Fairness-Aware Range Queries for Selecting Unbiased Data

Suraj Shetiya†, Ian P. Swift‡, Abolfazl Asudeh‡, Gautam Das†
†*University of Texas at Arlington*, ‡*University of Illinois at Chicago*
†{suraj.shetiya@mavs., gdas@}uta.edu, ‡{iswift2, asudeh}@uic.edu

***Abstract*—We are being constantly judged by automated decision systems that have been widely criticised for being discriminatory and unfair. Since an algorithm is only as good as the data it works with, biases in the data can significantly amplify unfairness issues.**

In this paper, we take initial steps towards integrating fairness conditions into database query processing and data management systems. Specifically, we focus on selection bias in range queries. We formally define the problem of fairness-aware range queries as obtaining a fair query which is most similar to the user's query. We propose a sub-linear time algorithm for single-predicate range queries and efficient algorithms for multi-predicate range queries. Our empirical evaluation on real and synthetic datasets confirms the effectiveness and efficiency of our proposal.

I. Introduction

In the era of big data and advanced computation models, we are all constantly being judged by the analysis, algorithmic outcomes, and AI models generated using data about us. Such analysis are valuable as they assist decision makers take wise and just actions. For example, the abundance of large amounts of data has enabled building extensive big data systems to fight COVID-19, such as controlling the spread of the disease, or in finding effective factors, decisions, and policies [1]. Similar examples can be found in almost all corners of human life including resource allocation and city policies, policing, judiciary system, college admission, credit scoring, breast cancer prediction, job interviewing, hiring, and promotion, to name a few. In particular, let us consider the following as a running example:

Example 1. *(Part 1) Consider a company that would like to make a policy decision, targeted at its "profitable" employees. Following our real experiment in § V-B, suppose the company has around 150K employees. Using salary as an indicator of how profitable an employee is, the business management office of the company considers the query* `SELECT * FROM EMP WHERE salary≥$65K`*, which includes around 18% of employees. Surveying this group, the company wants to develop some mechanisms to motivate and retain these employees.*

Looking at these analyses through the lens of fairness, algorithmic decisions look promising as they seem to eliminate human biases. However, *"an algorithm is only as good as the data it works with"* [2]. In fact, the use of data in all aforementioned applications have been highly criticised for being discriminatory, racist, sexist, and unfair [3], [4]. Probably the main reason is that real-life social data is almost always "biased" [2]. Using biased data for algorithmic decisions create fairness dilemmas such as impossibility and inherent trade-offs of fairness [5], [6], [7]. Besides historical biases and false stereotypes reflected in data, other sources such as *selection bias* can amplify unfairness issues [2]. To highlight a real example, let us continue with Example 1:

Example 1. *(Part 2) As we shall later elaborate in § V-B, it turns out the company has more female employees than male. Still, due to the known historical discrimination [8], the selected group of employees contain noticeably more males. As a result, targeting this group for the analysis, the company will end up favoring the preferences of the male employees, which is unfair to female employees and will, in a feedback loop, result in losing more of the "profitable" female candidates.*

Fortunately, recently different computer science communities, such as machine learning and, in particular, data management, have taken fairness issues seriously. In past three years alone, there have been many publications in related topics such as fairness-aware data repair, cleaning, and integration [9], [10], [11], [12], data bias detection/resolution [13], [14], [15], [16], [17], [18], [19], and data/model annotation [20], [21], systems [22], [23], [24], ranking [25], [26], [27], [28], crowdsourcing [29], as well as different keynotes [30], [31] and tutorials [32], [33], [3] dedicated to this topic in premier database conferences, that underscore this community's role in properly addressing this problem.

Despite extensive efforts within the database community, there is still a need to integrate fairness requirements with database systems. Existing work is limited to query formulation for achieving data coverage (minimum count on demographic (sub)groups) [34], [35], [36]. To our knowledge, this paper is *the first to integrate fairness (parity on counts) with (selection) query answering*. In particular, as our first attempt, we consider range queries and pay attention to the facts: (i) the conditions in the range query may be selected intuitively by the human user. For instance, in Example 1 the user could have chosen $65K as the query bound because it was (roughly) a good choice that would make sense for them; (ii) considering the ethical obligations and consequences, the user might be interested in accepting a "similar enough" query to their initial choice, if it returns a "fair" outcome.

In Example 1, we note that the company could, for instance, in a post-query processing step, remove some male employees from the selected group, or it could add some females to the selected pool, even though they do not belong to the query

result. While such fixes are technically easy, those are illegal in many jurisdictions [37], because those amount to *disparate-treatment* discrimination: "when the decisions an individual user receives change with changes to her sensitive attribute information" [38], [2], [25]. For instance, one cannot simply increase or decrease the grade of a student, because of their race or gender. Instead, they should design a "fair rubric" that is not discriminatory. Therefore, instead of practicing disparate treatment, we propose to adjusting the range to find a range (similar to finding a rubric for grading) with a fair output.

Following the above argument, our system allows the user to specify the fairness and similarity constraints (in a declarative manner) along with the selection conditions, and we return an output range that satisfies these conditions. To further clarify this, let us continue with Example 1 in the following.

EXAMPLE 1. *(Part 3) Being aware of the historical discrimination, ethical obligations, and the potential negative impacts on the company, besides knowing that the choice of salary lower-bound has been fuzzy, the business management office would like to find a query whose output is similar enough to the initial query and the number of male employees returned is at most 1000 (around 5%) more than the females. Using our system, they can issue a SQL query to find such a set. As explained in § V-B, our system found the most similar fair range as* `SELECT * FROM EMP WHERE $60.5K≤salary≤$152K`. *Its outcome is 75% similar to the initial range query, and satisfies the fairness requirement. Observing the high Jaccard similarity between these two sets, the company now has the option to use this for their analysis, to make sure they are not discriminating against their female employees, hence not losing their valuable candidates.*

Our system provides *an alternative* to the initial query provided by the user. This is useful since often the choice of filtering ranges is ad-hoc, hence our system helps the user responsibly tune their range. If the discovered range is not satisfactory to the user, they can change the fairness and similarity requirements and *explore different choices* until they select the final result in a responsible manner [39], [25].

Summary of contributions: Initiating fairness-aware query answering, in this paper we tackle non-trivial technical challenges and propose proper algorithms to address them. In particular, we make the following contributions in this paper:

- We initiate research on integrating fairness conditions into database query processing and data management systems. While the specific problem investigated in our paper focuses on fairness in range queries, we hope this work will spur further research in this important and emerging field.
- We study the problem of fairness-aware range queries. That is, finding the most similar fair range to a user-provided range query for the database $\mathbb{D}$. We propose using SQL for declaring the fairness-aware queries.
- For single-predicate (SP) range queries, we propose the algorithm *SPQA* with *sub-linear* query time. The algorithm uses an innovative linear-size index *Jump pointers*.
- We model the problem for multi-predicate (MP) range queries as the traversal over a graph where nodes represent different queries and there is an edge between two nodes if their outputs differ by one tuple. In particular, we propose *Best First Search MP (BFSMP)* algorithm that, starting from the input range, efficiently explores neighbouring nodes to find the most similar fair range.
- Inspired by the A* algorithm, we propose *Informed BFSMP* that improves *BFSMP*, using an upper-bound on the *Jaccard similarity* for effective graph exploration.
- We conduct comprehensive experiments to evaluate our SP and MP algorithms. Our results demonstrate the efficiency and efficacy of *SPQA* for SP queries. Similarly, *IBFSMP* performs well for MP queries.

id	A_0	A_1	color	id	A_0	A_1	color
t_0	3.1	1.5	red	t_7	13	5.4	red
t_1	0.7	2.3	red	t_8	11.3	2.6	blue
t_2	8	0.65	blue	t_9	2.3	8.4	blue
t_3	10.9	1.5	red	t_{10}	5.6	4.7	red
t_4	4.4	8.7	blue	t_{11}	12.7	2.8	red
t_5	1.2	4.1	red	t_{12}	7	0.3	blue
t_6	6.2	6.3	blue	t_{13}	9.1	9.4	red

TABLE I: A toy example database $\mathbb{D}$ with two attributes A_0 and A_1 and the sensitive attribute *color*.

II. PRELIMINARIES

A. Database Model

Database: We consider a relation $\mathbb{D}$ with *n* objects, t_1 to t_n. Each object in $\mathbb{D}$, consists of numeric attributes A_1 to A_d. We refer to the value of attribute $A_j \in \mathbb{R}$ of object t_i as $t_i[j]$. While the numeric attributes of the database $\mathbb{D}$ can be used for range queries, objects may also consist of categorical attributes. These categorical attributes are used for filtering the objects based on the user's criteria.

A toy example of a database can be seen in Table I with 14 objects t_0 to t_{13}. The attributes of this database are in the form of A_i, namely A_0, A_1. The database also includes the non-ordinal attribute color used in the fairness model. We use this database for illustrating various techniques across the paper.

Range Query: Given a database $\mathbb{D}$, a range query is made up of conjunction of range constraints placed on some/all of the attributes of $\mathbb{D}$. A range constraint $\{start, end\}$ (aka a range predicate) on an attribute A_i filters the objects such that the attribute A_i lies within the filter range($start \leq A_i \leq end$). For instance, Query 1 is a query with one range predicate on the toy example of Table I that returns the objects $\{t_4, t_6, t_{10}, t_{12}\}$:

QUERY 1: `select id from` $\mathbb{D}$ `where` $4 \leq A_0 \leq 7$

In this paper, we consider the conjunction of multiple range predicates using "and" operation.

Similarity Measure: The distance between two range queries is the dissimilarity of the two queries. Without loss of generality, the distance measured between two range queries can be normalized to lie in the range $[0, 1]$. The similarity between two range queries can be computed as one minus dissimilarity. Various similarity models can be used to measure the similarity between two range queries. Without loss of generality, we use *Jaccard similarity* and leave other models for future work.

Jaccard similarity between two range queries can be computed by the ratio of intersection between the output of the two range queries to the union of the output to the two range queries.

$$\text{SIM}(Q_1, Q_2) = \frac{out(\mathbb{D}, Q_1) \cap out(\mathbb{D}, Q_2)}{out(\mathbb{D}, Q_1) \cup out(\mathbb{D}, Q_2)}$$

where *out* is the output of the range query on $\mathbb{D}$.

QUERY 2: `select id from` $\mathbb{D}$ `where` $3 \leq A_0 \leq 6.2$

For instance, the similarity between the example queries Query 1 and Query 2 ($out(\mathbb{D}, \text{Query2}) = \{t_0, t_4, t_6, t_{10}\}$) is:

$$\text{SIM(Query 1, Query 2)} = \frac{|\{t_4, t_6, t_{10}\}|}{|\{t_0, t_4, t_6, t_{10}, t_{12}\}|} = 0.6$$

B. Fairness Model

Our definition of fairness is based on group fairness [40] and the notion of demographic parity, aka statistical parity and disparate impact [41], [42], [40], [2], [3].

Sensitive attribute: Group fairness is defined as parity over different demographic groups such as `white` and `black`. The demographic groups are identified by a non-ordinal attribute, such as `race` or `gender`, known as *sensitive attributes*. In many of the existing applications, sensitive attributes are binary, separating a minority group (e.g. `female`) from the majorities (e.g. `male`). Therefore, in this paper, we follow the existing work such as [43] and consider the sensitive attribute to be binary in nature. We leave the non-binary sensitive attributes and more general cases for the future work.

As highlighted in our sample database in table I, we use the attribute `color` (with two values `red` and `blue`) to abstract the sensitive attribute (and the demographic groups).

