FairHash: A Fair and Memory/Time-efficient Hashmap

Anonymous Author(s)*

ABSTRACT

Hashmap is a fundamental data structure in computer science. There has been extensive research on constructing hashmaps that minimize the number of collisions leading to efficient lookup query time. Recently, the data-dependant approaches, construct hashmaps tailored for a target data distribution that guarantee to uniformly distribute data across different buckets and hence minimize the collisions. Still, to the best of our knowledge, none of the existing technique guarantees group fairness among different groups of items stored in the hashmap.

Therefore, in this paper, we introduce FairHash, a data-dependant hashmap that guarantees uniform distribution at the group-level across hash buckets, and hence, satisfies the statistical parity notion of group fairness. We formally define, three notions of fairness and, unlike existing work, FairHash satisfies all three of them simultaneously. We propose three families of algorithms to design fair hashmaps, suitable for different settings. Our ranking-based algorithms reduce the unfairness of data-dependant hashmaps without any memory-overhead. The cut-based algorithms guarantee zero-unfairness in all cases, irrespective of how the data is distributed, but those introduce an extra memory-overhead. Last but not least, the discrepancy-based algorithms enable trading off between various fairness notions. In addition to the theoretical analysis, we perform extensive experiments to evaluate the efficiency and efficacy of our algorithms on real datasets. Our results verify the superiority of FairHash compared to the other baselines on fairness at almost no performance cost.

ACM Reference Format:
Anonymous Author(s). 2018. FairHash: A Fair and Memory/Time-efficient Hashmap. In *Proceedings of (Conference acronym 'XX).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

1.1 Motivation

As data-driven technologies become ingrained in our lives, their drawbacks and potential harms become increasingly evident [13, 75, 91]. Subsequently, algorithmic fairness has become central in computer science research to minimize machine bias [20, 22, 69, 73]. Unfortunately, despite substantial focus on data preparation, machine learning, and algorithm design, *data structures and their potential to induce unfairness in downstream tasks have received limited attention* [98].

R1O1

Towards filling the research gap to understand potential harms and designing fair data structures, this paper revisits the *hashmap* data structure through the lens of fairness. To the best of our knowledge, this is *the* **first** *paper to study group fairness in a data structure design.* Hashmaps are a founding block in many applications such as bloom filters for set membership [28, 31, 78], hash sketches for cardinality estimation [48, 52], count sketches for frequency estimation [41], min-hashes in similarity estimation [30, 40], hashing techniques for security applications [7, 88], and many more.

Collision in a hashmap happens when the hash of two different entities is the same. Collisions are harmful as those cause *false positives*. For example, in the case of bloom filters when the hash of a query point collides with a point in a queried set, the query point is falsely classified as a set member. In such cases, in the least further computations are needed to resolve the false positives. However, this would require to explicitly storing all set members in the memory, which may not be possible in all cases. Note that false negative is impossible in case of hashmaps. That simply is because when $x = x'$, the hash of x and x' is always the same. To further motivate the problem, let us consider Example 1.

Example 1: Consider an airline security application, which aims to identify passengers who may pose a threat, and subject them for further screening and potential prevention from boarding flights. A set of criminal records is used to create a no-fly list. Due to privacy reasons, the criminals' identities are hashed and the list is a pair of {hash, gender} of individuals. The passenger hashes are matched against the no-fly list for this purpose. False positives in airline security can lead to significantly inconveniencing passengers. □

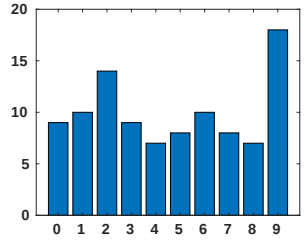


Figure 1: Distribution of 100 random integers in [0,9].

Traditional k-wise independent hashing [77, 87] aims to randomly map a key (an entity) to a *random* value (bucket) in a specific output range. However, given a set of points, it is unlikely that independent random value assignment to the points uniformly distribute the points to the buckets. For example, Fig. 1 shows the distribution of 100 independent and identically distributed (iid) random numbers, we generated in range [0,9]. While in a uniform distribution of the points, each bucket would have exactly 10 points, the random assignment did not satisfy it. On the other hand, the number of collision is minimized, when the uniform distribution is satisfied. In order to resolve this issue, *data-informed approaches* are designed, where given a set of data, the goal is to "learn" a proper hash function that uniformly distributes the data across different buckets [63, 71, 81]. Particularly, given a data set of n entities, the cumulative density function (CDF) of (the distribution represented by) the data set is constructed. Then the hashmap is created by partitioning the range of values into m buckets such that each buckets contains $\frac{n}{m}$ entities. We refer to this approach [63] as CDF-based hashmap. It has been shown that such index structures [63, 81] can outperform traditional hashmaps on practical workloads.

Conference acronym 'XX, ,

https://doi.org/XXXXXXX.XXXXXXX

To the best of our knowledge, none of the existing hashmap schemes consider fairness in terms of equal performance for different demographic groups. Given the wide range of hashmap applications, this can cause discrimination against minority groups, at least for social applications. Therefore, in this paper, we study *group fairness* defined as *equal collision probability (false positive rate) for different demographic groups*, in hashmaps. We consider the CDF-based hashmap for designing our fair data structure.

While there are many definitions of fairness, at a high level group fairness notions fall under three categories of independence, seperation, and sufficiency [22]. Our proposed notion of group fairness falls under the *independence* category, which is satisfied when the output of an algorithm is independent of the demographic groups (protected attributes). Specifically, we adopt *statistical parity*, A well-known definition under the independence category. R2O1

1.2 Applications

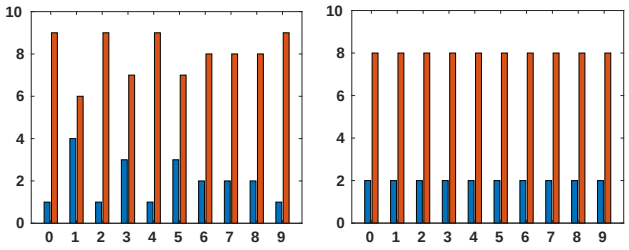
R1O1; Meta1

FairHash is a **data-informed** hashmap, extended upon CDF-based hashmap. Data-informed hashmaps are proper for applications where a large-enough workload is available for learning to uniformly distribute the data across various buckets. Particularly, data-informed hashmaps are preferred when the underlying data distribution is not uniform. Besides, the choice between data-informed hashmaps and traditional hashmaps hinges on other factors such as conflict resolution policy and memory constraints.

Nevertheless, data-informed hashmaps effectively address a broad range of practical challenges where traditional approaches fall short. For instance, when dealing with larger payloads or distributed hash maps, it is advantageous to minimize conflicts, making data-informed hashmaps more beneficial. On the other hand, in scenarios involving small keys, small values, or data following a uniform distribution, traditional hash functions are likely to perform effectively [63]. In addition to Example 1, in the following we outline a few examples of the applications of data-informed hashmaps, which demonstrate the need for FairHash in real-world scenarios.

Table Joins in Data Lakes: Consider an enterprise seeking to share its data with third-party companies. The data is stored in a data lake and includes sensitive information such as email, phone number, or even social security number of the clients serving as the primary keys for select tables. Disclosing such sensitive information constitutes a breach of client privacy, hence it needs to be masked through hashing. In such cases, where the join operation is on the hashed columns, hash-value collisions would add invalid rows to the result table. As a result, higher occurrence of false positives correlated with particular demographic groups can make the result table biased against that group. This underscores the need to employ a fair hashing scheme when anonymizing sensitive columns.

Machine Learning Datasets: Consider a scenario for constructing an ensemble model, where a large dataset is partitioned into multiple smaller random subsets, each used for training for a base model. In this setting, ensuring that the sampling process does not introduce any bias is crucial; hence, each subset of the data should maintain the same ratio of each subpopulation as present in the original distribution. While the conventional random sampling method fails to ensure such distribution alignment, employing FairHash to bucketize the data, guarantees this goal.



(a) CDF-based hashmap (b) FairHash

Figure 2: Distribution of 100 points belonging to two groups `blue` and `red` in 10 buckets.

Distributed Hashmaps and Load Balancing: Collisions incur a substantial cost in distributed hashmaps, as each collision necessitates an extra lookup request on the remote machine through RDMA, taking on the order of microseconds. Therefore, higher collision rates linked to keys from a specific demographic group may cause a notable performance disadvantage against that group. Using FairHash, for example, web servers can distribute incoming requests across multiple server instances. This guarantees an equitable distribution of the load, mitigating potential harm to a specific client if a server becomes unavailable.

Task Scheduling: FairHash can be applied in task scheduling, where tasks associated with a property (easy, medium, hard) are assigned to workers based on a hashed key. This helps distribute tasks evenly among workers, optimizing the overall system performance.

1.3 Technical Contributions

(I) Proposing FairHash. We begin our technical contributions by proposing two notions of group fairness (single and pairwise) based on collision probability disparity between demographic groups. Intuitively, single fairness is satisfied when the data is uniformly distributed across different buckets, i.e., the buckets are equi-size. Consequently, as reflected in Figure 1 traditional hashmaps do not satisfy single fairness. On the other hand, CDF-based hashmap satisfies single fairness as all of its buckets have the same size. Pairwise fairness is a stronger notion of fairness that, not only requires equi-size buckets but it also demands equal ratio of demographic groups across all buckets. To better clarify this, let us consider the distribution of 100 random (synthetic) points from two groups red and blue into 10 buckets, using CDF-based hashmap and FairHash in Figure 2a and Figure 2b. Although assigning equal number of points to each bucket, CDF-based hashmap fails to satisfy equal ratio of groups across all buckets, and hence fails on the pairwise fairness. On the other hand, using FairHash equal group ratios, hence pairwise fairness, is satisfied. Also, from the figures it is evident that pairwise fairness is a stronger notion, not only requiring equi-size buckets but also equal group ratios. We shall prove of this in § 2 after providing the formal terms and definitions. R2O2; R3O1; R1M1

(II) Ranking-based Algorithms. Next, making the observation that only the *ranking* between the points (not their value distribution) impacts the fairness of the CDF-based hashmap, we use geometric techniques to find alternative ranking of points for fair hashing. We propose multiple algorithmic results with various benefits. At a high level, our ranking-based approach, maintains the

same time and memory efficiency as of CDF-based hashing, while minimizing the unfairness (but not guaranteeing zero unfairness).
(III) Cut-based Algorithms. Our next contribution is based on the idea of adding more bins than the number of hash buckets. We propose Sweep&Cut to prove that independent of the initial distribution of points, there *always exists a fair hashing* based on the cutting approach. While guaranteeing fair hashing, Sweep&Cut is not memory efficient. Therefore, we make an interesting connection to the *necklace splitting* problem [9, 11, 70], and using some of the recent advancements [10] on it, provide multiple algorithms for FairHash. While guaranteeing the fair hashing, our algorithms achieve the same time efficiency as of the CDF-based hashing with a small increase in the memory requirement.
(IV) Discrepancy-based Algorithms. We also propose discrepancy-based algorithms that trade-off single fairness to achieve improve pairwise fairness and memory efficiency. In addition to the theoretical analysis, we conduct experiments to verify the efficiency and effectiveness of our algorithms.

2 PRELIMINARIES

Let P be a set of n points[1] in $\mathbb{R}^d$ (each point represents a tuple with d attributes), where $d \geq 1$. Let $\mathcal{G} = \{\mathbf{g}_1, \ldots, \mathbf{g}_k\}$ be a set of k demographic groups[2] (e.g., `male`, `female`). Each point $p \in P$ belongs to group $\mathbf{g}(p) \in \mathcal{G}$. By slightly abusing the notation we use $\mathbf{g}_i$ to denote the set of points in group $\mathbf{g}_i$.

Let $\mathcal{H}$ be a hashmap with m buckets, $b_1, \ldots, b_m$, and a hash function $h : \mathbb{R}^d \rightarrow [1, m]$ that maps each input point $p \in P$ to one of the m buckets. Given the pair $(P, \mathcal{H})$, we define three quantitative requirements that are used to define fairness in hash functions.

(1) *Collision Probability (Individual fairness)*: For any pair of points p, q taken uniformly at random from P, it should hold that $Pr[h(p) = h(q)] = \frac{1}{m}$.
(2) *Single fairness*: For each $i \leq k$, for any point p_i taken uniformly at random from $\mathbf{g}_i$ and any point x taken uniformly at random from P, it should hold that $Pr[h(p_i) = h(x)] = \ldots = Pr[h(p_k) = h(x)] = \frac{1}{m}$.
(3) *Pairwise fairness*: For each $i \leq k$, for any pair of points p_i, q_i taken uniformly at random from $\mathbf{g}_i$, it should hold that $Pr[h(p_i) = h(q_i)] = \ldots = Pr[h(p_k) = h(q_k)] = \frac{1}{m}$.

In our technical report [2], we show that *our definitions ensure pareto-optimality*. It means there is no other hashmap (not necessarily fair) that dominates the fair hashmap, i.e., assigns smaller collision probabilities to all groups. That is because not only we require the probabilities among different groups to be equal but we also require those to be equal to the best case.

