Algorithmic Techniques for Independent Query Sampling

Yufei Tao
taoyf@cse.cuhk.edu.hk
Department of Computer Science and Engineering
Chinese University of Hong Kong
Hong Kong, China

ABSTRACT

Unlike a reporting query that returns all the elements satisfying a predicate, *query sampling* returns only a sample set of those elements and has long been recognized as an important method in database systems. PODS'14 saw the introduction of *independent query sampling* (IQS), which extends traditional query sampling with the requirement that *the sample outputs of all the queries be mutually independent*. The new requirement improves the precision of query estimation, facilitates the execution of randomized algorithms, and enhances the fairness and diversity of query answers. IQS calls for new index structures because conventional indexes are designed to report complete query answers and thus become too expensive for extracting only a few random samples. The phenomenon has created an exciting opportunity to revisit the structure for every reporting query known in computer science. There has been considerable progress since 2014 in this direction. This paper distills the existing solutions into several generic techniques that, when put together, can be utilized to solve a great variety of IQS problems with attractive performance guarantees.

CCS CONCEPTS

• **Theory of computation** → **Data structures design and analysis**;

KEYWORDS

Independent Query Sampling; Data Structures; Algorithms; Theory

ACM Reference Format:
Yufei Tao. 2022. Algorithmic Techniques for Independent Query Sampling. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, 10 pages. https://doi.org/10.1145/3517804.3526068

1 INTRODUCTION

The standard query mechanism in database systems returns all the elements that satisfy a given predicate. It works well when the number of result elements is small. In the big-data era, however, along with the explosion in data volume comes the huge increase in query output size. For example, while a query with selectivity 1% returns only 10000 elements on a dataset of size a million, the number surges to 10^{10} when the data cardinality reaches a trillion (this is roughly at the tera-byte level, which is only moderately "big" by today's metric). In the latter scenario, even just reporting the query answer would take intolerably long, especially when disk I/Os or network communication is involved.

A remedy of the situation is query sampling, an approach dating back to the 1990s. The rationale is to return only a sample set of the query result, rather than the result itself. The usefulness of such samples has been long recognized even in the non-big-data days; for example, Olken's Ph.D. thesis [21], published in 1993, presents a nice exposition on how query sampling benefits database systems. Indeed, in many applications, complete query results are not compulsory and result samples already serve the purposes adequately. Retrieving only the samples can dramatically reduce query response time, a phenomenon that is increasingly conspicuous as the database volume continues to escalate.

In PODS'14, Hu et al. [18] introduced the concept of *independent* query sampling (IQS). The novelty was to guarantee that the result sample returned for a query should be independent of those returned for all previous queries. The guarantee must hold no matter how the predicates of different queries correlate to each other. For a better explanation, let us consider a specific IQS problem. Let S be a set of n elements from the real domain $\mathbb{R}$. Given an interval $q := [x, y]$, a conventional *range query* reports $S_q := q \cap S$. The IQS version, on the other hand, takes two parameters: q (as before) and an integer $s \geq 1$. Like traditional query sampling, the IQS query outputs a set Q of s elements, each independently taken from S_q uniformly at random. In other words, $\mathbf{Pr}[Q = \Sigma]$ is the same for every $\Sigma \in (S_q)^s$.[1]. Unlike traditional query sampling, however, IQS must also fulfill cross-query independence:

$$\mathbf{Pr}[Q = \Sigma \mid \text{outputs of previous IQS queries}] = \frac{1}{|S_q|^s} \quad (1)$$

for every $\Sigma \in (S_q)^s$. It is worth pointing out that even if two IQS queries coincide in q, their samples must still be independent. In fact, one can repeatedly issue the same query to obtain more and more samples of S_q, all of which must be mutually independent.

The notion of IQS can be integrated — in a straightforward manner — with every type of reporting queries known in database research. It brings many benefits that cannot be offered by traditional, *dependent*, query sampling (some benefits will be discussed in Section 2). IQS can take place in various forms. What is illustrated earlier is a variant of IQS under the *with replacement* (WR) sampling scheme. The following are two other major variants.

[1] $(S_q)^s := S_q \times S_q \times ... \times S_q$, where there are $s - 1$ cartesian products.

PODS '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9260-0/22/06…$15.00
https://doi.org/10.1145/3517804.3526068

- *Sampling without replacement (WoR)*: Q should have the same chance to be any of the $\binom{|S_q|}{s}$ size-s subsets of S_q, assuming $s \leq |S_q|$.
- *Weighted sampling*: Each element $e \in S$ carries a positive weight $w(e)$. Q is a set of s elements, each independently sampled from S_q with probabilities proportional to weights, namely, each element $e \in S_q$ is sampled with probability $w(e)/\sum_{e' \in S_q} w(e')$.

In any case, Q must be independent of all the previous queries' outputs.

IQS poses exciting research challenges. A naive solution is to first retrieve the full query result — denoted by S_q following the notation in the above example — and then sample from it, but this defeats the main purpose of query sampling, i.e., reducing computation time. Instead, our objective should be to settle a query in time *significantly less* than $|S_q|$ in the typical situation where $s \ll |S_q|$. Achieving the goal demands organizing the input dataset S in ways drastically different from the existing data structures designed to report S_q in its entirety. The topic, therefore, provides a fresh perspective to revisit every reporting query that has ever been studied in computer science.

There has been considerable progress towards understanding IQS since the concept's inception in 2014. This paper aims to distill the current methods — primarily from papers [2, 3, 6–8, 17, 18, 20, 21, 24, 25, 27] — into generic techniques that allow us to tackle the IQS counterparts of a great variety of reporting queries with attractive performance guarantees. Many of the resulting data structures are friendly to practical implementation and some of them have already been implemented in the system community [27]. In spite of all the development, the research on IQS is far from conclusive, with numerous non-trivial questions still open, some of which will be pointed out at the end of the paper.

2 INDEPENDENCE IS GOOD

Next, we give three benefits of IQS that cannot be provided without enforcing cross-query independence.

It helps for the reader to get a solid idea of what would be a conventional (dependent) query sampling structure. Consider again the range query in Section 1 where the input dataset S contains n values in $\mathbb{R}$. Let us discuss WoR sampling first: given an interval q in $\mathbb{R}$ and a sample size s between 1 and $|S_q|$, a query outputs a random size-s subset Q of $S_q := q \cap S$. In preprocessing, we can randomly permute the elements in S and define the *rank* of each element as its position in the permutation. Given q and s, a query simply returns the set $Q \subseteq S_q$ of s elements having the lowest ranks in S_q. It is clear that Q is a random WoR sample set of S_q. Equally obvious is that the outputs of different queries are correlated; e.g., repeating the query with the same q and s always yields the same Q. The retrieval of Q can be accomplished in $O(\log n + s)$ time [12].[2].

A remark is in order before we proceed. The above approach can be easily adapted for WR sampling: a WoR sample set of size s can be converted to a WR sample set of the same size in $O(s)$ time [19]. The dependence issue persists, nevertheless. The reader may refer to [7, 21, 26] for more (dependent) query sampling structures.

Benefit 1: Query Estimation and Randomized Algorithms. Random sampling is a building brick for many estimation tasks. For example, consider a relation R with a real-valued attribute A and an arbitrary attribute B. Fix a query interval q_A in $\mathbb{R}$ and an arbitrary predicate q_B. Define $R_{q_A} := \{t \in R \mid t.\mathsf{A} \in q_A\}$; note that R_{q_A} is the result of a range query. Suppose that we want to estimate the percentage of tuples $t \in R_{q_A}$ whose t.B values satisfy q_B. It is folklore that, by sampling $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ elements from R_{q_A}, we can estimate the percentage up to an absolute error ϵ with probability at least $1 - \delta$. As $\frac{1}{\epsilon^2} \log_2 \frac{1}{\delta}$ can be far less than $|R_{q_A}|$, retrieving only the samples would be much faster than reporting R_{q_A}.

