
# Query Refinement for Diversity Constraint Satisfaction

Jinyang Li
University of Michigan
jinyli@umich.edu

Yuval Moskovitch
Ben Gurion University of the Negev
yuvalmos@bgu.ac.il

Julia Stoyanovich
New York University
stoyanovich@nyu.edu

H. V. Jagadish
University of Michigan
jag@umich.edu

## ABSTRACT

Diversity, group representation, and similar needs often apply to query results, which in turn require constraints on the sizes of various subgroups in the result set. Traditional relational queries only specify conditions as part of the query predicate(s), and do not support such restrictions on the output. In this paper, we study the problem of modifying queries to have the result satisfy constraints on the sizes of multiple subgroups in it. This problem, in the worst case, cannot be solved in polynomial time. Yet, with the help of provenance annotation, we are able to develop a query refinement method that works quite efficiently, as we demonstrate through extensive experiments.

**PVLDB Reference Format:**
Jinyang Li, Yuval Moskovitch, Julia Stoyanovich, and H. V. Jagadish. Query Refinement for Diversity Constraint Satisfaction. PVLDB, 17(2): 106 - 118, 2023.
doi:10.14778/3626292.3626295

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/JinyangLi01/Query_refinement.

## 1 INTRODUCTION

With increasing awareness of the need to improve representation of historically underrepresented population groups, many companies, government agencies, educational institutions, professional societies, and other organizations have adopted diversity initiatives as part of their selection processes for recruitment, award nomination, etc., aiming to challenge long-standing biases, creating a diverse environment in the long run. For example, fellowship programs often require a diverse pool of nominees, ensuring representation from underrepresented demographic groups in a particular field [1, 2]. The NFL's Rooney Rule [3] mandates that teams interview at least one candidate from racially or ethnically diverse backgrounds for senior leadership positions, aiming to disrupt entrenched biases, although it does not dictate the final hiring decision. In these contexts, relational databases are frequently employed to select candidates.

Traditional relational queries output tuples that meet specified conditions in the query predicates. If a query is used as part of some high-stakes selection process, it would be natural to state diversity requirements as cardinality constraints for including specific demographic groups in the query result, like inviting a certain number of individuals from different demographic groups for job interviews. Unfortunately, such constraints are currently not supported by relational systems: We can specify criteria on how to select candidates for a job interview, but cannot impose constraints on demographic group membership of the selected candidates. In this paper, we propose a method to augment relational queries with group cardinality constraints to fulfill diversity requirements. Our approach is based on existing *query refinement* literature, which modifies queries to meet overall cardinality bounds [4–12]. However, existing methods do not handle constraints over specific groups in the query result. We further explain the need for such constraints and provide an overview of our approach using an example.

**Table 1: `Applicants` table with primary key ID. Job applicants selected by queries `Q1`, `Q2`, and `Q3` are marked with ✓.**

| ID | Gender | Race | Major | GPA | Q1 | Q2 | Q3 |
|---|---|---|---|---|---|---|---|
| 1 | F | White | ME | 3.65 | | | |
| 2 | F | White | CS | 3.95 | ✓ | ✓ | ✓ |
| 3 | F | Black | CS | 3.40 | | | |
| 4 | F | White | ME | 3.60 | | | |
| 5 | F | White | EE | 3.85 | | ✓ | |
| 6 | F | Black | EE | 3.90 | | ✓ | ✓ |
| 7 | F | Asian | EE | 3.85 | | ✓ | |
| 8 | M | White | CS | 3.65 | | | |
| 9 | M | White | CS | 3.90 | ✓ | ✓ | ✓ |
| 10 | M | Black | CS | 3.85 | ✓ | ✓ | |
| 11 | M | White | CS | 3.40 | | | |
| 12 | M | White | EE | 3.85 | | ✓ | |
| 13 | M | Asian | EE | 3.95 | | ✓ | ✓ |
| 14 | M | Black | ME | 3.60 | | | |

**Table 2: `Internships` table, with primary key (ID, Type).**

| ID | Type | Hours |
|---|---|---|
| 2 | r.l. | 80 |
| 2 | t.c. | 90 |
| 3 | t.c. | 90 |
| 6 | g.a. | 120 |
| 7 | t.c. | 40 |
| 7 | g.a. | 60 |
| 9 | r.l. | 60 |
| 13 | g.a. | 100 |
| 14 | t.c. | 100 |

*Example 1.1.* Table 1 shows a dataset `D` of 14 job candidates applying to a tech company, with five attributes: ID, gender, race, (college) major, and GPA. Among these, gender and race are the *sensitive attributes* that denote demographic group membership. The employer may wish to state cardinality constraints w.r.t. such attributes, to counteract the effects of historical discrimination. Major and GPA are *qualification attributes*, and can be used in selection conditions. The company would like to interview applicants who graduated from college with a technical major and a high GPA. To start, the data analyst uses the following query to select candidates:

```
Q1: SELECT *
    FROM Applicants AS a
      WHERE a.Major = 'CS' AND a.GPA >= 3.85
```

Query `Q1` selects candidates #2, #9, and #10, as marked in Table 1. Notably, only one female applicant is chosen despite half of the


Proceedings of the VLDB Endowment, Vol. 17, No. 2 ISSN 2150-8097.
doi:10.14778/3626292.3626295

applicants being female, and no Asian is selected, although Whites, Blacks and Asians are all present in the applicant pool. To increase diversity, the company aims to interview at least two females, and at least one applicant of each race. These requirements can be expressed as *group cardinality constraints*, achieved by slightly adjusting the selection criteria, such as expanding the set of majors to also include Electrical Engineering (EE), thus *relaxing* the query:

```
Q2: SELECT *
    FROM Applicants AS a
      WHERE (a.Major = 'CS' or a.Major = 'EE')
      AND a.GPA >= 3.85
```

In addition to Q1, query Q2 adds applicants #5, #6, #7, #12 and #13 to be interviewed, as marked in Table 1. The selected set includes four women and at least one individual of each race, satisfying the diversity requirement on both gender and race.

Next, the data analyst observes that a total of eight candidates will be interviewed based on Q2. This will require a substantial time commitment from the company's human resources department, who then impose an additional requirement — to limit the number of selected candidates to no more than five, thus introducing an upper-bound constraint on the size of the overall result set. To satisfy this additional constraint, the query can, once again, be refined, this time *restricting* the GPA range:

```
Q3: SELECT *
    FROM Applicants AS a
      WHERE (a.Major = 'CS' or a.Major = 'EE')
      AND a.GPA >= 3.90
```

The result of Q3 consists of 4 applicants, #2, #6, #9, and #13, with 2 of each gender and at least 1 of each race, satisfying all constraints.

*Example 1.2.* Next, suppose the employer introduces an additional requirement of selecting applicants with internship experience, with information in Table 2. To select applicants with at least 80 internship hours, the data analyst uses this query:

```
Q4: SELECT *
    FROM Applicants, Internships AS a, i
      WHERE a.ID = i.ID
      WHERE (a.Major = 'CS' or a.Major = 'EE')
      AND a.GPA >= 3.85 AND i.Hours >= 80
```

To have a team with diverse professional backgrounds, the company may wish to interview at least one applicant for each internship type: research lab (r.l.), tech company (t.c.), and government agency (g.a.). This can, once again, be expressed as a cardinality constraint, using attributes from `Internships` (Table 2). Thus, together with predicates and constraints from Example 1.1, we now have multiple selection predicates and cardinality constraints involving attributes from both tables. Then we can continue to refine the predicates in this query, as already discussed.

*Outline of our approach.* In this paper, we study the problem of finding minimal refinements of a query so that its result satisfies the given group cardinality constraints. Our goal is to modify selection predicates *just enough* to meet all the constraints, while preserving the intention of the original query as much as possible. Note that unlike the running example, our solution does not divide the process into steps. Instead, it considers all constraints and returns all minimal refinements that comply with them.

Finding minimal refinements presents two challenges: 1) the time-consuming execution of candidate query refinements to test constraint satisfaction, especially for large datasets or remote databases, and 2) the computationally expensive exhaustive analysis due to the combinatorially large number of possible predicate changes. To address these challenges, we adopt a provenance model that annotates tuples with relevant predicate information and translates cardinality constraints into algebraic expressions for constraint testing, following Green *et al.* [13]. Additionally, we introduce Possible Value Lists (PVL), a data structure encompassing all possible predicate values, and employ an efficient searching algorithm with pruning techniques to identify minimal refinements.

Diversity is a compelling need when distributing resources and opportunities in a society, as we saw in our running example. Furthermore, it is also desirable in many other settings. For example, we usually want search and recommendation systems to produce diverse results [14]. While we use a running example about fairness (in the sense of representation), we underscore that the techniques we propose in this paper can be applied in many other contexts.

*Alternative approaches.* One way to satisfy group cardinality constraints is to modify the result set directly in a post-processing step, adding or removing tuples: Adding candidate #7, who is both female and Asian, to the result of Q1 would meet the diversity requirements. However, this method may be illegal in some jurisdictions and application contexts (e.g., it would be illegal in the US in the context of employment, housing, and lending), because it uses demographic group membership explicitly as part of decision making, effectively subjecting applicants from different demographic groups to different processes. Even where legal, this method may have undesirable side effects, such as tarring all members of a group, including its best-qualified members, as "weaker", on account of there being a different standard applied to the group.

Unlike post-processing, modifying the query predicates is usually legal, as it applies the same evaluation process to all individuals. One famous case involves the University of Texas, which was not permitted to give African Americans explicit (post-processing) preference in admissions; however, it was allowed to use rank in school as the basis for admission, thereby admitting top students from poor-performing segregated schools in preference to second-tier white students from top schools who may have better test scores.

Another alternative approach proposed by Shetiya *et al.* [15] shares similar motivation with our work but differs in four important ways. The first three are technical. First, they only handle constraints over a single binary sensitive attribute (e.g., male vs. female gender *or* majority vs. minority ethnicity), while we handle multiple sensitive attributes and do not limit them to be binary. Second, we support both query relaxation (i.e., generating more result tuples, asquery Q2) and query contraction (i.e., generating fewer result tuples, as Q3), while [15] only supports query relaxation. Third, [15] only focuses on queries over a single relation, while we support SPJ queries with predicates and constraints of attributes over multiple tables joined together, as shown in Example 1.2.

The fourth difference between our work and [15] is conceptual. Their diversity objective aims to minimize the distance between the *result sets* of the original query and the rewritten query, while our objective is to minimize the distance between the *queries themselves*. Although this difference might seem subtle, it leads to fundamentally different objectives in the rewriting process: preserving the salient properties of the selection process while potentially changing the result substantially vs. preserving the result as much as

possible while potentially making substantial changes to the selection process. For example, in the running Example 1.1, Q2 and Q3 are two refinements of Q1. Shetiya et al. would consider Q3 to have higher similarity (0.4) to Q1 than Q2 (0.375), while we consider Q2 to have higher similarity to Q1 based on the query predicates.