Next, we give the exact close forms for computing the collision probability, the single fairness, and the pairwise fairness. Let $\alpha_{i,j}$ be the number of items from group i at bucket j and let $n_j = \sum_{i=1}^{k} \alpha_{i,j}$. The probabilities are calculated as follows, collision probability: $\sum_{j=1}^{m} \left(\frac{n_j}{n}\right)^2$, single fairness: $\sum_{j=1}^{m} \frac{\alpha_{i,j}}{|\mathbf{g}_i|} \cdot \frac{n_j}{n}$, pairwise fairness: $\sum_{j=1}^{m} \left(\frac{\alpha_{i,j}}{|\mathbf{g}_i|}\right)^2$. We observe that, if and only if $|\mathbf{g}_i|/m$ tuples from each group $\mathbf{g}_i$ are placed in every bucket then all quantities above are equal to the optimum value $\frac{1}{m}$.

The relationship between the three requirements: In this paper we aim to construct a hashmap $\mathcal{H}$ that satisfies (approximately) all the three requirements.

Proposition 1. *Collision probability is satisfied if and only if all buckets contain exactly the same number of points i.e., for each bucket b_j, $|b_j| = \frac{n}{m}$.*

R2O3

Proposition 2. *If collision probability is satisfied then single fairness is also satisfied.*

Proposition 3. *Pairwise fairness is satisfied if and only if for every group $\mathbf{g}_i \in \mathcal{G}$, every bucket b_j contains the same number of points from group $\mathbf{g}_i$, i.e., b_j contains $\frac{|\mathbf{g}_i|}{m}$ items from group $\mathbf{g}_i$. If pairwise fairness is satisfied then both single fairness and the collision probability are satisfied but the reverse may not necessarily hold.*

R2O3

Proof: We give the proofs to all propositions above.

R1O3; Meta3

For Proposition 1, if each bucket contains n/m items then the collision probability is $\sum_{j=1}^{m} \left(\frac{n/m}{n}\right)^2 = \frac{1}{m}$, so it is satisfied. For the other direction, we assume that the collision probability holds. Notice that $\sum_{j=1}^{m} \left(\frac{n_j}{n}\right)^2 = \frac{1}{m} \Leftrightarrow \sum_{j=1}^{m} n_j^2 = \frac{n^2}{m}$. For any integer values $n_j \geq 0$ with $\sum_{j=1}^{m} n_j = n$, it holds that $\sum_{j=1}^{m} n_j^2 \geq \sum_{j=1}^{m} \left(\frac{n}{m}\right)^2 = \frac{n^2}{m}$ and the minimum value is achieved only for $n_j = \frac{n}{m}$ for each $j \in [1, m]$. The result follows.

For Proposition 2, we have that if the collision probability is satisfied then from Proposition 1, it holds $n_j = \frac{n}{m}$. Then the single fairness for any group $\mathbf{g}_i$ is computed as $\sum_{j=1}^{m} \frac{\alpha_{i,j}}{|\mathbf{g}_i|} \frac{n_j}{n} = \sum_{j=1}^{m} \frac{\alpha_{i,j}}{|\mathbf{g}_i|} \frac{n/m}{n} = \frac{1}{m \cdot |\mathbf{g}_i|} \sum_{j=1}^{m} \alpha_{i,j} = \frac{1}{m}$, so it is satisfied.

The first part of Proposition 3 follows directly from Proposition 1, because the pairwise fairness of group $\mathbf{g}_i$ is equivalent to the collision probability assuming that $P = \mathbf{g}_i$. For the second part, if pairwise fairness is satisfied, then from the first part of Proposition 3, we know that $n_j = \frac{n}{m}$. Then from Proposition 1 the collision probability is satisfied, so from Proposition 2, the single fairness is also satisfied. Finally, the construction in the proof of Lemma 1 shows that the reverse may not necessarily hold. □

From Propositions 1 to 3, in order to satisfy the three requirements of collision probability, single fairness, and pairwise fairness, it is enough to develop a hashmap that satisfies pairwise fairness (which will generate equal-size buckets). In other words, *pairwise fairness is the strongest property*, compared to the other two.

Our goal is to design hashmaps that, while satisfying collision probability and single fairness requirements, satisfies *pairwise fairness* as the stronger notion of fairness[3]. Specifically, we want to find the hashmap $\mathcal{H}$ with m buckets to optimize the pairwise fairness.

Measuring unfairness: For a group $\mathbf{g} \in \mathcal{G}$, let $Pr_{\mathbf{g}}$ be the pairwise collision probability between its members. That is, $Pr_{\mathbf{g}} = Pr[h(p) = h(q)]$, if p and q are selected uniformly at random from the same

[1] Throughout this paper, we assume access to P is the complete set of points. For cases where instead an unbiased set of samples from P is available, our results in Tables 1 and 2 will remain in an expected manner.
[2] Demographic groups can be defined as the intersection of multiple sensitive attributes, such as `(race, gender)` as {`black-female`, ⋯}.
[3] To simplify the terms, in the rest of the paper, we use "fairness" to refer to pairwise fairness. We shall explicitly use "single fairness" when we refer to it.

Table 1: Summary of the algorithmic results with exact $(1/m)$ collision and single fairness probability.

Algorithm	Assumptions: No. Attributes	Assumptions: No. Groups	Performance*: (ε, α)-hashmap	Performance*: Query time	Performance*: Pre-processing time
Ranking	$d \geq 2$	$k \geq 2$	$(\varepsilon_R, 1)$	$O(\log m)$	$O(n^d \log n)$
Sweep&Cut	$d \geq 1$	$k \geq 2$	$(0, \frac{n}{m})$	$O(\log n)$	$O(n \log n)$
Necklace$_{2g}$	$d \geq 1$	2	$(0, 2)$	$O(\log m)$	$O(n \log n)$
Necklace$_{kg}$	$d \geq 1$	$k > 2$	$(0, k(4 + \log n))$	$O(\log(km \log n))$	$O(mk^3 \log n + knm(n+m))$

*: n is the dataset size, m is the number of buckets, and k is the number of groups.

Table 2: Summary of the algorithmic results with approximate collision and single fairness probability. The output of Ranking$^+$ holds with probability at least $1 - 1/n$.

Algorithm	Assumptions: No. Attr.	Assumptions: No. Groups	Performance*: (ε, α)-hashmap	Performance*: Query time	Performance*: Pre-processing time
Ranking	$d \geq 2$	$k \geq 2$	$(\varepsilon_D, 1)$	$O(\log m)$	$O(n^{d+2} m \log k)$
Ranking$^+$	$d \geq 2$	$k \geq 2$	$((1+\delta)\varepsilon_D + \gamma, 1)$	$O(\log m)$	$O(n + \frac{k^{d+2} m^{2d+5}}{\gamma^{2d+4}} \text{polylog}(n, \frac{1}{\delta}))$
Necklace$_{kg}$	$d \geq 1$	$k > 2$	$(\varepsilon, k(4 + \log \frac{1}{\varepsilon}))$	$O(\log(km \log \frac{1}{\varepsilon}))$	$O(mk^3 \log \frac{1}{\varepsilon} + knm(n+m))$

group $\mathbf{g}(p) = \mathbf{g}(q) = \mathbf{g}$. We measure unfairness as the max-to-min ratio in pairwise collision probabilities between the groups. Particularly, using $\frac{1}{m}$ as the min on the collision probability, we say a hashmap is ε-**unfair**, if and only if

$$\frac{\max_{\mathbf{g} \in \mathcal{G}}(Pr_{\mathbf{g}})}{1/m} \leq (1+\varepsilon) \implies \max_{\mathbf{g} \in \mathcal{G}}(Pr_{\mathbf{g}}) \leq \frac{1}{m}(1+\varepsilon) \qquad (1)$$

It is evident that for a hashmap that satisfies pairwise fairness, $\varepsilon = 0$. We use $\frac{1}{m}$ as the min on the collision probability to ensure pareto-optimality.

Memory efficiency: A hashmap with m buckets needs to store at least $m-1$ *boundary points* to separate the m buckets. While our main objective is to satisfy fairness, we also would like to minimally increase the required memory to separate the buckets. We say a hashmap with m buckets satisfies α-**memory**, if and only if it stores at most $\alpha(m-1)$ boundary points. Evidently, the most memory-efficient hashmap satisfies $\alpha = 1$.

Definition 1 ((ε, α)-hashmap). *A hashmap $\mathcal{H}$ is an (ε, α)-***hashmap** *if and only if it is ε-unfair and α-memory.*

Problem formulation: Given P, $\mathcal{G}$, and parameters m, ε, α, the goal is to design an (ε, α)-hashmap.

In this work we mostly focus on $(\varepsilon, 1)$-hashmaps and $(0, \alpha)$-hashmaps, minimizing ε and α, respectively[4]. Note that, in the best case, one would like to achieve $(0, 1)$-hashmap. That is a hash-map that is 0-unfair and does not require additional memory, i.e., satisfies 1-memory.

(Review) CDF-based hashmap [63]: is a data-informed hashmap that "learns" the cumulative density function of values over a specific attribute, and use it to place the boundaries of the m buckets such that an equal number of points $(\frac{n}{m})$ fall in each bucket. Traditional hash functions and learned hash functions (CDF) usually satisfy the collision probability and the single fairness probability, however they violate the pairwise fairness property.

[4]For simplicity, throughout the paper, we consider that the cardinality of each group $\mathbf{g}_i$ is divisible by m.

Lemma 1. *While CDF-based hashmap satisfies collision probability, hence single fairness, it may not satisfy pairwise fairness.*

Proof: From [63], it is always the case that each bucket contains the same number of tuples, i.e., $\frac{n_j}{n} = \frac{1}{m}$ for every $j = 1, \ldots, m$. Hence, from Proposition 1, the collision probability is satisfied. Then, from Proposition 2, the single fairness is also satisfied. R1O3; Meta3

Next, we show that CDF-based hashmap does not always satisfy pairwise fairness using a counter-example. Let $k = 2$, $m = 2$, $|\mathbf{g}_1| = 3$, $|\mathbf{g}_2| = 3|$, and $n = 6$. Assume the 1-dimensional tuples $\{1, 2, 3, 4, 5, 6\}$ where the first 3 of them belong to $\mathbf{g}_1$ and the last three belong to $\mathbf{g}_2$. The CDF-based hashmap will place the first three tuples into the first bucket and the last three tuples into the second bucket. By definition, the pairwise fairness is 1 (instead of 1/2) for both groups. □

Data-informed hashmaps vs. traditional hashmaps: A summary of the comparison between CDF-based and traditional hashmaps is presented in Table 3. The most major difference between CDF-based and hashmaps is that the former is data-informed. That is, the CDF-based hashmap is tailored for a specific data workload, while traditional hashmaps are data-independent, i.e., their behavior does not depend on the data those are applied on. As a result, as mentioned in § 1.2, the main assumption and the requirement of the CDF-based hashmap is access to a large-enough workload P for learning the CDF function. On the other hand, traditional hashmaps do not require access to any data workload. While the traditional hashmaps compute the hash value of a query point in constant time, CDF-based hashmap requires to run a binary search on the bucket boundaries, and hence has a query time logarithmic to m. Having learned the data distribution, the CDF-based hashmap guarantees a uniform distribution of data across different buckets, while as shown in Figure 1 traditional hashmaps cannot guarantee that. As a result, CDF-based hashmap guarantees equal collision probability and single fairness, while traditional hashmaps do not. Finally, both traditional and CDF-based hashmaps fail to guarantee pairwise fairness, a requirement that FairHash satisfies. R1O2; Meta2; R3O3

Table 3: Comparison between CDF-based and traditional hashmaps.

Hashmap	Architecture	Query time	Collision probability	Single fairness	Pairwise fairness
traditional	data-independent	$O(1)$	✗	✗	✗
CDF-based	data-dependent	$O(\log m)$	✓	✓	✗
FairHash	data-dependent	$O(\log m)$	✓	✓	✓

2.1 Overview of the algorithmic results

We propose two main approaches for defining the hashmaps, called *ranking-based* approach, and *cut-based* approach. A summary of our algorithmic results with perfect collision probability and single fairness for all groups, is shown in Table 1.

Meta1 Let W be the set of all possible unit vectors in $\mathbb{R}^d$. Given a vector $w \in W$, let P_w be the ordering defined by the projection of P onto w. Based on this ordering, we construct m equi-size buckets. In the ranking-based approaches, we focus on finding the best vector w to take the projection on that minimizes the unfairness. Let $OPT_R(P_w)$ be the smallest parameter such that an $(OPT_R(P_w), 1)$-hashmap exists in P_w with collision probability and single fairness equal to $1/m$. We define $\varepsilon_R = \min_{w \in W} OPT_R(P_w)$.

In the cut-based approaches, we define $\beta(m-1)+1$ intervals $\mathcal{I} = \{I_1, \ldots, I_{\beta(m-1)+1}\}$ defined by the $\beta(m-1)$ boundary points, such that for each point $p \in P_w$ (the projection of P on a vector w) there exists an interval $I \in \mathcal{I}$ where $p \in I$. Each interval $I \in \mathcal{I}$ is assigned to one of the m buckets. Our focus on cut-based approaches is on finding the best way to place the boundary points on a given ordering P_w. In all cases the hashmap $\mathcal{H}$ stores the vector w along with the $\beta(m-1)+1$ intervals $\mathcal{I}$ and their assigned buckets. During the query phase, given a point $q \in \mathbb{R}^d$, we first apply the projection
R3O2; $\langle w, q \rangle$ and we get the value $q_w \in \mathbb{R}$. Then we run a binary search
R3O3 on the boundary points to find the interval $I \in \mathcal{I}$ such that $q_w \in I$. We return $b(I)$ as the bucket q belongs to.

