance metric.

The rational for repairing the query, instead of seeking other modifications such as selecting a subset of the query result, lies in applications where a subset of results should be selected and the criteria for selection have to be applied uniformly to all candidate answers. Such uniformity is often required in domains where fairness or regulatory compliance is critical, the same selection criteria must be applied to all job candidates, or government agencies must select contractors based on a predetermined set of requirements to prevent corruption. This thesis focuses on computing the top-k repairs with respect to their distance to the user query. The rationale is that to preserve the original semantics of the user's query as much as possible.

Among prior work, the most closely related is Erica (Li et al., 2023), which proposes an efficient refinement strategy for queries that violate group cardinality constraints. These constraints require to fulfill a conjunction of cardinality constraints, e.g., the query should return at least 5 answers where gender = female. Erica exploits the monotonicity of such constraints to prune large portions of the repair space efficiently. However, this thesis departs from Erica in a key way: the aggregate constraints targeted here are not limited to monotonic patterns. Instead, they can be non-monotonic and are often expressed as arithmetic combinations of multiple aggregate queries such as the Statistical Parity Difference (SPD) in Example 1.1, which compares rates across groups. These expressions do not exhibit monotonic behavior with respect to predicate refinement which invalidates most of the optimizations considered in prior work.

Consider a user query with a conjunctive selection condition $\bigwedge_{i=1}^{m} a_i \,\mathrm{op}\, c_i$ where a_i is an attribute, c_i is a constant, and op is a comparison operator, e.g., $<$. A brute force approach for finding a solution to the query repair problem is to enumerate all possible candidate repairs. A candidate repair can be encoded as a combination of constants c'_1 to c'_m representing an updated condition $\bigwedge_{i=1}^{m} a_i \,\mathrm{op}\, c'_i$. The candidate repairs are then sorted based on their distance to the user query and each candidate is evaluated by running the modified query and checking whether the candidate fulfils the aggregate constraint. The algorithm tests candidates until k repairs have been found that fulfil the aggregate constraint. This approach is referred as *Brute Force (BF)*. The main problem with this approach is that the number of repair candidates is exponential in the number of predicates in the user query and for each repair candidate the approach has to evaluate the modified user query and one or more aggregate queries on top of the user query result. Given that the repair problem is NP-hard this work cannot hope to avoid this cost in general.

Nonetheless, two opportunities are identified for optimizing this process: (i) when two repair candidates are similar (in terms of the constants they use in selection conditions), then typically there will be overlap between the aggregate constraint computations for the two candidates and (ii) bounds on the aggregate constraint result can be computed for a set of similar candidate repairs at once using interval arithmetic (De Figueiredo and Stolfi, 2004), a technique for over-approximating the outputs of a computation over a set of values efficiently. If none or all values within the computed bounds fulfill the aggregate constraint, then the approach has shown for a set of candidate repairs that none of them is a repair or all of them are repairs.

To exploit (i), a kd-tree (Bentley, 1975a) is used to partition the input dataset. For each cluster (node in the kd-tree) the approach materializes the result of evaluating the aggregation functions needed for a constraint on the set of tuples contained in the cluster as well as store bounds for the values of attributes within the cluster (as is done in zonemaps / small materialized aggregates (Moerkotte, 1998; Ziauddin et al., 2017)). Then to calculate the result of an aggregation function for a repair candidate, the bounds are used 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 the same test is applied to the children of the cluster in the kd-tree). This approach is referred as *Full Cluster Filtering (FF)*. The main advantage of this algorithm over the brute force approach is that it can reuse the aggregate query results materialized for a cluster if all tuples in the cluster fulfil the condition of the repair candidate and can skip any clusters that do not contain any tuples fulfilling the conditions.

This idea is extended to bound the aggregation constraint result for sets of repair candidates at once to exploit observation (ii). A set of candidates are represented through intervals of values for each c_i for which the user query contains a predicate $a_i \,\mathrm{op}\, c_i$, e.g., $c_i \in [33, 37]$. the approach again reasons about whether all / none of the tuples in a cluster will fulfill the condition for *every* repair candidate whose c_i values are within the bounds (e.g., $[33, 37]$ in the example). The result will be valid bounds on the aggregation constraint result for any candidate repair with constants within the bounds for the candidate set. Then such bounds are exploited to determine all repair candidates within such bounds are repairs or none of them are. If the result is

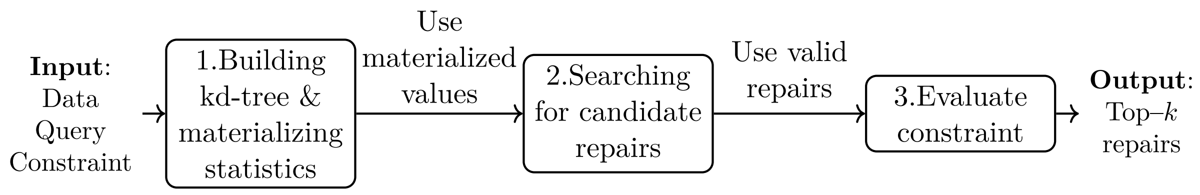


FIGURE 1.1: Black-box view of the three-stage repair pipeline.

inconclusive, the approach can partition such a set of repair candidates into multiple smaller sets and apply the same approach to these sets. This approach is referred as *Cluster Range Pruning (RP)*. The advantage of this approach is that it often enables to prune sets of repair candidates or confirm all of them to be repairs without individually evaluating them.

In summary, FF reuses materialized aggregation results for a set of tuples to avoid unnecessary computations, and RP leverages interval arithmetic to evaluate sets of repair candidates at once, further reducing computational cost. Both algorithms follow the same three main stages pipeline (see Figure 1.1):

1. Partition the input data into clusters and pre-aggregate the needed aggregation functions by the aggregate constraint and the bounds of the attribute values used in the user's predicates.
2. Evaluate the candidate repairs in order of their distance to the original query, one-by-one in FF, or as intervals in RP, and test them against these clusters.
3. Evaluate the constraint on the valid candidate repairs and return the top-k.

1.3 Challenges

Repairing a query under an arithmetic combination of aggregate constraints presents three major challenges:

- **High evaluation cost.** Each repair candidate requires re-evaluating the modified query over the full dataset and then executing one or more aggregate computations on its result. This cost quickly becomes prohibitive.
- **Exponential repair space.** The number of possible repairs grows exponentially with the number of predicates in the original query. Exhaustively checking every candidate is therefore infeasible for realistic workloads (Li et al., 2023).
- **Non-monotonicity of constraints:** Many aggregate constraints (e.g., fairness measures like SPD (Bellamy et al., 2019; Mehrabi et al., 2022)) are non-monotone. This renders traditional pruning strategies, which rely on monotonicity, ineffective (Albarrak and Sharaf, 2017).

1.4 Research Scope

This research focuses on the problem of repairing selection queries such that the query output satisfies the aggregate constraints, which may include arithmetic combinations of aggregate functions.

The specific scope of this work includes:

- **Query Structure:** This work considers SPJ (select-project-join) queries with conjunctive selection predicates.
- **Constraints:** Constraints are expressed as a combination of arithmetic expression over the results of one or more aggregation queries. These constraints may involve aggregate functions such as `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`, and can include arithmetic operators $(+, -, \times, \div)$.
- **Query Repair:** A candidate repair is a modified query whose selection predicates use different constants.
- **Repair Distance:** In this thesis, the repair distance quantifies how much a repaired query deviates from the user's original query. The intuition is that the best repairs are those that preserve the original intent of the user's query as closely as possible. Formally, the distance between two queries is computed based on the differences in their selection conditions. For numeric attributes, it reflects the relative change in predicate constants (e.g., how much a threshold value has shifted), while for categorical attributes, it measures whether the predicate value has changed. This distance metric is then used to rank all possible repairs and to select the top-k repairs that are closest to the original query. A complete mathematical definition and example are provided later in Section 4.4.

1.5 Research Questions and Contributions

This work studies the problem of repairing queries to satisfy aggregate constraints that consist of arithmetic combinations of aggregate functions. Existing query repair techniques focus primarily on monotone constraints and cannot handle these more complex, non-monotone aggregate constraints. After reviewing existing approaches that address query repair in the literature, this work pose the following research questions:

- **Main Research Question:** How can query repair be effectively performed under non-monotone aggregate constraints?

- **RQ1:** How can the shape of an instance of a dataset be leveraged to develop a more efficient approach for query repair under non-monotone aggregate constraints?
- **RQ2:** What pruning strategies can be designed to reduce the search space of candidate repairs in query repair under non-monotone aggregate constraints?

To address these questions, this thesis makes the following contributions:

- **Formalization of the query repair problem for non-monotonic aggregate constraints.** This work provides a formal definition of the query repair problem under constraints expressed as arithmetic combinations of aggregate functions, unlike prior research that focused only on monotonic aggregate functions (Chapter 4). This formulation generalizes the scope of query repair and establishes the theoretical foundation for subsequent algorithmic developments.
- **Design of two novel algorithms for efficient query repair.** Two new algorithms are proposed for solving the aggregate constraint repair problem:
  - **FF (Fully Filtering):** A tree-based approach that exploits precomputed materialized aggregates and attribute bounds to efficiently evaluate repair candidates while avoiding redundant computation (Chapter 5).
  - **RP (Range-based Pruning):** A pruning algorithm that applies interval arithmetic to bound aggregate constraint expressions over groups of repair candidates (Chapter 6).
- **Comprehensive experimental validation.** A detailed experimental evaluation across multiple datasets, queries, and constraint types demonstrates that the proposed methods handles a border class of constraints (Chapter 7). The results highlight the effectiveness of RP in handling complex, non-monotonic constraints efficiently.

Publications

The research conducted in this dissertation has resulted in the following peer-reviewed publications:

- **Algarni, S.**, Glavic, B., Lee, S. & Chapman, A. (2024) ‘Solving why-not questions for aggregate constraints through query repair’, *Proceedings of the IEEE European Symposium on Security and Privacy Workshops (EuroS&PW 2024)*, pp. 592–596. **Published**.
- **Algarni, S.**, Glavic, B., Lee, S. & Chapman, A. (2026) ‘Efficient query repair for aggregate constraints’, *Proceedings of the VLDB Endowment.* **Accepted**.

These papers present the key concepts, algorithms, and experimental results that underpin the contributions of this thesis.

Chapter 2

Literature Review

Data is a central asset for organizations across domains such as healthcare, finance, and education, where it supports decision-making, analysis, and compliance. These needs are met through a range of data systems, including relational databases (Minker and Sable, 1970; Yang et al., 2009), online analytical processing (OLAP) systems (Alkharouf et al., 2005; Joglekar et al., 2017; Patel and Sharma, 2020), cloud data warehouses (Rehman et al., 2018; Li et al., 2024), and more recently, data marketplaces where structured datasets are traded between entities (Abbas et al., 2021; Azcoitia and Laoutaris, 2022).

Databases play a vital role in a wide range of domains that require reliable and large-scale storage and access to information, including enterprise applications (Jani, 2022; Oloruntoba, 2025), scientific research (Subash et al., 2023), healthcare (Park and Lee, 2021), and e-commerce (Kedah, 2023; Petrenko and Kravtsov, 2023). With the ongoing growth in both the volume and complexity of data, the role of databases has expanded beyond basic data retrieval to include support for query formulation, data exploration, and decision-making (Elmasri and Navathe, 2016). Databases are commonly classified into two main categories: relational databases and non-relational (NoSQL) databases. Relational databases are a type of database that organizes data into formally defined tables consisting of rows and columns which makes them well-suited for structured data to facilitate efficient storage, retrieval, and manipulation of information. Fundamentally, a database system is intended to manage large scale of data efficiently while ensuring critical properties like consistency, integrity, and availability. Modern database management systems (DBMSs) provide high-level query languages, most notably SQL, allowing users to express complex data retrieval and transformation tasks declaratively (Abiteboul et al., 1995). In contrast, non-relational databases can store data in various formats such as key-value pairs, XML, or multidimensional structures, rather than using traditional tables (Jatana et al., 2012). Beyond these two categories, other types also exist, such as column-oriented databases (Abadi et al., 2009) and network databases (Belfkih et al., 2019) where each with its own model and specialized use cases.

The foundational theory of relational database systems (SQL-based databases) began in the 1970s with Codd's introduction of the relational model (Codd, 1970). Despite their early origins, these technologies continue to play a central role in modern data management. In fact, the rise of Big Data and the increasing adoption of data science techniques for decision-making have only reinforced the relevance of relational databases and SQL. While advanced methods such as machine learning and complex statistical analysis are expanding the scope of data insights, a significant portion of decision-support tasks still depend on SQL-based descriptive statistics and grouped aggregations (Kaufmann and Meier, 2023).

Relational databases can be deployed in different configurations depending on the purpose and workload. Two common setups are Online Transaction Processing (OLTP) (El-Sayed et al., 2021; Plattner, 2009) and Online Analytical Processing (OLAP) (Alkharouf et al., 2005; Joglekar et al., 2017; Patel and Sharma, 2020). OLTP systems are optimized for handling a large number of short, concurrent transactions and are widely used in applications such as banking, retail, and reservation systems. In contrast, OLAP systems are designed for complex analytical queries over multi-dimensional database, often used in decision-support and reporting tasks (Kaufmann and Meier, 2023). These systems focus on aggregated summaries and trends rather than individual records. To evaluate performance across these workloads, the Transaction Processing Performance Council (TPC) has established a set of benchmarks, including TPC-C and TPC-E for Online Transaction Processing (OLTP), and TPC-H and TPC-DS capture key features of decision support systems, such as complex queries and ongoing data maintenance. These benchmarks are widely used in both research and industry to assess scalability, query performance, and overall system efficiency (Nambiar et al., 2012). Because relational databases (SQL-based databases) are the central focus of this thesis, this work expand on their operational configurations and evaluation methods. In particular, this thesis uses TPC-H as a benchmark for assessing the performance of the proposed techniques.

Practitioners such as analysts and data scientists query these relational databases to extract meaningful insights through queries. However, these queries often serve goals that extend beyond retrieving records that match specific conditions, they must also satisfy broader organizational objectives that includes fairness, representativeness, or business rules. For example, a public-sector analyst may require a dataset that includes a balanced distribution across demographic groups, or a marketing team may need results that proportionally represent customer segments. Although modern Database Management Systems DBMSs are built, users still face difficulties when queries do not produce the expected results. As a result, there has been growing research interest in improving database usability and supporting users in understanding query behaviour (Herschel et al., 2017; Chapman and Jagadish, 2009). This includes techniques such as query debugging, data and query refinement, and provenance tracking. Among these, data provenance plays a foundational role by helping to determine where data comes from, how query results were produced, and why certain tuples are included or missing. Provenance has been instrumental in supporting query debugging, explanations (why/why-not answers), data trust, and auditing. A comprehensive survey by Cheney et al. (2009) outlines the major motivations, models, and applications of provenance in databases. More recent work integrates provenance with goals such as fairness and bias mitigation (Moskovitch et al., 2022; Shetiya et al., 2022; Li et al., 2023; Campbell et al., 2024), demonstrating its relevance to constrained query evaluation and responsible data analysis.

Data refinement (DR) and Query refinement (QR) are two closely related paradigms, both of which aim to bridge the gap between user intent and the actual query output. DR techniques modify the data instance to ensure that query results satisfy user-defined constraints, whereas QR techniques focus on rewriting or relaxing the query itself. Both have been studied in settings involving tuple inclusion/exclusion, cardinality thresholds, or user feedback. This thesis uses the term *query repair* to refer to changes made to query predicates through a combination of relaxation (weakening a predicate) and refinement (strengthening a predicate). The more specific terms query refinement and query relaxation are used in prior work to distinguish between these operations when applied individually. However, for simplicity, this thesis will collectively refer to both refinement and relaxation as refinement, unless otherwise specified.

This chapter reviews prior work in two areas relevant to this study: data refinement and query refinement (Section 2.1), with particular emphasis on the latter, which forms the core of this thesis. Existing query refinement techniques are categorized based on whether they address non-aggregate constraints (Section 2.1.2.1) or aggregate constraints (Section 2.1.2.2). For each category, the analysis focuses on how different approaches handle query similarity, constraint satisfaction, and support for arithmetic expressions. Key limitations in existing work are also highlighted to position the contribution of this thesis within this under explored area. Finally, Section 2.2 summarizes the key ideas discussed in this chapter.

2.1 What is Data Refinement versus Query Refinement?

Data Refinement (DR) and Query Refinement (QR) are two distinct techniques in database management, both aimed at satisfying user-defined constraints on query output. While DR focuses on modifying the underlying input data, QR involves modifying queries to retrieve the intended results. The following sections briefly review existing techniques for DR, while providing a more detailed discussion of QR approaches, as query refinement constitutes the primary focus of this research.

2.1.1 Data Refinement Techniques (DR)

The goal of Data Refinement (DR) techniques is to modify the input data to ensure the desired output is achieved (Meliou and Suciu, 2012; Wang et al., 2017). Unlike query refinement approaches, DR focuses on modifying the database instance rather than the query structure or parameters.

A key example is how-to queries (Meliou and Suciu, 2012), which focus on modifying the input data rather than modifying the query itself to obtain the intended result. The authors present a novel approach that translates how-to queries into Mixed Integer Programming (MIP) problems, enabling automated data modifications that satisfy specific constraints. To achieve this, they introduce TiQL (Tiresias Query Language), an extension of Datalog, which generates hypothetical tables representing possible database modifications, such as modifying attribute values, deleting tuples, or creating new tuples. These modifications are formulated as MIP variables, with constraints expressed as linear inequalities. A MIP solver is then used to determine the minimal or most efficient changes required to satisfy the constraints while minimizing or maximizing a specified objective function by the user (e.g., minimizing changes to the database). The system considers optimization techniques such as partitioning the problem into smaller sub-problems, and eliminating redundant variables and constraints to handle large datasets and complex queries efficiently. The final result is a set of hypothetical database updates that meet the desired constraints.

Explaining missing answers is a related problem addressed in works such as (Herschel and Hernández, 2010). This work proposes a system that facilitates the analysis of SQL queries created by developers by allowing the investigation of why certain tuples are missing from the query results. It specifically targets missing answer explanations for queries involving selection, projection, join, union, aggregation, and grouping (SPJUA) operations. The authors introduce a general framework for generating instance-based explanations by identifying the data that would need to be added to produce the missing answer. A key element of their approach is the use of a constraint solver to ensure that all generated explanations are consistent with both the database constraints and user-specified requirements. To achieve this, the problem is formulated as a set of logical constraints, and a solution is considered valid only if the constraint solver finds it satisfiable solution. This method is particularly useful for debugging complex queries, where the absence of expected results may not be immediately clear. The framework also considers potential side effects caused by introducing new data and provides mechanisms to help users reduce or eliminate such effects. Additionally, the system aims to return explanations that are both complete and minimal.

These data refinement approaches differ fundamentally from query refinement in terms of their objective. Data refinement (DR) focuses on modifying the database content to satisfy desired query outputs. However, this is not always a viable or appropriate solution, particularly in contexts like fairness where altering data may be legally or ethically problematic. In contrast, query refinement (QR) aims to adjust the query structure or predicate constants to achieve specified constraints. In scenarios involving aggregate functions or complex constraints, DR may require substantial modifications to the data, whereas query refinement can often provide more targeted and efficient solutions. Furthermore, existing DR approaches typically do not support query refinement or handle aggregate constraints of the kind addressed in this work. Nonetheless, DR techniques remain valuable in use cases where the query must remain fixed due to application constraints or legacy systems, and adjustments to the data are permissible to meet output requirements.

2.1.2 Query Refinement Techniques (QR)

Query Refinement is the process of modifying a query when the initial results of the query does not align with the user's expectations (Chaudhuri, 1990). Specifically, this involves automatically modifying the query to ensure that the modified results better reflect the user's intended constraints (Albarrak and Sharaf, 2017). A constraint on a query result reflects the user's expectations regarding the desired output and can take various forms. These constraints can be broadly categorized into two types. One category is *Query Refinement for Diverse Non-Aggregate Constraints (QRN)*, which includes constraints such as requiring the query output to include or exclude specific tuples, either explicitly provided by the user or inferred as expected or unexpected. QRN also includes constraints on the size of the result set. Another category is *Query Refinement for Aggregate Constraints (QRA)*, which involves constraints based on aggregation functions such as **MIN**, **MAX**, **SUM**, and **COUNT**. Each of these constraint types requires refinement strategies to adjust query predicates so that the resulting output better aligns with user intent.

An important consideration in refining queries under aggregate constraints is the *monotonicity* of the aggregation function. An aggregate function is said to be *monotonic* if adding tuples to the dataset does not violate the constraint, i.e., the output of the function changes in a predictable direction. For instance, functions like `COUNT` and `SUM` exhibit monotonic behaviour because adding more tuples to the input cannot cause their values to decrease. This property allows for effective pruning in query refinement, as any candidate query that already violates the constraint will continue to do so even if more data is included.

More formally, consider a candidate query R defined by d range predicates, where each predicate P_i filters the attribute a_i using a value from its domain. Let each attribute a_i

have n_i distinct values. The space of possible outcomes for such queries can be represented as a d-dimensional grid $\mathcal{G}$, where each dimension corresponds to a predicate's possible values. A query result, such as its cardinality (i.e., the number of tuples returned), can be represented as a value associated with a point in this grid. This representation enables reasoning over query behaviour across different predicate settings.

Let $\mathcal{G}[t_1, \ldots, t_d]$ represent the number of tuples returned when each predicate is assigned the smallest distinct value of its corresponding attribute a_i. The query space exhibits *monotonicity* if increasing the threshold for any predicate (i.e., $t_i \leq y_i$ for all i) results in a query that returns at least as many tuples: $\mathcal{G}[t_1, \ldots, t_d] \leq \mathcal{G}[y_1, \ldots, y_d]$. This means that relaxing the predicates (i.e., expanding the filter ranges) will never reduce the number of returned results (Bruno et al., 2006).

However, some constraints are defined over *non-monotonic* aggregates, where adding or removing tuples can cause the aggregate value to increase or decrease unpredictably. For instance, ratio-based constraints often exhibit non-monotonic behaviour. In such cases, common optimization strategies like monotonic pruning break down. This thesis addresses precisely this challenge by designing query refinement methods for non-monotonic aggregate constraints.

Example 2.1. *A computer science department within a university needs to assign at least three faculty members to review a research proposal on "Deep Learning for Medical Image Analysis." Initially, the database is queried for faculty members with expertise in both deep learning and medical imaging. If fewer than three reviewers are found, the query is relaxed to include faculty with expertise in machine learning and medical imaging, as machine learning expertise encompasses deep learning. If the required number of reviewers is still not met, the constraints may be further relaxed to include faculty with expertise in artificial intelligence and Medical Image.*

In Example 2.1, the absence of expected number of tuples from the query results represents one form of constraint.

Refining a query involves modifying the query's predicates in various ways. These modifications can include adding or removing predicates, changing a constant within a predicate to another constant, as well as relaxing the predicate (i.e., generating more tuples in the output) or tightening the constants within predicates (i.e., generating fewer tuples in the output) (Li et al., 2023), etc. For example, in Example 2.1 where a query initially searches for reviewers with expertise in both deep learning and medical imaging, relaxing the predicates constants to include those with expertise in machine learning and medical imaging allows for a broader set of reviewers, thereby generating more result tuples. On the other hand, tightening the predicate constant by only considering reviewers with expertise in deep learning results in fewer returned tuples, narrowing the scope of the query.

The following sections review existing query refinement techniques developed to address user-defined constraints. They examine how these techniques search for refined queries, the way used to modify query, and the optimization strategies employed. Additionally, user-defined constraints are categorized into two main types: diverse non-aggregate constraints and aggregate constraints.

2.1.2.1 Query Refinement for Diverse Non-aggregate Constraints (QRN)

QRN is a query refinement technique designed to satisfy user-defined non-aggregate constraints (Mottin et al., 2016b; Muslea, 2004; Muslea and Lee, 2005; Mottin et al., 2014b). These constraints include ensuring that the query result is not too large, too small, or empty (Mottin et al., 2016b), enforcing the inclusion or excluding of specific tuples in the output (Muslea, 2004; Muslea and Lee, 2005), and filtering out outliers or undesired tuples (Roy and Suciu, 2014; Wu and Madden, 2013; Tran et al., 2014; Shen et al., 2014; Dimitriadou et al., 2016). **Non-aggregate constraints** refer to conditions applied to the overall composition of the query output. The following sections review existing approaches that address such constraints. While these methods are relevant to the discussion in this thesis, this section briefly cover three main categories: query refinement for user-specified tuples, expected and unexpected tuples, and size constraints, as they address problems that are related to, but fundamentally different from, the focus of this work. Table 2.1 builds upon the original table by Albarrak (2018), which categorized query refinement (QR) techniques based on constraint types such as expected tuples, unexpected tuples, and result size. The original table summarized each constraint with an informal definition and cited supporting literature. In this thesis, the table has been restructured to better highlight the relevance of these techniques to the problem of this thesis. Specifically, the following enhancements were made: (i) the **Constraint** types have been reorganized into three main categories that reflect the key types of non-aggregate constraints relevant to this section. Constraints involving aggregates are discussed separately in Section 2.1.2.2. (ii) a new **Limitations** column has been added to analyse why existing techniques for non-aggregate constraints fall short in addressing the more complex aggregate constraints explored in this thesis. In these approaches, refining a query requires navigating a large search space to identify an optimal refined query. This task is challenging, as it often involves interactive user feedback to guide the refinement process or requires the adoption of optimization strategies, such as ensuring similarity to the original user query.

2.1.2.1.1 Query Refinement for User-specified Tuples This section reviews related work that addresses the problem of refining queries to produce results that include user-specified tuples (Zloof, 1975; Dimitriadou et al., 2016; Shen et al., 2014;

TABLE 2.1: Adapted from Albarrak (2018), this table summarizes query refinement techniques for diverse non-aggregate constraints with added limitations relevant to this thesis.

Technique	**Constraint Type**	**Informal Definition**	**Related Works**	**Limitations**
Query Refinement for User-specified Tuples	Query's result contains a set of tuples T.	Formulate a query Q based on example tuples T provided by the user.	(Zloof, 1975; Tran et al., 2009; Shen et al., 2014; Dimitriadou et al., 2014; Tran et al., 2014; Li et al., 2015; Dimitriadou et al., 2016; Mottin et al., 2016a; Barceló and Romero, 2016; ten Cate et al., 2025)	Constraints are defined at the level of individual tuples, and users are required to provide specific examples that should appear in the result. In contrast, the constraints in this work are specified at a higher level and users are typically unfamiliar with the specific tuples that will satisfy them.
Query Refinement for Expected and Unexpected Tuples	Ensure expected tuples E are included and unexpected tuples U are excluded from the query result	Refine Q to Q' such that Q' includes E and excludes U.	(Chapman and Jagadish, 2009; Tran and Chan, 2010; He and Lo, 2012; Islam et al., 2012, 2013a,b, 2015; Sharaf and Ehsan, 2021; Erbacher et al., 2022; Lee et al., 2023)	Constraints assume the user can identify specific tuples that should or should not appear in the result. While this work considers higher-level constraints based on aggregate computations over groups of tuples. Users are not expected to know which individual tuples should satisfy the constraint, making these refinement techniques impractical for the setting in this work.
Query Refinement for Size Constraints	Query's result size $\geq K$, $\leq K$, or > 0	Given that Q returns too few, too many or an empty result, minimally refine Q to Q' to satisfy the constraint.	(Muslea, 2004; Chaudhuri et al., 2004; Muslea and Lee, 2005; Ilyas et al., 2008; Mottin et al., 2014b, 2016b; Ikeda et al., 2024)	Designed to improve query results that are too small, too large, or completely empty.

Mottin et al., 2016a; Dimitriadou et al., 2014; Tran et al., 2009; Li et al., 2015; ten Cate et al., 2025; Tran et al., 2014; Barceló and Romero, 2016).

Early foundational work by Zloof (1975) introduced Query-by-Example (QBE), which enables users to specify query conditions via example tuples. QBE is particularly useful for users without database expertise such as professionals from domains like life sciences, the arts, or business, who may be unfamiliar with traditional query languages and possess limited programming knowledge (Bonifati et al., 2016, 2019). Such users are typically able to supply data examples, but they frequently find it challenging to express their information needs using correct SQL syntax (Shen et al., 2014; Martins, 2019).

Query by Example (QBE) aims to reconstruct queries to support a range of applications, such as retrieving relevant information from datasets, improving users' understanding of database sachems and selecting data that satisfies user interests (Martins, 2019; Tran et al., 2009). This concept evolved into modern frameworks like TALOS (Tran et al., 2009), a data-driven framework for Query Reverse Engineering (QRE) that reconstructs the user query based on the result of an unknown query and the underlying database. The core idea is to frame the task as a data classification problem, where database tuples are labelled as positive if they appear in the result set, and negative otherwise. TALOS constructs a decision tree by recursively splitting tuples based on attributes, selecting splits that optimize a criterion such as the Gini index or entropy, until each leaf node contains only positive or negative tuples. Each path from the root to a positive leaf corresponds to a candidate query, and these queries are then ranked using two variations of the popular F-measure, capturing both precision and recall. TALOS leverages several optimization techniques to enhance the performance, including precomputed join indices, hub tables, and mapping tables to efficiently minimize the computational cost of tree construction.

Shen et al. (2014) addresses the challenge of efficiently discovering minimal project-join SQL queries that include user-provided example tuples in their output. The large search space and the high cost of evaluating each candidate query make this problem computationally challenging. Two phases proposed by the authors. First, candidate generation to produce all set of potential refined queries. Second, candidate verification to filter out invalid queries. The paper introduces a filter-based pruning technique to optimize this process, that exploits dependency relationships among queries. So, if a candidate query is invalid, all of its sub-queries can be pruned, because they are more restrictive and unlikely to produce any valid results. Similarly, Mottin et al. (2014a) introduce the concept of exemplar queries, where the user provides a representative query instance, and the goal is to retrieve structurally similar query patterns from the database. This approach is particularly useful in scenarios where users may not fully understand the structural characteristics of their intended queries. Both the user input and the database are modeled as labeled graphs, and the task becomes one of identifying isomorphic subgraphs in the data graph that resemble the exemplar query. To address the high computational cost of subgraph matching, the authors propose two optimized algorithms. FastXQ treats the exemplar query as a connected subgraph and prunes unqualified nodes by compactly summarizing each node's neighborhood, thereby avoiding exhaustive traversal of the entire data graph.

To further improve scalability, ApFastXQ offers an approximate solution by leveraging an adaptive Personalized PageRank strategy to prioritize and restrict the search to the most relevant regions of the data graph.