*Contributions and roadmap.* In Section 2, we formally define the Query Refinement Problem, the problem of finding minimal refinements of a query sequence given a dataset and a set of constraints on the query results, analyze its theoretical complexity, and prove its hardness. In Section 3, we describe our provenance model based on [13]. Then, Section 4 introduces the Possible Value Lists (PVL), a data structure that represents possible refinements based on provenance expressions. The PVL aims to enhance the efficiency of searching for minimal refinements. Our algorithm for generating minimal refinements using the PVL is detailed in Section 5, along with proposed optimizations for special cases. In Section 6, we experimentally evaluate our solution with multiple datasets, queries, and constraints. We give an overview of related work in Section 7 and conclude in Section 8.

## 2 PROBLEM DEFINITION

In this section, we introduce the notion of query refinement, cardinality constraints, and formally define the Query Refinement Problem. Additionally, we prove that the defined problem is not solvable in polynomial time, in the worst case. Table 3 summarizes the notations used in this paper.

### 2.1 Query Refinement

We support the class of queries considered in [4], conjunctive Select-Project-Join (SPJ) queries with selection predicates over *numerical* or *categorical* attributes. Selection predicates over numerical attributes include range ($<, \leq, \geq, >$) and equality ($=$). Categorical predicates are of the form `attribute = constant_1 OR...OR attribute = constant_n`. For ease of presentation, in the rest of the paper, we assume numerical predicates are of the form `attribute <op> constant.`[1], where `<op>` can be one of $\{<, \leq, \geq, >\}$. Equality predicates `attribute = constant` are translated into two predicates `attribute >= constant` and `attribute <= constant`.

For a query $Q$, we use $Q_n$ and $Q_c$ to denote the set of numerical and categorical selection predicates, respectively. The parts of a predicate are represented as $p.A$, $p.op$, and $p.C$. Specifically, for numerical predicates in the form `attribute >= constant`, $p.A$ represents the attribute, $p.op$ represents the operator $\geq$, and $p.C$ represents the constant. Similarly, for categorical predicates of the form `attribute = constant_1 OR...OR attribute = constant_n`, $p.A$ represents the attribute, and $p.C$ represents the set of constants `{constant_1,...,constant_n}` in predicate $p$.

We use the notion of query refinement defined in [4] to formally state our problem. For a numerical predicate, refinements are changes to the value of the constant; while for categorical predicates, a refinement is done by adding and/or removing predicates from the original constant list. We say that a query $Q'$ is a *refinement* of query $Q$ if $Q'$ is obtained from $Q$ by refining some predicates of $Q$. For example, query Q3 (selecting applicants with CS or EE major and GPA at least 3.90) is a refinement of Q1 (selecting applicants with CS major and GPA at least 3.85) from Example 1.1.

[1]Extending our solution to support other forms of numerical predicates, such as `attribute1 <op> constant·attribute2` is straightforward, and we do not discuss them in this paper.

**Table 3: Notation Table**

| Notation | Description |
|---|---|
| $D$ | dataset |
| $Q$ | query or refinement query |
| $Q(D)$ | result of executing $Q$ over $D$ |
| $Q_n, Q_c$ | numerical predicates, categorical predicates of $Q$ |
| $p.A, p.C, p.op$ | attribute, constant, and operator of predicate $p$ |
| $\mathcal{G}$ | conjunction of conditions |
| $Q(D)_{\mathcal{G}}$ | group of tuples in $Q(D)$ satisfying $\mathcal{G}$ |
| $Cr$ | cardinality constraints defining a group |
| $\mathcal{V}_{Q(D)}$ | set of variables in provenance expressions |
| $prov(t)$ | annotation of tuple $t \in D$ |
| $\mathcal{I}^{Cr}_{Q(D)}$ | provenance expression set |
| $val_{Q'}(A_v)$ | valuation of $A_v \in \mathcal{V}_{Q(D)}$ with refinement $Q'$ |
| $Tval_{Q'}(\mathcal{I}^{Cr}_{Q(D)})$ | truth value of expression set $\mathcal{I}^{Cr}_{Q(D)}$ |
| $L_{D,Q}$ | PVL given $D, Q$ |
| $Q.R$ | list of indices representing $Q$ in PVL $L_{D,Q}$ |
| $Q\|_P$ | partial query of $Q$ containing predicates in set $P$ |

### 2.2 Group Cardinality Constraints

Let $Q(D)$ be the result of executing query $Q$ over dataset $D$, and $\mathcal{G}$ be a group defined by a conjunction of conditions with specified the value of some attributes. For instance, the group of Black females is $\{Gender = F, Race = Black\}$. We use $Q(D)_{\mathcal{G}}$ to denote the group of tuples in $Q(D)$ that satisfy the condition $\mathcal{G}$. A cardinality constraint $Cr$ over $Q(D)_{\mathcal{G}}$ is a conjunction of expressions of the form $|Q(D)_{\mathcal{G}}|\ op\ x$, where $op \in \{=, \leq, <, >, \geq\}$ and $x$ is a constant, or a function of the size of some other data group defined using $\mathcal{G}'$. We say that $Q(D)$ satisfies $Cr$ if all the expressions hold. Namely, multiple cardinality constraints should be satisfied conjunctively.

Given a dataset $D$, a query $Q$, and a set of cardinality constraints $Cr$ such that $Q(D)$ does not satisfy $Cr$, there can be multiple ways to refine $Q$ so as to satisfy the constraints, as we demonstrate next.

*Example 2.1.* Recall in that our running example (Example 1.1), the original query (Q1) selects applicants with CS major and GPA of at least 3.85, resulting in only one female being selected. However, the company aims to select at least two females, which can be formally expressed as $Q(D)_{Gender=F} \geq 2$. Both Q2 (adding EE major) and Q3 (adding EE major and restriting GPA to be at least 3.90) are refinements of Q1 that select at least two females. Another plausible refinement that qualifies at least two females is to adjust the GPA predicate to be `a.GPA >= 3.40`.

The above example suggests that the company can achieve its diversity goal through various query modifications. Minimal modifications to the original query are preferred, prioritizing `a.GPA >= 3.60` over `a.GPA >= 3.55`. However, refinements modifying different attributes might be incomparable without additional preference information provided by the end user (as long as they all satisfy the constraints). Thus, we define the set of minimal refinements.

For query $Q$ and its refinements $Q'$ and $Q''$, $Q'$ *dominates* $Q''$ if $Q'$ is "closer" to $Q$ than $Q''$ for every refined predicate. For example, in the job applicant context, `a.GPA >= 3.85` has refinement `a.GPA >= 3.90` dominating `a.GPA >= 3.60`. Similarly, for the predicate `a.Major = 'CS'`, refinement `a.Major = 'EE'` dominates refinement `a.Major = 'EE' or a.Major = 'ME'`. A *minimal refinement* of $Q$ w.r.t. $Cr$ satisfies three conditions: (i) $Q'$ is a

refinement of $Q$, (ii) $Q'(D)$ satisfies $Cr$, and (iii) there is no other refinement $Q''$ that satisfies conditions (i) and (ii) while dominating $Q'$. Our goal is to find all minimal refinements of a query with respect to a set of cardinality constraints as we next define.

Problem 2.2 (Query Refinement Problem). *Given a dataset $D$, a SPJ query $Q$, and a set of cardinality constraints $Cr$, find all minimal refinements of $Q$ with respect to $Cr$.*

Note that multiple minimal refinements may exist, and our objective is to report all of them. It is also possible that no refinement meets the constraints, resulting in an empty result set. A straightforward approach to address the Query Refinement Problem would be to traverse all possible refinements. For each refinement, we would check whether it satisfies the constraints and if so, add it to a result set $R$. Queries in $R$ that are dominated by others will finally be removed. However, this naive approach exhibits exponential time complexity, and as demonstrated shortly, no polynomial time algorithm can solve the Query Refinement Problem. Due to space limitations, the proof is omitted and can be found in [16].

Theorem 2.3. *Given a dataset, an SPJ query, and a set of cardinality constraints, no polynomial time algorithm in the number of selection predicates can guarantee the enumeration of the set of all minimal refinements.*

## 3 PROVENANCE MODEL

Given a query, the number of possible refinement queries can be extremely large, especially when dealing with queries involving multiple tables and numerous attributes. Evaluating their satisfaction of cardinality constraints can be expensive, particularly with large or remote databases, and/or a large number of constraints. However, this costly query evaluation process can be eliminated using data annotations based on provenance theory. In fact, we can generate provenance annotations within a linear pass over the dataset, and then propagate them throughout the query evaluation to generate provenance expression as we next explain.

Our provenance model is inspired by the idea of hypothetical reasoning using provenance presented in [6]. Intuitively, we generate provenance expressions, using the data, the given SPJ query, and the cardinality constraints. We leverage the idea of conditional tables (c-tables) [17], where tuples are associated with conditions. To capture the possible refinements, we annotate tuples in the data with the query selection conditions.

### 3.1 Provenance expressions

We first define a set of variables used in the provenance expressions. Given a query $Q$ over the data $D$, the set of variables $\mathcal{V}_Q$ is defined as $\mathcal{V}_{Q(D)} = \{A_{[t.A]} \mid \exists p \in Q_n \cup Q_c, p.A = A, t \in D\}$, where$[t.A]$ denotes the value of the attribute $A$ in $t$.

*Example 3.1.* In Examples 1.1 and 1.2, when the company aims to select job applicants based on their majors ($M$), GPA values ($G$) (Table 1), and internship hours ($H$) (Table 2), we have the following variables involved in this query: $\mathcal{V}_{Q(D)} = \{M_{CS}, M_{EE}, M_{ME}, G_{3.95}, G_{3.90}, G_{3.85}, G_{3.65}, G_{3.60}, G_{3.40}, H_{40}, H_{60}, H_{80}, H_{90}, H_{100}, H_{120}\}$.

We use the set of variables $\mathcal{V}_{Q(D)}$ to generate provenance annotations, and use $\widetilde{Q}$ to denote the query obtained from $Q$ when omitting the selection predicates. We generate provenance annotations for each tuple $t \in \widetilde{Q}(D)$: $prov(t) = \prod_{i=1}^{i=k} A_{i[t.A_i]}$ where each $A_i$ is an attribute that is involved in $Q$ (i.e., $\exists p \in Q_n \cup Q_c, p.A = A_i$), and $k$ is the number of such attributes. Intuitively, the annotated output $\widetilde{Q}(D)$ includes all the tuples after executing only join operations, and it represents the output of all possible refinements of $Q$, where the provenance annotation of each tuple can be used to determine whether it satisfies any given refinement.

Finally, given a dataset $D$ and a query $Q$, and a cardinality constraint $Q(D)_{\mathcal{G}}$ $op$ $x$, where $op \in \{>, \geq, <, \leq\}$, we define expressions that express the cardinality constraints using the resulting provenance annotations: $\left(\sum_{t \in Q(D)_{\mathcal{G}}} prov(t)\right)$ $op$ $x$. For a given set of cardinality constrains $Cr$ over $Q(D)$ we use $\mathcal{I}^{Cr}_{Q(D)}$ to denote the corresponding set of provenance expressions.

