Efficient Dynamic Weighted Set Sampling and Its Extension

Fangyuan Zhang
fzhang@se.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong SAR

Mengxu Jiang
mxjiang@se.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong SAR

Sibo Wang
swang@se.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong SAR

ABSTRACT

Given a weighted set S of n elements, *weighted set sampling (WSS)* samples an element in S so that each element a_i is sampled with a probability proportional to its weight $w(a_i)$. The classic alias method pre-processes an index in $O(n)$ time with $O(n)$ space and handles WSS with $O(1)$ time. Yet, the alias method does not support dynamic updates. By minor modifications of existing dynamic WSS schemes, it is possible to achieve an expected $O(1)$ update time and draw t independent samples in expected $O(t)$ time with linear space, which is theoretically optimal. But such a method is impractical and even slower than a binary search tree-based solution. How to support both efficient sampling and updates in practice is still challenging. Motivated by this, we design *BUS*, an efficient scheme that handles an update in $O(1)$ amortized time and draws t independent samples in $O(\log n + t)$ time with linear space.

A natural extension of WSS is the *weighted independent range sampling (WIRS)*, where each element in S is a data point from $\mathbb{R}$. Given an arbitrary range $Q = [\ell, r]$ at query time, WIRS aims to do weighted set sampling on the set S_Q of data points falling into range Q. We show that by integrating the theoretically optimal dynamic WSS scheme mentioned above, it can handle an update in $O(\log n)$ time and can draw t independent samples for WIRS in $O(\log n + t)$ time, the same as the state-of-the-art static algorithm. Again, such a solution by integrating the optimal dynamic WSS scheme is still impractical to handle WIRS queries. We further propose WIRS-BUS to integrate BUS to handle WIRS queries, which handles each update in $O(\log n)$ time and draws t independent samples in $O(\log^2 n + t)$ time with linear space. Extensive experiments show that our BUS and WIRS-BUS are efficient for both sampling and updates.

PVLDB Reference Format:
Fangyuan Zhang, Mengxu Jiang, and Sibo Wang. Efficient Dynamic Weighted Set Sampling and Its Extension . PVLDB, 16(1): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

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

1 INTRODUCTION

Let S be a set of n elements, where each element a_i is associated with a positive real weight $w(a_i) \in \mathbb{R}^+$ $(1 \le i \le n)$. The *weighted set sampling (WSS)* samples element a_i from S with probability $w(a_i)/\sum_{j=1}^{n} w(a_j)$. WSS has been widely used in real applications. For instance, it is shown in existing studies on query processing, e.g., [16, 18, 46], that weighted set sampling, rather than uniform sampling, is essential for dealing with join sampling for better effectiveness; In social network analysis, personalized PageRank (PPR) is widely used for social recommendation by IT companies like Twitter [23] and Tencent [31]. They simulate random walks to estimate PPRs, and at each step, the walk randomly chooses an out-neighbor of the current node v according to their weights [31]; In online advertisement [29], an advertising system maintains a large set of ads provided by advertisers. The probability of each ad being displayed is determined by a weight assigned to this ad through the advertising system. Moreover, in reinforcement learning, replay buffer [30], a buffer pool that contains a set of historical data, plays a crucial role in learning algorithms. Each historical data is called an "experience" that consists of a tuple representing the historical interaction data gained by the reinforcement learning algorithm at a time stamp. The replay buffer then maintains the set of "experiences" and provides samples of such "experiences" for training. It is shown in existing studies that with prioritized replay buffer [40] that assigns different weights to different "experience", it helps achieve better resource utilization than uniformly sampling adopted by previous the replay buffer [30], as demonstrated in the literature. Apart from above examples, WSS is widely used in database [28, 35, 45], computer graphics [33], network analysis [12, 13], computational chemistry [20], bio-informatics [19], recommendation system [41], and so on. A classic solution for WSS is the *alias method* [43], which pre-processes set S in $O(n)$ time and $O(n)$ space and draws t independent samples in $O(t)$ time.

A natural extension of the WSS problem is the *weighted independent range sampling (WIRS)*. Over the weighted set S, given a range Q at query time, the goal of WIRS is to do a WSS on the subset S_Q of elements in S belonging to Q. WIRS has many applications. For example, in query processing, reporting the query results falling into a range can be expensive if the number of records in the answer set is huge, e.g., tens of millions of records. In such scenarios, returning a sample set from the query results becomes a better solution and has received significant attention [8, 17, 26, 42]. Besides, WIRS naturally has applications when we want to sample elements from a specific range, e.g., within a specific price range. Consider the application of the advertising system mentioned above for WSS. The advertising system may want to display the ads according to the historical purchase behavior of the targeted users. It may only display ads whose price fall into the range that the targeted users might be interested in. For WIRS, as set S_Q depends on Q at query time, we cannot directly apply the alias method to solve the WIRS

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

problem as many choices of the range Q may exist. Tao [42] proposes combining a binary search tree (BST) structure with the alias method, where the BST is used to efficiently identify the points falling into the query range Q. The solution takes linear space and can draw t independent samples with $O(\log n + t)$ running cost.

1.1 Limitations of existing solutions

Nowadays, real-world data are changing rapidly. For example, in query processing, the database gets frequently updated with newly inserted/deleted elements. In social network analysis, social networks are dynamically changing [23, 31], e.g., the following list of users changes over time. When estimating PPRs in this context, the sampling structure needs to be updated. Similarly, in online advertisement, the set of ads maintained by the advertising system will also change as advertisers want to promote new ads or withdraw old ones. Furthermore, in reinforcement learning, the historical data, i.e., "experience", in the prioritized replay buffer may dynamically change with the learning process [25, 40], such as adding new "experiences" or removing some old "experiences". Such dynamic updates require sampling schemes to support efficient updates. However, existing solutions for WSS and WIRS either cannot outperform simple baselines or focus on static data.

Limitations of existing WSS schemes. For the WSS problem, since the alias structure does not support dynamic addition and removal of elements, we need to rebuild the structure when the weighted set changes. This incurs $O(n)$ update time for each insertion/deletion. Such a cost is too prohibitive when the set is large. Another widely adopted solution is to use a balanced binary search tree (BST). By recording the sum of the weights of the nodes in each sub-tree, we can sample an element in $O(\log n)$ time. It naturally supports inserting/removing an element in $O(\log n)$ time. Existing state-of-the-art dynamic solutions for WSS [24, 32] focus on the case when the number of elements is fixed, and only the weights change. The solution in [24] limits the weights to be integers, while the solution in [32] can allow arbitrary positive real weights. By minor modifications to the solution in [32], it is possible to achieve an amortized $O(1)$ update time and draw t independent samples in expected $O(t)$ time with linear space, which is theoretically optimal.