Fairness constraint: The fairness measure is defined as the parity over the demographic groups, identified by the colors `blue` and `red`. The parity condition is identified using a criteria that decides whether the output of a query is fair. Let C_r and C_b be the number of red and blue objects in the output.

In some application, the parity can be defined as having equal number of objects from the demographic groups in the output set. That is, $C_r = C_b$. In other words, the objects in the selected set should have equal chance of belonging to each demographic group. For instance, Query 1 in our toy example, returns three blue objects ($\{t_4, t_6, t_{12}\}$) and one red object ($\{t_{10}\}$) and does not satisfy the parity condition $C_r = C_b$, while Query 2 returns two blue ($\{t_4, t_6\}$) and two red object ($\{t_0, t_{10}\}$) – hence satisfies the parity condition.

Alternatively, some applications consider the underlying distributions and require that the set of selected objects to represent the underlying demographic from which they were chosen from. In other words, the objects from different demographic groups should have equal chances of being selected in the output set. That is, $C_r/n_r = C_b/n_b$, where n_r and n_b are the total number of `red` and `blue` objects in $\mathbb{D}$. Similarly, different applications may require different notions of parity based on societal norms. To support all these cases under the same fairness model, we abstract the fairness constraint, using a weight parameter W as following: $W_r C_r = W_b C_b$. Specifically, we refer to the case where $W_1 = W_2$ as *unweighted fairness* and other cases as *weighted fairness* model.

FAIR RANGE QUERY PROBLEM: *Given a database $\mathbb{D}$, a range query Q and a disparity value ε, find a fair range that is most similar to Q with a disparity value at most ε.*

Fig. 1: Problem Formulation

DECLARATIVE FAIRNESS-AWARE QUERY:

```
SELECT ... FROM DATABASE
WHERE
    RANGE-PREDICATES
SUBJECT TO
    |W_r C_r - W_b C_b| <= eps  and SIM >= tau
```

Fig. 2: Declarative Query Model

Achieving perfect demographic parity in form of equality is rarely practical in the real-world, hence we use a threshold ε to identify an acceptable disparity [41]. Using this threshold, the fairness constraint can be rewritten as

$$|W_r C_r - W_b C_b| \leq \varepsilon \tag{1}$$

C. Problem definition

Having formally defined the database and fairness notions, we are now ready to provide our problem formulation. We consider the problem of finding the most similar fair range to a user provided range query for the database $\mathbb{D}$. Figure 1 provides the formal formulation of the fair range query problem. This problem formulation helps the data scientists to slightly change their query to find the data that is similar to their initial query output and is also fair.

Declarative query model: Our problem formulation follows a *declarative fairness-aware query model* as specified in Figure 2. Using this declarative interface, we expect the user to easily formulate the fairness-aware queries. In particular, we realize that the user might not be interested to accept the queries that are far from their initial choices, hence might require to identify a constraint on the minimum similarity they would find relevant. This, along with the fairness constraint, is identified as part of the constraints, followed after the "subject to" phrase. Note that there may be multiple queries *equally* near to the input range query that satisfy the constraints. The problem of finding all fair range queries nearest to the given query is an interesting direction for future work, discussed in section VII. For example, knowing that Query 1 does not satisfy the demographic parity for unweighted fairness, the user can reformulate the query in form of Query 3 to discover a similar query (with at least 80% similarity) that has at most a disparity of 1.

QUERY 3: `select id from` $\mathbb{D}$ `where` $4 \leq A_0 \leq 7$ `subject to` $|C_b - C_r| \leq 1$ `and` SIM $\geq 80\%$

Formulating Query 3 as fair range query problem, the optimal solution, by changing the range predicate to $3.1 \leq A_0 \leq 7$, adds t_0 to the output set, satisfying both the fairness and similarity constraints specified by the user.

After providing the terms and discussing the problem formulation, we now turn attention to designing efficient algorithms for our problem. In particular, we realize that a large portion of the queries in practice have a single range predicate. Therefore, in § III, we focus on this case, designing a tailored solution for it. Then in § IV, we will devise an algorithm for the queries with multiple range predicates. We show empirical results in § V.

III. Single-predicate Range Queries

Next, we consider queries with a single range condition. We first provide definitions and theorems that form the basis for our algorithms, in § III-A. Then we design *SPQA*, our algorithm for the unweighted case in § III-B, discuss preprocessing details in § III-C, and present weighted *SPQA* in § III-D.

A. Jump pointers

Query answering systems usually conduct offline preprocessing (indexing) that facilitates online query answering. One extreme approach for finding fair range queries, that optimises for query time, is to precompute and store the answer to all possible range queries during preprocessing. Such an approach, using proper data structures, enables constant-time query answering. This, however, requires an extensive space of $\mathcal{O}(n^2)$ to store the answer to all possible single-predicate range queries, which might not be reasonable, specifically for databases with millions of objects. The other extreme is to optimize for the space and to delay the computation to online query answering time. This, however, might require enumerating $\mathcal{O}(n^2)$ possible ranges, which makes query answering inefficient – $\mathcal{O}(n^2)$. Our proposal is between these two extremes, by building a *linear*-size index (*Jump pointers*) that enables the *sublinear* query answering time of $\mathcal{O}(\log n + disparity)$, where $disparity$ is the unfairness of the input query.

The idea behind the *Single Predicate Query Answering* (*SPQA*) algorithm is to quickly lookup fair ranges, each of which have a similarity of ε. Theorem 1 proves that for any unfair unweighted query, the nearest fair query has a disparity value of exactly ε. Among the fair ranges which have a disparity of ε, the ones which have a potential to be nearest by *Jaccard similarity* to the input range are explored by *SPQA* to find the most similar fair range.

Definition 1. *(Jump pointers): Consider a database $\mathbb{D}$ and the attribute A for the single-predicate range query model. A* right *(resp.* left*)* blue *jump pointer from location o_i points to the nearest/closest location b_r(resp. b_l) such that the number of blues in the range $[o_i + 1, b_r]$ (resp. $[b_l, o_i - 1]$) is equal to the number of reds plus one. Red jump pointers are also defined in the same manner.*

For the sample database of Table I, Figure 3a depicts the right and left jump pointers for attribute A_0. The index is constructed on top of the sorted list of object ids according to their values on A_0. Therefore, since $t_1[0] = 0.7$ is the minimum of A_0, t_1 is the first object in the list. For example, consider the object t_{12}, where $t_{12}[0] = 7$; the range $[8, 10.9]$ ($8 \leq A_0 \leq 10.9$) consists of the smallest range starting from 8 that has one additional *red* than the *blues* in the range. Hence, its right *red* jump pointer points to t_3 ($t_3[0] = 10.9$). Note that not all objects have jump pointers; for example there is no *right red* jump pointer from t_{10} as no location ahead of t_{10} has one additional *red* that than the number of *blues* in the same range. Even though there exists four jump pointers ({left *or* right} and {blue *or* red}) for every object in the list, two of those pointers are trivial. For example, the range $[8, 8]$ ($8 \leq A_0 \leq 8$) consists of t_2 and is the smallest range after t_{12} that consists of one additional *blue*. As this trivial information can be quickly determined in $\mathcal{O}(1)$ time, this pointer need not maintained and can be looked up at query time. Left jump pointer follows a similar pattern; for example the node t_8 maintains a left *blue* pointer to t_6 as the range $[6.2, 10.9]$ ($6.2 \leq A_0 \leq 10.9$) consist of one additional *blue* than *reds*.

We refer to travelling along the jump pointer from location o_i as following a jump pointer. Following a *blue jump pointer* k times from location o_i gives us the closest location o_j from o_i such that the range from o_i to o_j has k *blues* more than the range that would end at o_i. The algorithm to find jump pointers is described in § III-C.

Lemma 1. *Following the red colored jump pointer k times from o_i lands at a location where the range ending at it is has k more reds than the same range ending at o_i.*

Proof. We provide the proof by induction:
Base case: By the definition of jump pointer, a *red* colored jump pointer points to the nearest location which has one additional red. This gives us the base case for $k = 1$.

Induction step: Assume that the lemma holds for $k - 1$ *red* jump pointers. Let the $k - 1^{th}$ jump pointer point at location o_l and k^{th} jump pointer point at location o_m. Suppose there exists a $o_{m'}$ which is closer to o_i than o_m while also satisfying the k additional reds criteria. If $o_{m'}$ lies to the left of o_l then we already have a contradiction as the $k - 1^{th}$ jump pointer would lie to the left of $o_{m'}$. On the other hand if $o_{m'}$ lies to the right of o_l, then the red jump pointer should point to $o_{m'}$ as the range o_l to $o_{m'}$ consists of one additional red than blue. As both cases are not possible, $o_{m'}$ is the same as o_m. □

Jump pointers will be used for two operations (expansion and shrink) in single-predicate range queries. An expansion operation expands a range to include more objects. Excluding objects by shrinking the range is done using shrink operations.

Definition 2. *(Cumulative sum): Consider an attribute A for the single-predicate range query model. The cumulative sum c_i at a location o_i is the difference between the number of reds and blues from the left most location along A to o_i.*

As an illustration from the sample dataset, consider computing the cumulative sum for t_0. There is one blue and three reds until t_0 from the left most position t_1. Hence, the cumulative sum for t_0 is -2 ($1 - 3 = -2$).

Given the two locations o_i and o_j ($[o_i, o_j]$), cumulative sums can be used to obtain the disparity between the start and end location in $\mathcal{O}(1)$ time.

$$disparity = c[i] - c[j - 1] \tag{2}$$

Fox example, consider the example query II-A, with range $4 \leq A_0 \leq 7$. The objects lying in the input range are $\{t_4, t_{10}, t_6, t_{12}\}$. With 3 *blues* and a *red*, the query has a disparity of 2 which can be computed using cumulative sum by $c[12] - c[0] = 2$. Note that any two locations with the same cumulative sum represent a range with *perfect parity*.

B. Query answering for unweighted fairness

A range predicate is made up of two points, *start* and *end*. If one were to fix one of the two end points of the range query, to make the range query fair, the other end point can be moved to shrink the range or to expand it. An expansion would require an addition of $|disparity - \varepsilon|$ deficient colored objects to make it fair (proved later in theorem 1). For example, consider the input range query to be $[4.4, 7]$ $(4.4 \leq A_0 \leq 7)$ with $\varepsilon = 0$. Figure 4a shows the query range $[4.4, 7]$ $(4.4 \leq A_0 \leq 7)$ marked with an enclosing box. As there are two more *blues* in the range query, we require two additional *reds* to expand the query range to make it fair. Expanding the range by following two red jump pointers gives us a fair range. Figure 4a shows a fair range when the start of the range, t_4, is fixed and the other end point is allowed to expand to incorporate two additional reds, thus obtaining the range $[4.4, 13]$ $(4.4 \leq A_0 \leq 13)$, marked by the dotted line. In general, one can follow the deficient pointer $|disparity - \varepsilon|$ times.

Our goal is to find the most similar range that resolves the disparity of $|disparity - \varepsilon|$. Such a disparity can be covered by moving either end points, that is $|disparity - \varepsilon|$ should equal to the sum of the changes on the right and left. For instance, figure 4b shows the end of the input query expanded by one red pointer and start expanded (resp. shrunk) by one red (resp. blue) pointer. Similarly, figures 4c, 4d and figure 4e shows the input query expanding/shrinking to cover the disparity of 2. We design a window-sweeping algorithm to find such a range.

Algorithm 1, *SPQA*, describes our approach, for finding the most similar fair range to the input query. Algorithm 3, JP describes the algorithm used for moving along a jump pointer.