So far, we consider that the collision probability and the single fairness should be (optimum) $\frac{1}{m}$. However, this restricts the options when finding fair hashmaps. What if, there exists a hashmap with better pairwise fairness having slightly more or less than n/m items in some buckets? Hence, we introduce the notion of $\gamma-$*discrepancy* [10]. The goal is to find a hashmap with m buckets such that each bucket contains at most $(1+\gamma)\frac{|g_i|}{m}$ and at least $(1-\gamma)\frac{|g_i|}{m}$ points from each group $i \leq k$. A summary of our algorithmic results with approximate collision probability and single fairness for all groups, is shown in Table 2. Let W be the set of all possible unit vectors in $\mathbb{R}^d$. Let $OPT_D(P_w)$ be the smallest parameter such that a hashmap with $OPT_D(P_w)$-discrepancy exists in P_w. We define $\varepsilon_D = \min_{w \in W} OPT_D(P_w)$.

R1O4; **Fairness and memory efficiency trade-off.** Meta4 Ideally, one would like to develop a hashmap that is 0-unfair and 1-memory. But in practice, depending on the distribution of the data, achieving both at the same time may not be possible. For cases where fairness is a hard constraint, cut-based algorithms are preferred, as those guarantee 0-unfairness, irrespective of the data distribution. But that is achieved at a cost of increasing memory usage. On the other hand, ranking-based algorithms minimize unfairness without requiring any extra memory but do not 0-unfairness; hence those are fit when memory is a hard constraint. Last but not least, the discrepancy-based algorithms provide a trade-off between pairwise and single fairness. Specifically, these algorithms do not guarantee to contain exactly $\frac{n}{m}$ points in each bucket, and hence, may not satisfy the first two requirements (individual and single fairness). However, for cases where adding a small amount of single unfairness is tolerable, the discrepancy-based algorithms may further reduce the pairwise unfairness of the ranking-based and the memory bound of the cut-based algorithms.

Remark. In all cases, for simplicity, we assume that our algorithms have access to the entire input set in order to compute a near-optimal hashmap. However, our algorithms can work in expectation, if an unbiased sample set from the input set is provided. We verify this, experimentally in Section 6.7.

3 RANKING-BASED ALGORITHMS

We start our contribution by defining a space of $(\varepsilon, 1)$-hashmaps, and designing algorithms to find the (near) optimum hashmap with the *smallest* ε. By definition, recall that ε_R is the smallest unfairness we can find with this technique (assuming 1-memory).

Our key observation is that *only the ordering between the tuples matters* when it comes to pairwise fairness, not the attribute values. Hence, assuming that $d > 1$, our idea is to combine the attribute values of a point $p \in P$ into a single score $f(p)$, using a function $f : \mathbb{R}^d \to \mathbb{R}$ called the ranking function, and consider the ordering of the points based on their scores for creating the hashmap. Then, in a class of ranking functions, the objective is to find the one that returns the best $(\varepsilon, 1)$-hashmap with the smallest value of ε. Of course, $f(p)$ needs to be computed efficiently, ideally in *constant time*. Therefore, we select linear ranking functions, where the points are linearly projected on a vector $w \in \mathbb{R}^d$. That is, $f_w(p) = \langle p, w \rangle = p^\top w$. Notice that the value $f_w(p)$ can be computed in $O(d) = O(1)$ time and it defines an ordering between the different tuples $p \in P$. For a vector w, let P_w be the ordering defined by f_w and let $P_w[j]$ be the j-th largest tuple in the ordering P_w. Given the ordering defined by a ranking function f_w, we construct $(m-1)$ boundaries to partition the data into m equi-size buckets (each containing $\frac{n}{m}$ points). Then a natural algorithm to construct an $(\varepsilon, 1)$-hashmap is to run the subroutine for each possible ranking function and in the end return the best ranking function we found.

For simplicity, we describe our first method in $\mathbb{R}^2$. All our algorithms are extended to any constant dimension d.

It is known that for $P \subset \mathbb{R}^2$ there exist $O(n^2)$ combinatorially different ranking functions [50]. We can easily compute them if we work in the dual space [50]. For a point $p = (x_p, y_p) \in P$ we define the dual line $\lambda(p) : x_p x_1 + y_p x_2 = 1$. Let $\Lambda = \{\lambda(p) \mid p \in P\}$ be the set of n lines. Let $\mathcal{A}(\Lambda)$ be the arrangement [50] of Λ, which is defined as the decomposition of $\mathbb{R}^2$ into connected (open) cells of dimensions 0, 1, 2 (i.e., point, line segment, and convex polygon) induced by P. It is known that $\mathcal{A}(\Lambda)$ has $O(n^2)$ cells and it can be computed in $O(n^2 \log n)$ time [50].

Given a vector w, the ordering P_w is the same as the ordering defined by the intersections of Λ with the line supporting w. Hence, someone can identify all possible ranking functions f_w by traversing all intersection points in $\mathcal{A}(\Lambda)$. Each intersection in $\mathcal{A}(\Lambda)$ is represented by a triplet (p, q, w) denoting that the lines $\lambda(p), \lambda(q)$ are intersecting and the intersection point lies on the line supporting the vector w. Let $\mathcal{W}$ be the array of size $O(n^2)$ containing all intersection triplets sorted in ascending order of the vectors' angles

with the x-axis. Let $\mathcal{W}[i]$ denote the i-th triplet (p_i, q_i, w_i). It is known that the orderings $P_{w_i}, P_{w_{i+1}}$ differ by swapping the ranking of two consecutive items. Without loss of generality, we assume that for $(p_i, q_i, w_i) \in \mathcal{W}$, the ranking of p_i is higher than the ranking of q_i for vectors with angle greater than w_i. The array $\mathcal{W}$ can be constructed in $O(n^2 \log n)$ time.

Algorithm. Using $\mathcal{W}$, the goal is to find the best $(\varepsilon, 1)$-hashmap over all vectors satisfying the collision probability and the single fairness. The high level idea is to consider each different vector $w \in \mathcal{W}$, and for each ordering P_w, find the hashmap that satisfies the collision probability and the single fairness measuring the unfairness. In the end, return the vector w along with the boundaries of the hashmap with the smallest unfairness we found. If we execute it in a straightforward manner, we would have $O(|\mathcal{W}| \cdot n \log n) = O(n^{d+1} \log n)$ time algorithm. Next, we present a more efficient implementation applying fast update operations each time that we visit a new vector.

The pseudo-code of the algorithm in $\mathbb{R}^2$ is provided in Algorithm 1. The algorithm starts with the initialization of some useful variables and data structures. Let w_0 be the unit vector with angle to the x-axis slightly smaller than w_1's angle. We visit each point in P and we find the (current) ordering, denoted by P_w, sorting the projections of P onto w_0. Next, we compute the best $(\varepsilon, 1)$-hashmap in P_w. The only way to achieve optimum collision probability and single fairness in P_w is to construct exactly m buckets containing the same total number of tuples in each of them. Identifying the buckets (and constructing the hashmap) in P_w is trivial because every bucket should contain exactly n/m items. Hence, the boundaries of the j-th hashmap bucket are defined as $P_w[(j-1)\frac{n}{m}+1], P_w[j\frac{n}{m}]$, for $j \in [1, m]$. Next, we compute the unfairness with respect to w_0. For each group $\mathbf{g}_i$ that contains at least one item in j-th bucket we set $\alpha_{i,j} = 0$. We use the notation $\alpha_{i,j}$ to denote the number of tuples from group i in j-th bucket. Let $P_w[\ell]$ be the next item in the j-th bucket, and let $P_w[\ell] \in \mathbf{g}_i$. We update $\alpha_{i,j} \leftarrow \alpha_{i,j} + 1$. After traversing all items in P_w, we compute the pairwise fairness of group $\mathbf{g}_i$ as $Pr_i = \sum_{j=1}^{m} \left(\frac{\alpha_{i,j}}{|\mathbf{g}_i|}\right)^2$. The unfairness with respect to w_0 is $\varepsilon = m \cdot \max_{i \le k} Pr_i - 1$. After computing all values Pr_i, we construct a max heap M over $\{Pr_1, \ldots, Pr_k\}$. Let $w^* = w_0$. We run the algorithm visiting each vector in $\mathcal{W}$ maintaining $\varepsilon, M, w^*, P_w, Pr_i, \alpha_{i,j}$ for each i and j.

As the algorithm proceeds, assume that we visit a triplet (p_s, q_s, w_s) in $\mathcal{W}$. If p_s and q_s belong in the same bucket, we only update the positions of p_s, q_s in P_w and we continue with the next vector. Similarly, if both p_s, q_s belong in the same group $\mathbf{g}_i$ we only update the position of them in P_w and we continue with the next vector. Next, we consider the more interesting case where $p_s \in \mathbf{g}_i$ belongs in the j-th bucket and $q_s \in \mathbf{g}_\ell$ belongs in the $(j+1)$-th bucket of P_w, with $i \neq \ell$, just before we visit (p_s, q_s, w_s). We update P_w as in the other cases. However, now we need to update the pairwise fairness. In particular, we update,

$$Pr_i = Pr_i - \left(\frac{\alpha_{i,j}}{|\mathbf{g}_i|}\right)^2 - \left(\frac{\alpha_{i,j+1}}{|\mathbf{g}_i|}\right)^2 + \left(\frac{\alpha_{i,j}-1}{|\mathbf{g}_i|}\right)^2 + \left(\frac{\alpha_{i,j+1}+1}{|\mathbf{g}_i|}\right)^2,$$

and similarly

$$Pr_\ell = Pr_\ell - \left(\frac{\alpha_{\ell,j+1}}{|\mathbf{g}_\ell|}\right)^2 - \left(\frac{\alpha_{\ell,j}}{|\mathbf{g}_\ell|}\right)^2 + \left(\frac{\alpha_{\ell,j+1}-1}{|\mathbf{g}_\ell|}\right)^2 + \left(\frac{\alpha_{\ell,j}+1}{|\mathbf{g}_\ell|}\right)^2.$$

Algorithm 1 RANKING$_{2D}$

Input: Set of points $P \in \mathbb{R}^2$
Output: vector w^* and corresponding boundaries B

Construct and sort the vectors in $\mathcal{W}$ with respect to their angles;
$w_0 \leftarrow$ vector with angle to the x-axis slightly smaller than w_1's angle;
$P_w \leftarrow$ sorted projection of P onto w_0;
$b_j \leftarrow \left(P_w[(j-1)\frac{n}{m}+1], P_w[j\frac{n}{m}]\right) \quad \forall j = 1, \ldots, m$;
$\alpha_{i,j} \leftarrow |\mathbf{g}_i \cap b_j|, \quad \forall i = 1, \ldots, k, j = 1, \ldots, m$;
$Pr_i \leftarrow \sum_{j=1}^{m} \left(\frac{a_{i,j}}{|\mathbf{g}_i|}\right)^2, \quad \forall i = 1, \ldots, k$;
$M \leftarrow$ Max-Heap on $Pr_i, \forall i = 1, \ldots, m$;
$\varepsilon \leftarrow m \cdot \max_{i \le k} Pr_i - 1$; $w^* = w_0$;
for $(p_s, q_s, w_s) \in \mathcal{W}$ **do**
  if p_s, q_s belong in the same bucket OR $\mathbf{g}(p_s) == \mathbf{g}(q_s)$ **then**
    Swap p_s, q_s and update P_w;
  else
    Let $p_s \in b_j, \mathbf{g}(p_s) = \mathbf{g}_i, q_s \in b_{j+1}, \mathbf{g}(q_s) = \mathbf{g}_\ell$;
    $Pr_i \leftarrow Pr_i - \left(\frac{\alpha_{i,j}}{|\mathbf{g}_i|}\right)^2 - \left(\frac{\alpha_{i,j+1}}{|\mathbf{g}_i|}\right)^2 + \left(\frac{\alpha_{i,j}-1}{|\mathbf{g}_i|}\right)^2 + \left(\frac{\alpha_{i,j+1}+1}{|\mathbf{g}_i|}\right)^2$;
    $Pr_\ell \leftarrow Pr_\ell - \left(\frac{\alpha_{\ell,j+1}}{|\mathbf{g}_\ell|}\right)^2 - \left(\frac{\alpha_{\ell,j}}{|\mathbf{g}_\ell|}\right)^2 + \left(\frac{\alpha_{\ell,j+1}-1}{|\mathbf{g}_\ell|}\right)^2 + \left(\frac{\alpha_{\ell,j+1}}{|\mathbf{g}_\ell|}\right)^2$;
    $\alpha_{i,j}$=$\alpha_{i,j}$−1, $\alpha_{i,j+1} = \alpha_{i,j+1}$+1, $\alpha_{\ell,j+1} = \alpha_{\ell,j+1}-1$, $\alpha_{\ell,j} = \alpha_{\ell,j}+1$;
    Update Pr_i, Pr_ℓ in M;
    if $m \cdot M.top() - 1 < \varepsilon$ **then** $\{\varepsilon = m \cdot M.top() - 1$; $w^* = w_s$; $\}$
for $j = 1$ to m **do** $B_j \leftarrow \frac{P_{w^*}[j\frac{n}{m}]+P_{w^*}[j\frac{n}{m}+1]}{2}$; // right boundary of b_j
return (w^*, B);

Based on the new values of Pr_i, Pr_ℓ we update the max heap M. We also update $\alpha_{i,j} = \alpha_{i,j} - 1$, $\alpha_{i,j+1} = \alpha_{i,j+1} + 1$, $\alpha_{\ell,j+1} = \alpha_{\ell,j+1} - 1$, $\alpha_{\ell,j} = \alpha_{\ell,j} + 1$. If the top value of M is smaller than ε, then we update ε with the top value of M and we update $w^* = w_s$. After traversing all vectors in $\mathcal{W}$ we return the best vector w^* we found. The boundaries can easily be constructed by finding the ordering P_{w^*} satisfying that each bucket contains exactly n/m items.

