Query Refinement for Diversity Constraint Satisfaction

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 permit conditions as part of the query predicate(s), and result subgroup size constraints are difficult or impossible to specify. 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.

ACM Reference Format:

. 2018. Query Refinement for Diversity Constraint Satisfaction. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

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. For instance, fellowship programs frequently require a diverse nominee pool, such that at least one nominee is a member of each demographic group that is underrepresented in a particular field [1, 2]. Relational databases are often used to select and prioritize candidates in such settings.

Traditional relational queries specify conditions as part of the query predicates, and output tuples that satisfy these predicates. If a query is used as part of some high-stakes selection process, then it would be natural to state diversity requirements as cardinality constraints over the presence of some demographic groups in the query result. For example, a company may want to invite at least three women and at least one individual of each race for job interviews. Unfortunately, such constraints are currently not supported by relational systems: We can specify criteria on which 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 for augmenting relational queries with group cardinality constraints to satisfy diversity requirements.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

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			

Figure 1: Job applicant dataset used in Example 1.1. Applicants selected by queries Q1, Q2, and Q3 are marked with ✓.

An important starting point for our work is the substantial literature on imposing constraints on the size of the overall query result with the help of *query refinement*—introducing slight modifications to the query so that the output satisfies the specified cardinality bounds [3–11]. However, cardinality constraints over specific groups in the query result are not supported by these methods. We further motivate the need for such constraints and give an overview of our proposed approach next, using an example.

Example 1.1. Figure 1 shows a dataset D of 14 job candidates applying to a tech company, with four attributes: gender, race, (college) major, and GPA. Of these, gender and race are the *sensitive attributes* that denote membership in demographic groups. These attributes may not be used directly in selection conditions of a query, to avoid direct discrimination in decision-making (in the US, this is based on the doctrine of disparate treatment). However, cardinality constraints may be stated w.r.t. such attributes, to counteract the effects of historical discrimination (in the US, this is based on the doctrine of disparate impact, and corresponds to an affirmative action-style intervention). Major and GPA are the *qualification attributes* that may be used directly as part of the selection process.

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 the corresponding column in Figure 1. Note that, although half of the applicants are female, only one of them is among the selected set. Further, although White, Black and Asian individuals are present among the applicants, there are no Asians among the selected set. To increase diversity, the company would like to interview at least two female applicants, and at least one applicant of each race. This requirement can be expressed as a set of cardinality constraints on groups in the query result. We call these *group cardinality constraints*. This can be accomplished by slightly adjusting the selection criteria, for example, by expanding the set of majors to also include Electrical Engineering (EE), thus *relaxing* the query:

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

Query Q2 selects applicants #5, #6, #7, #12 and #13 to be interviewed in addition to applicants #2, #9, and #10, as marked in Figure 1. The selected set includes four women and at least one individual of each race, and so the refined query satisfies 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 on the part of the company's human resources department, who then impose an additional requirement — to reduce 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 in ('CS', 'EE') AND a.GPA >= 3.90
```

The result of Q3 consists of four applicants, #2, #6, #9, and #13, with two of each gender and at least one of each race.

Diversity is a compelling need when distributing access to resources and opportunities in a society, as we see 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 [12]. Even if a user's stated preferences for location, amenities, and price may lead a hotel booking site to pick large downtown chain hotels for her trip, it is likely a good idea for the site to throw in a few smaller boutique hotel options into the result set as well. This allows the user to see how much more she would have to pay — in terms of worse locations, fewer amenities, or higher prices — to get a boutique hotel instead, and then make a decision about what she prefers. Group cardinality constraints are required in all of these situations.

One way to satisfy group cardinality constraints is to modify the result set directly in a post-processing step, adding or removing some tuples: Adding candidate #7, who is both female and Asian, to the result of Q1 Example 1.1 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.

In this paper, we study the problem of finding minimal refinements of a query so that the query result satisfies given cardinality constraints. Namely, we aim to modify predicates in the query just enough so that the query result meets the constraints, while preserving the intention of the original query as much as possible. There are two main challenges. First, to test whether a candidate query modification has a result that satisfies the constraints, we need to execute this query, which can be time-consuming, especially when the dataset is large or stored in a remote database. Second, there are many ways to change a predicate in a query, and there can be many predicates in the query, so the number of different possible changes is combinatorially large, making exhaustive analysis computationally expensive. To address the first challenge, based on [13], our provenance model annotates tuples in the source data with the necessary information with respect to the predicate, and translates cardinality constraints to algebraic expressions [5], so that it can be used to test whether the query satisfies the constraints. To address the second challenge and get a minimal refinement such that changing any predicate closer to the original one would fail to meet the constraints, we generate refinements based on how the constraints can be potentially satisfied.

Our work shares motivation with recent work by Shetya et al. [14], whose goal is also to satisfy group cardinality constraints of the kind we consider here over a query result. However, their work differs from ours in several important ways. First, they only handle constraints over a single binary sensitive attribute (e.g., male vs. female gender *or* majority vs. minority ethnicity). In contrast, we handle multiple sensitive attributes and do not limit them to be binary. Example 1.1 illustrates this, and includes two sensitive attributes: binary gender and non-binary race. Second, we support both query relaxation (i.e., generating more result tuples, as illustrated by query Q2) and query restriction (i.e., generating fewer result tuples, as illustrated by Q3), while Shetya et al. only support query relaxation. Third, their diversity objective is phrased in terms of minimizing the distance between the *result sets* produced by the original query and the rewritten query, while we aim to minimize the distance between the *queries themselves*. This difference may appear to be subtle, but it corresponds to intrinsically different objectives in the rewriting: 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.

Recall two refinements of query Q1 presented in Example 1.1: Q2 modifies one of the predicates of Q1, adding EE to the list of majors, while Q3 modifies both the predicate on major and the predicate on GPA. Shetya et al. [14] use Jaccard similarity to compare query results, computing: $sim_{Jacc}(\mathtt{Q1(D)}, \mathtt{Q2(D)}) = \frac{3}{8} = 0.375$ and $sim_{Jacc}(\mathtt{Q1(D)}, \mathtt{Q3(D)}) = \frac{2}{5} = 0.4$. Thus, they would consider Q3 to have higher similarity to Q1 over D , 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. We theoretically analyze the complexity of the problem, proving its hardness. In Section 3, we describe our provenance model based on [13]. Then, in Section 4, we present the Possible Value Lists (PVL), a data structure for representing the possible refinements based on provenance expressions, with the goal of improving the efficiency of search for minimal refinements. Our algorithm for generating minimal refinements utilizing the PVL is presented in Section 5, where we also propose optimization to the algorithm that can be applied in 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 start by presenting the notion of query refinement. We then introduce the cardinality constraints and formally define Query Refinement Problem. Finally, we prove that the problem as defined cannot be solved in polynomial time, in the worst case. Figure 2 summarizes the notations used in this paper.

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
$p.A, p.op, p.c$	attribute, operator, constant of p ($p \in Q_n$)
$p.C$	list of constants of p ($p \in Q_c$)
$\mathcal{G}$	conjunction of conditions
$Q(D)_{\mathcal{G}}$	size of the group of tuples in $Q(D)$ satisfying $\mathcal{G}$
Cr	cardinality constraints
$\mathcal{V}_{Q(D)}$	set of variables in provenance inequalities
$prov(t)$	annotation of tuple $t \in D$
$\mathcal{I}^{Cr}_{Q(D)}$	set of provenance inequalities
$val_{Q'}(A_v)$	valuation of $A_v \in \mathcal{V}_{Q(D)}$ with refinement Q'
$Tval_{Q'}(\mathcal{I}^{Cr}_{Q(D)})$	truth value of inequality set $\mathcal{I}^{Cr}_{Q(D)}$
$L_{D,Q}$	PVL given D, Q
l_A	a list in PVL
$Q.R$	list of indices representing Q in PVL $L_{D,Q}$
$Q\|_P$	partial query of Q containing predicates in set P

Figure 2: Notation table.

2.1 Query Refinement

We consider SPUJ (Select-Project-Union-Join) queries with selection predicates over numerical and categorical attributes. Predicates over numerical attributes are of the form `attribute <op> constant` where `<op>` can be one of $\{<, \leq, \geq, >\}$. For categorical attributes the selection predicates are of the form `attribute = constant_1 OR...OR attribute = constant_n`. For a query Q, we use Q_n and Q_c to denote the set of numerical and categorical predicates respectively. Furthermore, for a given numerical predicate $p \in Q_n$ we refer to its parts, i.e., the attribute, operator and constant using $p.A$, $p.op$ and $p.c$ respectively. For categorical predicates $p \in Q_c$ we use $p.A$ to denote the attribute in the predicate, and $p.C$ and the list of constants in the p, i.e., the set {`constant_1,...,constant_n`} where p is the predicate `attribute = constant_1 OR...OR attribute = constant_n`. We say two predicates p and p' are equal ($p = p'$) if all of their parts are equal.

We use the notion of query refinement defined in [3] 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.

Definition 2.1 (Predicate refinement (adopted from [3])). Let Q be a SPJU query, a refinement of numerical predicate $p_n \in Q_n$ is p'_n such that $p'_n.A = p_n.A$, $p'_n.op = p_n.op$, and $p'_n.c \neq p_n.c$; and a refinement of categorical predicate $p_c \in Q_c$ is p'_c such that $p'_c.A = p_c.A$ and $p'_c.C \neq p_c.C$.

Example 2.2. Consider the predicates in query `Q1` given in Example 1.1. For numeric predicate `a.GPA >= 3.85`, a refinement can be `a.GPA >= 3.90`. For categorical predicate `a.Major = 'CS'`, a refinement can be `a.Major = 'CS' or a.Major='EE'`.

The categorical refinement of [3] considers a hierarchy that also includes roll-up and drill-down. Indeed, we can convert any hierarchical categorical refinement to Definition 2.2. For instance, rolling up `Country = 'US' and (State = 'CA')` to get `Country = 'US'` is semantically equivalent to adding all the values in the domain of `State` to its conjunction. In the rest of the paper, for simplicity, we assume no hierarchy over categorical attributes.

Definition 2.3 (Query refinement). A query Q' is a refinement of query Q if Q' is obtained from Q by refining some predicates of Q.

Example 2.4. Consider again query `Q1` given in Example 1.1. We refine it by applying the refinement of the GPA predicate and Major predicate in Example 2.2 and get query `Q3`, which is a refinement of `Q1` with respect to the categorical attribute Major and the numerical attribute GPA.

2.2 Groups Cardinality Constraints

Let D be a dataset, Q a query, and $Q(D)$ the result of executing the query over D. We define groups in $Q(D)$ using a conjunction of conditions $\mathcal{G} = \wedge_i(A_i = v_i)$ where A_i are distinct data attributes in $Q(D)$. For instance, in our running example, one of the conditions $\mathcal{G}$ is $\{Gender = F\}$. We use $Q(D)_{\mathcal{G}}$ to denote the size of the group of tuples in $Q(D)$ that satisfy the condition $\mathcal{G}$. We can then define cardinality constraints over data groups as follows.

Definition 2.5 (Cardinality Constraints). Let D be a dataset, Q a query, and $\mathcal{G}$ a group definition condition set. A cardinality constraint Cr over $Q(D)_{\mathcal{G}}$ is an expression of the form $|Q(D)_{\mathcal{G}}|\ op\ x$, where $op \in \{\leq, <, >, \geq\}$ and x can be a constant or $\alpha \cdot |Q(D)_{\mathcal{G}'}|$ for some other data group defined using $\mathcal{G}'$ and $\alpha \in \mathbb{R}$. We say that $Q(D)$ satisfies Cr if the inequality holds.

Example 2.6. Continuing with our example, the cardinality constraints over the number of females can be formally expressed as $Q(D)_{Gender=F} \geq 2$, and the cardinality constraints that the total number of applicants selected is no greater than 5 can be expressed as $Q(D)_{\emptyset} \leq 5$.

Given a dataset D, a query Q, and a cardinality constraint Cr such that $Q(D)$ does not satisfy Cr, there may be multiple ways to refine Q in order to satisfy the constraint as we demonstrate next.

Example 2.7. In Example 1.1, the results of the query `Q2` or `Q3`, which are both refinements of `Q1`, satisfy cardinality constraint $Q(D)_{Gender=F} \geq 2$ (at least 2 females). Another plausible refinement of `Q1` that qualifies at least two females is to adjust the GPA predicate to be `a.GPA >= 3.40`.

The refinements depicted in the above example suggest the company may have various ways to achieve its diversity goal. Each applies different modifications to the query. Intuitively, minimal modifications to the original query are preferred, e.g., a query that refines the predicate GPA to be `a.GPA >= 3.6` is preferred over one that refines the predicate GPA to be `a.GPA >= 3.55`, however, refinements that modify different attributes may be incomparable when no additional preferences information is provided by the end-user. To this, we define the set of minimal refinements. We first define the domination between predicate refinements as follows.

Definition 2.8 (Domination in Refinement). Let Q be a query with a predicate p and let p', p'' be two refinements of p. For numerical predicate $p \in Q_n$, we say p' dominates p'', if $|p.c - p'.c| < |p.c - p''.c|$. For categorical predicate $p \in Q_c$, we say p' dominates p'', if $p.C \setminus p'.C \subsetneq p.C \setminus p''.C$ and $p'.C \setminus p.C \subsetneq p''.C \setminus p.C$. Let Q' and Q'' be two refinements of Q. We say that Q' dominates Q'' (with respect to Q) if $\forall p' \in Q'$, for $p'' \in Q''$ such that $p'.A = p''.A$, we have $p'' = p'$ or p' dominates p''.

Example 2.9. In our running example, the original GPA predicate in `Q1` is `a.GPA = 3.85`. Refinement `a.GPA = 3.90` dominates refinement `a.GPA = 3.60`. For Major predicate `a.Major = CS`, refinement `a.Major = EE` dominates `a.Major = EE or a.Major = ME`.

Definition 2.10 (Minimal Refinement). Given a dataset D, a query Q, and a (set of) cardinality constraint(s) over data group(s) Cr, we say that Q' is a minimal refinement of Q with respect to Cr if (i) Q' is a refinement of Q, (ii) $Q'(D)$ satisfies Cr and (iii) $\nexists Q''$ such that Q'' satisfies conditions (i) and (ii), and Q'' dominates Q'.

The problem we aim to solve in this paper is to find all minimal refinements of a query with respect to a set of cardinality constraints, defined as follows.

Problem 2.11 (Query Refinement Problem). *Given a dataset D, a SPUJ query Q, and a set of cardinality constraints Cr, find all minimal refinements of Q with respect to Cr.*

A naive solution to the Query Refinement Problem would be traversing all possible refinements to see whether the result satisfies the cardinality constraints. There is a result set R maintained during the whole process, and for each refinement r whose result satisfies the constraints, if r is not dominated by any refinement in R, r is added to R. We next present our solution which largely reduces the redundant information by using provenance.