Whether query sampling ensures independence makes a huge difference on the number of correct estimates in the long run. Assume, for simplicity, that each estimate fails with probability δ. After a period of time, if m estimates have been performed in total, an IQS solution automatically guarantees that the number of erroneous estimates concentrates *sharply* around $m\delta$. In contrast, query sampling without independence can only guarantee the number to be $m\delta$ in expectation, while little can be said about how likely the number would significantly deviate from $m\delta$. Such a difference has a direct impact on the service quality for the underlying system.

The functionality of extracting random samples satisfying a predicate is a prerequisite for many randomized algorithms. The functionality becomes even more essential today as machine learning attracts an unprecedented amount of attention (a learning algorithm trains a model using random samples drawn from a selected portion of a dataset). Just like how index structures facilitate query answering in general, a query sampling structure is crucial for achieving high efficiency for the aforementioned algorithms. Cross-query independence is critical to guaranteeing excellent service quality in the long run, for the reasons explained earlier.

Benefit 2: Fairness. Let S be an input dataset and, given a query predicate q, let S_q be the set of elements in S satisfying q. As mentioned in Section 1, S_q can be excessively large to report in full. In this case, which elements in S_q should be returned? The question has great implications in recommender systems; e.g., among the set S_q of product items meeting a user's inqury, which ones should we return if the user's browser can display only a small number, say s, items?

Query sampling offers a simple and natural answer: s random items.[3] IQS, however, adds another layer of fairness: the random items will be chosen *afresh* for every user inquiry. Several authors [6–8, 17] have picked up this sense of fairness to define an interesting problem called *fair near neighbor search*. There, S is a set of points in d-dimensional space $\mathbb{R}^d$. Let r be a fixed positive value. Given a point q in $\mathbb{R}^d$, an *r-near query* returns $S_q := \{e \in S \mid dist(q, e) \leq r\}$ where $dist(q, e)$ is the distance between points q and e. The fair version — the *fair r-near neighbor query* — returns a

[2]This is an instance of so-called *top-k range reporting*, a systematic coverage about which can be found in [22].

[3]When there is a clear notion of priority, the issue could be remedied by returning the items having the highest priorities. Such a notion, however, does not always exist. Furthermore, even if it does, the issue arises again when items tie in priorities.

uniformly random point from S_q, which must be independent of all the past queries' outputs; note that this is IQS with $s = 1$. Fair near neighbor search has numerous applications in practice as discussed in [7].

Benefit 3: Representativeness (a.k.a. Diversity). There have been considerable interests in studying how to reduce a query's output by returning only a few *representatives* that can illustrate the *diversity* amid the elements satisfying the query (see [23] for an excellent survey). For example, the inquiry "find restaurants in New York" would return hundreds of restaurants. Suppose that we want to return only 10 restaurants that are diverse in multiple criteria simultaneously, e.g., cuisine, location, price, ambiance, etc. If there is a prominent diversity metric, a system can achieve the purpose by maximizing that metric. In the absence of such a metric, a plausible solution is to return 10 random restaurants because intuitively they would spread evenly in the "criteria space". IQS ensures that the 10 random restaurants be picked anew for every query. This is not merely for the sake of fairness but actually helps to exhibit representativeness. To see why, note that the criteria space can be so vast that 10 random elements would hardly be enough to demonstrate all the interesting tradeoffs among the dimensions. Cross-query independence presents an increasingly clear picture of the diversity as more queries are answered over time.

3 BASIC METHODS

This section will explain two rudimentary techniques for IQS. The first one, *the alias method* (Section 3.1), settles an IQS problem that is the foundation of many other IQS problems. The second technique, *tree sampling* (Section 3.2), is a simple way to adapt data structures to IQS.

3.1 The Alias Method

In this subsection, we will consider:

> **Weighted Set Sampling.** The input dataset S has n elements where each element $e \in S$ carries a positive weight $w(e)$. Let $W := \sum_{e \in S} w(e)$. A *weighted sample* from S is a random variable X such that $\mathbf{Pr}[X = e] = w(e)/W$ for each $e \in S$.

W.l.o.g., our discussion will assume $W = 1$. The alias method, proposed by Walker [25], gives a structure of $O(n)$ space that allows a sample to be independently drawn in constant time. To explain the method's rationale, imagine that we have somehow produced a set Υ of n *urns*, where each urn $\Lambda \in \Upsilon$ contains either one or two elements from S (the urns do not need to be disjoint). For each element e in Λ, assign a positive value $w(\Lambda, e)$. The assignment needs to satisfy two conditions:

(1) If an urn $\Lambda \in \Upsilon$ contains only one element e, then $w(\Lambda, e) = 1/n$. If Λ contains two elements e_1 and e_2, then $w(\Lambda, e_1) + w(\Lambda, e_2) = 1/n$.

(2) For every $e \in S$, it holds that

$$w(e) = \sum_{\Lambda \in \Upsilon : e \in \Lambda} w(\Lambda, e) \quad (2)$$

namely, the weight of e has been spread into all the urns where e appears.

After the above preparation, we can perform weighted sampling in constant time. First, pick an urn Λ from Υ uniformly at random. If Λ has only a single element, we return it as the sample. Otherwise, suppose that Λ has elements e_1 and e_2. We make a random choice between them, which returns e_i as the sample with probability $w(\Lambda, e_i) \cdot n$ for $i = 1$ or 2. The algorithm ensures that each element $e \in S$ be sampled with probability $\sum_{\Lambda \in \Upsilon : e \in \Lambda} \frac{w(\Lambda, e) \cdot n}{n} = w(e)$, where the equality used (2), as desired.

It is possible to prepare the urns in $O(n)$ time with n steps. Each step produces a new urn, removes an element from S, and adjusts the weight of another element remaining in S. As an invariant, before Step $i \in [1, n]$, the weights of all the $n - i + 1$ elements still in S must sum up to $1 - \frac{i-1}{n}$. Specifically, Step i first identifies an arbitrary element $e_1 \in S$ with $w(e_1) \leq 1/n$ and another arbitrary element $e_2 \in S$ with $w(e_2) \geq 1/n$. If $w(e_1) = 1/n$, the step creates an urn Λ containing just e_1 and assigns $w(\Lambda, e_1) = w(e_1)$. Otherwise, it creates an urn Λ containing e_1 and e_2, assigns $w(\Lambda, e_1) = w(e_1)$ and $w(\Lambda, e_2) = \frac{1}{n} - w(e_1)$, and decreases $w(e_2)$ by $\frac{1}{n} - w(e_1)$. In both cases, e_1 is removed from S after the urn creation. It is rudimentary to implement the above algorithm in $O(n)$ time.

The theorem below summarizes the above discussion.

Theorem 1. *For the weighted set sampling problem, there is a structure of $O(n)$ space that can be used to draw a weighted sample in $O(1)$ time. The sample is independent of all the samples previously drawn. The structure can be built in $O(n)$ time.*

We will refer to the above structure as the *alias structure*.

3.2 Tree Sampling

Let us start by defining the *tree sampling* problem:

> **Tree Sampling.** Let T be a tree with n nodes where each leaf z carries a positive weight $w(z)$. For each internal node u of T, define $w(u)$ as the total weight of all the leaves in the subtree of u. For a node q of T, a *weighted sample* from the subtree of q is a random variable X such that $\mathbf{Pr}[X = z] = w(z)/w(q)$ for each leaf z in the subtree. Given a node q and an integer $s \geq 1$, a query returns s independent weighted samples from the subtree of q. The query's output must be independent of those of the previous queries.