We denote such a solution as OPT and the backbone of OPT is bucketing, which divides the weights into disjoint ranges, e.g., $[2^i, 2^{i+1})$, where i is an integer. Then, the bucket I_i includes the elements whose weight fall into $[2^i, 2^{i+1})$. When sampling from a bucket, it first samples an element e uniformly at random, and then either *(i)* accept this record with $w(e)/2^{i+1}$ probability, or *(ii)* reject this record and repeat this sampling process until a sample is accepted. We may divide the ranges with other bases, e.g., $[10^i, 10^{i+1})$, yet this will cause a higher sampling cost due to the increased probability of rejection. Hence, typically the base is set to 2 [24, 32]. With the bucketing idea alone, gaining the optimal time complexity is still difficult. To achieve the optimal complexity, they first present a lookup table solution that supports $O(1)$ update time when the weight of an element changes (no insertion or deletion supported) and $O(1)$ sampling at the sacrifice of space consumption. In particular, given x elements whose weight fall into the range $[1, m]$, they design a lookup table solution that achieves $O(1)$ time per weight update and $O(1)$ time per sampling with $O(x \cdot m^{x+1})$ space. To make full use of the lookup table, OPT combines buckets into different groups and normalizes the values so that each non-empty group has a problem of size $\lceil \log_2 n \rceil$ and the values fall into the range $[1, n^2]$. They further reduce the problem size using the group idea with one more level, making the problem size be $x = \lceil \log_2 \lceil \log_2 n \rceil \rceil$ and the values fall into the range $[1, m = \lceil \log_2 n \rceil^2]$. The space cost $O(x \cdot m^{x+1})$ can be bounded by $O(n)$, which helps achieve the final time and space complexity. Yet, OPT is even slower than the BST-based method mentioned above for both sampling and updates in practice. Besides, maintaining the lookup table with the current design will take a prohibitive space cost. It is possible to reduce the table size (more details in Sec. 2.2) but comes at the degraded sampling and update performance, making the disadvantage more pronounced. *How to gain high practical efficiency for both sampling and updates for the WSS problem is still challenging.*

Limitations of existing WIRS schemes. Hu et al. [26] proposed a dynamic solution for a special case of the WIRS query when all weights are equal. When the weights might differ, Tao [42] proposes combining the binary search tree (BST) and the alias method, where each node v in the BST maintains an alias structure for all data points falling in the subtree rooted at v. Given a range Q, it first identifies a set of $O(\log n)$ nodes, called *canonical nodes*, such that the data points falling into the subtrees rooted at these canonical nodes are disjoint and their union is precisely the set S_Q of data points falling into Q. Thus, we can first sample a canonical node v_c according to the total weight of the elements falling into the subtree rooted at v_c. Then, we sample a data point falling into the subtree rooted at v_c via an alias structure. With such an augmented BST structure, it is difficult to deal with dynamic updates. *How to effectively solve the WIRS problem in dynamic settings is still open.*

1.2 Our contribution

In this paper, we tackle the WSS and WIRS problems in the dynamic setting to support dynamic addition or removal of elements to the input set, from both theoretical and practical aspects. The contributions are summarized as follows.

An efficient dynamic WSS scheme. Similar to both OPT [32] and the solution in [24], we use the bucketing idea as the backbone and present an efficient solution *BUS (BUcket Sampling)* that can support updates in $O(1)$ time and can draw t samples in expected $O(\log n + t)$ time using linear space. Yet, different from [24, 32], we do not maintain the lookup table or reduce the problem size by two levels, which degrades the sampling efficiency. Instead, our solution greatly simplifies the data structure using a 1-level bucket without maintaining a space-consuming lookup table. We only need to insert into or delete from the corresponding bucket with $O(1)$ time to deal with updates. For sampling, the key of our solution is that we can draw a sample by checking $O(\log n)$ buckets with $1-1/n$ probability and then do rejection sampling, which assures an expected $O(\log n)$ time to draw a sample. This makes the sampling highly efficient and outperforms alternatives. The examined buckets can be reused when drawing t independent samples, and BUS achieves a sampling time of $O(\log n + t)$. Extensive experiments show that our proposed BUS is efficient for both sampling and updates.

The first theoretical result for dynamic WIRS scheme. This paper presents the first dynamic scheme for the WIRS problem that

achieves the same sampling complexity as the state-of-the-art static solution while supporting $O(\log n)$ amortized update cost. For the dynamic WIRS problem, we first need to modify the existing static solution proposed by Tao [42] so that the augmented alias structure maintained at each node is replaced with a dynamic sampling scheme, e.g., OPT or our proposed BUS scheme. Yet, a key challenge to dealing with updates for the WIRS sampling structure is that the popular height-balanced BST, e.g., red-black tree or AVL-tree, may need rotations during the updates, causing $O(n)$ changes in the nodes falling into a subtree rooted at such rotated nodes in the worst case. To tackle this issue, we combine the dynamic WSS scheme with the idea of weight-balanced BST, e.g., BB$[\alpha]$-tree [34], where the subtree is adjusted only when the weight, i.e., the number of nodes in the subtree, becomes unbalanced. This strategy can amortize the update cost to $O(s)$ nodes if an update is triggered to a subtree with s nodes. Such a solution still incurs $O(n \log n)$ space as it needs to maintain the sampling scheme at each node. To reduce the space cost, we apply a chunk-based solution, which divides the input data into different chunks, where each chunk includes $\Theta(\log n)$ records. This helps reduce the space cost to $O(n)$. If the sampling structure at each node is OPT, then we can achieve the claimed $O(t + \log n)$ sampling time to draw t samples and $O(\log n)$ amortized update time. We denote such a solution as WIRS-OPT.

A practically efficient dynamic WIRS scheme. WIRS-OPT suffers from a similar limitation to OPT and is impractical for large datasets. To overcome these limitations, we propose a new method called *WIRS-BUS* that leverages BUS as the sampling structure at each node. We further reduce the space and enhance the efficiency of our WIRS-BUS method by introducing chunk-based optimizations. Since chunks are typically stored in consecutive cache lines, scanning them for sampling is as efficient as sampling from an alias structure. This optimization considerably reduces the storage requirements without compromising sampling performance. We also present a weight update solution for the BUS structure to avoid the need for repeated element insertions and deletions when chunks do not merge or split after updates. Our proposed method, WIRS-BUS, can draw t samples with a time complexity of $O(\log^2 n + t)$, which is superior to the simple multiplication of t by the sample cost of BUS (i.e., $O(t \log n)$). This efficiency is achieved by reusing the intermediate result from the first sample for subsequent samples. In practice, when t is sufficiently large (which is often the case), our WIRS-BUS method has similar time complexity to that of WIRS-OPT and the state-of-the-art static solution [42].

Excellent sampling and update performance. We evaluate our proposed solutions for WSS and WIRS problems against existing solutions. We show that our BUS can achieve up to 10x improvement on WSS sampling and 7x-10x speed-up on updates compared to the current state-of-the-art dynamic solution on WSS. For the WIRS problem, our WIRS-BUS again achieves up to 5x improvement on sampling over the state-of-the-art dynamic solution for WIRS while consuming the same amount of space and achieving similar update efficiency. Remarkably, our WIRS-BUS achieves identical sampling performance as the state-of-the-art static scheme for WIRS, indicating that WIRS-BUS supports efficient updates without any compromise on its sampling performance.