Another type of refinement framework focused on producing query results that include user-specified tuples is interactive query refinement, which incorporates user feedback into the refinement process (Li et al., 2015). Automatic Interactive Data Exploration (AIDE) is designed for scenarios where users cannot formulate precise queries but can instead label a small number of data samples as relevant or irrelevant. The system uses this feedback to incrementally build a classification model that captures the user's interests, which is then translated into a query when the user decides to terminate the process. AIDE's goal is to reduce the number of samples the user needs to label while accurately identifying all relevant data. It follows a three-phase iterative process to achieve this. First, the data space is divided into equal-width grid cells. For each cell, the system selects the tuple closest to the center of the cell for the user to label. Cells that yield irrelevant tuples are recursively split into finer-grained sub-cells, as they may partially overlap with relevant regions. Second, the system refines the classification model by focusing on misclassified examples, specifically false positives (tuples labelled irrelevant by the user but predicted as relevant) and false negatives (tuples labelled relevant but predicted as irrelevant). This polishing phase helps adjust the decision boundaries and better align the model with the user's actual interests. In the third phase, the system chooses a few examples that lie near the boundary between what it predicts as relevant and irrelevant. The user labels these examples to help the system refine the boundary and improve the final query. To reduce the overhead of repeated sampling, AIDE applies two key optimizations: (1) using simple random sampling of the dataset to create a manageable working dataset, and (2) adaptively reducing sample tuples that will be labelled by the user by tracking and remembering boundary changes, if a region's boundary changes only minimally between iterations, AIDE considers it well-learned and reduces further sampling there.

The refinement approaches addressed in the above works are fundamentally different from those considered in this thesis, which involves refining queries to satisfy constraints expressed as arithmetic combinations of aggregate functions. In particular, these approaches focus on constraints defined at the level of individual tuples, whereas the constraints considered in this thesis apply to groups of tuples in the query result. Moreover, these methods generally assume that users are able to identify specific tuples that should appear in the result. In contrast, the setting in this work assumes that users do not know in advance which tuples will satisfy the constraints, which makes these techniques unsuitable for the problem problem considered in this thesis. As a result, the strategies proposed in these works cannot be directly applied to the type of constraints address here.

2.1.2.1.2 Query Refinement for Expected and Unexpected Tuples This section reviews related work that addresses the problem of refining queries to produce results that either include user-expected tuples or exclude unexpected ones (Chapman and Jagadish, 2009; He and Lo, 2012; Islam et al., 2012, 2013a,b, 2015; Tran and Chan, 2010; Lee et al., 2023; Sharaf and Ehsan, 2021; Erbacher et al., 2022).

This section examines the line of work addressing the refinement of queries to satisfy user expectations by including expected tuples and excluding non-expected tuples, which are often referred to as why and why-not (Chapman and Jagadish, 2009; Tran and Chan, 2010; He and Lo, 2012). This type of techniques are beneficial in environments where users have limited access to the underlying database, such as web-based airline ticket search platforms (Chapman and Jagadish, 2009). The aim of this type of query refinement is to reconcile the gap between the user's desired output and the actual query result, either by modifying the query to include missing (expected) tuples or to filter out irrelevant (non-expected) results.

A foundational contribution to this area is the ConQueR (Constraint-based Query Refinement) framework, which addresses user questions about why certain expected tuples are missing from query results (Tran and Chan, 2010). ConQueR automatically generates refined queries that return the original result plus the missing tuples, while preserving the user's original intent as much as possible. The framework supports both SPJ and SPJA queries and evaluates refined queries using two metrics: dissimilarity (how much the refined query deviates from the original query) and imprecision (how many irrelevant tuples are introduced). ConQueR returns a set of skyline refined queries that are optimal trade-offs between these two metrics. It proceeds in two stages: ConQueRs, which generates minimally modified queries by relaxing selection predicates to include the missing tuples while keeping the refined query as similar as possible to the original (measured by edit distance), and ConQueRp, which enhances precision of these queries by adding new predicate to filter out irrelevant results. When the original query's structure is not suitable for retrieving the expected tuples, ConQueR explores alternative query schemas, modifying joins and relations if needed. For queries with aggregates, it supports refinement strategies that consider aggregation constraints. To ensure scalability especially when dealing with large databases and complex queries, ConQueR employs greedy search and pruning technique to minimize the number of candidate queries it needs to evaluate. The greedy search strategy means that the system incrementally builds refined queries by choosing relaxing or adding predicates that seem most promising at each step, rather than exhaustively exploring all possible modifications. This helps the system reach good solutions quickly without evaluating every possible option. At the same time, pruning techniques are applied to discard parts of the search space early. If a partial refinement is already too dissimilar or returns too many irrelevant tuples, that query path is no longer explored.

Similarly, Islam et al. (2012) presents a framework that generates explanations for both unexpected tuples returned by the query and expected tuples that are missing, based on user-provided feedback. These explanations are derived by analysing which clauses in the user's original query are satisfied or violated by the feedback tuples. By identifying the precise conjuncts (in Conjunctive Queries) or disjuncts (in Disjunctive Queries) responsible for including or excluding the tuples, the system pinpoints which parts of the query may need modification. This explanation-driven refinement provides a structured and minimal-change approach to modifying the query. The model also applies point domination theory to infer additional feedback by comparing unlabelled tuples with user-labelled ones across multiple attributes. If an unlabelled tuple dominates or is dominated by a labelled example, the system assumes the user would likely consider it similarly expected or unexpected. Building on the work by Islam et al. (2012), Islam et al. (2013a) operationalizes their model through a practical query refinement system. The authors introduce a taxonomy of feedback (explicit and implicit) and demonstrate how implicit feedback can be inferred using point domination, which helps reduce the burden on users to manually label all tuples. The refinement process is formalized as an optimization problem over query predicates, aiming to either relax predicates to include missing expected tuples or tighten them to exclude unexpected ones. Since selecting the optimal set of predicates is NP-hard, the framework adopts a greedy approximation algorithm that maximizes information gain while minimizing deviation from the original query. The system supports two refinement strategies: soft exclusion, which aims to eliminate unexpected (irrelevant) tuples without removing any expected ones, and hard exclusion, which prioritizes removing all unexpected tuples, even if some expected ones are also excluded in the process.

Beyond purely query-based refinement, recent work by Lee et al. (2023) addresses the limitations of traditional query debugging approaches that treat query and data errors in isolation. The authors introduce hybrid explanations and repairs, where both the query and the data may be modified to resolve user complaints i.e., expected tuples that are missing or unexpected tuples that are present. Unlike traditional methods that return a single optimal repair, the authors introduce an interactive framework that enables users to navigate multiple repair options while defining limits on how much the query, data, and output can be changed. A hybrid repair is formalized as a pair consisting of a modified query and an updated database that together satisfy the user's intent with bounded side effects. The system extends provenance-based techniques to generate graph-based hybrid explanations, which visually highlight how specific parts of the query and data contribute to the erroneous results. These explanations guide the user by indicating which components might need to be changed. Instead of relying on a single optimization criterion, the framework promotes a flexible, user-guided approach that balances correctness with minimal changes to the original query and data. This empowers users to explore meaningful repair strategies that best align with their domain knowledge and preferences.

Query refinement has also been explored beyond traditional structured query processing, especially in the domains of visual data exploration and technical information retrieval. In the context of visual data, Sharaf and Ehsan (2021) introduce the QuRVe framework, which refines user queries to suggest more insightful visualizations. The goal of query refinement here is to enhance exploratory analysis by automatically proposing alternative visual views that are interesting, relevant to the user's intent, and statistically significant. The authors formulate this as a multi-objective optimization problem, balancing three factors: the deviation from the global data distribution (i.e., how interesting the result is), similarity to the original user query (preserving intent), and the statistical strength of the visualized output. To address the scalability challenge posed by the large space of possible refinements, they introduce two optimized variants: uQuRVe, which uses tighter deviation bounds to prune low-utility queries, and pQuRVe, which applies a round-robin prioritization strategy to focus on the most promising refinements.

These studies show that in practice, users rarely formulate the perfect query on the first try, which makes it important to support ways of refining the query so that the results better reflect what the user is actually looking for. Incorporating user feedback, whether provided directly or inferred automatically, has become a central element in recent query refinement approaches. Recent work has further advanced this area by introducing interactive, hybrid, and visualization-oriented techniques that systematically address both the inclusion of relevant tuples and the exclusion of irrelevant ones, while maintaining scalability and practical usability.

In summary, the query refinement problems addressed by the techniques discussed above rely on constraint types that differ fundamentally from those considered in this thesis. Extending those methods to support the kinds of constraints explored here is not practical. In the setting of this work, users do not specify particular tuples that should or should not appear in the query result. Instead, they define higher-level constraints that the output must satisfy. These constraints are typically based on aggregate computations over groups of tuples, where a single aggregate condition may involve a large number of tuples. As such, expressing these constraints at the level of individual tuples would be highly impractical. Moreover, while prior approaches focus on the inclusion or exclusion of specific tuples in the query output, the approach in this thesis ensures that arithmetic expressions over aggregate values satisfy user-defined thresholds on the query result. These fundamental differences in both the nature of the constraints and the underlying problem formulation make existing techniques unsuitable for direct application to the setting addressed in this thesis.

2.1.2.1.3 Query Refinement for Size Constraints Another line of work in query refinement focuses on handling problems where the query output is either too small, too large, or completely empty. These kinds of issues often lead to practical challenges in data exploration systems. The proposed approaches aim to modify queries, either automatically or with user interaction, to produce output results that are practical and usable size.

The empty-answer problem arises when a user submits a query that is overly restrictive, resulting in no returned results. It is considered a special case of the broader too-few answers problem, where the query yields very limited or no relevant results at all. To address this problem, a series of works have proposed interactive query relaxation frameworks that aim to incrementally guide users toward modified queries that produce meaningful answers. These methods are especially important in exploratory search scenarios such as e-commerce or information portals, where users may not have complete knowledge of the database content and thus reformulate queries that fail to retrieve any relevant items.

An early system that addresses this issue is IQR (Interactive Query Relaxation) (Mottin et al., 2014b), which introduces a step-by-step interactive approach to relaxing queries that return empty results. The system incrementally proposes one predicate relaxation at a time, aiming to eventually yield a non-empty result or determine that no such result is possible. Each relaxation is evaluated using a probabilistic cost model that considers both the likelihood of user acceptance and how well the resulting tuples align with a user-defined optimization goal (e.g., maximizing relevance or minimizing effort). This is achieved through two key components: a prior function, which estimates the likelihood that a user believes a tuple satisfying the relaxed query exists in the database, and a preference function, which models how desirable the user finds the tuple. These are combined to score and rank candidate relaxations. To address the computational challenge of the large relaxation space, IQR includes both exact and approximate algorithms that explore only the most promising parts of the relaxation tree. It uses pruning strategies based on upper and lower cost bounds to eliminate subtrees that cannot yield optimal results, significantly improving scalability.

Building on the foundations of IQR, a more comprehensive solution is proposed in (Mottin et al., 2016b), which extends the model into a holistic and principled optimization-based interactive framework. The authors present a probabilistic approach to query relaxation, where the system incrementally suggests relaxed versions of the original query and incorporates user feedback to guide the process. Query relaxation is modelled as a tree, in which each node represents a query generated by removing one or more atomic predicates, and the root corresponds to the original query that yields no results. Each relaxation path is evaluated using a probability score that reflects the likelihood of user acceptance of the proposed query. Two algorithms are introduced for navigating the relaxation tree: FullTree, which performs a complete depth-first exploration of all possible relaxations to find the optimal one; and FastOpt, which improves scalability by pruning unpromising branches using upper and lower bounds on acceptance probabilities. This pruning significantly reduces computational overhead by avoiding exploration of unpromising paths. A major limitation of the proposed approach is that it was originally developed for Boolean databases, which limits its direct applicability to environments involving categorical or numerical attributes.

Machine learning-based approaches also play a central role in addressing the empty-answer problem. LOQR by Muslea (2004) and its enhanced version TOQR by Muslea and Lee (2005) use online learning techniques to generate relaxed queries that avoid returning empty results. LOQR applies rule-based learning and nearest-neighbor matching to identify minimal changes that can lead to a non-empty result. TOQR improves on this by using Bayesian networks to model causal relationships between query attributes. These networks are trained on a representative dataset to learn how certain attributes influence one another. This allows TOQR to determine a more effective order for applying relaxations, prioritizing those that are more likely to produce valid results.

Recent work by Ikeda et al. (2024) addresses the empty answer problem with a focus on producing diverse and relevant query results. Traditional query relaxation methods often generate results that may similar to each other, which can limit their usefulness to users seeking a broader view of the data. To tackle this, the authors introduce a two-stage method that incorporates Maximal Marginal Relevance (MMR), a ranking strategy that balances similarity to the original query and diversity in the output. In the first stage, the system generates multiple relaxed versions of the original query and ranks them using a query-level MMR score. In the second stage, it selects a diverse and relevant top-k set of records using a record-level MMR score. To ensure efficiency, the method employs a greedy relaxation strategy and a hybrid evaluation that combines cardinality estimation with selective direct evaluation, reducing computation while maintaining result quality.

On the other hand, when a query returns too many results often due to under-specification or broad selection conditions, it becomes difficult for users to identify relevant information. One solution is top-k query processing, which returns only the most relevant k tuples based on a scoring function (Ilyas et al., 2008). Another is the skyline query approach, which filters out dominated tuples and returns only those that are optimal across multiple dimensions (Borzsony et al., 2001). A more adaptive strategy involves probabilistic ranking functions, which consider consider not only the attributes explicitly specified in the query but also those left unspecified. Chaudhuri et al. (2004) propose ranking tuples using two components: a global score, which reflects the overall importance of unspecified attribute values in the dataset, and a conditional score, which measures how strongly these unspecified values are correlated with the attributes specified in the query. This allows the system to prioritize tuples that are more likely to align with the user's intent, even when that intent is only partially expressed.

In summary, the query refinement techniques reviewed above focus on constraints that are fundamentally different from those addressed in this thesis. These methods are designed to improve query results that are too small, too large, or completely empty, often through user interaction, adaptive learning, or relevance-based ranking. While effective in their respective settings, they do not handle constraints involving arithmetic expressions over aggregate functions, which are central to the problem of this thesis. Some of the reviewed techniques, such as IQR and its extensions (Mottin et al., 2014b, 2016b), are limited to boolean attributes and cannot be directly applied to queries involving categorical or numerical values. Others, like LOQR and TOQR (Muslea, 2004; Muslea and Lee, 2005), support refinement based on the size of the query result.

Moreover, a key distinction lies in the interaction model. Many of the existing approaches depend on iterative user feedback or preference learning to guide the refinement process. In contrast, the framework in this thesis assumes that the user specifies a target numerical constraint in advance, such as a threshold defined over arithmetic combinations of aggregate functions, and expects the system to refine the query accordingly without requiring further interaction. Given these fundamental differences in both of the constraints and the refinement objectives, existing techniques are not applicable to the problem addressed in this thesis.

2.1.2.2 Query Refinement for Aggregate Constraints (QRA)

QRA is a query refinement technique designed to satisfy user-defined aggregate constraints. These constraints require refinement strategies to ensure that the query results align with user intentions. **Aggregate constraints** refer to conditions imposed on the results of aggregation functions such as **min**, **max**, **sum**, and **count**, which operate on various database attributes to derive summary information. For instance, a constraint like *sum(Salary)* $\geq$ 15000 ensures that the total salary meets a specified threshold (Ross et al., 1998).

Aggregate constraints can be applied either to the overall query results (Mishra et al., 2008; Bruno et al., 2006; Albarrak et al., 2014; Mishra and Koudas, 2009; Koudas et al., 2006; Tran and Chan, 2010) or, as explored in more recent studies, to specific groups within the results (Li et al., 2023; Shetiya et al., 2022). In the following sections, a review of existing works on aggregate constraints in these two categories is presented, by examining their way of modifying the query, their search strategies and the optimizations they employ.

A summary comparison of these techniques is shown in Table 2.2. The table highlights key differences across existing query refinement approaches that target aggregate constraints, focusing on the types of aggregation functions supported, whether refinement preserves similarity to the original query, the types of predicate modifications applied, the scope at which constraints are enforced (e.g., result-level, group-level, or top-k), and whether the technique can handle arithmetic expressions over aggregates. The final column summarizes the limitations of each method and explains why it is not applicable to the setting considered in this thesis.

2.1.2.2.1 Query Refinement for Aggregate Constraints over the Result Size Research has increasingly focused on refining database queries to meet specific criteria, particularly aggregate constraints on the result size. The literature reviewed here presents a range of approaches, optimizations, and conceptual frameworks that address these challenges.

Several studies have proposed automatic query refinement techniques that incrementally adjust selection predicates to satisfy global aggregate constraints on the query output. For instance, in database testing, it is often necessary to assess the impact of integrating a new component by executing queries that meet predefined test criteria (Bruno and Chaudhuri, 2005). These criteria typically involve constraints on result sizes, and satisfying them requires automatically modifying query predicates until the resulting output aligns with the intended test scenario.

As demonstrated by Bruno et al. (2006), generating a query based on a single cardinality constraint on its output has been proven to be an NP-hard problem. In the same work, the authors proposed a heuristic Hill-Climbing (HC) technique to address the challenge of automatically generating queries that meet cardinality constraints, allowing for approximate solutions that closely align with the specified constraints. The HC method iteratively searches for predicate constants of a query that reduce the average relative error in satisfying cardinality constraints. For a given cardinality constraint, the relative error is defined as the maximum of two ratios: the target cardinality divided by the observed cardinality, and the observed cardinality divided by the target cardinality. The HC process begins by assigning initial predicate constants that best approximate the desired constraints with a minimum relative error, assuming independence among predicates. From this starting point, it iteratively modifies individual predicate constants, selecting the change that most reduces the relative error. If no reduction is found, the step size is halved, and this halving process continues until the step size is small enough to distinguish between individual distinct values in the predicate domain. A key limitation of this technique, is that during the refinement process it does not account for similarity to the original query. This approach focuses on satisfying cardinality constraints by adjusting predicate parameters using heuristics hill climbing based on local error measures. Incorporating query similarity, ensuring that the refined query remains close to the user's original intent, introduces new challenges. For instance, HC selects refinement steps by evaluating local relative error, so it can easily become trapped in local minima when similarity constraints are also considered.

TABLE 2.2: Comparison of some Aggregate-based Query Refinement Techniques

Techniques	**Aggregates Supported**	**Similarity to Original Query Considered**	**Query Modification Type**	**Constraint Application Level**	**Arithmetic Expression Supported**	**Limitations**
(Bruno et al., 2006)	COUNT	✗	Constants of selection predicates	Result size	✗	Lacks support for similarity to the original query, assumes monotonic cardinality constraints (invalid for non-monotonic cases), and restricts to COUNT aggregates over result size.
(Mishra et al., 2008)	COUNT	✗			✗	
(Mishra and Koudas, 2009)	COUNT	✗			✗	
(Vartak et al., 2010)	COUNT	✓	Constants of selection and join predicates		✗	Requires manual tuning, relies on monotonicity, does not consider pruning unpromising candidates, which increases computational overhead.
(Vartak et al., 2016)	COUNT, SUM, MIN, MAX, AVG, UDA	✓			✗	
(Albarrak and Sharaf, 2017)	COUNT, SUM, MIN, MAX, AVG	✓	Constants of selection predicates		✗	Relies on monotonic aggregate constraints.
(Kadlag et al., 2004)	COUNT	✗			✗	Only simple group COUNT constraints, no similarity maintained to the original query.
(Shetiya et al., 2022)	COUNT	✓		Result Groups	✗	Only simple group COUNT constraints, assumes monotonicity.
(Li et al., 2023)	COUNT	✓			✗	Only simple group COUNT constraints, relies on monotonic pruning, not valid for non-monotonic aggregates.

(Campbell et al., 2024)	COUNT	✓		Top-k Results	✗	Solves a different problem (diversity in Top-k ranking).

Kadlag et al. (2004) introduced the SAUNA framework to assist users in formulating exploratory queries that satisfy cardinality constraints. Starting from an initial query and a desired result size, SAUNA automatically adjusts predicate ranges while preserving the relative proportions between them which referred to as the aspect ratio. The system uses a distance function tailored to range queries, aiming to minimize the L_2-norm between the original and refined queries in the predicate space. This approach helps maintain the interpretability of results by ensuring they remain close to the user's initial intent. SAUNA employs multi-dimensional histograms for cardinality estimation and uses a binary search strategy to efficiently navigate predicate space. However, the method focuses solely on satisfying only cardinality (COUNT) constraints and does not consider semantic similarity beyond predicate alignment, limiting its effectiveness for more nuanced refinement goals.

Mishra et al. (2008) expanded this line of work by generalizing the problem to more complex scenarios. They formalized the Targeted Query Generation (TQG) problem, aiming to simultaneously generate or refine queries that satisfy multiple cardinality constraints. The authors proposed Targeted Query Generation (TQGen), a practical algorithm that generates queries whose outputs approximately match target multiple cardinality constraints. TQGen minimizes the sum of squared logarithmic relative errors between the observed and desired cardinalities across multiple predicates. The algorithm operates in two phases: a bounding phase, which uses a binary search over the domain of each predicate to determine lower and upper bounds on the search space, and an exploration phase, which navigates the bounded search space to identify queries that minimize the sum of squared logarithmic relative errors across all cardinality constraints. In the bounding phase, a relaxed version of each predicate is evaluated to generate an upper-bound query that overshoots the specified cardinalities. During exploration, the algorithm partitions the bounded space into a grid, scores each cell based on potential improvement, and recursively explores the most promising cells to find a query that best approximates the constraints. A key efficiency factor in TQGen uses a monotonic error function, which allows for the early pruning of suboptimal subspaces. Despite its scalability and optimization capabilities, TQGen does not incorporate similarity to the original query and assumes monotonic cardinality constraints, which limits its applicability in scenarios considered in this work that involve non-monotonic constraints.

Mishra and Koudas (2009) introduced the Stretch 'n' Shrink (SnS) framework, which extends earlier approaches such as the Hill-Climbing (HC) method (Bruno et al., 2006) and the Targeted Query Generation (TQGen) algorithm (Mishra et al., 2008) by incorporating explicit user interaction into the query refinement process. Unlike previous methods that rely on fixed scoring functions to guide optimization, SnS enables users to interactively refine selection predicates by either relaxing or tightening them, offering a more adaptive refinement process aligned with user intent. SnS operates through an iterative feedback loop where users guide the refinement of query predicates over selected attributes. To ensure that this interaction remains efficient, especially over large and complex datasets, SnS uses sampling-based techniques for estimating cardinalities and leverages advanced indexing structures such as multidimensional indexes and binary search strategies to identify appropriate predicate ranges quickly. The system supports both numeric attributes and categorical attributes organized in hierarchies. The framework's optimization strategy is focused exclusively on satisfying cardinality constraints. It does not incorporate mechanisms for preserving the similarity between the refined and original queries. This limitation makes it less suitable in scenarios where the user's intent includes both cardinality and similarity objectives. Furthermore, SnS assumes monotonic behaviour in predicate adjustments, which restricts its ability to handle more complex constraints such as non-monotonic aggregates. Addressing these challenges remains an open research direction.

Vartak et al. (2010) introduced a novel framework, QRelX, designed to address the limitations of earlier query refinement approaches by automatically generating refined queries that provide cardinality assurance while maintaining semantic similarity to the original query. QRelX constructs a structured refinement search space and employs a proximity based search strategy, prioritizing candidate queries that exhibit minimal semantic deviation from the original. To reduce redundant computations, it uses an incremental cardinality estimation method that reuses results from previously evaluated, similar queries. Importantly, QRelX supports the refinement of both selection and join predicates. Building on this foundation, Vartak et al. (2016) proposed the ACQUIRE framework, which generalizes query refinement to support Aggregation Constrained Queries (ACQs). ACQUIRE extends the refinement process to handle a broader class of constraints involving common SQL aggregation functions such as COUNT, SUM, and AVG. The framework explores a discretized refinement space by evaluating candidate queries in order of their semantic similarity to the original query. Each candidate is checked against a user-defined aggregate threshold and is considered a valid solution if the constraint is met. Rather than evaluating each candidate from scratch, ACQUIRE incrementally computes aggregate values by leveraging the algebraic properties of common SQL aggregates such as COUNT, SUM, MAX, and AVG. This reuse of previously computed results reduces redundant computation.

Despite their effectiveness, both QRelX and ACQUIRE share important limitations. They require users to manually set thresholds for acceptable deviations in cardinality and similarity parameters that may be unintuitive for non-expert users. Moreover, both frameworks evaluate promising and unpromising candidate queries during the refinement process, which can lead to increased computational overhead, particularly on large datasets. Furthermore, while ACQUIRE introduces support for a broader set of aggregate functions, it relies on the monotonicity and additive properties of these functions. This reliance restricts its applicability in scenarios involving non-monotonic aggregate constraints, such as those addressed in this work, and as such, extending these frameworks to effectively handle non-monotonic aggregates while maintaining query similarity remains a non-trivial challenge.

Albarrak and Sharaf (2017) introduced EAGER, which addresses the limitations of earlier refinement techniques by incorporating similarity-awareness. EAGER aims to balance user-specified aggregate constraints and query similarity to maximize overall user satisfaction. Notably, this work extends previous approaches by generalizing the aggregate constraints beyond cardinality to include other standard SQL aggregation functions such as COUNT, SUM, AVG, MIN, and MAX. It implements similarity-based and aggregate-based pruning techniques to efficiently navigate the large space of potential refinements and significantly reduces computational costs. EAGER also employs a hierarchical representation of the query space, enabling tighter aggregate bound estimations and faster identification of optimal queries. Additionally, it includes optimization and approximation strategies such as query materialization and selective exploration of promising regions in the search space to enhance efficiency. A key feature of EAGER is its reliance on the monotonicity property of aggregate functions to establish bounds and prune the search space. However, while this is effective for many common aggregate types, it limits the applicability of EAGER in contexts where the aggregate constraints are non-monotonic, as is the case in the problem addressed in this work.

To summarize, despite the efficient contributions of the reviewed techniques for refining queries to satisfy aggregate constraints, several limitations render them unsuitable for addressing the problem tackled in this work. First, many approaches including HC (Bruno et al., 2006), TQGen (Mishra et al., 2008), SnS (Mishra and Koudas, 2009), and SAUNA (Kadlag et al., 2004) are restricted to cardinality (COUNT) constraints over the entire result set and do not support more expressive aggregate functions or arithmetic combinations of aggregates. Second, several techniques, such as HC, TQGen, SnS, and SAUNA, do not incorporate semantic proximity to the original query. Instead, they rely on local error metrics or geometric similarity, which may not fully capture user intent. Although more recent frameworks like QRelX (Vartak et al., 2010), ACQUIRE (Vartak et al., 2016), and EAGER (Albarrak and Sharaf, 2017) incorporate similarity-aware strategies, they rely on the monotonicity of aggregate functions which is a property that fails in scenarios involving non-monotonic constraints. Third, methods such as QRelX and ACQUIRE suffer from computational inefficiencies, often due to exhaustive evaluation of candidate queries, reliance on fixed step sizes, or approximation via sampling, which can result in suboptimal or inaccurate refinements. Finally, many of these techniques require manual tuning of parameters (e.g., aggregate error thresholds, similarity bounds), which can be impractical in exploratory scenarios where users lack detailed knowledge of the data or constraints (Vartak et al., 2010, 2016). These limitations collectively underscore why existing approaches cannot be directly extended to solve the problem in this thesis, which demands efficient, similarity-aware refinement under non-monotonic aggregate constraints.

2.1.2.2.2 Query Refinement for Aggregate Constraints over Groups of the Result Recent research has increasingly recognized the importance of refining database queries to meet not just global result size constraints but also group-level aggregate constraints, such as constraints for fairness and diversity. However, only a few recent studies have explicitly tackled this problem, each with distinct goals and methodologies (Shetiya et al., 2022; Li et al., 2023; Campbell et al., 2024). The work presented in this thesis is positioned within this emerging line of research, focusing on refining queries to satisfy constraints within result groups.

One of the earliest efforts in this area is the framework introduced by Shetiya et al. (2022), which marks a first step toward incorporating fairness constraints into database query processing and data management systems. It focuses on refining queries to satisfy cardinality constraints over protected groups (e.g., gender, ethnicity) in the query result. Given a user's input query, the system selectively relaxes predicates to ensure that the refined query satisfies group-based constraints, expressed as minimum or maximum thresholds on representing protected groups in the output.

The framework supports a broad class of selection queries involving both categorical and numeric attributes. It enables fairness-aware refinements by adjusting predicate values to improve group representation, while enforcing a similarity constraint based on the Jaccard index to ensure that the refined result set remains close to the original. The refinement process is formalized as a search problem, where the system uses a best-first traversal of the space of candidate queries. Candidate queries are prioritized using an upper-bound estimate on the fairness objective, allowing the system to guide the search toward promising refinements. To improve computational efficiency, the framework prioritizes candidate refinements using upper-bound fairness estimates and leverages monotonicity to prune unpromising search paths. Also, it avoids redundant evaluations by reusing partial computations and guiding the search using fairness-based scoring functions that estimate how well a candidate query is likely to satisfy the fairness constraints However, the framework is primarily designed for simple group cardinality (**COUNT**) constraints over a single binary protected attribute (e.g., gender), and is limited to relaxing selection predicates involving straightforward numeric ranges or Boolean attributes. In addition, it supports only single-relation queries and does not extend to queries over joined tables. The framework does not accommodate more complex aggregate expressions (e.g., **SUM**, **AVG**) or arithmetic combinations of multiple aggregates, and its pruning strategies rely heavily on monotonicity, which does not generalize to non-monotonic constraints.

Li et al. (2023) presents a comprehensive approach to the problem of query refinement that aims to satisfy cardinality constraints on specific demographic groups within query results by modifying the constants in the selection predicates of the user query. For numerical attributes, refinements involve adjusting the constant values used in comparisons. For categorical attributes, refinements may take the form of set membership predicates, such as $A \in \{c_1, \ldots, c_n\}$, where each c_i is a value drawn from the domain of the attribute A. Their method specifically supports a conjunction of so-called *group cardinality constraints*. A single group cardinality constraint compares the result of a **Count** query with a conjunctive selection condition against a constant. Their work addresses an important need in data selection processes where diversity and group representation are required. It enables users to impose constraints on the sizes of various subgroups in the result set. Compared to the work by Shetiya et al. (2022), this proposed work by Li et al. (2023) supports multiple, non-binary sensitive attributes, handles both relaxation and contraction constraints, and extends to SPJ queries over multiple joined tables. It also adopts a different similarity objective by measuring distance between queries rather than between result sets. The authors address the problem of finding minimal refinements of Select-Project-Join (SPJ) queries such that the resulting output satisfies constraints on the sizes of multiple subgroups within the query result. A refinement is deemed minimal if it fulfills all constraints while preserving the user's original intent as closely as possible. This intent preservation is quantified by measuring the similarity of constants used in the selection predicates between the original and refined queries. The refinement problem is formulated as skyline queries. Specifically, the objective is to identify all minimal refinements such that no other refinement is closer to the user query in at least one dimension (predicate) and at least as close in all dimensions.