2.3 Hardness Analysis

Theorem 2.12. *Given a dataset with n tuples, N attributes, a SPUJ query, and a set of cardinality constraints, no polynomial time algorithm can guarantee the enumeration of the set of all minimal refinements.*

Proof. We prove the theorem by constructing an example with an exponential number of minimal refinements. Consider a dataset with n tuples and query Q with N predicates over attributes $A_1 \cdots A_N$. Assume all N predicates are numeric predicates where the value of attribute $A_i (1 \leqslant I \leqslant N)$ can be either 0 or 1. Without loss of generality, we assume $n > N$ and $N\%2 = 0$. Figure 1 shows the dataset, where the values of an attribute A_i for a tuple t is represented as

	A_1	A_2	A_3	$\dots$	A_N	Gender
t_1	0	1	1	$\dots$	1	F
t_2	1	0	1	$\dots$	1	F
t_3	1	1	0	$\dots$	1	F
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	F
t_N	1	1	1	$\dots$	0	F
t_{N+1}	0	0	0	0	0	F
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	F
t_n	0	0	0	0	0	F

Table 1: Dataset for proof

either 1 or 0, meaning the value of $t.A_i$. For t_i where $1 \leq i \leq N$, $t_i.A_i = 0$, and $t_i.A_j = 1, j \neq i$. That is, for $t_1 - t_N$, only values on the diagonal are 0. For t_i where $i \geq N+1$, $t_i.A_j = 0, 1 \leq j \leq N$.

As in Figure 1, all n tuples have attribute gender as female, and we assume the cardinality constraint is to have at least $N/2$ females. With the original query Q, no female is selected, so we need to refine Q so that at least $N/2$ tuples satisfy the query. Since all values are binary, a refinement is to refine some predicates from $p.c = 1$ to $p.c = 0$. Any refinement that refines $N/2$ of the predicates to be $p.c = 0$ is a minimal refinement, for the following two reasons. First, such a refinement R makes $N/2$ of the tuples among $t_1 - t_N$ selected by the query, thus satisfying the cardinality constraint. Second, for any refinement R' dominated by R, R' has at most $N/2-1$ predicates refined with $p.c = 0$, making at most $N/2 - 1$ tuples among $t_1 - t_N$ selected by the query, and zero tuples among $t_{n+1} - t_n$ selected, thus not satisfying the cardinality constraint.

Therefore, any refinement refining half of the predicates is a minimal refinement, so the number of minimal refinements is $\binom{N}{N/2} = O(\sqrt{2}^N)$. □

3 PROVENANCE MODEL

Given a query, the number of possible refinement queries can be extremely large. Furthermore, determining whether the output of a refinement query satisfies the cardinality constraints requires the evaluation of the query over the data. This process can be expensive, particularly for large or remote databases. However, this costly query evaluation process can be eliminated using data annotations developed in the context of 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 [5]. Intuitively, we generate provenance inequalities, using the annotated data, the given query, and the cardinality constraints. We leverage the idea of conditional tables (c-tables) [15], where tuples are associated with conditions. To capture the possible refinements, we annotate tuples in the data with the query selection conditions. Finally, we follow the semiring model [13] to propagate the annotations through query valuation.

3.1 Provenance inequalities

We first define a set of variables used to construct the provenance expressions. This set is determined by the given query Q and the data D.

Definition 3.1. 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 3.2. In our running example, we have two attributes in the query: Major and GPA, with three and six values, respectively. We use M and G as short for Major and GPA.

$$\mathcal{V}_{Q(D)} = \{M_{CS}, M_{EE}, M_{ME}, G_{3.95}, G_{3.90}, G_{3.85}, G_{3.65}, G_{3.60}, G_{3.40}\}$$

We use the set of variables $\mathcal{V}_{Q(D)}$ to generate provenance annotations for each tuple $t \in D$. Intuitively, when applying a selection predicate over the data, instead of actually deleting tuples that do not meet the selection criteria, we annotate them with expressions that can be used to determine whether it satisfies any possible refinement query of Q.

Definition 3.3. Let Q be a query over D, the annotation of each tuple $t \in D$ is

$$prov(t) = \prod_{i=1}^{i=k} A_{i[t.A_i]}$$

Example 3.4. Continue with our running example. The annotations of applicants #2 and #6 are as follows.

$$prov(t_2) = M_{CS} \cdot G_{3.95}$$
$$prov(t_6) = M_{EE} \cdot G_{3.90}$$

Finally, using the resulting provenance expression we define inequality expressions that express the cardinality constraints.

Definition 3.5 (Provenance inequality of a constraint). 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 the provenance inequality of the constraint as

$$\left(\sum_{t \in Q(D)_{\mathcal{G}}} prov(t)\right) op\ x$$

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 inequalities

Example 3.6. Consider again our running example with the data in Figure 1 and query Q1. The set of provenance inequalities $\mathcal{I}^{Cr}_{Q(D)}$ has the following five inequalities, corresponding to five constraints in turn: at least two females, at least one white, at least one black, at least one Asian, and at most five qualified candidates.

$$\begin{aligned} &M_{ME} \cdot G_{3.65} + M_{CS} \cdot G_{3.95} + M_{CS} \cdot G_{3.40} + M_{ME} \cdot G_{3.60} \\ &+ M_{EE} \cdot G_{3.85} + M_{EE} \cdot G_{3.90} + M_{EE} \cdot G_{3.85} \geq 2 \end{aligned} \quad (1)$$

$$\begin{aligned} &M_{ME} \cdot G_{3.65} + M_{CS} \cdot G_{3.95} + M_{ME} \cdot G_{3.60} + M_{EE} \cdot G_{3.85} \\ &+ M_{CS} \cdot G_{3.65} + M_{CS} \cdot G_{3.90} + M_{CS} \cdot G_{3.40} + M_{EE} \cdot G_{3.85} \geq 1 \end{aligned} \quad (2)$$

$$M_{CS} \cdot G_{3.40} + M_{EE} \cdot G_{3.90} + M_{CS} \cdot G_{3.85} + M_{ME} \cdot G_{3.60} \geq 1 \quad (3)$$

$$M_{EE} \cdot G_{3.85} + M_{EE} \cdot G_{3.95} \geq 1 \quad (4)$$

$$\begin{aligned} &M_{ME} \cdot G_{3.65} + M_{CS} \cdot G_{3.95} + M_{CS} \cdot G_{3.40} + M_{ME} \cdot G_{3.60} \\ &+ M_{EE} \cdot G_{3.85} + M_{EE} \cdot G_{3.90} + M_{EE} \cdot G_{3.85} + M_{CS} \cdot G_{3.65} \\ &+ M_{CS} \cdot G_{3.90} + M_{CS} \cdot G_{3.85} + M_{CS} \cdot G_{3.40} + M_{EE} \cdot G_{3.85} \\ &+ M_{EE} \cdot G_{3.95} + M_{ME} \cdot G_{3.60} \leq 5 \end{aligned} \quad (5)$$

Note that each term in the provenance inequalities is in the form of $\prod x_i$, where x_i is a provenance variable.

3.2 Refinements through valuation

We now establish the connection between possible refinements and valuations of the provenance inequality of cardinality constraints. Intuitively, a refinement that satisfies a given cardinality constraint should correspond to a valuation that satisfies the inequalities. For a given query Q over the data D, each possible refinement query Q' corresponds to an assignment to the variables set $\mathcal{V}_{Q(D)}$.

Definition 3.7 (Valuation). Let $A_v \in \mathcal{V}_{Q(D)}$ and Q' be a refinement query of Q.

$$val_{Q'}(A_v) = \begin{cases} 1 & \text{if } p \in Q'_n \text{ s.t. } p.A = A, v\ op\ p.c \\ 1 & \text{if } p \in Q'_c \text{ s.t. } p.A = A, v \in p.C \\ 0 & \text{otherwise} \end{cases}$$

By assigning the value $val_{Q'}(A_{iv})$ to each variable in the provenance inequality, the above value assignment 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 inequalities. Given a set of provenance inequalities $\mathcal{I}^{Cr}_{Q(D)}$ and the value assignment $val_{Q'}$, we use $Tval_{Q'}(I)$ to denote the truth value of the inequality $I \in \mathcal{I}^{Cr}_{Q(D)}$, 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 inequality 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 3.8. Consider the provenance inequality Equation 1 (denoted as I_1) and Equation 2 (denoted as I_2) from Example 3.6. With the original query Q1 (`a.Major = 'CS' and a.GPA >= 3.85`), the following three valuations of variables are 1, and others are 0: $val_{Q1}(M_{CS}), val_{Q1}(G_{3.95}), val_{Q1}(G_{3.90})$. Thus the truth values of I_1 and I_2 are false and true, respectively: $Tval_{Q1}(I_1) = False, Tval_{Q}(I_2) = True$, so the truth value of the inequality set $\{I_1, I_2\}$ is false: $Tval_{Q1}(\mathcal{I}^{\{I_1,I_2\}}_{Q(D)}) = False$. The refinement query Q2 in Example 1.1 flips the valuation of variable M_{EE} from 0 to 1, thus flips the truth value of I_1 from false to true while maintaining the truth value of I_2 as unchanged. As a result, the truth value of the inequality set is true: $Tval_{Q2}(\mathcal{I}^{\{I_1,I_2\}}_{Q(D)}) = True$.

PROPOSITION 3.9. *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 inequalities 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 inequalities 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 and 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 inequality, 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. Recall that a query can be refined by disjunctively adding and/or removing values from the original categorical predicate lists in the query, or by varying the constants in the conditions over numerical attributes. Moreover, for numerical predicates, minimal refinements can only consist of numerical values from the data. E.g., in our running example, a refinement query that refines the predicate GPA to be `a.GPA >= 3.45` can not be minimal since it results in the same output as refining the predicate GPA to be `a.GPA >= 3.60`, since no applicant has a GPA such that $3.45 \leq GPA < 3.60$, but the latter dominates the former.

Given a dataset D and a query Q, a PVL denoted as $L_{D,Q}$, depicts the possible modifications using lists of possible values. For each numerical predicate $p_n \in Q_n$, $L_{D,Q}$ consists of a list consisting $p_n.c$ and all the values of $p_n.A$ in D, denoted as $l_{p_n.A}$. Note that depending on $p_n.op$ 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. For example, 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)`. For each categorical predicate $p_c \in Q_c$, $L_{D,Q}$ consists of a set of lists, one for each possible value v in the domain of $p_c.A$, denoted as $l_{p_c.A_v}$, which includes two possible values, 1 and 0, that stands for the existence (1) or absence (0) of v in $p_n.C$. Intuitively, this set of lists corresponding to a categorical predicate p_c can be used to describe the set of all possible values' subgroups of attribute $p_c.A$.

The values in the lists are sorted based on their distance from the values in the original query Q. For $p_n \in Q_n$, values in list $l_{p_n.A}$ are sorted by the distance from $p_n.c$, such that values closer to C_i are placed higher. For $p_c \in Q_c$, similarly, $l_{p_c.A_v} = [1, 0]$ if $v \in p_c.C$, and $l_{p_c.A_v} = [0, 1]$, otherwise. 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 3a depicts the PVL $L_{D,Q1}$ of our running example with dataset D and query Q1. 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.

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

(a) The PVL $L_{D,Q}$ of the running example (Example 1.1)

	$l_{M_{EE}}$	$l_{M_{ME}}$	$l_{M_{CS}}$
1	0	0	1
2	0	1	0

(b) The PVL of Example 5.3

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

(c) The PVL of Example 5.6

Figure 3: Possible value lists (PVL).

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

(1) Every minimal refinement query Q 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 $L_{D,Q}$ such that $Q'.R[l]$ is the index in the list $l \in L_{D,Q}$. $\forall p_n \in Q'_n, \exists l_{p_n.A} \in L_{D,Q}, p_n.c = l_{p_n.A}[Q'.R[l_{p_n.A}]]$. $\forall p_c \in Q'_c, \exists l_{p_c.A_v} \in L_{D,Q}$ 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 $Q' \in \mathbf{Q}$.

Example 4.3. Consider again the running example with PVL in Figure 3a. Refinement query Q2 that adds EE as a recognized major can be represented as $Q2.R = [1, 2, 1, 1]$, corresponding to values $l_G[1] = 3.85, l_{M_{EE}}[2] = 1, l_{M_{ME}}[1] = 0, l_{M_{CS}}[1] = 1$ The refinement query Q3 in Example 1.1 can be represented as $Q3.R = [2, 2, 1, 1]$, as highlighted in blue. Also, index list $[5, 1, 2, 2]$ corresponds to the refinement query with predicates `a.GPA >= 3.60 AND a.Major = 'ME'`.

Our algorithm for generating minimal refinements utilizes the PVL to traverse the possible refinements. The representation of refinements using the PVL holds the following properties that are the basis for the correctness of our algorithm.

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

(1) Q_1 dominates Q_2 if and only if $\forall j, 1 \leq j \leq n_l, Q_1.R[j] \leq Q_2.R[j]$.
(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 $Q_1.R[j'] > Q_2.R[j']$

In the next section, we will explain how to search for minimal refinements with PVL.

5 GENERATING MINIMAL REFINEMENTS

We now present the algorithm for minimal query refinements search using the PVL. We first describe the algorithm and then present optimization that can be applied in some cases

5.1 The algorithm

The algorithm uses the notion of partial queries 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 our running example 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.

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 $Res$
```