Initially, the *start* end-point of the range is fixed and *SPQA* expands *end* end-point until a fair query is found, as shown in Figure 4a. The window is shown in the figure with the dotted lines indicating start and end of the fair range. The naming convention used for the boundaries is, S for start of input range and E for end of input range; L for left and R for right. Hence, SL (resp. SR) stands for start of input range expanded (resp. shrunk) to the left (resp. right). When the window is swept to the left, the *start* end point can perform a shrink or an expansion as shown in Figure 4b. The remaining steps of the exploration by *SPQA* can be seen in Figures 4c, 4d and 4e. Among each of these, the output of the fair range which is most similar to the input query is provided as output to the user.

Theorem 1. *Given a database $\mathbb{D}$, the disparity threshold ε, and the input query $Q(A_j : [start, end])$, the optimal range has a disparity value of exactly ε.*

Proof. Let d be the disparity of the given range query. Let the disparity of optimal range be d_{opt}. Knowing that both the left and right end points of the input range could have moved, let the range for optimal range be $[l_{opt}, r_{opt}]$. Let us consider two new ranges, $[l_{opt}, start - 1]$, $[end + 1, r_{opt}]$ and let their corresponding disparities be dl_{opt} and dr_{opt}. The total disparity d_{opt} can be written as the sum of disparity of three different ranges, dl_{opt} for range $[l_{opt}, start - 1]$, dr_{opt} for range $[end + 1, r_{opt}]$ and d for range $[start, end]$.

$$d - d_{opt} = dl_{opt} - dr_{opt}$$

Suppose $d_{opt} \neq \varepsilon$. Let us now construct a range $[l'_{opt}, r'_{opt}]$ such that the disparity is exactly ε. To construct the range we will modify l_{opt} and r_{opt}. If l_{opt} lies on the left of $start$, then there was an expansion operation that has been performed. Instead of expanding it to cover a disparity of dl_{opt}, one can only expand it to a smaller extend such that the over all disparity is exactly ε. As the expansion was smaller the *Jaccard* similarity measure would be larger. Applying a similar approach in case l_{opt} was on the right of $start$ would also result in a larger *Jaccard* similarity measure as intersection between the two sets would be larger. A similar approach can be applied to the r_{opt} end point. The newly constructed range $[l'_{opt}, r'_{opt}]$ is more similar and has disparity of exactly ϵ. □

Lemma 2. *(Correctness) Given a database $\mathbb{D}$, the disparity threshold ε, and the input query $Q(A_j : [start, end])$, SPQA Algorithm 1 finds the optimal solution (the most similar fair range to q).*

Proof. To prove this theorem we use the property that the fair optimal range has a disparity of exactly ε, which is proved in theorem 1.

SPQA uses a windowed approach to explore all ranges around the input range which have a disparity value of ε. Let the disparity covered by moving the left the left end point $disparity([left, start - 1])$ be dl and the disparity covered by the right end point, dr be equal to $disparity([end + 1, right])$. The sum of dl, dr and d is ε.

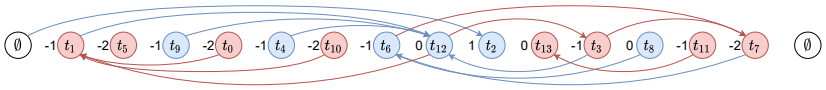
$$dl + dr = \varepsilon - d$$

SPQA explores $4 \times disparity$ number of windows, where every pair of dl, dr satisfies the above equation. Thus the optimal result must lie in one of these pairs. □

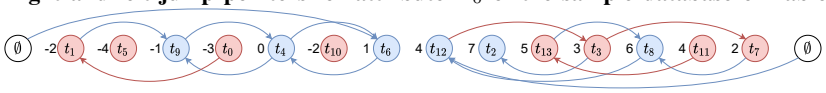
Time complexity: The total amount of time taken by *SPQA* to answer the unweighted fairness queries with one range predicate is $\mathcal{O}(\log(n) + disparity(input))$, which can be seen in theorem below.

Theorem 2. *Given a database $\mathbb{D}$ with n tuples and an input range with disparity d the total time taken by SPQA algorithm is $\mathcal{O}(\log(n) + d)$.*

Proof. Searching the jump pointer data structure to reach the end points takes $\mathcal{O}(\log(n))$ time. Once the end points are found in the data structure, *SPQA* algorithm uses a window based approach to explore the fair ranges whose disparity is exactly equal to ϵ. There are a total of $\mathcal{O}(4d)$ such ranges, each of which takes $\mathcal{O}(1)$ time to compute *Jaccard* similarity. Thus, the total amount of time taken is $\mathcal{O}(\log(n) + d)$. □

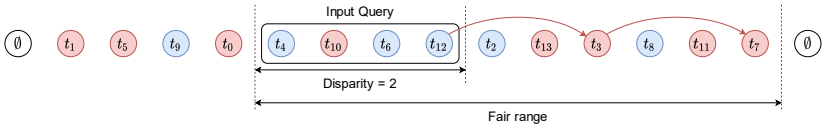


(a) Right and left jump pointers for attribute A_0 of the sample database of Table 1.

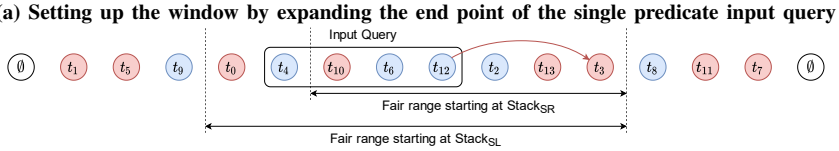


(b) Right and left jump pointers for attribute A_0 of the sample database of Table 1 *with weights*

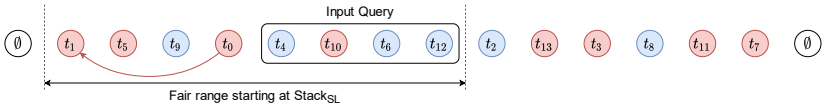
Fig. 3: Jump pointers for sample database of Table 1



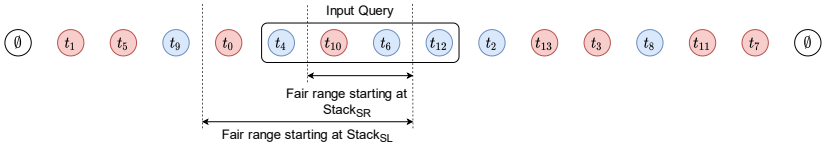
(a) Setting up the window by expanding the end point of the single predicate input query



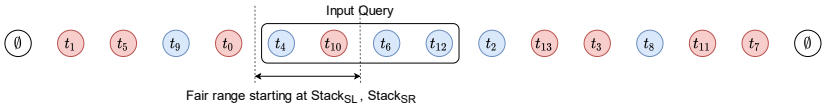
(b) Moving the window to the left from the initial position



(c) Window reaches the position where the *end* of the input lies



(d) The right end point of the window shrunk within the input query



(e) Exploration for all windows is completed

Fig. 4: Step wise movement of the window over the course a run of the single predicate algorithm

General positioning assumption: The algorithm and the definitions in the current section have been designed with the general positioning assumption. General positioning makes the assumption that no two points are co-located for the given attribute. In practice, with small modifications, our algorithms can handle the case when multiple points are present at a single location. Combining the co-located points into a single point with the aggregate weight would help us in creating a new dataset with no co-location. When the jump pointer encounters the new point with a variable weight, it needs to update the data structure with the variable weight value. Note though that the similarity function needs to take the number of points co-located into account while computing the similarity.

This algorithm holds unless there are too many of the same demographic data point at the same location so as to move a range from unfair because of lack of a group to unfair because of excess of that group. Note that this case is highly unlikely in practise. In such extreme cases, where the unfairness suddenly switches from one group being disadvantaged to the other, we choose two problems, one in which the aggregate point does not belong thus limiting one side of the search space or the second in which we explore further by looking for the inverse jump pointers past that point.

C. Prepossessing

For every attribute in $\mathbb{D}$, a jump pointers index is created during the preprocessing. For this, the objects in every list are sorted based on the corresponding attribute (Figure 3a) so that look ups can be performed quickly and then right and left pointers from each location of the database is calculated.

Finding jump pointer: *Jump* pointers play a key role in SP queries. A right *red jump* pointer points to the closest location to the right of the current location such that the number of *reds* in the range exceed the number of *blues* by 1. At any given location o_i, the range $[o_i + 1, o_i + 1]$ is a trivial range that satisfies this criteria. Hence, one of the 2 colored *jump*

Algorithm 1 *SPQA* Algorithm

Input : Database $\mathbb{D}$, Input query $Q(A_j : [start, end])$, acceptable disparity ε
Output : Most similar fair range query $fair$

$t_s \leftarrow binary_search(\mathbb{D}, A_j, start)$
$t_e \leftarrow binary_search(\mathbb{D}, A_j, end)$
$disparity \leftarrow c[j, t_s] - c[j, t_e]$
dColor $\leftarrow disparity > 0$ ▷ deficient: true-red, false-blue
if $disparity \leq \varepsilon$ **then** ▷ Input range is already fair
    return $[start, end]$
$LEP \leftarrow t_s$ ▷ LEP stands for Left End Point
while $disparity > \varepsilon$ **do**
    Push $JP(\mathbb{D}, A_j, LEP, "left", deficient)$ to LEP, update disparity for $[t_{LEP}, t_e]$
$fair \leftarrow \{\}$; $sim \leftarrow 0$
$WindowSweep(t_{LEP}, t_e,$ "shrink") ▷ Shift window by shrinking t_e
$WindowSweep(t_{LEP}, t_e,$"expand")
$WindowSweep(t_s, t_e,$"shrink") ▷ Shift win. by shrinking t_e
$WindowSweep(t_s, t_e,$ "expand")
return $fair$

Algorithm 2 $WindowSweep$ algorithm

Input : Database $\mathbb{D}$, Attribute: A_j, start end-point: t_s, end end-point: t_e, operation, input query $Q(A_j : [start, end])$, reference to $fair$, sim
Output : Update $fair$ based on most fair range found

$disparity \leftarrow c[j, t_e] - c[j, t_s - 1]$
$dColor \leftarrow disparity > 0$
if $t_s < start$ **then** $t_s \leftarrow$ Pop(LEP)
else $t_s \leftarrow JP(\mathbb{D}, A_j, t_s, "right", !dColor)$ ▷ Shrink t_s
while $disparity > \varepsilon$ **do** ▷ Adjust t_e pointer
    $disparity \leftarrow c[j, t_e] - c[j, t_s - 1]$
    $dColor \leftarrow disparity > 0$
    if $operation ==$ "$expand$" **then**
        $t_e \leftarrow JP(\mathbb{D}\ A_j, t_e, "right", dColor)$ ▷ Expand
    else $t_e \leftarrow JP(\mathbb{D}\ A_j, t_e, "left", !dColor)$ ▷ Shrink
if $Jaccard([t_s, t_e], Q) > sim$ **then**
    $sim \leftarrow Jaccard([t_s, t_e], Q)$; $fair \leftarrow [t_s, t_e]$
$WindowSweep(t_s, t_e, operation)$

pointers will point to o_{i+1}. The goal of the algorithm is to compute the other (non-trivial) right *jump* pointer.

In order to obtain the non-trivial right jump pointer, we need to find the location that differs by 1 (in the opposite sign than at location o_{i+1}). For example, in Figure 3a, the cumulative sum at location t_{12} is 0 and as the color at location t_2 is blue, the cumulative sum is 1. The non-trivial right jump pointer points the closest location with cumulative sum of -1.