The methodology consists of three primary components. First, the authors develop a provenance-based approach that annotates tuples with relevant predicate information. This model translates cardinality constraints into algebraic expressions for constraint testing. They use the provenance model to eliminate the need to repeatedly execute candidate query refinements against the database, which significantly improves efficiency. Second, the authors introduce a data structure called Possible Value Lists (PVL) that encompasses all possible predicate values organized by their distance from the original values. For numerical predicates, values are sorted by their absolute distance from the original constant, while categorical predicates are represented using multiple lists indicating the presence or absence of specific values. Third, the authors employ a recursive search process that begins with a partial query (a single predicate) and uses a top-down traversal of the PVL to find a minimal refinement. Once a refinement is found, the result set is updated, and new refinement candidates are generated by updating predicate values using entries from the PVL. The process then recursively searches for additional minimal refinements. Several optimizations enhance the efficiency of their proposed solution. For cases where all constraints are relaxation constraints (e.g., requiring at least a certain number of individuals from a group) or all are contraction constraints (e.g., requiring at most a specific number), the authors leverage the monotonicity property of such constraints to avoid exhaustive traversal when finding refinements. The algorithm also avoids exploring refinements that cannot satisfy certain constraints based on partial queries. If a partial query fails to satisfy specific relaxation constraints, any complete refinement containing those predicates will also fail. Additionally, the algorithm exploits the pre-computed aggregations if they have been examined in previous iterations, preventing redundant evaluation of the same refinements. Although the approach of Li et al. (2023) effectively satisfies cardinality constraints on specific demographic groups within query results, it cannot be directly applied to the setting in this thesis, as it relies on monotonicity to prune the search space. In contrast, the problem in this thesis involves non-monotonic aggregate constraints, where such pruning strategies are invalid.

Campbell et al. (2024) tackles a related but distinct problem by focusing on query refinement for diverse top-k selection. While the works of Shetiya et al. (2022) and Li et al. (2023) aim to satisfy cardinality constraints over protected groups, Campbell et al. (2024) addresses diversity from a ranking perspective, specifically within the top-k results of an `ORDER BY` query. This is particularly relevant for decision-making domains such as hiring or school admissions, where it is critical to ensure that the highest ranked candidates reflect desired diversity criteria. Their approach formulates the refinement task as a Mixed-Integer Linear Program (MILP), which seeks to adjust query selection predicates so that the top-k output satisfies user defined diversity constraints while remaining as close as possible to the original query's intent. Like Li et al. (2023), they adopt provenance-based annotations to represent candidate refinements efficiently, but extend them to handle ranking semantics. To enhance computational efficiency, the authors introduce several optimizations. First, a relevancy-based optimization eliminates tuples that cannot appear in any valid top-k result under any refinement. Second, a lineage-based optimization merges MILP variables for tuples with identical provenance. Third, a score relaxation technique simplifies the objective function for tuples that belong to groups constrained by only a lower-bound or only an upper-bound (but not both) to reduce the complexity of the MILP formulation. Although this work is effective for improving diversity in top-k rankings, it is fundamentally different in scope from the work by Shetiya et al. (2022) and Li et al. (2023). It is restricted to queries with `ORDER BY` clauses and top-k semantics, and does not generalize to arbitrary aggregate constraints or arithmetic expressions over group-level aggregates, which are central to the setting of this thesis.

Despite their contributions, these recent works are not directly applicable to the setting in this work. The framework by Shetiya et al. (2022); Li et al. (2023) supports only simple group cardinality (COUNT) constraints and relies on monotonicity assumptions for efficient search, which makes it unsuitable for non-monotonic aggregate constraints. Finally, the method proposed by Campbell et al. (2024) specifically targets diversity constraints within top-k results under ranking-based queries and addresses a different problem than the one considered in this work.

2.2 Summary

This chapter presents a comprehensive survey of existing work in the areas of data refinement (DR) and query refinement (QR), with particular emphasis on the latter due to its central role in this thesis. While both DR and QR techniques aim to align query outputs with user expected constraints, they differ fundamentally in scope: DR methods modify the underlying data, whereas QR strategies refine the query itself. This thesis focuses on scenarios in which the query must be modified in order to satisfy user-specified output constraints, particularly those involving aggregate expressions, making QR techniques the primary subject of interest.

Prior research in data refinement has addressed how-to queries (Meliou and Suciu, 2012), missing answer explanations (Herschel and Hernández, 2010), and error tracing in update logs (Wang et al., 2017). These approaches formulate data-level changes using optimization frameworks such as Mixed Integer Programming, but they do not generalize to query-level refinements or aggregate constraints, particularly in settings where modifying the data is infeasible or undesirable.

In the space of query refinement, existing work largely falls into two categories: (1) refinement for non-aggregate constraints, such as including/excluding specific tuples or controlling result size (Tran and Chan, 2010; Islam et al., 2013a; Mottin et al., 2014b; Muslea, 2004; Mottin et al., 2016b), and (2) refinement for aggregate constraints, typically involving functions like `COUNT` or `SUM` over the entire result set (Mishra et al., 2008; Vartak et al., 2016; Albarrak and Sharaf, 2017). While several of these approaches such as ACQUIRE (Vartak et al., 2016) and EAGER (Albarrak and Sharaf, 2017) incorporate query similarity and pruning techniques based on monotonicity, they are not suitable for non-monotonic arithmetic expressions over aggregate constraints. Many rely on cardinality estimations, require iterative user feedback, or support only limited forms of aggregation.

More recently, query refinement under group-level aggregate constraints has been explored in fairness and diversity settings (Shetiya et al., 2022; Li et al., 2023; Campbell et al., 2024). These frameworks demonstrate the feasibility of refining queries to satisfy constraints over protected groups. However, they are typically limited to `COUNT`-based constraints, rely on monotonic behaviour for pruning the search space, or are designed specifically for top-k ranking queries.

In summary, current QR techniques do not adequately support the kinds of non-monotonic arithmetic expressions over aggregate constraints considered in this thesis. This gap motivates the development of a new query refinement framework capable of: (1) handling arithmetic expressions over standard aggregates; (2) supporting non-monotonic constraints; and (3) preserving similarity to the user's original query. The remainder of this thesis builds on this foundation to introduce and evaluate such a framework.

Chapter 3

Methodology

This work is the first to propose a query repair approach for handling non-monotone constraints, marking a novel direction in the field of query repair. Accordingly, the primary objective is to develop effective approaches to tackle this problem.

The research questions guiding this thesis are:

- **RQ1:** How can the shape of an instance of a dataset be leveraged to develop a more efficient approach for query repair under non-monotone aggregate constraints?

  Objective: To reduce per-tuple predicate evaluations and eliminate redundant aggregation computations.
  Contribution: A concise description of the contribution to this question appears in Sections 3.1, 3.2, and 3.3; a more detailed description is provided in Chapter 4 and Chapter 5.

- **RQ2:** What pruning strategies can be designed to reduce the search space of candidate repairs in query repair under non-monotone aggregate constraints?

  Objective: Minimize exhaustive evaluation of every repair candidate by early elimination of repair candidate sets that cannot satisfy the aggregate constraint.
  Contribution: A concise description of the contribution to this question appears in Section 3.4; a more detailed description is provided in Chapter 6.

This dissertation adopts the classic system development methodology of (Nunamaker Jr et al., 1990): defining the problem (Section 3.1), developing, designing, and building repair algorithms (Sections 3.2–3.4), and empirically evaluating them (Section 3.5). A review of existing query repair techniques reveals a common pattern that aligns with this approach: first formalize the repair problem, then design and build algorithms, and finally evaluate them. In this dissertation, that same sequence is followed: problem definition, prototype implementation, and evaluation, treating the proposed algorithms as a prototype system rather than a production-grade tool.

3.1 Formalizing the Problem

This thesis begins by formalizing the aggregate constraint repair problem, which aims to modify the constants of the selection conditions of a user query similar to (Kalashnikov et al., 2018; Mishra and Koudas, 2009), so that its result satisfies a user-defined aggregate constraint. The focus is on a class of Select-Project-Join (SPJ) queries where the selection predicate is a conjunction of attribute comparisons. Aggregate constraints are defined as comparisons between a threshold and an arithmetic expression over the results of aggregation queries. A distinction is made between monotone and non-monotone constraints, noting that non-monotonicity poses significant computational challenges. Many existing query refinement and relaxation techniques rely on monotonicity to optimize the search process by pruning unpromising candidates without exhaustively exploring the entire repair space of repair candidates (Wu and Madden, 2013; Bruno et al., 2006; Li et al., 2023). However, these optimizations are not applicable to the problem in this work, as general non-monotone constraints are considered that cannot benefit from such assumptions. A definition of a distance metric over query predicates is introduced to rank repair candidates in order to preserve the original semantics of the user's query as much as possible. The repair problem is formulated as finding the top-k closest query repairs that satisfy the constraint. It is shown that the search space of repair candidates grows exponentially with the number of query predicates, and that the query repair problem considered in this work is NP-hard in general (Bruno et al., 2006), highlighting the need for efficient algorithms. Further details about the problem definition are provided in Chapter 4.

3.2 Creating and Implementing the Brute Force (BF) Approach

To establish a reliable baseline for evaluating the proposed query repair algorithms, a comprehensive Brute Force (BF) approach is implemented. This exhaustive method explores the entire search space of possible query repairs, sorted according to the distance metric defined in Section 4.4, by enumerating all possible constant modifications for selection predicates. Each candidate repair is then evaluated by applying the modified predicates to every tuple in the complete dataset (per-tuple evaluation) to determine the set of satisfying tuples. Then, computing the required aggregate values over those tuples and substituting those values into the arithmetic expression of the given constraint. After that, checking whether the resulting expression satisfies the constraint. Valid repairs are collected until k satisfying repair candidates have been found, at which point the process terminates.

The implementation of this Brute Force (BF) algorithm serves a critical purpose in the research methodology by providing a complete enumeration of all possible repairs within the defined search space, thereby establishing ground truth for evaluating the performance of more efficient approaches similar to (Albarrak, 2018; Li et al., 2023). While the BF algorithm is valuable for validation, it is not practical for real-world scenarios: the number of repair candidate queries grows exponentially with both the number of selection predicates and the range of possible constant values, making exhaustive evaluation computationally infeasible. In particular, evaluating each refinement individually by applying its selection condition over all tuples incurs significant runtime overhead. To overcome these challenges, the Full Cluster Filtering (FF) strategy is proposed, substantially reducing the search space by leveraging data partitioning and reusing precomputed aggregates.

3.3 Creating and Implementing the Approach that Partitions the Input Data

A key insight behind the Full Cluster Filtering (FF) approach is the observation that many candidate repairs are structurally similar, which differ only slightly in the constants used in selection predicates, and therefore often operate over overlapping subsets of the data. This redundancy suggests that aggregation constraints can be evaluated more efficiently by reasoning over clusters of data rather than individual tuples. To exploit this, FF organizes the input dataset into hierarchical partitions. This grouping avoids the redundant tuple-by-tuple evaluations required by Brute Force (BF) approach.

Full Cluster Filtering (FF) begins by partitioning the input dataset into disjoint clusters based on the attributes appearing in the original selection predicates. For each cluster, it materializes both the minimum and maximum values of each predicate attribute and the results of all aggregate functions required by the user-defined constraint. When a repair candidate is generated by modifying the constants in the selection predicates, Full Cluster Filtering (FF) first performs a per-cluster predicate evaluation by comparing the repaired predicate against each cluster's attribute bounds to determine which clusters may contain satisfying tuples. For those qualifying clusters, the algorithm retrieves the precomputed aggregate values and combines them across clusters to obtain the candidate's overall aggregate results. These aggregated values are then substituted into the arithmetic expression defined by the constraint. If the expression is satisfied, the repair candidate is added to the set of valid repairs. The process repeats by generating and evaluating candidates until the top-k repairs (ranked by distance) have been identified, at which point the algorithm terminates.

The main advantage of this algorithm over the brute force approach is that it can reuse the aggregate query results materialized for a cluster if all tuples in the cluster fulfil the condition of the repair candidate and can skip any clusters that do not contain any tuples fulfilling the conditions. Further details about Full Cluster Filtering (FF) approach are provided in Chapter 5.

3.4 Creating and Implementing the Approach that Reduces the Search Space of Possible Candidates

While algorithm Full Cluster Filtering (FF) reduces the effort needed to evaluate aggregation constraints for repair candidates, it has the drawback that still have to evaluate each repair candidate individually. A key insight behind the Cluster Range Pruning (RP) approach is that many repair candidates differ only slightly and can be grouped into ranges that behave similarly with respect to the constraint. This observation enables reasoning over sets of candidates rather than individual ones. By structuring the search space as intervals and computing bounds over those intervals, RP can prune or accept entire candidate ranges without explicitly evaluating each one.

Cluster Range Pruning (RP) extends the Full Cluster Filtering (FF) approach by reasoning about intervals of candidate repairs rather than evaluating them individually. The algorithm begins by representing the search space as a collection of intervals over the constants in the selection predicates, where each interval encompasses multiple similar repair candidates. These intervals are sorted based on their similarity to the original user query. For each candidate interval, the algorithm leverages the attribute bounds stored in each cluster of the hierarchical partitions to determine whether all, none, or some tuples in that cluster satisfy the predicate conditions. These clusters are categorized accordingly: full-cover clusters include their materialized aggregate results directly, no-cover clusters are skipped, and partial-cover clusters require recursive exploration of their sub-clusters (if any) or are handled as is if they are leaf nodes. Once the relevant clusters for an interval are identified, the algorithm merges aggregate results from full and partial clusters to compute bounds on each aggregate function. These are substituted into the arithmetic constraint expression to derive an overall bound, allowing the algorithm to decide whether all, some, or none of the repair candidates in the interval satisfy the constraint. If the interval is guaranteed to satisfy the constraint, it is accepted in full; if it is guaranteed to fail, it is pruned entirely; otherwise, it is recursively split into smaller intervals for finer-grained evaluation. Candidate intervals are processed in order of distance from

Table 3.1: 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 3.2: Constraints for Experimentation

ID	Constraint
ω_1	$\frac{\textbf{count}(\text{race} = 1 \wedge \text{label} = 1)}{\textbf{count}(\text{race} = 1)} - \frac{\textbf{count}(\text{race} = 2 \wedge \text{label} = 1)}{\textbf{count}(\text{race} = 2)} \in [B_l, B_u]$
ω_2	$\frac{\textbf{count}(\text{ageGroup} = 1 \wedge \text{label} = 1)}{\textbf{count}(\text{ageGroup} = 1)} - \frac{\textbf{count}(\text{ageGroup} = 2 \wedge \text{label} = 1)}{\textbf{count}(\text{ageGroup} = 2)} \in [B_l, B_u]$
ω_3	$\frac{\textbf{count}(\text{sex} = 1 \wedge \text{PINCP} \geq 20k)}{\textbf{count}(\text{sex} = 1)} - \frac{\textbf{count}(\text{sex} = 2 \wedge \text{PINCP} \geq 20k)}{\textbf{count}(\text{sex} = 2)} \in [B_l, B_u]$
ω_4	$\frac{\textbf{count}(\text{race} = 1 \wedge \text{PINCP} \geq 15k)}{\textbf{count}(\text{race} = 1)} - \frac{\textbf{count}(\text{race} = 2 \wedge \text{PINCP} \geq 15k)}{\textbf{count}(\text{race} = 2)} \in [B_l, B_u]$
ω_5	$\frac{\sum Revenue_{\text{ProductsSelectedFromUK}}}{\sum Revenue_{\text{SelectedProducts}}} \in [B_l, B_u]$
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	$\omega_{81} := \textbf{count}(\text{Sex} = \text{Female}) \leq B_{u_1}$ $\omega_{82} := \textbf{count}(\text{Race} = \text{Black}) \leq B_{u_2}$ $\omega_{83} := \textbf{count}(\text{Marital} = \text{Divorced}) \leq B_{u_3}$

the original query, and satisfying candidates are extracted until k valid repairs are identified, at which point the algorithm terminates.

The advantage of this approach is that it often enables to prune sets of repair candidates or confirm all of them to be repairs without individually evaluating them. Further details about Cluster Range Pruning (RP) approach are provided in Chapter 6.

3.5 Experimental Setup

This section presents the experimental setup and evaluation methodology for the proposed query repair approaches. Performance is assessed across different datasets, queries, and constraints. The selected datasets, queries and constraints are chosen to be representative of the types of queries and constraints that arise in real-world senarios of query repair. The description begins with details on datasets, queries, constraints, metrics, and parameter settings, followed by an outline of the evaluation methodology.

To ensure a comprehensive evaluation, two real-world datasets and the TPC-H benchmark dataset are used. A set of queries and constraints are defined, and key experimental parameters are carefully configured. This setup enables a study of algorithms behaviour under varied data, queries, and constraints.

Datasets. Two real-world datasets, *ACSIncome* (Friedler et al., 2019) and *Healthcare* (Grafberger et al., 2021), each of size 50 K, are chosen because they are widely used in the fairness literature and provide realistic, heterogeneous data for evaluating fairness-aware query repair.

The ACSIncome dataset (Adult Census Income) is a dataset that comprises data about individuals from the 1994 United States Census (Friedler et al., 2019). It contains 14 attributes describing demographic and employment characteristics such as age, gender, race, education, working hours, occupation, and income level. This dataset is commonly used to assess bias in socioeconomic decision-making and serves as a representative example of fairness critical applications where selection conditions can reflect real hiring or income-related disparities.

The Healthcare dataset simulates medical decision-making scenarios with features such as income, number of children, county, and health complications, allowing the evaluation of fairness and robustness in prescreening queries applied to healthcare contexts. These two datasets collectively capture distinct fairness domains, socioeconomic decision making and healthcare screening, making them suitable and representative benchmarks for evaluating fairness query repair.

Additionally, the standard benchmark *TPC-H*[1] is used with dataset sizes varying from 25 K to 500 K. Unlike the fairness-focused datasets, TPC-H models business and supply-chain data involving parts, suppliers, and nations, and is used to test aggregate constraints (e.g., revenue bounds) rather than demographic fairness. This dataset is included to demonstrate that the proposed query repair is not limited to fairness scenarios but can be generalised to other domains, such as business analytics and

[1] https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp

supply-chain optimisation. All categorical columns are converted to numeric values, as the algorithms are designed for numerical data.

Queries. Table 3.1 shows the queries used in the experiments. For Healthcare, queries Q_1 and Q_2 from (Li et al., 2023) are included, along with a newly defined query Q_3. For ACSIncome, query Q_4 from (Li et al., 2023) is included, along with newly defined queries Q_5 and Q_6. Query Q_7 is generated with three predicates inspired by TPC-H's Q_2.

Each query represents a different real-world scenario to evaluate how fairness query repair behaves under various selection conditions. For the Healthcare dataset, Q_1–Q_3 simulate a medical screening process that filters patients based on income, number of children, complications, and county. The goal is to test whether fairness constraints can correct potential biases that arise when selection criteria favour certain demographic groups (e.g., race or age group). For the ACSIncome dataset, Q_4–Q_6 emulate employment-related filters using census data, selecting individuals based on working hours, educational attainment, and class of worker. These queries allow us to study how fairness constraints affect inclusion of under-represented socioeconomic groups. Finally, the TPC-H-based query Q_7 models a supply-chain selection scenario that joins the *part*, *supplier*, *partsupp*, *nation*, and *region* tables to retrieve products meeting certain conditions such as part size, type, and supplier region (e.g., ‘LARGE BRUSHED‘ parts supplied from ‘EUROPE‘). This query reflects a typical industrial use case where the objective is to control or limit the proportion of revenue sourced from specific regions, thereby minimizing the impact of supply chain disruptions.

Constraints. For Healthcare and ACSIncome, the SPD between two demographic groups is enforced to be within a certain bound. Table 3.2 shows the details of the constraints used. In some experiments, the lower and upper bounds B_l and B_u of a constraint are varied to control where in the search space the valid repairs appear, ranging from those close to the original query to those farther away. To denote a specific variant of a constraint ω_i used in these experiments, the notation $\omega_i^{d=p}$ is used, where the superscript indicates that the bounds have been set such that the top-k repairs appear within the first p% of all repair candidates ordered by their distance from the original user query. In other words, an algorithm that explores the individual repair candidates in this order would have to examine the first p% of the candidate search space to find the top-k repairs. For ACSIncome, the groups for SPD are determined based on gender and race. For Healthcare, demographic groups are determined based on race and age group. For TPC-H, the constraint ω_5 is enforced as described in Example 1.2 to minimize the impact of supply change disruption, where the company wants only a certain amount of expected revenue to be from countries with import/export issues. Ω is used to denote a set of aggregate constraints. Ω_6 through Ω_8 are sets of cardinality constraints for comparison with Erica. While repair methods are presented for single aggregate constraint, the methods can be trivially extended to find repairs for a set of constraints, i.e., the repair fulfils $\bigwedge_{\omega \in \Omega} \omega$.

Each constraint type corresponds to a specific fairness or balance requirement applied to the query results. Constraints ω_1–ω_4 enforce SPD bounds that ensure differences in positive outcomes (e.g., selected records) between demographic groups such as race, gender, or age remain within the tolerance interval $[B_l, B_u]$. Constraint ω_5 imposes a business-oriented aggregate condition that limits the proportion of revenue attributed to suppliers from certain regions, ensuring diversity in supply sources. Constraints Ω_6–Ω_8 define sets of cardinality constraints that bound the number of tuples associated with specific demographic attributes (e.g., sex, race, marital status), providing a way to compare fairness repair results with systems like Erica.

Metrics. The following measures are used:

- **Execution Time (s):** Total time to produce the top-k repairs.
- **Candidates Evaluated (NCE):** This metric counts how many repair candidates the algorithm checks before finding the top-k valid ones. A repair candidate is a modified version of the original query where one or more of the selection conditions have been modified. To evaluate a candidate, the algorithm runs the modified query, calculates the relevant aggregate values, and checks whether the result satisfies the user-defined constraint. If it does, the candidate is accepted as a valid repair; if not, it is discarded.
- **Clusters Accessed (NCA):** This metric measures how many clusters of data the algorithm needs to access in order to find the top-k repairs. This metric provides insight into the level of data access required by each approach. In Brute Force (BF), every candidate is evaluated by scanning each tuple in the dataset individually (the NCA counts the number of tuples accesses). In contrast, Full Cluster Filtering (FF) leverages precomputed aggregates over data clusters to evaluate candidate repairs at the cluster level. Cluster Range Pruning (RP) further optimizes this by evaluating entire intervals of candidate repairs to reuse the same clusters for multiple candidates or prune them altogether.

Parameters. There are three key tuning parameters that could impact the performance of the proposed methods. Recall that the kd-tree is used to perform the clustering as described in Section 5.1. Two tuning parameters are used 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. Nodes with less than or equal to $\mathcal{S}$ tuples are not split. When one of the proposed algorithms reaches such a leaf node, the computations on each tuple in the cluster are just evaluated, e.g., to determine which tuples fulfill a condition.

Also, k is controlled, which is the number of repairs returned by the proposed methods. The default setting for these parameters is as follows: $\mathcal{B} = 5$, $k = 7$, and $\mathcal{S} = 15$. The default dataset size is 50K tuples. The choice of $\mathcal{B}$ and $\mathcal{S}$ depend on the data and query. Intuitively, setting $\mathcal{S}$ too small results in a deep tree with many leaf clusters which will increasing overhead and setting it too large will reduces pruning effectiveness due to overly large clusters. Similarly, a very large $\mathcal{B}$ scatters data across many children, limiting subtree pruning, while a very small value leads to deep trees with limited branching. The values chosen for the default settings represent moderate configurations suitable for a wide range of datasets.

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

3.6 Experiments

This section is presented to understand **RQ1** and **RQ2**. It begins by comparing the performance of the proposed methods, Full Cluster Filtering (FF) and Cluster Range Pruning (RP), against the exhaustive Brute Force (BF) Section 3.7 as well as against each other Section 3.8. It then examines the impact of key parameters such as data size, clustering structure (branching factor and bucket size), exploration distance, and the top-k value on the performance of the proposed repair methods Section 3.9. Finally, it compares the proposed approaches with the most relevant prior technique, Erica (Li et al., 2023).

3.7 Comparison with Brute Force (BF)

This evaluation provides a concrete evaluation of how much computational efficiency is gained by moving from a Brute Force (BF) approach to more optimized repair strategies. Since BF explores the entire search space exhaustively, it offers a reliable ground truth for validating the proposed methods. Also, it helps to quantify the benefits of data partitioning and range-based pruning in optimizing query repair under non-monotonic constraints.

Section 7.1 first compares the proposed methods, Full Cluster Filtering (FF) and Cluster Range Pruning (RP), with the exhaustive Brute Force (BF) method using the Healthcare, queries Q_1 and Q_2, the constraint ω_1 and default settings in Section 3.5. The full results and detailed discussion appear in Section 7.1.

3.8 Performance of Full Cluster Filtering (FF) and Cluster Range Pruning (RP)

This section presents experiments measuring the performance of Full Cluster Filtering (FF) and Cluster Range Pruning (RP) to assess how the optimized algorithm Cluster Range Pruning (RP) effectively reduce computational overhead by reasoning over repair candidate intervals. It uses the Healthcare and ACSIncome datasets with queries in Table 3.1, constraints in Table 3.2, and default settings in Section 3.5. For the Healthcare, the constraints ω_1 and ω_2 are used while ω_3 and ω_4 are considered for the ACSIncome. In addition to runtime, number of candidates evaluated (NCE) is also measured which is the total of number of repair candidates for which the aggregate constraint is evaluated, and number of clusters accessed (NCA) which is the total number of clusters accessed by an algorithm. The full results and detailed discussion appear in Section 7.1.

3.9 Performance-Impacting Factors

To gain deeper insights into the behaviour observed in Section 7.1, Section 7.2 investigates the relationship between the exploration distance (ED) and performance. In addition, performance for Full Cluster Filtering (FF) and Cluster Range Pruning (RP) is evaluated with respect to the parameters defined in Section 3.5 (Bucket Size, Branching Factor, and Top-k) using the Healthcare, ACSIncome, and TPC-H datasets. Section 7.2 examines the impact of varying these key parameters to understand how sensitive the proposed algorithms are to different configuration settings and to identify performance trade-offs.

3.9.1 Exploration distance ED:

The intuition behind this experiment is to assess the effectiveness of the proposed algorithms under varying exploration distances. Specifically, it evaluates how well the methods perform when the top-k repairs are located near the beginning of the search space (i.e., highly similar to the original query) versus when they are distributed further away (i.e., less similar). This distinction is important because Cluster Range Pruning (RP), which reasons about intervals of repair candidates, may incur additional overhead when top-k repairs are concentrated near the start, requiring recursive subdivision of broad candidate ranges. In contrast, it is expected to perform more efficiently when the top-k are located deeper in the search space, as its candidate interval reasoning can prune large ranges of irrelevant candidates. In contrast, Full

Cluster Filtering (FF) evaluates candidates individually but can access promising repairs directly when repairs are near the original query.

Exploration distance (ED) is the fraction of the search space that must be explored before locating the top-k repairs. Queries Q_1–Q_3 and the constraint ω_1 are used on Healthcare, while queries Q_4–Q_6 and the constraint ω_3 are used on ACSIncome, with bounds varied to control the ED. The full results and detailed discussion appear in Section 7.2.

3.9.2 Bucket Size $\mathcal{S}$:

This experiment is varying the bucket size $\mathcal{S}$. The reasoning behind this is to examine how the granularity of data grouping affects the performance of the proposed methods. A smaller $\mathcal{S}$ results in finer-grained clusters. In contrast, a larger $\mathcal{S}$ produces coarser clusters.

$\mathcal{S}$ is defined in Section 3.5, the bucket size $\mathcal{S}$ is varied for Q_1 with ω_1 using (bounds $[0.44, 0.5]$) on the Healthcare dataset and Q_4 with ω_3 using (bounds $[0.34, 0.39]$) on the ACSIncome dataset. The bucket size ranges from 5 to 2,500. Using the default branching factor $\mathcal{B}$ of 5, the structure of the kd-tree for this evaluation is as follows:

- Level 1: 5 clusters, each with 10,000 data points;
- Level 2: 25 clusters, each with 2,000 data points;
- Level 3: 125 clusters, each with 400 data points;
- Level 4: 625 clusters, each with 80 data points;
- Level 5: 3,125 clusters, each with 16 data points;
- Level 6: 15,625 clusters, each with 3 or 4 data points.

Note that the algorithms will generate kd-tree up to the level where the capacity of each cluster at that level is less than or equal to $\mathcal{S}$. For example, for $\mathcal{S} = 200$, the tree will have 4 levels. The full results and detailed discussion appear in Section 7.2.

3.9.3 Branching Factor $\mathcal{B}$:

This experiment is varying the branching factor $\mathcal{B}$. The intuition behind this is to investigate how the structure of the clustering tree, controlled by the branching factor $\mathcal{B}$, influences the performance of the proposed methods. A smaller $\mathcal{B}$ results in a deeper tree with fewer branches at each level. In contrast, a larger $\mathcal{B}$ produces a shallower tree with many clusters at each level.

$\mathcal{B}$ is defined in Section 3.5, the relationship between the branching factor $\mathcal{B}$ and the runtime of Full Cluster Filtering (FF) and Cluster Range Pruning (RP) are examined. The same queries, constraints, bounds, and datasets from the previous evaluation are used. In this experiment, the branching factor is varied from 5 to 30. The corresponding number of leaf nodes in the kd-tree is shown in Table 7.1. Using the default bucket size $\mathcal{S} = 15$, the branching factor affects the depth of the tree. The full results and detailed discussion appear in Section 7.2.

TABLE 3.3: 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

3.9.4 Top-k:

This experiment is varying the the number of repairs k returned by the methods. The intuition behind this is to assess how the computational overhead of the proposed algorithms scales with increasing user demand. As k increases, the algorithms are required to explore a larger portion of the search space to retrieve additional valid repairs.

The relationship between k and the runtime of Full Cluster Filtering (FF) and Cluster Range Pruning (RP) is examined using the same queries, constraints, bounds, and datasets from the previous evaluation. In this experiment, the parameter k is varied from 1 to 15. The full results and detailed discussion appear in Section 7.2.

3.9.5 Dataset scale:

The experiment is conducted to assess how the performance of the proposed algorithms scales with increasing data size.