2 PRELIMINARIES

2.1 Problem Definition

Let S be a set of n elements in $\mathbb{R}$, where each element a_i $(1 \le i \le n)$ is associated with a positive real weight $w(a_i) \in \mathbb{R}^+$. Over set S, the *weighted set sampling problem (WSS)* is defined as follows.

Definition 2.1 (WSS). Let S be an input set S of n elements in $\mathbb{R}$, where each element a_i $(1 \le i \le n)$ is associated with a positive real weight $w(a_i) \in \mathbb{R}^+$. Given an input positive integer t, the WSS returns t independent random samples from S, so that for each sample, element a_i is sampled with a probability of $w(a_i)/\sum_{j=1}^{n} w(a_j)$.

Notice that WSS is sampling with replacement. There are scenarios where we only want to sample elements falling into a specific range from the set S, which is the *weighted independent range sampling (WIRS)*. The definition is as follows.

Definition 2.2 (WIRS). Given a set S as above, an input positive integer t and a range $Q = [\ell, r]$ at query time, WIRS returns t samples via executing a WSS with input t on the set $S_Q = \{a \in [\ell, r] | a \in S\}$, i.e., the set of elements falling into range Q.

Note that the range Q and number t of samples are given at query time. This makes the problem more challenging since we aim to support efficient sampling for an arbitrary range without explicitly constructing the subset S_Q of elements falling into Q.

Dynamic setting. In this paper, we consider the dynamic setting where elements are dynamically added into S or removed from S. In the dynamic setting, the WSS or WIRS needs to be performed on the updated set after the insertion or removal of the elements.

Next, we review existing solutions for WSS and WIRS. Tab. 1 summarizes the sampling time, update time, space cost, and preprocessing time of each method, including our proposed solutions.

2.2 Existing Solutions for WSS

Alias method for WSS. Let $S = \{a_1, a_2, \cdots, a_n\}$ be a weighted set. Further let $w(S) = \sum_{a_i \in S} w(a_i)$. To solve the WSS problem, the alias method works as follows. Suppose we have n equal weight bins $B_1, B_2, \cdots, B_n$, where each bin holds an average weight $\bar{w} = w(S)/n$. Note that there exists at least one element, denoted as a_{nl}, whose weight is no larger than $\bar{w}$, and one element, denoted as a_{ns}, whose weight is no smaller than $\bar{w}$. We create two linked lists L_{nl} and L_{ns} to maintain elements with weights no larger and no smaller than $\bar{w}$, respectively. To build the alias structure, we take out an a_{nl} from L_{nl} and an a_{ns} from L_{ns}. Then, we add a pair $p_1 = (a_{nl}, w(a_{nl}))$ in the first bin B_1. We use $p_1.ele$ to indicate the element and $p_1.weight$ to indicate the weight stored in the pair. If $p_1.weight$ is smaller than $\bar{w}$, we further add a pair $p_2 = (a_{ns}, \bar{w} - w(a_{nl}))$ to B_1. Next, we update the weight of a_{ns} to $w(a_{ns}) - (\bar{w} - w(a_{nl}))$. We add a_{ns} to the appropriate list according to its weight. We repeat this process until all n bins take $\bar{w}$ weights and each bin includes at most two entries. To draw a sample, we pick a bin uniformly at random. If it contains one entry p_1, we return $p_1.ele$. Otherwise, if it has two entries p_1 and p_2, we generate a random number $r \in [0, \bar{w}]$ uniformly at random. If $r \le p_1.weight$, we return $p_1.ele$; otherwise, return $p_2.ele$. The above *alias structure* can draw t samples with $O(t)$ time with linear space.

Table 1: Comparison of different methods for WSS and WIRS

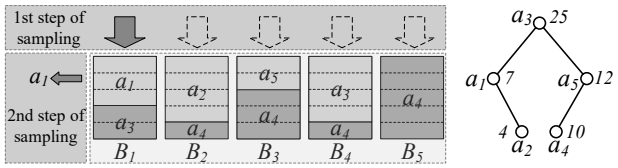
Problem	Method	The time to draw t samples	Update time	Space cost	Preprocessing time	Remark
WSS	Alias Method	$O(t)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$ update cost
	Binary Search Tree	$O(t \log n)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(\log n)$ factor for update/sampling
	OPT-Extension	$O(t)$	$O(1)$	$O(n)$	$O(n)$	Large constant factor
	BUS	$O(\log n + t)$	$O(1)$	$O(n)$	$O(n)$	-
WIRS	Tao's Method	$O(\log n + t)$	$\Omega(n)$	$O(n)$	$O(n \log n)$	$\Omega(n)$ update cost
	Xie et al.'s Method	$O(t \log n)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(\log n)$ factor for update/sampling
	WIRS-OPT	$O(\log n + t)$	$O(\log n)$	$O(n)$	$O(n \log n)$	Large constant factor
	WIRS-BUS	$O(\log^2 n + t)$	$O(\log n)$	$O(n)$	$O(n \log n)$	-

Example 2.3. Assume that we have a weighted set $S = \{a_1, a_2, a_3, a_4, a_5\}$ where the weights are 3, 4, 6, 10, 2, respectively. Then, the average weight $\bar{w} = 5$. Initially, L_{nl} includes $\{a_1, a_2, a_5\}$ and L_{ns} includes $\{a_3, a_4\}$. Then, we take a_1 from L_{nl} and a_3 from L_{ns}. Next, we add pair $p_1 = (a_1, 3)$ into bin B_1. Since the weight is still smaller than $\bar{w} = 5$. We add the second pair $(a_3, 2)$ to bin B_1. Next, we remove a_1 from the linked list L_{nl}. We move a_3 from L_{ns} to L_{nl} according to its new weight of $6-2=4$. Similarly, we can create bin $B_2 = \{(a_2, 4), (a_4, 1)\}$, $B_3 = \{(a_5, 2), (a_4, 3)\}$, $B_4 = \{(a_3, 4), (a_4, 1)\}$, and $B_5 = \{(a_4, 5)\}$. To do sampling, we first generate a random integer from $[1, 5]$. Assume that the integer is 1. Then, we generate a random number r from $[0, \bar{w}]$. Assume that $r = 1.2$. By checking elements in bin B_1, we return a_1 as the sample. The alias structure and sampling process in this example are illustrated in Fig. 1(a).

BST for WSS. An alternative method is to maintain a balanced BST for elements in S. For each node u, it maintains the total weight $w_{left}(u)$ (resp. $w_{right}(u)$) of the elements in the left (resp. right) subtree of u. If node u has no left (resp. right) subtree, then $w_{left}(u) = 0$ (resp. $w_{right}(u) = 0$). To sample an element, we start from the root u. Let the weight of the element stored at root u be $w(u)$. A random number $x \in [0, w(S)]$ is first sampled. Then, *(i)* if $w_{left}(u) < x \leq w_{left}(u) + w(u)$, we directly return the element stored at u; *(ii)* if $x \leq w_{left}(u)$ we update u as its left child and repeat from Step (i); *(iii)* otherwise, we update $x = x - w_{left} - w(u)$, update u as its right child, and repeat from Step (i). As a balanced BST has a height of $O(\log n)$, it incurs $O(t \log n)$ cost to draw t samples. Besides, it takes linear space and naturally supports the insertion/deletion of an element with $O(\log n)$ cost.