The algorithm to find the jump pointers is given in Algorithm 4. The algorithm to find the jump pointers maintains a balanced binary search tree (BST) for the cumulative sums, which are used as keys for the *BST*. The indices which will be resolved when the specific cumulative sum is seen are stored as values within the *BST*. For example, when resolving the non-trivial red right jump pointer for t_{12}, closest location with a cumulative sum of -1 needs to be found. Hence, -1 is used as a key within the *BST* with the index t_{12} as a value.

The total time taken in the pre-processing step is $\mathcal{O}(n \log(n))$, as proved in theorem below.

Algorithm 3 *JP* algorithm for *left* jump pointer

Input : Database $\mathbb{D}$, Attribute: A_j, Database object o_i, color c, direction dir
Output : *Database object o_j pointed by jump pointer*

if $dir ==$ "left" **then**
    if $A_j[o_i - 1]$ is of color c **then**
        return $o_i - 1$
    else
        return $LJP[o_i]$ ▷ LJP stands for Left Jump Pointer array
else
    if $A_j[o_i + 1]$ is of color c **then**
        return $o_i + 1$
    else
        return $RJP[o_i]$ ▷ RJP stands for Left Jump Pointer array

Algorithm 4 (Preprocessing) Building left jump pointers

Input : Database $\mathbb{D}$, attribute A_j
Output : *jump pointers*

Sort $\mathbb{D}$ along attribute A_j
$BST \leftarrow \{\}$
cumulative $\leftarrow 0$
for $i \leftarrow 0$ *to* n **do**
    cumulative $\leftarrow$ cumulative + color(o_i)
    $c[j, i] \leftarrow cumulative$ ▷ Cumulative sum is contained in c
    if $cumulative$ present in BST **then**
        for $obj \in$ values of $BST[cumulative]$ **do**
            $LJP[j, obj] \leftarrow i$ ▷ Left jump pointer
    Insert i into $BST[cumulative - 2 * color(o_i)]$

Theorem 3. *Given a database $\mathbb{D}$ with n tuples and an attribute A_i pre-processing step takes $\mathcal{O}(n \log(n))$ time.*

Proof. The sorting step of pre-processing takes $\mathcal{O}(n \log(n))$ time for the given attribute A_i. Establishing the right and left *jump pointers* makes use of a balanced binary search tree (*BST*). A total of n indexes need to be inserted/deleted into the *BST*, which consumes $\mathcal{O}(n \log(n))$ time. Hence, the total time taken for building the jump pointer structure for given attribute A_i is in $\mathcal{O}(n \log(n))$. □

Note that, while the initial pre-processing takes $\mathcal{O}(n \log(n))$ time, query processing is sub-linear: $\mathcal{O}(\log(n) + d)$, where d is the disparity of input query.

Note on space complexity: During the pre-processing phase, *SPQA* algorithm creates a linear space data structure to aid in query processing. The query processing stores a total of disparity jump pointers to find the the most similar fair range which is small compared to the linear space data structure. Thus the total space complexity is $\mathcal{O}(n)$.

D. Generalization to weighted fairness

So far, our attention has been on the unweighted fairness model. In this section, we move to our general model of fairness: weighted fairness. As explained in § II, the fairness

constraint for this model is in the form of $|W_r C_r - W_b C_b| \leq \varepsilon$, where W_r and W_b are the weights for the red and blue counts, respectively. That is, the difference between the weighted sum of the number of objects from the two demographic groups should not be bounded by the threshold ε. Note that any rational values for weights can be expressed as integer weights by scaling these weights. For example, weights $W_r = 1.1$ and $W_b = 1.2$ are equivalent to $W_r = 11$ and $W_b = 12$.

The fair range query problem in the generalized case would refer to finding the most similar fair range to the input fair range such that the disparity is within a value of ε. Note that finding cases where the disparity is less than $max(W_r, W_b)/2$ would infer finding ranges with a level of precision less than a single unit of disparity (less than a single weighted colored object). For the sake of simplicity and practicality, we omit such cases and assume that the value of $\varepsilon \geq max(W_r, W_b)/2$.

The algorithm that deals with the weighted case uses similar concepts like jump pointer and cumulative sum. In the general case, a *blue* (resp. *red*) right jump pointer from location o_i points to the closest location right of o_i, j_i such that the range $[o_i+1, j_i]$ contains more *blues* than *reds* (resp. *reds* than *blues*) by weight. A similar definition for left jump pointers can be defined. Jump pointers for the sample dataset presented in Table I is presented in Figure 3b using weights of 3 for *blue* and 2 for *red*.

For the weighted case, the cumulative sum at a location o_i indicates the weighted difference of *blues* and *reds*.

Theorem 4. *(Correctness) Given a database $\mathbb{D}$, the disparity threshold ε, weights W_r and W_b, and a query $Q(A : [start, left])$, algorithm 1 finds the the most similar fair range to Q.*

Time complexity: The total time taken by *SPQA* algorithm is same as the unweighted case, $\mathcal{O}(\log(n) + d)$, where d is the disparity in the input range. The details of the time complexity for the unweighted case which is mentioned in theorem 2 also applies to the weighted case.

Note on space complexity: During the pre-processing phase, *SPQA* algorithm creates a linear space data structure to aid in query processing. The query processing stores a total of disparity jump pointers to find the the most similar fair range which is small compared to the linear space data structure. Thus the total space complexity is $\mathcal{O}(n)$.

Preprocessing for the weighted fairness model In the weighted case, a *blue* pointer points to a location that has more *blue*s than *red*s. We use the same notation as that of the unweighted case and denote the cumulative sum at location o_i for the index of attribute A_j as $c[j, i]$. If the location next to o_i had a blue then the pointer would be trivial as it is pointing to the immediate next location. Let us consider the case where the immediate next location had a *red* instead. In such a case we need to find the nearest location whose cumulative sum is larger that the $c[j, i]$. In the opposite case where the neighbour was a blue one would like to find the closest location whose cumulative sum is smaller that the $c[j, i]$. Based on this approach, two data structures can be maintained. One for the locations whose pointers can be resolved if we found a cumulative sum with a smaller value than in the data structure. And the other data structure whose pointers can be resolved if we found a cumulative sum with a larger value. We use a *balanced binary search tree* for the two data structures. The pseudo-code to find the *jump pointers* for the weighted single predicate range query is in algorithm 5. The algorithm is an extension of the unweighted *jump pointers*. The algorithm uses a sort over the database as a the first step before adding and removing n items from a balanced BST in order to find the jump pointers, it takes $\mathcal{O}(n \log n)$ time.

General positioning assumption: The algorithm and the definitions in the current section have been designed with the general positioning assumption. General positioning makes the assumption that no two points are co-located for the given attribute. In practice, with small modifications, our algorithms can handle the case when multiple points are present at a single location. Combining the co-located points into a single point with the aggregate weight would help us in creating a new dataset with no co-location. When the jump pointer encounters the new point with a variable weight, it needs to update the data structure with the variable weight value. Note though that the similarity function needs to take the number of points co-located into account while computing the similarity.

This algorithm holds until and unless we have too many of the same demographic data point at the same location so as to move a range from unfair because of lack of reds to unfair because of excess of reds. Note that this case is highly unlikely in practise. In such extreme cases, where the unfairness suddenly switches from one the advantaged group to the disadvantaged one, we choose two problems, one in which the aggregate point does not belong thus limiting one side of the search space or the second in which we try further exploration.

IV. Multi-predicate range queries

Next, we study the queries that contain multiple range predicates. Unfortunately, moving from single-predicate (SP) range queries to multi-predicate (MP) range queries complicates the problem significantly and the idea of jump pointers does not carry over. The reason is that in MP queries, there are different directions (along different angles) which a single jump can occur, while in SP there is only one direction (along x-axis) to make a jump. Moreover, while a SP query is identified by its two end-points, a MP query with d range predicates forms a hyper-cube with $2d$ sides. Hence, instead of the two end points of an SP range, one may need to move all sides of the hyper-cube to obtain the closest fair range, even when the disparity is slightly above the allowed fairness threshold, ε.

An observation that helps us with the MP cases is that the user may not be interested in fair ranges that are far away from the input query. Hence, the fair range query should be highly similar to the input range, otherwise it is not valuable for the user. We use this observation to design a *best-first search* (BFS) fair range query algorithm for the MP query.

Algorithm 5 (Preprocessing) Left jump pointers for weighted case

Input : Database $\mathbb{D}$, attribute A_j
Output : *jump pointers*

Sort $\mathbb{D}$ along attribute A_j
$larger_BST \leftarrow \{\}; smaller_BST \leftarrow \{\}$
cumulative $\leftarrow 0$
for $i \leftarrow 0$ *to* n **do**
  if color(o_i) == *'blue'* **then** ▷ *Blue* always has a positive score
    $smaller_BST[cumulative] \leftarrow i$
  else
    $larger_BST[cumulative] \leftarrow i$
  $cumulative \leftarrow cumulative$ + $weight(color(o_i))$
  $iterator \leftarrow$Find $cumulative$ in $larger_BST$
  while $iterator \neq \emptyset$ **do** ▷ Until end of $larger_BST$
    $LJP[j, iterator] \leftarrow i$ ▷ Left jump pointer
    Increment $iterator$
  $iterator \leftarrow smaller_BST.begin()$ ▷ Start of $smaller_BST$
  while $cumulative > iterator$ **do**
    $LJP[j, iterator] \leftarrow i$
    Increment $iterator$

A. Best First Search algorithm

At a high level, the BFS algorithm can be viewed as a "smart" traversal over a graph where every range is modeled as a node and there is an edge between two nodes if the outputs of their corresponding queries vary only in one tuple. That is, a node Q_2 is a *neighbor* of Q_1 if the output of query Q_1 differs from the output of query Q_2 by exactly 1 element. Mathematically, sets $out(\mathbb{D}, Q_1)$ and $out(\mathbb{D}, Q_1)$ have a symmetric difference of size 1.

The unfair input range provided by the user serves as a starting point in the graph traversal. This can be viewed as starting from the node with *Jaccard* similarity of 1 (*Jaccard distance* of 0), discovering its neighbors, deciding which node to visit next, and pruning the blanket of nodes in the graph that their corresponding ranges have similarity less than the current best fair range discovered.

Starting from the node of the input query, the algorithm first needs to discover its neighboring nodes in the graph. For this, we rely on the existence of an oracle $neighbors(Q)$ that discovers the neighbors of a query Q. It turns out, due to the frequency of calling this oracle, it can significantly impact the performance of the BFS algorithm. We shall provide a careful development of this oracle in § IV-B.

At any point of traversal, the algorithm selects the node that has the *maximum Jaccard similarity* with the *input query* for being visited next. The Jaccard similarity can be represented as the ratio of the intersection of two sets to their union, and the neighboring range to a given range can differ only by a single element. Accordingly the neighboring ranges are the ranges with the smallest Jaccard similarity.

Upon visiting a node, the algorithm checks if it satisfies the fairness requirements. If so, the algorithm stops and returns this range as the most similar fair range with query input. Otherwise, it calls the $neighbor$ oracle to discover the unseen neighbors of this node to be considered for traversal. The pseudocode of the BFS algorithm is provided in Algorithm 6. It uses a max-heap for efficient traversal of the graph. Using the heap data structure, adding the new nodes to the list of discovered nodes and identifying the most similar node to the input range is done in logarithmic time to the size of heap.

Algorithm 6 Best-First Search algorithm for MP : BFSMP

Input : Database $\mathbb{D}$, attribute list A, input query Q
Output : *most similar fair range*

$Heap \leftarrow Q$
while $|Heap| \neq 0$ **do**
  $top \leftarrow Heap.pop()$
  if $fair(top)$ **then** return top
  for $neighbor \in neighbors(top)$ **do**
    $Heap.push(neighbor)$
return $\emptyset$

Lemma 3. *(Correctness) Algorithm 6 finds the most similar fair range to the input range.*

Proof. The *Jaccard* similarity of the set being explored is I/U, and the sets being added can have a reduced *Jaccard* similarity of $(I-1)/U$ or $I/(U+1)$. These are the smallest possible decreases in *Jaccard* similarity possible by removing or adding points.