*Example 3.2.* Equation 1 represents the constraint of selecting at least two females based on their majors, GPA values and internship hours in our job applicant example (Example 1.1 and 1.2). We get this expression by joining Table 1 and Table 2 by attribute ID, and choosing annocations of females applicants #2, #3, #6, #7.

$$M_{CS} \cdot G_{3.95} \cdot H_{170} + M_{CS} \cdot G_{3.40} \cdot H_{90} + M_{EE} \cdot G_{3.90} \cdot H_{120} + M_{EE} \cdot G_{3.85} \cdot H_{100} \geq 2 \quad (1)$$

### 3.2 Refinements through valuation

For a query $Q$ on data $D$, each possible refinement query $Q'$ corresponds to an assignment to the variables set $\mathcal{V}_{Q(D)}$ in the corresponding provenance expression, known as a *valuation* in the provenance literature. Variables that satisfy the query are assigned the value 1 ($val_{Q'}(A_v) = 1$), while others get 0 ($val_{Q'}(A_v) = 0$). Such value assignments can then be used for the valuation of the provenance expressions associated with each tuple in the data and in turn, the truth values of the provenance expressions. Given provenance expressions $\mathcal{I}^{Cr}_{Q(D)}$ and the value assignment $val_{Q'}$, we use $Tval_{Q'}(I)$ to denote the truth value of the expression $I \in \mathcal{I}^{Cr}_{Q(D)}$, obtained by applying $val_{Q'}(A_{iv})$ on each variable $A_{iv}$ in $I$. Overloading notation we use $Tval_{Q'}(\mathcal{I}^{Cr}_{Q(D)})$ to denote the truth value of the expression set $\mathcal{I}^{Cr}_{Q(D)}$, namely $Tval_{Q'}(\mathcal{I}^{Cr}_{Q(D)}) = \wedge_{I \in \mathcal{I}^{Cr}_{Q(D)}} Tval_{Q'}(I)$

*Example 3.3.* Recall in our running example (Example 1.1 and 1.2), the company would like to have at least two females selected, which gives us the provenance expression Equation (1) from Example 3.2. With query Q4 (selecting applicants with CS or EE major, GPA at least 3.85 and at least 80 hours of internship experience), only variable $G_{3.40}$ is 0, and others are 1, so the truth value is true.

Proposition 3.4. *Let $D$ be a dataset and $Q$ a query. $Q$ satisfies a set of cardinality constraints $Cr$ if and only if $Tval_Q(\mathcal{I}^{Cr}_{Q(D)}) = True$.*

Namely, given the provenance expressions of the cardinality constraints, we can efficiently examine the effect of refinements on constraint satisfaction without the need to access the data and execute the potential refinements. Furthermore, the provenance expressions may guide the refinement search as we next explain.

## 4 POSSIBLE VALUE LISTS (PVL)

With a provenance model available for testing potential refinements without accessing the data itself or repetitively evaluating refinement queries, the next question is how to find all the minimal refinements efficiently. A straightforward method is to traverse all possible refinements one by one, but this can be computationally prohibitive given the combinatorial number of refinements. Our solution uses a set of lists depicting the possible refinements based

on the variables in the provenance expression, called the Possible Value Lists (PVL), as we next describe.

The PVL is used to enumerate the possible refinement queries. Each such query can be defined by the modifications, to the original query, in the predicates' values. For numerical predicates, minimal refinements can only include numerical values from the tuples in the data that are involved in cardinality constraints. For example, for the constraint over the number of female applicants in our running example, a refinement changing the GPA predicate to `a.GPA >= 3.45` cannot be minimal. This is because it yields the same output as refining the GPA predicate to `a.GPA >= 3.60` since no female applicant has a GPA within the range of $3.45 \leq \text{GPA} < 3.60$. However, the latter refinement dominates the former.

Given a dataset $D$, query $Q$, and cardinality constraints $Cr$, the PVL $L_{D,Q,Cr}$ represents possible modifications using lists of values. The values in the lists are sorted based on their distance from the values in the original query $Q$. Each numerical predicate $p_n$ ($A$ *op* $c$) are stored in a single list ($l_{p_n.A}$) of possible values for $p_n.A$, sorted by their absolute distance from $c$ ($|x - c|, x \in l_{p_n.A}$). Categorical predicates $p_c$ ($A \in C$) are represented using multiple lists, including $l_{A_v}$ for each possible value $v$ of $A$, with values of 1 (existence) or 0 (absence) to indicate the presence of $v$ in $C$. The order of 1 and 0 is determined based on whether $v$ is already part of $p_c$. In other words, 1 (existence) is placed higher than 0 (absence) if $v$ is already part of the original predicate $p_c$; and vice versa.

*Example 4.1.* Figure 1a depicts the PVL $L_{D,Q1,Cr}$ of Example 1.1 with dataset $D$, query `Q1` (`a.Major = 'CS' AND a.GPA >= 3.85`), and $Cr$ over the number of females, people of each race, and the overall size of the result. The Major predicate has CS but not EE or ME, so 1 is placed above 0 in the list $l_{M_{CS}}$, and 0 is placed above 1 in the lists $l_{M_{EE}}, l_{M_{ME}}$. Note that values 4.00, 3.95, and 3.90 in $l_G$ correspond to values 3.95, 3.90 and 3.85 in $D$, respectively. The meaning of colors highlighting some values will be explained later.

Note that depending on $p_n.op$ (operators in predicates) and values in $D$, the values in list $l_{p_n.A}$ may be $v \pm precision(p_n.A)$, where $v$ is a value of numerical attribute $p_n.A$ in $D$ and $precision(p_n.A)$ is its precision, e.g., in our running example $precision(GPA) = 0.05$. Specifically, when $p_n.op$ is $\geq$ ($<$), for values $v \geq p_n.C$, $l_{p_n.A}$ should contain $v + precision(p_n.A)$ instead of $v$; when $p_n.op$ is $\leq$ ($>$), for values $v \leq p_n.C$, $l_{p_n.A}$ should contain $v - precision(p_n.A)$ instead of $v$. In the running example, to contract GPA predicate (`a.GPA >= 3.85`) so that it disqualifies a 3.85 (3.90, 3.95) GPA value, it needs to be refined to `a.GPA >= 3.90 (3.95, 4.00)`.

Intuitively, each value in a list of PVL $L_{D,Q,Cr}$ demonstrates a possible refinement of a specific predicate. A combination of one value from each list corresponds to a refinement of the query $Q$. In this manner, PVL encompasses all minimal refinements.The relationship between PVL and refinements of query $Q$ is summarized in the following proposition.

Proposition 4.2. *Given a query $Q_0$ over the dataset $D$ and cardinality constraint $Cr$, let $\mathbf{Q}^*$ be the set of all possible refinement queries that can be generated using values from $D$.*

*(1) Every minimal refinement query must be in $\mathbf{Q}^*$.*

*(2) $\forall Q' \in \mathbf{Q}^*$, $Q'$ can be represented using a set of indexes $Q'.R$ in the PVL $L_{D,Q_0,Cr}$ such that $Q'.R[l]$ is the index in the list $l \in L_{D,Q_0,Cr}$. $\forall p_n \in Q'_n, \exists l_{p_n.A} \in L_{D,Q_0,Cr}, p_n.C = l_{p_n.A}[Q'.R[l_{p_n.A}]]$. $\forall p_c \in Q'_c, \exists l_{p_c.A_v} \in L_{D,Q_0,Cr}$ such that $l_{p_c.A_v}$ consists of 0 and 1. If $v \in p_c.C$ then $Q'.R[l_{p_c.A_v}]$ is the index of 1, and 0 otherwise.*

*(3) Every possible index set $Q'.R$ corresponds to a query $Q' \in \mathbf{Q}^*$.*

| | $l_{M_{EE}}$ | $l_{M_{ME}}$ | $l_{M_{CS}}$ | $l_G$ |
|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 3.85 |
| 2 | 1 | 1 | 0 | 3.90 |
| 3 | | | | 3.95 |
| 4 | | | | 4.00 |
| 5 | | | | 3.65 |
| 6 | | | | 3.60 |
| 7 | | | | 3.40 |

**(a) The PVL $L_{D,Q1,Cr}$ of running example (Example 1.1)**

| | $l_{M_{ME}}$ | $l_{M_{CS}}$ | $l_G$ |
|---|---|---|---|
| 1 | 0 | 1 | 3.85 |
| 2 | 1 | 0 | 3.90 |
| 3 | | | 3.65 |
| 4 | | | 3.60 |
| 5 | | | 3.40 |

**(b) The PVL of Example 5.3**

| | $l_{M_{EE}}$ | $l_{M_{ME}}$ | $l_G$ |
|---|---|---|---|
| 1 | 0 | 0 | 3.85 |
| 2 | 1 | 1 | 3.65 |
| 3 | | | 3.60 |
| 4 | | | 3.40 |

**(c) The PVL of Example 5.6**

**Figure 1: Possible value lists (PVL).**

*Example 4.3.* Revisiting Example 1.1 with PVL in Figure 1a, refinement query `Q2`, compared to the original query `Q1`, adds EE as a recognized major, so can be represented as $Q2.R = [2, 1, 1, 1]$. This corresponds to values $l_{M_{EE}}[2] = 1, l_{M_{ME}}[1] = 0, l_{M_{CS}}[1] = 1, l_G[1] = 3.85$. The refinement query `Q3`, which adds EE and adjust GPA range to be `a.GPA >= 3.90`, can be represented as $Q3.R = [2, 1, 1, 2]$, as highlighted in blue.

Our algorithm for generating minimal refinements utilizes the PVL to traverse the possible refinements. As stated in Proposition 4.2, all minimal refinements can be represented from the PVL. This representation holds the following properties that serve as the foundation for the correctness of our algorithm.

Proposition 4.4. *Let $D$ be a dataset, $Q$ be a query, and $Cr$ be constraints, and $L_{D,Q,Cr}$ be the corresponding PVL consisting of $n_l$ lists. Additionally, let $Q_1, Q_2 \in \mathbf{Q}$ be two different refinement queries.*

*(1) $Q_1$ dominates $Q_2 \iff \forall j, 1 \leq j \leq n_l, Q_1.R[j] \leq Q_2.R[j]$.*

*(2) If $Q_1$ and $Q_2$ are minimal refinements then $\exists j, 1 \leq j \leq n_l$, such that $Q_1.R[j] \leq Q_2.R[j]$ and $\exists j', 1 \leq j' \leq n_l$, such that $Q_1.R[j'] > Q_2.R[j']$*

# 5 GENERATING MINIMAL REFINEMENTS

We present the algorithm for searching minimal query refinements using PVL. We will describe the algorithm and then discuss optimizations that can be applied in specific scenarios.

## 5.1 The algorithm

The algorithm uses the notion of partial queries as we next define.

*Definition 5.1 (Partial queries).* Given a query $Q$ with the set of predicates $Q_n \cup Q_c$ and a subset $P \subseteq Q_n \cup Q_c$, the query $Q|_P$ is similar to $Q$ but consists only of the predicates in the set $P$. We say that $Q|_P$ is a *partial query* of $Q$.

*Example 5.2.* Consider the original query `Q1` in Example 1.1 with predicates `a.Major = 'CS' AND a.GPA >= 3.85`. For $P = \{$`a.GPA >= 3.85`$\}$, $Q1|_P$ is a partial query of `Q1`.

The high-level idea of the algorithm is to generate refinement queries that satisfy the constraints by completing the predicate set of partial queries to obtain refinements of the given query $Q$. This process is done for different partial queries using a recursive function. We next provide details on the different parts of the algorithm.