Analysis. The correctness of the algorithm follows from the definitions. Next, we focus on the running time. We need $O(n^2 \log n)$ to construct $\mathcal{A}(\Lambda)$ and $O(n \log n)$ additional time to initialize $\varepsilon, M, w^*, P_w, Pr_i, \alpha_{i,j}$. For each new vector w_s we visit, we update P_w in $O(1)$ time by storing the position of each item $p \in P$ in P_w using an auxiliary array. All variables $Pr_i, Pr_\ell, \alpha_{i,j}, \alpha_{i,j+1}, \alpha_{\ell,j}, \alpha_{\ell,j+1}$ are updated executing simple arithmetic operations so the update requires $O(1)$ time. The max heap M is updated in $O(\log m)$ time. Hence, for each vector $w_s \in \mathcal{W}$ we spend $O(\log m)$ time. There are $O(n^2)$ vectors in $\mathcal{W}$ so the overall time of our algorithm is $O(n^2 \log n)$.

Extension to $d \ge 2$: The algorithm can straightforwardly be extended to any constant dimension d. Using known results [50], we can construct the arrangement of $O(n)$ hyperplanes in $O(n^d \log n)$ time. Then, in $O(n^d \log n)$ time in total, we can traverse all combinatorially different vectors such that the orderings $P_{w_i}, P_{w_{i+1}}$ between two consecutive vectors w_i, w_{i+1} differ by swapping the ranking of two consecutive items. Our algorithm is applied to all $O(n^d)$ vectors with the same way as described above. Hence, we conclude to the next theorem.

THEOREM 1. *Let P be a set of n tuples in $\mathbb{R}^d$. There exists an algorithm that computes an $(\varepsilon_R, 1)$-hashmap satisfying the collision probability and the single fairness in $O(n^d \log n)$ time.*

Sampled vectors. So far, we consider all possible vectors $w \in \mathcal{W}$ to return the one with the optimum pairwise fairness. In practice, instead of visiting $O(n^d)$ vectors, we sample a large enough set of vectors $\widehat{\mathcal{W}}$ from $\mathbb{R}^d$. We run our algorithm using the set of vectors $\widehat{\mathcal{W}}$ instead of $\mathcal{W}$ and we return the vector that leads to the minimum unfairness ε. This algorithm runs in $O(|\widehat{\mathcal{W}}|n \log n)$.

4 CUT-BASED ALGORITHMS

The ranking-based algorithms proposed in Section 3, cannot guarantee 0-unfairness. In other words, by re-ranking the n points using linear projections in $\mathbb{R}^d$, those can only achieve an $(\varepsilon_R, 1)$-hashmap.

In this section, we introduce a new technique with the aim to *guarantee 0-unfairness*. So far, our approach has been to partition the values (after projection) into m equi-size buckets. In other words, each bucket b_i is a continuous range of values specified by two boundary points. The observation we make in this section is that the *buckets do not necessarily need to be continuous*. Specifically, we can partition the values into more than m bins while in a many-to-one matching, several bins are assigned to each bucket. Using this idea, in the following we propose two approaches for developing fair hashmaps with 0-unfairness.

4.1 Sweep&Cut

An interesting question we explore in this section is whether a cut-based algorithm exists that always guarantee 0-unfairness.

Theorem 2. *Consider a set of n points in $\mathbb{R}$, where each point p belongs to a group $g(p) \in \{\mathbf{g}_1, \cdots, \mathbf{g}_k\}$. Independent of how the points are distributed and their orders, there always exist a cut-based hashmap that is* **0-unfair**.

We prove the theorem by providing the Sweep&Cut algorithm (Algorithm 2) that always finds a 0-unfair hashmap. Without loss of generality, let $L = \langle p_1, p_2, \cdots, p_n \rangle$ be the sorted list of points in P based on their values on an attribute x. Sweep&Cut sweeps through L from p_1 to p_n twice. During the first sweep (Lines 3 to 5), the algorithm keeps track of the number of instances it has observed from each group $\mathbf{g}_i$. The algorithm uses H^{tmp} to mark which bucket each point should fall into, such that each bucket contains $\frac{|\mathbf{g}_i|}{m}$ instances from each group $\mathbf{g}_i$. During the second pass (Lines 7 to 12), the algorithm compares the neighboring points and as long as those should belong to the same bucket (Line 8), there is no need to introduce a new boundary. Otherwise, the algorithm adds a new boundary (in array B) to introduce a new bin, while assigning the bucket numbers in H. Finally, the algorithm returns the bin boundaries and the corresponding buckets.

4.1.1 Analysis. Sweep&Cut makes two linear-time passes over P. Therefore, considering the time to sort P based on x, its time complexity is $O(n \log n)$. The algorithm assigns $\frac{n}{m}$ point to each bucket; hence, following Propositions 1 and 2 is satisfies collision probability and single fairness. More importantly, Sweep&Cut assigns $\frac{|\mathbf{g}_i|}{m}$ point from each group $\mathbf{g}_i$ to each bucket. As a result, for any pair p_i, q_i in $\mathbf{g}_i$, $Pr[h(p_i) = h(q_i)] = \frac{1}{m}$. Therefore, the hashmap generated by Sweep&Cut is **0-unfair**, proving Theorem 2.

Algorithm 2 Sweep&Cut

Input: The set of points P
Output: Bin boundaries B and corresponding buckets H

1: $\langle p_1, p_2, \cdots, p_n \rangle \leftarrow$ **sort** P based on an attribute x
2: **for** $j = 1$ to k **do** $c_j \leftarrow 0$; // # of instances observed from $\mathbf{g}_i$
3: **for** $i = 1$ to n **do**
4: let $\mathbf{g}_j = \mathbf{g}(p_i)$; $c_j \leftarrow c_j + 1$
5: $H_i^{tmp} \leftarrow \left\lfloor \frac{c_j \times m}{|\mathbf{g}_j|} \right\rfloor + 1$
6: $i \leftarrow 0$; $j \leftarrow 0$
7: **while True do**
8: **while** ($i < n$ and $H_i^{tmp} == H_{i+1}^{tmp}$) **do** $i \leftarrow i + 1$
9: $H_j \leftarrow H_i^{tmp}$; // the bucket assigned to the j-th bin
10: **if** $i == n$ **then break**
11: $B_j = \frac{p_i[x] + p_{i+1}[x]}{2}$; // the right boundary of the j-th bin
12: $j \leftarrow j + 1$
13: **return** (B, H)

Despite guaranteeing 0-unfairness, Sweep&Cut is not efficient in terms of memory. Particularly, in the worst case, it can introduce as much as $O(n)$ boundaries.

In a best case, where the points are already ordered in a way that dividing them into m equi-size buckets is already fair, Sweep&Cut will add m bins ($m-1$ boundaries). On the other hand, in adversarial setting, a large potion of the neighboring pairs within the ordering belong to different groups with different hash buckets assigned to them. Therefore, in the worst-case Sweep&Cut may add up to $O(n)$ boundaries, making it satisfy $\frac{n}{m}$-**memory**. Applying the binary search on the $O(n)$ bin boundaries, the query time of Sweep&Cut hashmap is in the worst-case $O(\log n)$.

Lemma 2. *In the binary demographic groups cases, where $\mathcal{G} = \{\mathbf{g}_1, \mathbf{g}_2\}$ and $r = |\mathbf{g}_1|$, the expected number of bins added by Sweep&Cut is bounded by $2\left(\frac{r(n-r)}{n} + m\right)$.*

R1O3; Meta3

Proof: Sweep&Cut adds at most $m - 1$ boundaries between the neighboring pairs that both belong to the same group, simply because a boundary between neighboring pair can only happen when moving from one bucket to the next while there are m buckets.

In order to find the upper-bound on the number of bins added, in the following we compute the expected number of neighboring pairs from different groups. Consider the in the sorted list of points $\langle p_1, \cdots, p_n \rangle$ based on the attribute x. The probability that a point p_i belongs to group g_1 is $Pr_1 = Pr(\mathbf{g}(p_i) = \mathbf{g}_1) = \frac{r}{n}$. Now consider two consecutive points p_i, p_{i+1}, in the list. The probability that these two belong to different groups is $2Pr_1(1-Pr_1)$. Let $\mathcal{B}$ be the random variable representing the number of pairs from different groups. We have, $E[\mathcal{B}] = \sum_{i=1}^{n-1} 2Pr_1(1 - Pr_1) = 2(n-1)\frac{r}{n}(1 - \frac{r}{n}) < \frac{2r(n-r)}{n}$. Therefore, $E[\text{No. bins}] \leq E[\mathcal{B}] + 2m < 2\left(\frac{r(n-r)}{n} + m\right)$. □

4.2 Transforming to Necklace Splitting

Although guaranteeing 0-unfairness, Sweep&Cut is not memory efficient. The question we seek to answer in this section is whether it is possible to guarantee 0-unfairness while introducing significantly less number of bins, close to m. In particular, we make an interesting connection to the so-called *necklace splitting* problem [9, 11, 70], and use the recent advancements [10] on this problem by the math and theory community to solve the fair hashmap problem.

(Review) Necklace Splitting: Consider a necklace of T beads of n' types. For each type $i \le n'$, let m'_i be the number of beads with type i, and let $m' = \max_i m'_i$. The objective is to divide the beads between k' agents, such that (a) all agents receive exactly the same amount of beads from each type and (b) the number of splits to the necklace is minimized.

Reduction: Given an instance of the fair hashmap problem, let $L = \langle p_1, p_2, \cdots, p_n \rangle$ be the ordering of P based on an attribute x. The problem gets reduced to necklace splitting as following: The points in P get mapped into the $T = n$ bead, distributed with the ordering $\langle p_1, p_2, \cdots, p_n \rangle$ in the necklace. The k groups in $\mathcal{G}$ get translated to the $n' = k$ bead types $\{\mathbf{g}_1, \cdots, \mathbf{g}_k\}$. The m buckets translate to the $k' = m$ agents.

Given the necklace splitting output, each split of the necklace translates into a bin. The bin is assigned to the corresponding bucket of the agent who received the necklace split. Using this reduction, an optimal solution to the necklace splitting is the fair hashmap with minimum number of cuts: first, since all party in necklace splitting receive equal number of each bead type, the corresponding hashmap satisfies **0-unfairness**, as well as the collision probability and single fairness requirements; second, since the necklace splitting minimizes the number of splits to the necklace, it adds minimum number of bins to the fair hashmap problem, i.e., it finds the optimal fair hashmap on L, with minimum number of cuts. Using this mapping, in the rest of the section we adapt the recent results for solving necklace splitting for fair hashmap. In particular, Alon and Graur [10] propose polynomial time algorithms (with respect to number of beads) for the Necklace Splitting problem and the ε-approximate version of the problem.

4.2.1 Binary groups. The fair hashmap problem when there are two groups $\{\mathbf{g}_1, \mathbf{g}_2\}$, maps to the necklace splitting instance with two bead types. While a straightforward implementation of the algorithm in [10]-(Proposition 2) leads to an $O(n(\log n + m))$ algorithm, in Necklace$_{2g}$ Algorithm 3, we propose an optimal time algorithm that guarantees splitting a necklace with *at most* $2(m-1)$ cuts in only $O(n \log n)$ time.

Algorithm. Without loss of generality, let $C = \langle p_1, p_2, \cdots, p_n \rangle$ be the sorted list of points in P based on their values on an attribute x. Necklace$_{2g}$ views C as a circle by considering p_n before p_1. It uses modulo to size of the list ($\%|C|$) to move along the circle. The key idea is that the circle C *always* has at least one consecutive window of size $\frac{n}{m}$ that contains $\frac{|\mathbf{g}_1|}{m}$ points from $\mathbf{g}_1$ (and hence $\frac{|\mathbf{g}_2|}{m}$ points from $\mathbf{g}_2$), see [10]. Hence, we design an algorithm to find such windows efficiently. We initialize a list T such that $T[j]$ contains the number of items from group $\mathbf{g}_1$ between $C[j]$ and $C[(j+n/m-1)\%|C|]$. Furthermore, we initialize the list X such that $X[j]$ is true if and only if $T[j] = |\mathbf{g}_1|/m$, i.e., the window from $C[j]$ to $C[(j+n/m-1)\%|C|]$ is a good candidate for a cut. All indexes are initialized in lines 4–9 of Algorithm 3. In order to bound the running time of the new algorithm, we assume that $X[j]$ also stores a pointer to the j-th elements in lists C and T. Furthermore, we assume that all Boolean variables in X are stored in a max heap M_X(if $X[j]$ = true and $X[i]$ = false then $X[j] > X[i]$). We use M_X to call $M_X.top()$ that returns the top item in max heap, i.e., it returns a j such that $X[j]$ = true, in $O(1)$ time.

The algorithm is executed in iterations until the list C is non-empty. In each iteration, we find a window of size n/m containing exactly $|\mathbf{g}_1|/m$ items from group $\mathbf{g}_1$ (so it also contains exactly $|\mathbf{g}_2|/m$ items from group $\mathbf{g}_2$). More specifically, in line 11 we find j such that $T[j] = |\mathbf{g}_1|/m$. In line 20 we define the new cut we find and in lines 21–23 we remove the cut from our lists to continue with the next iteration. The points within the cut are marked for the bucket bkt. In lines 13–19 we update the values of T (and hence the values of X) so that T and X have the correct values after removing the window from $C[j]$ to $C[(j+n/m-1)\%|C|]$. Hence, we can continue searching for the next window containing $|\mathbf{g}_1|/m$ tuples from group $\mathbf{g}_1$ in the next iteration. Finally, it sorts the discovered cuts and assigns the bin boundaries B and the corresponding buckets H.