Given q, we can draw one weighted sample from its subtree using a top-down strategy. If q is a leaf, return it directly. Otherwise, let $v_1, v_2, ..., v_f$ be the child nodes of q, where $f \geq 1$ is the *fanout* of q (f need not be a constant). Sample a node X such that $\mathbf{Pr}[X = v_i] = w(v_i)/w(u)$ for each $i \in [1, f]$. Then, we recursively sample a leaf from the subtree of X. To implement the strategy efficiently, we build an alias structure at each internal node u to perform the child sampling described above in constant time. If u has fanout f, the structure occupies $O(f)$ space and can be built in $O(f)$ time. This implies that the alias structures of all the internal nodes consume $O(n)$ total space and can be constructed in $O(n)$ time in total. A query can then be answered in time proportional to the height of the subtree of q. To draw s samples, simply repeat the procedure s times.

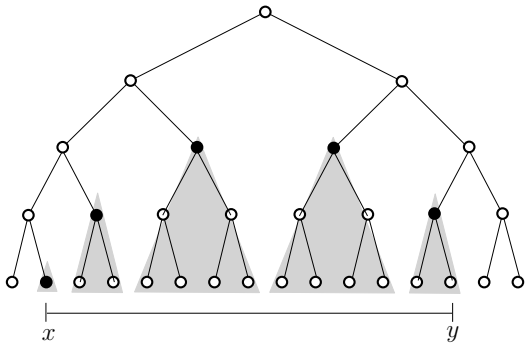

Figure 1: The tree is a BST. The black nodes are the canonical nodes of the query interval $q := [x, y]$. The shaded triangles indicate the subtrees of the canonical nodes.

The above approach can be applied to convert many structures originally designed for reporting queries to their IQS counterparts. To illustrate, we will look at:

> **Weighted Range Sampling.** The input dataset S has n elements from $\mathbb{R}$ where each element $e \in S$ carries a positive weight $w(e)$. Given an interval q in $\mathbb{R}$ and an integer $s \geq 1$, a query returns s independent weighted samples from $S_q := q \cap S$. The outputs of all queries must be mutually independent.

The reader may review Section 3.1 for the definition of "weighted sample from a set". If the goal is to report S_q in full, one can create a binary search tree (BST) $\mathcal{T}$ on S, which permits the discovery of S_q in $O(\log n + |S_q|)$ time for any q. For our discussion, we consider $\mathcal{T}$ to obey the following conventions:

- $\mathcal{T}$ has height $O(\log n)$.
- $\mathcal{T}$ has n leaves each storing a distinct value in S as the *key*.
- Every internal node u in $\mathcal{T}$ has two children. The leaf keys in the left subtree of u are less than those in the right subtree. The *key* of u equals the smallest leaf key in the right subtree.

Next, we explain how to adapt $\mathcal{T}$ for IQS by resorting to tree sampling. For each node u in $\mathcal{T}$, define $w(u)$ as the total weight of the leaf keys in the subtree of u. As a well-known fact, for any q, we can identify a set C of $O(\log n)$ *canonical nodes* in $\mathcal{T}$ such that

- the nodes in C have disjoint subtrees;
- the leaf keys in the subtrees of the nodes in C constitute S_q.

See Figure 1 for an illustration. To draw a weighted sample of S_q, we first sample a node X from C such that $\mathbf{Pr}[X = u] = w(u)/\sum_{u' \in C} w(u')$ for each $u \in C$. Then, we perform tree sampling to obtain a leaf z from the subtree of X, and the key of z serves as a weighted sample from S_q. The above algorithm takes $O(\log n)$ time to draw a sample. Hence, s samples can be obtained in $O(s \log n)$ time. It is clear that all queries have independent outputs.

Remark. The above structure for weighted range sampling is close to a structure described by Olken [21]. Recently, Martinez [20] integrated tree sampling with the range tree [11, 15] to tackle:

> **Multi-dimensional Weighted Range Sampling**. The input dataset S has n points from $\mathbb{R}^d$ where d is a fixed constant. Given a rectangle q of the form $[x_1, y_1] \times [x_2, y_2] \times ... \times [x_d, y_d]$ and an integer $s \geq 1$, a query returns s independent weighted samples from $S_q := q \cap S$. All queries' outputs must be mutually independent.

The structure of [20] uses $O(n \log^{d-1} n)$ space and answers a query in $O(\log^d n + s \log n)$ time. Focusing on 2D space, Looz and Meyerhenke [24] applied tree sampling to the quadtree to obtain a structure of $O(n)$ space and $O((\sqrt{n} + s) \log n)$ query time (under certain assumptions on the data).

4 TECHNIQUE 1: ALIAS AUGMENTATION

Section 3.2 has given a structure for the weighted range sampling problem that uses $O(n)$ space and answers a query in $O(s \log n)$ time. In this section, we will improve the query time to $O(\log n + s)$. The new structure illustrates a technique we call *alias augmentation*.

4.1 A Structure of $O(n \log n)$ Space

Build a BST $\mathcal{T}$ on S, obeying the conventions listed in Section 3.2. For each node u in $\mathcal{T}$, we use $S(u) \subseteq S$ to denote the set of elements from S that are stored at the leaves in the subtree of u. At each u, we store $w(u) := \sum_{e \in S(u)} w(e)$ and create an alias structure A_u (Theorem 1) on $S(u)$. Because A_u occupies $O(|S(u)|)$ space, the alias structures of all the nodes at the same level of $\mathcal{T}$ together use $O(n)$ space. The total space of the structure is therefore $O(n \log n)$.

Next, we explain how to answer a query with interval q and sample size s. As in Section 3.2, we start by identifying a set C of canonical nodes for q (see Figure 1) in $O(\log n)$ time. Let the nodes in C be $u_1, u_2, ..., u_t$ for some $t = O(\log n)$. Remember that the sets $S(u_1), S(u_2), ..., S(u_t)$ constitute a partition of $S_q := q \cap S$, namely, they are mutually disjoint and their union is exactly S_q.

The next step is to determine how many samples to take from each $S(u_i)$, $i \in [1, t]$. Notice that this is an instance of weighted set sampling (Section 3.1). Specifically, a weighted sample of S_q should come from $S(u_i)$ with probability $w(u_i)/\sum_{j=1}^{t} w(u_j)$ for each $i \in [1, t]$. Let s_i be the number of weighted samples to originate from $S(u_i)$, i.e., $s_1, ..., s_t$ are random variables with sum s. We can generate these random variables in $O(t + s) = O(\log n + s)$ time

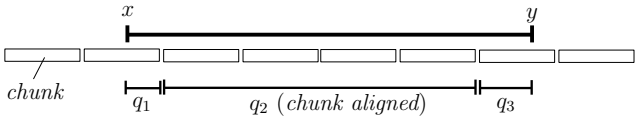

Figure 2: Query processing with chunks

using Theorem 1. First, construct an alias structure on C in $O(t)$ time, by treating $w(u)$ as the weight for each $u \in C$. Then, we draw s weighted samples from C in $O(s)$ time, after which s_i ($1 \leq i \leq t$) can be set to the number of occurrences of u_i in those samples.

For each $i \in [1, t]$, we take s_i weighted samples from $S(u_i)$ using the alias structure A_{u_i} in $O(s_i)$ time. The time of doing so for all i is $O(\sum_{i=1}^{t} s_i) = O(s)$. The overall query time is therefore $O(\log n + s)$.

From the above, we can now claim:

Lemma 2. *For the weighted range sampling problem, there is a structure of $O(n \log n)$ space answering a query in $O(\log n + s)$ time.*

4.2 Reducing the Space to $O(n)$

We can improve the space of our structure with a chunking idea. Divide $\mathbb{R}$ into interior-disjoint intervals $I_1, I_2, ..., I_g$ for some $g = \Theta(n/\log n)$ satisfying the following conditions:

- If we define $\mathcal{I}_i := I_i \cap S$ for each $i \in [1, g]$, then $\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_g$ are mutually disjoint and their union is S.
- $|\mathcal{I}_i| = \Theta(\log n)$ for each $i \in [1, g]$.