*Finding a minimal refinement (from a partial query).* Given a partial query $Q'|_P$ (where $Q'$ is a refinement of $Q$), the algorithm generates one minimal refinement query by completing the predicates of $P$ to obtain a refinement of the input query $Q$, which is

**Algorithm 1:** Search for all minimal refinements

**input** : PVL $L$, query $Q$, and cardinality constraints $\mathcal{I}$.
**output**: All minimal relaxations
**Function** `Search`$(L, Res, Q|_P, \mathcal{I})$:

1 $Q_b \leftarrow$ `findRefinement` $(L, \mathcal{I}, Q|_P)$
2 **if** $Q_b.R \neq \emptyset$ **then**
3 `update` $(Res, Q_b)$
4 **if** *there are more than one list in* $L$ **then**
5 **foreach** $l \in L$ **do**
6 $k \leftarrow Q_b.R[l]$
7 **for** $i$ *from* $k-1$ *to* $1$ **do**
8 $Q'|_{P'} \leftarrow$`addPredicate` $(Q|_P, l[i])$
9 $\mathcal{I}_r \leftarrow$ inequalities of $\mathcal{I}$ requiring relaxations
10 **if** $Tval_{Q'|_{P'}}(\mathcal{I}_r) = True$ *and there are other lists* $l'$ *s.t.* $Q_b.R[l]$ *is not the last index in* $l'$ **then**
11 $L' \leftarrow$ remove $l$ from $L$
12 $Res \leftarrow$ `Search`$(L', Res, Q'|_{P'}, \mathcal{I})$
13 remove values above index $Q_b.R[l]$ in list $l$
14 **return** $Res$

procedure `findRefinement` (line 1). Procedure `findRefinement` traverses the PVL in a top-down manner, setting each predicate of $P$ according to values in the lists of PVL. It attempts all possible refinements by exhaustive search but with pruning methods (as shown in paragraph "Reducing the search space") until a suitable refinement is found. Each refinement is represented by an index set, one in each list. The general principle is to check the index set with smaller indexes earlier. For instance, when searching in PVL in Figure 1a, we start from an index set with three 1's and one 2: $[2, 1, 1, 1]$, $[1, 2, 1, 1]$, $[1, 1, 2, 1]$, $[1, 1, 1, 2]$, but nothing works. Then we continue with an index set with two 1's and two 2's: $[2, 2, 1, 1]$, $[2, 1, 2, 1]$, $\cdots$. We stop when we find the first refinement satisfying the cardinality constraints. Since the traversal starts from values at the top of PVL, which are the closest to the original predicates, the first first satisfying refinement found must be a minimal one. There can be more than one minimal satisfying refinement, and `findRefinement` returns the one it encounters first.

*Recursive search.* The core of the algorithm is a recursive search function depicted in Algorithm 1. Given a partial query $Q|_P$ with a fixed predicate set $P$ (as part of a minimal refinement), a temporary result set *Res*, a PVL $L$, and a set of provenance inequalities, `Search` function performs the following steps. First, using the `findRefinement` procedure, the function identifies a minimal refinement $Q_b$ (line 1). If no such refinement exists, the current result set *Res* is returned. Otherwise, *Res* is updated with $Q_b$ (line 3). Next, the algorithm generates new refinements by adding a predicate with a value from a list $l \in L$ to $P$ (`addPredicate`), removing $l$ from $L$, and recursively searching for new refinements (lines 4–13). In this process, for each list $l \in L$, the algorithm considers all possible predicates that can be generated using values above $Q_b.R[l]$ in $l$. This adds a new predicate to $P$ and removes one list from $L$ in each iteration. The recursive search continues until no refinements are found in the PVL or when the PVL contains only one list.

*Reducing the search space.* When we traverse the possible refinements, the search space can be reduced using the notion of *partial dissatisfaction* as we next explain. For a given set of provenance inequalities $\mathcal{I}$, we use $\mathcal{I}_r \subseteq \mathcal{I}$ to denote the subset of inequalities that are of the form $Q(D)_{\mathcal{G}}\ op\ x$, where $op \in \{>, \geq\}$ and $x$ is a constant. These constraints are referred to as *relaxation constraints*. For any given (refinement) query $Q'$, if there exists a predicate set $P \subset Q'_n \cup Q'_c$ such that $Tval_{Q'|_P}(\mathcal{I}_r) = False$, then $Tval_{Q'}(\mathcal{I}^{Cr}_{Q(D)}) = False$. In other words, if a partial query $Q|_P$ fails to satisfy certain relaxation constraints, it implies that the entire constraint set is not satisfied by $Q'$. For example, if a partial query $Q|_P$ disqualifies 6 out of 7 females, then any query $Q$ would not satisfy the constraint of selecting at least 2 females.

*Putting it all together.* Given a query $Q$ over $D$ and a set of provenance inequalities $\mathcal{I}^{Cr}_{Q(D)}$ generated form the cardinality constraints $Cr$, we find the set of all minimal refinements by calling the `Search` function. We initiate the search by invoking `Search` with the initial parameters: $LD, Q, Cr$, an empty result set *Res*, a partial query $Q|P$ where $P = \emptyset$, and $\mathcal{I}^{Cr}Q(D)$. `Search` calls `findRefinement` procedure, as described earlier, to generate new refinements. The search is optimized by avoiding refinements that can not satisfy the constraints based on partial dissatisfaction (lines 9 – 10). Moreover, to prevent redundant checks, at the end of the recursive call with values in list $l$, the algorithm removes all these values (values above index $Q_b.R[l]$) from consideration (line 13). This is because refinements with these values have already been examined.

*Example 5.3.* Consider Example 1.1, where job applicants are selected based on their majors and GPA values shown in Figure 1. The original query `Q1` (CS major and GPA at least 3.85) has constraints over the number of females, individuals of each race, and the overall result size. Provenance expressions of the constraints are generated using the provenance model, and PVL is built as shown in Figure 1a. To reduce the search space, 3.95 and 4.00 in $l_G$ (highlighted in grey) are removed since partial queries with `a.GPA >= 3.95` (or `a.GPA >= 4.00`) do not satisfy the requirement of selecting at least two females. We search in PVL for base minimal refinement by a top-down traversal, and obtain $Q_b$ s.t. $Q_b.R = [2, 1, 1, 2]$, as highlighted in blue, refining Major predicates to `a.Major = 'EE' or a.Major = 'CS'` and GPA predicate to `a.GPA >= 3.90`. Next, we explore further refinements by checking values above the blue ones in PVL. We examine the value in $l_{M_{EE}}[1]$ and $l_G[1]$. For $l_{M_{EE}}[1]$, we add EE to the Major predicate, generate a new PVL (Figure 1b), and recursively search, but no base minimal refinements are found. Similarly, when fixing GPA predicate with value $l_G[1] = 3.85$, we generate another new PVL (Figure 1a), and the recursive search does not find any more minimal refinements either. The final result contains only one minimal refinement.

Theorem 5.4. *Given a dataset $D$, a SPJ query $Q$, and a set of provenance inequalities $\mathcal{I}^{Cr}_{Q(D)}$, our algorithm can find all minimal refinements, and all of those reported are minimal refinements.*

The above theorem states that our algorithm can find all minimal refinements, without reporting any non-minimal refinements. The second half of the theorem is trivial as the algorithm always verifies each refinement before adding it. We use induction to prove the first half by showing that each iteration of Search$(L, Res, Q'_p)$ of Algorithm 1 can find all minimal refinements. It is trivial when there is only one list in PVL. We assume that the above statement is true for an $x$-list PVL, and then prove it is still true for an $(x+1)$-list PVL. Due to space limitations, the full proof can be found in [16].

*Complexity analysis.* Generating the provenance inequalities is polynomial in the data size, as it requires a single linear pass over the dataset to annotate the data in addition to the query evaluation.

**Algorithm 2:** Find a minimal relaxation

```
   input  : A PVL L and cardinality constraints I.
   output : A minimal relaxation R
1  Procedure findRelaxation(L, I, Q|_P)
2      Q_b ← binarySearch(L, I, Q|_P)
3      if Q_b.R ≠ ∅ then
4          foreach l ∈ L do
5              foundMin ← False
6              while Q_b.R[l] ≥ 1 or foundMin do
7                  Q_b.R[l] ← Q_b.R[l] − 1
8                  if Tval_{Q_b}(I) = False then
9                      Q_b.R[l] ← Q_b.R[l] + 1
10                     foundMin ← True
11     return Q_b
```

For the searching algorithm, the worst case is to traverse all possible refinements, so the time complexity is the number of all possible refinements is the product of the length of all lists in PVL is

$$O\left(\left(\prod_{i=1}^{N_n}(|Dom(A_i)|+1)\right)\cdot 2^{\sum_{j=1}^{N_c}|Dom(A_j)|}\right)$$

where $N_n, N_c$ denote the number of numeric predicates and categorical predicates, respectively; $A_i, A_j$ denote numeric attributes and categorical attributes, respectively, and $Dom(A)$ denotes the domain of attribute $A$. In the worst case, $|Dom(A_i)| = |D| = n$, thus we have $O\left(\left(n^{N_n}\right)\cdot 2^{\sum_{j=1}^{N_c}|Dom(A_j)|}\right)$

While the time complexity is exponential to the number of numerical predicates and the size of domains of categorical predicates in the worst case, we show in our experiments that in practice, the running time of our algorithm is typically much better, due to our search space pruning.

## 5.2 Optimizations

We next describe some optimizations that can be applied when 1) numerical predicates do not have an equality (=) operator; and 2) all the constraints are relaxation constraints or all are contraction constraints, i.e., $C(D)_{\mathcal{G}}$ $op$ $x$, where $x$ is a constant and $op \in \{>, \geq\}$ (or $op \in \{<, \leq\}$) for *all* the constraints. For simplicity of presentation, in what follows, we present the case of relaxation. The case of contraction is symmetric.

First of all, note that tuples in $D$ that already satisfy the cardinality constraints do not impact the refinement process and can therefore be excluded from both the provenance inequalities and the PVL. Furthermore, when all constraints are relaxation constraints, the process of finding a minimal refinement from a partial query and the subsequent recursive searches can be optimized, owing to the monotonicity property of relaxation queries in the following proposition. Details of the proof can be found in [16].

Proposition 5.5. *Let $D$ be a dataset, $Q$ be a query, and $L_{D,Q,Cr}$ be the corresponding PVL which consists of $n_c$ lists. Let $\mathcal{I}^{Cr}_{Q(D)}$ be a set of provenance inequalities such that $Cr$ are all relaxation constraints. Let $Q_1, Q_2$ be two different relaxations such that $Tval_{Q_1}(\mathcal{I}^{Cr}_{Q(D)}) = True$ and $Q_1$ dominates $Q_2$. Then $Tval_{Q_2}(\mathcal{I}^{Cr}_{Q(D)}) = True$.*

Based on the above proposition, minimal refinements can be efficiently identified. Rather than employing a top-down search that traverses all potential refinements, we optimize this procedure by implementing a binary search to locate a refinement that satisfies the cardinality constraints. Following that, we "tighten" the refinement to make it minimal. This process is depicted in Algorithm 2. We first use a binary search (line 2) to obtain an initial query $Q_b$ such that $\forall l \in L, Q_b.R[l] = k$ and $val_{Q_b}(\mathcal{I}^{Cr}_{Q(D)}) = True$. For lists $l$ with length smaller than $k$, we set $Q.R[l]$ to be the length of $l$. Next, we tighten $Q_b$ – make it minimal by moving up the index values in each list as much as possible (lines 4 – 10). The refinement we get at the end of the process is a minimal relaxation.