Theorem 3. *In the binary demographic group cases, there exists an algorithm that finds a $(0,2)$-hashmap satisfying the collision probability and the single fairness in $O(n \log n)$ time.*

Proof: In each iteration of the algorithm, we remove a window containing $|\mathbf{g}_1|/m$ items from group $\mathbf{g}_1$ and $|\mathbf{g}_2|/m$ items from group $\mathbf{g}_2$. Hence, the correctness of our algorithm follows by the discrete intermediate value theorem and [10]. Next, we show that the running time is $O(n \log n)$. It takes $O(n \log n)$ to sort based on attribute x. Then it takes $O(n/m + n) = O(n)$ to initialize T and X. In each iteration of the while loop at line 10 we remove n/m items, so in total it runs for m iterations. In each iteration, we get j at line 11 in $O(1)$ time using the max heap. The for loop in line 13 is executed for $O(n/m)$ rounds. In each round, we need $O(1)$ time to update T. It also takes $O(1)$ time to update a value in X, and $O(\log n)$ time to update the max heap. Finally, the for loop in line 21 runs for $O(n/m)$ rounds. In each round it takes $O(1)$ time to remove an item from lists C, T, X and $O(\log n)$ time to update the max heap. Overall Algorithm 3 runs in $O(n \log n + m\frac{n}{m} \log n) = O(n \log n)$ time. □

5 DISCREPANCY-BASED HASHMAPS

A hashmap satisfies γ-discrepancy if and only if each bucket contains at least $(1-\gamma)\frac{|\mathbf{g}_i|}{m}$ and at most $(1+\gamma)\frac{|\mathbf{g}_i|}{m}$ points from each group $\mathbf{g}_i$. In this section, we first show that a hashmap that satisfies γ-discrepancy has bounded collision probability, single fairness, and pairwise fairness. Then, we propose efficient algorithms that construct γ-discrepancy hashmaps, where γ is close to the optimum. R2O4

Let P_w be the ordering of points in P based on a vector w and let $\mathcal{H}$ be a hashmap constructed on P_w. Recall that Pr_i is defined as the pairwise fairness value of $\mathcal{H}$ for group $\mathbf{g}_i$. Let Cp be the collision probability and Sp_i the single fairness of group $\mathbf{g}_i$.

Lemma 3. *Let $\mathcal{H}$ be a hashmap satisfying γ-discrepancy. Then $Cp \le \frac{1+\gamma}{m}$, $\frac{1-\gamma}{m} \le Sp_i \le \frac{1+\gamma}{m}$ and $Pr_i \le \frac{1+\gamma}{m}$ for each group $\mathbf{g}_i$.*

Proof: Recall that n_j is the number of items in bucket j and $\alpha_{i,j}$ is the number of items from group i in bucket j. Notice that $\sum_{j=1}^{m} \alpha_{i,j} = |\mathbf{g}_i|$ and $\sum_{j=1}^{m} n_j = n$. We have, R1O3; Meta3

$$Pr_i = \sum_{j=1}^{m} \left(\frac{\alpha_{i,j}}{|\mathbf{g}_i|}\right)^2 \le \sum_{j=1}^{m} \frac{(1+\gamma)\frac{\mathbf{g}_i}{m}}{|\mathbf{g}_i|} \cdot \frac{\alpha_{i,j}}{|\mathbf{g}_i|} = \frac{1+\gamma}{m} \sum_{j=1}^{m} \frac{\alpha_{i,j}}{|\mathbf{g}_i|} = \frac{1+\gamma}{m}.$$

Similarly we show that $Cp = \sum_{j=1}^{m} \left(\frac{n_j}{n}\right)^2 \le \frac{1+\gamma}{m}$.

Algorithm 3 Necklace$_{2g}$

Input: The set of points P (with two groups $\{\mathbf{g}_1, \mathbf{g}_2\}$)
Output: Bin boundaries B and corresponding buckets H

$C = \langle p_1, p_2, \cdots, p_n \rangle \leftarrow$ **sort** P based on an attribute x;
for $i = 0$ **to** $n-1$ **do** $\{T[i] \leftarrow 0; X[i] \leftarrow$ false$\}$
$M_X \leftarrow$ max heap storing X; $\sigma_1 \leftarrow 0$; $bkt \leftarrow 0$;
//Initialize T
for $i = 0$ **to** n/m **do** {**if** $\mathbf{g}(C[i]) == \mathbf{g}_1$ **then** $\sigma_1 \leftarrow \sigma_1 + 1$}
for $i = 0$ to $n-1$ **do**
  $T[i] \leftarrow \sigma_1$;
  if $T[i] == |\mathbf{g}_1|/m$ **then** $X[i] \leftarrow$ true; M_X.update$(X[i])$;
  if $\mathbf{g}(C[i]) == \mathbf{g}_1$ **then** $\sigma_1 \leftarrow \sigma_1 - 1$;
  if $\mathbf{g}(C[(i+n/m)\%|C|]) == \mathbf{g}_1$ **then** $\sigma_1 \leftarrow \sigma_1 + 1$;
while $|C| > 0$ **do**
  $j \leftarrow M_X.top()$; //Find a window $[j, (j+n/m-1)\%|C|]$ with $X[j] =$ true
  //Update T and X removing the window $[j, (j+n/m-1)\%|C|]$
  $\sigma \leftarrow T[(j+n/m)\%|C|]$;
  for $i = j-1$ to $j - n/m + 1$ with step -1 **do**
    if $i < 0$ **then** $i \leftarrow |C| + i$;
    if $\mathbf{g}(C[(i+2\cdot n/m)\%|C|]) == \mathbf{g}_1$ **then** $\sigma \leftarrow \sigma - 1$;
    if $\mathbf{g}(C[i]) == \mathbf{g}_1$ **then** $\sigma \leftarrow \sigma + 1$;
    $T[i] \leftarrow \sigma$;
    $X[i] \leftarrow$ true **if** $(T[i] == |\mathbf{g}_1|/m)$ **else** false
    $M_X.update(X[i])$;
  cuts $\leftarrow$ cuts $\cup \{j, (j+n/m-1)\%|C|\}$
  //Remove window $[j, (j+n/m-1)\%|C|]$
  for $i \in [0, n/m)$ **do**
    $H^{tmp}_{C[(i+j)\%|C|]} \leftarrow bkt$; $Remove(C[(i+j)\%|C|])$;
    $Remove(T[(i+j)\%|C|])$; $Remove(X[(i+j)\%|C|])$;
  $bkt \leftarrow bkt + 1$;
sort(cuts)
for $j = 0$ to |cuts| **do**
  Let p_i be the rightmost tuple in the j-th bin;
  $B_j \leftarrow \frac{p_i[x]+p_{i+1}[x]}{2}$; // the right boundary of the j-th bin
  $H_j \leftarrow H^{tmp}_i$; // the bucket assigned to the j-th bin
return (B, H)

Using the same arguments it also holds that

$$Sp_i = \sum_{j=1}^{m} \frac{\alpha_{i,j}}{|\mathbf{g}_i|}\frac{n_j}{n} \leq \frac{1+\gamma}{m} \quad \text{and} \quad Sp_i \geq \frac{1-\gamma}{m}.$$

□

Next, we describe a dynamic programming algorithm to find a hashmap with the smallest discrepancy. In technical report [2] we show a faster randomized algorithm approximating the smallest discrepancy. Finally, we describe a simple heuristic that works as a post-processing method to further improve the pairwise fairness.

R2O4 **Dynamic Programming algorithm.** Let P_w be the ordering of the items for a vector $w \in \mathcal{W}$. Let Disc$[i, j]$ be the discrepancy of the optimum partition among the first i items in P_w using j buckets. Let also $\mathcal{D}(a, b)$ be the discrepancy of the bucket including all items in the window $[a, b]$ in P_w, i.e., $\{P_w[a], P_w[a+1], \ldots, P_w[b]\}$. We define the recursive relation

$$\text{Disc}[i, j] = \min_{1 \leq x < i} \max\{\text{Disc}[x-1, j-1], \mathcal{D}(x, i)\}$$

Given i, j, our algorithm computes the j-th bucket with right boundary i, trying all left boundaries $x < i$, that leads to a partition with minimum discrepancy over the first i items. In order to efficiently implement the algorithm, each time we try a new left boundary x, we do not compute $\mathcal{D}(x, i)$ from scratch. Instead, we maintain and update a max-heap of size k storing the discrepancy of every group $\mathbf{g}_i$ in the window $[x, i]$. When we compute $[x-1, i]$ we update one element in the max-heap and compute $\mathcal{D}(x-1, i)$ in $O(\log k)$ time. The table Disc has $O(nm)$ cells and for each cell we spend $O(n \log k)$ time. By definition, Disc$[n, m]$ computes ε_D. Doing standard modifications, it is straightforward to return the partition, instead of the discrepancy ε_D. By repeating the algorithm above for every $w \in \mathcal{W}$, we conclude to the next theorem.

Theorem 4. *Let P be a set of n tuples in $\mathbb{R}^d$. There exists an algorithm that computes an $(\varepsilon_D, 1)$-hashmap in $O(n^{d+2} m \log n \log k)$ time, satisfying ε_D-approximation in collision probability and single fairness.*

For parameters γ, δ, in technical report [2], we show a randomized algorithm, to compute a $((1+\delta)\varepsilon_D + \gamma, 1)$-hashmap with collision probability at most $\frac{1+(1+\delta)\varepsilon_D+\gamma}{m}$ and single fairness in the range $[\frac{1-(1+\delta)\varepsilon_D-\gamma}{m}, \frac{1+(1+\delta)\varepsilon_D+\gamma}{m}]$, in time $O(n + \text{poly}(m, k, \delta, \gamma))$.

Local-search based heuristic. So far, we consider algorithms that return a hashmap satisfying (approximately) ε_D-discrepancy. From Lemma 3 we know that a hashmap satisfying ε_D-discrepancy is a $(\varepsilon_D, 1)$-hashmap. However, there is no guarantee that $\varepsilon_D \leq \varepsilon_R$. R2O4; R1O5

In this section, we design a practical algorithm that returns an $(\varepsilon, 1)$-hashmap with $\varepsilon \leq \varepsilon_R$ allowing a slight increase in single fairness (and collision probability). In practice, as we see in Section 6, it holds that $\varepsilon \ll \varepsilon_R$. The new algorithm is a local-search based algorithm and works as a post-processing procedure to any ranking-based algorithm (for example Algorithm 1). The intuition is the same to other discrepancy-based algorithms: The fact that we use the same number of items per bucket, restricts our options to compute a fair hashmap. Given the buckets computed by a ranking-based algorithm, we try to (slightly) modify the boundaries of the buckets to compute a new hasmap with smaller unfairness.

The high level idea is that in each iteration of the algorithm we slightly move one of the boundaries that improves the unfairness the most, maintaining a sufficient single fairness and collision probability. Let T be the maximum number of iterations we execute our algorithm, and let f^-, f^+, c^+ be the minimum single fairness, the maximum single fairness, and the maximum collision probability, respectively, that the returned hashmap should satisfy. Let B be the boundaries returned by Algorithm 1. For an iteration $i \leq T$, for every boundary $B_j \in B$ we move B_j one position to the left or to the right. For each movement of the boundary B_j, we compute the unfairness ε_j, single fairness f_j, and collision probability c_j of the new partition. If $f^- \leq f_j \leq f^+$ and $c_j \leq c^+$ then this is a valid partition/hashmap satisfying the requested single fairness and collision probability. In the end of each iteration, we modify the boundary B_{j^*} that leads to a valid hashmap with the smallest unfairness, i.e., $j^* = \arg\min_{j: f^- \leq f_j \leq f^+, c_j \leq c^+} \varepsilon_j$.

By definition, in each iteration, we find a partition having at most the same unfairness as before. In practice, we expect to find a hashmap with much smaller unfairness. This is justified in our experiments, Figure 32. For the running time, the algorithm runs in T iterations. In each iteration, we go through all the $O(m)$ boundaries, and we compute ε_j, f_j, c_j. Using a max heap to maintain the

unfairness for every group $\mathbf{g}_i$ (similar to the max-heap we used in the previous dynamic programming algorithm) we can compute the unfairness ε_j in $O(\log k)$ time. We need the same time to compute f_j and c_j. Our algorithm runs in $O(n + Tm \log k)$.

Due to space limitations, we show the cut-based algorithm that finds a $(\varepsilon, k(4 + \log \frac{1}{\varepsilon}))$-hashmap in technical report [2].

Table 4: Overview of datasets

Dataset	Size	Sensitive Attrs.	No. Attr.
Adult [23]	~49K	sex	15
Compas [1]	~61K	sex, race	29
Diabetes [92]	~102K	sex	49
ChicagoPop [74]	1M	race	5

6 EXPERIMENTS

In addition to the theoretical analysis, we conduct extensive experiments on a variety of settings to confirm the fairness and memory/time efficiency of our proposed algorithms. In short, aligned with the theoretical guarantees shown in the previous sections, the results of our experiments demonstrate the effectiveness and efficiency of our algorithms in real-world settings.

6.1 Experiments Setup

The experiments were conducted on a 3.5 GHz Intel Core i9 processor, 128 GB memory, running Ubuntu. The algorithms were implemented in Python 3.