Finding a minimal refinement (from a partial query). Given a partial query $Q'|_P$ (where Q' is a refinement of Q), the algorithm generates minimal refinement queries by completing the predicates of P to obtain a refinement of the input query Q. This is done by traversing the PVL in a top-down fashion, i.e., starting from the values closest to the original predicate, so that the first refinement satisfying cardinality constraints must be a minimal refinement (since no refinements located above it satisfies them).

Recursive search. The core of the algorithm is a recursive search function depicted in Algorithm 1. The function gets as input a partial query $Q|_P$ with some predicate set P that is already fixed (as part of a minimal refinement), a temporary result set Res, a PVL L, and a set of provenance inequalities. `Search` first finds a minimal refinement Q_b using the procedure `findRefinement` (line 1). If there is no such refinement, the current result set Res is returned. Otherwise, the result set is updated with the new refinement Q_b (line 3), and the algorithm uses it to generate new refinements by adding a predicate with a value from a list $l \in L$ to the predicates set P, removing the list l from the PVL and recursively search for new refinements (lines 4 – 13). Specifically, for the list $l \in L$, we consider every possible predicate that can be generated using values located above $Q_b.R[l]$ in l. This procedure continues as long as the PVL consists of more than one list.

Reducing the search space. When we traverse the possible refinements, the search space can be reduced using the notion of *partial dissatisfaction* 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 (we call such constraints *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 $Tval_{Q'}(\mathcal{I}^{Cr}_{Q(D)}) = False$. Namely, if a partial query of Q' does not satisfy some relaxation constraints, then Q' does not satisfy the whole constraint set.

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 with the initial parameters $L_{D,Q}$, an empty result set Res, a partial query $Q|_P$ with $P = \emptyset$, and $\mathcal{I}^{Cr}_{Q(D)}$. `Search` calls `findRefinement` as explained above and use it to generate new refinements. The search is optimized by avoiding refinements that can not satisfy the constraints based on partial dissatisfaction (lines 9 – 10). Finally, to avoid repetitive checking, at the end of the recursive call with values in list l, we remove all these values, i.e., remove values above index $Q_b.R[l]$ in this list (line 13). This is because refinements with these values are already checked.