Proposition 4.4 also provides us with another optimization so that some recursions can be avoided. When we recursively search for other relaxations by fixing a predicate with values above $Q_b.R$, (lines 7 – 12 in Algorithm 2), if there is a value $l[i]$ that fixing it ends up adding no new minimal relaxation to the result set, we do not need to check values above index $i$ in list $l$. In other words, from line 7 to line 12 in Algorithm 2, if one of the iterations in the for loop does not update the result set *Res* with any new minimal relaxations, we execute a "break" after line 12 to end the for loop in advance, because the following iterations would check values that relax the query less, and based on monotonicity, will definitely not satisfy the cardinality constraints.

*Example 5.6.* To show a simple example with relaxation constraints, in our running example (Example 1.1) where job applicants are selected based on their majors and GPA values in Table 1, we only consider the constraint that at least two females are selected. As mentioned before, when all constraints are relaxation constraints, provenance inequalities can be simplified by removing tuples satisfying query already, so the provenance expression is

$$\begin{aligned} &M_{ME}\cdot G_{3.65} + M_{CS}\cdot G_{3.40} + M_{ME}\cdot G_{3.60} \\ &+ M_{EE}\cdot G_{3.85} + M_{EE}\cdot G_{3.90} + M_{EE}\cdot G_{3.85} \geq 1 \end{aligned} \tag{2}$$

Compared to Figure 1a, the PVL in Figure 1c does not contain list $l_{M_{CS}}$ and values greater than 3.85 in list $l_G$ since they already satisfy `Q1`. To get an initial relaxation $Q_b$, we do a binary search and find that 2 is the smallest index $k$ such that $\forall l \in L, Q_b.R[l] = k$ and $val_{Q_b}(\mathcal{I}^{Cr}_{Q1(D)}) = True$. Then we tighten $Q_b$ by moving up values in each list as much as possible, and get $Q'_b.R = [2, 1, 1]$, highlighted in blue color, which is a minimal relaxation that adds EE to be a recognized major. Then, we fix the value in $l_{M_{EE}}[1]$ and search for other relaxations recursively, and find another two minimal relaxations: (1) modifying GPA predicate to be `a.GPA >= 3.40`; and (2) adding ME to the major list, and modifying GPA predicate to be `a.GPA >= 3.65`.

# 6 EXPERIMENTS

We empirically evaluate our proposed solutions to demonstrate their usefulness and effectiveness. We describe the experimental setup and provide a real-world case study, and then present a quantitative study assessing the efficiency of our algorithm, considering various datasets, queries, and cardinality constraints, and comparing it to the baseline solution. We analyze the impact of data size, query selectivity, and constraint properties on running time and the number of checked refinements. Moreover, we validate the effectiveness of the optimization presented in Section 5.2, and provide a comparison between our approach and [15]. The results show that our PVL-based search algorithm performs well across diverse scenarios, achieving up to 100 times faster execution than baseline traversal. The optimizations improve performance by up to 99.87%. For more comprehensive experiments, please refer to [16].

| $Q_1^H$ | $Q_2^H$ |
|---|---|
| SELECT * FROM Healthcare WHERE | |
| **income >= 200K** and **num-children >= 3** and **county in ("county2", "county3")** | **income <= 100K** and **complications >= 5** and **num-children >= 4** |
| $Q_1^A$ | $Q_2^A$ |
| SELECT * FROM ACSIncome WHERE | |
| **working_hours >= 40** and **Educational_attainment >= 19** and **Class_of_worker in ("local_gov", "state_gov", "federal_gov")** | **working_hours <= 40** and **Educational_attainment <= 19** and **income in ("<20K", "20K-40K")** |
| $Q_3^T$ | |
| SELECT * FROM customer, order, lineitem WHERE c_custkey = o_custkey and l_orderkey = o_orderkey and **c_mktsegment = 'BUILDING'** and **o_orderdate < date 1995-03-28** and **l_shipdate > date 1995-03-28** | |
| $Q_{12}^T$ | |
| SELECT * FROM order, lineitem WHERE l_orderkey = o_orderkey and l_commitdate < l_receiptdate and l_shipdate < l_commitdate **l_shipmode in ('RAIL', 'AIR')** and **l_receiptdate >= date 1995-01-01** and **l_shipdate < date 1996-01-01** | |

**Figure 2: Queries used in experiments**

| $C_1^H$ | $C_2^H$ | $C_3^H$ |
|---|---|---|
| $\{race = race2\}$ $\{age = group1\}$ | $\{race = race1\}$ $\{age = group2\}$ $\{age = group3\}$ | $\{age = group1\}$ $\{race = race1\}$ |
| $C_1^A$ | $C_2^A$ | $C_3^A$ |
| $\{sex = F,$ $marital = Married\}$ $\{race = Black\}$ | $\{sex = M, relationship = husband/wife\}$ $\{age = 30 - 60,$ $marital = divorced\}$ | $\{sex = M, race = Asian\}$ $\{sex = F, marital = Married\}$ |
| $C_1^{T,3}$ | $C_2^{T,3}$ | $C_3^{T,3}$ |
| $\{c\_nationkey = 23\}$ $\{l\_shipmode =$ MAIL$\}$ | $\{c\_nationkey = 2\}$ $\{o\_orderstatus = P,$ $l\_returnflag = N\}$ $\{l\_shipmode =$ MAIL$\}$ $\{l\_shipinstruct =$ COLLECT COD$\}$ $\{l\_linenumber = 5\}$ | $\{l\_linenumber = 6,$ $l\_shipmode =$ SHIP, $o\_orderstatus = F,$ $o\_orderpriority =$ 1-URGENT$\}$, $\{l\_shipinstruct =$ DELIVER IN PERSON, $o\_orderstatus = P,$ $o\_orderpriority =$ 2-HIGH$\}$ |
| $C_1^{T,12}$ | $C_2^{T,12}$ | $C_3^{T,12}$ |
| $\{o\_orderpriority$ $=$ 5-LOW, $l\_linenumber = 3\}$, $\{l\_shipinstruct =$ COLLECT COD$\}$, $\{l\_returnflag = A\}$ | $\{o\_orderpriority =$ 1-URGENT$\}$, $\{l\_linenumber = 1\}$ $\{l\_returnflag = A,$ $o\_orderstatus = F\}$; | $\{l\_linenumber = 7,$ $l\_shipinstruct =$ NONE, $o\_orderpriority =$ 5-LOW, $o\_orderstatus = F\}$, $\{o\_orderpriority =$ 4-NOT SPECIFIED, $l\_shipinstruct =$ TAKE BACK RETURN, $l\_linenumber = 7,$ $l\_returnflag = A\}$ |

**Figure 3: Constraints used in experiments**

## 6.1 Experiment Setup

**Datasets:** We use three different datasets to do experiments:

- **The Healthcare dataset**[2] is a dataset used in [18] to train a classifier identifying patients at risk for serious complications. It contains demographic and clinical history information of 887 patients, with attributes like the number of children, income, number of complications, county, race, age, etc..
- **ACSIncome dataset** is one of five datasets created by [19] as an improved alternative to the popular UCI Adult dataset. The data was compiled from the American Community Survey (ACS) Public Use Microdata Sample (PUMS). It covers all 50 states and Puerto Rico, with 1,664,500 rows and 11 features.
- **TPC-H benchmark**[3] is a standard benchmark for measuring the performance of relational database management systems (RDBMS) and data warehousing systems. The benchmark consists of a suite of 22 SQL queries and a database schema with 8 relational tables, and is designed to simulate the decision support systems of a typical data warehousing environment. We use TPC-H 3.0.1 to generate datasets and queries 100M data size.

To the best of our knowledge, there is no available benchmark for query refinement testing that provides datasets, queries, and constraints. For our evaluation, we choose the Healthcare and ACSIncome datasets, which are commonly used public demographic datasets in the context of fairness and diversity. We select attributes such as gender, race, age, marital status, and relationships to form the constraints, as these attributes are commonly employed as sensitive attributes that reflect membership in various demographic groups. Other attributes are used to form the queries. We also use TPC-H benchmark which includes queries, and we use attributes such as order priority and order status to form constraints.

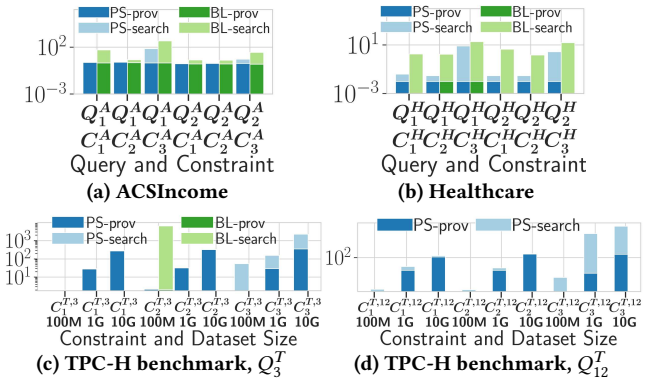
**Queries:** We generated two queries each for Healthcare and ACSIncome dataset (denoted as $Q_1^H, Q_2^H, Q_1^A, Q_2^A$) without JOIN operations as each dataset has a single relation. From TPC-H benchmark queries, we randomly chose #3 and #12 ($Q_3^T, Q_{12}^T$), which both include JOIN operations. The TPC-H query was simplified by removing GROUP BY, ORDER BY, and LIMIT clauses which are not considered in our setting and only affect tuple display. This modification doesn't affect the experiment's validity. Additionally, we replaced attribute sets in the query projections with `SELECT *` for simplicity and to ensure all attributes in the cardinality constraints were included. One predicate over l_receiptdate in query #12 is replaced with attribute l_shipdate, which enables us to explore different scenarios and apply optimizations. An overview of all queries used in the experiments is presented in Figure 2.

**Cardinality constraints:** Figure 3 summarizes the constraints used in experiments. We form three sets of cardinality constraints for each of the Healthcare, ACSIncome, and TPC-H datasets, by using representative sensitive attributes such as race and gender for the first two demographic datasets, and attributes from multiple tables that are not used in queries for non-demographic TPC-H dataset. Each set contains 2 to 5 constraints that must be fulfilled simultaneously. The first set of constraints for each dataset contains relaxation constraints, the second contains contraction constraints, and the third contains a combination of both: first group being relaxed to 105% and second being contracted to 95%. Similar results were observed for other queries and constraints. Due to space limitations, please refer to [16] for more detailed experiments involving more queries and constraints (queries $Q_3^H, Q_4^H$, constraints $C_4^H, C_5^H, C_6^H$ on the Healthcare dataset, as well as queries $Q_3^A, Q_4^A$, constraints $C_4^A, C_5^A, C_6^A$ on the ACSIncome dataset).