Starting from the input range, let us now consider the neighborhood path from the input range to the most similar fair range. As at every stage of the algorithm all the neighborhood ranges which account to the smallest possible decreases in *Jaccard* similarity have been added to the heap, the fair output range that the algorithm produces is the most similar one. □

B. Neighboring range computation

Having explained the BFS algorithm, we now turn our attention to developing the $neighbors$ Oracle. Computing the neighboring ranges is an important step of the BFS algorithm. The challenge here is to make sure *all* neighbors of a range have been discovered in an efficient manner.

To better explain the oracle, let us consider a sample dataset as shown in Figure 5. Consider a sample range be as shown in Figure 6. Suppose we want to expand the range outwards in order to add a new point. One simple approach of expansion that can be thought of is to move a side while maintaining either the height or the width constant. Figure 7 shows the expanded rectangle while moving the lower bound while maintaining the width constant. A similar approach can be performed on the left bound as seen in Figure 8. Note that these are not the only possible expansions. These expanded ranges will later be used to limit our search for finding the other neighboring ranges along the diagonal direction. For 2D, there are 4 such expansions. As a generalization, one can obtain $2d$ such expansions in d dimensions. Such points can be found out in $\mathcal{O}(\log^d n + k)$ using a range tree [44].

One can think of adding an additional point by moving a corner point along the diagonal direction. One such diagonal expansion can be seen in Figure 9. The bottom left corner can

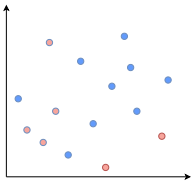


Fig. 5: Sample set of points

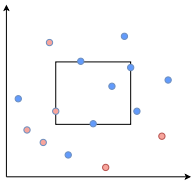


Fig. 6: Sample input range

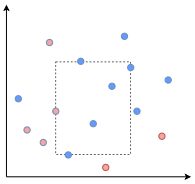


Fig. 7: Expanding the rectangle downwards

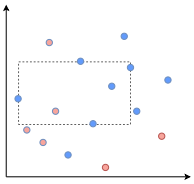


Fig. 8: Expanding the rectangle towards left

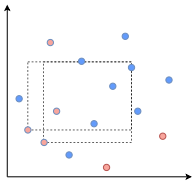


Fig. 9: Neighboring ranges in diagonal

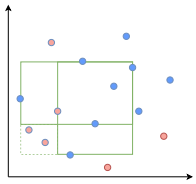


Fig. 10: Skyline computation over a range query

be expanded to add one more point along the diagonal. As one may observe, such expansions are limited by the expansion of the sides which border the corner. The expanded boundary shown in Figures 7 shows the extent to which one may expand the bottom boundary downwards until they find a point. If a point laid in the diagonal beyond such a boundary, it would not account to a neighborhood range as such an expansion would contain two points instead of one.

Figure 10 shows the expanded vertical and horizontal ranges that add a single point in solid green lines. The expanded boundaries form the limits of the region containing the diagonal expansion end points. *The problem of finding all possible diagonal expansion points can be formulated as finding a skyline* within the range shown by the dotted green lines: Given a corner point p_o for diagonal expansion[1] in a d-dimensional space, consider the bounding box specified by the side expansion (e.g. the dashed rectangle in the bottom-left of Figure 10). A point p_1 inside the bounding box dominates another point p_2 in the bounding box if $\forall 0 < i \leq d : |p_1[i] - p_o[i]| < |p_2[i] - p_o[i]|$. The skyline of the points in the bounding box is the set of points not dominated by any other point. Every skyline point is a valid diagonal expansion. As a result, in order to find all neighbors of a given range, it is enough to find (a) all neighbors by side expansion/shrinking and (b) all diagonal expansion points in the skylines. A MP query with d range predicates contains 2^d corners. Each corner can be expanded away from the center of the MP query in order to find queries that differ by a single point, i.e. a neighbouring range. There can be many neighbouring ranges for each corner. A range skyline query can be constructed for each corner using the intersection of the boundaries of the side expansion as one of the end points of the range query and the corner point's coordinates itself as the other end point. One naive approach is to obtain the points that lie within the range using a R-tree and then apply a skyline algorithm on the points obtained. This is not efficient. We use studies that efficiently compute the skyline on range queries. In particular, we use the the *Range-Skyline-Query* algorithm by Janardan et. al. [45] for skyline discovery. This algorithm has a complexity of $\mathcal{O}((k+1)\log^d n)$, where k is size of skyline. Note that k should generally be a small number. In particular, as the number of dimensions increase, and as the size of the range grows, the expected number of points that occur within the corner ranges will decrease. That is because with each dimension the number of ranges it must occur within increases by one; and it will decrease with the size of the range, as more points that are potentially the nearest point will equate to a decrease in the size of the corner.

[1]Shrinking a range is done similarly.

C. Informed best first search

The BFS algorithm discussed so far searches for the fair range by exploring the node with the maximum *Jaccard similarity* first. Branching out from a node to explore for a fair range requires discovering it neighbors, adding them to the heap, and repeating the same process for its neighbors in a recursive manner – which is time-consuming. On the other hand, given the amount of disparity at a node, it may be clear that its neighbor up to a certain number of hops cannot fill the disparity gap. That simply is because every neighboring node has a difference of exactly one element with the current node and, hence, in the best case can drop the disparity *by one unit*. In other words, if the current disparity is equal to δ and the fairness threshold is $\varepsilon < \delta$, at least $\delta - \varepsilon$ hops are needed to fill the disparity gap.

Every hop in the path from the current node reduces the similarity from the initial query to a certain degree. As a result, combining the minimum number of hops to achieve fairness with the similarity decay per hop, we can compute an upper-bound threshold on the maximum similarity for a fair range (referred as U-threshold) that one can hope to achieve by branching out from the current node.

The above observation enables to design a more efficient algorithm, *Informed Best Frist Search algorithm for Multi-Predicate* (IBFSMP), with an early stop criteria, that delays exploring the branches that their U-threshold is not the maximum. In other words, instead of selecting the most similar node to be explored next, IBFS selects the node with *maximum U-threshold* to be explored next. IBFSMP is inspired from the *A* algorithm* [46] which utilizes the lower-bound on the remaining distance to the destination to perform an efficient search. However, IBFSMP differs from the A* in details and the way the bounds are calculated. We still need to compute the U-threshold of a node, which is done in Theorem 5.

Theorem 5. *The U-threshold of a node Q is:*

$$J_U(Q) = \begin{cases} \max\limits_{C'_r \leq \lceil \frac{\delta-\varepsilon}{W_r} \rceil} \frac{I - \lceil \frac{max(\delta-\epsilon-W_r \cdot C'_r, 0)}{W_b} \rceil}{U + C'_r} & W_r > W_b;\ \delta > \varepsilon \\ \max\limits_{C'_b \leq \lceil \frac{\delta-\varepsilon}{W_b} \rceil} \frac{I - C'_b}{U + \lceil \frac{max(\delta-\epsilon-W_b \cdot C'_b, 0)}{W_r} \rceil} & W_b > W_r;\ \delta > \varepsilon \\ \max\limits_{C'_r \leq \lceil \frac{\delta-\varepsilon}{W_r} \rceil} \frac{I - C'_r}{U + \lceil \frac{max(-\delta-\epsilon-W_r \cdot C'_r, 0)}{W_b} \rceil} & W_r > W_b; \delta < -\varepsilon \\ \max\limits_{C'_b \leq \lceil \frac{\delta-\varepsilon}{W_b} \rceil} \frac{I - \lceil \frac{max(-\delta-\epsilon-W_b \cdot C'_b, 0)}{W_r} \rceil}{U + C'_b} & W_b > W_r; \delta < -\varepsilon \end{cases} \quad (3)$$

where $\delta = W_b \cdot C_b - W_r \cdot C_r$.

Proof. Let the node Q have an intersection of I and union of U with the input range. Let the disparity of the unfair range Q be $\delta = W_b \cdot C_b - W_r \cdot C_r$. As the range Q is unfair, $|\delta| > \varepsilon$.

Let us consider the case that the range Q is unfair because of the presence of too many blues in Q compared to the reds.

$$\delta = W_b \cdot C_b - W_r \cdot C_r > \varepsilon$$

As we are trying to find the upper bound, we would like to maximize the *Jaccard similarity* such that such a range can *potentially* exist. In order to obtain a fair range, either blues can be removed, reds can be added or both can be done. Let B' be the blues that are removed and R' be the reds that are added to Q to make it a fair range.

$$\begin{aligned} -\varepsilon \leq W_b(C_b - C'_b) - W_r(C_r + C'_r) &\leq \varepsilon \\ -\varepsilon \leq \delta - W_b \cdot C'_b - W_r \cdot C'_r &\leq \varepsilon \end{aligned}$$

Moving around the terms, we get

$$\delta - \varepsilon \leq W_b \cdot C'_b + W_r \cdot C'_r \leq \delta + \varepsilon \tag{4}$$

In order to maximize the *Jaccard similarity*, various values of B' and R' need to be checked which satisfy the equation 4. Note that for a given value of B'(resp. R'), using the smallest R'(resp. B') that satisfies the equation 4 would provide a larger *Jaccard similarity*. Thus, given B' the smallest value of red satisfying the equation would be,

$$C'_r = \lceil \frac{\delta - \varepsilon}{W_r} \rceil$$

The *U-threshold* thus can be expressed as a maximization in terms of R',

$$\max_{0 \leq C'_r \leq \lceil \frac{\delta-\varepsilon}{W_r} \rceil} \frac{I - \lceil max(\delta - \epsilon - W_r \cdot C'_r, 0)/W_b \rceil}{U + C'_r} \tag{5}$$

Similarly, given R' the *U-threshold* thus can be expressed as a maximization in terms of B' as,

$$\max_{0 \leq C'_b \leq \lceil \frac{\delta-\varepsilon}{W_b} \rceil} \frac{I - C'_b}{U + \lceil max(\delta - \epsilon - W_b \cdot C'_b, 0)/W_r \rceil} \tag{6}$$

The amount of time taken to compute the *U-threshold* using the equation 5 is $\lceil \frac{\delta-\varepsilon}{W_r} \rceil$. The amount of time taken to compute the *U-threshold* using the equation 6 is $\lceil \frac{\delta-\varepsilon}{W_b} \rceil$. In case W_r is larger than W_b the complexity for exploring all the values for reds using equation 5 is better. Equation 6 can be used to explore all the values for blues when W_b is larger than W_r. A similar approach can be applied when the range is unfair because of excessive reds to obtain the final two cases in equation 3.

□

Replacing the selection criteria for traversing the graph with U-threshold, the only component of Algorithm 6 that needs to change is the max-heap and the rest remains unchanged, i.e., instead of structuring the heap according to similarity, IBFS builds the heap according to U-threshold (Equation 3).

Note that the IBFS algorithm is agnostic to the heuristic and similarity measure satisfying two important properties. (1) The similarity measure being used must be a set based similarity measure based on the points in the output range. (2) As can be seen from U-threshold, the heuristic must provide a upper-bound threshold on the maximum similarity for a fair range.

Note on space complexity: BFSMP algorithm explores neighbouring ranges to reach the fair range query that is nearest to the input query. Along the process a large number of ranges are explored and stored in memory in a *heap*. The space consumed by the algorithm depends on the number of neighboring ranges explored. Thus the space complexity for BFSMP algorithm is $\mathcal{O}(\text{number of explored ranges})$.