It is easy to obtain such intervals with sorting. We will refer to each $\mathcal{I}_i$ as a *chunk* and define $w(\mathcal{I}_i) := \sum_{e \in \mathcal{I}_i} w(e)$.

We create a structure $\mathcal{T}_{chunk}$ to support weighted range sampling at the chunk level. Specifically, the input dataset is $\mathcal{I} := \{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_g\}$ where each $\mathcal{I}_i$ ($1 \leq i \leq g$) carries weight $w(\mathcal{I}_i)$. Given $q_{chunk} := [a, b]$ — where a and b are integers in $[1, g]$ — and an integer $s \geq 1$, a query on $\mathcal{T}_{chunk}$ returns s independent weighted samples of $\mathcal{I}_{q_{chunk}} := \{\mathcal{I}_i \mid i \in q_{chunk}\}$. It should be noted that each sample here is a chunk (rather than an element in S). We can implement $\mathcal{T}_{chunk}$ as a structure of Lemma 2. The space of $\mathcal{T}_{chunk}$ is $O(|\mathcal{I}| \log |\mathcal{I}|) = O(\frac{n}{\log n} \log n) = O(n)$.

We also build a *range sum* structure which allows us to calculate $\sum_{i=a}^{b} w(\mathcal{I}_i)$ in $O(\log n)$ time for any a and b satisfying $1 \leq a \leq b \leq g$; such a structure can be a slightly augmented BST (see Chapter 14 of [14]), which uses $O(g) = o(n)$ space. In addition, for each $i \in [1, g]$, we build an alias structure to support weighted range sampling from chunk $\mathcal{I}_i$. The alias structures of all the chunks use $O(\sum_{i=1}^{g} |\mathcal{I}_i|) = O(n)$ space in total. The overall space consumption is therefore $O(n)$.

Consider now a query with interval $q := [x, y]$ and sample size s. Let us first assume that the query is *chunk aligned*, namely, $q = I_a \cup I_{a+1} \cup ... \cup I_b$, for some a and b such that $1 \leq a \leq b \leq g$. The values of a and b can be determined in $O(\log n)$ time with binary search. Conceptually, a weighted sample from S_q can be extracted in a two-step manner. We first take a weighted sample $\mathcal{I}$ from the set of chunks $\{\mathcal{I}_a, \mathcal{I}_{a+1}, ..., \mathcal{I}_b\}$ and then acquire a weighted sample e from $\mathcal{I}$. The element e, which is sampled with probability

$$\frac{w(\mathcal{I})}{\sum_{j=1}^{g} w(\mathcal{I}_j)} \cdot \frac{w(e)}{w(\mathcal{I})} = \frac{w(e)}{\sum_{e' \in S_q} w(e')},$$

is thus a weighted sample of S_q. Motivated by this, we answer the query with the following two-level sampling strategy. First, extract s weighted samples from $\{\mathcal{I}_a, \mathcal{I}_{a+1}, ..., \mathcal{I}_b\}$, which can be done in $O(\log n + s)$ time using $\mathcal{T}_{chunk}$. For each $i \in [a, b]$, let s_i be the number of occurrences of $\mathcal{I}_i$ in those s samples. Then, we extract s_i weighted samples from $\mathcal{I}_i$ using the alias structure on $\mathcal{I}_i$ in $O(s_i)$ time. The total query cost is therefore $O(\log n + \sum_{i=a}^{b} s_i) = O(\log n + s)$.

It remains to discuss the scenario where $q := [x, y]$ can be an arbitrary interval in $\mathbb{R}$. Suppose that x and y fall in I_a and I_b, respectively, where $1 \leq a \leq b \leq g$. We can break q into three intervals q_1, q_2 and q_3 as follows:

- q_1 is the portion of q in the interior of I_a,
- $q_2 := I_{a+1} \cup I_{a+2} \cup ... \cup I_{b-1}$, and
- q_3 is the remaining portion of q after trimming q_1 and q_2 (it must hold that $q_3 \subseteq I_b$).

See Figure 2 for an illustration. Define $S_1 := S \cap q_1$, $S_2 := S \cap q_2$, and $S_3 := S \cap q_3$. For each $j \in [1, 3]$, define $w(S_j) := \sum_{e \in S_j} w(e)$. We can obtain S_1 and S_3 in their entirety in $O(\log n)$ time by reading the chunks $\mathcal{I}_a$ and $\mathcal{I}_b$ directly. For S_2, we cannot afford to enumerate its elements but can obtain $w(S_2)$ in $O(\log n)$ time from the range sum structure. To extract s weighted samples from S_q, we first determine the number s_j of samples that will come from S_j for each $j = 1, 2$, and 3 (this can be easily done in $O(s)$ time). By resorting to Theorem 1, the s_1 and s_3 samples from S_1 and S_3 can be fetched in $O(\log n + s_1 + s_3)$ time. Acquiring s_2 samples from S_2 requires only a chunk-aligned query, which we already know how to solve in $O(\log n + s_2)$ time. We thus conclude that the overall query time is $O(\log n + s_1 + s_2 + s_3) = O(\log n + s)$.

We have arrived at:

Theorem 3. *For the weighted range sampling problem, there is a structure of $O(n)$ space answering a query in $O(\log n + s)$ time.*

4.3 Remarks

The above solution is close to a structure of Hu et al. [18], although their discussion focused on the WR sampling scheme (a special case of weighted sampling where all elements have the same weight). Hu et al. [18] also showed that their structure (for WR sampling) supports an insertion and deletion in $O(\log n)$ time. In contrast, the structure in Section 4.2 cannot be easily modified to support updates (because it is not easy to dynamize the alias structure).

Afshani and Wei [3] considered weighted range sampling in the scenario where all the elements of S come from the integer domain $[1, U]$. They obtained a static structure of $O(n)$ space whose query time is $O(\log\log U + s)$. Their result can actually be stated in a more general way, which we will clarify when proving Lemma 4 in the next section.

5 TECHNIQUE 2: COVERAGE

Before introducing the next technique, let us first improve our solution to the tree sampling problem in Section 3.2. Recall that we gave a structure of $O(n)$ space that answers a query in $O(s \cdot h)$ time where h is the height of the tree. Next, we show how to achieve query time $O(\log n + s)$ using Theorem 3.

Perform a depth first traversal (DFT) of T. Let Π be the sequence of leaves encountered in the traversal (note: although the traversal visits internal nodes as well, Π concerns only the leaves). DFT ensures a nice, rudimentary, property:

Proposition 1. *For each node u in T, the leaves in the subtree of u constitute a contiguous portion of Π.*

Consider a tree sampling query with parameters q and s (the goal is to draw s independent weighted samples from the subtree of node q). The query can now be converted to weighted range sampling. By Proposition 1, the leaves in the subtree of q are the nodes in $\Pi[a : b]$ — for some a and b satisfying $1 \leq a \leq b \leq n$ — which represents the subsequence of Π starting from the a-th position and ending at the b-th position (both positions inclusive). In preprocessing, we can store the values of a and b at q using constant words so that they can be obtained directly once q is known. It is clear that the query essentially extracts s weighted samples from $\Pi[a : b]$. Theorem 3 thus implies a structure of $O(n)$ space that answers a query in $O(\log n + s)$ time.

It turns out that we can do even better:

Lemma 4. *There is a structure for the tree sampling problem that uses $O(n)$ space and answers a query in $O(1 + s)$ time.*

Proof. Afshani and Wei [3] proved the following result on the weighted range sampling problem defined in Section 3.2. Suppose that the input dataset S consists of n distinct elements drawn from the integer domain $[1, U]$ for some $U \geq n$. Consider a query with interval q and sample size s. If q always has the form $[a, b]$ where both a and b are elements from S, then there is a structure of $O(n)$ space that can answer a query in $O(1 + s)$ time. Combining this result with the earlier discussion yields the lemma (for tree sampling, $U = n$). □

The lemma enhances the power of tree sampling, which as mentioned in Section 3.2 serves as a generic technique to adapt a traditional reporting structure for IQS. Next, we will generalize the discussion of Section 3.2 to a class of tree-based structures.