**Compared algorithms:** In the experimental section, we compare three approaches: a baseline method, our fully optimized proposed solution (as detailed in Sec. 5.2), and our proposed solution without optimizations. Each method operates on in-memory data and utilizes provenance expressions to evaluate potential refinements. The baseline approach also uses provenance expressions, as managing data on an external database would be less efficient due

[2]https://github.com/stefan-grafberger/mlinspect/tree/master/example_pipelines/healthcare

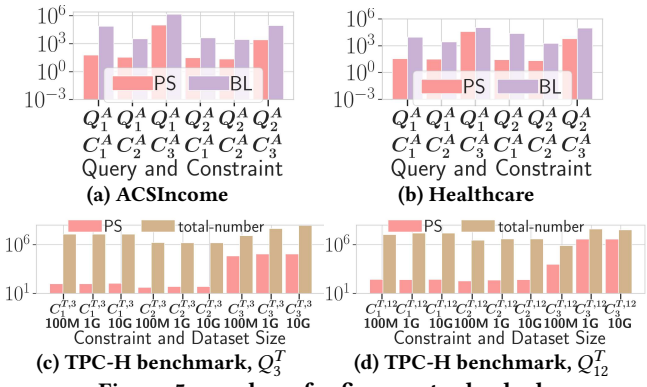
[3]https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp




(a) ACSIncome (b) Healthcare

(c) TPC-H benchmark, $Q_3^T$ (d) TPC-H benchmark, $Q_{12}^T$

**Figure 4: running time**




(a) ACSIncome (b) Healthcare

(c) TPC-H benchmark, $Q_3^T$ (d) TPC-H benchmark, $Q_{12}^T$

**Figure 5: number of refinements checked**

to multiple I/O accesses. In contrast, our solution only requires a single access to generate provenance inequalities. We begin by comparing our fully optimized proposed solution with the baseline approach. Then, we investigate the impact of various parameters on the performance of our optimized solution. Next, we compare our optimized solution with the non-optimized version. Finally, we contrast our approach with existing work in the field. Here are the three approaches used in the experiments. Note that they all report all minimal refinements, so the quality of results are the same. They differ in efficiency due to different searching strategies.

- **Baseline Algorithm (Baseline).** This naive algorithm, introduced in Section 4, utilizes a provenance model to avoid multiple database accesses. However, it identifies minimal refinements by navigating through all potential refinements.
- **PVL-based Search (PS).** This is Algorithm 1 with all optimizations in Section 5.2 to identify minimal refinements with PVL.
- **PVL-based Search without optimization (PS_no_opt).** This version of our proposed approach is also Algorithm 1, but without the optimizations described in Section 5.

**Metrics:** We compare the running time and number of possible refinements checked by each algorithm.

**Platform:** All experiments were performed on a macOS Ventura 13.2 machine with an Apple M2 Max chip, 64 GB memory. The algorithms were implemented using Python3.

## 6.2 Real-world applications

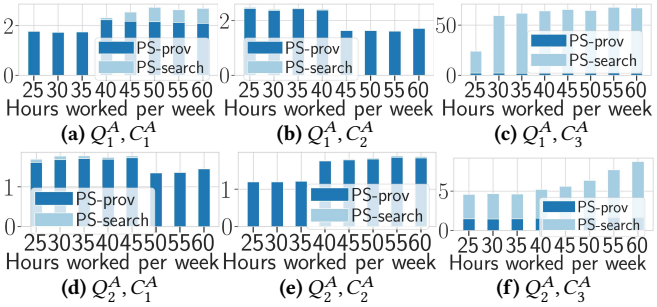
As mentioned in Section 1, adding cardinality constraints to query results has many important real-world applications, particularly where diversity matters. For example, query refinements can be necessary in the selection of individuals from the Healthcare dataset and ACSIncome dataset for healthcare subsidies, governmental surveys, and social science studies, etc., for diversity considerations. Here we show a detailed scenario used in our experiments. Consider a hospital planning to offer financial assistance to patients in need, specifically those with low incomes, a high number of children, and a high complication rate, using the Healthcare dataset. They execute query $Q_2^H$ (please see Figure 2) to select patients with an income less than 100K, more than 4 children, and more than 5 complications, resulting in 42 out of 887 patients being selected. However, the hospital notices under-representation of certain demographic groups in the query result. Only 17% of the selected patients belong to race2 among the three races, and there are fewer individuals from age group1 compared to other age groups. The hospital worries about criticism for a financial assistance program that is skewed in this manner. To achieve a more balanced demographic distribution among the recipients of financial assistance, the hospital introduces cardinality constraint $C_1^H$ to relax query $Q_2^H$, aiming to increase the representation of individuals from these two groups to 105%. Our algorithm provides three methods to minimally refine $Q_2^H$: 1) adjusting the lower bound of the number of children to 3, 2) modifying the lower bound of the number of complications to 4, and 3) changing the upper bound of income to 134K. The hospital can choose any of these refinement queries based on their preference. Additionally, the hospital may utilize constraint $C_2^H$ to limit the selection of patients from dominant groups, or combine both approaches, as demonstrated by $C_3^H$.

## 6.3 Experiment Results

We compare the running time of our algorithm with the baseline in experiments. Then, we analyze the impact of different parameters on our algorithm's running time by systematically altering them, enabling us to identify the scalability of our algorithm.

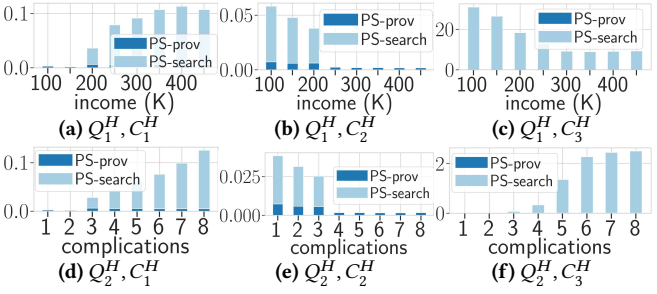
*Running time.* The first set of experiments compares the running time of our algorithm with a baseline naive algorithm. In all tested scenarios, our algorithm consistently outperforms the baseline by more than 100 times, as shown in Figure 4. The blue bar represents the running time of our solution with PVL, while the green bar shows the running time for the baseline algorithm. The dark section represents provenance generation time, and the light section shows the searching time for minimal refinements. In some cases, the green bar is omitted because the baseline approach cannot terminate within a 3-hour time-out. Our solution performs well with various constraint sets, including relaxation, contraction, and a mix of both, due to the efficient search checking fewer refinements using the PVL. The running time for different query-constraint combinations varies significantly due to factors such as the search space and the distance between the original query and queries satisfying cardinality constraints, as demonstrated next.

*Number of refinements checked.* The number of refinements checked by the algorithms indicates algorithm effectiveness. Figure 5a and 5b compare the refinements checked by the PVL-based search and the baseline. Figures 5c and 5d compare the PVL-based search with the total number of refinements represented by PVL, considering that the baseline algorithm cannot complete in most cases (Figures 4c and 4d). In all scenarios, our algorithm checks up to 99% fewer refinements compared to both the baseline algorithm




(a) $Q_1^A, C_1^A$ (b) $Q_1^A, C_2^A$ (c) $Q_1^A, C_3^A$
(d) $Q_2^A, C_1^A$ (e) $Q_2^A, C_2^A$ (f) $Q_2^A, C_3^A$

**Figure 6: Effect of query selectivity on the running time, ACSIncome data, x-axis varies constants in a predicate, y-axis is running time (s)**



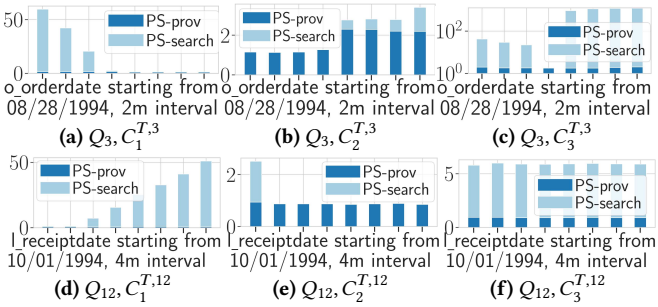

(a) $Q_1^H, C_1^H$ (b) $Q_1^H, C_2^H$ (c) $Q_1^H, C_3^H$
(d) $Q_2^H, C_1^H$ (e) $Q_2^H, C_2^H$ (f) $Q_2^H, C_3^H$

**Figure 7: Effect of query selectivity on the running time, Healthcare data, x-axis varies constants in a predicate, y-axis is running time (s)**

and the total number. This reduction is particularly notable for relaxation/contraction constraints, benefiting from the optimizations described in Section 5.2. These results align with the running time comparison, as the running time is influenced by the search space, which can be inferred from the number of refinements checked. For simplicity, we focus on comparing the running time in the following experiments and omit the number of refinements checked.

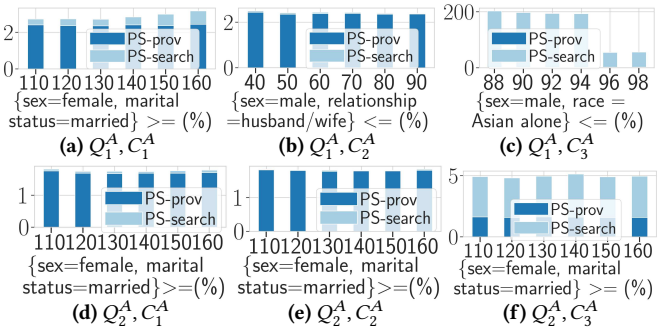
*Data size.* We examined the effect of data size on running time using the TPC-H dataset, and present our findings in Figures 4c and 4d, which shows the running time for data sizes of 100M, 1G, and 10G. Data size affects running time in two primary ways: the time to generate provenance expressions and the search space size. The time needed to generate provenance expressions and assign values to them scales linearly with the data size, while the size of the Provenance Value List (PVL) generally with the number of attribute values in larger datasets. Due to these factors, for each query and constraints pair, we observed a clear trend of increasing running time as the data size increased.