Example 2.4. With the same set S in Example 2.3, we can use a BST to maintain S. Assume that a_1 to a_5 are in ascending order. Fig. 1(b) shows the corresponding BST structure. Suppose we generate a random number $x = 23.5$ from $[0, 25]$. Firstly, we check root a_3, we find $x > w(a_3) + w_{left}(a_3)$. Then we go to the right child of a_3 and update x to 10.5. For the node a_5, we find $w_{left}(a_5) \leq x \leq w(a_5)$. Thus, we return a_5 as the sample result.

Dynamic WSS schemes. In the literature, there exist studies [24, 32] that aim to solve the dynamic WSS problem in different settings. Both solutions mainly combine the idea of bucketing and lookup table to achieve the desired time complexity. They use the idea of bucketing to reduce the problem size, say from $O(n)$ to $O(\log n)$. By reducing the problem size twice, the problem size becomes sufficiently small and they apply a lookup table solution to use a larger space to support $O(1)$ sampling and update complexity. As we mentioned earlier in Section 1, given x elements that fall into the range $[1, m]$, the lookup table solution achieves $O(1)$ time per weight update and $O(1)$ time per sampling with $O(x \cdot m^{x+1})$ space.



(a) Alias method (b) BST

Figure 1: Examples of WSS solutions

After reducing the size of the problem twice, $m = \lceil \log_2 n \rceil^2$ and $x = \lceil \log_2 \lceil \log_2 n \rceil \rceil$. The space cost of the lookup table is bounded by $O(n)$. Yet, such a lookup table solution incurs a huge space cost under its default setting. For example, when $n = 10^5$, we have that $m = \lceil \log_2 n \rceil^2 = 289$, $x = \lceil \log_2 \lceil \log_2 n \rceil \rceil = 5$, and the lookup-table requires $x \cdot m^{x+1} = 2.9 \times 10^{15}$ entries. Assume that each entry takes a byte, then OPT already takes more than 1PB to store such a table. It is possible to reduce the table size by reducing the problem size, say to $\lceil \log_2 \lceil \log_2 \lceil \log_2 n \rceil \rceil \rceil$, or taking a larger bucket base, e.g., taking a base of 10 so that the buckets have ranges of $[10^i, 10^{i+1})$. But either solution will increase the sampling cost due to a more complicated sampling scheme or a higher rejection probability inside each bucket, making the disadvantage of sampling and updating more pronounced.

The above solution assumes that the number of elements is fixed to be n and only the weights of elements may change. It is not difficult to come up with a dynamic rebuilding solution that supports updates. In particular, given a set of n elements, we first create a sampling structure in [32] for the problem size of $2n$, where n of them have a weight of zero. Then, if a new element e is inserted, we may simply update the weight from 0 to $w(e)$. When the number of elements in the set becomes $2n$, we rebuild the whole structure. Besides, if the number of elements is less than $n/2$, we rebuild the sampling structure in [32] for the problem size of n. This can amortize the $O(n)$ reconstruction cost to $\Theta(n)$ elements, making the amortized insertion/deletion to be $O(1)$.

2.3 Existing Solutions for WIRS

Tao's method for WIRS. To deal with the WIRS problem, Tao presents a method [42] that combines the BST and the alias method to draw t independent samples with $O(\log n + t)$ time for a given query range Q. Following [42], to facilitate the range query, we consider a BST $\mathcal{T}$ that stores data points at leaf nodes, and every internal node has two children. The key of a leaf node is exactly the value of the data point in $\mathbb{R}$. The key of an internal node u is the smallest leaf key in the right subtree of u. The advantage of such a key choice is that it can efficiently identify the range of the data points (leaf nodes) falling into the subtrees. For example, consider

Fig. 2. Given the internal node with key 5 and the information from its parent, we know that the range of data points falling into the right-subtree of node 5 is $[5, 8)$ and the range of data points falling into the left-subtree of node 5 is $(-\infty, 5)$. Let L_Q denote the set of leaf nodes in $\mathcal{T}$ that has a key falling into the range Q and $L(u)$ denote the set of leaf nodes in the subtree rooted at u. Given an arbitrary query range Q, an important property of BST [42] is that we can identify a set C of canonical nodes in the BST such that *(i)* for any two different nodes u and v in C, $L(u) \cap L(v) = \emptyset$; *(ii)* the union of leaf nodes of the subtrees rooted at canonical nodes is exactly the set of leaf nodes fulfilling Q, i.e., $\cup_{u \in C} L(u) = L_Q$.

Given the above facts, Tao proposes maintaining an alias structure for each node u for all points stored in the subtree rooted at u. This, in total, incurs $O(n \log n)$ space as each point is stored in at most $O(\log n)$ nodes. Then, to handle a WIRS query, it first derives the set C of $O(\log n)$ canonical nodes and derives an alias structure on the fly for these $O(\log n)$ nodes according to the sum of weights of the points in each sub-tree. This incurs $O(\log n)$ running cost. Then, it draws a sample according to the alias structure built on the fly to choose the subtree of the canonical node and uses the alias structure maintained at each node to sample a data point in the subtree. Clearly, this returns a correct sample for the WIRS. Given a range Q, it can draw t independent samples for the WIRS problem with $O(\log n + t)$ time using $O(n \log n)$ space. By storing $O(\log n)$ points as a chunk [26], it reduces the space to $O(n)$. Yet, it is a static method and cannot be easily adapted to a dynamic setting.

Example 2.5. Consider the binary search tree $\mathcal{T}_1$ in Fig. 2. Assume that the query range $Q = [3, 11]$. Then, we have a set C of four canonical nodes $C = \{u_1, u_2, u_3, u_4\}$ (colored in black) such that the leaf nodes of the subtree rooted at u_1, u_2, u_3, u_4 are disjoint and their union is exactly the leaf nodes (thus the data points) falling into the range $Q = [3, 11]$. To do WIRS with range $Q = [3, 11]$, we first construct an alias structure T_C for C, where the weight of each node is the sum of the weights of the elements in the corresponding subtree. For example, the weight of u_2 is the sum of weights of elements 5, 6, and 7. To draw a sample, it first uses T_C to draw a canonical node. Assume that it is u_2. Then, we use the alias structure maintained at node u_2 to sample an element from $\{5, 6, 7\}$. Assume that it is 6. Then 6 is returned as the sample.