Let S be the input dataset. A reporting query, in general, specifies a predicate q and returns $S_q := \{e \in S \mid e \text{ satisfies } q\}$. Consider a structure designed for such queries. For our discussion, we assume that the structure builds a tree $\mathcal{T}$ such that each element is stored at a distinct leaf of $\mathcal{T}$. For each node u in $\mathcal{T}$, define by $S(u)$ the set of elements stored in the subtree of u. Given a predicate q and a set C of nodes, we call C a *cover* of q if

- the nodes in C have disjoint subtrees;
- $S_q = \bigcup_{u \in C} S(u)$.

A cover definitely exists because we can always form C by adding every leaf that stores an element of S_q. We assume that, given a q, the structure is able to find a cover of q of a small size; let us denote that particular cover as C_q.

Now, consider the query's IQS version under weighted sampling. Specifically, each element in S carries a weight. Given a predicate q and an integer $s \geq 1$, an IQS query returns s independent weighted samples from S_q. The outputs of all queries must be mutually independent. The next theorem gives a generic approach to convert $\mathcal{T}$ into an IQS structure:

Theorem 5. *Consider the tree-based structure as described earlier. Let m be the number of nodes in $\mathcal{T}$. With $O(m)$ additional space, we can obtain a structure that, given an IQS query (weighted sampling) with predicate q and sample size s, answers the query in $O(|C_q| + s)$ time, plus the time for finding C_q.*

Proof. In preprocessing, we store at each node u of $\mathcal{T}$ the value $w(u) := \sum_{e \in S(u)} w(e)$. Then, build a structure of Lemma 4 to support tree sampling on $\mathcal{T}$, treating $w(z)$ as the weight of each leaf z. The additional space consumption is $O(m)$.

Now, consider a query with parameters q and s. We first instruct the structure to find the cover C_q. The rest of the algorithm is similar to what was illustrated in Section 3.2. Specifically, use Theorem 1 to build an alias structure on C_q on the fly in $O(|C_q|)$ time, treating $w(u)$ as the weight of each node $u \in C_q$. To obtain a weighted sample from S_q, we perform two steps. First, draw a weighted sample u (i.e., a node) from C_q, which needs $O(1)$ time by Theorem 1. Second, take a weighted sample e from S_u, which again needs $O(1)$ time (Lemma 4). The element e is a weighted sample from S_q. Repeating the steps s times produces s samples. □

Many index structures in the database area are tree based, and the theorem converts all of them into IQS structures. Section 3.2 has explained that the BST always ensures a cover C_q of size $O(\log n)$ (i.e., containing the canonical nodes) for a range query with interval q. Hence, Theorem 5 indicates (once again) that there is a structure for weighted range sampling that occupies $O(n)$ space and has $O(\log n + s)$ query time. We give two extra notable examples below.

- Let S be a set of n points in $\mathbb{R}^d$ where d is a constant. Given a rectangle q of the form $[x_1, y_1] \times [x_2, y_2] \times ... \times [x_d, y_d]$, an *orthogonal range query* reports $S_q := S \cap q$. A kd-tree [10] on S uses $O(n)$ space and permits us to find a cover C_q of size $O(n^{1-1/d})$ for every q. Theorem 5 directly gives an IQS structure of $O(n)$ space and $O(n^{1-1/d} + s)$ query time for the multi-dimensional weighted range sampling problem (Section 3.2) (improving the structure of Looz and Meyerhenke [24] mentioned in Section 3.2).
- Consider again orthogonal range queries on S. A range tree [11, 15] on S uses $O(n \log^{d-1} n)$ space and permits us to find

a cover C_q of size $O(\log^d n)$ for every q. Theorem 5 yields[4] a structure for multi-dimensional weighted range sampling that uses $O(n \log^{d-1} n)$ space and guarantees $O(\log^d n + s)$ query time (improving the structure of Martinez [20] mentioned in Section 3.2).[5]

Remarks. Lemma 4 was first formally stated by Afshani and Phillips [2]. Theorem 5 can be regarded as a generalization of a technique deployed by Xie et al. [27] to derive the kd-tree-based IQS result mentioned earlier.

6 TECHNIQUE 3: APPROXIMATE COVERAGE

Next, we will explain how to strength the coverage technique in the previous section. To keep our discussion simple, let us focus on WR sampling (or equivalently, weighted sampling where all elements carry the same weight).

As in Section 5, let S be a set of elements and define $S_q := \{e \in S \mid e \text{ satisfies } q\}$ for a predicate q. Let $\mathcal{T}$ be the tree built on S by a data structure in the context of Theorem 5. Define $S(u)$ again as the set of elements in the subtree of u. Given a predicate q and a set C of nodes in $\mathcal{T}$, we call C an *approximate cover* of q if

- the nodes in C have disjoint subtrees;
- $S_q \subseteq \bigcup_{u \in C} S(u)$;
- $|S_q| = \Omega(|\bigcup_{u \in C} S(u)|)$.

Note that we no longer require $S_q = \bigcup_{u \in C} S(u)$ (as was needed in Section 5 for C to qualify as a cover). However, there is an extra condition (3rd bullet), which says that at least a constant portion of the elements in $\bigcup_{u \in C} S(u)$ need to be in S_q. Intuitively, this means that if we sample WR an element from $\bigcup_{u \in C} S(u)$, the element falls in S_q with constant probability. Hence, we expect to obtain a WR sample from S_q by repeating $O(1)$ times.

A cover of q is definitely an approximate cover, but the opposite may not be true. For the same q, an approximate cover can be even smaller than the smallest cover. For example, consider a BST on a set S of n real values, and each predicate q as an interval in $\mathbb{R}$ such that $S_q := S \cap q$. There exist intervals q for which any cover must have a size $\Omega(\log n)$. In contrast, for any q, there is always an approximate cover with size at most 2, as shown in [18].

Henceforth, we assume that, for each q, the data structure can identify an approximate cover $\tilde{C}_q$ of a small size. The next result echoes Theorem 5.

Theorem 6. *Consider the tree-based structure described earlier. Let m be the number of nodes in $\mathcal{T}$. With $O(m)$ additional space, we can obtain a structure that answers an IQS query (weighted sampling) in $O(|\tilde{C}_q| + s)$ expected time, plus the time for finding $\tilde{C}_q$, where q is the query predicate and s is the number of samples.*

Proof. The preprocessing is the same as in the proof of Theorem 5. To answer a query with parameters q and s, we instruct the structure to find $\tilde{C}_q$ and apply Theorem 1 to build an alias structure on $\tilde{C}_q$ in $O(|\tilde{C}_q|)$ time. To obtain a WR sample from S_q, we first draw a weighted sample u from $|\tilde{C}_q|$ in $O(1)$ time and then draw a WR sample e from S_u also in $O(1)$ time (Lemma 4). If e satisfies q, it is a WR sample from S_q; otherwise, discard e and obtain another sample by repeating the two steps. In expectation, a constant number of repeats will churn out a sample from S_q. The time to produce s samples is thus $O(s)$ expected. □

Different predicates may share the same approximate cover, that is, $\tilde{C}_{q_1} = \tilde{C}_{q_2}$ even though $q_1 \neq q_2$. Define

$$\tilde{\mathbb{C}} = \left\{\tilde{C}_q \mid \text{all predicates } q\right\};$$

in other words, $\tilde{\mathbb{C}}$ collects all the distinct approximate covers that the data structure may identify for a predicate. The corollary below explains a way to trade space usage for query efficiency.