Example 5.3. Consider again our running example with the data in Figure 1, the original query `Q1` and five constraints from Example 1.1. The provenance inequalities of those constraints are given in Example 3.6, Equation 1 – Equation 5. We build PVL shown in Figure 3a. Note that 3.95 and 4.00 in l_G, highlighted in grey, should be removed because it dissatisfies Equation 1, i.e., $Tval_{Q'|_P}(I_1) = False$, where $Q'|_P$ is a partial query with predicate `a.GPA >= 3.95` (or `a.GPA >= 4.00`), to reduce the search space. We search in PVL for base minimal refinement by a top-down traversal, and get Q_b s.t. $Q_b.R = [2, 2, 1, 1]$, as highlighted in blue color. It refines the Major predicates to be `a.Major in ('EE', 'CS')`. Any other minimal refinement must have some values above these blue values in PVL, based on Proposition 4.4, so we check the value in $l_G[1]$ and $l_{M_{EE}}[1]$. For value in $l_G[1]$, we fix the GPA predicate with it, i.e., `a.GPA >= 3.85`, get a new PVL in Figure 3b by removing list l_G from PVL, and do the search in the same way recursively. There is no base minimal refinements found in Figure 3b, thus this recursion ends without new results. Similarly, we fix the Major predicate with value in $l_{M_{EE}}[1]$, i.e., let Major not contain EE, get a new PVL by removing list $l_{M_{EE}}$ from PVL in Figure 3a, and do the search in the same way recursively, but getting no more minimal refinements either. The final result set contains one minimal refinement.

Now we are ready to prove that our algorithm can find all minimal refinements without reporting any non-minimal refinements, as in the following theorem.

THEOREM 5.4. *Given a dataset D, a SPUJ 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.*

Proof. The second half of the theorem is trivial because we always check a refinement against the current result set before adding it, anything reported by the algorithm must be a minimal refinement. We prove the first half of the theorem by induction. Based on Algorithm 1, the lists in PVL are predicates that remain to be relaxed, and Q'_P is a partial query with predicates P that are already fixed. We aim to prove each iteration of Search(L, Res, Q'_P) can find all minimal refinements.

Base case: When there is only one list in PVL, there is only one predicate p remaining to refine with at most one minimal refinement. If there is a refinement of p with which the updated query $Q'|_P$ have $Tval_{Q'|_P}(\mathcal{I}^{Cr}_{Q(D)}) = True$, it can be found out by the search in Algorithm 1, line 1.

Induction step: Assume that with an x-list PVL, Algorithm 1 can find all minimal refinements. We need to prove it still can with an $(x+1)$-list PVL. Suppose there are y minimal refinements with an $(x+1)$-list PVL, where $y \geq 1$. One of these y minimal refinements will be found as our base minimal refinement Q_b. For each of the other $(y-1)$ minimal refinements Q, compared with Q_b, Q is refined less w.r.t. some lists, and more w.r.t. some other lists, based on Proposition 4.4. So there is at least one list l in PVL such that $Q.R[l] < Q_b.R[l]$. Let l be the left-most list satisfying this condition. If l is a numerical list corresponding to predicate p, according to Algorithm 1, we will get updated partial query $Q'|_{P'}$ by adding predicate p' s.t. $p'.A = p.A, p'.op = p.op, p'.c = l[Q.R[l]]$. If l is a categorical list corresponding to A_v of predicate p, similarly, we update partial query $Q|_P$ by adding or not adding v to $p.C$ depending on the value to fix is 1 or 0. And then we do iteration Search($L'_{D,Q}, Res, Q'|_{P'}$), where $L'_{D,Q}$ is the PVL without list l. Based on our induction hypothesis, Search($L'_{D,Q}, Res, Q'|_{P'}$) can find all minimal refinements since $L'_{D,Q}$ has x lists. Thus, Q can be found by Search($L'_{D,Q}, Res, Q'|_{P'}$). Removing values above the base minimal refinement Q_b (line 13 in Algorithm 1) does not affect generating Q, because they are removed after being checked already. □

5.2 Optimizations

We next describe further optimizations that can be applied in the case where all inequalities are of the form $C(D)_{\mathcal{G}}\ op\ x$, where x is a constant and $op \in \{>, \geq\}$ (or $op \in \{<, \leq\}$) for *all* the constraints. Namely, all the constraints are relaxation constraints (or contraction constraints in the case where $op \in \{<, \leq\}$). 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 affect the refinement, and thus can be excluded from the provenance inequalities and the PVL. Additionally, when all the constraints are relaxation constraints, the procedure of finding a minimal refinement from a partial query, and the recursive searches can be optimized due to the following monotonicity property of relaxation queries.

Proposition 5.5. *Let D be a dataset, Q be a query, and $L_{D,Q}$ 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 refinement can be quickly found. Instead of a top-down search traversing all possible refinement, we optimize this process by applying a binary search to locate a refinement that satisfies the cardinality constraints and then "tightening" it to make it minimal. This process is depicted in Algorithm 2. We first use a binary search (line 2) to get 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.

Algorithm 2: Find a minimal relaxation

```
input  : A $PVL\ L$ and cardinality constraints $\mathcal{I}$.
output : A minimal relaxation $\mathcal{R}$
1  Procedure findRelaxation($L, \mathcal{I}, Q|_P$)
2      $Q_b \leftarrow$ binarySearch($L, \mathcal{I}, Q|_P$)
3      if $Q_b.R \neq \emptyset$ then
4          foreach $l \in L$ do
5              $foundMin \leftarrow False$
6              while $Q_b.R[l] \geq 1$ or $foundMin$ do
7                  $Q_b.R[l] \leftarrow Q_b.R[l] - 1$
8                  if $Tval_{Q_b}(\mathcal{I}) = False$ then
9                      $Q_b.R[l] \leftarrow Q_b.R[l] + 1$
10                     $foundMin \leftarrow True$
11     return $Q_b$
```

Example 5.6. Consider again our running example with the original query Q1. To show a simple example with relaxation constraints, we only consider the constraint that there should be at least two females selected, with inequality in Equation 1. As mentioned before, when all constraints are relaxation constraints, provenance inequalities can be simplified by removing tuples satisfying query already, so Equation 1 becomes

$$M_{ME} \cdot G_{3.65} + M_{CS} \cdot G_{3.40} + M_{ME} \cdot G_{3.60}$$
$$+ M_{EE} \cdot G_{3.85} + M_{EE} \cdot G_{3.90} + M_{EE} \cdot G_{3.85} \geq 1$$

Moreover, PVL does not need to contain values or lists that already satisfy Q1, so compared to Figure 3a, the PVL in Figure 3c does not have list $l_{M_{CS}}$, and does not have values greater than 3.85 in list l_G. 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 $Q'_b.R = [1, 2, 1]$, 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 experimentally evaluate the proposed solutions, showing the usefulness and effectiveness of our approach to finding minimal refinements for SPUJ queries to satisfy a given set of cardinality constraints. We start with our experiment setup and then present a quantitative experimental study whose goal is to assess the efficiency of our algorithm. . In particular, we examine our algorithm's performance with different datasets, queries, and cardinality constraints, and compare it to the naive algorithm. We further study the effect of the data size, the selectivity of the given query, and the constraint properties on the running time of the algorithm.

6.1 Experiment Setup

Datasets: We used three real-world datasets with different numbers of tuples and attributes as follows.