*Query Selectivity.* We analyzed query selectivity by varying a single constant in each query's predicate. Results are shown in Figures 6, 7, 8, with the $x$-axis indicating the constant's value and the $y$-axis showing running time. For relaxation constraints, increasing selectivity expands the search space, leading to longer running times (Figures 6a, 6e, 7a, 7d, 8b, 8d). On the other hand, for contraction constraints, the trend is opposite (Figures 6b, 6d, 7b, 7e, 8a, 8d). In the general case (constraint set $C_3^A, C_3^H, C_3^{T,3}, C_3^{T,12}$), increasing the constant in a predicate does not affect the size of the provenance expression containing the entire data. However, it impacts the relaxation and contraction constraints in opposite ways, resulting in a trade-off between the constraints. The running




(a) $Q_3, C_1^{T,3}$ (b) $Q_3, C_2^{T,3}$ (c) $Q_3, C_3^{T,3}$
(d) $Q_{12}, C_1^{T,12}$ (e) $Q_{12}, C_2^{T,12}$ (f) $Q_{12}, C_3^{T,12}$

**Figure 8: Effect of query selectivity on the running time, TPC-H benchmark, 100M data sizes, x-axis varies constants in a predicate, y-axis is running time (s)**

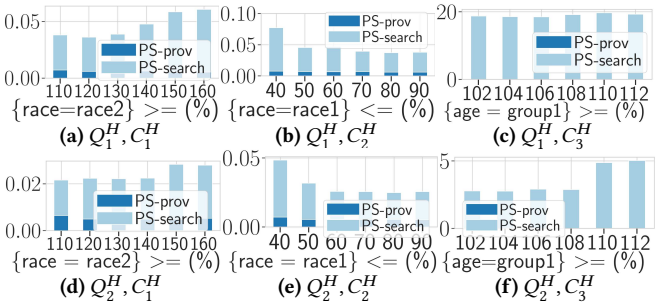



(a) $Q_1^A, C_1^A$ (b) $Q_1^A, C_2^A$ (c) $Q_1^A, C_3^A$
(d) $Q_2^A, C_1^A$ (e) $Q_2^A, C_2^A$ (f) $Q_2^A, C_3^A$

**Figure 9: Effect of constraints on the running time, ACSIncome dataset, x-axis shows how much groups are relaxed/-contracted, y-axis is running time (s)**

time is primarily influenced by the contraction constraint for lower selectivity and by meeting the relaxation constraint for higher selectivity, which may result in a turning point. A similar trend is observed in Figure 8c. However, some figures, such as Figure 6c, 6f, 7c, 7f, 8f, display a monotonic increase or decrease without a turning point. This occurs when either the contraction or relaxation constraints dominate the overall running time of the algorithm.

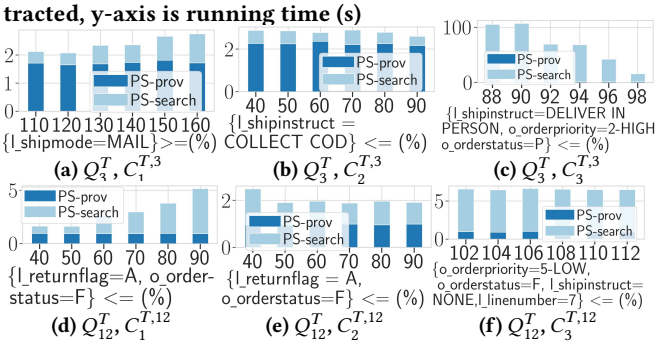
*Constraints satisfaction properties.* To analyze the effect of cardinality constraints on running time, we varied the relaxation or contraction ratio of one group to different percentages while keeping the ratios of other groups unchanged, as shown in Figures 9, 10, and 11. The size of the Provenance Value List (PVL) remains unaffected by these changes since it depends on dataset, query, and group definitions rather than the constants. As constraints are tightened or relaxed, we expect the algorithm runtime to increase or decrease, respectively, as queries satisfying the constraints move further away from or closer to the original query. An increase in running time is observed when constraints become more difficult to satisfy, as seen in Figure 9a, 10a, 10c, 10d, 10f, 11a, 11d, while the opposite trend is observed when queries become easier to satisfy. However, some figures show minimal changes in running time when constraints are altered, such as Figure 9b, 9d, 9e, 9f, 11b, 11f. This can be attributed to the uneven data group distribution in the constraints, where adjusting a predicate to a particular value can simultaneously qualify or disqualify multiple tuples, satisfying various relaxation/contraction ratios. This effect is particularly noticeable when there are many tuples with the same value in an attribute, resulting in minimal refinements that are nearly the same for different relaxation/contraction ratios.




(a) $Q_1^H, C_1^H$ (b) $Q_1^H, C_2^H$ (c) $Q_1^H, C_3^H$

(d) $Q_2^H, C_1^H$ (e) $Q_2^H, C_2^H$ (f) $Q_2^H, C_3^H$

**Figure 10: Effect of constraints on the running time, Healthcare dataset, x-axis shows how much groups are relaxed/contracted, y-axis is running time (s)**




(a) $Q_3^T, C_1^{T,3}$ (b) $Q_3^T, C_2^{T,3}$ (c) $Q_3^T, C_3^{T,3}$

(d) $Q_{12}^T, C_1^{T,12}$ (e) $Q_{12}^T, C_2^{T,12}$ (f) $Q_{12}^T, C_3^{T,12}$

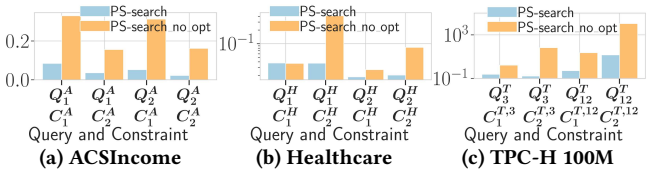
**Figure 11: Effect of constraints on the running time, TPC-H benchmark, x-axis shows how much groups are relaxed/contracted, y-axis is running time (s)**

*Effect of optimizations.* To assess the impact of the optimizations described in Section 5.2, we conducted experiments using algorithms both with and without optimizations and compared their search times. These optimizations incorporate the use of binary search instead of a top-down traversal when searching for a refinement, exploiting the monotonicity property of queries when all constraints are relaxation constraints or all are contraction constraints. We used queries ($Q_1^H, Q_2^H, Q_1^A, Q_2^A, Q_3^T, Q_{12}^T$) and the first two sets of constraints ($C_1^H, C_2^H, C_1^A, C_2^A, C_1^{T,3}, C_2^{T,3}, C_1^{T,12}, C_2^{T,12}$) for the three datasets, and the results are shown in Figure 12. The light blue bar represents the searching time with optimization, and the light orange bar represents the time without optimization. Our results indicate that the optimizations significantly reduced execution times across all tested scenarios and enhance the efficiency of the algorithm with gains of up to 99.87%.

## 6.4 Comparison with Related Work

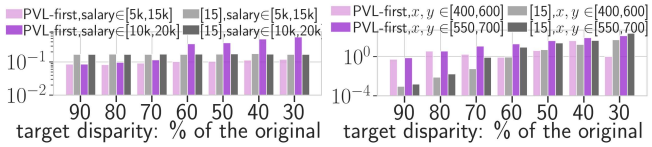
We compared our algorithm with the one by Shetiya *et al.* [15], which also addresses group cardinality constraints but with different objectives and settings. Since [15] only supports numerical predicates with a single binary sensitive attribute, all the comparison experiments are conducted in this setting. We use the C++ implementation provided by Shetiya *et al.* [15].

*Result comparison.* To be consistent with [15] and allow for easy comparison, we use TexasTribune dataset (used by [15]), containing salary information for Texas state employees, to compare the returned results. Our query imposes both upper and lower bounds for the salary attribute (`a.salary >= 10000` and `a.salary <= 20000`), resulting in a gender disparity of 69 with 100 males and 169 females in the result set. For [15], the query selects tuples with salaries in




(a) ACSIncome (b) Healthcare (c) TPC-H 100M

**Figure 12: Effect of optimizations on the searching time, y-axis is searching time (s)**




(a) 1-attribute query (b) 2-attribute query

**Figure 13: Comparison between [15] and our PVL-based searching algorithm, y-axis is running time (s)**

range [10000, 20000]. As for cardinality constraints, we require the gender disparity to be no larger than 62 (90% of the original disparity), resulting in constraint: $|Q(D)_{gender=M} - Q(D)_{gender=F}| \leq 62$. The refinement returned by [15] is a salary range $r_1 = [10135.7, 19701.5]$ (Jaccard similarity 0.95), which is also returned by our algorithm. Additionally, our algorithm returns seven other refinements: $r_2 = [10000, 19629.6]$, $r_3 = [11305.8, 20000]$,$r_4 = [11274.1, 19999.9]$, $r_5 = [11205.2, 19999.2]$, $r_6 = [11138.4, 19759.9]$, $r7 = [11036.2, 19723.8]$, and $r_8 = [10764.0, 19705.7]$ (Jaccard similarity 0.95, 0.90, 0.90, 0.90, 0.88, 0.89, 0.91, respectively). All the returned refinements are minimal. The differences in the results are due to the different minimality definitions: distance between queries v.s. distance in the result sets. Our definition of minimality allows the user to select the most suitable refinement based on their preferences. For instance, if the user prefers to minimize changes to the lower bound of the salary, $r_2$ would be the best refinement with the same best Jaccard similarity as $r_1$, but it would be ignored by [15].

*Running time comparison.* To compare the running time of the two algorithms, considering [15] uses different algorithms for single-attribute and multiple-attribute queries, we employ both the one-dimensional TexasTribune dataset used earlier and a two-dimensional uniform dataset from [15]. The uniform dataset consists of 10,000 points uniformly sampled within a square of length 1,000 units, with two numerical attributes ($x$ and $y$) and a sensitive attribute (color). For each dataset, we use two queries with different disparities. For TexasTribune dataset, we use queries 1) selecting salary $\in [5000, 15000]$, with a disparity of 62, and 2) selecting salary $\in [10000, 20000]$, with a disparity of 69. For uniform dataset, we use queries 1) selecting records with $x \in [400, 600]$ and $y \in [400, 600]$, with a disparity of 30, and 2) selecting records with $x \in [550, 700]$ and $y \in [550, 700]$, with a disparity of 35. We evaluate the running time for different original queries, setting target disparities at percentages of the original disparity as cardinality constraints (ranging from 80% to 20%). Figures 13 presents the comparison results with 1-attribute and 2-attribute queries, with the x-axis representing the target disparity and the y-axis showing the execution time. To account for the possibility of our algorithm outputting more than one refinement, the figures include a bar labeled "PVL-first," representing the time to find the first minimal refinement using our algorithm, in comparison with [15]. In general, the running time increases as the target disparities become

smaller, as satisfying harder constraints takes more time. For single-attribute queries, [15] maintains stable runtime because disparity has a limited impact on their algorithm, primarily depending on data size. In most cases, PVL-first and [15] exhibit comparable running times, with the same order of magnitude, indicating similar efficiency in finding the first minimal refinement. However, there is an exception in multiple-attribute queries, where [15] is much faster when the target disparity is high (90% or 80% of the original), but even so, both algorithms finish within 3 seconds. It is worth noting that besides comparable execution times in the covered settings of [15], our algorithm handles multiple relations, constraints, and non-binary sensitive attributes, providing more versatility than [15]. Additionally, our algorithm includes further optimizations to improve efficiency in some situations, as discussed in Section 5.2.

## 7 RELATED WORK

We next overview multiple lines of related work.

*Query refinement.* The problem of query refinement has been addressed in previous studies such as [4, 5, 15, 20–23]. They focus on modifying queries to satisfy cardinality constraints, mostly emphasizing the overall output size rather than specific data groups within the output. For example, [5, 20] aim to relax queries with an empty result set to produce some answers. Other works like [4, 21] address the issues of too many or too few answers by refining queries to meet specific cardinality constraints on the result's size. An exception is the recent work by Shetiya et al.[15], which aims to refine queries to satisfy constraints on the size of specific data groups in the result. We discuss and demonstrate the differences in Section6.4. Other related work by [24, 25] aims to satisfy cardinality constraints through minimal query refinement but does not consider categorical attributes and provides only cardinality estimations without exact results. Furthermore, the work of [22, 23] studies the problem of explaining missing tuples in the query result through query refinement. A key difference is that [22, 23] aim to include user-specified missing tuples in the result set through refinement, rather than imposing constraints over groups, where including or removing tuples not specified by the user may apply.