Corollary 7. *Consider the same tree-based structure in Theorem 6. With $O(m + \sum_{\tilde{C} \in \tilde{\mathbb{C}}} |\tilde{C}|)$ additional space, we can obtain a structure that answers an IQS query (WR sampling) in $O(s)$ expected time, plus the time for finding $\tilde{C}_q$, where q is the query predicate and $s \geq 1$ is the number of samples.*

Proof. The $|\tilde{C}_q|$ term in the query time of Theorem 6 comes from constructing an alias structure on $\tilde{C}_q$. We can eliminate the term by computing the structure in preprocessing. Storing such a structure for each $\tilde{C} \in \tilde{\mathbb{C}}$ increases the space by $O(\sum_{\tilde{C} \in \tilde{\mathbb{C}}} |\tilde{C}|)$. □

The astute reader may wonder whether Theorem 5 would admit a corollary similar to the above. The answer is affirmative but the usefulness of Corollary 7 is mainly due to *approximate* covers. Specifically, approximate covers can be much easier to form compared to their exact counterpart, because of which we can hope to (approximately) cover all predicates with a relatively small $\tilde{\mathbb{C}}$. Indeed, this is the key reason why Corollary 7 can yield interesting IQS structures (see the remark below).

Remarks. Our discussion generalizes an approach developed by Afshani and Wei [3] in tackling the IQS version of 3D *halfspace reporting* [1] under WR sampling. Viewed through the lens of Corollary 7, their solution serves as an excellent example of how to construct a $\tilde{\mathbb{C}}$ whose $\sum_{\tilde{C} \in \tilde{\mathbb{C}}} |\tilde{C}|$ is linear to the size of the input dataset. Approximate coverage can be adapted to work for weighted sampling by combining the above discussion with ideas from [2] where Afshani and Phillips extended the halfspace IQS result of [3] to weighted sampling.

Considering that the query time in Theorem 6 is expected, one may wonder if it is possible to make it hold in the worst case. Recently, Afshani and Phillips [3] established some hardness results suggesting that such improvement would be unlikely.

7 TECHNIQUE 4: RANDOM PERMUTATION

Earlier in Section 2 we have seen how random permutation helps to design a structure for query sampling, although that structure did not guarantee cross-query independence. In this section, we will show that random permutation is useful for IQS as well. For this purpose, we will consider:

[4] Strictly speaking, a range tree does not satisfy our requirement for $\mathcal{T}$ because the same element can be stored at multiple leaves in a range tree. However, this is a minor issue that can be easily remedied. We omit the details here.

[5] We note that the query time can be further reduced to $O(\log^{d-1} n + s)$, by incorporating additional ideas based on fractional cascading [13].

Set Union Sampling. Let $\mathcal{F}$ be a collection of sets, namely, each member of $\mathcal{F}$ is a set. The elements of the sets in $\mathcal{F}$ originate from an identical domain (the domain's details are irrelevant). For any subset $\mathcal{G} \subseteq \mathcal{F}$, define $\bigcup \mathcal{G} := \bigcup_{S \in \mathcal{G}} S$, namely, the union of the sets in $\mathcal{G}$. Given a $\mathcal{G} \subseteq \mathcal{F}$, a query returns an element sampled uniformly at random from $\bigcup \mathcal{G}$. The outputs of all queries must be mutually independent.

The problem was introduced by Har-Peled and Mahabadi [17] and stands at the core of all the known solutions [6–8, 17] to fair near neighbor search introduced in Section 2.

Define $n := \sum_{S \in \mathcal{F}} |S|$ and $U := |\bigcup \mathcal{F}|$. Note the subtle difference between n and U: the former is the total size of all the sets in $\mathcal{F}$, whereas the latter is the total number of distinct elements in those sets.

Consider a query with parameter $\mathcal{G}$. We define $g := |\mathcal{G}|$ and $U_{\mathcal{G}} := |\bigcup \mathcal{G}|$. Denote by $S_1, S_2, ..., S_g$ the sets in $\mathcal{G}$. The problem is easy if those sets are mutually disjoint. In that case, we can first choose a set X from $\mathcal{G}$ randomly according to the rule $\mathbf{Pr}[X = S_i] = |S_i| / \sum_{j=1}^{g} |S_j|$ and then return a random element from X. The overall query cost is $O(g)$. The approach, however, no longer works if the sets of $U_{\mathcal{G}}$ may overlap with each other. Overcoming the obstacle requires additional ideas, as explained next.

Structure. Perform a random permutation of the elements in $\bigcup \mathcal{F}$ and denote the permuted sequence as Π. Define the *rank* of each element $e \in \bigcup \mathcal{F}$ as its position in Π. For each set $S \in \mathcal{F}$, we build a range reporting structure to return the elements in S whose ranks fall in $[a, b]$, for any a and b satisfying $1 \leq a \leq b \leq U$. The structure can be a BST, which uses $O(|S|)$ space and returns all the qualifying elements in $O(\log n + k)$ time, where k is the number of reported elements. The overall space consumption is $O(n)$ and the construction time is $O(n \log n)$.

Query. Next, we explain how to answer a query with parameter $\mathcal{G} = \{S_1, ..., S_g\}$. Let us first assume the availability of a value $\hat{U_{\mathcal{G}}}$ which satisfies $U_{\mathcal{G}}/2 \leq \hat{U_{\mathcal{G}}} \leq 1.5U_{\mathcal{G}}$. The assumption will be removed in the end.

Conceptually, cut the rank space $[1, U]$ into $\hat{U_{\mathcal{G}}}$ disjoint intervals $I_1, I_2, ..., I_{\hat{U_{\mathcal{G}}}}$ each with length $U/\hat{U_{\mathcal{G}}}$. For each $i \in [1, g]$ and each $j \in [1, \hat{U_{\mathcal{G}}}]$, define

$$S_i(I_j) := \{e \in S_i \mid \text{rank of } e \text{ is in } I_j\}.$$

Define for each $j \in [1, \hat{U_{\mathcal{G}}}]$

$$\bigcup I_j := \bigcup_{i=1}^{g} S_i(I_j) \tag{3}$$

namely, the set of elements in $\bigcup \mathcal{G}$ whose ranks are in I_j. As $\bigcup \mathcal{G}$ has $U_{\mathcal{G}}$ elements and Π is a random permutation, we expect to see only $U_{\mathcal{G}}/\hat{U_{\mathcal{G}}} = \Theta(1)$ elements in $\bigcup I_j$. Let $m := c \log_2 n$ where c is a sufficiently large constant to make following event occur with probability at least $1 - \delta = 1 - 1/n^3$: it holds for every $j \in [1, \hat{U_{\mathcal{G}}}]$ that

$$\left|\bigcup I_j\right| \quad \leq \quad m. \tag{4}$$

We will proceed by assuming the event's occurrence.

The query algorithm runs as follows:

1. pick an interval I uniformly at random from $\{I_1, ..., I_{\hat{U_{\mathcal{G}}}}\}$
2. retrieve $S_i(I)$ for each $i \in [1, g]$
3. toss a coin with heads probability $|\bigcup I|/m$
   /* $\bigcup I := \bigcup_{i=1}^{g} S_i(I)$, as defined in (3) */
4. **if** the coin comes up heads **then**
5. return a uniformly random element from $\bigcup I$

The above algorithm returns nothing if the coin at Line 3 comes up tails. In that case, we simply repeat the algorithm until a sample is finally returned.

Let e be an arbitrary element in $\bigcup \mathcal{G}$, and define $j \in [1, \hat{U_{\mathcal{G}}}]$ as the integer such that $e \in \bigcup I_j$. We have:

$$\mathbf{Pr}[\text{Line 5 returns } e] = \frac{1}{\hat{U_{\mathcal{G}}}} \cdot \frac{|\bigcup I_j|}{m} \cdot \frac{1}{|\bigcup I_j|} = \frac{1}{\hat{U_{\mathcal{G}}} \cdot m}. \tag{5}$$

As the probability is the same for all $e \in \bigcup \mathcal{G}$, we can assert that the query outputs a uniformly random sample from $\bigcup \mathcal{G}$ at Line 5. Observe that the summation of (5) for all $e \in \bigcup \mathcal{G}$ equals $\frac{1}{m} \cdot (U_{\mathcal{G}}/\hat{U_{\mathcal{G}}}) = \Theta(1/m)$. In other words, the algorithm returns a sample with probability $\Theta(1/m)$, meaning that we can get a sample with $\Theta(m) = \Theta(\log n)$ repeats in expectation.