For evaluation purposes, we used three real-world and one semi-synthetic datasets to evaluate our algorithms. With the importance of the scalability of our proposed methods to large settings in mind, we chose datasets that are large enough to represent real-world applications. For each dataset, we selected the two columns that were most uncorrelated to construct the hashmaps. The values in either column are normalized to be in the $[0, 1]$ range. As the sensitive attribute, we follow the existing literature on group fairness and study fairness over demographic information such as *sex* and *race*. A summary of the datasets is presented in Table 4. For a more detailed description of the datasets, refer to the technical report [2].

6.2 Evaluation Plan

We evaluate our proposed algorithms based on three metrics: 1) unfairness, 2) space, and 3) efficiency (preprocessing time and query time). For each of the above metrics, we study the effect of varying three variables: dataset size n, minority-to-majority ratio, and number of buckets m. In our experiments, we vary n from 0.2 to 1.0 fraction of the original dataset with an increasing step of 0.2. We vary m from 100 to 1000 increasing by 100 at each step and finally, the minority-to-majority ratio from 0.25 to 1.0 increasing by 0.25 at each step. Throughout our experiments, while varying a variable, we fix the others as follows: dataset size $n = 0.2 \times |\text{original dataset}|$; number of buckets $m = 100$; minority-to-majority ratio $= 0.25$. Due to the space limitations, we present the results for two datasets for each setting and present the extended results in the technical report [2]. Particularly, for fairness and space evaluation, we report the results for Compas and Adult, the fairness benchmark datasets. For run-time evaluation though, we report on the larger-scale datasets, Diabetes and ChicagoPop. We confirm that we obtained similar results for all datasets.

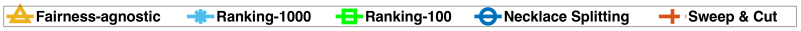
6.2.1 Evaluated Algorithms. In our experiments, we evaluate Ranking, Necklace$_{2g}$, Sweep&Cut algorithms, and CDF-based hashmap [63] (referred as Fairness-agnostic) as the baseline, for all of the datasets using the binary sensitive attributes (sex or binary race). We also evaluate the Ranking and Sweep&Cut algorithms using Compas dataset with race attribute to demonstrate that our algorithms extend to non-binary sensitive attributes. Given the potentially large number of rankings, we use the sampled vectors approach for Ranking. Particularly, we report the results based on two samples of vectors of size 100 and 1000. Finally, we evaluate the effectiveness of our local-search based heuristic on the output the Ranking algorithm.

6.3 Unfairness Evaluations

We start our experiments by evaluating our algorithms for unfairness. Recall that Ranking returns an ε_R-unfair hashmap while Sweep&Cut and Necklace$_{2g}$ output 0-unfair hashmaps, i.e., $\varepsilon = 0$. In the first experiment, we study the effect of varying dataset size n on the unfairness. As shown in Figures 3 and 4, irrespective of the dataset size, Sweep&Cut and Necklace$_{2g}$ always exhibit zero unfairness. On the other hand, Ranking, while improving compared to Fairness-agnostic, still shows a small degree of unfairness that, similar to the baseline, decreases as the size of the dataset grows. R1M2 We also studied the impact of increasing the number of sampled vectors in Figures 29 and 30 and noticed a consistent decrease in the unfairness with the increasing number of sampled vectors. Next, we study the effect of the minority-to-majority ratio on unfairness. The results are illustrated in Figures 5 and 6 with Sweep&Cut and Necklace$_{2g}$ showing no unfairness, while Ranking reducing the unfairness compared to Fairness-agnostic. It is worth mentioning that all unfairness values approach to zero when the dataset includes an equal number of records from each group. This further accentuates the role of unequal base rate [61] in unfairness. Last but not least, we evaluate the effect of increasing the number of buckets m on unfairness (Figures 7 and 8). Sweep&Cut and Necklace$_{2g}$ are independent of the number of buckets and show zero unfairness as m increases. Fairness-agnostic and Ranking unfairness values however increase in a linear fashion as m grows. Consistent with the two previous experiments, we observe that Ranking methods moderately improve the unfairness. Overall, confirming our theoretical analysis, Sweep&Cut and Necklace$_{2g}$ are preferred from the fairness perspective.

6.4 Space Evaluations

Next, we evaluate our algorithms for memory demands a.k.a. space. Recall that Ranking is a 1-memory hashmaps, meaning that no additional memory is required and the number of boundary points is exactly $m - 1$. Necklace$_{2g}$ guarantees the number of cuts to be at most $2(m - 1)$ while Sweep&Cut can create up to $O(n)$ cuts in the worst-case. We investigated the effect of varying dataset size n (Figures 9 and 10), minority-to-majority ratio (Figures 11 and 12), and number of buckets (Figures 13 and 14) on the required space for each algorithm. As expected the results are consistent with the theoretical bounds. The number of boundaries created by Ranking is independent of the dataset size n and minority-to-majority ratio and only depends on the number of buckets m and therefore it is always a constant $(m-1)$. Our experiments also verify similar results for Necklace$_{2g}$ being independent of n and minority-to-majority



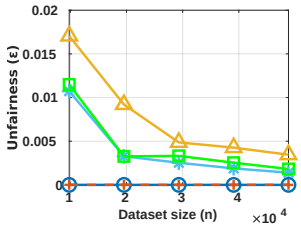


Figure 3: Effect of varying dataset size n on unfairness, Adult, sex

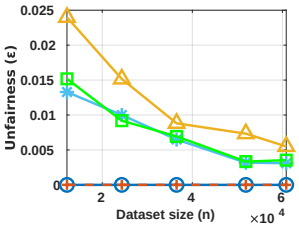


Figure 4: Effect of varying dataset size n on unfairness, Compas, sex

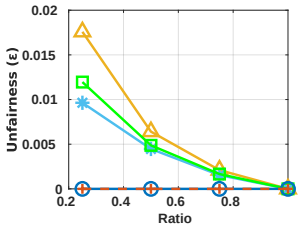


Figure 5: Effect of varying minority-to-majority ratio on unfairness, Adult, sex

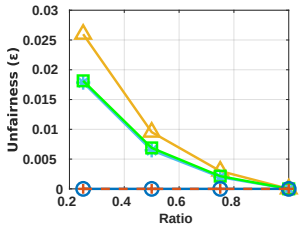


Figure 6: Effect of varying minority-to-majority ratio on unfairness, Compas, sex

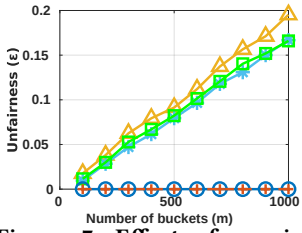


Figure 7: Effect of varying number of buckets m on unfairness, Adult, sex

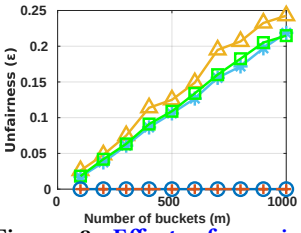


Figure 8: Effect of varying number of buckets m on unfairness, Compas, sex

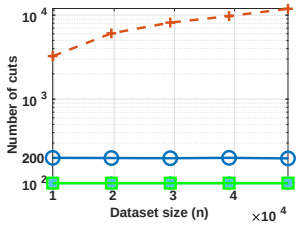


Figure 9: Effect of varying dataset size n on space, Adult

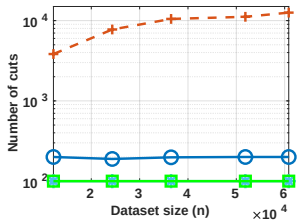


Figure 10: Effect of varying dataset size n on space, Compas

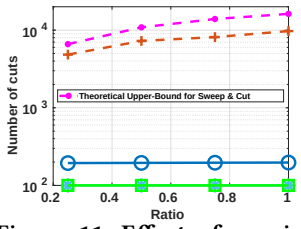


Figure 11: Effect of varying minority-to-majority ratio on space, Adult

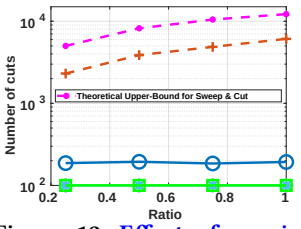


Figure 12: Effect of varying minority-to-majority ratio on space, Compas

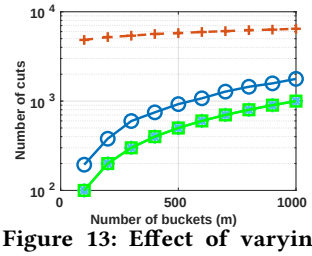


Figure 13: Effect of varying number of buckets m on space, Adult

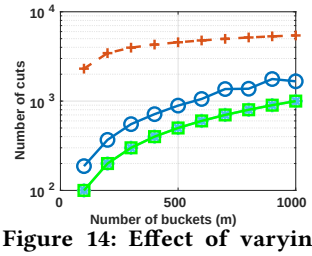


Figure 14: Effect of varying number of buckets m on space, Compas

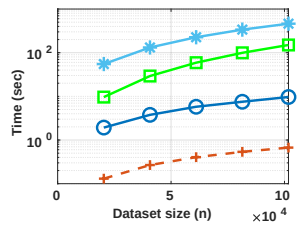


Figure 15: Effect of varying dataset size n on preprocessing time, Diabetes

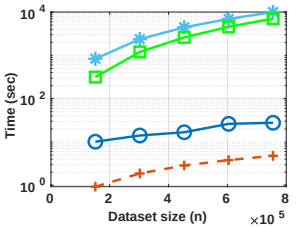


Figure 16: Effect of varying dataset size n on preprocessing time, ChicagoPop

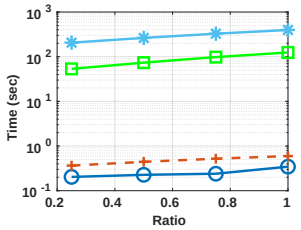


Figure 17: Effect of varying minority-to-majority ratio on preprocessing time, Diabetes

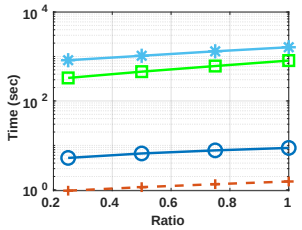


Figure 18: Effect of minority-to-majority ratio on preprocessing time, ChicagoPop

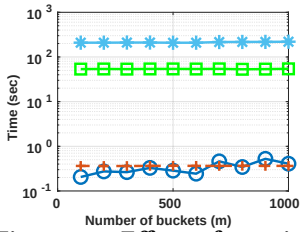


Figure 19: Effect of varying number of buckets m on preprocessing time, Diabetes

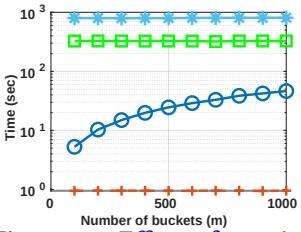


Figure 20: Effect of varying number of buckets m on preprocessing time, ChicagoPop

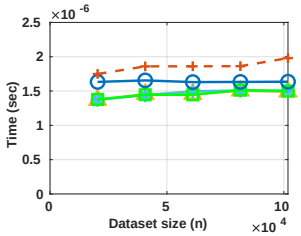


Figure 21: Effect of varying dataset size n on query time, Diabetes

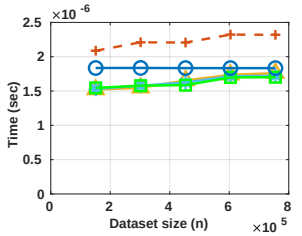
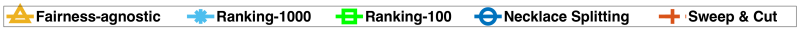


Figure 22: Effect of varying dataset size n on query time, ChicagoPop



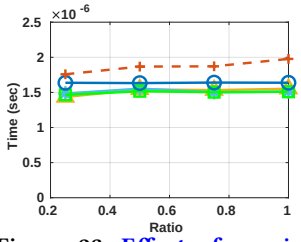


Figure 23: Effect of varying minority-to-majority on query time, DIABETES

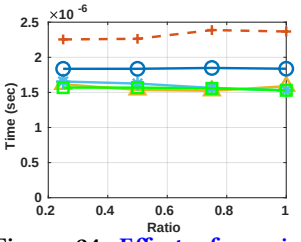


Figure 24: Effect of varying minority-to-majority on query time, CHICAGOPOP

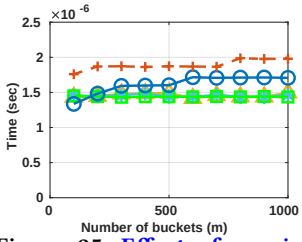


Figure 25: Effect of varying number of buckets m on query time, DIABETES

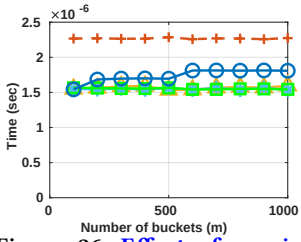


Figure 26: Effect of varying number of buckets m on query time, CHICAGOPOP

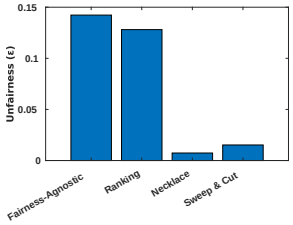


Figure 27: Learning setting: Unfairness evaluation over held out data, ADULT

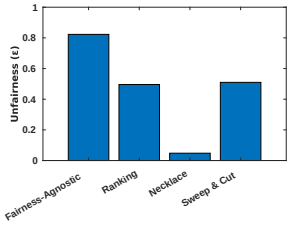


Figure 28: Learning setting: Unfairness evaluation over held out data, CHICAGOPOP