*Query result diversification.* Query result diversification is another related topic aiming to increase result diversity while maintaining relevance of results to the original query [26–29]. In some settings, relevance and diversity are tradeoffs [30], and in others, user information is used to select the result set. However, there are two main differences between this line of research and our approach. First, query result diversification involves ranking items based on their relevance to the query, aiming to select items that are ranked as high as possible and, at the same time, as diverse as possible. In contrast, our approach does not involve ranking: an item is either selected or not based on the query. Second, while query result diversification explores item selection based on the given query but without modifying the query, our approach achieves result diversity through query refinement itself.

*Provenance.* Data provenance has been studied extensively in recent years. An early approach to user provenance for RDBMS [31, 32] finds a set of tuples that contribute to a certain output tuple. The model proposed in [33] collects contributing input tuples separately for each step in the derivation of the output tuple so that it shows how the output is obtained. Then a general framework proposed in [13] introduced tuple annotations using a semiring $K$ to evaluate queries over annotated relations. The Caravan system [6] enables efficient what-if analysis by creating provisioned representations for hypothetical reasoning, eliminating the need to access original data or execute complex queries. MONDRIAN [34] proposes an annotation-oriented data model, facilitating manipulation and querying of both data and annotations. It allows annotations to be specified on sets of values, thus enabling efficient querying of information based on their association. Glavic et al. [35] outlined a propagation-based method for provenance generation through operator instrumentation. Their approach modifies operators to generate and transfer detailed provenance information across the query network. Our provenance model is inspired by [6], with tuple annotations propagated through query valuation as in [13].

*Fairness constraints in other settings.* While our work does not only apply to fairness context, fairness is an important topic in data selection and there are many prior works focusing on satisfying fairness constraints in some settings. Some works aim to satisfy fairness constraints under a certain diversity model [36–38], and others focus on fairness constraints in the optimization of an additive utility [39]. There is also some work on fairness in set selection and ranking [40], fairness in classification [41], and diversity in top-k results [42–44]. Other works consider constraint query languages [45], presenting a declarative language with extensions to SQL to support fairness constraints in queries [46].

## 8 CONCLUSION AND FUTURE WORK

In this paper, we address the problem of finding all minimal refinements of a given query that satisfy specific cardinality constraints for data groups within the result set. Unlike many previous studies that primarily examine query refinement based on the overall result set size, our focus lies in satisfying size constraints for specific data groups within the result set. Furthermore, our approach extends beyond existing research by supporting a wider range of queries and constraints. Specifically, we enable SPJ queries that involve multiple constraints on multiple relations, encompassing sensitive attributes that are more than binary, and supporting both query relaxations and query contractions. To achieve this, we have proposed a solution that uses a provenance model to annotate tuples in the source data with the necessary predicate information. The cardinality constraints are translated into algebraic expressions, allowing for efficient verification of constraint satisfaction without accessing the original data. Furthermore, we also propose a search algorithm that efficiently explores the search space and finds potential refinements, which are then verified using our proposed provenance-based approach. Experimental results demonstrate the effectiveness and efficiency of our solution on a variety of datasets and constraints. For future work, it would be beneficial to investigate the general applicability of the proposed approach to other SQL clauses. For instance, queries involving NOT operators would require searching the complementary space. Moreover, queries combining ranges with both AND and OR operators would translate provenance expressions that incorporate both logical AND and OR operations. We left these for future research.

## ACKNOWLEDGMENTS

This work was supported in part by NSF under grants 1916505, 1922658, 1934565, 2106176, and 2312931, and by ISF grant 2121/22.

# REFERENCES


[1] https://www.microsoft.com/en-us/research/academic-program/phd-fellowship/canada-us/.
[2] https://research.google/outreach/faq/?category=phd.
[3] https://en.wikipedia.org/wiki/Rooney_Rule.
[4] Chaitanya Mishra and Nick Koudas. Interactive query refinement. In *EDBT*, 2009.
[5] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. Relaxing join and selection queries. In *VLDB*, 2006.
[6] Daniel Deutch, Zachary G. Ives, Tova Milo, and Val Tannen. Caravan: Provisioning for what-if analysis. In *CIDR*. www.cidrdb.org, 2013.
[7] Babak Salimi, Johannes Gehrke, and Dan Suciu. Bias in OLAP queries: Detection, explanation, and removal. In *SIGMOD*, 2018.
[8] Bienvenido Vélez, Ron Weiss, Mark A Sheldon, and David K Gifford. Fast and effective query refinement. In *ACM SIGIR Forum*, volume 31(SI), pages 6–15. ACM New York, NY, USA, 1997.
[9] Jessie Ooi, Xiuqin Ma, Hongwu Qin, and Siau Chuin Liew. A survey of query expansion, query suggestion and query refinement techniques. In *2015 4th International Conference on Software Engineering and Computer Systems (ICSECS)*, pages 112–117. IEEE, 2015.
[10] Michael Ortega-Binderberger, Kaushik Chakrabarti, and Sharad Mehrotra. An approach to integrating query refinement in sql. In *International Conference on Extending Database Technology*, pages 15–33. Springer, 2002.
[11] Kaushik Chakrabarti, Kriengkrai Porkaew, and Sharad Mehrotra. Efficient query refinement in multimedia databases. In *ICDE Conference*, 2000.
[12] Yannis Papakonstantinou and Vasilis Vassalos. Query rewriting for semistructured data. *ACM SIGMOD Record*, 28(2):455–466, 1999.
[13] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*. ACM, 2007.
[14] Marina Drosou, HV Jagadish, Evaggelia Pitoura, and Julia Stoyanovich. Diversity in big data: A review. *Big data*, 5(2):73–84, 2017.
[15] Suraj Shetiya, Ian P Swift, Abolfazl Asudeh, and Gautam Das. Fairness-aware range queries for selecting unbiased data. In *Proc. of the Int. Conf. on Data Engineering, ICDE*, 2022.
[16] https://github.com/JinyangLi01/Query_refinement/blob/master/FullPaper/Query_Refinement.pdf.
[17] Tomasz Imielinski and Witold Lipski Jr. Incomplete information in relational databases. *J. ACM*, 31(4), 1984.
[18] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. Mlinspect: A data distribution debugger for machine learning pipelines. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2736–2739, 2021.
[19] Frances Ding, Moritz Hardt, John Miller, and Ludwig Schmidt. Retiring adult: New datasets for fair machine learning. *Advances in Neural Information Processing Systems*, 34, 2021.
[20] Ion Muslea and Thomas J Lee. Online query relaxation via bayesian causal structures discovery. In *AAAI*, pages 831–836, 2005.
[21] Wesley W. Chu and Qiming Chen. A structured approach for cooperative query answering. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):738–749, 1994.
[22] Quoc Trung Tran and Chee-Yong Chan. How to conquer why-not questions. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 15–26, 2010.
[23] 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.
[24] Chiara Accinelli, Barbara Catania, Giovanna Guerrini, and Simone Minisi. covrew: a python toolkit for pre-processing pipeline rewriting ensuring coverage constraint satisfaction. In *EDBT*, pages 698–701, 2021.
[25] Chiara Accinelli, Simone Minisi, and Barbara Catania. Coverage-based rewriting for data preparation. In *EDBT/ICDT Workshops*, 2020.
[26] Marcos R Vieira, Humberto L Razente, Maria CN Barioni, Marios Hadjieleftheriou, Divesh Srivastava, Caetano Traina, and Vassilis J Tsotras. On query result diversification. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1163–1174. IEEE, 2011.
[27] Kaiping Zheng, Hongzhi Wang, Zhixin Qi, Jianzhong Li, and Hong Gao. A survey of query result diversification. *Knowledge and Information Systems*, 51(1):1–36, 2017.
[28] Filip Radlinski, Robert Kleinberg, and Thorsten Joachims. Learning diverse rankings with multi-armed bandits. In *Proceedings of the 25th international conference on Machine learning*, pages 784–791, 2008.
[29] Reinier H Van Leuken, Lluis Garcia, Ximena Olivares, and Roelof van Zwol. Visual diversification of image search results. In *Proceedings of the 18th international conference on World wide web*, pages 341–350, 2009.
[30] Ting Deng and Wenfei Fan. On the complexity of query result diversification. *Proceedings of the VLDB Endowment*, 6(8):577–588, 2013.
[31] Yingwei Cui and Jennifer Widom. Practical lineage tracing in data warehouses. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*, pages 367–378. IEEE, 2000.
[32] Yingwei Cui, Jennifer Widom, and Janet L Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179–227, 2000.
[33] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In *International conference on database theory*, pages 316–330. Springer, 2001.
[34] Floris Geerts, Anastasios Kementsietsidis, and Diego Milano. Mondrian: Annotating and querying databases through colors and blocks. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 82–82. IEEE, 2006.
[35] Boris Glavic, Kyumars Sheykh Esmaili, Peter Michael Fischer, and Nesime Tatbul. Ariadne: Managing fine-grained provenance on data streams. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 39–50, 2013.
[36] Zafeiria Moumoulidou, Andrew McGregor, and Alexandra Meliou. Diverse data selection under fairness constraints. *arXiv preprint arXiv:2010.09141*, 2020.
[37] Elisa Celis, Vijay Keswani, Damian Straszak, Amit Deshpande, Tarun Kathuria, and Nisheeth Vishnoi. Fair and diverse dpp-based data summarization. In *International Conference on Machine Learning*, pages 716–725. PMLR, 2018.
[38] Ke Yang, Vasilis Gkatzelis, and Julia Stoyanovich. Balanced ranking with diversity constraints. *arXiv preprint arXiv:1906.01747*, 2019.
[39] Julia Stoyanovich, Ke Yang, and HV Jagadish. Online set selection with fairness and diversity constraints. In *Proceedings of the EDBT Conference*, 2018.
[40] Meike Zehlike, Ke Yang, and Julia Stoyanovich. Fairness in ranking, Part I: Score-based ranking. *ACM Comput. Surv.*, apr 2022.
[41] Alexandra Chouldechova and Aaron Roth. A snapshot of the frontiers of fairness in machine learning. *Commun. ACM*, 63(5):82–89, 2020.
[42] Albert Angel and Nick Koudas. Efficient diversity-aware search. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 781–792, 2011.
[43] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. Diversifying top-k results. *arXiv preprint arXiv:1208.0076*, 2012.
[44] Julia Stoyanovich, Sihem Amer-Yahia, and Tova Milo. Making interval-based clustering rank-aware. In Anastasia Ailamaki, Sihem Amer-Yahia, Jignesh M. Patel, Tore Risch, Pierre Senellart, and Julia Stoyanovich, editors, *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 437–448. ACM, 2011.
[45] Paris C Kanellakis, Gabriel M Kuper, and Peter Z Revesz. Constraint query languages. *Journal of computer and system sciences*, 51(1):26–52, 1995.
[46] Matteo Brucato, Azza Abouzied, and Alexandra Meliou. Package queries: efficient and scalable computation of high-order constraints. *The VLDB Journal*, 27(5):693–718, 2018.