- **The Adult dataset**[1] has information about 48,842 individual's annual income based on the census data. It was collected and used to explore the possibility of predicting income level (over/under $50K$ a year) based on the individual's personal information. We use attributes capital gain, capital loss, educational number, occupation group, and hours-per-week in queries; and use race, gender, age group, work class, and marital status as sensitive attributes in cardinality constraints.
- **The Compas Dataset**[2] was collected and published by ProPublica as part of their investigation into racial bias in criminal risk assessment software. It contains the demographics, recidivism scores produced by the COMPAS software, and criminal offense information for 6,889 individuals.
- **The Healthcare dataset**[3] is a dataset used in [16] to train a classifier identifying patients at risk for serious complications. It contains demographic and clinical history information of 887 patients, with attributes: number of children, income, number of complications, county, race, age group, and smoker or not. We use the attributes income, number of children, and complications in the queries; and use race and age group to define the cardinality constraints. We round all the incomes to thousands.

Compared algorithms: We evaluate the performance, in terms of the running time of our proposed algorithms, and compare the result to baseline solutions for each problem as follows.

[1]https://archive.ics.uci.edu/ml/datasets/adult
[2]https://www.propublica.org/datastore/dataset/compas-recidivism-risk-score-data-and-analysis
[3]https://github.com/stefan-grafberger/mlinspect/tree/master/example_pipelines/healthcare

- **Provenance Search (PS).** The generalized algorithm searches for minimal refinements using provenance inequalities described in Section 5.
- **Naive Algorithm (Naive).** The baseline (naive) algorithm for minimal refinements searching is described at the end of Section 2.2.

Queries and constraints: We evaluated the performance of the algorithm using various queries and cardinality constraints. For each dataset, we use representative queries and cardinality constraints to report the results. The predicates used for the queries, denoted as Q_1, Q_2 and the constraints C_1, C_2, C_3 are detailed in Figure 4. For each dataset, the constraints in C_1 are relaxations constraints, the constraints in C_2 are contraction constraints, and C_3 is a general set of constraints, consisting of a mixture of relaxation and contraction constraints. Similar results were observed with other queries and constraints.

All experiments were performed on a macOS machine with a 2.8 GHz Quad-Core Intel Core i7 CPU and 8GB memory. The algorithms were implemented using Python3.7. We set a 5-min time out for each algorithm.

Dataset	Queries' Predicates	Cardinality Constraints
Adult	Q_1: education-num >= 14 hours-per-week >= 21 capital-gain >= 5501	C_1: $Female \geq 80$
		C_2: $Female \leq 20$ $Male, White \leq 180$
	Q_2: education-num >= 15 hours-per-week <= 40 relationship in {'H', 'W', 'O'}	C_3: $Female \geq 30$ $Male, White \leq 200$
Compas	Q_1: juv-fel-count >= 1 decile-score >= 4 r-charge-degree in {'F'}	C_1: $Caucasian \geq 23$ $older\ than\ 45 \geq 15$
		C_2: $Af-American \leq 60$ $Male \leq 90$
	Q_2: juv-fel-count <= 4 decile-score >= 8 c-charge-degree in {'O'}	C_3: $Af-American \geq 90$ $younger\ than\ 25 \leq 39$
Healthcare	Q_1: income >= 100K num-children >= 3 county in {'county2', 'county3'}	C_1: $race2 \geq 50$ $race3, age-group1 \geq 10$
		C_2: $race2 \leq 30$
	Q_2: income >= 150K num-children >= 4 complications <= 8 county in{ 'county2', 'county4'}	C_3: $race = race3 \geq 17$ $race2, age-group2 \leq 10$

Figure 4: Queries and constraints

6.2 Experiment Results

Running time. The first set of experiments aims at assessing the running time for different scenarios. To this end, we execute our algorithm and compare its running with the baseline naive algorithm. The running times for all six combinations of the two queries and three cardinality constraint sets are shown in Figure 5. The blue bar depicts the running time of our solution (provenance search) and the green bar shows the running time for the naive algorithm. To better analyze the running time of our algorithm, we present a breakdown that shows the time spent on each one of the two parts of our solution. The dark blue part depicts the provenance generation time and light blue presents the searching time for minimal refinements using the PVL. $QxCy$ on the x-axis means we use query Q_x and cardinality constraints C_y of the corresponding dataset. Our algorithm significantly outperforms the naive solution in all cases and runs up to more than 100 times faster. Moreover, in most cases, the naive algorithm was unable to terminate within the 5-min time out (and thus is omitted from the graphs). This is because our solution searches through fewer refinements with the help of the PVL.

In general, the results show our solution can handle various flavors of constraint sets (relaxations, contractions, and mixture thereof). We note that the running time for the general case, where the constraints include both relaxations and contractions (C_3), the running time is relatively longer. This demonstrates the usefulness of the optimizations described in Section 5.2, which are applicable only for the case where all the constraints are in the form of relaxations or contractions.

The running time is affected by multiple properties of the query and the cardinality constraints. In the next sets of experiments, we examine the effect of different properties of the problem's input on the running time.

Query Selectivity. Intuitively, the selectivity of the query (with respect to the constraints) determines the search space the algorithm traverses to find refinement queries. When increasing the selectivity of the query for relaxation, we expect the search space to increase, as it starts from refinements close to the original query. Thus we expect to see an increase in the running time. For contraction constraints, we expect to see an opposite trend. For the general case with both, relaxation and contraction, the trend may change depending on the constraints and the data. To examine the effect of query selectivity, we varied the constant in the constraints. Figures 6, 7, and 8 show the result for Q_1 for each dataset, when varying selectivity of the query by modifying the a value of single predicate in it. For example, we vary the income predicate (for Q_1 in the Healthcare dataset) from `income >= 0` to `income >= 450K` and observed the effect on the running time.

In these figures, the x-axis is the value of the constant used in the predicate of the corresponding query. Higher values correspond to higher selectivity. The y-axis is the running time. Figures 6a, 7a and 8a show the running time when the constraint set consists only of relaxation constraints (i.e., C_1). As expected, we observed an increase in the running time in all cases. For example, in Figure 6a, with a higher threshold of hours-per-week, there are fewer people selected by the query, thus fewer females in the result and the farther the output from satisfying the constraint that at least 80 females are selected. Similarly, for constraint set with contraction constraints (i.e., C_2), we observed a decrease in the running time (see Figures 6b, 7b and 8b).

For the general case (constraint set C_3), when increasing the education-num threshold for Q_1 in the Adult dataset, we observed an increase in the running time and then a decrease as shown in Figure 6c. As we increase the education-num threshold, there are fewer people selected, thus the output is farther from satisfying constraints over the number of female ($Female \geq 30$) in it, but closer to the constraint over the number of white males ($Male, White \leq 200$). This trade-off between the constraints can then explain the change in the running time. For lower selectivity, the running time is mostly affected by the time it takes to satisfy the relaxation constraint, while for higher selectivity the trend is changing and most of the effort is in satisfying the contraction constraint. In Figure 7c (Q_1 for the Compas dataset), we observe a similar trend although the change in the running time is minor (less than a second), because changes of decile-score in the query has a low effect on the satisfaction of the constraints set (C_3) and the search space for refinement queries remains roughly the same. Finally, Figure 8c presents the results for Q_1 in the Healthcare dataset, where the running time is decreasing. This indicates that the time consumed for the satisfaction of the contraction constraint ($race = race3 \geq 17$) dominates the overall running time of the algorithm.