Xie et al.'s method for WIRS. To gain a better trade-off between space cost and sampling cost in practice, Xie et al. [45] propose to use a dyadic tree to do both sampling and dealing with the range. In particular, similar to Tao's method, they first use the dyadic tree to find the set C of $O(\log n)$ canonical nodes and their corresponding subtrees. Next, an alias structure is built on the fly for the $O(\log n)$ canonical node in C. It first samples a canonical node via the alias structure, then uses the binary search tree method similar to WSS (Section 2.2) to sample a node at this BST, which takes $O(\log n)$ time. To summarize, it takes $O(t \log n)$ time to draw t independent samples. But the dyadic tree is static and cannot be extended to the dynamic setting. To remedy this, it is easy to use a balanced BST to extend to the dynamic setting and use the same strategy as mentioned above. Then, it can draw t independent samples with $O(t \log n)$ running time and update with $O(\log n)$ running time for each insertion/deletion. Yet such a solution is still inferior in terms of sampling cost, as we will show in our experiment. One may further consider using LSM-tree [38] to do insertions and only mark the deleted elements as mentioned in [45]. In such cases, the sampling will still consider the weight of deleted elements during the rejection sampling. For instance, consider n elements where there is a *heavy* element with a weight to be n^2 while other $n-1$ elements are *light* elements with a weight of 1. If we delete the heavy element, we will still sample with the total weight $n^2 + n - 1$. Then, it will reject with $n^2/(n^2 + n - 1)$ probability and need expected $O(n)$ sampling time, which is too expensive for large-scale data.

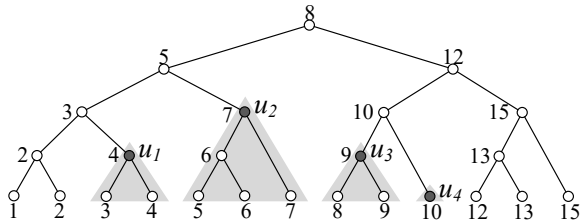


Figure 2: A Binary Search Tree $\mathcal{T}_1$

Example 2.6. Still consider the same set of elements in Example 2.5 and the same query range $Q = [3, 11]$. This approach also first finds the set $C = \{u_1, u_2, u_3, u_4\}$ of canonical nodes and constructs an alias structure T_C as well. To draw a sample, it first uses T_C to draw a canonical node. Assume that it is u_2. It then samples an element from the set $\{5, 6, 7\}$, i.e., the set of elements in the subtree rooted at u_2. It uses the BST rooted at u_2 to do WSS from $\{5, 6, 7\}$. Assume that 7 is sampled. Then, 7 is returned as the final sample.

2.4 Related Work

Sampling from a weighted set is a fundamental problem widely used in many applications. Some research works, e.g., [15], consider how to reduce the space for the WSS problem. A lot of works consider the problem based on different scenarios such as parallel sampling [4], distributed sampling [27, 28] or data stream [14, 21].

Independent range sampling is a classic problem in the database community and has been studied for decades. Olken et al. have studied this problem based on B^+-tree more than 20 years ago [37]. They also further study this problem in the high-dimensional case based on R-tree [36] and present a detailed survey [37] for this problem. From a theoretical perspective, there are many works [6, 7, 26] that design different sampling structures. Most of these works are based on low-dimensional or static weighted data. There is a lot of work [9–11, 44, 45] to study the sampling of high-dimensional data from the perspective of the application. Xie et al. [45] also study the sampling method of weighted data. Tao [42] points out that the main idea of [45] is to utilize the tree structure of high-dimensional data through DFS traversal. A high-dimensional query will eventually be transformed into multiple one-dimensional queries.

3 DYNAMIC WEIGHTED SET SAMPLING

Our dynamic WSS scheme is inspired by the bucketing strategy as we mentioned in Section 1 where we divide the weights into different buckets. Unlike existing solutions that maintain multiple levels of buckets and require a lookup table with a large space cost, our solution relies on a single-level bucket to handle all sampling tasks. However, a major challenge of the bucket-based approach is the potential existence of $O(n)$ non-empty buckets in the worst case.

Algorithm 1: BUS-Construction(S)

1 Initialize H_B and H where H_B is for non-empty buckets maintenance and H is used for elements;
2 **for** *each* $a \in S$ **do**
3 $\quad i \leftarrow \lfloor \log_2 w(a) \rfloor$;
4 $\quad$ **if** *i exists in* H_B **then** $w(I_i)$ += $w(a)$;
5 $\quad$ **else**
6 $\quad\quad$ Add i into H_B and set $w(I_i) = w(a)$.
7 $\quad\quad$ Create a dynamic array A_i for bucket I_i;
8 $\quad$ **end**
9 $\quad$ Insert a into A_i and let p_a be the position of a in A_i;
10 $\quad$ Add $\langle a, p_a \rangle$ into H;
11 **end**
12 **return** BUS structure;

Algorithm 2: BUS-Sample(BUS structure)

1 Use Lemma 3.3 to locate the largest non-empty bucket and record the bucket id as r.
2 $w_I \leftarrow Rand(0, w(S))$, $flag \leftarrow 0$, $i \leftarrow r, w_{\le r+1}(S) \leftarrow w(S)$;
3 **for** *i from r down to* $r - 2 \cdot \lceil \log_2 n \rceil$ **do**
4 $\quad$ **if** $w_I \ge w_{\le i}(S) - w(I_i)$ **then** $flag \leftarrow 1$, break;
5 $\quad w_{\le i}(S) = w_{\le i+1}(S) - w(I_i)$;
6 **end**
7 **if** $flag = 0$ **then** find I_i by scanning all non-empty buckets ;
8 **while True do**
9 $\quad x \leftarrow Rand(0, 2^{i+1})$, sample a point a uniformly from I_i;
10 $\quad$ **if** $x \le w(a)$ **then return** a;
11 **end**

Our analysis shows that it is possible to examine $O(\log n)$ buckets with a probability of $1 - 1/n$. Even in the scenario where we need to examine all buckets, the cost remains at $O(n)$ for a remaining probability of $1/n$. This helps to bound the expected sampling cost to $O(\log n)$. Next, we explain the details of our BUS scheme.

BUS structure and its construction. Firstly, we divide the domain of the weights into disjoint ranges $[2^i, 2^{i+1})$ where i is an integer that can be negative. We further maintain a bucket I_i to include all the elements whose weights fall into the range $[2^i, 2^{i+1})$ and let $w(I_i)$ be the sum of the weights of elements falling into bucket I_i. Notice that $w(I_i)$ can be maintained easily even when there exist updates to elements in bucket I_i. Similarly, we can easily maintain $w(S)$ during the update. There might exist an infinite number of buckets, but at most n of them are non-empty. To maintain non-empty buckets, we use a hash table, denoted as H_B, to manage non-empty buckets by recording IDs of non-empty buckets as keys. Such a hash table can use dynamic perfect hashing, e.g., cuckoo hashing [39], to achieve $O(1)$ time to insert/delete an arbitrary record by key in linear space. With this hash table, we can locate a non-empty bucket by ID, record a new non-empty bucket via its ID, and delete an empty bucket in $O(1)$ time.

For each bucket I_i, we further maintain a dynamic array (e.g., vector in C++), denoted as A_i, to store elements with weights falling into range $[2^i, 2^{i+1})$. A dynamic array supports *(i)* inserting a new element after the last element or deleting the last element with amortized $O(1)$ time, *(ii)* accessing the i-th element in $O(1)$ time. However, this is still insufficient. To explain, the dynamic array can only delete the last element in $O(1)$ time, but the deletion of an element may occur at an arbitrary position in the dynamic array. To support the removal of an arbitrary element from A_i, we further maintain a hash table H_i to record the position of each non-empty element in A_i. Then, we can efficiently deal with deletions of elements by swapping the last element and the deleted element in A_i and updating the positions in H_i to guarantee that the elements are consecutive in A_i. This allows us to *(i)* insert/delete elements to I_i in $O(1)$ time; *(ii)* sample an element in I_i uniformly at random. For bucket I_i, we maintain a counter $w(I_i)$ to store the total weight of the elements in this bucket, which will be used during the sampling. Notice that we can actually maintain only one hash table for all elements instead of maintaining a hash table H_i for elements falling into I_i since the elements in S are different by default and thus will not result in duplicate keys even using only one hash table.