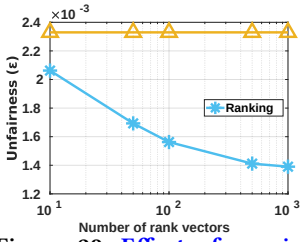


Figure 29: Effect of varying number of sampled vectors on unfairness, ADULT

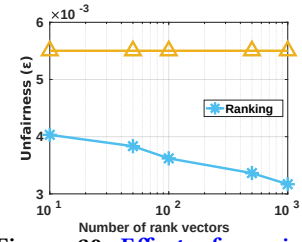


Figure 30: Effect of varying number of sampled vectors on unfairness, COMPAS, sex

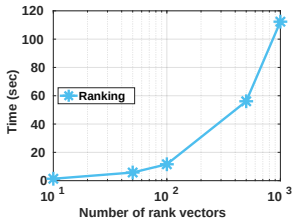


Figure 31: Effect of varying number of sampled vectors on preprocessing time, ADULT

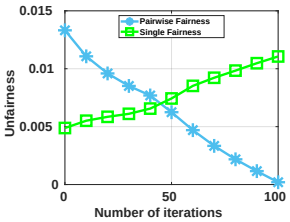


Figure 32: Pairwise unfairness reduction using the local-search heuristic, ADULT

ratio. Interestingly, in almost all settings, the actual number of cuts created by NECKLACE$_{2g}$ is close to the upper-bound $2(m-1)$. The results for SWEEP&CUT however, verify that the major drawback of this algorithm is the memory demands with close to the worst case of $O(n)$ cuts. Lemma 2 provides an upper-bound on the expected number of cuts as a function of the minority ratio in the data set. To evaluate how tight this upper-bound is in practice, in Figures 11 and 12, we also present the actual number of cuts while varying the minority-to-majority ratios. At least in this experiment, the upper-bound was tight as it was always less than 30% larger than the actual number. In general, if an application requires maintaining the space at a minimum while satisfying fairness constraints, as empirically observed, RANKING is the leading alternative. NECKLACE$_{2g}$ is also a favorable choice as it provides a practical trade-off between fairness (0-unfair) and space ($\leq 2(m-1)$ cuts).

6.5 Efficiency Evaluations

In this set of experiments, we evaluate our proposed algorithms for efficiency. More specifically, we measure efficiency from two perspectives: 1) the preprocessing time that is required to construct the fair hashmap, and 2) the query time needed to return a hash (bucket) for new records when the hashmap is constructed.

6.5.1 Preprocessing Time. We start our efficiency experiments by revisiting the preprocessing time complexity of the proposed algorithms. RANKING has a time complexity of $O(n^2 \log n)$ in 2D while SWEEP&CUT and NECKLACE$_{2g}$ both run in $O(n \log n)$. In our first experiment, we study the impact of varying the dataset size n. First, in Figures 15 and 16, one can confirm that, overall, the run-time increases with the dataset size. For RANKING, the exact time depends on the number of rankings generated. This is also evident in Figures 31, where we study the impact of increasing the number of sampled vectors on the preprocessing time. Next, as demonstrated in Figures 17 and 18, we confirm that the preprocessing time is independent of the minority-to-majority ratio. As with the preceding experiment, we expect that varying m should not impact the run-time of any of the algorithms, which is consistent with our experiment results in Figures 19 and 20. Overall, in time-sensitive applications, both SWEEP&CUT and NECKLACE$_{2g}$ offer the fastest results, all the while ensuring 0-unfairness. R1M2

6.5.2 Query Time. Recall that the output of our algorithms consists of a sequence of boundaries along with a corresponding set of hash buckets. After constructing the hashmap, obtaining the hash bucket for a new query is a simple process: just execute a binary search on the boundaries and retrieve the bucket linked to the boundaries within which the query point resides. Therefore, query time is in $O(\log |B|)$ and only depends on the number of boundaries. Our empirical results are consistent with the preceding analysis, confirming that query time remains independent of both dataset size (see Figures 21 and 22) and the minority-to-majority ratio (refer to Figures 23 and 24). The increase in the number of buckets (m) leads to a logarithmic growth in query time across all our algorithms, as depicted in Figures 25 and 26. Thanks to the logarithmic efficiency of binary search, all our algorithms offer remarkably fast query times, with with practically negligible variations.

R2O4; R1O5

6.6 Local-search-based Heuristic Evaluation

In this experiment, we apply the local-search-based heuristic to the boundaries generated by the Ranking algorithm on an instance of Adult dataset. We run the heuristic in 1000 iterations, with single fairness lower and upper bounds as $f^- = 0, f^+ = 0.05$ respectively and the collision probability upper bound as $c^+ = 0.05$. The results are shown in Figure 32. The local-search-based heuristic effectively enhances pairwise fairness by making minimal adjustments to the bin boundaries, incurring a slight cost in single fairness within 100 iterations before it concludes.

6.7 Learning Settings Evaluation

So far in our experiments, we assumed that the algorithms have access to the entire input set. In this experiment, we demonstrate that our methods can work in expectation if an unbiased sample set from the input set is provided. To do so, we partition the input datasets into training and test sets with a ratio of 0.8 to 0.2. Next, we utilize our algorithms to create a hashmap on the training set. We then use the test set for evaluation: each test entry is queried on the constructed hashmap to identify their buckets. Having created a hashmap that exclusively contains the test entries, we proceed to measure the pairwise unfairness. The results are illustrated in Figures 27 and 28. Although all methods demonstrate an enhancement in unfairness compared to the Fairness-agnostic baseline, the most notable improvement is observed with Necklace$_{2g}$, where the unfairness decreases from 0.81 to 0.03 for the ChicagoPop and from 0.15 to 0.007 for Adult. Aligned with our previous results, Ranking only moderately improves the unfairness. Although Sweep&Cut consistently improves the unfairness in the learned settings, depending on the number of cuts it generates, it may exhibit signs of overfitting based on the number of cuts it generates. This tendency becomes more apparent, especially when dealing with large training data, as illustrated in Figure 28. However, this overfitting phenomenon is mitigated when the size of training data is smaller and the number of cuts is reduced, resulting in a substantial decrease in unfairness, as depicted in Figure 27.

7 RELATED WORK

Hashing: Hashing has a long history in computer science [37, 94]. Hashing-based algorithms and data structures find many applications in various areas such as theory, machine learning, computer graphics, computational geometry and databases [5, 34, 36, 42, 47, 62, 68]. Due to its numerous applications, the design of efficient hash functions with theoretical guarantees are of significant importance [3, 4, 14, 24, 26, 43, 44, 59, 64, 95]. In traditional hashmaps, the goal is to design a hash function that maps a key to a random value in a specified output range. The goal is to minimize the number of collisions, where a collision occurs when multiple keys get mapped to the same output value. There are several well-known schemes such as chaining, probing, and cuckoo hashing to handle collisions. Recently, machine learning is used to learn a proper hash function [63, 71, 81]. In a typical scenario, a set of samples is received and they learn the CDF of the underlying data distribution. Then the hashmap is created by partitioning the range into equal-sized buckets. It has been shown that such learned index structures [63, 81], can outperform traditional hashmaps on practical workloads. However, to the best of our knowledge, none of these hashmap schemes can handle fair hashing with theoretical guarantees. Finally, learning has been used to obtain other data structures as well, such as B-trees [63] or bloom filters [63, 71, 93].

Algorithmic Fairness: Fairness in data-driven systems has been studied by various research communities but mostly in the context of machine learning (ML) [21, 69, 79]. Most of the existing work is on training a ML model that satisfies some fairness constraints. Some of the pioneering fair-ML efforts include [32, 33, 49, 51, 54, 60, 96]. Biases in data has also been studied extensively [76, 82, 86, 91] to ensure data has been prepared responsibly [72, 83, 84]. Recent studies of fair algorithm design include fair clustering [6, 25, 29, 38, 39, 57, 67, 85], fairness in resource allocation and facility location problem [27, 45, 53, 55, 58, 100], min cut [65], max cover [15], game theoretic approaches [8, 46, 99], hiring [12, 80], ranking [16, 89, 90, 97], recommendation [35, 66], representation learning [56], etc.

Fairness in data structures is significantly under studied, with the existing work being limited to [17–19], which study *individual fairness* in near-neighbor search. Particularly, a rejection sampling technique has been added on top of local sensitive hashing (LSH) that equalizes the retrieval chance for all points in the ρ-vicinity of a query point, independent of how close those are to the query point. To the best of our knowledge, we are the first to study *group fairness* in hashing and more generally in data structure design.

8 FINAL REMARKS AND FUTURE WORK

In this paper we studied hashmaps through the lens of fairness and proposed several fair and memory/time efficient algorithms. Some the interesting directions for future work are as following.

Memory-efficient 0-unfair hashmaps for more than two groups: Our ranking-based algorithms do not depend on the number of groups, however, those are not 0-unfair. The Sweep&Cut algorithm also does not depend on the number of groups and it is even 0-unfair. Its memory requirement, however, can be as high as $O(n)$. The performance of the necklace splitting algorithm, on the other hand, depends on the number of groups. While for two groups, the number of boundaries is independent from n and at most $2(m-1)$, the state-of-the-art algorithm for more than two groups suddenly increases this requirement by a factor of $O(\log(n))$. Developing a fair $(0, c)$-hashmap for this case, for a constant value of c, remains an interesting open problem for future work.

Beyond the class of Linear Ranking functions: We developed our ranking-based algorithms using the class of linear ranking functions. It would be interesting to expand the scope to more general non-linear classes (such as monotonic functions). Indeed one can add non-linear attributes before running our ranking-based algorithms. But this will increase the number of dimensions, exponentially reducing its run time.

Lower-bounds and trade-offs on (ε, α): Our ranking-based algorithms satisfy 1-memory requirement but cannot achieve 0-unfairness. The cut-based algorithms, on the other hand, are 0-unfair but require additional memory. This suggests a trade-off between fairness and memory requirements. Last but not least, formally studying this trade-off and identifying the lower-bound Pareto frontier for fairness and memory is an interesting future work.

REFERENCES

[1] 2015. COMPAS Recidivism Risk Score Data and Analysis. www.propublica.org/datastore/dataset/compas-recidivism-risk-score-data-and-analysis.

[2] 2023. Technical Report details omitted for double-blind review. Pleaes refer to "appendix.pdf" in the code repository for the review.

[3] Anders Aamand, Jakob Bæk Tejs Knudsen, Mathias Bæk Tejs Knudsen, Peter Michael Reichstein Rasmussen, and Mikkel Thorup. 2020. Fast hashing with strong concentration bounds. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 1265–1278.

[4] Anders Aamand and Mikkel Thorup. 2019. Non-empty bins with simple tabulation hashing. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2498–2512.

[5] Thomas D Ahle and Jakob BT Knudsen. 2020. Subsets and supermajorities: Optimal hashing-based set similarity search. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 728–739.

[6] Sara Ahmadian, Alessandro Epasto, Marina Knittel, Ravi Kumar, Mohammad Mahdian, Benjamin Moseley, Philip Pham, Sergei Vassilvitskii, and Yuyan Wang. 2020. Fair hierarchical clustering. *Advances in Neural Information Processing Systems* 33 (2020), 21050–21060.

[7] Saif Al-Kuwari, James H Davenport, and Russell J Bradford. 2011. Cryptographic hash functions: Recent design trends and security notions. *Cryptology ePrint Archive* (2011).

[8] Encarnación Algaba, Vito Fragnelli, and Joaquín Sánchez-Soriano. 2019. The Shapley value, a paradigm of fairness. *Handbook of the Shapley value* (2019), 17–29.

[9] Noga Alon. 1987. Splitting necklaces. *Advances in Mathematics* 63, 3 (1987), 247–253.

[10] Noga Alon and Andrei Graur. 2021. Efficient splitting of necklaces. In *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[11] Noga Alon and Douglas B West. 1986. The Borsuk-Ulam theorem and bisection of necklaces. *Proc. Amer. Math. Soc.* 98, 4 (1986), 623–628.

[12] Mohammad Reza Aminian, Vahideh Manshadi, and Rad Niazadeh. 2023. Fair markovian search. *Available at SSRN 4347447* (2023).

[13] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. 2022. Machine bias. In *Ethics of data and analytics*. Auerbach Publications, 254–264.

[14] Sepehr Assadi, Martin Farach-Colton, and William Kuszmaul. 2023. Tight bounds for monotone minimal perfect hashing. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 456–476.

[15] Abolfazl Asudeh, Tanya Berger-Wolf, Bhaskar DasGupta, and Anastasios Sidiropoulos. 2023. Maximizing coverage while ensuring fairness: A tale of conflicting objectives. *Algorithmica* 85, 5 (2023), 1287–1331.

[16] Abolfazl Asudeh, HV Jagadish, Julia Stoyanovich, and Gautam Das. 2019. Designing fair ranking schemes. In *Proceedings of the 2019 international conference on management of data*. 1259–1276.

[17] Martin Aumuller, Sariel Har-Peled, Sepideh Mahabadi, Rasmus Pagh, and Francesco Silvestri. 2021. Fair near neighbor search via sampling. *ACM SIGMOD Record* 50, 1 (2021), 42–49.