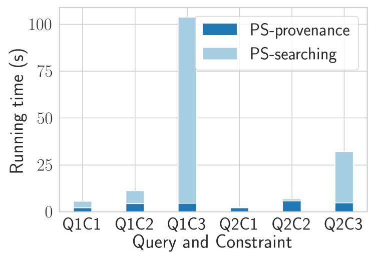
Query structure (number of predicates). Besides the value of constant in the predicates, the number of predicates in the query also affects the running time. Figure 9 shows the running time of the PS and naive algorithms on the healthcare dataset as a function of the number of query predicates, for query Q_2 in the healthcare dataset. I.e., we used subsets of the predicates shown in Figure 4. Recall that the search space is exponential in the number of predicates, thus we expect to see a rapid increase in running time. Indeed, the running time of both algorithms rapidly increases, however, the running times of our algorithm are significantly better than the naive solution. The running time of the naive algorithm and PS for up to 3 predicates for the constraints C_1 (Figure 9a) are similar since the constraints are already satisfied by the queries and no searching is required (the running time is only the construction of the provenance inequalities).

Constraints satisfaction properties. To study the effect of the cardinality constraint on the running time, we vary the constant of one inequality in the cardinality constraint and see how the running time changes. Different from the above sets of experiments, changing the constant in the cardinality constraint does not change the PVL, which depends only on the dataset and query. However, intuitively, as the constraints are restricted (or relaxed), the running time is expected to increase (or decrease) as the queries satisfying the constraints are farther (or closer) to the original query. Figure 10 shows the result for modification in one of the constraints in the constraints sets C_1 and C_2 on the running time of the algorithms for queries Q_1 and Q_2 on the Healthcare dataset. The running time observed was pretty stable in most cases, indicating the modifications to the constraints have a relatively low effect. We observed a slight increase when restricting the constraint for the constraint C_2 for Q_2 (Figure 10d).

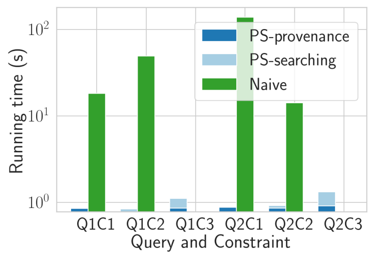
7 RELATED WORK

There are multiple lines of related work.

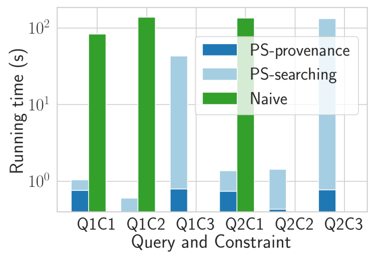
Query refinement. There are several works providing methods for query refinement by slightly modifying the query so that output size satisfies the cardinality constraints. Typically, the focus is on the size of the whole output, but not the size of some data groups in the output, and thus cannot be applied to our problem. For example, the work in [4, 17] focuses on relaxing failed queries that return an empty result and aims to have the relaxed queries yield some answers. Other works in [3, 18] study many/few answers problems by refining the query to satisfy some cardinality constraint on the size of the result. Recent work by Shetya et al. [14] is an exception in that it aims to refine queries to satisfy constraints on the size of some data groups in the result, rather than the cardinality of the entire result. They propose an approach to refine range queries, but differ from our methods in several ways, including that they consider only numerical attributes and do not support categorical attributes, and



(a) Adult dataset

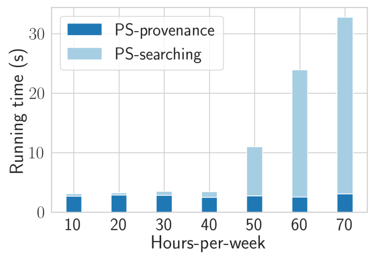


(b) Compas dataset (log scale)

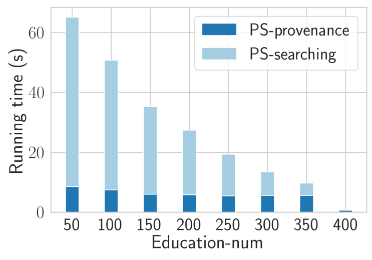


(c) Healthcare dataset (log scale)

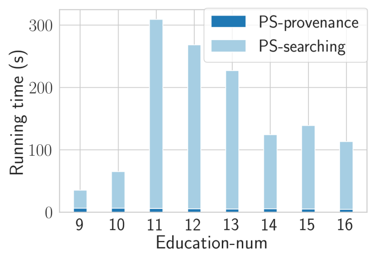
Figure 5: Running time of different queries and constraint sets



(a) Q1 varying hours-per-week, C1

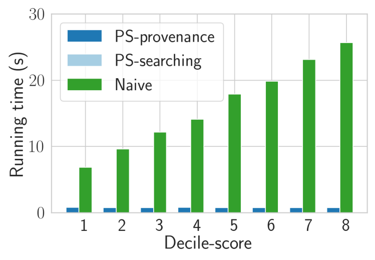


(b) Q1 varying education-num, C2

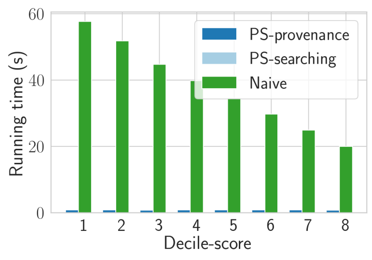


(c) Q1 varying education-num, C3

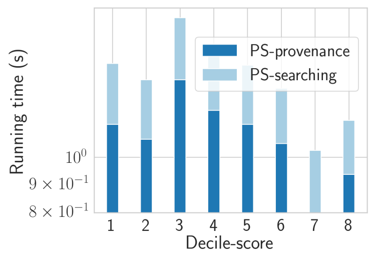
Figure 6: Effect of query selectivity on the running time, Adult dataset



(a) Q1 varying decile-score, C1

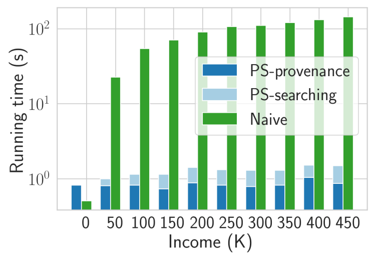


(b) Q1 varying decile-score, C2

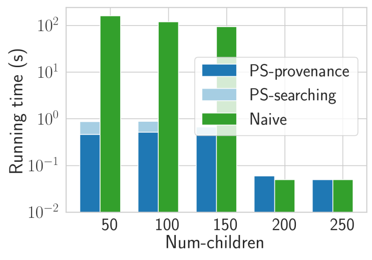


(c) Q1 varying decile-score, C3

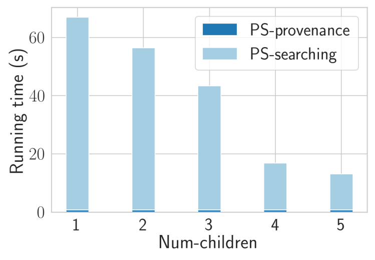
Figure 7: Effect of query selectivity on the running time, Compas dataset



(a) Q1 varying income, C1



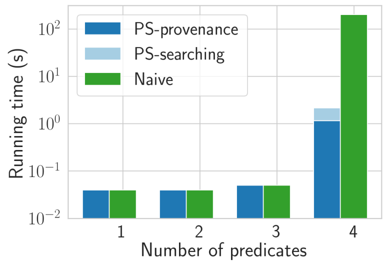
(b) Q1 varying num-children, C2



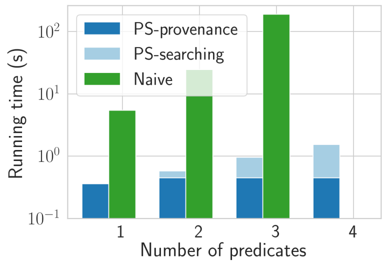
(c) Q1 varying num-children, C3

Figure 8: Effect of query selectivity on the running time, Healthcare dataset

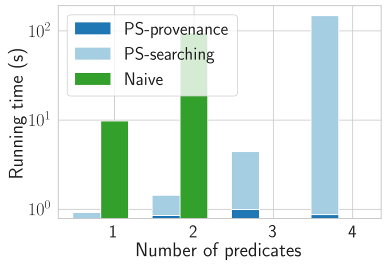
they do not support more than one constraint. Another difference is that they have a different definition of minimality: they minimize the difference between query result sets rather than between queries themselves. We discussed these and other distinctions between our work and Shetiya et al., in the Introduction. The approach proposed by Accinelli et al. [19, 20] also aims to satisfy cardinality constraints by minimally refining queries, with a definition of minimality the same as ours: minimizing the difference between queries. But they do not consider categorical attributes either, and they use cardinality estimation which means the correctness and minimality of the result