Example 3.1. Consider a weighted set $S = \{a_1, a_2, a_3, a_4, a_5, a_6\}$ where weights are 0.01, 0.1, 4, 6, 100, and 2, respectively. Then, we have 5 non-empty buckets: $I_{-7} = \{a_1\}, I_{-4} = \{a_2\}, I_1 = \{a_6\}, I_2 = \{a_3, a_4\}, I_6 = \{a_5\}$. The IDs of the buckets are $-7, -4, 2, 6, 1$. We insert these IDs of non-empty buckets into the hash table H_B. For each bucket I_i, we maintain a dynamic array A_i and use the hash table H to maintain the position information. For example, for bucket I_2, we maintain a dynamic array A_2 to store a_3 and a_4. Then, we further maintain $H[a_3] = 0$ and $H[a_4] = 1$, indicating that a_3 (resp. a_4) is stored at position 0 (resp. position 1) in A_2.

Given the details of our sampling structure, we show how to construct the BUS scheme in Algorithm 1. We first initialize the hash table H_B for maintaining the non-empty buckets and hash table H to record the position of each element a in its corresponding bucket (Line 1). Then, we go through each element a in S and find the corresponding bucket I_i (Lines 2-3). If I_i is still empty, we add bucket ID i into H_B, set $w(I_i) = w(a)$, and initialize a dynamic array A_i to record elements falling into I_i (Lines 6-7); otherwise, we simply update the weight of bucket I_i. Next, we add element a into A_i, record the position p_a of a in A_i, and store p_a in H (Lines 9-10). When all elements are processed, it returns the BUS structure.

BUS sampling. Next, we explain how to do sampling with the BUS scheme. Given a bucket I_i with a larger ID than bucket I_j (i.e., $i > j$), there is no guarantee that $w(I_i) < w(I_j)$ since there might exist more elements in bucket I_j and thus contributes to a higher total weight. But still, we can make full use of such ID information to do efficient sampling. Let r be the largest ID of non-empty buckets. For instance, in Example 3.1, $r = 6$ since I_6 has the largest ID of all non-empty buckets. Now, we assume that r is known in advance and will later clarify how to derive r in $O(\log n)$ time.

Given the largest ID r of the non-empty buckets, we then only consider the preceding buckets with IDs $r, r-1, \cdots$, down to $r - 2 \cdot \lceil \log_2 n \rceil$. We have the following lemma to show that by examining these $2\lceil \log_2 n \rceil + 1$ buckets, the total weights of these buckets account for at least $n/(n+1)$ of the total weight of set S.

Lemma 3.2. *If the largest ID of the non-empty buckets is r, the sum of the weight of all elements falling into buckets with IDs smaller than $r - 2 \cdot \lceil \log_2 n \rceil$ is at most $\frac{1}{n+1} \cdot w(S)$.*

Algorithm 3: BUS-Update(BUS structure, a)

1 Find the bucket ID i where a is to be inserted/deleted;
2 Insert/delete a in A_i and hash table H;
3 Update the weight of bucket I_i in H_B and remove bucket I_i from H_B if it becomes empty;
4 Update the total weight $w(S)$ and **return** the BUS structure;

Proof. Consider the extreme case and assume that only one element is in I_r and its weight is 2^r and all the remaining elements are in buckets with ID no larger than $r - 2 \cdot \lceil \log_2 n \rceil - 1$. Then their weight must be less than $(n-1) \cdot 2^{r-2\cdot\lceil \log_2 n \rceil}$. Then we know the ratio of their weights as:

$$\frac{(n-1)\cdot 2^{r-2\cdot\lceil \log_2 n\rceil}}{2^r} = \frac{n-1}{2^{2\cdot\lceil \log_2 n\rceil}} \leq \frac{n-1}{n^2} \leq \frac{1}{n}.$$

Since $w(S) \leq (n-1)\cdot 2^{r-2\cdot\lceil \log_2 n\rceil} + 2^r$, we prove this lemma. □

Given Lemma 3.2, it is easy to verify that we have at least $1 - 1/n$ probability to sample a bucket from IDs $2\lceil \log_2 n \rceil + 1$ to r. We then design our sampling algorithm as shown in Algo. 2. It first derives the largest ID r of non-empty buckets (Line 1). Next, it generates a random number $w_I \in [0, w(S)]$ uniformly at random. Then, it samples in two steps. It first samples the bucket according to the weights of the buckets (Lines 1-7). After sampling the bucket, it then samples an element in the bucket with rejection sampling (Lines 8-11). To identify the sampled bucket by the random number w_I, it finds the bucket with ID i so that:

$$w_{\leq i-1}(S) = \sum_{j=-\infty}^{i-1} w(I_j) < w_I \leq \sum_{j=-\infty}^{i} w(I_j) = w_{\leq i}(S). \quad (1)$$

To find the ID i, it scans from the bucket with ID r down to $r - 2\lceil \log_2 n \rceil$. We maintain $w_{\leq i}(S)$ as the total sum of the weights of all elements falling into buckets with ID no larger than i. If we find the ID $i \in [r-2\lceil \log_2 n \rceil, r]$ such that it satisfies Equation 1, then we stop the search and mark a flag to indicate that i falls in $[r-2\lceil \log_2 n \rceil, r]$ (Lines 3-6). Otherwise, we will check all non-empty bucket IDs via a brute-force approach and identify the bucket I_i, which takes $O(n)$ cost (Lines 7). However, notice that this extreme case only happens with at most $1/n$ probability, making the expected cost of finding the correct bucket still bounded by $O(\log n)$ cost. After locating the bucket I_i, we then randomly sample an element falling into bucket I_i. We do rejection sampling according to the weight of the sampled element a. In particular, we accept a with probability $w(a)/2^{i+1}$. Since $w(a) \in [2^i, 2^{i+1})$, it incurs $O(1)$ trials in expectation. As we can sample an element from the bucket I_i with constant time via the dynamic array A_i, the cost to sample an element from bucket I_i incurs an expected $O(1)$ cost. Adding the cost to find the largest ID r of non-empty buckets ($O(\log n)$ cost), the cost to sample a bucket ($O(\log n)$ cost), and the cost to sample an element from the sampled bucket ($O(1)$ cost), the total cost to sample an element can be bounded by $O(\log n)$. If we need to draw t samples, we can build an alias structure for the non-empty buckets among all buckets with IDs in $[r - 2\lceil \log_2 n \rceil, r]$. Then, we can identify the bucket i with $O(1)$ cost. By this strategy, if we want to draw t samples, we can then finish in $O(\log n + t)$ time.