The TPC-H dataset is used with sizes varying from 25 K to 500 K. For each size, the runtime, number of candidates evaluated (NCE), and number of clusters accessed (NCA) are measured using query Q_7 from Table 3.1 and constraint ω_5 from Table 3.2. Default settings for all parameters are applied (Section 3.5). The full results and detailed discussion appear in Section 7.2.

3.10 Comparison with Related Work

The proposed approaches are compared with Erica (Li et al., 2023), 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 aggregate constraints supported in this work. Erica returns all repairs that are not *dominated* by any other repair where a repair dominates another repair if it is at least as close to the user query for every condition θ_i and strictly closer in at least one condition. That is, Erica returns the skyline (Borzsony et al., 2001). Thus, different from the proposed approach, the number of returned repairs is not an input parameter in Erica. For a fair comparison, the minimal repairs are determined, and then k is set such that the proposed methods return a superset of the repairs returned by Erica. To conduct the evaluation for Erica, the available Python implementation is used (`https://github.com/JinyangLi01/Query_refinement`). [2] The queries, constraints, and the dataset from (Li et al., 2023) are adopted. Generated refinements and runtime of the proposed techniques are compared with Erica using Q_1 and Q_2 (Table 3.1) on the Healthcare dataset (size 50K) with constraints Ω_6 and Ω_7 (Table 3.2), respectively. The full results and detailed discussion appear in Section 7.3.

3.11 Summary

This thesis addresses the challenge of ensuring that SQL query results meet complex real-world constraints, such as fairness, representativeness, and compliance, which are increasingly important in decision making. These constraints often reflect broader organizational objectives and are especially relevant in domains such as healthcare, finance, education, and public policy, where data processes must adhere to ethical standards and societal expectations. Although analysts and data professionals can formulate queries to extract relevant data, standard SQL predicates are not designed to express such constraints over the structure or distribution of the result set. For example, a machine learning engineer may require a training dataset that is both demographically balanced and representative, or a policymaker may need data summaries that reflect equitable outcomes across regions or groups. These goals go beyond simple filtering conditions and constraints and require careful adjustments to query predicates to align output with more expressive constraints. This chapter outlines the methodology for developing and evaluating techniques to modify SQL queries so that their results satisfy complex constraints, while preserving the user's original intent as closely as possible.

[2]To achieve a fairer comparison of algorithms regarding the evaluation of constraints, the code is modified because this component is implemented in pure Python, whereas Erica used Pandas DataFrame operations implemented in C. A full implementation in a lower-level language such as C++ is left for future work.

Chapter 4 defines the aggregate constraint repair problem considered in this thesis formally, including user queries, aggregate constraints, and the concept of repair distance. Chapter 5 presents the Full Cluster Filtering (FF) algorithm, which clusters the data and evaluates repair candidates using aggregate bounds. Chapter 6 extends Full Cluster Filtering (FF) by introducing interval-based representations of repair candidates to allow more efficient pruning of the search space. Chapter 7 provides an experimental evaluation that tests the proposed algorithms on multiple queries, constraints and datasets. Finally, Chapter 8 summarizes the thesis contributions, discusses the proposed techniques, and outlines directions for future work.

Chapter 4

Problem Definition

This chapter defines the core components of the query repair problem studied in this thesis. It begins by introducing the structure of the user query and the types of aggregate constraints considered, since formalizing these elements is the first step toward developing an efficient query repair approach. Next, it formalizes the notion of query repair as modifying selection predicate constants to satisfy these constraints, and defines a distance metric to rank candidate repairs. Finally, it describes the size and complexity of the repair search space and establishes the problem's computational hardness.

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

4.1 User Query

The class of select-project-join (SPJ) is considered as a user query Q, i.e., relational algebra expressions of the form:

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

This expression represents a database query operation. It means that a set of relations (from R_1 to R_l) are first joined together using the join operation ($\bowtie$). Then, a filter is applied using certain conditions (θ), represented by the selection operator (σ_θ). Finally, only specific columns (A) are kept from the result, which is done using the projection operator (π_A).

The selection predicate θ of such a query is assumed as a conjunction of comparisons of the form $a_i \,\mathrm{op}\, c_i$. For numerical attributes a_i, $\mathrm{op} \in \{<, >, \leq, \geq, =, \neq\}$ are allowed and for categorical attributes a_i, op $\in \{=, \neq\}$ are only allowed. $Q(D)$ is used to denote the result of evaluating Q over D.

4.2 Aggregate Constraints (AC)

The user specifies requirements on the result of their query as an aggregate constraint AC. An AC is a comparison between a threshold and an arithmetic expression over the result of filter-aggregation queries. Such queries are of the form:

$$\gamma_{f(a)}(\sigma_\theta(Q(D)))$$

In this expression, the outer operator $\gamma_{f(a)}$ represents an aggregate function f applied to an attribute a. The input to this aggregation is given by $\sigma_\theta(Q(D))$, where σ_θ is a selection operator that filters the results of the base query $Q(D)$ (e.g., a condition like "age $\geq$ 30").

Where f is an aggregate function – one of $\mathbf{count}, \mathbf{sum}, \mathbf{min}, \mathbf{max}, \mathbf{avg}$ – and θ is a selection condition. Q^ω is used 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.

The aggregate constraints considered in this work are not monotone in general. An aggregate function f is monotonically increasing (decreasing), if $f(S_1) \leq f(S_2)$ when $S_1 \subseteq S_2$ (if $f(S_1) \geq f(S_2)$ when $S_1 \supseteq S_2$) for any two bags of values S_1 and S_2. Many query refinement and relaxation techniques (Li et al., 2023) exploit monotonicity to optimize search as relaxing (refining) a query Q's selection conditions is bound to increase (decrease) $f(Q(D))$ if f is monotone, e.g., by pruning unpromising candidates to find a refined query without enumerating all candidates in the search space (Wu and Madden, 2013; Bruno et al., 2006; Li et al., 2023). A constraint $\omega := \tau \,\text{op}\, \Phi$ may be non-monotone if

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

4.3 Query Repair

Given a user query Q, database D, and aggregate constraint ω that is violated on $Q(D)$, this work wants to generate a repaired version Q_{fix} of Q such that $Q_{fix}(D)$ fulfills ω. In this work, repairs are restricted to changes of the selection condition θ of Q. Recall that Q is an SPJ query with a conjunctive selection condition. That is, the user query condition is of the form:

$$\theta = \theta_1 \wedge \ldots \wedge \theta_m$$

The symbol $\wedge$ represents a logical AND, meaning that all individual conditions $\theta_1, \theta_2, \ldots, \theta_m$ must be satisfied simultaneously for a tuple to be included in the query result. Each θ_i is a comparison of the form $a_i \, \mathrm{op} \, c_i$.

A *repair candidate* is a query Q_{fix} that differs from Q only in the constants used in selection conditions, i.e., Q_{fix} uses a condition:

$$\theta' = {\theta_1}' \wedge \ldots \wedge {\theta_m}'$$

Where ${\theta_i}'$ is a condition $a_i \, \mathrm{op} \, {c_i}'$. A repair candidate is called a *repair* if

$$Q_{fix}(D) \models \omega.$$

In this expression, $Q_{fix}(D)$ denotes the result set returned by the repaired query Q_{fix} when evaluated over the database D. The symbol $\models$ indicates logical satisfaction, meaning that the result set of Q_{fix} satisfies the aggregate constraint ω.

4.4 Repair Distance

Ideally, this work would want to achieve a repair that minimizes the changes to the user's original query. Many different optimization criteria are reasonable and which criteria is the most important will depend on the application. This work focuses on minimizing changes to the user's query. It defines a distance metric between repair candidates based on their selection conditions. Consider the user query Q with selection condition $\theta_1 \wedge \ldots \wedge \theta_m$ and repair Q_{fix} with selection condition ${\theta_1}' \wedge \ldots \wedge {\theta_m}'$. Then the distance $d(Q, Q_{fix})$ is defined as:

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

where the distance between two predicates $\theta_i = a_i \operatorname{op} c_i$ and $\theta_i{}' = a_i \operatorname{op} c_i{}'$ for numeric attributes a_i is:

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

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

Example 4.1. *For the use case in Example 1.1, the repair candidate with conditions* $\texttt{Major} = EE$*,* $\texttt{Testscore} \geq 33$*, and* $\texttt{GPA} \geq 3.9$ *is more similar to the user query than the candidate with conditions* $\texttt{Major} = EE$*,* $Testscore \geq 37$*, and* $GPA \geq 3.85$ *based on the distance metric considered in this work. For the first candidate, the distance is* $1 + \frac{\mathbf{33}-33}{\mathbf{33}} + \frac{\mathbf{3}.9-3.8}{\mathbf{3}.8} = 1.026$ *while for the second candidate it is* 1.134*.*

The distance metric adopted here based on relative differences for numerical attributes and discrete mismatches for categorical attributes which is commonly used because it is simple, intuitive, and computationally efficient. Its main strength is that it captures proportional changes in numeric conditions and provides a clear binary distinction for categorical mismatches, which makes it easy to interpret and compare across repairs. However, this formulation also has limitations. For example, it treats all categorical mismatches as equally distant, regardless of semantic similarity, and it normalizes numeric differences relative to the original constant, which may overemphasize small denominators.

It is important to note that this metric was chosen primarily for convenience and clarity in this work, rather than because it is optimal in all settings. The framework developed here is general and could incorporate alternative distance metrics tailored to different use cases, for example, semantic similarity measures for categorical attributes or weighted distances to prioritize more important conditions. In practical deployments, the choice of distance metric would depend on the domain and the priorities of the application (e.g., minimizing semantic drift versus minimizing numerical deviation).

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

Aggregate constraint repair problem:

- **Input**: user query Q, database D, constraint ω, threshold k
- **Output**: The set of $\text{topk}_{Q_{fix}}$ repairs $\{Q^1_{\text{fix}}, \ldots, Q^k_{\text{fix}}\}$ satisfying ω and having the k smallest distances $d(Q, Q^i_{\text{fix}})$.

This work focuses on general solutions that also work for non-monotone constraints as they are more challenging. In a practical solution, the system may first detect if a constraint is monotone and then apply existing optimizations (Li et al., 2023) if it is.

4.5 Search Space

To generate a repair Q_{fix} of Q, this work must explores the search space of possible candidate repairs. Consider a query with a conjunction of conditions of the form $a_i \, \mathrm{op}_i \, c_i$ for $i \in [1, m]$. Let N_i denote the number of values in the active domain of a_i. Each candidate repair corresponds to choosing constants $[c'_1, \ldots, c'_m]$. This work will use $\vec{c} = [c'_1, \ldots, c'_m]$ to denote a repair and use $\mathbb{Q}_{fix}$ to denote the set of all candidate repairs. The number of candidate repairs depends on which comparison operators are used, e.g., for $\leq$ there are at most $N_i + 1$ possible values that lead to a different result in terms of which of the input tuples will fulfill the condition. To see why this is the case assume that the values in a_i sorted based on $\leq$ are $a_1, \ldots, a_p$. Then for any constant c, the condition $a_i \leq c$ includes tuples with values in $\{a_i \mid a_i \leq c\}$ and this filtered set of a_i values is always a prefix of $a_1, \ldots, a_p$. Thus, there are $N_i + 1 = p + 1$ for choosing the length of this sequence (0 to p). The size of the search space is $O(\prod_{i=1}^{m} N_i)$, exponential in m, the number of conditions in the user query. Unsurprisingly, the aggregate constraint repair problem is NP-hard in the schema size.

4.6 Mapping the Motivating Example to the Formal Model

This section instantiates the fairness-motivating example (Example 1.1) using the notation and definitions introduced in this chapter.

Database and Schema.

Let $D = \{\, R \,\}$, where R is a single relation with schema

$$R(\texttt{ID}, \mathit{Gender}, \mathit{Major}, \texttt{GPA}, \texttt{TestScore}, \mathit{OfferInterview}),$$

corresponding to the six attributes:

- `ID`: unique identifier for each applicant;
- `Gender` G: values in $\{\texttt{M},\texttt{F}\}$;
- `Major` M: e.g., `CS`, `EE`, $\ldots$;
- `GPA`: numerical grade e.g., 3.5, 4.8, $\ldots$;
- `TestScore` TS: numerical e.g., 80, 78, $\ldots$;
- `OfferInterview` Y: (binary, 1 for "yes," 0 for "no").

Each tuple $t \in R$ has the form

$$t = (\texttt{ID}, \mathit{Gender}, \mathit{Major}, \texttt{GPA}, \texttt{TestScore}, Y),$$

and D is the set of all such tuples.

User Query.

The prescreening query from Example 1.1 is an SPJ query Q over R:

$$Q(D) \; = \; \pi_A\Big(\sigma_\theta(R)\Big),$$

where the selection predicate

$$\theta \; = \; (\mathit{Major} = \texttt{CS}) \; \wedge \; (\texttt{TestScore} \geq 32) \; \wedge \; (\texttt{GPA} \geq 3.80).$$

In the formal notation:

$$\theta_1 : \texttt{Major} = \texttt{CS}, \quad \theta_2 : \texttt{TestScore} \geq 32, \quad \theta_3 : \texttt{GPA} \geq 3.80,$$

and

$$\theta \; = \; \theta_1 \; \wedge \; \theta_2 \; \wedge \; \theta_3.$$

Here `Major` is categorical, and `TestScore` and `GPA` are numerical, with comparison operators as allowed in Section 4.1.

Aggregate Constraint.

The fairness requirement is expressed as a statistical parity difference (SPD) constraint on G (gender) and Y (interview outcome). Define four filter-aggregation queries over $Q(D)$:

$$Q_1^\omega \; = \; \gamma_{\mathbf{count}(Y)}\Big(\sigma_{G=\texttt{M} \wedge Y=1}(Q(D))\Big), \quad Q_2^\omega \; = \; \gamma_{\mathbf{count}(G)}\Big(\sigma_{G=\texttt{M}}(Q(D))\Big),$$

$$Q_3^\omega \; = \; \gamma_{\mathbf{count}(Y)}\Big(\sigma_{G=\texttt{F} \wedge Y=1}(Q(D))\Big), \quad Q_4^\omega \; = \; \gamma_{\mathbf{count}(G)}\Big(\sigma_{G=\texttt{F}}(Q(D))\Big).$$

Using these, the SPD expression is:

$$\Phi(Q_1^\omega, Q_2^\omega, Q_3^\omega, Q_4^\omega) \; = \; \frac{Q_1^\omega}{Q_2^\omega} \; - \; \frac{Q_3^\omega}{Q_4^\omega}.$$

The employer's requirement **SPD** ≤ 0.2 becomes the aggregate constraint

$$\omega := 0.2 \; \geq \; \frac{Q_1^\omega}{Q_2^\omega} \; - \; \frac{Q_3^\omega}{Q_4^\omega}.$$

Here, $\boldsymbol{\Phi}$ is an arithmetic expression using operators $(-, /)$ over $\{Q_i^\omega\}$, $\geq$ is the comparison operator *op*, and 0.2 is the threshold τ.

Since division and subtraction appear, ω is non-monotone in general (see Section 4.2).

Repair Candidates.

A repair candidate Q_{fix} differs from Q only in the constants of θ. Formally, let

$$\theta'_1 : \texttt{Major} = c_1, \quad \theta'_2 : \texttt{TestScore} \geq c_2, \quad \theta'_3 : \texttt{GPA} \geq c_3,$$

where each c_i ranges over the active domain of the corresponding attribute. Then

$$Q_{fix}(D) = \pi_A\Big(\sigma_{\theta'}(R)\Big), \quad \theta' \; = \; \theta'_1 \; \wedge \; \theta'_2 \; \wedge \; \theta'_3.$$

Any triple (c_1, c_2, c_3) defines a repair candidate. For the categorical attribute $Major$, possible c_1 include $\{\texttt{CS}, \texttt{EE}, \dots\}$. For numerical attributes `TestScore` and `GPA`, c_2 and c_3 range over values present in $R.\texttt{TestScore}$ and $R.\texttt{GPA}$, respectively.

Distance Metric.

The distance between the original predicate θ_i and modified predicate θ'_i is defined in Section 4.4. Concretely:

$$d(\theta_1, \theta'_1) = \begin{cases} 0, & c_1 = \texttt{CS}, \\ 1, & c_1 \neq \texttt{CS}, \end{cases} \quad d(\theta_2, \theta'_2) = \frac{|c_2 - 32|}{32}, \quad d(\theta_3, \theta'_3) = \frac{|c_3 - 3.80|}{3.80}.$$

Thus, the overall distance is

$$d(Q, Q_{fix}) = d(\theta_1, \theta'_1) + d(\theta_2, \theta'_2) + d(\theta_3, \theta'_3).$$

Search Space and Complexity.

Let $_M$ be the number of distinct `Major` in attribute M, $_{TS}$ the number of distinct `TestScore` values, and $_{GPA}$ the number of distinct `GPA` values. Then the total number of repair candidates is

$$|\mathbb{Q}_{fix}| = {}_F \; \times \; {}_{TS} \; \times \; {}_{GPA},$$

which is exponential in the number of conjuncts (three in this example). As shown in Section 4.5, exploring all candidates is computationally in the worst case, motivating the efficient algorithms (FF, RP) developed later.

Summary of Formal Mapping.

In summary, the motivating example is captured by:

- $D = \{\, R \,\}$, with $R($ID, $Gender$, $Major$, GPA, TestScore, $OfferInterview)$.
- Original query Q with $\theta = (\texttt{Major} = \texttt{CS}) \wedge (\texttt{TestScore} \geq 32) \wedge (\texttt{GPA} \geq 3.80)$.

- Aggregate constraint

$$\omega := 0.2 \geq \frac{\textbf{count}(G = \texttt{M} \wedge Y = 1)}{\textbf{count}(G = \texttt{M})} - \frac{\textbf{count}(G = \texttt{F} \wedge Y = 1)}{\textbf{count}(G = \texttt{F})}.$$

- Repair candidates Q_{fix} vary constants c_1, c_2, c_3 in $\{\texttt{Major} = c_1,\ \texttt{TestScore} \geq c_2,\ \texttt{GPA} \geq c_3\}$.
- Distance metric $d(Q, Q_{fix}) = \sum_{i=1}^{3} d(\theta_i, \theta'_i)$.

With this formal mapping, the fairness motivating use case becomes an instance of Aggregate constraint repair problem, and the algorithms in subsequent chapters seek the top-k repairs of Q that satisfy ω with minimal distance.

4.7 Summary

This chapter formalizes the core components of query repair under non-monotone aggregate constraints. A user query is modeled as an SPJ expression with a conjunctive selection predicate (Section 4.1), and aggregate constraints compare a threshold to an arithmetic expression over filter-aggregation results (Section 4.2). Repairs are defined as modifications of selection-predicate constants that ensure the repaired query satisfies the aggregate constraint (Section 4.3), and a distance metric over predicate constants ranks candidate repairs by their closeness to the original query (Section 4.4). The repair search space grows exponentially with the number of predicates, which renders an exhaustive enumeration infeasible (Section 4.5). Finally, the motivating fairness example is mapped to this formal model, illustrating how a statistical parity difference constraint on a job-applicant query becomes an instance of the aggregate constraint repair problem (Section 4.6).

Chapter 5

The Full Cluster Filtering (FF) Algorithm

This chapter is addressing (RQ1) How can the shape of an instance of a dataset be leveraged to develop a more efficient approach for query repair under non-monotone aggregate constraints? by presenting *Full Cluster Filtering* (FF), the first proposed 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}.

The Full Cluster Filtering (FF) algorithm proceeds generally through three key steps to efficiently identify query repairs that satisfy a user-defined aggregate constraint:

- **Step 1: Partitioning the Space and Materializing Statistics** The dataset is partitioned into clusters organized in a tree structure based on the attributes appearing in the query's selection predicates. The rationale is that the selection conditions of a repair candidate filter data along these attributes. Then, each cluster is scanned once to record (i) the minimum and maximum values of the selection attributes, and (ii) precomputed results of relevant aggregate functions. These statistics allow the algorithm to avoid redundant computation during evaluation.
- **Step 2: Searching for Candidate Repairs** Repair candidates are generated by varying the constants in the query's selection predicates. They are evaluated in order of increasing distance from the original query. Each candidate is matched against cluster statistics to decide whether a cluster can be used entirely, skipped, or further explored.
- **Step 3: Evaluating the Constraints and Computing Top-k Repairs** For each valid repair candidate, the algorithm merges aggregate results from satisfying clusters and substitutes them into the aggregate constraint. This determines

whether the candidate qualifies as a valid repair. Candidates that satisfy the constraint are returned in order of increasing distance to the original query. The algorithm stops when k valid repairs are found.

The following sections describe each of these steps in detail. It begins in Section 5.1 which shows how the input data is partitioned into a kd-tree of clusters and how each cluster's filter-aggregate results and attribute bounds are precomputed.
Next, Section 5.2 explains how each repair candidate is mapped to a covering set of clusters and how the aggregate constraint ω is evaluated. After that, Section 5.3 describes how Top-k candidates are generated and returned. Finally, Section 5.4 provides a concise summary of the chapter.

5.1 Clustering and Materializing Aggregations

For ease of presentation, the algorithm considers a database consisting of a single table R from now on. However, it 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. The algorithm uses a K-dimensional tree (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.

kd-tree represents a hierarchical space-partitioning data structure that organizes points in k-dimensional space. Originally formulated by (Bentley, 1975a; Friedman et al., 1977), kd-tree generalizes binary search tree to accommodate multidimensional data, which enables efficient retrieval based on multiple attributes simultaneously. The architecture of a kd-tree begins with a root node encompassing the complete point space. As the structure develops, each internal node divides the space into number of distinct branches $\mathcal{B}$ (represented as child nodes) along one of the k dimensions. The splitting dimension typically alternates as one traverses down the tree, cycling through all dimensions and choosing the mid point. Every node contains a k-dimensional point along with two pointers that either reference child nodes or contain null values in the case of leaf nodes. Points with a smaller value in the splitting dimension are directed to the left subtree, while those with larger values proceed to the right subtree.

Example 5.1 (Running Example)**.** *Consider the toy relation R and the user's query and constraint from Example 1.1, simplified to a single predicate on `TestScore`:*

***Data** R:*

***Query** Q_1:*

TABLE 5.1: Example dataset R with TestScore, Gender, and label Y.

Id	*TestScore* (T)	*Gender* (G)	Y
t_1	30	F	1
t_2	27	F	0
t_3	37	M	1
t_4	34	M	1
t_5	31	M	1

```
SELECT *
  FROM R WHERE T >= 32;
```

Constraint $\omega_{\#}$***:***

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

The employer would like to ensure that the SPD between male and female is below 0.2.

A 1-dimensional kd-tree is constructed over the five `TestScore` *values* $\{27, 30, 31, 34, 37\}$ *in Figure 5.1.*

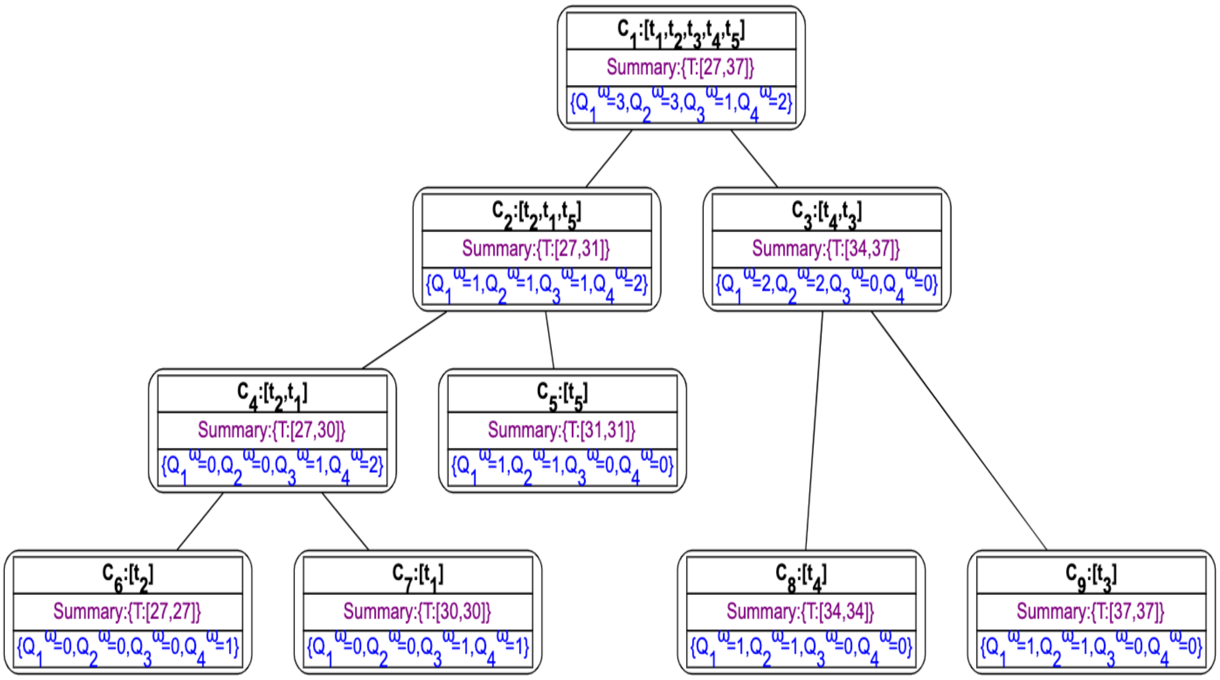


FIGURE 5.1: Hierarchical 1D k-d tree over TestScore. Each C_i node lists its point set, the $[\,\min, \max\,]$ bounds (purple) and the pre-aggregated values (blue).

To evaluate the aggregate constraint (AC) ω for a candidate $Q_{fix} = [c'_1, \ldots, c'_m]$, the algorithm determines a set of clusters (nodes in the kd-tree) that cover exactly the subset of D that fulfils the selection condition of the candidate. It can then merges 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, the algorithm records 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 it generates clusters bottom up.

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

An example kd-tree is shown in Figure 5.1. 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_2, t_1, and t_5, $\textsc{bounds}_T = [27, 31]$ as the lowest T value is 27 (from tuple t_2) and the highest value is 31 (tuple t_5). The value of $Q_2^{\omega} = \textbf{count}(Gender(G) = M)$ for C_2 is 1 as there is one male in the cluster. Consider a repair candidate with the condition $T \geq 37$. Based on the bounds $\textsc{bounds}_T = [27, 31]$, it is clear 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, FF algorithm assumes that the leaf nodes of the kd-tree contain a single tuple each. As this would lead to very large trees, the implementation of the algorithm does not further divide clusters C if it contains less tuples than a threshold $\mathcal{S}$ ($|C| \leq \mathcal{S}$). This parameter is called the *bucket size.*

5.2 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 precisely the subset of the relation R that fulfills θ':

$$\bigcup_{C \in \mathbf{C}} = \sigma_{\theta'}(R) \tag{5.1}$$

Algorithm 1 COVERINGCLUSTERS

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

```
1:  $stack \leftarrow [C_{root}]$
2:  $\mathbf{C} \leftarrow \emptyset$  ▷ Initialize result set
3:  while $stack \neq \emptyset$ do
4:      $C_{cur} \leftarrow \mathbf{pop}(stack)1$
5:      $in \leftarrow \mathbf{true}, notin \leftarrow \mathbf{false}$
6:      for all $\theta'_i = (a_i \, \mathrm{op}_i \, c'_i) \in \theta'$ do
7:          $in \leftarrow in \wedge \mathbf{eval}_\forall(\theta'_i, \textsc{bounds}_{a_i}(C_{cur}))$  ▷ All tuples fulfill $\theta'_i$?
8:          $notin \leftarrow notin \vee \mathbf{eval}_\forall(\neg\theta'_i, \textsc{bounds}_{a_i}(C_{cur}))$
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 $C \in \mathbf{children}(C_{cur})$ do  ▷ Process children
13:             $stack \leftarrow stack \cup \{C\}$
14: return $\mathbf{C}$
```

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

5.2.1 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 (5.1) has been determined (lines 3-14). For each cluster C, the algorithm distinguishes 3 cases (lines 6-8): (i) it uses the bounds on the selection attributes recorded for the cluster to show that all tuples in the cluster fulfil 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, it determines that none of the tuples in the cluster fulfil the condition (line 8). Then this cluster can be ignored; (iii) either a non-empty subset of C fulfils θ' or based on the bounds $\textsc{bounds}_{a_i}(C)$ it cannot demonstrate that $\sigma_{\theta'}(C) = \emptyset$ or $\sigma_{\theta'}(C) = C$ hold. In this case, the algorithm adds the children of C to the stack to be evaluated in future iterations (lines 11-13). It uses the function $\mathbf{eval}_\forall$ shown in Table 6.1 to determine based on the bounds of the cluster C, the comparison condition θ'_i is guaranteed to be true for all $t \in C$. Additionally, it checks whether case (ii) holds by applying $\mathbf{eval}_\forall$ to the negation θ'_i. Note that to negate a comparison, the algorithm simply push the negation to the comparison operator, e.g., $\neg(a < c) = (a \geq c)$. As the selection condition of any repair candidate is a conjunction of comparisons $\theta'_1 \wedge \ldots \wedge \theta'_m$, the cluster is *fully covered* (case (i)) if $\mathbf{eval}_\forall$ returns true for all θ'_i and *not covered at all* (case (ii)) if $\mathbf{eval}_\forall$ returns true for at least one comparison $\neg\theta'_i$.

TABLE 5.2: Given the bounds $[\underline{a}, \overline{a}]$ for the attribute a of a condition $a \,\mathrm{op}\, c$ or $a \in [c_1, c_2]$, function $\mathbf{eval}_\forall$ does return true if the condition evaluates to true for all values in $[\underline{a}, \overline{a}]$.

Op.	$\mathbf{eval}_\forall$
$>, \geq$	$\underline{a} > c, \quad \underline{a} \geq c$
$<, \leq$	$\overline{a} < c, \quad \overline{a} \leq c$
$=$	$\underline{a} = \overline{a} = c$
$\neq$	$c \notin [\underline{a}, \overline{a}]$
$\in [c_1, c_2]$	$c_1 \leq \underline{a} \wedge \overline{a} \leq c_2$

5.2.2 Determining Coverage

In Table 6.1, the algorithm defines the function $\mathbf{eval}_\forall$ which takes a condition $a \,\mathrm{op}\, c$ and bounds $\textsc{bounds}_a(C)$ for attribute a in cluster C and returns true if it is guaranteed that all tuples $t \in C$ fulfill the condition. 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, the algorithm 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 Example 5.1 and the kd-tree in Figure 5.1, The sorted candidates set for T is $\{31, 30, 34, 37, 27\}$. Consider a repair candidate with the condition $T \geq 31$, where $c_1 = 31$. 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, it proceeds to the children of C_1: C_2 and C_3. The same situation occurs for C_2 leading to further exploration of their child $\{C_4$ and $C_5\}$. In contrast, C_3 has $\textsc{bounds}_T(C_3) = [34, 37]$, so all of its tuples satisfy $T \geq 31$; the algorithm adds C_3 to $\mathbf{C}$ and do not explore further. Cluster C_4 has $\textsc{bounds}_T(C_4) = [27, 30]$, so none of its tuples can satisfy $T \geq 31$; C_4 and its entire subtree are pruned without visiting any leaves. In contrast, C_5 is confirmed to meet the condition and is added to $\mathbf{C}$. In this example, the leaf clusters C_6, C_7, C_8 and C_9 are not visited, as the algorithm prunes and confirms 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 $T \in [27, 34]$.