D. Using MP algorithms for SP

Before concluding this section, we would like to note that MP algorithms also work for SP. However, *SPQA* has a provably better time complexity than *BFS*, in all instances. This is because *SPQA* takes advantage of pre-computed jump pointers which is only available when the possible changes in the bounds of the range are restricted to one degree of freedom. As explained in § III-B, *SPQA* has a worst-case time complexity of $\mathcal{O}(\log(n)+disparity)$. One can easily establish a *best* case complexity for *BFS* algorithms that is at least as slow as this. First, in order to reach it's destination, *BFS* can adjust its range with each step by adding or removing a point. In the best case, it visits only points which monotonically decrease the disparity, and stops after *disparity* more steps. Additionally, *BFS* constructs a one dimensional range tree (which is equivalent to a balanced binary search tree) as pre-processing to find the closest point. This requires an initial setup time of $n \log(n)$. Therefore, the best case time-complexity of *BFS*s is $\Omega(n \log(n) + disparity)$. This demonstrates that the time-complexity of *SPQA* is comprehensively better, and accordingly, we favor it for SP queries.

V. Experiments

A. Experimental setup

Datasets: We used both real and synthetic datasets for our experiments. For the real world datasets we use *TexasTribune* and *UrbanGB* datasets. Along with the real world datasets, a synthetic dataset *Uniform* was generated for the experiments. Below we provide a brief description of these datasets.

Dataset name	Items	d	Sens. attribute	Weights
Texas Tribune [47]	149,481	21	gender,race	gen. (1:1) race (4:5)
COMPASS [48]	60,842	12	race	2:1
UrbanGB [49]	1,600,000	33	#vehicles in accident	2:1
(Synthetic)*Uniform*	10,000	4	Synthetic	1:1

- (Real dataset) *TexasTribune*[2]: Texas Tribune dataset consists of 149,481 records with the salary/compensation information for Texas state employees. The dataset has 21 attributes with gender and race being the main sensitive attributes and salary/compensation being numeric.

[2]https://salaries.texastribune.org/

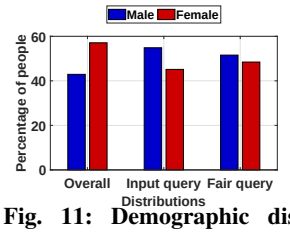


Fig. 11: Demographic distributions in dataset, input query, and similar fair query.

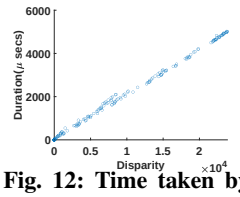


Fig. 12: Time taken by *SPQA* v.s. disparity - *Texas Tribune* with gender as SA

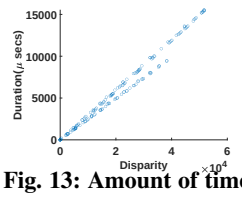


Fig. 13: Amount of time taken by *SPQA* against disparity - *Texas Tribune* with race as SA

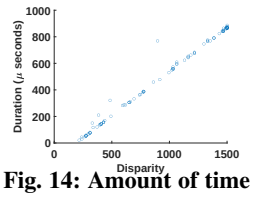


Fig. 14: Amount of time taken by *SPQA* against disparity - COMPASS dataset

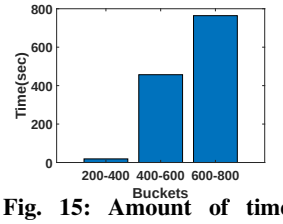


Fig. 15: Amount of time taken by IBFSMP - Uniform dataset 3 range predicates

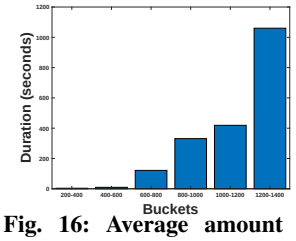


Fig. 16: Average amount of time taken by IBFSMP algorithm for different bucket sizes - *UrbanGB* dataset

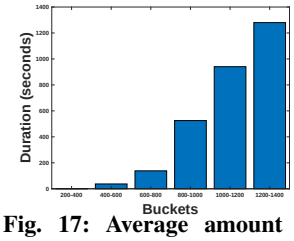


Fig. 17: Average amount of time taken by IBFSMP algorithm for different bucket sizes - *Uniform* dataset

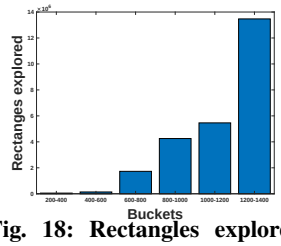


Fig. 18: Rectangles explored by IBFSMP algorithm for different bucket sizes - *Urban GB* dataset

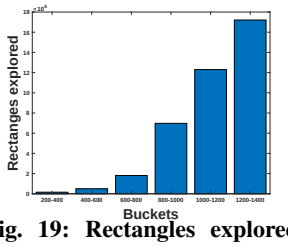


Fig. 19: Rectangles explored by IBFSMP algorithm for different bucket sizes - *Uniform* dataset

- (Real dataset) *COMPASS [48]*: COMPASS dataset (unprocessed) consists of 60,842 data points collected with 12 attributes with race as sensitive attribute and one real numbered attribute(*raw score*). The dataset has around 21K Caucasian (blue) and 39K non-Caucasian (red) warranting a ratio of 2:1.
- (Real dataset) *UrbanGB*[3]: UrbanGB dataset consists of 1.6 million records of accidents over a period 2000 and 2016. The dataset has 33 attributes, including latitude, longitude, accident severity, number of vehicles involved in the accident, date and time of accident. For the experiments, $10,000$ records from the *UrbanGB* dataset have been used. As *UrbanGB* dataset does not have a sensitive attribute inherent to it, we use the number of vehicles that were involved in the accident to create a sensitive attribute. There are 3,088 records where a single vehicle was involved in an accident and 6,912 records where more than one vehicle was involved. Hence, we use a weight of 2 for the 3,088 records and 1 for the 6,912 records.
- (Synthetic dataset) *Uniform*: The dataset consists of 10,000 points sampled uniformly from a cube which has a side of length 1,000 and a uniformly sampled binary sensitive attribute. The dataset has 4,967 blues and 5,033 reds.

The *Texas Tribune* and *COMPASS* datasets consist of one numerical attribute, a few other categorical attributes and sensitive attribute. Hence, the two datasets have been used with *SPQA*. As *Uniform* and *UrbanGB* datasets consist of multiple numeric attributes it is used for MPQA queries.

Our experiments were conducted on a Intel(R) Core(TM) i7-6850K CPU @ 3.60GHz with 64GB of main memory using Linux operating system (Ubuntu 18.04.5 LTS).

Algorithms implemented: Along with *SPQA*, weighted *SPQA* and IBFSMP algorithms, to evaluate multi-predicate range queries, we implemented a local search baseline algorithm.

- *Baseline*: The baseline approach for the multi-predicate range queries is based on limiting the search space to obtain a bounding box within which to search for fair queries. The maximum number of elements that can be added to the input range without violating the *Jaccard similarity* criteria is first computed. The boundaries of the expanded box are found by finding the smallest expansion along each of the directions/dimensions without violating the similarity criteria. This expansion limits the search space while still providing a valid box to search for the most similar fair range. This expanded range is then searched in a brute-force manner to obtain the closest fair range.
- *Coverage based algorithm* [35], [36]: We have used the code from [35], [36] to compare against our methods for multi-predicate range queries. The different techniques in the papers [35], [36] modify the input range to produce a range which covers at least a given threshold number from each demographic group.

All the algorithms in the paper are implemented using C++. The implemented code can be found at the github location[4].

Experimental parameters: The value for ε plays an important role in simulating a real scenario. Satisfying perfect parity may not always be possible in practice and may require significant changes in the initial setting. In particular, in our problem setting, the ranges that are not similar enough to the user input may not be valuable. In our experiments, we allow a disparity of 5% between demographic groups. The corresponding value of ε is then computed as:

$$\varepsilon = \frac{0.05\ (|B\ W_b| + |R\ W_r|)|out(Q)|}{2(B+R)}$$

where B and R are the total number of blues and reds in the universe respectively. W_b (resp. W_r) refer to the weight of each blue (resp. red). The entity $(|B\ W_b| + |R\ W_r|)/(B+R)$

[3]kaggle.com/daveianhickey/2000-16-traffic-flow-england-scotland-wales/data

[4]https://github.com/surajshetiya/fairness-range-queries-icde-2022

gives us the expected magnitude of the weight each point carries. The scaled weight for the given query Q with an allowed disparity of 5% thus turns out to be $0.05(|B\ W_b| + |R\ W_r|)|out(Q)|/(B+R)$.

Choosing an appropriate value of ε: Our system enables responsible data selection through exploration. While the choice of ε varies based on the application, an exploration based approach helps the application owner to choose an appropriate value of ε. That is, after the user specifies a range query and a value of ε, we return the most similar query, satisfying it. The user then has the choice to accept our recommendation, or to adjust the value of ε and/or the range and continue exploring until a satisfactory range is identified.

B. Proof of Concept - TEXAS TRIBUNE

For a proof of concept, we use the Texas Tribune dataset. The dataset consists of 149,481 records of compensation for Texas state employees. We use `gender` as the sensitive attribute. As the representation of males and females in the dataset is similar (64,153 men and 85,328 female), the weights for male and female in the query are considered to be the same. The distribution of male and female employees in the overall dataset can be seen in Figure 11. Following Example 1 on this dataset, we assume the business office is interested in finding the employees who earn a salary of more than $65,000. There are a total of 14,803 men and 12,182 women earning more than $65,000, with a difference of 2,621 (around 10%). Considering the ethical considerations, the business management office would like to find a query whose output is similar to the initial query and within which there are at most 1000 (around 5%) males more than females.

Using *SPQA*, a fair query which is most similar to the input input query is determined. The output of the fair query ($60562 \leq salary \leq 152000$) consists of 32,064 employees with 16,532 male and 15,532 female. The *Jaccard* similarity of the fair query with the input query is around 75% (76.23%). Figure 11 shows the distribution of male and female employees in input query and the most similar fair query.

Extended PoC: A function $F(x)$ which for which datapoints with the same x values but from different demographic groups have different results will underperform on a given demographic group if that demographic group is unfairly represented in designing the function. In the TexasTribune dataset, there is a function that meets this requirement where $F(x)$ predicts the salary and the parameters x are their level (or lack of level) of employment and the part of the state for which of they are employed. We trained an auto-sklearn regressor on this task, using two datasets: the result of the original query in Example 1, and the result of the modified fair query. We then analyzed the dataset using R^2 scores over male and female data points. We observed that with the initial dataset there was a fractional difference of 0.088, while with the unbiased data, that fractional difference was reduced to 0.022, where a lower fractional difference represents a regressor predicting the salary for men and women more consistently.

Systems Integration PoC: As an integration with a DBMS system, we create a thin web based interface based on *postgres*. For this PoC, we use the *TexasTribune* dataset. The *postgres* database is created with index on the numeric attribute - salary. For the PoC, we compare the $\mathcal{O}(n^2)$ naive algorithm with the jump pointer algorithm.

For the naive algorithm, the time is computed for processing the query. For the jump pointer algorithm, there is a preprocessing phase where we calculate the cumulative sum and jump pointers and populate a separate table with these details. We measure the time taken for naive query, pre-processing to create jump pointers and *SPQA* query processing. Average time taken by the pre-processing algorithm taken by *SPQA* 0.043 second. We run around 500 randomly sampled queries and measured the time taken by each of these queries. While average time taken by *SPQA* algorithm is 0.0054 seconds, average time taken by naive algorithm is 6.938 seconds. This shows the efficienly of *SPQA* when integrated with DBMS.