For each $i \in [1, g]$, the BST on S_i allows us to find $S_i(I)$ at Line 2 in $O(\log n + |S_i(I)|)$ time which is at most $O(\log n + m) = O(\log n)$ because of (4). It thus becomes clear that the query algorithm runs in $O(g \log n)$ time. As we need $O(\log n)$ repeats in expectation, the total running time is $O(g \log^2 n)$ in expectation.

Deriving $\hat{U_{\mathcal{G}}}$. Deriving $U_{\mathcal{G}}$ precisely requires reading all the sets in $\mathcal{G}$, which incurs cost we cannot afford. However, it is much cheaper to derive an estimate $\hat{U_{\mathcal{G}}}$ meeting our requirement $\hat{U_{\mathcal{G}}}/2 \leq U_{\mathcal{G}} \leq 1.5\hat{U_{\mathcal{G}}}$ by resorting to sketches.

In general, given a set S of elements drawn from a domain of size m, we can compute a sketch of [9], which stores $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ words, is constructible in $O(|S| \log \frac{1}{\delta})$ expected time, and can be used to derive an estimate of $|S|$ with relative error at most ϵ in constant time. Furthermore, suppose that such a sketch is available on sets S_1 and S_2, respectively (their elements are drawn from the same domain of size m). We can merge their sketches into a sketch on $S_1 \cup S_2$ in $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ expected time.

For our purposes, we parameterize m to U, δ to $1/n^3$, and ϵ to $1/2$. In preprocessing, build a sketch for each $S \in \mathcal{F}$ with $|S| \geq \log_2 n$. The sketch on S requires $O(\log \frac{1}{\delta}) = O(\log n) = O(|S|)$ words and can be built in $O(|S| \log n)$ time. The total space consumption thus remains $O(n)$ and the total reconstruction time is $O(n \log n)$ expected.

Given a query with $\mathcal{G}$ (as before, define $g := |\mathcal{G}|$), we merge the sketches of the sets in $\mathcal{G}$ to obtain a sketch on $\bigcup \mathcal{G}$. Note, however, that a set $S \in \mathcal{G}$ may not have a pre-built sketch. But in that case, the size of S must be at most $\log_2 n$ and we can build its sketch on the fly in $O(|S| \log \frac{1}{\delta}) = O(\log^2 n)$ expected time. Thus, in $O(g \log^2 n)$ expected time, we make sure that all the sets in $\mathcal{G}$ have sketches, which can then be merged in $O(g \log \frac{1}{\delta}) = O(g \log n)$ time. After

that, with probability at least $1 - 1/n^3$, we can derive in constant time an estimate $\hat{U_{\mathcal{G}}}$ of $U_{\mathcal{G}} := |\bigcup \mathcal{G}|$ with relative error at most $1/2$. This $\hat{U_{\mathcal{G}}}$ fulfills our purposes.

We thus have shown:

Theorem 8. *For the set union sampling problem, there is a structure of $O(n)$ space that can be built in $O(n \log n)$ expected time and answers a query correctly with probability at least $1 - 2/n^3$. The query cost is $O(g \log^2 n)$ expected where g is the number of sets in the query's parameter $\mathcal{G}$.*

The above theorem, as stated in the way above, holds for one query. However, by applying standard rebuilding techniques, we can extend the result to hold on all queries. It suffices to rebuild the structure after n queries. The rebuilding cost $O(n \log n)$ expected can be amortized over those queries, each of which bears $O(\log n)$ expected. With probability at least $1 - \Theta(1/n^2)$, all those n queries are answered correctly. The queries' outputs are always mutually independent.

Remarks. The above solution follows ideas due to Aumuller et al. [8], which were later refined in [7]. The discussion of [7, 8], however, did not touch upon space consumption. A straightforward implementation would create a sketch on every set in $\mathcal{F}$ and end up with $O(n \log n)$ space. The issue can be fixed by building sketches only on sets of size at least $\log_2 n$ and modifying the query algorithm accordingly, as shown earlier.

8 EXTERNAL MEMORY

Our discussion has so far assumed the RAM computation model. In database systems, the input dataset is often stored in the disk such that a query needs to fetch data into memory via disk I/Os. In such scenarios, RAM ceases to be a reasonable model because CPU time is no longer the performance bottleneck. The predominant model for studying I/O-efficient algorithms is the *external memory* (EM) model introduced by Aggarwal and Vitter [4].

In EM, a machine has M words of memory and a disk that has an unbounded size but has been formatted into blocks of size B words. $M \geq 2B$, namely, the memory can hold at least two blocks of data. An *I/O* either reads a disk block of data into memory, or writes B words in memory to a disk block. The *cost* of an algorithm is the number of I/Os performed (CPU time is free), while the *space* of a structure is the number of disk blocks occupied.

In RAM, an algorithm outputting k elements needs $\Omega(k)$ time to just to read the elements from memory. In EM, however, the cost of reading k elements can be as low as $\lceil k/B \rceil$ I/Os. This is why an interesting EM algorithm usually has an output term $O(k/B)$, rather than $O(k)$. For example, in RAM, the BST answers a range query (see Section 1) in $O(\log n + k)$ time, where n is the size of the input dataset. In EM, the BST's cost becomes $O(\log n + k)$ I/Os, which, however, is not attractive at all. In contrast, the B-tree achieves the purpose in $O(\log_B n + k/B)$ I/Os.

Many IQS algorithms in RAM rely on random accesses to achieve satisfactory efficiency and become prohibitively expensive in EM. To illustrate, let us consider perhaps the simplest IQS problem:

Set Sampling. The input dataset S has n elements. Given an integer $s \geq 1$, a query returns s independent WR samples from S. The outputs of all queries must be mutually independent.

In RAM, the problem can be easily settled as follows. Simply store S in an array and answer a query by returning s uniformly random elements from the array. The space consumption is $O(n)$ and the query time is $O(s)$; both complexities are optimal in RAM. In EM, the same approach uses $O(n/B)$ space and incurs $O(s)$ I/Os per query. The space complexity is optimal but the query cost is terrible.

Naturally, one would seek a structure that can guarantee $O(s/B)$ query cost (which would "match" the RAM result). This turned out to be impossible. A fundamental result, due to Hu et al. [18], is that, when $B \leq s \leq n^{0.99}$, every structure must spend $\Omega(\min\{s, \frac{s}{B} \log_{M/B} \frac{n}{B}\})$ I/Os answering a query, *regardless of how much space is used.* In fact, this is true even if the query cost is amortized. Indeed, for set sampling, a sequence of t queries with parameters $s_1, s_2, ..., s_t$ is equivalent to a single query with parameter $s = \sum_{i=1}^{t} s_i$. The total cost of all those queries must be $\Omega(\min\{s, \frac{s}{B} \log_{M/B} \frac{n}{B}\})$.