5.2.3 Constraint Evaluation

After identifying the covering set of clusters $\mathbf{C}$ for a repair candidate Q_{fix}, the algorithm evaluates the AC ω over $\mathbf{C}$. Recall that for each cluster C the algorithm materializes the result of each filter aggregate query Q_i^ω used in ω. For aggregate function **avg** that is not decomposable, the algorithm applies the standard approach of storing **count** and **sum** instead. It then computes $Q_i^\omega(Q(D))$ over the materialized aggregation results for the clusters. Concretely, for such an aggregate query $Q^\omega := \gamma_{\mathbf{f}(a)}(\sigma_{\theta'}(Q(D))$ it computes 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 used by the algorithm to merge aggregation results for multiple subsets of the database. This function depends on $\mathbf{f}$, e.g., for both **count** and **sum** $\mathbf{f}' = \mathbf{sum}$, for **min** $\mathbf{f}' = \mathbf{min}$, and for **max** $\mathbf{f}' = \mathbf{max}$. Then, the algorithm substitutes these aggregation results into ω and evaluate the resulting expression to determine whether Q_{fix} fulfils the constraints and is a repair or not.

For the running example in Example 5.1 the covering set of clusters for the repair candidate with $c_1 = 31$ is $\mathbf{C} = \{C_3, C_5\}$. Evaluating $Q_1^\omega = \mathbf{count}(Gender(G) = M \wedge Y = 1)$ over $\mathbf{C}$, the algorithm sums the counts:

$$Q_1^\omega = Q_{1C_3}^\omega + Q_{1C_5}^\omega = 2 + 1 = 3.$$

Similarly:

$$Q_2^\omega = Q_{2C_3}^\omega + Q_{2C_5}^\omega = 2 + 1 = 3,$$

$$Q_3^\omega = Q_{3C_3}^\omega + Q_{3C_5}^\omega = 0 + 0 = 0,$$

$$Q_4^\omega = Q_{4C_3}^\omega + Q_{4C_5}^\omega = 0 + 0 = 0.$$

Substituting these values into $\omega_{\#}$, yields $1 \leq 0.2 =$ **false**. Since the candidate $T \geq 31$ does not satisfy the constraint it is not a valid repair. The algorithm then proceeds to evaluate subsequent candidates in the sorted list, such as $T \geq 30$, $T \geq 34$, and $T \geq 37$, in the same way. All of these candidates fail to yield an SPD value below 0.2, and therefore do not satisfy the fairness constraint. However, when evaluating the candidate $T \geq 27$ in the same way as evaluating the candidate $T \geq 31$, the resulting SPD value falls below 0.2, indicating that the constraint is satisfied. Hence, $T \geq 27$ constitutes a valid repaired query for this example. The evaluation results for all candidate repairs are summarized in Table 5.3.

TABLE 5.3: Evaluation of candidate repairs for the running example.

Candidate Query	**SPD < 0.2?**	**Repair Status**
$T \geq 31$	No	Not a repair
$T \geq 30$	No	Not a repair
$T \geq 34$	No	Not a repair
$T \geq 37$	No	Not a repair
$T \geq 27$	Yes	**Valid repaired query**

5.3 Computing Top-k Repairs

To compute the top-k repairs, the algorithm enumerates all repair candidates in increasing order of their distance to the user query using the distance measure from Chapter 4. For each candidate Q_{fix} it applies the FF to determine a covering clusterset, evaluate the constraint ω, and output Q_{fix} if it fulfills the constraint. Once it has found k results, the algorithm terminates.

Table 5.3 lists the candidate repairs evaluated for the running example. The valid repaired query $T \geq 27$ demonstrates how the repair process modifies the original condition $T \geq 31$ to satisfy the fairness constraint. In practice, several top-k repaired queries may exist within a narrow region of the search space, for example, $T \geq 26$, $T \geq 27$, and $T \geq 28$. Presenting the top-k repairs allows users or analysts to inspect their respective results and determine which best fits their application context.

Future work could explore methods for diversifying the set of generated repairs. For instance, diversity-aware ranking or clustering techniques could be used to avoid producing near-identical candidate repairs within the same local region of the search space.

5.4 Summary

This chapter has presented the Full Cluster Filtering (FF) algorithm, the first proposed solution to the aggregate constraint repair problem. Its idea is to partition the input database D into a kd-tree of clusters and, in one scan, to materialize for each cluster both (a) the results of the filter–aggregate queries Q_i^ω and (b) the $[\min, \max]$ bounds on the predicate attributes a_i. Then, these precomputed aggregates and bounds are reused to evaluate any candidate Q_{fix} and the aggregate constraint ω for each repair Q_{fix}. The Full Cluster Filtering (FF) approach consists of three main phases:

1. **Building the kd-tree and materializing statistics** Partition the input relation R into a kd-tree of clusters and, in a single scan, compute and store each cluster's results for the filter–aggregate queries Q_i^ω together with the $[\underline{a_i}, \overline{a_i}]$ bounds on the predicate attributes a_i.
2. **Searching for candidate repairs** Enumerate repair candidates $Q_{fix} = [c'_1, \ldots, c'_m]$ in order of increasing distance from the original query. For each candidate, traverse the kd-tree with a stack, using cluster bounds to decide whether to (i) include a cluster's precomputed aggregates, (ii) prune the entire cluster, or (iii) recurse into its children.

3. **Evaluating the constraint and returning the Top–k.** Evaluate the aggregate constraint ω over the set of satisfying clusters $\mathbf{C}$, and terminate as soon as the first k valid repairs have been found.

The main advantage of this algorithm over the Brute Force (BF) approach is that it can reuse the aggregate query results materialized for a cluster if all tuples in the cluster fulfil the condition of the repair candidate and can skip any clusters that do not contain any tuples fulfilling the conditions.

Chapter 6

Cluster Range Pruning (RP)

While algorithm Full Cluster Filtering (FF) reduces the effort needed to evaluate aggregation constraints for individual repair candidates, it has the drawback that it still has to evaluate each repair candidate separately. To address this, this chapter presents an enhanced approach that reduces the search space by reasoning about sets of repair candidates. For a user query condition $\theta_1 \wedge \ldots \wedge \theta_m$ where $\theta_i := a_i \, \mathrm{op}_i \, c_i$, the algorithm uses ranges of constant values instead of constants to represent such a set of repairs $\mathbf{Q}$: $[[\underline{c_1}, \overline{c_1}], \ldots, [\underline{c_m}, \overline{c_m}]]$. Such a list of ranges $\mathbf{Q}$ 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 $\omega := \tau \, \mathrm{op} \, \Phi(Q_1^\omega, \ldots, Q_n^\omega)$. The enhanced approach Cluster Range Pruning (RP) uses a modified version of the kd-tree from Full Cluster Filtering (FF), but instead of storing constants it maintains, for each cluster, the precomputed *lower* and *upper* bounds of for each filter-aggregation query Q^ω in constraint ω. This allows the approach 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 $\mathbf{Q}$. Based on such bounds, if (i) $\tau \, \mathrm{op} \, c$ holds for every $c \in [\underline{\Phi}, \overline{\Phi}]$, then every $Q_{fix} \in \mathbf{Q}$ is a valid repair, if (ii) $\tau \, \mathrm{op} \, c$ is violated for every $c \in [\underline{\Phi}, \overline{\Phi}]$, then no $Q_{fix} \in \mathbf{Q}$ is a valid repair and can skip the whole set. Otherwise, (iii) there may or may not exist some candidates in $\mathbf{Q}$ that are repairs. In this case, the algorithm partitions $\mathbf{Q}$ into multiple subsets and applies the same test to each partition.

The Cluster Range Pruning (RP) algorithm proceeds mainly through three key steps to efficiently identify query repairs that satisfy a user-defined aggregate constraint:

- **Step 1: Searching for Candidate Repairs** The search space of repair candidates is represented as a set of intervals over the selection predicate values. Each interval encapsulates multiple individual repair conditions to reason about entire ranges at once rather than single constant modifications. These intervals are prioritized based on similarity to the original user query, typically using the minimum value in each interval for sorting. For a given interval, the algorithm determines by using the attribute bounds in the clusters whether: (i) all tuples in a cluster satisfy every repair condition in the interval (*full cover*); (ii) no tuple satisfies any repair in the interval (*no cover*); or (iii) some tuples may satisfy some repair conditions (*partial cover*). Full-cover clusters contribute precomputed aggregates directly, no-cover clusters are pruned, and partial clusters are recursively explored unless they are leaves, in which case they are included as partially satisfying.

- **Step 2: Evaluating Constraints for Repair Candidates** Once full and partial clusters are identified for an interval, the algorithm computes lower and upper bounds for each aggregate function using the materialized statistics. These bounds are then substituted into the arithmetic expression of the aggregate constraint to infer whether the constraint is satisfied over the entire interval. Three outcomes are possible: (i) if the interval bounds guarantee the constraint is satisfied for all repairs, the entire interval is accepted; (ii) if the interval partially overlaps the constraint threshold, the interval is subdivided and re-evaluated recursively; (iii) if the bounds indicate no repair in the interval satisfies the constraint, the interval is discarded.
- **Step 3: Computing Top-k Repairs** Repair intervals are evaluated in increasing order of distance from the original query. When an interval is found to fully satisfy the constraint, its individual repair candidates are extracted and added to the result set. This process continues until k valid repairs are found, at which point the algorithm terminates.

The following sections describe each of these steps in detail. It introduces an algorithm that utilizes such repair candidate sets and bounds on the aggregate constraint results and then explain how to use the kd-tree to compute such bounds. It begins in Section 6.1 which shows how the algorithm generates the set of top-k repairs $\mathcal{Q}_{top-k}$ by using ranges of constant values instead of constants to represent such a set of repairs $\mathbf{Q}$. Next, Section 6.2 explains how each repair candidate is mapped to a covering set of clusters and Section 6.3 shows how to compute bounds on constraints ω. Finally, Section 6.4 provides a concise summary of the chapter.

6.1 Computing Top-k Repairs

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

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

```
Input: kd-tree with root $C_{root}$, constraint AC $\omega$, repair candidate set $\mathbf{Q} = [[\underline{c_1}, \overline{c_1}], \ldots, [\underline{c_m}, \overline{c_m}]]$, user query condition $\theta = \theta_1 \wedge \ldots \wedge \theta_m$, user query $Q$
Output: Top-$k$ repairs $\mathcal{Q}_{top-k}$
 1: $\mathcal{Q}_{top-k} \leftarrow \emptyset$   ▷ Queue of repairs $Q'$ sorted on $d(Q, Q')$
 2: $rcand \leftarrow \emptyset$   ▷ Queue of repair sets $\mathbf{Q}'$ sorted on $\underline{d(Q, \mathbf{Q}')}$
 3: $queue \leftarrow [\mathbf{Q}]$   ▷ Queue of repair candidate sets $\mathbf{Q}'$ sorted on $\underline{d(Q, \mathbf{Q}')}$
 4: while $queue \neq \emptyset$ do
 5:     $\mathbf{Q}_{cur} \leftarrow$ POP($queue$)
 6:     $\mathbf{Q}_{next} \leftarrow$ PEEK($queue$)   ▷ Peek at next item in queue
 7:     $(\mathbf{C}_{full}, \mathbf{C}_{partial}) \leftarrow$ COVERINGCLUSTERSET($\mathbf{Q}_{cur}, C_{root}, \theta$)
 8:     if ACEVAL$_\forall$($\omega, \mathbf{C}_{full}, \mathbf{C}_{partial}$) then   ▷ All $Q' \in \mathbf{Q}_{cur}$ are repairs?
 9:         $rcand \leftarrow$ INSERT($rcand, \mathbf{Q}_{cur}$)
10:     else if ACEVAL$_\exists$($\omega, \mathbf{C}_{full}, \mathbf{C}_{partial}$) then   ▷ Some repairs?
11:         for $\mathbf{Q}_{new} \in$ RANGEDIVIDE($\mathbf{Q}_{cur}$) do   ▷ divide ranges
12:             if HASCANDIDATES($\mathbf{Q}_{new}$) then
13:                 $queue \leftarrow$ INSERT($queue, \mathbf{Q}_{new}$)
14:     $\mathcal{Q}_{top-k} \leftarrow$ TOPKCONCRETECAND($rcand, k$)   ▷ Top $k$ repairs
15:     if $|\mathcal{Q}_{top-k}| \geq k$ then   ▷ Have $k$ repairs?
16:         if $\underline{d(Q, \mathbf{Q}_{next})} > d(Q, \mathcal{Q}_{top-k}[k])$ then   ▷ Rest inferior?
17:             break
18: return $\mathcal{Q}_{top-k}$
```

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 $\mathbf{Q}$ encoded as ranges as shown above. For each $\mathbf{Q}$ the algorithm has established that for all $Q_{fix} \in \mathbf{Q}$, Q_{fix} is a repair. This query is sorted on the lower bound $\underline{d(Q, Q_{fix} \in \mathbf{Q})}$ of the distance of any repair in $\mathbf{Q}$ to the user query. Finally, (iii) *queue* is a queue where each element is a repair candidate set $\mathbf{Q}$. This queue is also sorted on $\underline{d(Q, Q_{fix} \in \mathbf{Q})}$. 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 $\mathbf{Q}$. The algorithm is called with a repair candidate set that covers the whole search space (line 1-3). The algorithm's main loop processes one repair candidate $\mathbf{Q}_{cur}$ at a time (line 5) while keeping track of the next candidate $\mathbf{Q}_{next}$ (line 6) until a set of top-k repairs fulfilling aggregate constraint (AC) ω has been determined (lines 4–17). For the current repair candidate set $\mathbf{Q}_{cur}$, function COVERINGCLUSTERSET (Algorithm 3) is used 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 $Q_{fix} \in \mathbf{Q}_{cur}$ and for every cluster $C \in \mathbf{C}_{partial}$, there may exist some tuples in C such that for some repair candidates $Q_{fix} \in \mathbf{Q}_{cur}$, the tuples fulfill the condition of Q_{fix}. These two sets of clusters are used 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 $Q_{fix} \in \mathbf{Q}_{cur}$. For that, the bounds $[\underline{\Phi}, \overline{\Phi}]$ are computed on Φ that hold for every $Q_{fix} \in \mathbf{Q}_{cur}$. If this is the case then all $Q_{fix} \in \mathbf{Q}_{cur}$ are repairs and $\mathbf{Q}_{cur}$ is added to *rcand* (lines 8–9); (ii) function ACEVAL$_\exists$ determines $\mathbf{C}_{full}$ and $\mathbf{C}_{partial}$ to check whether some repair candidates $Q_{fix} \in \mathbf{Q}_{cur}$ 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 $Q_{fix} \in \mathbf{Q}_{cur}$ is a repair and $\mathbf{Q}_{cur}$ can be discarded. These functions will be discussed in depth in Section 6.3.

For example, if $\omega := 0.7 \leq \Phi$ and the bounds $[\underline{\Phi}, \overline{\Phi}] = [0.5, 1]$ are computed that hold for all $Q_{fix} \in \mathbf{Q}_{cur}$, then ACEVAL$_\forall$ returns false as some $Q_{fix} \in \mathbf{Q}_{cur}$ may not fulfill the constraint. However, ACEVAL$_\exists$ return true as some $Q_{fix} \in \mathbf{Q}_{cur}$ may fulfill the constraint. In this case, the algorithm partitions $\mathbf{Q}_{cur}$ into smaller sub-ranges $\mathbf{Q}_{new}$ using the function RANGEDIVIDE($\mathbf{Q}_{cur}$) (line 11). Assume that $\mathbf{Q}_{cur} = [[\underline{c_1}, \overline{c_1}], \ldots, [\underline{c_m}, \overline{c_m}]]$. RANGEDIVIDE splits each range $[\underline{c_i}, \overline{c_i}]$ into a fixed number of fragments $\{[\underline{c_{i_1}}, \overline{c_{i_1}}], \ldots, [\underline{c_{i_l}}, \overline{c_{i_l}}]\}$ such that each $[\underline{c_{i_j}}, \overline{c_{i_j}}]$ is roughly of the same size and returns the following set of repair candidate sets:

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

That is, each $\mathbf{Q}_{new}$ 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 $\mathbf{Q}_{cur}$. $l = 2$ is used in the implementation of the algorithm. The function HASCANDIDATES (line 12-13) checks whether each range in $\mathbf{Q}_{new}$ 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 the discussion of the search space at the end of Chapter 4 that only values from the active domain of an attribute are considered as constants for repair candidates. That is, the algorithm can skip candidate repair sets $\mathbf{Q}_{new}$ 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, $\mathbf{Q}_{new}$ is inserted into the priority queue *queue* to be processed in future iterations of the main loop. In each iteration the algorithm uses function TOPKCONCRETECAND (line 14) to determine the k repairs Q_i across all $\mathbf{Q} \in rcand$ with the lowest distance to the user query Q. If the algorithm can find k such candidates (line 15), then it tests whether no repair candidate from the next repair candidate set $\mathbf{Q}_{next}$ may be closer to Q then the kth candidate $\mathcal{Q}_{top-k}[k]$ from $\mathcal{Q}_{top-k}$ (line 16). This is the case if the lower bound on the distance of any candidate in $\mathbf{Q}_{next}$ is larger than the distance of $\mathcal{Q}_{top-k}[k]$. 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, $\mathcal{Q}_{top-k}$ contains exactly the top-k repairs and the algorithm returns $\mathcal{Q}_{top-k}$.

Algorithm 3 COVERINGCLUSTERSET

```
Input: kd-tree with root $C_{root}$, repair candidate set $\mathbf{Q} = [[\underline{c_1}, \overline{c_1}], \ldots, [\underline{c_m}, \overline{c_m}]]$, condition $\theta$
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:      $C_{cur} \leftarrow$ pop$(stack)$
5:      $in \leftarrow$ true$, pin \leftarrow$ true
6:      for all $\theta_i = (a_i \, \mathrm{op}_i \, c_i) \in \theta$ do   ▷ $C_{cur}$ fully / part. covered?
7:          $in \leftarrow in \wedge \mathbf{reval}_{\forall}(\theta_i, [\underline{c_i}, \overline{c_i}], \mathrm{BOUNDS}_{a_i}(C_{cur}))$
8:          $pin \leftarrow pin \wedge \mathbf{reval}_{\exists}(\theta_i, [\underline{c_i}, \overline{c_i}], \mathrm{BOUNDS}_{a_i}(C_{cur}))$
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 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 $C \in$ children$(C_{cur})$ do
16:                 $stack \leftarrow stack \cup \{C\}$
17: return $(\mathbf{C}_{full}, \mathbf{C}_{partial})$
```

6.2 Determining Covering Cluster Sets

Similar to FF, the kd-tree is used to determine a covering cluster set $\mathbf{C}$. However, as the RP algorithm now deals with a set of candidate repairs $\mathbf{Q}$, a $\mathbf{C}$ must be found such that for all $Q_{fix} \in \mathbf{Q}$, the following holds: $Q_{fix}(D) = \bigcup_{C \in \mathbf{C}} C$. Such a covering cluster set is unlikely to exist as for any two $Q_{fix} \neq Q'_{fix} \in \mathbf{Q}$ it is likely that $Q_{fix}(D) \neq Q'_{fix}(D)$. Instead the algorithm relaxes the condition and allows clusters C that are *partially covered*, i.e., for which some tuples in C may be in the result of some candidates in $\mathbf{Q}$. Algorithm 1 is modified 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 $Q_{fix} \in \mathbf{Q}$ and $\mathbf{C}_{partial}$ which contains clusters that are only partially covered.

Analogous to Algorithm 1, the updated algorithm (Algorithm 3) maintains a stack of clusters to be processed that is initialized with the root node of the kd-tree (line 1). In each iteration of the main loop (line 3-16), the algorithm determines whether all tuples of the current cluster C_{cur} fulfill the conditions θ_i for all repair candidates $Q_{fix} \in \mathbf{Q}$. This is done using function $\mathbf{reval}_{\forall}$ (line 7). Additionally, the algorithm checks whether it is possible that at least one tuple fulfills the condition of at least one repair candidate $Q_{fix} \in \mathbf{Q}$. This is done using a function $\mathbf{reval}_{\exists}$ (line 8). If the cluster is fully covered, the algorithm adds it to the result set $\mathbf{C}_{full}$ (line 10). If it is partially covered, then it distinguishes between two cases (line 11- 16). Either the cluster is a

TABLE 6.1: For RP, the algorithm considers 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 $\in$. $\textbf{reval}_\forall$ 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 $\textbf{reval}_\exists$ determines whether for some $c \in [\underline{c}, \overline{c}]$, the condition may evaluate to true for $a \in [\underline{a}, \overline{a}]$.

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

leaf node (line 12-13) or it is an inner node (line 14-16). If the cluster is a leaf, then the algorithm cannot further divide the cluster and add it to $\mathbf{C}_{partial}$. If the cluster is an inner node, then it processes its children, as some children may be determined to be either fully covered or not covered at all.

Table 6.1 shows how conditions are evaluated by $\textbf{reval}_\forall$ and $\textbf{reval}_\exists$. 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 fulfil the condition for all $Q_{fix} \in \mathbf{Q}$. 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.

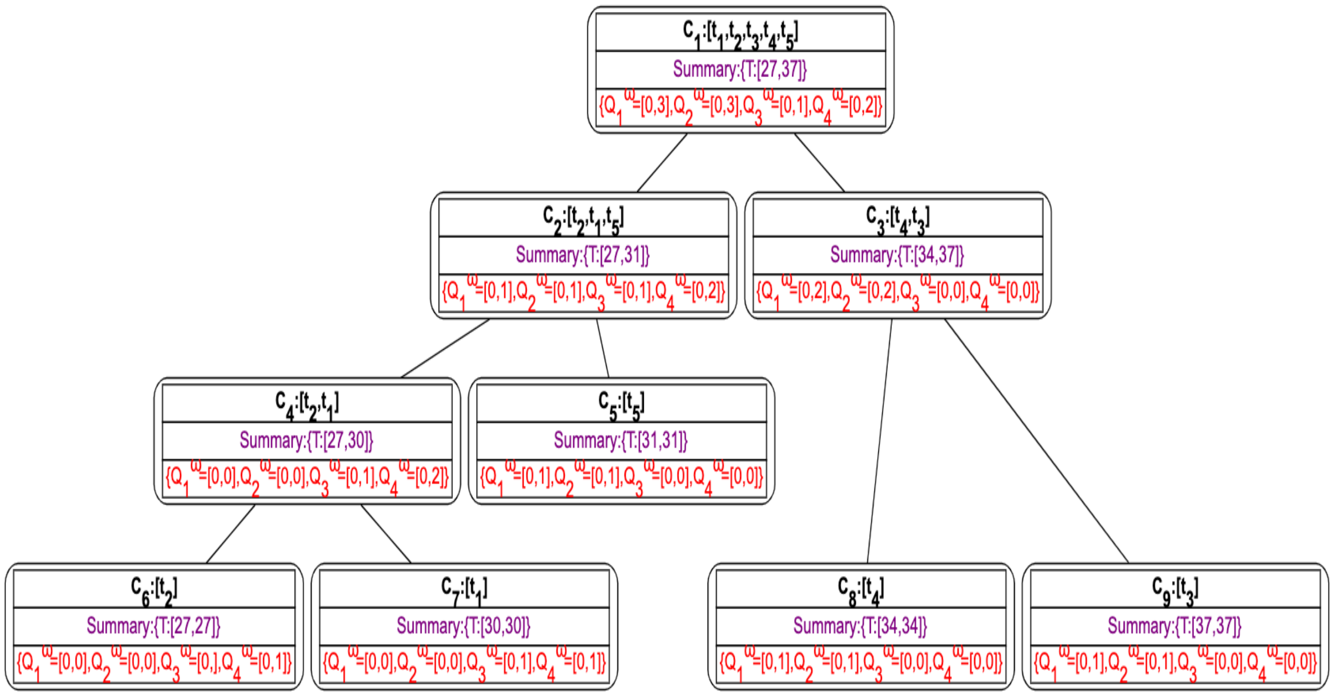

C1:[t1,t2,t3,t4,t5]
Summary:{T:[27,37]}
{Q1ω=[0,3],Q2ω=[0,3],Q3ω=[0,1],Q4ω=[0,2]}
C2:[t2,t1,t5]
Summary:{T:[27,31]}
{Q1ω=[0,1],Q2ω=[0,1],Q3ω=[0,1],Q4ω=[0,2]}
C3:[t4,t3]
Summary:{T:[34,37]}
{Q1ω=[0,2],Q2ω=[0,2],Q3ω=[0,0],Q4ω=[0,0]}
C4:[t2,t1]
Summary:{T:[27,30]}
{Q1ω=[0,0],Q2ω=[0,0],Q3ω=[0,1],Q4ω=[0,2]}
C5:[t5]
Summary:{T:[31,31]}
{Q1ω=[0,1],Q2ω=[0,1],Q3ω=[0,0],Q4ω=[0,0]}
C6:[t2]
Summary:{T:[27,27]}
{Q1ω=[0,0],Q2ω=[0,0],Q3ω=[0,],Q4ω=[0,1]}
C7:[t1]
Summary:{T:[30,30]}
{Q1ω=[0,0],Q2ω=[0,0],Q3ω=[0,1],Q4ω=[0,1]}
C8:[t4]
Summary:{T:[34,34]}
{Q1ω=[0,1],Q2ω=[0,1],Q3ω=[0,0],Q4ω=[0,0]}
C9:[t3]
Summary:{T:[37,37]}
{Q1ω=[0,1],Q2ω=[0,1],Q3ω=[0,0],Q4ω=[0,0]}

FIGURE 6.1: Hierarchical 1D k-d tree over TestScore. Each C_i node lists its point set, the $[\,\min, \max\,]$ bounds (purple) and the pre-aggregated bounds (red).

In the running example from Example 5.1 and the updated kd-tree in Figure 6.1. The sorted candidates set for T is {[33, 37], [27, 32]}. Consider a repair candidate $[[33, 37]]$. Recall that the single condition in this example is $T \geq c$. C_{root} has $\textsc{bounds}_T = [27, 37]$. 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, $\underline{a} = 34$, is not greater than the upper bound of the constant, $\overline{c} = 37$ and additional clusters must be processed, C_8 and C_9. C_2, fails both $\mathbf{reval}_\forall$ and $\mathbf{reval}_\exists$, so no tuple in the range $[[27, 31]]$ can satisfy any repair candidate in the range $[[33, 37]]$. Consequently, its entire subtree is pruned without visiting any leaves. Finally, the algorithm applies $\mathbf{reval}_\forall$ and $\mathbf{reval}_\exists$ if necessary to the clusters C_8 and C_9, confirming that $C_8 \in \mathbf{C}_{partial}$ and $C_9 \in \mathbf{C}_{full}$, as $t_4.T = 34 \geq c$ is only true for some $c \in [33, 37]$ and $t_3.T = 37 \geq c$ is true for all $c \in [33, 37]$.

6.3 Computing Bounds on Constraints

Given the cluster sets $(\mathbf{C}_{full}, \mathbf{C}_{partial})$ computed by Algorithm 3, the approach next (i) computes bounds on the results of the aggregation queries Q_i^ω used in the constraint, then (ii) uses 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 $\mathbf{Q}$. These bounds are conservative in the sense that all possible results are guaranteed to be included in these bounds. Then, finally, (iii) function $\textsc{aceval}_\forall$ uses the computed bounds to determine whether all candidates in $\mathbf{Q}$ fulfill the constraint by applying $\mathbf{reval}_\forall$ from Table 6.1. For a constraint $\omega := \tau \text{ op } \Phi$, $\textsc{aceval}_\forall$ calls $\mathbf{reval}_\forall$ with $[\underline{\Phi}, \overline{\Phi}]$ and τ. $\textsc{aceval}_\exists$ uses $\mathbf{reval}_\exists$ instead to determine whether some candidates in $\mathbf{Q}$ 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.

6.3.1 Bounding Aggregation Results

Bounds estimation is a fundamental computational methodology that determines upper and lower limits for the possible values of mathematical expressions, aggregate functions, or variable sets without necessitating exact value computation. Bounding aggregation functions involves calculating accurate upper and lower limits for the possible values of aggregate functions such as **SUM**, **COUNT**, **MIN**, and **MAX** without computing all possible aggregates for all possible results (Zhang et al., 2007).

Table 6.2: Bounds on applying an operator to the result of expressions E_1 and E_2 with interval bounds (Zhang et al., 2007).

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

This section discusses how to compute bounds for the results of the filter-aggregation queries Q_i^ω of an aggregate constraint ω based on the cluster sets $(\mathbf{C}_{full}, \mathbf{C}_{partial})$ returned by Algorithm 3. As every cluster C in $\mathbf{C}_{full}$ is fully covered for all repair candidates in $\mathbf{Q}$, i.e., all tuples in the cluster fulfill the conditions of each $Q_{fix} \in \mathbf{Q}$, 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 Full Cluster Filtering (FF). For partially covered clusters $(\mathbf{C}_{partial})$, worst case assumptions must be made to derive valid lower and upper bounds. For the lower bound, the approach has to consider the minimum across two options: (i) no tuples from the cluster will fulfill the condition of at least one Q_{fix} in $\mathbf{Q}$. 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 **min**(a) the approach has to reason about two cases: (i) it can add $\underline{a}$ to the aggregation in case of negative numbers; (ii) otherwise should ignore this cluster for computing lower bounds. For **sum** it has the two cases: (i) the attribute for the aggregation has negative numbers. In this case the approach sums the negative numbers for the lower bound. (ii) otherwise should ignore this cluster for computing lower bounds. For the upper bound the approach has 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, the aggregation bounds are included in the computation for the upper bound.

6.3.2 Bounding Results of Arithmetic Expressions

Given the bounds on aggregate-filter queries, *interval arithmetic* (Stolfi and de Figueiredo, 2003; De Figueiredo and Stolfi, 2004) is used which computes sound bounds for the result of arithmetic operations when the inputs are bound by intervals. In the case considered by this work, 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 used in this approach is similar to (Zhang et al., 2007). Table 6.2 shows the definitions for arithmetic operators supported 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 $\underline{E_1} + \underline{E_2}$.