C. Performance of SPQA and weighted SPQA

The performance of *SPQA* depends on the disparity of the input query. For these series of experiments, we measure the amount of time taken by the *SPQA* algorithm when provided with an input query. The experiment were averaged over five runs of *SPQA* algorithm for more reliable time measurements. For these experiments, the*TexasTribune* dataset was used. As *TexasTribune* has 149,481 records with almost the same number of `male` as `female` we use `gender` as the sensitive attribute for the unweighted case. For the weighted case, we use `race` as the sensitive attribute while using `white` (majority) and `non-white` (minority) as the demographic groups. There are a total of $67,142$ `white` records and $82,339$ `non-white` records. As the ratio of `white` to `non-white` is very near to $4:5$ (0.815), we use weights of 4 and 5 for the weighted *SPQA*. For the SP queries, we use salary as the attribute for range predicates. A large part of the records of the database (95.6%) have a salary less than $100,000$. Hence, to set the range query boundary, we pick all points in multiples of $5,000$ between $5,000$ and $100,000$ as start and end points. For every query, time taken by *SPQA* algorithm is measured along with the input query's disparity. For both the weighted and unweighted case, the ε value was set to 500 for this set of experiments.

For the COMPASS dataset, the risk score varied between $-4.79 - 51.0$. Starting of the input range was generated between -4.79 and 51.0 with multiples of 3.0. Ending of the input range varied from the starting in multiples of 3.0. The *COMPASS* dataset has around 21K Caucasian (blue) and 39K non-Caucasian (red) warranting a ratio of 2:1.

Figures 12 and 13 show the scatter plot for amount of time for *SPQA* against the input query's disparity value for the *Texas Tribune* dataset and figure 14 shows the scatter plot for the weighted *SPQA* queries run on the *COMPASS* dataset. As a baseline, we ran IBFS for single range predicate (weighted and unweighted). On average, IBFS ran about 3 orders of magnitude slower than the jump pointer algorithm. The plots show a linear scaling of time with the input query's disparity. This empirically validates the running time of both the unweighted and weighted *SPQA* algorithms.

D. Performance evaluation of MP algorithms

For the multi-predicate range query evaluation, we use *UrbanGB* and *Uniform* datasets. For the experiments, $10,000$ records from the *UrbanGB* dataset have been used. There a total of 3,088 *blues* and 6,912 *reds*. Hence, we use a weight of 2 for the *blue* records and 1 for the *red* records. Latitude and longitude attributes from the database were used to form the range queries. The latitude values in the 10K records varied from -0.507015 to 0.297345 and the longitude values varied from 51.306584 to 51.660974. *Uniform* dataset consists of 10K records uniformly sampled from within a square of length $1,000$. The sensitive attribute is made of an almost equal number of blues and reds and hence we use a weight of 1 for both these colors. For all sets of experiments, a disparity of 5% between demographic groups is allowed which is indicated by the value of ε used. The baseline algorithm restricts the search to a bounding box and performs a thorough search of all the ranges in this rectangle. In this section we compare the run times of IBFSMP and baseline algorithms.

1) Effect of input query size on the run time: Query size is an important measure as it impacts the performance of our algorithms. It impacts the number of points that are being added or removed to find a fair range. While there are many other factors which may impact the performance of the query, we choose many queries in each bucket and repeat our experiments with each of these and aggregate our results to reduce the impact of other factors. For our experiments, query sizes vary from 200 to 1400 that are bucketized with intervals of 200. That is, if for example a query result contains 558 points, it fall in the query size bucket of 400-600. Each bucket has 20 range queries sampled for the experiment using rejection sampling. The input queries are chosen from different buckets using rejection sampling based on the points which satisfy the query. In each bucket, we execute 30 queries each and aggregate the results for comparison. The average run-time is measured for both the algorithms under different bucket sizes. For the queries in each bucket, the mean time taken during the run of the IBFSMP algorithm for *UrbanGB* and *Uniform* datasets is shown in Figure 16 and Figure 17, respectively. In case of the baseline algorithm which restricts the search space, experimental results for the bucket 200-400 show a mean of 697.4 seconds and 557.1 seconds for the *UrbanGB* and *Uniform* datasets respectively. The cases for the larger bucket sizes *did not complete even after* 3 *hours* and thus are not tabulated. The aggregated values of mean show that IBFSMP outperforms the baseline algorithm by orders of magnitude. For each individual query, the IBFSMP outperforms the baseline algorithm. But, due to space constraints, the details of each query executed is not included.

IBFSMP shows similar trend when run in higher dimensions. The experiments with 3 range predicates show that the time taken grows with input range size as seen in figure 15. One difference we observed was that the larger part of the computation was spent in computing skylines than in lower dimensions. One can observe the increase in run times between the two and dimension charts even for small input sizes.

2) Effect of input query size on the number of ranges explored: For the next set of experiments, we evaluate the effect of input query size on the ranges explored. As the algorithm has a dependence on many factors, we choose input query arbitrarily to analyse the impact of input query size, *Jaccard similarity* from the input range on running time. The number of ranges explored by the IBFSMP algorithm is measured as a parameter along with the time taken. We use the same set of queries with different sizes, bucketized with intervals of 200, as in our previous experiment.

The number of ranges explored by *IBFSMP* for *UrbanGB* and *Uniform* datasets are shown in Figure 18 and Figure 19 respectively. As can be seen in the figures, the number of ranges explored grows significantly with increase in query size. As the amount of time taken is proportional to the number of ranges, the mean time taken grows with the number of ranges explored as can be seen in the both the figures.

We did not include the performance of *BFSMP* in the table as *IBFSMP* significantly outperformed it in all cases. For example, while *IBFSMP* on average required only 1.1 seconds for the 200-400 bucket in *Uniform* dataset, *BFSMP* on average took 11.1 seconds. That is because, on average, it explored 145K ranges (SD=394K) while this number was 15K for *IBFSMP*. Similarly, for *UrbanGB* dataset, *BFSMP* on average took 15.3 seconds while *IBFSMP* took 3.6 seconds for this experiment. The reason was that *BFSMP* on average explored 199K ranges , while this number was 51K for *IBFSMP*. In all cases, *IBFSMP* outperformed *BFSMP* for every query.

E. Comparison with coverage based algorithms

Coverage based algorithms [36], [35] output a range query by modifying the given query such that at least a given number of items from each sensitive group are present. Note that coverage based *CRBase* makes use of a threshold value for each demographic group. On the other hand demographic parity measure is based on the notion of weighted difference between the demographic groups. As a range expands by addition of the minority group, items from the majority group are also added which may increase the disparity. To find ranges which satisfy demographic parity measure, we make use of numerous values of threshold to find different ranges that satisfy demographic parity fairness measure. Among these fair ranges, we record the ones which have the most similarity.

We have run these experiments with the *uniform* and *Urban GB* datasets. We measure the fair ranges from *CRBase* algorithm and record the one which has the most similarity. *CRBase* algorithm was run with 4, 8, 16 and 32 bins. *CRBase* algorithm produces a fair range 33.9% of the time with the *Uniform* dataset. We measure the error by computing $1-CRSim/Optimal$, where $CRSim$ is the similarity of the $CRBase$ algorithm where as the $Optimal$ is the similarity of the optimal range. For the ranges where *CRBase* algorithm does not satisfy the fairness or similarity criteria we mark $CRSim$ as 0. An average error measure of 0 means that optimal range is always obtained, while an error of 1 means that the range produced never satisfies the criteria. The error

produced by *CRBase* is 0.682 on average. For the *UrbanGB* dataset, we used a weighted fairness measure. $CRBase$ was able to produce a fair range for only 3 out of 120 sample ranges. The experiment shows that the two optimization problems and hence, the solutions are different in nature.

F. Summary of experimental results

At a high level, the experiments verify the efficiency and efficacy of our methods. Firstly, we empirically show the efficiency of the unweighted and weighted *SPQA* algorithm. Secondly, for a wide spectrum of range queries, we show that BFS algorithms outperform the baseline algorithm by orders of magnitude. Moreover, IBFSMP outperformed BFSMP since it explored far less number of ranges before it found the optimal solution. Finally, we also show the effect of input range size on IBFSMP, the larger the set size the more the time taken by IBFSMP to find the most similar fair range.

VI. Related work

Query answering: Efficiency is critical requirement in query answering. A large amount of research has focused on different aspects of query answering over the past few decades. One of the popular methods that has been explored is the query answering using views [50], [51], [52], [53], where the goal is to efficiently answer a query using a set of previously materialized views on the database. Srivastava et. al. [52] answer SQL queries with grouping and aggregation in the presence of multiset tables by detecting when the information existing in a view is sufficient to answer a query. Chaudhuri et. al. [53] solve the problem of optimizing queries in the presence of materialized views. Approximately answering queries has also been studied extensively in many works [54], [55], [56], [57], [58]. While there have been many works in the area of query answering, none of these works can be modified to incorporate fairness into them.

Fairness: Reducing racial disparities has recently been a key research [3], [59], [60], [61], [62], [63], [11], [25]. Feldman et. al. [59] propose methods to make make data unbiased by modifying the fields/attributes. Hajian et. al. [60] propose a data transformation that can consider combination of attributes to perform data transformation. While [59], [60] perform data modification, we do not modify any data point to remove bias from data instead we provide the *nearest* fair data points to work with. While [61], [62] propose methods that learn to produce fair machine learning models from the given data they do not eliminate bias from the data itself.

Query reformulation: Salimi et. al. [64] created a system for detecting statistical dependencies which impact the result of the original query. In their work, they reformulate queries by modifying the attributes queried to account for these statistical anomalies. In other works [34], [35], [36], a system has been proposed which minimally relaxes a query to provide coverage for sensitive groups. The objective of [35], [36] is to modify the original query satisfying demographic coverage constraints (minimum number of items from a each group). Coverage constraint satisfaction involves only relaxing the constraints, which may not help in reducing disparity. Note that, trying to satisfy coverage can further *increase* the disparity between the groups. Similar to these works [34], [35], [36], our algorithms also modify the original query. However, unlike existing work, our objective is to find queries (i) similar to the initial query that (ii) satisfy a disparity (unfairness) threshold on counts from different demographic groups.

VII. Discussion and future work

Fairness model: There are many fairness models which one can consider when the data contains demographic sensitive attributes. In this paper, we have used the fairness model in which objects from different demographic groups have equal chances of being selected in the output set. There are other fairness models like the demographic parity based on ratio which we consider for future work. Such a fairness model has the form, $\delta \geq C_r/C_b \geq \delta^{-1}$.

Operators: In our current work, we have considered a conjunctive operator to join different predicates. Query models like *SQL* support operators like *NOT* and *OR*. Note that the subset of operations (*OR* and *AND*) would allow the output queries to allow for union of ranges. We consider the addition of these different operators to the query model as an extension of the paper for future work.

All nearest fair ranges: The declarative query in 2 can have multiple range queries which are *equally* near while satisfying fairness constraints. An interesting area of research would be to enumerate all these nearest fair ranges.

Demographic group based extensions: Fairness problems based on binary demographic groups have been well studied [65], [66], [67], [68] for various applications like clustering, PCA and other optimization problems. We note that a significant portion of existing literature fairness and its definitions consider binary cases, as there usually is an advantaged/majority v.s. disadvantaged/minority group(e.g. COMPAS dataset(black vs non-black), adult and salary dataset(female vs male)). While binary case for fairness is an important case, extensions to these problems are valuable in many scenarios. We consider extending the fair range queries to non-binary demographic groups and demographic parity constraints on multiple sensitive attributes as future work.