On the other hand, we can achieve $O(\min\{s, \frac{s}{B} \log_{M/B} \frac{n}{B}\})$ amortized query cost with a linear-space structure (i.e., space $O(n/B)$), thus matching the aforementioned lower bound. We consider only $\frac{s}{B} \log_{M/B} \frac{n}{B} \leq s$ because otherwise it suffices to run the RAM solution in EM directly. In preprocessing, store all the elements of S in an array and, in addition, store n samples WR from S in a separate array called the *sample pool*. With sorting, the samples can be obtained in $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/Os [4]. When the structure is newly built, all the samples are marked as *clean*. To answer a query with parameter s, we simply return the next s clean samples from the pool and mark them *dirty*. When the pool runs out of clean samples, we rebuild the pool in $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/Os. The cost can be amortized on the n samples already returned, so that each of them is charged only $O(\frac{1}{B} \log_{M/B} \frac{n}{B})$ I/Os. The amortized query cost is therefore $O(\frac{s}{B} \log_{M/B} \frac{n}{B})$. We can also achieve a worst-case bound of $O(1 + \frac{s}{B} \log_{M/B} \frac{n}{B})$ for every query with standard deamortization techniques [5].

Building upon the above idea, Hu et al. [18] presented two EM structures for the weighted range sampling problem (see Section 3.2) in EM, under the special scenario where all elements have equal weights (i.e., WR sampling). Their structures achieve different space and query tradeoffs. The first one uses $O(\frac{n}{B} \log^* \frac{n}{B})$ space[6] and answers a query in $O(\log_B n + \frac{s}{B} \log_{M/B} \frac{n}{B})$ I/Os amortized. Note that the term $\frac{s}{B} \log_{M/B} \frac{n}{B}$ is necessary because weighted range sampling generalizes set sampling. The second structure uses $O(n/B)$ space and answers a query in $O(\log^* \frac{n}{B} + \log_B n + \frac{s}{B} \log_{M/B} \frac{n}{B})$ I/Os amortized.

9 CONCLUDING REMARKS

Today, the IQS literature [2, 3, 6–8, 17, 18, 20, 21, 24, 25, 27] harbors a trove of interesting ideas, many of which, unfortunately, cannot be included in this article. However, we are still far from understanding

[6] The iterated logarithm $\log^* x$ is the smallest t such that $\underbrace{\log_2 \log_2 ... \log_2}_{t} x \leq 2$.

IQS thoroughly. Next, we list several promising directions — beyond what has been eluded earlier — for future investigation.

- **Direction 1: Dynamization.** With few exceptions [18], the previous IQS research has focused on static data. In practice, however, datasets are subject to frequent updates. Thus, it is an urgent task to extend the existing structures to support fast insertions and deletions. A fundamental problem is to dynamize the alias method in Section 3.1, namely, how to support fast insertions and deletions in the input set S, while still allowing independent sample extraction in constant time. The reader may refer to [16] for an optimal solution when the elements in S have integer weights.
- **Direction 2: EM.** There has not been much progress on IQS structures in EM since [18]. The EM model continues to retain its practical importance: data volume has been increasing at a faster pace than memory capacity. As explained in Section 8, IQS in EM requires techniques drastically different from those in RAM. Even weighted range sampling (which has been well understood in RAM) remains open in EM: it is a major challenge to design a structure of $O(n/B)$ space and $O(\log_B n + \frac{s}{B} \log_{M/B} \frac{n}{B})$ amortized query cost.
- **Direction 3: Generic Reductions.** In RAM, nearly every known IQS structure "matches" the corresponding reporting structure in terms of space and query complexities. For example, the structure for weighted range sampling in Section 4 uses $O(n)$ space and answers an IQS query in $O(\log n + s)$ time, "matching" the performance of the BST which uses $O(n)$ space and resolves a range query in $O(\log n + k)$ time. Is all the correspondence we have witnessed merely a coincidence? Or maybe there exists a reduction from IQS to the counterpart "reporting query"? More generally, can we prove that IQS is equivalent to solving a certain myriad of queries? Some progress has been reported in [2] in this regard.
- **Direction 4: Approximate IQS.** Many estimation tasks can be carried out with *approximate sampling*, namely, the sample probability of a possible outcome is allowed to slightly deviate from its intended value. For example, while uniform sampling is supposed to take each element from a set of size n with probability $1/n$, *ϵ-uniform sampling* would take each element with a probability between $\frac{1}{(1+\epsilon)n}$ and $\frac{1+\epsilon}{n}$. How does the value ϵ affect the space and query (possibly also update) complexities of IQS? Research on this topic has emerged recently [7].

ACKNOWLEDGEMENTS

This work was partially supported by GRF projects 142034/21 and 142078/20 from HKRGC.

REFERENCES

[1] Peyman Afshani and Timothy M. Chan. Optimal halfspace range reporting in three dimensions. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 180–186, 2009.
[2] Peyman Afshani and Jeff M. Phillips. Independent range sampling, revisited again. In *Proceedings of Symposium on Computational Geometry (SoCG)*, volume 129, pages 4:1–4:13, 2019.
[3] Peyman Afshani and Zhewei Wei. Independent range sampling, revisited. In *Proceedings of European Symposium on Algorithms (ESA)*, volume 87, pages 3:1–3:14, 2017.
[4] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.
[5] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM Journal of Computing*, 32(6):1488–1508, 2003.
[6] Martin Aumuller, Sariel Har-Peled, Sepideh Mahabadi, Rasmus Pagh, and Francesco Silvestri. Fair near neighbor search via sampling. *SIGMOD Rec.*, 50(1):42–49, 2021.
[7] Martin Aumuller, Sariel Har-Peled, Sepideh Mahabadi, Rasmus Pagh, and Francesco Silvestri. Sampling a near neighbor in high dimensions - who is the fairest of them all? *ACM Transactions on Database Systems (TODS)*, 2021.
[8] Martin Aumuller, Rasmus Pagh, and Francesco Silvestri. Fair near neighbor search: Independent range sampling in high dimensions. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 191–204, 2020.
[9] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Proceedings of International Workshop on Randomization and Approximation Techniques (RANDOM)*, volume 2483, pages 1–10, 2002.
[10] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM (CACM)*, 18(9):509–517, 1975.
[11] Jon Louis Bentley. Decomposable searching problems. *Information Processing Letters (IPL)*, 8(5):244–251, 1979.
[12] Gerth Stolting Brodal, Rolf Fagerberg, Mark Greve, and Alejandro Lopez-Ortiz. Online sorted range reporting. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 173–182, 2009.
[13] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2):133–162, 1986.
[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.
[15] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008.
[16] Torben Hagerup, Kurt Mehlhorn, and J. Ian Munro. Maintaining discrete probability distributions optimally. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 253–264, 1993.
[17] Sariel Har-Peled and Sepideh Mahabadi. Near neighbor: Who is the fairest of them all? In *Proceedings of Neural Information Processing Systems (NuerIPS)*, pages 13176–13187, 2019.
[18] Xiaocheng Hu, Miao Qiao, and Yufei Tao. Independent range sampling. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 246–255, 2014.
[19] Xiaocheng Hu, Miao Qiao, and Yufei Tao. External memory stream sampling. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 229–239, 2015.
[20] Andres Lopez Martinez. Parallel minimum cuts: An improved crew pram algorithm. Master's thesis, KTH Royal Institute of Technology, 2020.
[21] Frank Olken. *Random Sampling from Databases*. PhD thesis, University of California at Berkeley, 1993.
[22] Saladi Rahul and Yufei Tao. Efficient top-k indexing via general reductions. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 277–288, 2016.
[23] Rodrygo L. T. Santos, Craig MacDonald, and Iadh Ounis. Search result diversification. *Found. Trends Inf. Retr.*, 9(1):1–90, 2015.
[24] Moritz von Looz and Henning Meyerhenke. Querying probabilistic neighborhoods in spatial data sets efficiently. In *Proceedings of International Workshop on Combinatorial Algorithms*, volume 9843, pages 449–460, 2016.
[25] A.J. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, 10, 1974.
[26] Lu Wang, Robert Christensen, Feifei Li, and Ke Yi. Spatial online sampling and aggregation. *Proceedings of the VLDB Endowment (PVLDB)*, 9(3):84–95, 2015.
[27] Dong Xie, Jeff M. Phillips, Michael Matheny, and Feifei Li. Spatial independent range sampling. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 2023–2035, 2021.