6.3.3 Bounding Aggregate Constraint Results

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

In the running example and the updated kd-tree in Figure 6.1, the covering set of clusters for repair candidate set $\mathbf{Q}$:=[[33, 37]] are $\mathbf{C}_{partial} = \{C_8\}$ and $\mathbf{C}_{full} = \{C_9\}$. 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_8, the lower bound of $Q^\omega_{1\,C_8}$ 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, the following bounds are obtained for $Q^\omega_{1\,C_8} = [0, 1]$. Similarly, the remaining aggregation bounds are computed:

$$Q^\omega_{1\,C_9} = [1, 1],$$

$$Q^\omega_{2\,C_8} = [0, 1], \quad Q^\omega_{2\,C_9} = [1, 1],$$

$$Q^\omega_{3\,C_8} = [0, 0], \quad Q^\omega_{3\,C_9} = [0, 0],$$

$$Q^\omega_{4\,C_8} = [0, 0], \quad Q^\omega_{4\,C_9} = [0, 0].$$

Next, the lower and upper bounds for each aggregation Q_i^ω are summed across all clusters in $\mathbf{C}$:

$$Q_1^\omega = Q^\omega_{1\,C_8} + Q^\omega_{1\,C_9} = [1, 2],$$

$$Q_2^\omega = Q^\omega_{2\,C_8} + Q^\omega_{2\,C_9} = [1, 2],$$

$$Q_3^\omega = Q_{3\,C_8}^\omega + Q_{3\,C_9}^\omega = [0, 0],$$

$$Q_4^\omega = Q_{4\,C_8}^\omega + Q_{4\,C_9}^\omega = [0, 0].$$

The computed values $\{Q_1^\omega, Q_2^\omega, Q_3^\omega, Q_4^\omega\}$ are then substituted into $\omega_\#$ and evaluate the resulting expression using interval arithmetic (Table 6.2). Given: $\omega_\# = \mathbf{Q}_1^!/\mathbf{Q}_2^! - \mathbf{Q}_3^!/\mathbf{Q}_4^!$ the lower and upper bounds for the first term $\mathbf{Q}_1^!/\mathbf{Q}_2^!$ are computed as:

$$[\underline{\mathbf{E_1}/\mathbf{E_2}}, \quad \overline{\mathbf{E_1}/\mathbf{E_2}}] = [1/2, 2].$$

Similarly, for the second term: $\mathbf{Q}_3^!/\mathbf{Q}_4^! = [0, 0]$. Applying interval arithmetic to compute the subtraction yields:

$$[\underline{\mathbf{E_1} - \mathbf{E_2}}, \quad \overline{\mathbf{E_1} - \mathbf{E_2}}].$$

Thus, bounds $[\underline{\Phi_\#}, \overline{\Phi_\#}] = [1/2, 2]$ are obtained. Since $\underline{\Phi_\#} = 1/2 > 0.2$, none of the candidates in $\mathbf{Q} = [[33, 37]]$ can be repairs, allowing $\mathbf{Q}$ to be pruned. The algorithm then proceeds to the next candidate in the sorted candidates set, namely $[[27, 32]]$.

6.4 Summary

This chapter presented Cluster Range Pruning (RP), an enhanced algorithm that reasons about *sets* of repair candidates in one shot. Rather than testing each Q_{fix} individually, the algorithm uses ranges of constant values instead of constants to represent such a set of repairs $\mathbf{Q}$: $[[\underline{c_1}, \overline{c_1}], \ldots, [\underline{c_m}, \overline{c_m}]]$.

The RP pipeline consists of three main phases:

1. **Building the kd-tree and materializing bounds statistics** As in Full Cluster Filtering (FF), partition the input relation R into a kd-tree of clusters and, in a single scan, compute and store each cluster's $[\min, \max]$ bounds for the filter–aggregate queries Q_i^ω together with the $[\underline{a_i}, \overline{a_i}]$ bounds on the predicate attributes a_i.
2. **Searching for candidate repairs** Treat the entire repair space as a ranges $\mathbf{Q}$: $[[\underline{c_1}, \overline{c_1}], \ldots, [\underline{c_m}, \overline{c_m}]]$, ordered by its minimum distance to the original query. Repeatedly pop the next repair candidate and evaluate each cluster by comparing the bounds of the predicate attributes $[\underline{a}, \overline{a}]$ to the candidate's range: (i) If every value in $[\underline{a}, \overline{a}]$ falls within the candidate's range, consider the cluster as $\mathbf{C}_{full}$. (ii) If every value in $[\underline{a}, \overline{a}]$ lies outside the candidate's range, discard the cluster and its subtree. (iii) Otherwise the cluster is partially covered: if it is a leaf, add it to $\mathbf{C}_{partial}$ while if it is an internal node, push its children onto the stack for further processing.
3. **Evaluating the constraint and returning the Top–k** Compute bounds for the results of the filter-aggregation queries Q_i^ω of an aggregate constraint ω based

on the cluster sets $(\mathbf{C}_{full}, \mathbf{C}_{partial})$. Use these bounds to compute bounds $[\underline{\mathbf{\Phi}}, \overline{\mathbf{\Phi}}]$ on the result of the arithmetic expression $\mathbf{\Phi}$ of the AC ω over repair candidates in $\mathbf{Q}$ to determine if (i) $\tau \,\mathrm{op}\, c$ holds for every $c \in [\underline{\mathbf{\Phi}}, \overline{\mathbf{\Phi}}]$, then every $Q_{fix} \in \mathbf{Q}$ is a valid repair, if (ii) $\tau \,\mathrm{op}\, c$ is violated for every $c \in [\underline{\mathbf{\Phi}}, \overline{\mathbf{\Phi}}]$, then no $Q_{fix} \in \mathbf{Q}$ is a valid repair and can skip the whole set. Otherwise, (iii) there may or may not exist some candidates in $\mathbf{Q}$ that are repairs. In this case, the algorithm partitions $\mathbf{Q}$ into multiple subsets and applies the same test to each partition. The algorithm terminates as soon as the first k valid repairs have been found.

The advantage of this approach over Full Cluster Filtering (FF) is that it often enables the algorithm to prune sets of repair candidates or confirm all of them to be repairs without individually evaluating them.

Chapter 7

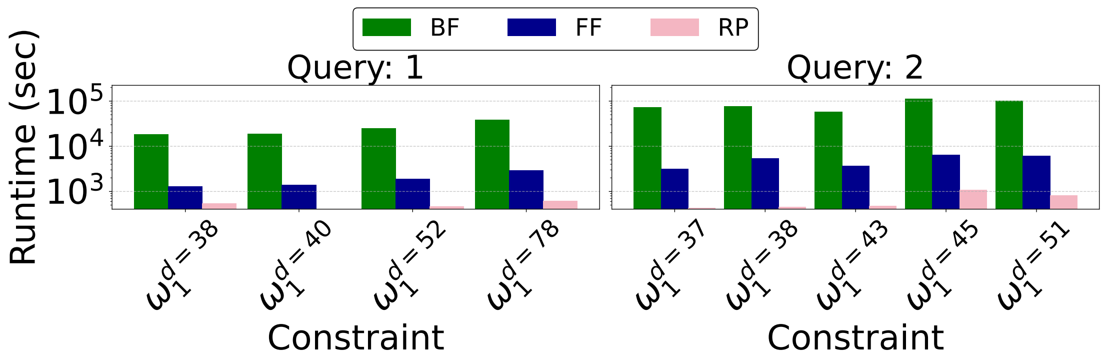
Evaluation

The performance of the proposed algorithms, Full Cluster Filtering (FF) and Cluster Range Pruning (RP), is evaluated using both real-world datasets and the TPC-H benchmark. The evaluation involves a variety of queries and arithmetic constraints, as described in Section 3.5. This chapter begins by comparing the performance of the proposed methods, Full Cluster Filtering (FF) and Cluster Range Pruning (RP), against the exhaustive Brute Force (BF) (Section 7.1) as well as against each other (Section 7.1). It then examines the impact of key parameters such as data size, clustering structure (branching factor and bucket size), exploration distance, and the top-k value on the performance of the proposed repair methods (Section 7.2). Finally, it compares the proposed approaches with the most relevant prior technique, Erica (Li et al., 2023), as detailed in (Section 7.3).

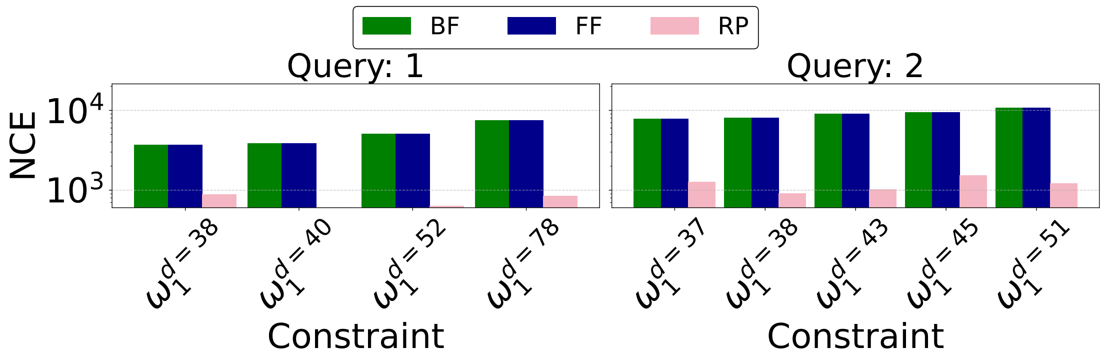
7.1 Performance of FF and RP

Experiments are conducted to measure the performance of Full Cluster Filtering (FF) and Cluster Range Pruning (RP) using the Healthcare and ACSIncome datasets with queries in Table 3.1, constraints in Table 3.2, and default settings in Section 3.5. For the Healthcare, the constraints ω_1 and ω_2 are used while ω_3 and ω_4 are considered for the ACSIncome. In addition to runtime, this experiment also measures number of candidates evaluated (NCE) which is the total of number of repair candidates evaluate the AC and number of clusters accessed (NCA) which is the total number of clusters accessed by an algorithm.

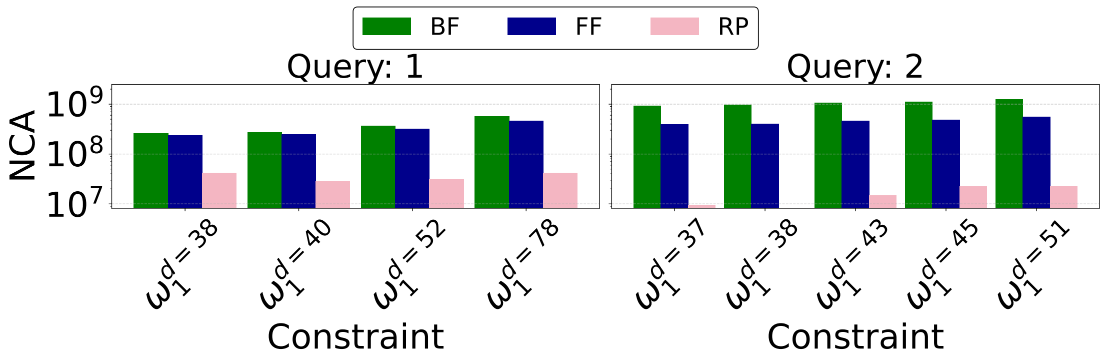
Comparison with Brute Force (BF). This section first compares the proposed methods, Full Cluster Filtering (FF) and Cluster Range Pruning (RP), with the Brute Force (BF) method using the Healthcare, queries Q_1 and Q_2, the constraint $\omega_1$and default settings in Section 3.5. As expected, both Full Cluster Filtering (FF) and



(A) Runtime (sec).



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

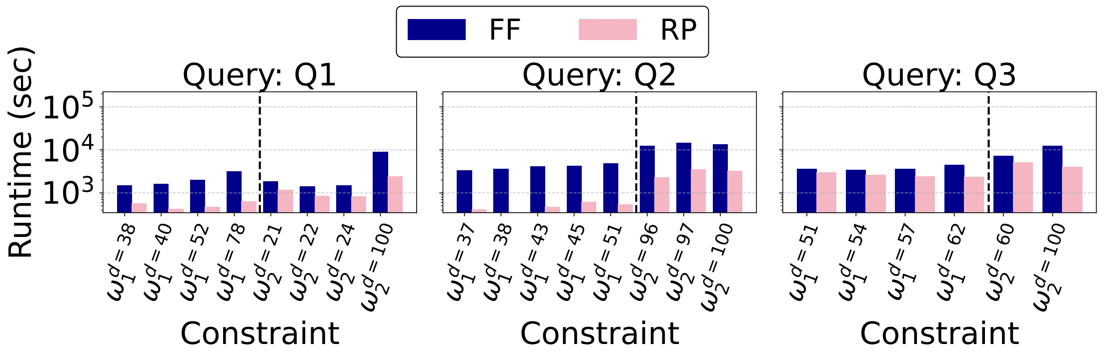


(C) Total number of cluster accessed (NCA).

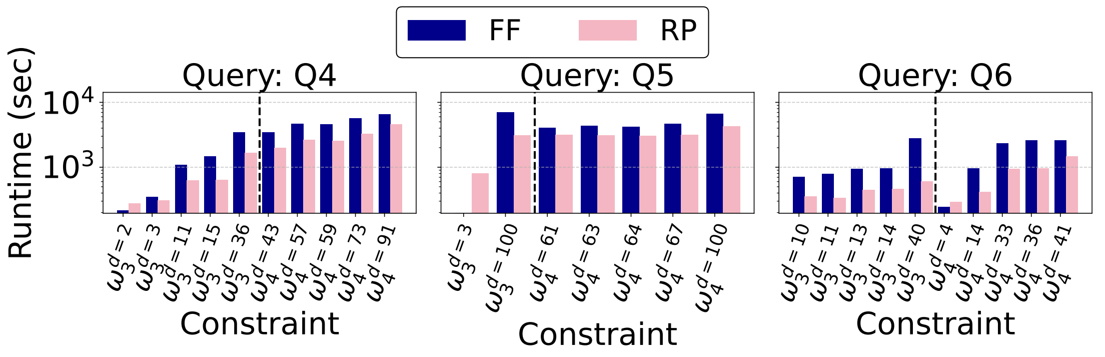
FIGURE 7.1: Runtime, NCE, and NCA for FF, RP, and BF over the Healthcare dataset.

Cluster Range Pruning (RP) outperform Brute Force (BF) by at least one order of magnitude in terms of runtime as shown in Figure 7.1a. This performance gain is primarily due to their ability to avoid evaluating every possible tuple and repair individually, unlike Brute Force (BF), which performs exhaustive checking. Full Cluster Filtering (FF) achieves this by computing aggregate statistics over a set of clusters, allowing it to reuse pre-aggregated summaries instead of scanning tuples repeatedly. Cluster Range Pruning (RP), on the other hand, reduces runtime by evaluating constraints over sets of repair candidates simultaneously.

The Cluster Range Pruning (RP) algorithm significantly reduces both the total number of candidates evaluated (NCE) and the number of clusters accessed (NCA).



(A) Runtime (sec) - Healthcare dataset.

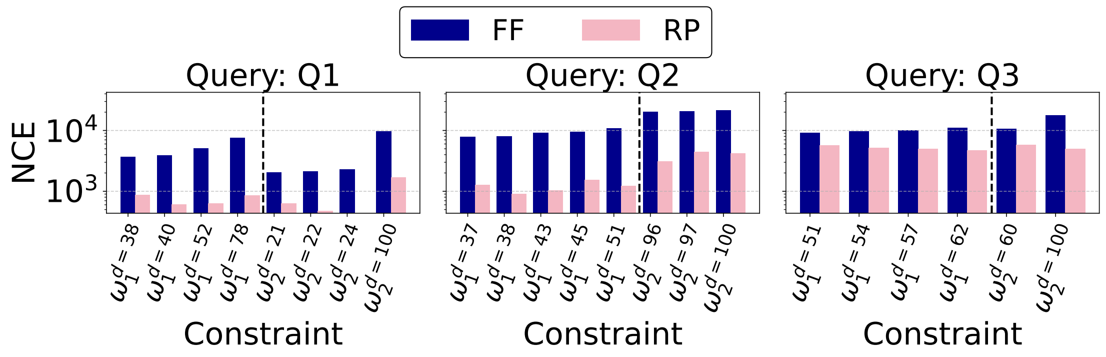


(B) Runtime (sec) - ACSIncome dataset.

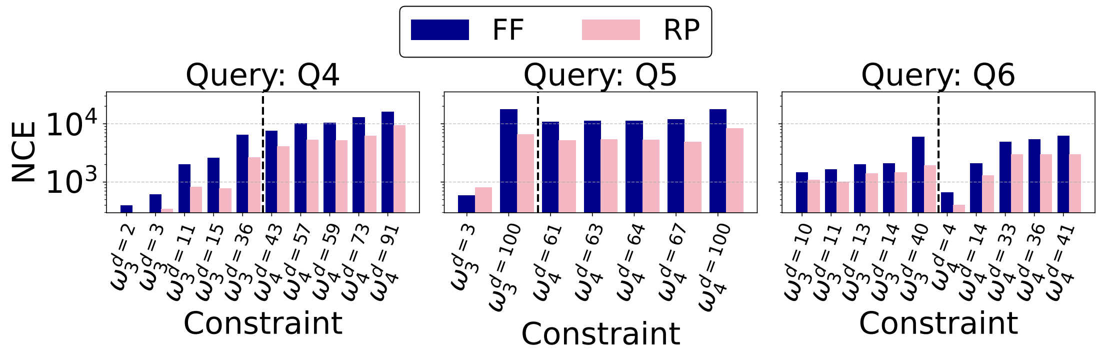
FIGURE 7.2: Runtime, NCE, and NCA for FF and RP over the Healthcare and ACSIncome datasets using the queries from Table 3.1.

This is achieved by using clustering and range-based pruning to skip large portions of the search space where no valid repairs can exist. While the Full Cluster Filtering (FF) method maintains the same number of candidates evaluated (NCE) as Brute Force (BF) because it still checks the same number of candidate repairs. However, it decreases the number of clusters accessed (NCA) compared to Brute Force (BF) (as BF does not use clusters, the number of tuple accesses is counted) as in Figure 7.1c and Figure 7.1b.

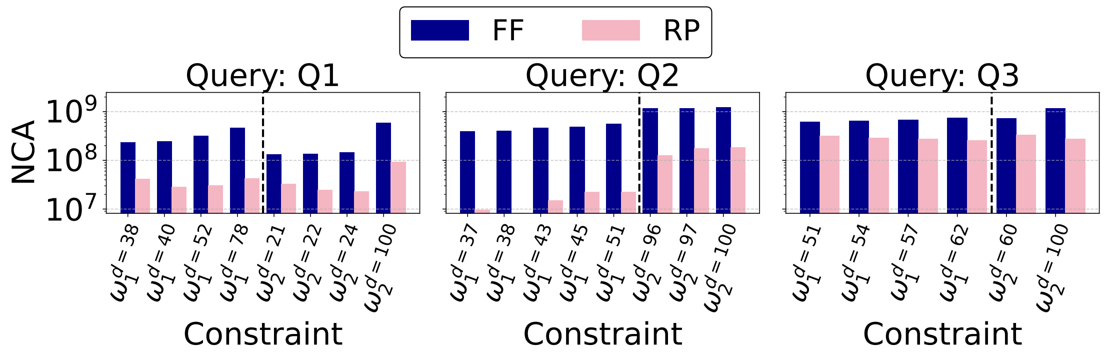
Runtime. Figures 7.2a and 7.2b show the runtime of the Full Cluster Filtering (FF) and Cluster Range Pruning (RP) algorithms for Healthcare and ACSIncome, respectively. For given constraint ω_i, the bounds $[B_l, B_u]$ are varied to control what percentage of repair candidates have to be processed by Brute Force (BF) and Full Cluster Filtering (FF) to determine the top-k repairs as explained in Section 3.5. For example, $\omega_1^{d=38}$ in Figure 7.2a for Q_1 is the constraint ω_1 from Table 3.2 with the bounds set such that 38% of the candidate solutions have to be explored by these algorithms. The thesis uses exploration distance (ED) to refer to this. Cluster Range Pruning (RP) (pink bars) generally outperforms Full Cluster Filtering (FF) (blue bars) for most settings. In Figure 7.2b, two algorithms exhibit similar performance for Q_4 with ω_3, where solutions are found after exploring only 2% and 3% of the search space.



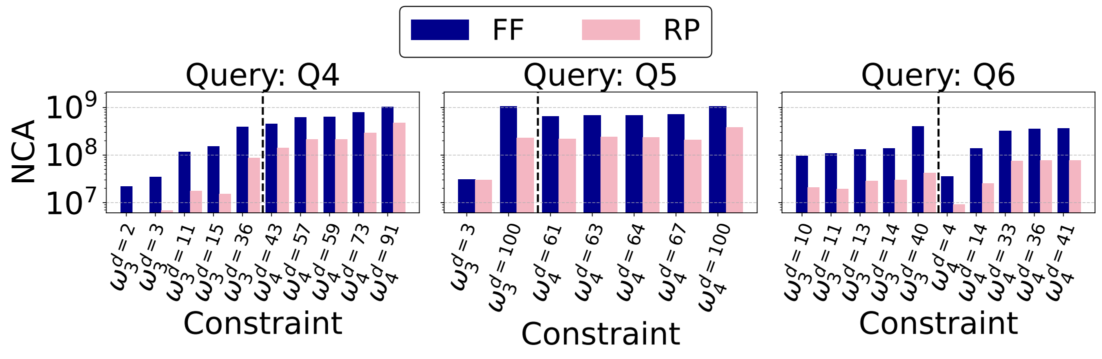
(A) Total number of constraints evaluated (NCE) - Healthcare dataset.



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



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



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

FIGURE 7.3: Runtime, NCE, and NCA for FF and RP over the Healthcare and ACSIncome datasets using the queries from Table 3.1.

A similar pattern is observed for Q_5 with $\omega_3^{d=3}$ and Q_6 with $\omega_4^{d=4}$. The relationship between the exploration distance (ED) and the runtime of the proposed algorithms is further investigated in Section 7.2. In general, Cluster Range Pruning (RP) significantly outperforms Full Cluster Filtering (FF), demonstrating an improvement of about an order of magnitude due to its capability of pruning and confirming sets of candidate repairs at once.

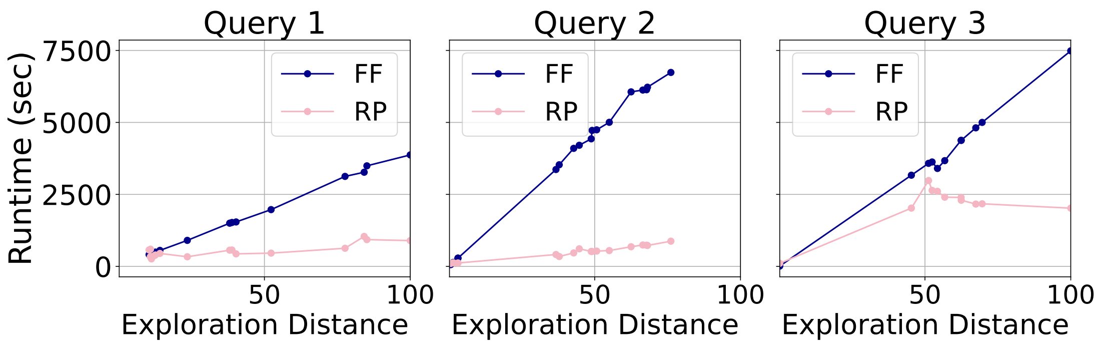
Total number of candidates evaluated (NCE). How number of candidates evaluated (NCE) affects the performance of the proposed methods is further analysed . Figures 7.3a and 7.3b shows the result of the number of candidates evaluated (NCE) (on the y axis) for Full Cluster Filtering (FF) and Cluster Range Pruning (RP) on Healthcare and ACSIncome, respectively. Cluster Range Pruning (RP) consistently checks fewer candidates compared to Full Cluster Filtering (FF), because it can reason over sets of candidate repairs at a time, allowing it to accept or prune entire groups of candidates collectively based on interval bounds, rather than evaluating each candidate individually.

As observed in the runtime evaluation, the exploration distance (ED) impacts the efficiency of the proposed algorithms, producing comparable results for Full Cluster Filtering (FF) and Cluster Range Pruning (RP) when a small number of candidates has to be explored, e.g., as shown in Figure 7.3b for Q_4 with $\omega_3^{d=2}$ and Q_5 with $\omega_3^{d=3}$.

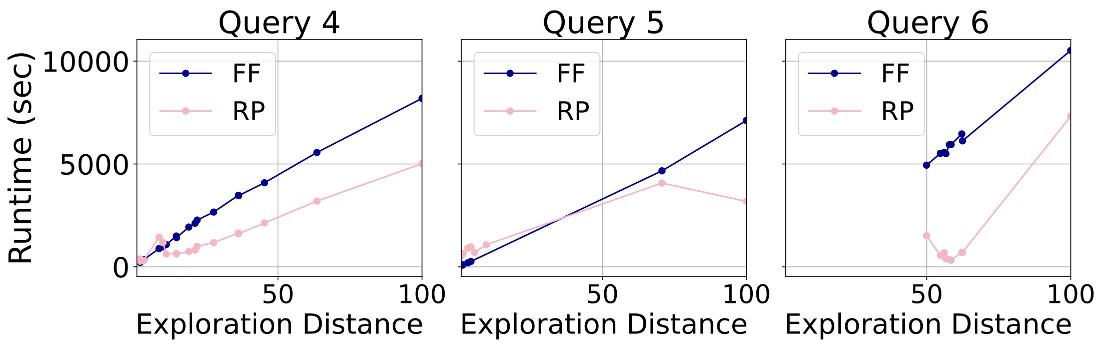
Total Number of Cluster Accessed (NCA). The results of the number of clusters accessed are shown in Figures 7.3c and 7.3d for Healthcare and ACSIncome, respectively. Similarly, Cluster Range Pruning (RP) accesses significantly fewer clusters than Full Cluster Filtering (FF), highlighting its efficiency in limiting the exploration of the search space. Furthermore, it follows the same trend of the previous evaluation results such that the benefit of Cluster Range Pruning (RP) becomes negligible and may even reverse when the proximity of solutions is low.

7.2 Performance-Impacting Factors

To gain deeper insights into the behaviour observed in Section 7.1, the relationship between the exploration distance (ED) and performance is investigated. Additionally, the performance of Full Cluster Filtering (FF) and Cluster Range Pruning (RP) is evaluated in terms of the parameters from Section 3.5. The Healthcare, ACSIncome, and TPC-H datasets are used.



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

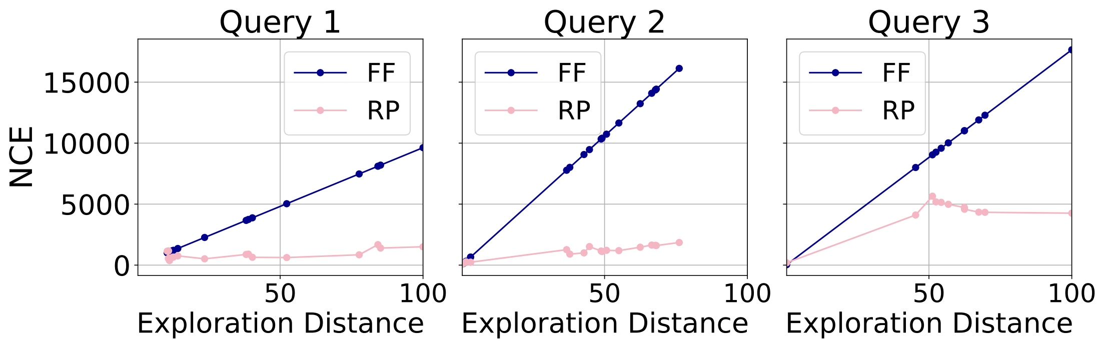


(B) Runtime of FF and RP, varying ED - ACSIncome dataset.

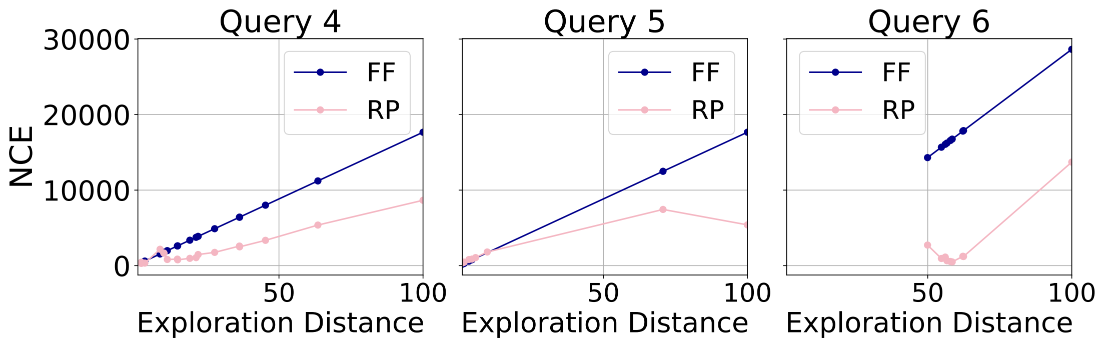
FIGURE 7.4: Runtime, NCE, and NCA for FF and RP over the Healthcare and ACSIncome datasets, varying ED.