VIII. Final remarks

In this paper, we initiated research on integrating fairness into data management systems. As our first attempt, we focused on selection bias in range queries, and proposed efficient algorithms. In particular, we proposed a sub-linear algorithm for single-predicate range queries and two algorithms based modeling the problem as graph traversal for multi-predicate range queries. Besides theoretical analysis, comprehensive experiments verified efficiency and effectiveness of our proposal.

We consider the extensive research required for the full integration of fairness, including a comprehensive database and query model with a broad coverage of bias, fairness notions, and a broad range of SQL operators as well as designing more efficient algorithms, for our future work.

REFERENCES

[1] Soumya Sen. How data, analytics, and technology are helping us fight covid-19. minnpost, 2020.

[2] Solon Barocas and Andrew D Selbst. Big data's disparate impact. *Calif. L. Rev.*, 104:671, 2016.

[3] Abolfazl Asudeh and HV Jagadish. Fairly evaluating and scoring items in a data set. *Proceedings of the VLDB Endowment*, 13(12):3445–3448, 2020.

[4] Cathy O'neil. *Weapons of math destruction: How big data increases inequality and threatens democracy*. Broadway Books, 2016.

[5] Alexandra Chouldechova. Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *Big data*, 5(2):153–163, 2017.

[6] Jon Kleinberg, Sendhil Mullainathan, and Manish Raghavan. Inherent trade-offs in the fair determination of risk scores. *arXiv preprint arXiv:1609.05807*, 2016.

[7] Sorelle A Friedler, Carlos Scheidegger, and Suresh Venkatasubramanian. On the (im) possibility of fairness. *arXiv preprint arXiv:1609.07236*, 2016.

[8] Francine D Blau and Lawrence M Kahn. The gender pay gap. *The Economists' Voice*, 4(4), 2007.

[9] Babak Salimi, Luke Rodriguez, Bill Howe, and Dan Suciu. Interventional fairness: Causal database repair for algorithmic fairness. In *SIGMOD*, pages 793–810, 2019.

[10] Babak Salimi, Bill Howe, and Dan Suciu. Database repair meets algorithmic fairness. *ACM SIGMOD Record*, 49(1):34–41, 2020.

[11] Fatemeh Nargesian, Abolfazl Asudeh, and H. V. Jagadish. Tailoring data source distributions for fairness-aware data integration. *PVLDB*, 14(11):2519–2532, 2021.

[12] An Yan and Bill Howe. Equitensors: Learning fair integrations of heterogeneous urban data. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2338–2347, 2021.

[13] Zhongjun Jin, Mengjing Xu, Chenkai Sun, Abolfazl Asudeh, and HV Jagadish. Mithracoverage: A system for investigating population bias for intersectional fairness. In *SIGMOD*, 2020.

[14] Abolfazl Asudeh, Zhongjun Jin, and HV Jagadish. Assessing and remedying coverage for a given dataset. In *ICDE*, pages 554–565. IEEE, 2019.

[15] Yin Lin, Yifan Guan, Abolfazl Asudeh, and HV Jagadish. Identifying insufficient data coverage in databases with multiple relations. *PVLDB*, 13(12):2229–2242, 2020.

[16] Abolfazl Asudeh, Nima Shahbazi, Zhongjun Jin, and HV Jagadish. Identifying insufficient data coverage for ordinal continuous-valued attributes. In *Proceedings of the 2021 International Conference on Management of Data*, pages 129–141, 2021.

[17] Ki Hyun Tae and Steven Euijong Whang. Slice tuner: A selective data acquisition framework for accurate and fair machine learning models. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1771–1783, 2021.

[18] Eliana Pastor, Luca de Alfaro, and Elena Baralis. Looking for trouble: Analyzing classifier behavior via pattern divergence. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1400–1412, 2021.

[19] Yuval Moskovitch and HV Jagadish. Countata: dataset labeling using pattern counts. *Proceedings of the VLDB Endowment*, 13(12):2829–2832, 2020.

[20] Chenkai Sun, Abolfazl Asudeh, HV Jagadish, Bill Howe, and Julia Stoyanovich. Mithralabel: Flexible dataset nutritional labels for responsible data science. In *CIKM*, pages 2893–2896, 2019.

[21] Ke Yang, Julia Stoyanovich, Abolfazl Asudeh, Bill Howe, HV Jagadish, and Gerome Miklau. A nutritional label for rankings. In *SIGMOD*, pages 1773–1776, 2018.

[22] Hantian Zhang, Xu Chu, Abolfazl Asudeh, and Shamkant B Navathe. Omnifair: A declarative system for model-agnostic group fairness in machine learning. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2076–2088, 2021.

[23] Hantian Zhang, Nima Shahbazi, Xu Chu, and Abolfazl Asudeh. Fairrover: explorative model building for fair and responsible machine learning. In *Proceedings of the Fifth Workshop on Data Management for End-To-End Machine Learning*, pages 1–10, 2021.

[24] Agathe Balayn, Christoph Lofi, and Geert-Jan Houben. Managing bias and unfairness in data for decision support: a survey of machine learning and data engineering approaches to identify and mitigate bias and unfairness within data management and analytics systems. *The VLDB Journal*, pages 1–30, 2021.

[25] Abolfazl Asudeh, HV Jagadish, Julia Stoyanovich, and Gautam Das. Designing fair ranking schemes. In *SIGMOD*, pages 1259–1276, 2019.

[26] Caitlin Kuhlman and Elke Rundensteiner. Rank aggregation algorithms for fair consensus. *PVLDB*, 13(12):2706–2719, 2020.

[27] Abolfazl Asudeh, HV Jagadish, Gerome Miklau, and Julia Stoyanovich. On obtaining stable rankings. *PVLDB*, 12(3), 2019.

[28] Yifan Guan, Abolfazl Asudeh, Pranav Mayuram, HV Jagadish, Julia Stoyanovich, Gerome Miklau, and Gautam Das. Mithraranking: A system for responsible ranking design. In *SIGMOD*, pages 1913–1916, 2019.

[29] Yan Zhao, Kai Zheng, Jiannan Guo, Bin Yang, Torben Bach Pedersen, and Christian S Jensen. Fairness-aware task assignment in spatial crowdsourcing: Game-theoretic approaches. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 265–276. IEEE, 2021.

[30] Lise Getoor. Responsible data science. In *SIGMOD*, 2019.

[31] Julia Stoyanovich, Bill Howe, and HV Jagadish. Responsible data management. *PVLDB*, 13(12):3474–3488, 2020.

[32] Nihar B Shah and Zachary Lipton. Sigmod 2020 tutorial on fairness and bias in peer review and other sociotechnical intelligent systems. In *SIGMOD*, pages 2637–2640, 2020.

[33] Suresh Venkatasubramanian. Algorithmic fairness: Measures, methods and representations. In *PODS*, pages 481–481, 2019.

[34] Chiara Accinelli, Barbara Catania, Giovanna Guerrini, and Simone Minisi. covrew: a python toolkit for pre-processing pipeline rewriting ensuring coverage constraint satisfaction demonstration paper. 2021.

[35] Chiara Accinelli, Barbara Catania, Giovanna Guerrini, and Simone Minisi. The impact of rewriting on coverage constraint satisfaction. In *EDBT/ICDT Workshops*, 2021.

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

[37] Michael J Zimmer. Slicing & dicing of individual disparate treatment law. *La. L. Rev.*, 61:577, 2000.

[38] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rodriguez, and Krishna P Gummadi. Fairness beyond disparate treatment & disparate impact: Learning classification without disparate mistreatment. In *Proceedings of the 26th international conference on world wide web*, pages 1171–1180, 2017.

[39] Abolfazl Asudeh. Enabling responsible data science in practice. *ACM SIGMOD Blog*, January 2021.

[40] Arvind Narayanan. Translation tutorial: 21 fairness definitions and their politics. In *FAT**, 2018.

[41] Solon Barocas, Moritz Hardt, and Arvind Narayanan. Fairness and machine learning: Limitations and opportunities. fairmlbook.org, 2019.

[42] Indrė Žliobaitė. Measuring discrimination in algorithmic decision making. *DATA MIN KNOWL DISC*, 31(4):1060–1089, 2017.

[43] Faisal Kamiran and Toon Calders. Data preprocessing techniques for classification without discrimination. *Knowledge and Information Systems*, 33(1):1–33, 2012.

[44] Jon Louis Bentley. Decomposable searching problems. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1978.

[45] Saladi Rahul and Ravi Janardan. Algorithms for range-skyline queries. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, pages 526–529, 2012.

[46] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[47] Texas tribune dataset. https://salaries.texastribune.org/, visited: 2021.

[48] John Monahan and Jennifer L Skeem. Risk assessment in criminal sentencing. *Annual review of clinical psychology*, 12:489–513, 2016.

[49] Urbangb dataset. kaggle.com/daveianhickey/2000-16-traffic-flow-england-scotland-wales/data, visited: 2021.

[50] Alon Y Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[51] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. Answering top-k queries using views. In *Proceedings of the 32nd international conference on Very large data bases*, pages 451–462, 2006.

[52] Divesh Srivastava, Shaul Dar, Hosagrahar V Jagadish, and Alon Y Levy. Answering queries with aggregation using views. In *VLDB*, volume 96, pages 318–329, 1996.

[53] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 190–200. IEEE, 1995.

[54] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS)*, 32(2):9–es, 2007.

[55] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 574–576, 1999.

[56] Saravanan Thirumuruganathan, Shohedul Hasan, Nick Koudas, and Gautam Das. Approximate query processing for data exploration using deep generative models. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1309–1320. IEEE, 2020.

[57] Qingzhi Ma and Peter Triantafillou. Dbest: Revisiting approximate query processing engines with machine learning models. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1553–1570, 2019.

[58] Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. Astrid: Accurate selectivity estimation for string predicates using deep learning.

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

[60] Sara Hajian, Josep Domingo-Ferrer, and Antoni Martinez-Balleste. Discrimination prevention in data mining for intrusion and crime detection. In *2011 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*, pages 47–54. IEEE, 2011.

[61] Moritz Hardt, Eric Price, and Nati Srebro. Equality of opportunity in supervised learning. In *Advances in neural information processing systems*, pages 3315–3323, 2016.

[62] Shahin Jabbari, Matthew Joseph, Michael Kearns, Jamie Morgenstern, and Aaron Roth. Fair learning in markovian environments. *arXiv preprint arXiv:1611.03071*, 2016.

[63] Matthew Joseph, Michael Kearns, Jamie H Morgenstern, and Aaron Roth. Fairness in learning: Classic and contextual bandits. In *Advances in Neural Information Processing Systems*, pages 325–333, 2016.

[64] Babak Salimi, Corey Cole, Peter Li, Johannes Gehrke, and Dan Suciu. Hypdb: a demonstration of detecting, explaining and resolving bias in olap queries. *Proceedings of the VLDB Endowment*, 11(12):2062–2065, 2018.

[65] Julia Dressel and Hany Farid. The accuracy, fairness, and limits of predicting recidivism. *Science advances*, 4(1):eaao5580, 2018.

[66] Vladimiro Zelaya, Paolo Missier, and Dennis Prangle. Parametrised data sampling for fairness optimisation. *KDD XAI*, 2019.

[67] Arturs Backurs, Piotr Indyk, Krzysztof Onak, Baruch Schieber, Ali Vakilian, and Tal Wagner. Scalable fair clustering. In *International Conference on Machine Learning*, pages 405–413. PMLR, 2019.

[68] Matt Olfat and Anil Aswani. Convex formulations for fair principal component analysis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 663–670, 2019.