[18] Martin Aumüller, Sariel Har-Peled, Sepideh Mahabadi, Rasmus Pagh, and Francesco Silvestri. 2022. Sampling a Near Neighbor in High Dimensions—Who is the Fairest of Them All? *ACM Transactions on Database Systems (TODS)* 47, 1 (2022), 1–40.

[19] Martin Aumüller, Rasmus Pagh, and Francesco Silvestri. 2020. Fair near neighbor search: Independent range sampling in high dimensions. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*. 191–204.

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

[21] Solon Barocas, Moritz Hardt, and Arvind Narayanan. 2017. Fairness in machine learning. *Nips tutorial* 1 (2017), 2017.

[22] Solon Barocas, Moritz Hardt, and Arvind Narayanan. 2023. *Fairness and machine learning: Limitations and opportunities*. MIT Press.

[23] Barry Becker and Ronny Kohavi. 1996. Adult. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5XW20.

[24] Michael A Bender, Martín Farach-Colton, John Kuszmaul, William Kuszmaul, and Mingmou Liu. 2022. On the optimal time/space tradeoff for hash tables. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*. 1284–1297.

[25] Suman Bera, Deeparnab Chakrabarty, Nicolas Flores, and Maryam Negahbani. 2019. Fair algorithms for clustering. *Advances in Neural Information Processing Systems* 32 (2019).

[26] Ioana Oriana Bercea and Guy Even. 2022. An extendable data structure for incremental stable perfect hashing. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*. 1298–1310.

[27] Víctor Blanco and Ricardo Gázquez. 2023. Fairness in maximal covering location problems. *Computers & Operations Research* 157 (2023), 106287.

[28] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[29] Matteo Böhm, Adriano Fazzone, Stefano Leonardi, and Chris Schwiegelshohn. 2020. Fair clustering with multiple colors. *arXiv preprint arXiv:2002.07892* (2020).

[30] Andrei Z Broder. 1997. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE, 21–29.

[31] Jehoshua Bruck, Jie Gao, and Anxiao Jiang. 2006. Weighted bloom filter. In *2006 IEEE International Symposium on Information Theory*. IEEE, 2304–2308.

[32] Flavio Calmon, Dennis Wei, Bhanukiran Vinzamuri, Karthikeyan Natesan Ramamurthy, and Kush R Varshney. 2017. Optimized pre-processing for discrimination prevention. *Advances in neural information processing systems* 30 (2017).

[33] L Elisa Celis, Lingxiao Huang, Vijay Keswani, and Nisheeth K Vishnoi. 2019. Classification with fairness constraints: A meta-algorithm with provable guarantees. In *Proceedings of the conference on fairness, accountability, and transparency*. 319–328.

[34] Moses Charikar and Paris Siminelakis. 2019. Multi-resolution hashing for fast pairwise summations. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 769–792.

[35] Jiawei Chen, Hande Dong, Xiang Wang, Fuli Feng, Meng Wang, and Xiangnan He. 2023. Bias and debias in recommender system: A survey and future directions. *ACM Transactions on Information Systems* 41, 3 (2023), 1–39.

[36] Lijie Chen, Ce Jin, R Ryan Williams, and Hongxun Wu. 2022. Truly Low-Space Element Distinctness and Subset Sum via Pseudorandom Hash Functions. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 1661–1678.

[37] Lianhua Chi and Xingquan Zhu. 2017. Hashing techniques: A survey and taxonomy. *ACM Computing Surveys (Csur)* 50, 1 (2017), 1–36.

[38] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, and Sergei Vassilvitskii. 2017. Fair clustering through fairlets. *Advances in neural information processing systems* 30 (2017).

[39] Eden Chlamtáč, Yury Makarychev, and Ali Vakilian. 2022. Approximating fair clustering with cascaded norm objectives. In *Proceedings of the 2022 annual ACM-SIAM symposium on discrete algorithms (SODA)*. SIAM, 2664–2683.

[40] Ondrej Chum, Michal Perd'och, and Jiri Matas. 2009. Geometric min-hashing: Finding a (thick) needle in a haystack. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 17–24.

[41] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[42] Artur Czumaj, Shaofeng H-C Jiang, Robert Krauthgamer, Pavel Veselỳ, and Mingwei Yang. 2022. Streaming facility location in high dimension via geometric hashing. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 450–461.

[43] Ivan Bjerre Damgård. 1989. A design principle for hash functions. In *Conference on the Theory and Application of Cryptology*. Springer, 416–427.

[44] Manik Dhar and Zeev Dvir. 2022. Linear Hashing with ℓ_∞ guarantees and two-sided Kakeya bounds. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 419–428.

[45] Kate Donahue and Jon Kleinberg. 2020. Fairness and utilization in allocating resources with uncertain demand. In *Proceedings of the 2020 conference on fairness, accountability, and transparency*. 658–668.

[46] Kate Donahue and Jon Kleinberg. 2023. Fairness in model-sharing games. In *Proceedings of the ACM Web Conference 2023*. 3775–3783.

[47] Ran Duan, Hongxun Wu, and Renfei Zhou. 2022. Faster matrix multiplication via asymmetric hashing. *arXiv preprint arXiv:2210.10173* (2022).

[48] Marianne Durand and Philippe Flajolet. 2003. Loglog counting of large cardinalities. In *Algorithms-ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings 11*. Springer, 605–617.

[49] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. 2012. Fairness through awareness. In *Proceedings of the 3rd innovations in theoretical computer science conference*. ACM, 214–226.

[50] Herbert Edelsbrunner. 1987. *Algorithms in combinatorial geometry*. Vol. 10. Springer Science & Business Media.

[51] Michael Feldman, Sorelle A Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and removing disparate impact. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 259–268.

[52] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete mathematics & theoretical computer science* Proceedings (2007).

[53] Pranay Gorantla, Kunal Marwaha, and Santhoshini Velusamy. 2023. Fair allocation of a multiset of indivisible items. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 304–331.

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

[55] Yuzi He, Keith Burghardt, Siyi Guo, and Kristina Lerman. 2020. Inherent trade-offs in the fair allocation of treatments. *arXiv preprint arXiv:2010.16409* (2020).

[56] Yuzi He, Keith Burghardt, and Kristina Lerman. 2020. A geometric solution to fair representations. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*. 279–285.

[57] Sedjro Salomon Hotegni, Sepideh Mahabadi, and Ali Vakilian. 2023. Approximation Algorithms for Fair Range Clustering. In *International Conference on Machine Learning*. PMLR, 13270–13284.

[58] Yanmin Jiang, Xiaole Wu, Bo Chen, and Qiying Hu. 2021. Rawlsian fairness in push and pull supply chains. *European Journal of Operational Research* 291, 1 (2021), 194–205.

[59] Praneeth Kacham, Rasmus Pagh, Mikkel Thorup, and David P Woodruff. 2023. Pseudorandom Hashing for Space-bounded Computation with Applications in Streaming. *arXiv preprint arXiv:2304.06853* (2023).

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

[61] Jon Kleinberg, Sendhil Mullainathan, and Manish Raghavan. 2017. Inherent trade-offs in the fair determination of risk scores. *Proceedings of Innovations in Theoretical Computer Science (ITCS)* (2017).

[62] Donald Ervin Knuth. 1997. *The art of computer programming*. Vol. 3. Pearson Education.

[63] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[64] William Kuszmaul. 2022. A hash table without hash functions, and how to get the most out of your random bits. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 991–1001.

[65] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, and Thatchaphol Saranurak. 2023. Near-Linear Time Approximations for Cut Problems via Fair Cuts. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 240–275.

[66] Yunqi Li, Hanxiong Chen, Shuyuan Xu, Yingqiang Ge, Juntao Tan, Shuchang Liu, and Yongfeng Zhang. 2022. Fairness in recommendation: A survey. *arXiv preprint arXiv:2205.13619* (2022).

[67] Yury Makarychev and Ali Vakilian. 2021. Approximation algorithms for socially fair clustering. In *Conference on Learning Theory*. PMLR, 3246–3264.

[68] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *Proceedings of the VLDB Endowment* 14, 1 (2020), 1–13.

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

[70] Frédéric Meunier. 2014. Simplotopal maps and necklace splitting. *Discrete Mathematics* 323 (2014), 14–26.

[71] Michael Mitzenmacher. 2018. A model for learned bloom filters and optimizing by sandwiching. *Advances in Neural Information Processing Systems* 31 (2018).

[72] Fatemeh Nargesian, Abolfazl Asudeh, and HV Jagadish. 2021. Tailoring data source distributions for fairness-aware data integration. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2519–2532.

[73] Fatemeh Nargesian, Abolfazl Asudeh, and HV Jagadish. 2022. Responsible Data Integration: Next-generation Challenges. In *Proceedings of the 2022 International Conference on Management of Data*. 2458–2464.

[74] Khanh Duy Nguyen, Nima Shahbazi, and Abolfazl Asudeh. 2023. PopSim: An Individual-level Population Simulator for Equitable Allocation of City Resources. *SDM Workshop on Algorithmic Fairness in Artificial intelligence, Machine learning, and Decision making* (2023).

[75] Eirini Ntoutsi, Pavlos Fafalios, Ujwal Gadiraju, Vasileios Iosifidis, Wolfgang Nejdl, Maria-Esther Vidal, Salvatore Ruggieri, Franco Turini, Symeon Papadopoulos, Emmanouil Krasanakis, et al. 2020. Bias in data-driven artificial intelligence systems—An introductory survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 10, 3 (2020), e1356.

[76] Alexandra Olteanu, Carlos Castillo, Fernando Diaz, and Emre Kıcıman. 2019. Social data: Biases, methodological pitfalls, and ethical boundaries. *Frontiers in big data* 2 (2019), 13.

[77] Anna Ostlin and Rasmus Pagh. 2003. Uniform hashing in constant time and linear space. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. 622–628.

[78] Anna Pagh, Rasmus Pagh, and S Srinivasa Rao. 2005. An optimal Bloom filter replacement.. In *Soda*, Vol. 5. Citeseer, 823–829.

[79] Dana Pessach and Erez Shmueli. 2022. A review on fairness in machine learning. *ACM Computing Surveys (CSUR)* 55, 3 (2022), 1–44.

[80] Manish Raghavan, Solon Barocas, Jon Kleinberg, and Karen Levy. 2020. Mitigating bias in algorithmic hiring: Evaluating claims and practices. In *Proceedings of the 2020 conference on fairness, accountability, and transparency*. 469–481.

[81] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. 2022. Can Learned Models Replace Hash Functions? *Proceedings of the VLDB Endowment* 16, 3 (2022), 532–545.

[82] Babak Salimi, Bill Howe, and Dan Suciu. 2019. Data management for causal algorithmic fairness. *arXiv preprint arXiv:1908.07924* (2019).

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

[84] Babak Salimi, Luke Rodriguez, Bill Howe, and Dan Suciu. 2019. Interventional fairness: Causal database repair for algorithmic fairness. In *Proceedings of the 2019 International Conference on Management of Data*. 793–810.

[85] Melanie Schmidt, Chris Schwiegelshohn, and Christian Sohler. 2020. Fair coresets and streaming algorithms for fair k-means. In *Approximation and Online Algorithms: 17th International Workshop, WAOA 2019, Munich, Germany, September 12–13, 2019, Revised Selected Papers 17*. Springer, 232–251.

[86] Nima Shahbazi, Yin Lin, Abolfazl Asudeh, and HV Jagadish. 2023. Representation Bias in Data: A Survey on Identification and Resolution Techniques. *Comput. Surveys* (2023).

[87] Alan Siegel. 1989. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *30th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 20–25.

[88] John Edward Silva. 2003. An overview of cryptographic hash functions and their uses. *GIAC* 6 (2003).

[89] Ashudeep Singh and Thorsten Joachims. 2018. Fairness of exposure in rankings. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 2219–2228.

[90] Ashudeep Singh and Thorsten Joachims. 2019. Policy learning for fairness in ranking. *Advances in neural information processing systems* 32 (2019).

[91] Julia Stoyanovich, Serge Abiteboul, Bill Howe, HV Jagadish, and Sebastian Schelter. 2022. Responsible data management. *Commun. ACM* 65, 6 (2022), 64–74.

[92] Beata Strack, Jonathan P DeShazo, Chris Gennings, Juan L Olmo, Sebastian Ventura, Krzysztof J Cios, John N Clore, et al. 2014. Impact of HbA1c measurement on hospital readmission rates: analysis of 70,000 clinical database patient records. *BioMed research international* 2014 (2014).

[93] Kapil Vaidya, Eric Knorr, Michael Mitzenmacher, and Tim Kraska. 2020. Partitioned Learned Bloom Filters. In *International Conference on Learning Representations*.

[94] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. 2014. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927* (2014).

[95] Chaoping Xing and Chen Yuan. 2023. Beating the Probabilistic Lower Bound on q-Perfect Hashing. *Combinatorica* (2023), 1–20.

[96] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rodriguez, and Krishna P Gummadi. 2017. Fairness beyond disparate treatment & disparate impact: Learning classification without disparate mistreatment. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1171–1180.

[97] Meike Zehlike, Ke Yang, and Julia Stoyanovich. 2022. Fairness in ranking, part i: Score-based ranking. *Comput. Surveys* 55, 6 (2022), 1–36.

[98] Qin Zhang. 2022. Technical perspective: Can data structures treat us fairly? *Commun. ACM* 65, 8 (2022), 82–82.

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

[100] Liping Zhou, Na Geng, Zhibin Jiang, and Xiuxian Wang. 2019. Public hospital inpatient room allocation and patient scheduling considering equity. *IEEE Transactions on Automation Science and Engineering* 17, 3 (2019), 1124–1139.