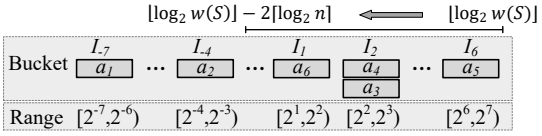


Figure 3: An example of BUS scheme

It remains to clarify how to find the largest ID r of non-empty buckets. The key is that given the total weight $w(S)$ of all elements in S, we can directly identify the largest ID r of non-empty buckets by examining buckets starting from $r' = \lfloor \log_2 w(S) \rfloor$ down to $r' - \lceil \log_2 n \rceil$. The claim is summarized as the following lemma.

Lemma 3.3. *Given the weight $w(S)$ of all elements in S, let $r' = \lfloor \log_2 w(S) \rfloor$. Then the largest ID r of non-empty buckets must fall within the range of $[r' - \lceil \log_2 n \rceil, r']$.*

Proof. We prove this by contradiction. Assume that the largest ID of non-empty buckets is larger than r'. Then as long as there is an element in $I_{r'+1}$, the weight of this bucket will satisfy that:

$$w(S) \geq 2^{r'+1} = 2^{\lfloor \log_2 w(S)\rfloor + 1} > 2^{\log_2 w(S)} > w(S),$$

which derives a contradiction. Then assume that all buckets in the range $[r' - \lceil \log_2 n \rceil, r']$ are empty, i.e., $\forall i \in [r' - \lceil \log_2 n \rceil, r'], I_i = \emptyset$. Even if all the elements fall into the bucket with ID $r' - \lceil \log_2 n \rceil - 1$, the upper bound of the total weight obeys the following inequality:

$$w(S) < n \cdot 2^{r' - \lceil \log_2 n \rceil} = n \cdot \frac{2^{r'}}{2^{\lceil \log_2 n \rceil}} \leq 2^{r'} \leq w(S).$$

This makes a contradiction and finishes the proof. □

Next, we show an example of how to draw a sample with BUS.

Example 3.4. Still consider S in Example 3.1 with 5 buckets $I_{-7} = \{a_1\}, I_{-4} = \{a_2\}, I_1 = \{a_6\}, I_2 = \{a_3, a_4\}, I_6 = \{a_5\}$ that are non-empty. Besides, $w(I_{-7}) = 0.01$, $w(I_{-4}) = 0.1$, $w(I_1) = 2$, $w(I_2) = 10$, $w(I_6) = 100$, and $w(S) = 112.11$. The corresponding BUS structure is shown in Fig. 3. To draw a sample, we generate a random number w_I from $[0, 112.11]$ and assume that $w_I = 10$. We first find r by using Lem. 3.3. It starts from ID $\lfloor \log_2 w(S) \rfloor = \lfloor \log_2 112.11 \rfloor = 6$. By checking the hash table H_B, we find that bucket I_6 is non-empty, and we derive that $r = 6$. Next, we check which ID satisfies Equation 1 for IDs falling into $[6 - 2\lceil \log_2 6 \rceil, 6] = [0, 6]$ in decreasing order of IDs. Thus, we check I_6 down to I_0 and find that ID 2 satisfies that $w_{\leq 1}(S) = 2.11 < w_I = 10 \leq w_{\leq 2}(S) = 12.11$ and thus bucket I_2 is sampled. Next, we randomly sample an element from I_2. Say a_3 is sampled. Then, a_3 is accepted with probability $w(a_3)/2^{2+1} = 4/8$. If it is rejected, we re-sampled from I_2 until an element is accepted.

BUS update. Finally, we explain how to handle updates with BUS scheme. Algo. 3 shows the pseudo-code of the update of BUS scheme. It first identifies the bucket i of element a to be inserted/deleted. This can be done in $O(1)$ cost by directly computing $\lfloor \log_2 w(a) \rfloor$. Then, we insert/delete the element in the dynamic array A_i and update the hash table H for the position. Next, the weight of bucket I_i is updated, and it is removed from the hash table H_B if I_i becomes empty. Lastly, the weight $w(S)$ of set S is updated. Every step incurs only constant cost, making the total amortized update cost to be $O(1)$. We have Theorem 3.5 to summarize our BUS scheme.

Theorem 3.5. *Given a weighted set S, the BUS scheme supports each insertion/deletion in amortized $O(1)$ time and takes $O(\log n + t)$ time to draw t independent samples for WSS queries with linear space.*

4 DYNAMIC WIRS

Recap that the state-of-the-art static solution for the WIRS problem proposed by Tao [42] is to combine the alias structure with the binary search tree (BST). The solution can draw t independent samples in $O(\log n + t)$ time. With the dynamic WIRS schemes like OPT and BUS, a straightforward idea is to replace the alias structure with OPT or BUS that are more efficient on updates. Yet, a key trouble with such a solution, as we mentioned in Section 1, is that a rotation of the height-balanced BST, e.g., AVL-Tree or Red–Black tree, may cause $O(n)$ changes of the elements under a subtree. To tackle this issue, we combine the idea of weight-balanced BST, e.g., BB[α]-tree [34] and the scapegoat tree [22], where the subtree is adjusted only when the weight, i.e., the number of nodes in the subtree, becomes unbalanced. By this strategy, we can amortize the update cost to $O(s)$ nodes if we do an update on a subtree with s nodes. As we will show, by first combining the above BST with OPT or BUS structure, we can bound the amortized update time to $O(\log^2 n)$. By further dividing elements into chunks, we can reduce the amortized update time to $O(\log n)$ with OPT and BUS.

4.1 Basic Solution

Dynamic WIRS index scheme. Similar to the method in [42], our WIRS index scheme also uses the BST to deal with the range $[\ell, r]$ efficiently. For ease of exposition, we use the following terminologies in BST. Let $\mathcal{T}$ be a BST built for a set S of n elements. Following [42], elements in the BST are only stored in leaf nodes. Besides, every internal node u in $\mathcal{T}$ has two children, and we use $u.left$ (resp. $u.right$) to indicate the left child (resp. right child) of node u. The key of an internal node u is the smallest leaf key in the right subtree of u. We use $size(u)$ to denote the number of leaf nodes (also the number of elements stored) in the subtree rooted at u. Besides, recap from Section 2.3 that given a range $Q = [\ell, r]$, it can be decomposed into a set of $O(\log n)$ canonical nodes (Ref. to Example 2.5) such that the leaf nodes of the subtree rooted at these canonical nodes are disjoint. Their union is exactly the leaf nodes falling into the range Q. In [42], it maintains an alias structure at each internal node u to sample a leaf node from the subtree rooted at u. As the alias structure does not support updates, we replace it with OPT or BUS to support more efficient updates.

Next, we show how to combine the idea of weight-balanced BST to bound the amortized update cost. The nodes in the weight-balanced BST $\mathcal{T}$ satisfy the following balancing condition.

Definition 4.1 (Balancing condition). Given a constant $\alpha \in [0.7, 1)$, for any non-leaf node u, either $\max\{size(u.left), size(u.right)\} \leq \alpha \cdot size(u)$ or $size(u) \leq 3$ must hold.