(a) Varying number of predicates in Q2, C1

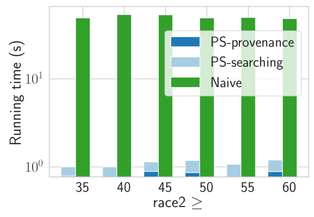


(b) Varying number of predicates in Q2, C2

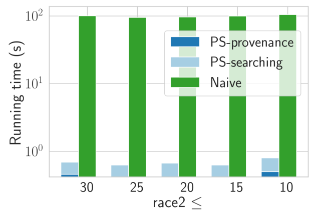


(c) Varying number of predicates in Q2, C3

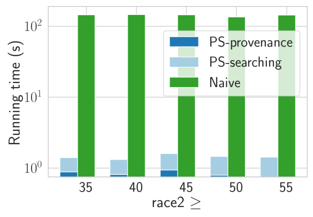
Figure 9: Effect of number of predicates on the running time, Healthcare dataset



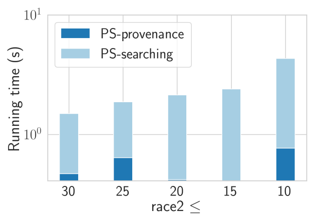
(a) C1 varying lower bound of race2, Q1



(b) C2 varying upper bound of race2, Q1



(c) C1 varying lower bound of race2, Q2



(d) C2 varying upper bound of race2, Q2

Figure 10: Effect of cardinality constraints on the running time, healthcare dataset

are not 100% guaranteed. We are different from previous works in that 1) we consider both numeric and categorical attributes; 2) we minimize the difference between queries and 3) we support a more general form of cardinality constraints such that there can be more than one constraint and a constraint can represent the comparison relationship between some data groups.

Query result diversification. Query result diversification is another related topic aiming to increase the diversity in the query result while maintaining the relevance of results to the original query [21–24]. In some settings, relevance and diversity are trade-offs [25], and in others, user information is used to select the result set. However, there are two main differences between this line of work and ours. First, there is a ranking in query result diversification: items are ranked by their relevance to the original query. The goal is to select items that are ranked as high as possible and at the same time as diverse as possible. On the contrary, there is no ranking in our setting: an item is either selected or not by the query. Second, while query result diversification studies how to select items based on the given query without changing the query, we achieve a diverse result set by refining the query itself.

Provenance. There are also several recent works studying provenance and its application in data science. An early approach to user provenance for RDBMS [26, 27] finds a set of tuples that contribute to a certain output tuple. The model proposed in [28] 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 is proposed in [13] that annotates tuples with elements from a semiring K so that queries can be evaluated over annotated relations. The Caravan system proposed in [5] enables what-if analysis by creating provisioned representations to efficiently compute hypothetical reasoning, so that what-if scenarios can be evaluated without accessing the original data or executing complex queries. Our provenance model is inspired by [5], with tuple annotations propagated through query valuation 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 [29–31], and others focus on fairness constraints in the optimization of an additive utility [32]. There is also some work on fairness in set selection and ranking [33], fairness in classification [34], and diversity in top-k results [35–37]. Other works consider constraint query languages [38], presenting a declarative language with extensions to SQL to support fairness constraints in queries [39].

8 CONCLUSION

In this paper, we have studied the problem of finding all minimal refinements of a given query so that the result set satisfies given cardinality constraints that are on the size of some data groups in the result. Our proposed solution uses a provenance model to annotate tuples in the source data with the necessary information with respect to the predicate and translates cardinality constraints to algebraic expressions, so that whether or not constraints are satisfied can be checked without checking the original data. We also propose a search algorithm that efficiently searches for potential refinements to satisfy the constraints. We experimentally examine our solution and show that it works efficiently to find all minimal refinements.

REFERENCES

[1] https://www.microsoft.com/en-us/research/academic-program/phd-fellowship/canada-us/.

[2] https://research.google/outreach/faq/?category=phd.

[3] Chaitanya Mishra and Nick Koudas. Interactive query refinement. In *EDBT*, 2009.

[4] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. Relaxing join and selection queries. In *VLDB*, 2006.

[5] Daniel Deutch, Zachary G. Ives, Tova Milo, and Val Tannen. Caravan: Provisioning for what-if analysis. In *CIDR*. www.cidrdb.org, 2013.

[6] Babak Salimi, Johannes Gehrke, and Dan Suciu. Bias in OLAP queries: Detection, explanation, and removal. In *SIGMOD*, 2018.

[7] Bienvenido Vélez, Ron Weiss, Mark A Sheldon, and David K Gifford. Fast and effective query refinement. In *ACM SIGIR Forum*, volume 31, pages 6–15. ACM New York, NY, USA, 1997.

[8] 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.

[9] 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.

[10] Kaushik Chakrabarti, Kriengkrai Porkaew, and Sharad Mehrotra. Efficient query refinement in multimedia databases. In *ICDE Conference*, 2000.

[11] Yannis Papakonstantinou and Vasilis Vassalos. Query rewriting for semistructured data. *ACM SIGMOD Record*, 28(2):455–466, 1999.

[12] Marina Drosou, HV Jagadish, Evaggelia Pitoura, and Julia Stoyanovich. Diversity in big data: A review. *Big data*, 5(2):73–84, 2017.

[13] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*. ACM, 2007.

[14] 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.

[15] Tomasz Imielinski and Witold Lipski Jr. Incomplete information in relational databases. *J. ACM*, 31(4), 1984.

[16] 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.

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

[18] 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.

[19] 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.

[20] Chiara Accinelli, Simone Minisi, and Barbara Catania. Coverage-based rewriting for data preparation. In *EDBT/ICDT Workshops*, 2020.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Ting Deng and Wenfei Fan. On the complexity of query result diversification. *Proceedings of the VLDB Endowment*, 6(8):577–588, 2013.

[26] 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.

[27] 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.

[28] 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.

[29] Zafeiria Moumoulidou, Andrew McGregor, and Alexandra Meliou. Diverse data selection under fairness constraints. *arXiv preprint arXiv:2010.09141*, 2020.

[30] 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.

[31] Ke Yang, Vasilis Gkatzelis, and Julia Stoyanovich. Balanced ranking with diversity constraints. *arXiv preprint arXiv:1906.01747*, 2019.

[32] Julia Stoyanovich, Ke Yang, and HV Jagadish. Online set selection with fairness and diversity constraints. In *Proceedings of the EDBT Conference*, 2018.

[33] Meike Zehlike, Ke Yang, and Julia Stoyanovich. Fairness in ranking, Part I: Score-based ranking. *ACM Comput. Surv.*, apr 2022.

[34] Alexandra Chouldechova and Aaron Roth. A snapshot of the frontiers of fairness in machine learning. *Commun. ACM*, 63(5):82–89, 2020.

[35] 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.

[36] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. Diversifying top-k results. *arXiv preprint arXiv:1208.0076*, 2012.

[37] 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.

[38] Paris C Kanellakis, Gabriel M Kuper, and Peter Z Revesz. Constraint query languages. *Journal of computer and system sciences*, 51(1):26–52, 1995.

[39] 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.