7.2.1 Effect of Exploration distance

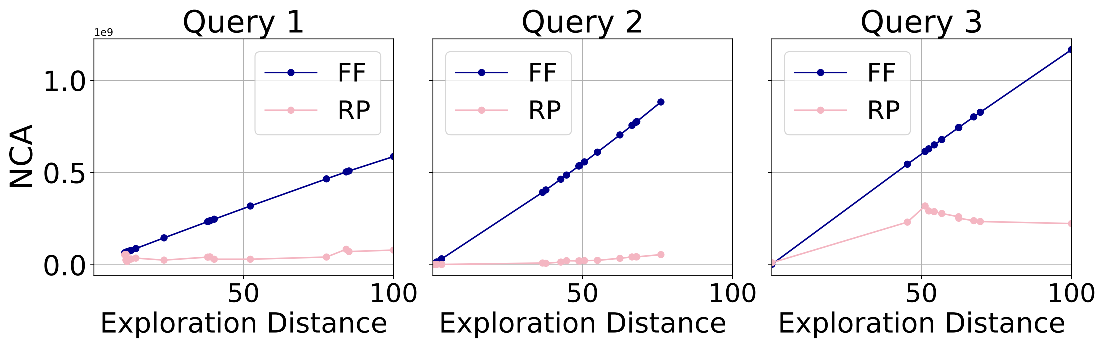
Queries Q_1–Q_3 and the constraint ω_1 on Healthcare are used and Q_4–Q_6 and the constraint ω_3 on ACSIncome and vary the bounds to control for exploration distance (ED). The result is shown in Figure 7.4a for Healthcare and in Figure 7.4b for ACSIncome. For Q_1 and Q_2, when exploration distance (ED) 10% or less, Full Cluster Filtering (FF) and Cluster Range Pruning (RP) exhibit comparable performance. A similar pattern is seen for Q_3, where Full Cluster Filtering (FF) performs better than Cluster Range Pruning (RP) for lower exploration distance (ED), but Cluster Range Pruning (RP) outperforms Full Cluster Filtering (FF) for exploration distance (ED) $> 50\%$ as shown in Figure 7.4a. The same trend holds for Q_4 and Q_5, while for Q_6, Cluster Range Pruning (RP) consistently outperforms Full Cluster Filtering (FF) for higher exploration distance (ED), as illustrated in Figure 7.4b. The number of candidates evaluated (NCE) and number of clusters accessed (NCA) follow similar patterns to runtime. For exploration distance (ED) $> 50\%$, Cluster Range Pruning (RP) significantly reduces both number of candidates evaluated (NCE) and number of clusters accessed (NCA). However, when exploration distance (ED) $< 10\%$, the difference between the two algorithms diminishes, with both performing similarly. These trends are shown in Figure 7.5c and Figure 7.5d for number of clusters accessed



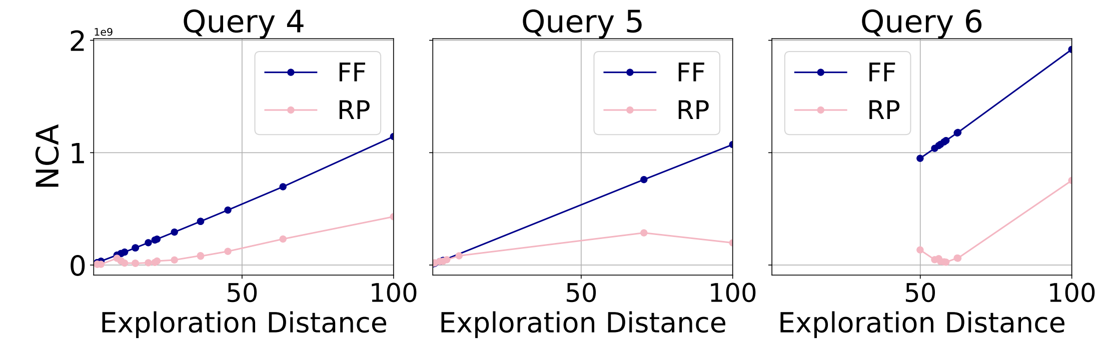
(A) Total number of constraints evaluated (NCE) - Healthcare dataset.



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



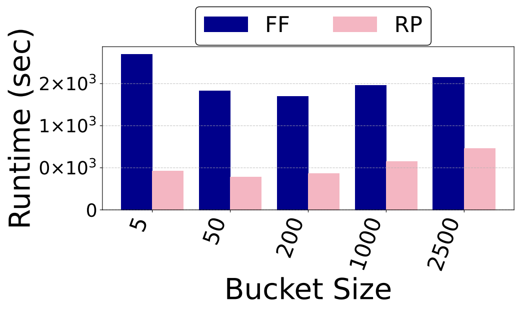
(C) Total number of cluster accessed (NCA) - Healthcare dataset.



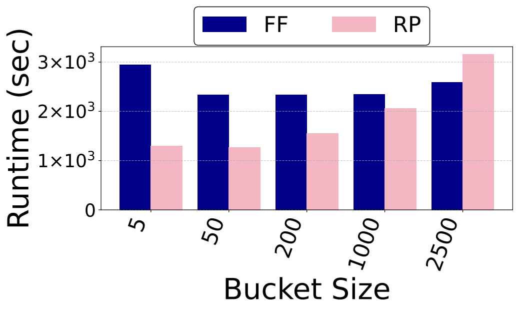
(D) Total number of cluster accessed (NCA) - ACSIncome dataset.

FIGURE 7.5: Runtime, NCE, and NCA for FF and RP over the Healthcare and ACSIncome datasets, varying ED.

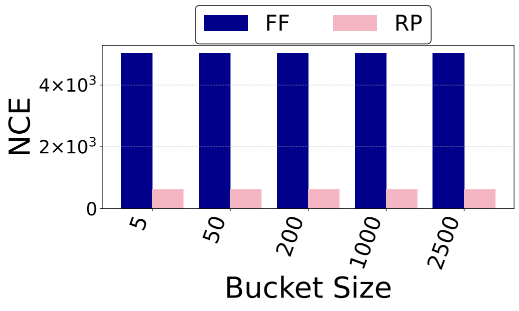
(NCA), and in Figure 7.5a and Figure 7.5b for number of candidates evaluated (NCE). The reason behind these trends is that when solutions are closed to the user query (smaller exploration distance (ED)), then there is a lower chance that Cluster Range Pruning (RP) can prune larger sets of candidates at once.



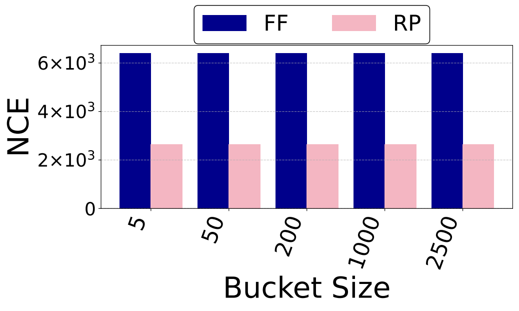
(A) Runtime - Healthcare dataset.



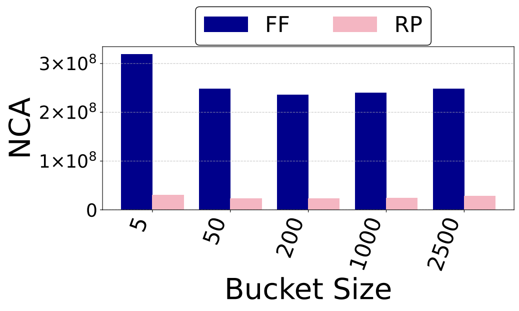
(B) Runtime - ACSIncome dataset.



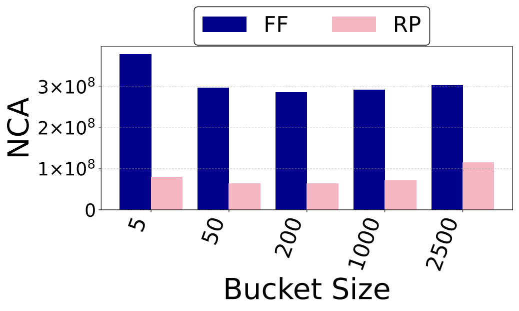
(C) Total number of constraints evaluated (NCE) - Healthcare dataset.



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



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



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

FIGURE 7.6: Runtime, NCE, and NCA for FF and RP over the Healthcare and ACSIncome datasets, varying bucket size $\mathcal{S}$.

7.2.2 Effect of Bucket Size

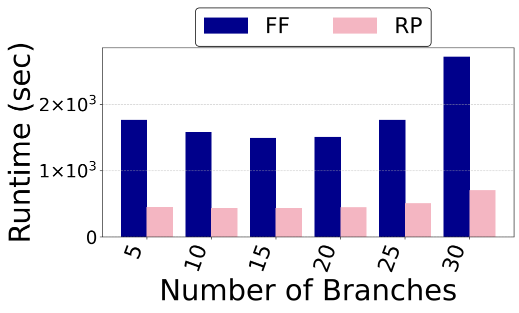
The runtime of Full Cluster Filtering (FF) and Cluster Range Pruning (RP) is now evaluated varying the bucket size $\mathcal{S}$ using Q_1 with ω_1 using bounds $[0.44, 0.5]$ for the Healthcare dataset and Q_4 with ω_3 using bounds $[0.34, 0.39]$ for the ACSIncome dataset. The $\mathcal{S}$ is varied from 5 to 2500. Using the default branching factor $\mathcal{B}$ of 5, the structure of the kd-tree for this evaluation is as follows: (i) Level 1: 5 clusters, each with 10,000 data points; (ii) Level 2: 25 clusters, each with 2,000 data points; (iii) Level 3: 125 clusters, each with 400 data points; (iv) Level 4: 525 clusters, each with 80 data points; (v) Level 5: 3,125 clusters, each with 16 data points; (vi) Level 6: 15,625 clusters, each with 3 or 4 data points. Note that the algorithms will generate kd-tree up to the level where the capacity of each cluster at that level is less than or equal to $\mathcal{S}$. For example, for $\mathcal{S} = 200$, the tree will have 4 levels. The results of the runtime are shown in Figure 7.6a and Figure 7.6b. Similarly, the number of clusters accessed (NCA) as shown in Figure 7.6e and Figure 7.6f exhibit the same trend as the runtime. The advantage of smaller bucket sizes is that it is more likely that a cluster that is fully covered / not covered at all can be found. However, this comes at the cost of having to explore more clusters. For number of candidates evaluated (NCE), as shown in Figure 7.6c and Figure 7.6d, 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, $\mathcal{S} = 15$ was identified to yield robust performance for a wide variety of settings and use this as the default.

Table 7.1: Branching Configuration and Data Distribution

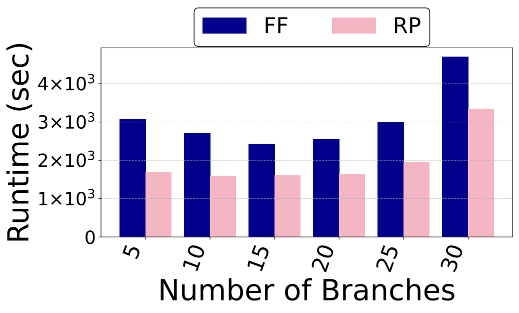
# of Branches	**# of Leaves**	**# of Branches**	**# of Leaves**
5	15625	20	8000
10	10000	25	15625
15	3375	30	27000

7.2.3 Effect of the Branching Factor

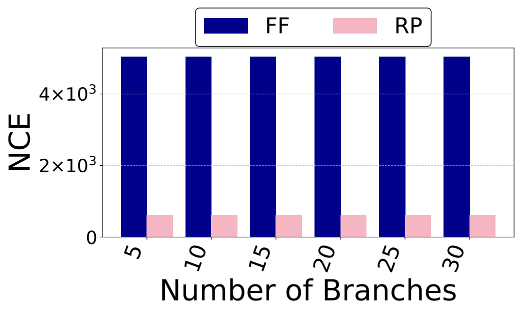
The relationship between the branching factor $\mathcal{B}$ and the runtime of the Full Cluster Filtering (FF) and Cluster Range Pruning (RP) is now examined. The same queries, constraints, bounds, and datasets are used as in the previous evaluation. In this experiment, The $\mathcal{B}$ from 5 to 30 is varied. The corresponding number of leaf nodes in the kd-tree is shown in Table 7.1. As the default bucket size $\mathcal{S} = 15$ is used, the branching factor affects the depth of the tree. The result shown in Figure 7.7a and Figure 7.7b confirms that the performance of Full Cluster Filtering (FF) and Cluster Range Pruning (RP) correlates with the number of clusters at the leaf level. For the Full Cluster Filtering (FF), the branching factors of 5 and 25 yield nearly identical runtime because both have the same number of leaves (15,625). A similar pattern can be observed for $\mathcal{B} = 10$ and $\mathcal{B} = 20$. At $\mathcal{B} = 15$, Full Cluster Filtering (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 Full Cluster Filtering (FF). This behavior is explained by the nature of Full Cluster Filtering (FF), which evaluates each candidate repair by aggregating over clusters. Fewer clusters reduce overhead, as each repair interacts with fewer



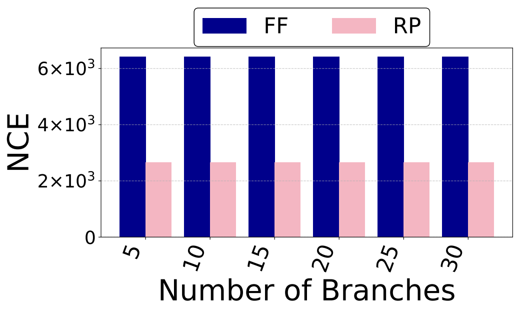
(A) Runtime - Healthcare dataset



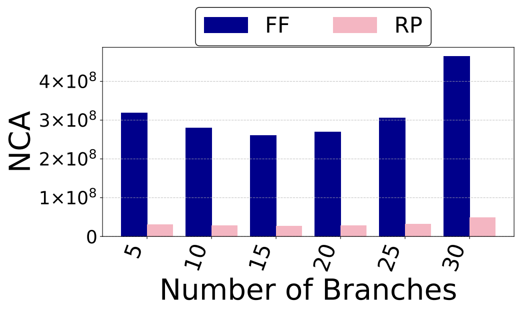
(B) Runtime - ACSIncome dataset



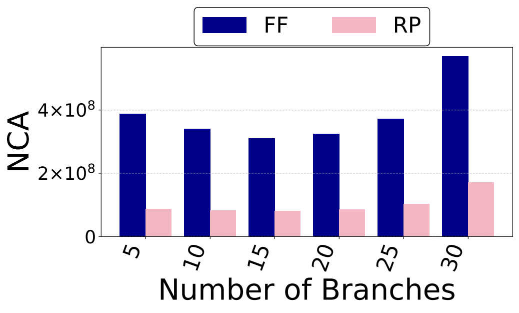
(C) Total number of constraints evaluated (NCE) - Healthcare dataset



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



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

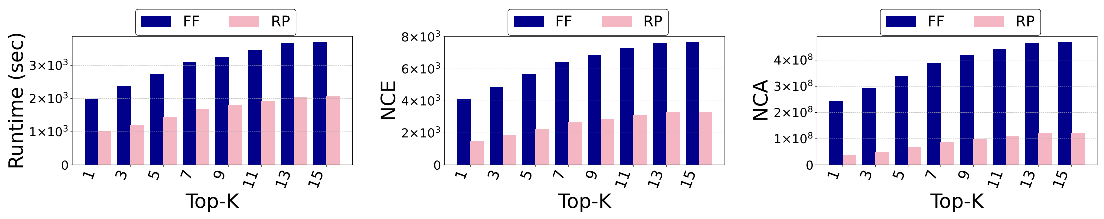


(F) Total number of cluster accessed (NCA) - ACSIncome dataset

FIGURE 7.7: Runtime, NCE, and NCA for FF and RP over the Healthcare and ACSIncome datasets, varying the number of branches $\mathcal{B}$.

segments of data. However, if the number of clusters becomes too large (as $\mathcal{B} = 30$), the overhead increases substantially due to more evaluations across more leaf nodes.

These results also demonstrate that the tree depths only has negligible impact on the runtime. Similarly, the number of clusters accessed (NCA) as shown in Figure 7.7e and Figure 7.7f exhibit the same trend as the runtime. For number of candidates evaluated (NCE), as shown in Figure 7.7c and Figure 7.7d, 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 Cluster Range Pruning (RP), overall performance trends align with those of Full Cluster Filtering (FF). However, Cluster Range Pruning (RP) is less influenced by the branching factor. This is because Cluster Range Pruning (RP) reasons over entire intervals of repair candidates, and can prune or accept whole



(A) Runtime (sec).

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

(C) Total number of clusters accessed (NCA).

FIGURE 7.8: Runtime, NCE, and NCA for FF and RP over top-k.

regions of the search space. As a result, even when the tree contains more clusters, Cluster Range Pruning (RP) can eliminate many of them in bulk without individual examination, thus reducing the impact of larger $\mathcal{B}$.

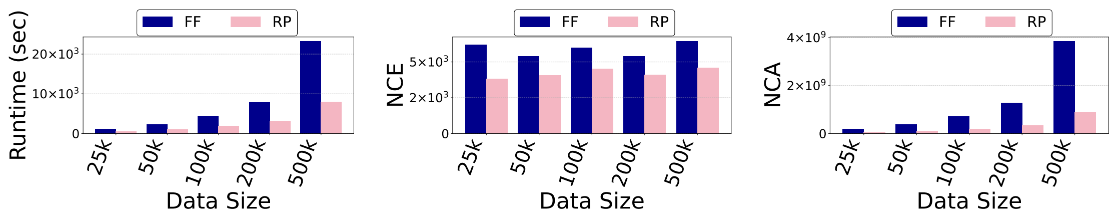
7.2.4 Effect of k

In this experiment, the parameter k is varied from 1 to 15. For both Full Cluster Filtering (FF) and Cluster Range Pruning (RP), as k increases, the runtime also increases, as shown in Figure 7.8a. This behavior is expected since finding a single repair (k=1) requires less computation than identifying multiple repairs. When k is larger, the algorithm must explore a larger fraction of the search space to find additional repairs. Similarly, both the number of candidates evaluated (NCE) as shown in Figure 7.8b and number of clusters accessed (NCA) as shown in Figure 7.8c exhibit the same increasing trend. This pattern is observed consistently across both Full Cluster Filtering (FF) and Cluster Range Pruning (RP), reinforcing the intuition that retrieving more solutions requires higher computational effort. Cluster Range Pruning (RP) consistently outperforms Full Cluster Filtering (FF).

7.2.5 Effect of Dataset Size

Next, the dataset size is varied and the runtime, number of candidates evaluated (NCE) and the number of clusters accessed (NCA) are measured for the tpch dataset, Q_7 from Table 3.1, ω_5 from Table 3.2. Default settings for all parameters (Section 3.5) are used. As shown in Figure 7.9a, as the data size increases, the runtime also increases. Dataset size impacts both the size of the search space and the size of the kd-tree. Nonetheless, the proposed algorithms scale roughly linearly in dataset size demonstrating the effectiveness of using materialized aggregation results for clusters and range-based pruning of candidate repair sets. This is further supported by the number of clusters accessed (NCA) measurements shown in Figure 7.9b, which exhibit the same trend as the runtime. For number of candidates evaluated (NCE), as shown in Figure 7.9c, 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 7.6 and Figure 7.7, where the number of candidates evaluated (NCE) remained constant due to the data being fixed across configurations. These results confirm that the number of candidates evaluated (NCE) is influenced by changes in the dataset content, rather than by variations in the branching factor $\mathcal{B}$ or bucket size $\mathcal{S}$ alone.



(A) Runtime (sec).

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

(C) Number of clusters accessed (NCA).

FIGURE 7.9: Runtime, NCE, and NCA for FF and RP over the TPC-H dataset, varying data size.

7.3 Comparison with Related Work

The proposed approaches Full Cluster Filtering (FF) and Cluster Range Pruning (RP) are compared with Erica (Li et al., 2023), 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 aggregate constraint (AC)s supported in this work. Erica returns all repairs that are not dominated by any other repair where a repair dominates another repair if it is at least as close to the user query for every condition θ_i and strictly closer in at least one condition. That is, Erica returns the skyline (Borzsony et al., 2001). Thus, different from the proposed approaches, the number of returned repairs is not an input parameter in Erica. For a fair comparison, the minimal repairs are determined and then set k such that the proposed methods returns a superset of the repairs returned by Erica. To conduct the evaluation for Erica, the available Python implementation is used (`https://github.com/JinyangLi01/Query_refinement`).[1] The queries,

[1]To enable a fair comparison between the proposed algorithm and Erica, the code was modified so that both systems evaluate constraints using the same data processing framework. Specifically, the original implementation of the proposed algorithm relied on pure Python for DataFrame operations, whereas Erica leveraged Pandas, which is backed by highly optimized C code. Comparing the two directly would conflate algorithmic effectiveness with differences in execution speed between Python and C. For this reason, Erica was re-implemented to use similar data structures to those employed in the proposed algorithm. While this inevitably reduces the performance of Erica relative to its original form, the purpose of this modification is not to demonstrate superiority in runtime, but rather to allow a fair assessment of the two approaches under equivalent conditions. The novel approach also addresses different problem settings compared to Erica, and future work on optimizing its implementation (e.g., in C++ or other lower-level languages) may further improve performance. Thus, the comparison here should be interpreted as showing broadly comparable behavior under consistent conditions, rather than as evidence of one system being strictly faster than the other.

constraints, and the dataset from (Li et al., 2023) were adopted. The generated refinements and runtime of the techniques in this work are compared with Erica using Q_1 and Q_2 (Table 3.1) on the Healthcare dataset (size 50K) with constraints Ω_6 and Ω_7 (Table 3.2), respectively.

Comparison on the Generated Repairs. A comparison is first conducted between the repairs generated by the proposed approaches and Erica. For Q_1 with Ω_6, Erica generates 7 minimal repairs whereas the the technique of this work generates 356, including those produced by Erica. Similarly, for Q_2 with Ω_7, Erica generates 9 minimal repairs while Full Cluster Filtering (FF) and Cluster Range Pruning (RP) approaches generates 1035, including those produced by Erica. In general, Full Cluster Filtering (FF) and Cluster Range Pruning (RP) techniques successfully generate all repairs produced by Erica. Note that the top-1 repair returned by Full Cluster Filtering (FF) and Cluster Range Pruning (RP) is guaranteed to be minimal (not dominated by any other repair). However, the remaining minimal repairs returned by Erica may have a significantly higher distance to the user query than the remaining top-k answers returned by Full Cluster Filtering (FF) and Cluster Range Pruning (RP). For example, in Q_1, given the condition `income >= 200k` of the user query, the second solution of Full Cluster Filtering (FF) and Cluster Range Pruning (RP) includes a refined condition `income >= 300k` whereas Erica provides a refinement `income >= 317k` which is far from the user query. As mentioned above, to ensure that the proposed approach returns a superset of the solutions returned by Erica's solutions, the parameter k was adjusted per query and constraint set to ensure that.

Table 7.2: Actual repaired queries generated by Erica for Q_1 with Ω_6.

Repairs	**Repaired Predicate Values** [income, num_children, county]
1	[298k, 5, 1]
2	[317k, 4, 1]
3	[424k, 5, 3]
4	[328k, 3, 1]
5	[414k, 3, 2]
6	[433k, 3, 3]
7	[328k, 5, 2]

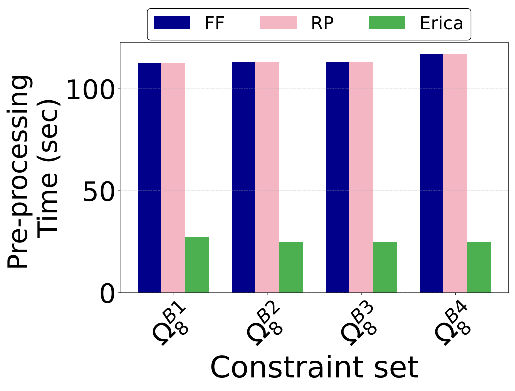
Detailed Comparison for Q_1 with Ω_6. To provide a clearer illustration of the comparison between Erica and the proposed approaches, Table 7.2 lists the *7 minimal actual repaired queries* produced by Erica, while Table 7.3 presents top-34 repairs generated by Full Cluster Filtering (FF) and Cluster Range Pruning (RP) for the same query and constraint.

The observed differences in the total number and ordering of repairs arise primarily from the distinct optimization criteria employed by the two approaches. The highlighted repairs in Tables 7.2 and 7.3 demonstrate that all of Erica's repaired

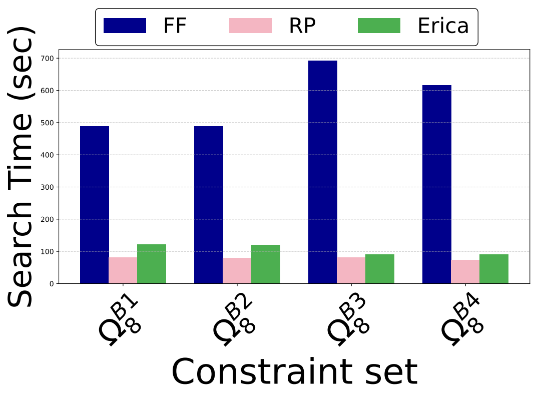
TABLE 7.3: Top repaired queries generated by the proposed approach for Q_1 with Ω_6 (highlighted repairs are also produced by Erica).

Rank	**Repaired Predicate Values** [income, num_children, county]
1	[298k, 5, 1]
2	[300k, 4, 1]
3	[301k, 5, 1]
4	[302k, 5, 1]
5	[303k, 5, 1]
6	[305k, 5, 1]
7	[308k, 5, 1]
8	[314k, 5, 1]
9	[317k, 4, 1]
10	[317k, 5, 1]
11	[319k, 4, 1]
12	[319k, 5, 1]
13	[320k, 4, 1]
14	[320k, 5, 1]
15	[321k, 4, 1]
16	[321k, 5, 1]
17	[322k, 4, 1]
18	[323k, 3, 1]
19	[322k, 5, 1]
20	[323k, 4, 1]
21	[323k, 5, 2]
22	[324k, 3, 1]
23	[323k, 5, 1]
24	[324k, 4, 1]
25	[324k, 5, 2]
26	[324k, 5, 1]
27	[326k, 3, 1]
28	[326k, 4, 1]
29	[326k, 5, 2]
30	[326k, 5, 1]
31	[328k, 3, 1]
32	[328k, 4, 1]
33	[328k, 5, 2]

queries appear among those generated by the proposed approaches. Specifically, the **first repair of Erica** (`[298k, 5, 1]`) exactly matches the **first repair** returned by the proposed approach. The **second repair of Erica** (`[317k, 4, 1]`) corresponds to the **ninth repair** in the proposed results. Similarly, the **fourth and seventh repairs of Erica** (`[328k, 3, 1]` and `[328k, 5, 2]`) appear as the **31st** and **33rd** repairs, respectively, in the proposed approach's ranked list. The remaining Erica repairs (`[424k, 5, 3]`, `[414k, 3, 2]`, and `[433k, 3, 3]`) are also included among the 356 total repairs generated by the proposed techniques, but occur later in the ranking (e.g.,



(A) Pre-processing



(B) Search

FIGURE 7.10: Runtimes of FF, RP, and Erica.

the third repair of Erica appears as the 245th in the proposed results).

This detailed alignment confirms that the proposed approach successfully reproduces all valid repairs generated by Erica. Moreover, the differing positions of the matching repairs reflect the distinct optimization strategies used: Erica returns all repairs that are not dominated by any other repair, whereas the proposed algorithms rank repairs by proximity to the original user query.

Runtime Comparison. The experiment utilizes Q_4 with Ω_8 on the 50K ACSIncome dataset, which is derived from Erica's dataset, query, and constraint. The same bounds in the constraints for both Erica and the proposed approach are used: $B1 := (B_{u_1} = 30, B_{u_2} = 150, B_{u_3} = 10)$ 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, the number of generated repairs (i.e., Top-k) is fixed in the proposed approach to equal to the number of repairs produced by Erica. The parameter k was set to 17 for constraint sets Ω_8^{B1} and Ω_8^{B2}, 11 for Ω_8^{B3}, and 13 for Ω_8^{B4}.

Due to the different optimization criterions, variations in the generated repairs between the proposed approach and Erica are expected. The results in Figure 7.10b reveal an advantage of the Cluster Range Pruning (RP) algorithm, which outperforms Erica in search time which is the time of exploring the search space to generate a repair. However, as shown in Figure 7.10a, in pre-processing time which is the time of materializing aggregates and constructing the kd-tree for the proposed methods and generating provenance expressions for Erica, Erica outperforms both Cluster Range Pruning (RP) and Full Cluster Filtering (FF), indicating that Erica summarizes information needed for the search process more effectively. The reason for this is that Erica only needs to compute the provenance expressions and, for each predicate, generate a list of candidate constants sorted by their distance from the original constant in the user's query. In contrast, the proposed algorithms must perform additional steps such as clustering the data, indexing the clusters, and materializing summaries for each cluster, operations that are significantly more computationally intensive.

Overall, the total runtime of Cluster Range Pruning (RP) and Erica are comparable, despite the fact that the proposed approach does not incorporate specialized optimizations that leverage the monotonicity of the constraints handled by Erica. Notably, the proposed framework supports a strictly more expressive class of constraints: group cardinality constraints, as addressed by Erica, are a subset of the filter-aggregate constraints expressible using the `count` function in the proposed approach. Moreover, the proposed approach enables constraints defined as arithmetic combinations of multiple filter-aggregate query results and supports aggregation functions beyond `count`, which are necessary to represent fairness metrics such as Statistical Parity Difference (SPD). These findings also underscore the importance of the Cluster Range Pruning (RP) approach, as the Full Cluster Filtering (FF) approach performs significantly slower than Erica.

7.4 Summary

This chapter evaluates the performance of the proposed query repair methods under non-monotone aggregate constraints. It begins by comparing against the brute-force Brute Force (BF), demonstrating that BF is unsuitable for practical use due to its inefficiency. Then, it compares Full Cluster Filtering (FF) against Cluster Range Pruning (RP) across real-world (Healthcare and ACSIncome) and TPC-H datasets, showing that Cluster Range Pruning (RP) often achieves an order-of-magnitude improvement in runtime, candidate evaluations (NCE), and cluster accesses (NCA) as exploration distance grows, while both methods perform similarly when exploration distances are very small. Key parameters: exploration distance, bucket size, branching factor, top-k, and dataset size are varied. Smaller buckets help Cluster Range Pruning (RP) prune more effectively (though very large buckets reduce pruning), branching factor mainly affects Full Cluster Filtering (FF) and Cluster Range Pruning (RP) via the number of clusters at the leaf level, and both methods scale roughly linearly with top-k and data size. To set bucket size $\mathcal{S}$ and branch factor $\mathcal{B}$ in practice, it is advisable to start with moderate values. If the resulting kd-tree contains too many leaf clusters, one should consider increasing $\mathcal{S}$ or decreasing $\mathcal{B}$. Conversely, if pruning is found to be ineffective, decreasing $\mathcal{S}$ or increasing $\mathcal{B}$ may improve performance. This tuning strategy helps balance runtime with pruning effectiveness based on the structure of the resulting tree.

It also benchmarks against Erica (Li et al., 2023) under cardinality constraints (a special case of the aggregate constraint (AC) supported in this work), showing that the proposed methods generate a superset of Erica's minimal refinements. In search time, Cluster Range Pruning (RP) outperforms Erica's searching time for repairs, although Erica's preprocessing can be faster; overall runtimes are comparable, but Cluster Range Pruning (RP) supports a far broader class of non-monotone aggregates. These findings underscore that Cluster Range Pruning (RP) is critical for constraints beyond simple cardinality. Practitioners are advised to prefer Cluster Range Pruning (RP) when repairs lie farther from the original query.

Chapter 8

Conclusions

This thesis provides foundational work to solve the problem of repairing user queries to satisfy constraints expressed as arithmetic combinations of aggregates over query results. The work begins by surveying the related work in Chapter 2. Then Chapter 3 introduces the overall methodological framework adopted to solve the problem. After that, the formal definition of query repair under constraints involving arithmetic combinations of aggregate functions is presented in Chapter 4. In addition to the brute force method, this thesis presents two algorithms, Full Cluster Filtering (FF) in Chapter 5 and Cluster Range Pruning (RP) in Chapter 6, that solve the aggregate constraint repair problem. Finally, an extensive experimental evaluation over multiple datasets, queries and constraints is presented in Chapter 7.

The contributions of this thesis are motivated by a growing gap between the technical capabilities of SQL-based data systems and the complex, real-world constraints that organizations increasingly seek to enforce over query results. For example, Balayn et al. (2021) argue that fairness and bias constraints should be incorporated directly into database management systems, rather than being addressed only in later stages of data analysis. Similarly, research on fairness-aware range queries has begun exploring database-level support for demographic balance and unbiased data selection (Shetiya et al., 2022). While SQL is highly effective for extracting data based on precise conditions, it lacks mechanisms to ensure that results satisfy complex and real-world properties such as fairness, proportionality, or compliance with policy constraints. This gap is particularly visible in domains like healthcare, finance, and recruitment, where decision outcomes must reflect not only technical correctness but also ethical and regulatory considerations.

Although prior work on query refinement under aggregate constraints such as the works by Shetiya et al. (2022); Li et al. (2023) has made meaningful progress particularly in the context of fairness and diversity enforcement, existing methods are typically limited to simple `count` based constraints and assume monotonicity to enable efficient pruning of the search space. These assumptions make such techniques unsuitable for more complex scenarios involving non-monotonic arithmetic expressions over aggregate functions. As such, they cannot accommodate many real-world constraints that require richer expressiveness. This thesis addresses these limitation by introducing the first query repair techniques Full Cluster Filtering (FF) and Cluster Range Pruning (RP) capable of handling non-monotone aggregate constraints that aim to fill this gap by enabling SQL queries to be automatically repaired when they violate aggregate constraints. These constraints are expressed as arithmetic combinations of multiple aggregate-filter queries and can capture a wide range of application needs, from demographic fairness to sourcing compliance. Unlike prior work which primarily supports monotone constraints, this thesis supports non-monotonic expressions, significantly broadening the scope of query repair.

A critical insight from this work is that, despite the complexity introduced by supporting non-monotonic aggregate constraints, there are structural properties in the space of repair candidates that can be exploited to make computation more tractable. In particular, this thesis finds that many candidate repairs are similar in structure, often differing only slightly in the constants used in their predicates which leads to shared computation when evaluating aggregate constraints. Rather than treating each candidate in isolation, the Full Cluster Filtering (FF) can benefit from shared evaluation strategies. This motivates the use of a kd-tree to cluster the dataset and materialize aggregate summaries for each region. This structure enables efficient reuse of computations: clusters where all tuples either satisfy or violate a predicate can be accepted or skipped wholesale, avoiding unnecessary re-computation across similar candidates. Moreover, this similarity between candidates can be leveraged more aggressively by reasoning over entire ranges of repairs at once. Cluster Range Pruning (RP) approach uses interval arithmetic to enable conservative but effective approximation of constraint outcomes for sets of candidate repairs. In practice, this means that instead of evaluating each repair individually, the algorithm can often prune or accept whole groups of candidates in a single step. These strategies led to significant performance improvements in practice, especially when many repair candidates were similar to each other.

The techniques proposed in this thesis should be preferred in scenarios where the constraints are complex, non-monotone, or involve arithmetic combinations of multiple aggregate functions. Such cases arise frequently in fairness-aware data analysis, compliance-driven reporting, and domains where multiple aggregate conditions interact in non-trivial ways. In particular, the Cluster Range Pruning (RP) approach demonstrates strong advantages when the search space is large, since its ability to reason over ranges of repair candidates allows substantial pruning. In practice, the choice of repair strategy should depend on both the nature of the constraints and the underlying data distribution. For example, when user constraints are relatively simple and monotone, prior techniques such as Li et al. (2023) may suffice and offer more lightweight performance. By contrast, when non-monotonicity or fairness-related measures such as Statistical Parity Difference (SPD) are involved, the methods in this thesis become essential to obtain valid and meaningful repairs. This suggests the potential for hybrid systems that analyse the constraints in advance of execution: such systems could dynamically decide whether to apply a simpler refinement approach like Li et al. (2023); Shetiya et al. (2022) or to leverage the more powerful, though computationally heavier, methods introduced in this thesis.

However, for non-monotone constraints, the solution space can grow exponentially, and even small changes in data statistics can have disproportionate effects on solution validity. Despite these challenges, certain characteristics in real-world instances often favor the performance of Full Cluster Filtering (FF) and Cluster Range Pruning (RP): (i) **Cluster homogeneity:** clusters with tight bounds that lie entirely above or below candidate thresholds allow for rapid accept/reject decisions without further subdivision. (ii) **Extreme search regions:** large portions of the candidate space are clearly inside or outside repaired predicate ranges, which allows entire ranges to be accepted or rejected in a single step. (iii) **Constraint's attributes correlation:** strong correlation between predicate attributes and the attributes of the aggregations used in the constraint lead to tighter bounds for the aggregate statistics of clusters which in turn lead to tighter bounds on the results of the constraint. These patterns suggest that, while worst-case performance is expensive, the algorithm performs well in many practical scenarios.

The comparison with Erica by Li et al. (2023), the work most closely related to this thesis, further highlights the contributions and trade-offs of this thesis. Erica is limited to a specific class of constraints which are group cardinality constraints that evaluate the `count` over the results of the query. While it is efficient for this restricted setting and benefits from optimizations tied to monotonicity, it does not support broader classes of constraints required by real-world properties such as fairness. In contrast, the proposed approach supports a strictly more expressive class: any group cardinality constraint expressible in Erica can also be modelled in the proposed approach as a filter-aggregate query with a `count` aggregation. Beyond this, the methods presented in this thesis can handle arithmetic combinations of multiple filter-aggregate results, and support aggregation functions beyond `count`, such as those needed to express fairness metrics like statistical parity difference (SPD). Although the algorithms were formally presented for a single constraint, the extension to conjunctions of constraints is straightforward: each constraint in the conjunction is evaluated independently, and their results are combined using logical conjunction. This broader expressiveness allows the proposed algorithms not only to subsume Erica's constraint model but to address more complex, real-world use cases where multiple aggregate-based criteria interact.

In summary, this thesis demonstrates that query repair under non-monotone aggregate constraints is both feasible and practically effective. By combining clustering with interval-based pruning, it introduces the first framework capable of ensuring that SQL queries meet complex real-world requirements such as fairness and compliance. These methods are efficient enough for deployment in interactive systems, enabling analysts to iteratively refine queries while adhering to organizational and regulatory constraints. Ultimately, this work lays the foundation for future systems that support the complex social, ethical, and business constraints of today's data-driven world.

The following sections present the limitations of the proposed approaches and outline promising directions for future research, concluding with a brief summary of the overall study.

8.1 Limitations and Future Work

While this thesis lays a solid foundation for repairing queries under constraints expressed as arithmetic combinations of aggregates functions over query results, several limitations remain and suggest promising directions for further investigation.

First, the current repair framework is limited to modifying the constants within selection predicates. In real-world scenarios, however, users may require more flexible forms of query repair, such as introducing or removing joins, altering logical operators, or modifying query structure altogether by introducing new predicates. Extending the repair framework to support such structural repairs would significantly broaden its applicability in practical settings.

Second, the query repair approach presented in this thesis relies on a basic distance metric that measures changes to predicate constants. While effective, this metric may not always reflect a user's true intent. For example, users may prefer repairs that preserve as many original result tuples as possible. Incorporating alternative optimization criteria such as user feedback, or minimal output divergence could make the repair process more aligned with practical analytical needs. This represents a valuable direction for future research.

Third, the use of interval-based bounds is only a first step toward efficient pruning strategy. Using more expressive domains than intervals for computing tighter bounds e.g., zonotopes (De Figueiredo and Stolfi, 2004) could provide more significant tighter bounds on aggregate values and constraints and thus improve pruning. Adapting the proposed algorithms to work with these richer domain is a valuable direction for future work.

Fourth, While the current implementation of repair framework assumes a static dataset, fixed predicates, a fixed aggregate constraint, and a fixed distance metric. In reality, any of these may change. There is a body of work on adjusting kd-tree (Bentley and Saxe, 1980; Bentley, 1975b); adapting those techniques to maintain the kd-tree in the proposed approach is a promising avenue. Thus, this work leaves comprehensive support maintenance under updates as an interesting and challenging direction for future work.

These future extensions would enhance the flexibility, adaptability, and effectiveness of the proposed query repair techniques. As data practitioners increasingly work under fairness, compliance, and representativeness constraints, expanding the capabilities of query repair systems will be critical for bridging the gap between technical queries and high-level organizational goals.

8.2 Summary of Contributions

This thesis establishes foundational techniques by addressing the problem of repairing a query to satisfy a constraint that evaluates a predicate over an arithmetic combination of the results of aggregation-filter queries evaluated on top of a user query result. Unlike prior work focused on repairing queries to meet cardinality constraints or constraints involving monotone aggregation functions, the proposed methods support non-monotone constraints, including fairness metrics such as statistical parity difference SPD. The exponential search space of candidate repairs is handled through two related optimizations: (i) materialization aggregation results for subsets of the input database in a kd-tree. Then these materialized results can be combined to determine the aggregation result for a repair candidate, effectively reusing computations across repair candidates; (ii) representation sets of repair candidates using bounds on the constants for each condition in the user query, and then derives bounds on the aggregate constraint results for these candidate sets using interval arithmetic (De Figueiredo and Stolfi, 2004). This enables the approach to confirm sets of candidates to be repairs or prune them at once. The results of the experiments demonstrate the efficiency of the proposed techniques across different datasets, queries, and constraints.

The work first formalizes query repair under the constraints of arithmetic combination of aggregation functions: given a user query whose selection predicates yield a result that violates an arithmetic constraint over aggregates, it seeks the top-k closest repairs to the user's predicates so that the repaired query satisfies the constraint. As shown in Chapter 4, the space of possible predicate changes grows exponentially, and the general repair problem is NP-hard, which motivates the need for more efficient solution strategies.

Building on this formulation,this work presents Full Cluster Filtering (FF) in Chapter 5, the thesis's first proposed algorithm for the aggregate constraint repair problem. Full Cluster Filtering (FF) uses kd-tree structure keyed on selection predicate attributes to partition the input dataset into set of clusters. For each cluster, it materializes two types of precomputed summaries: (i) the aggregate values required by the user's arithmetic constraint and (ii) lower and upper bounds on each attribute involved in the predicates. These summaries enable Full Cluster Filtering (FF) to determine (without revisiting every tuple in the dataset) whether all, none, or only some tuples in a cluster satisfy any given candidate predicate. The algorithm then reuses these summaries to avoid redundant aggregate evaluations across different repair candidates. Finally, Full Cluster Filtering (FF) ranks candidate repairs using the formally defined Repair Distance in Section 4.4 and returns the top-k modifications that both minimize distance to the original query and guarantee satisfaction of the specified aggregate constraint.

This approach's primary advantage over brute-force enumeration is that, by inspecting each cluster's precomputed attribute bounds, the algorithm can immediately accept all its tuples if they all satisfy a candidate's predicate, or skip the entire cluster if none do which eliminates the need to scan individual dataset tuples for every repair.

Then, this thesis introduces Cluster Range Pruning (RP), an enhanced algorithm that reasons about *ranges* of repair candidates (see Chapter 6). Whereas Full Cluster Filtering (FF) evaluates each candidate in isolation, Cluster Range Pruning (RP) groups candidates into intervals over their constant parameters and then prunes or accepts those intervals as a whole. As before, it uses a kd-tree to partition the dataset into clusters, but each cluster now stores the precomputed lower and upper bounds of every aggregate function required by the constraint (rather than individual constants). Given the bounds on aggregate functions within the constraint, the algorithm uses *interval arithmetic* which computes sound bounds for the result of arithmetic operations when the inputs (aggregate functions in this case) are bound by intervals. Specifically, the algorithm first computes the aggregate bounds for a cluster and then applies interval arithmetic to derive conservative bounds on the overall constraint expression for the entire interval of candidate repairs. Three outcomes are then possible:

1. If the bounds on the constraint expression for the entire interval of candidate repairs satisfies the given constraint threshold, then *every* candidate in the interval is guaranteed to satisfy the constraint. All candidates in this interval are accepted as repairs at once.
2. If the bounds on the constraint expression for the entire interval of candidate repairs violates the given constraint threshold, then *no* candidate in the interval can satisfy the constraint. The entire interval is discarded.
3. Otherwise, some candidates may satisfy the constraint and others may not, then the set of candidates may contain both valid and invalid repairs. In this case, the

algorithm subdivides the range into smaller intervals and repeats the same test on each subrange.

By operating on intervals rather than individual candidates and by exploiting precomputed cluster bounds, Cluster Range Pruning (RP) can confirms or eliminates large groups of candidates in a single step without individually evaluating them. The search terminates as soon as the first k valid repairs have been identified.

After proposing Full Cluster Filtering (FF) and Cluster Range Pruning (RP), an extensive experimental study is conducted across three real-world and benchmark datasets, a variety of queries, and diverse arithmetic aggregate constraints. The evaluation produced the following key insights:

- **Brute Force (BF)** Is unsuitable for practical use as it scans every tuple and evaluates each repair candidate individually.
- **Full Cluster Filtering (FF) and Cluster Range Pruning** Full Cluster Filtering (FF) reduces tuple's level evaluation by partitioning data by using clusters and pre-aggregating, yielding at least one order of magnitude in terms of runtime overBrute Force (BF). While Cluster Range Pruning further accelerates the search by reasoning over intervals of candidate repairs and pruning entire intervals at once. Full Cluster Filtering (FF) and Cluster Range Pruning (RP) exhibit comparable performance when exploration distance (ED) is small as when solutions are closed to the user query (smaller exploration distance (ED)), then there is a lower chance that Cluster Range Pruning (RP) can prune larger sets of candidates at once. As the exploration distance (ED) grows, Cluster Range Pruning (RP) consistently outperforms Full Cluster Filtering (FF) across varied queries, constraints and datasets.
- **Performance Impacting Factors** A detailed evaluation was conducted to assess how key parameters influence performance of the proposed algorithms:
  - **Exploration Distance (ED):** When the repairs are close to the original query (small exploration distance (ED)), both Full Cluster Filtering (FF) and Cluster Range Pruning (RP) perform similarly. As exploration distance (ED) increases, Cluster Range Pruning (RP) significantly outperforms Full Cluster Filtering (FF) by pruning large candidate intervals early.
  - **Bucket Size ($\mathcal{S}$):** Smaller bucket sizes ($\mathcal{S}$) make it more likely to find clusters that are fully covered or entirely excluded, which enhances the pruning effectiveness for Cluster Range Pruning. However, this comes at the cost of having to explore more clusters.
  - **Branching Factor ($\mathcal{B}$):** Performance correlates strongly with the number of clusters at the leaf level. Full Cluster Filtering (FF) is particularly sensitive to this, with runtime increasing as the number of leaf clusters grows. Cluster Range Pruning is more robust to changes in branching.

  - **Top-k Parameter:** Larger values of k increase runtime, number of candidates evaluated (NCE) and number of clusters accessed (NCA) evaluated for both algorithms. However, Cluster Range Pruning (RP) continues to outperform Full Cluster Filtering (FF) consistently.
  - **Dataset Size:** Both runtime and number of clusters accessed (NCA) scale roughly linearly with dataset size demonstrating the effectiveness of using materialized aggregation results for clusters and range-based pruning of candidate repair sets. For number of candidates evaluated (NCE) varies across different dataset sizes because the underlying data itself changes with the dataset size. Cluster Range Pruning (RP) continues to exhibit better scalability due to range-based pruning.
- **Comparison with state-of-the-art Erica:** When evaluated on the Erica benchmarks, Cluster Range Pruning (RP) achieves total runtimes comparable to those of Erica. As Cluster Range Pruning (RP) supports a far broader class of non–monotonic aggregate constraints, this underscores the critical role.

In general, Cluster Range Pruning (RP) is recommended over Full Cluster Filtering (FF) whenever repairs are expected to lie farther from the original query in the search space, due to substantial pruning benefits. For very small exploration distance (ED), Full Cluster Filtering (FF) remains a lightweight alternative. Together, these results demonstrate that the new algorithms (Full Cluster Filtering (FF) and Cluster Range Pruning (RP)) provide both the generality and the efficiency required to make the query repair for non-monotone aggregate constraint a practical tool in real database systems.

References

Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

Antragama Ewa Abbas, Wirawan Agahari, Montijn Van de Ven, Anneke Zuiderwijk, and Mark De Reuver. Business data sharing through data marketplaces: A systematic literature review. *Journal of Theoretical and Applied Electronic Commerce Research*, 16(7):3321–3339, 2021.

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

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

Abdullah M. Albarrak, Mohamed A. Sharaf, and Xiaofang Zhou. Saqr: An efficient scheme for similarity-aware query refinement. In *DASFAA*, volume 8421, pages 110–125, 2014.

Abdullah Mohammed Albarrak. *Similarity-aware query refinement for data exploration*. PhD thesis, School of Information Technology and Electrical Engineering, University of Queensland, Brisbane, QLD, Australia, 2018. [Online]. Available: `https://doi.org/10.14264/uql.2018.416`.

S. Algarni, B. Glavic, S. Lee, and A. Chapman. Solving why-not questions for aggregate constraints through query repair. In *IEEE European Symposium on Security and Privacy Workshops*, pages 592–596, 2024.

S. Algarni, B. Glavic, S. Lee, and A. Chapman. Efficient query repair for aggregate constraints. 2026. Under revision.

Nadim W Alkharouf, D Curtis Jamison, and Benjamin F Matthews. Online analytical processing (olap): a fast and effective data mining tool for gene expression databases. *BioMed Research International*, 2005(2):181–188, 2005.

Santiago Andrés Azcoitia and Nikolaos Laoutaris. A survey of data marketplaces and their business models. *ACM SIGMOD Record*, 51(3):18–29, 2022.

Agathe Balayn, Christoph Lofi, and Geert-Jan Houben. Managing bias and unfairness in data for decision support: a survey of machine learning and data engineering approaches to identify and mitigate bias and unfairness within data management and analytics systems. *The VLDB Journal*, 30(5):739–768, 2021.

Pablo Barceló and Miguel Romero. The complexity of reverse engineering problems for conjunctive queries. *arXiv preprint arXiv:1606.01206*, 2016.

Abderrahmen Belfkih, Claude Duvallet, and Bruno Sadeg. A survey on wireless sensor network databases. *Wireless Networks*, 25(8):4921–4946, 2019.

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. Ai fairness 360: An extensible toolkit for detecting and mitigating algorithmic bias. *IBM J. Res. Dev.*, 63(4/5):4:1–4:15, 2019.

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

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975a.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975b.

Jon Louis Bentley and James B Saxe. Decomposable searching problems i. static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. Refining sql queries based on why-not polynomials. In *TaPP*, 2016.

Angela Bonifati, Radu Ciucanu, and Sławek Staworko. Learning join queries from user examples. *ACM Transactions on Database Systems (TODS)*, 40(4):1–38, 2016.

Angela Bonifati, Ugo Comignani, Emmanuel Coquery, and Romuald Thion. Interactive mapping specification with exemplar tuples. *ACM Transactions on Database Systems (TODS)*, 44(3):1–44, 2019.

Stephan Borzsony, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proceedings 17th international conference on data engineering*, pages 421–430. IEEE, 2001.

N. Bruno, S. Chaudhuri, and D. Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1721–1725, 2006. .

Nicolas Bruno and Surajit Chaudhuri. Flexible database generators. In *Proceedings of the 31st international conference on Very large data bases*, pages 1097–1107, 2005.

Toon Calders, Faisal Kamiran, and Mykola Pechenizkiy. Building classifiers with independency constraints. In *2009 IEEE international conference on data mining workshops*, pages 13–18. IEEE, 2009.

Felix S Campbell, Alon Silberstein, Julia Stoyanovich, and Yuval Moskovitch. Query refinement for diverse top-k selection. *Proceedings of the ACM on Management of Data*, 2(3):1–27, 2024.

Adriane Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.

Surajit Chaudhuri. Generalization and a framework for query modification. *A Mediator Architecture for Abstract Data Access*, page 72, 1990.

Surajit Chaudhuri, Gautam Das, Vagelis Hristidis, and Gerhard Weikum. Probabilistic ranking of database query results. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 888–899, 2004.

James Cheney, Laura Chiticariu, Wang-Chiew Tan, et al. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases*, 1(4):379–474, 2009.

Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

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

Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. Explore-by-example: An automatic query steering framework for interactive data exploration. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 517–528, 2014.

Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. Aide: an active learning-based approach for interactive data exploration. *IEEE Transactions on Knowledge and Data Engineering*, 28(11):2842–2856, 2016.

Nosayba El-Sayed, Zhuoran Sun, Ke Sun, and Ricardo Mayerhofer. Oltp in real life: a large-scale study of database behavior in modern online retail. In *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2021.

Ramez Elmasri and Shamkant B Navathe. Fundamentals of database systems seventh edition, 2016.

Pierre Erbacher, Ludovic Denoyer, and Laure Soulier. Interactive query clarification and refinement via user simulation - abstract. In *Proceedings of the 2nd Joint Conference of the Information Retrieval Communities in Europe (CIRCLE 2022), Samatan, Gers, France, July 4-7, 2022*, volume 3178, 2022.

Michael Feldman, Sorelle A Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. Certifying and removing disparate impact. In *proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 259–268, 2015.

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

Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

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

Zhian He and Eric Lo. Answering why-not questions on top-k queries. *IEEE Transactions on Knowledge and Data Engineering*, 26(6):1300–1315, 2012.

Melanie Herschel and Mauricio A Hernández. Explaining missing answers to spjua queries. *Proceedings of the VLDB Endowment*, 3(1-2):185–196, 2010.

Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. A survey on provenance: What for? what form? what from? *The VLDB Journal*, 26:881–906, 2017.

Yuto Ikeda, Chuan Xiao, and Makoto Onizuka. An efficient diversity-aware method for the empty-answer problem. In *DOLAP*, pages 68–72, 2024.

Ihab F Ilyas, George Beskales, and Mohamed A Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):1–58, 2008.

Md Saiful Islam, Chengfei Liu, and Rui Zhou. On modeling query refinement by capturing user intent through feedback. 2012.

Md Saiful Islam, Chengfei Liu, and Rui Zhou. A framework for query refinement with user feedback. *Journal of Systems and Software*, 86(6):1580–1595, 2013a.

Md Saiful Islam, Rui Zhou, and Chengfei Liu. On answering why-not questions in reverse skyline queries. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 973–984. IEEE, 2013b.

Md Saiful Islam, Chengfei Liu, and Jianxin Li. Efficient answering of why-not questions in similar graph matching. *IEEE Transactions on Knowledge and Data Engineering*, 27(10):2672–2686, 2015.

Yash Jani. Optimizing database performance for large-scale enterprise applications. *International Journal of Science and Research (IJSR)*, 11(10):1394–1396, 2022.

Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, and Dishant Gosain. A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*, 1(6):1–5, 2012.

Manas Joglekar, Hector Garcia-Molina, and Aditya Parameswaran. Interactive data exploration with smart drill-down. *IEEE Transactions on Knowledge and Data Engineering*, 31(1):46–60, 2017.

Abhijit Kadlag, Amol V. Wanjari, Juliana Freire, and Jayant R. Haritsa. Supporting exploratory queries in databases. In *DASFAA*, volume 2973, pages 594–605, 2004.

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

Michael Kaufmann and Andreas Meier. Sql and nosql databases. *Switzerland: Springer. doi*, 10:978–3, 2023.

Zulkarnain Kedah. Use of e-commerce in the world of business. *Startupreneur Business Digital (SABDA Journal)*, 2(1):51–60, 2023.

Jon Kleinberg, Jens Ludwig, Sendhil Mullainathan, and Ashesh Rambachan. Algorithmic fairness. In *Aea papers and proceedings*, volume 108, pages 22–27. American Economic Association 2014 Broadway, Suite 305, Nashville, TN 37203, 2018.

Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.

Seokki Lee, Boris Glavic, Adriane Chapman, and Bertram Ludäscher. Hybrid query and instance explanations and repairs. In *Companion Proceedings of the ACM Web Conference 2023*, pages 1559–1562, 2023.

Hanzhe Li, Xiangxiang Wang, Yuan Feng, Yaqian Qi, and Jingxiao Tian. Integration methods and advantages of machine learning with cloud data warehouses. *International Journal of Computer Science and Information Technology*, 2(1): 348–358, 2024.

Hao Li, Chee-Yong Chan, and David Maier. Query from examples: An iterative, data-driven approach to query construction. *Proceedings of the VLDB Endowment*, 8(13):2158–2169, 2015.

Jinyang Li, Yuval Moskovitch, Julia Stoyanovich, and HV Jagadish. Query refinement for diversity constraint satisfaction. *Proceedings of the VLDB Endowment*, 17(2): 106–118, 2023.

Denis Mayr Lima Martins. Reverse engineering database queries from examples: State-of-the-art, challenges, and research opportunities. *Information Systems*, 83: 89–100, 2019.

Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. A survey on bias and fairness in machine learning. *ACM computing surveys (CSUR)*, 54(6):1–35, 2021.

Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. A survey on bias and fairness in machine learning. *ACM Comput. Surv.*, 54(6):115:1–115:35, 2022.

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

Jack Minker and Jerome D Sable. Relational data system study. Technical report, 1970.

Chaitanya Mishra and Nick Koudas. Interactive query refinement. In *EDBT*, pages 862–873, 2009.

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

Guido Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB*, pages 476–487, 1998.

Yuval Moskovitch, Jinyang Li, and HV Jagadish. Bias analysis and mitigation in data-driven tools using provenance. In *Proceedings of the 14th International Workshop on the Theory and Practice of Provenance*, pages 1–4, 2022.

Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. Exemplar queries: Give me an example of what you need. *Proceedings of the VLDB Endowment*, 7(5):365–376, 2014a.

Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. Iqr: an interactive query relaxation system for the empty-answer problem. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of Data*, pages 1095–1098, 2014b.

Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. Exemplar queries: a new way of searching. *The VLDB Journal*, 25:741–765, 2016a.

Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. A holistic and principled approach for the empty-answer problem. *VLDBJ*, 25(4):597–622, 2016b.

Ion Muslea. Machine learning for online query relaxation. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 246–255, 2004.

Ion Muslea and Thomas J Lee. Online query relaxation via bayesian causal structures discovery. In *AAAI*, pages 831–836, 2005.

Raghunath Nambiar, Meikel Poess, Andrew Masland, H Reza Taheri, Matthew Emmerton, Forrest Carman, and Michael Majdalany. Tpc benchmark roadmap 2012. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 1–20. Springer, 2012.

Jay F Nunamaker Jr, Minder Chen, and Titus DM Purdin. Systems development in information systems research. *Journal of management information systems*, 7(3): 89–106, 1990.

Oluwafemi Oloruntoba. Ai-driven autonomous database management: Self-tuning, predictive query optimization, and intelligent indexing in enterprise it environments. *World Journal of Advanced Research and Reviews*, 25(2):1558–1580, 2025.

Jin-Su Park and Chan Hee Lee. Clinical study using healthcare claims database. *Journal of rheumatic diseases*, 28(3):119–125, 2021.

Jigna Ashish Patel and Priyanka Sharma. Online analytical processing for business intelligence in big data. *Big data*, 8(6):501–518, 2020.

L Petrenko and SO Kravtsov. Design of an e-commerce database. *Publishing House "Baltija Publishing"*, 2023.

Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1–2, 2009.

Khawaja Ubaid Rehman, Umair Ahmad, and Sajid Mahmood. A comparative analysis of traditional and cloud data warehouse. *VAWKUM Trans. Comput. Sci*, 6:34–40, 2018.

Kenneth A Ross, Divesh Srivastava, Peter J Stuckey, and S Sudarshan. Foundations of aggregation constraints. *Theoretical Computer Science*, 193(1-2):149–179, 1998.

Sudeepa Roy and Dan Suciu. A formal approach to finding explanations for database queries. In *SIGMOD*, pages 1579–1590, 2014.

Mohamed A Sharaf and Humaira Ehsan. Efficient query refinement for view recommendation in visual data exploration. *IEEE Access*, 9:76461–76478, 2021.

Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. Discovering queries based on example tuples. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 493–504, 2014.

Suraj Shetiya, Ian P Swift, Abolfazl Asudeh, and Gautam Das. Fairness-aware range queries for selecting unbiased data. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1423–1436. IEEE, 2022.

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

Priyanka Subash, Alex Gray, Misque Boswell, Samantha L Cohen, Rachael Garner, Sana Salehi, Calvary Fisher, Samuel Hobel, Satrajit Ghosh, Yaroslav Halchenko, et al. A comparison of neuroelectrophysiology databases. *Scientific data*, 10(1):719, 2023.

Balder ten Cate, Phokion G Kolaitis, and Carsten Lutz. Query repairs. In *28th International Conference on Database Theory (ICDT 2025)*, pages 15–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025.

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

Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 535–548, 2009.

Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query reverse engineering. *The VLDB Journal*, 23(5):721–746, 2014.

Manasi Vartak, Venkatesh Raghavan, and Elke A Rundensteiner. Qrelx: generating meaningful queries that provide cardinality assurance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1215–1218, 2010.

Manasi Vartak, Venkatesh Raghavan, Elke Rundensteiner, and Samuel Madden. Refinement driven processing of aggregation constrained queries. 2016.

Sahil Verma and Julia Rubin. Fairness definitions explained. In *Proceedings of the international workshop on software fairness*, pages 1–7, 2018.

Bienvenido Vélez, Ron Weiss, Mark A. Sheldon, and David K. Gifford. Fast and effective query refinement. In *SIGIR*, pages 6–15, 1997.

Xiaolan Wang, Alexandra Meliou, and Eugene Wu. Qfix: Diagnosing errors through query histories. In *SIGMOD*, pages 1369–1384, 2017.

Eugene Wu and Samuel Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.

Xiaoyan Yang, Cecilia M Procopiuc, and Divesh Srivastava. Summarizing relational databases. *Proceedings of the VLDB Endowment*, 2(1):634–645, 2009.

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

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

Moshé M Zloof. Query-by-example: the invocation and definition of tables and forms. In *Proceedings of the 1st international conference on very large data bases*, pages 1–24, 1975.