If a node u does not satisfy the balancing condition, we call it *unbalanced*. If all nodes satisfy the balancing condition, we have the following lemma to bound the height of $\mathcal{T}$:

Lemma 4.2. *The height of $\mathcal{T}$ is $O(\log n)$.*

Algorithm 4: Dynamic-WIRS-Construction(S)

```
1  root(T) = BuildBST(S, 1, |S|);
2  return the WSS-augmented BST root(T);
3  procedure BuildBST(S, l, r):
4      Create a new node u for S' = {a_i ∈ S | l ≤ i ≤ r};
5      Create a WSS structure W(u) for set S' and add to u;
6      if l == r then return u;
7      u.key ← a_{⌊(l+r)/2⌋+1};
8      u.left ← BuildBST(S, l, ⌊(l+r)/2⌋);
9      u.right ← BuildBST(S, ⌊(l+r)/2⌋ + 1, r);
10     return u;
```

This is easy to derive as for any node u, we know $size(u.left) < \alpha \cdot size(u)$ and $size(u.right) < \alpha \cdot size(u)$. The size of each subtree decreases exponentially. So the height of $\mathcal{T}$ is bounded by $O(\log n)$.

To summarize, our dynamic WIRS index is an augmented version of the above weight-balanced BST, where we attach a OPT or BUS scheme at each internal node. In the following, we may directly use WSS-augmented BST to refer to our dynamic WIRS index.

WIRS index construction. We consider constructing the WSS-augmented BST for an ordered set $S = \{a_1, a_2, \ldots, a_n\}$ with n elements. If the set S is not sorted, we can first sort the data based on the value of each element in S (not the weight of the element), with $O(n \log n)$ time. Alg. 4 shows the pseudo-code of the index construction. Note that for an ordered set S, we can build the WSS-augmented BST recursively (Lines 3-10): *(i)* build a OPT or BUS scheme for set S and attach it to root u (Line 5); *(ii)* choose the middle element and set it as the key of root u (Line 7); *(iii)* construct a BST for all elements smaller than $u.key$ and set it as the left subtree of u (Lines 8); *(iv)* construct a BST for all elements no smaller than $u.key$ and set it as the right subtree of u (Line 9). When there is only one element in the set, we set it as a leaf node (Line 5). Note that the cost of building the whole BST (without considering the WSS structure at each internal node) can be bounded by $O(n)$ if the ordered set is maintained by an array or a BST. Adding the cost of WSS structure construction at each internal node, it incurs $O(n \log n)$ cost. Note that Algorithm 4 can be used to reconstruct any unbalanced subtrees rooted at an internal node.

Lemma 4.3. *Given a weighted set S of size n, the dynamic WIRS index can be built in $O(n \log n)$ time and takes $O(n \log n)$ space.*

Proof. For a set S with size n, we first spend $O(n \log n)$ time to sort it. Then we use Algo. 4 to build the WIRS index for the set, where it takes $O(n)$ time to create the WSS structure for the corresponding node (Line 3 of Algorithm 4). Next, we split the set into two sets of size at most $\lceil n/2 \rceil$ and recursively create WIRS index until the set size of each node is 1. It is easy to know that the total time cost is limited to $O(n \log n)$. Next, we consider the space cost of the WIRS index. Since the element of each leaf node appear in the WSS structure of all ancestor nodes from the root to the corresponding leaf node, whereas the height of the tree is $O(\log n)$ by Lemma 4.2, each node is added to the WSS structure of at most $O(\log n)$ internal nodes. As the space of sample structures OPT and BUS are all linear, the total space cost is limited to $O(n \log n)$. □

Example 4.4. Consider an ordered set $S = \{a_1 = 1, a_2 = 3, a_3 = 8, a_4 = 9\}$. We first get the middle position $\lfloor (1+4)/2 \rfloor + 1 = 3$ and set a_3 as the key of root u. Then, we build a OPT or BUS structure for the set S and add it to u. Next, it splits S into two sets: $\{a_1, a_2\}$ (resp. $\{a_3, a_4\}$) with elements smaller (resp. no smaller) than a_3. Then, it builds a BST for $\{a_1, a_2\}$ (resp. $\{a_3, a_4\}$) and set it as the left (resp. right) subtree of u. The final BST is shown in Fig. 4(a). The WSS structure for each internal node is omitted for simplicity.

WIRS sampling. Algo. 5 shows the pseudo-code of how to do sampling with the dynamic WIRS index. Given a range $Q = [\ell, r]$, we first find a set C of canonical nodes with size $O(\log n)$ (Line 2). These nodes in C satisfy that the elements contained in their subtrees are disjoint, and the union of elements (leaf nodes) in the subtrees rooted at these canonical nodes is exactly the set of elements falling into $[\ell, r]$. Here we show the pseudo-code of finding the set C of canonical nodes (Lines 9-13), which recursively identifies the ranges corresponding to an internal node u from its parents and checks if it is a subset of the given range $Q = [\ell, r]$. If it is, then we add the internal node to the canonical set C and return. Otherwise, we turn to the left subtree and right subtree to find the canonical nodes in a recursive manner.

After finding the set C of canonical nodes, we can then construct an alias structure T_C of size $O(\log n)$ for set C where the weight of each canonical node u is the sum of weights of leaf nodes in the subtree rooted at u (Line 3). We then use this T_C to handle all t independent samples. For each sample, we first use the alias structure T_C to return a canonical node u (Line 5). Then, we use the WSS sample structure $W(u)$ maintained at u to draw a sample of an element (Line 6), which takes $O(1)$ time when the sample structure is *OPT*. Since finding these nodes and building T_C takes $O(\log n)$ time (and then can be reused for these t samples), the total time to draw t independent samples given range Q is $O(\log n + t)$ for OPT. By replacing OPT with BUS, it takes $O(\log n)$ time to find the largest ID r of non-empty buckets and build an alias structure for the $2\lceil \log n \rceil$ buckets for each first sampling drawn from the same canonical node u. Then, the subsequent sampling at canonical node u can reuse the largest ID r and the built alias structure at u. Hence, later samples at u only take $O(1)$ time. Since the total time to initialize r and the alias structure for each canonical node takes $O(\log n)$ time, and we have $O(\log n)$ canonical nodes, the total initialization cost is at most $O(\log^2 n)$. Then, the time to draw t independent samples is bounded by $O(\log^2 n + t)$ with BUS. With the above analysis, we have Theorem 4.5 for the sampling time.

Theorem 4.5. *The above balanced BST $\mathcal{T}$ with OPT (resp. BUS) as the WSS sample structure at each internal node of $\mathcal{T}$ can draw t independent samples in $O(\log n + t)$ time (resp. $O(\log^2 n + t)$).*

Insertion with dynamic WIRS index. Algo. 6 shows the pseudo-code for dealing with insertions. We iterate through the root to find the position where the new element should be inserted (Lines 3-6). Notice that along with each internal node v visited during the search, it inserts the new element e into the WSS maintained at v (Line 3). It further maintains a list L of visited internal nodes (Line 4). We use L to identify unbalanced internal nodes for possible reconstruction. When we find the position where e is to be inserted, either the left child or the right child of v returned after the search, we do not directly insert there. To explain, we must keep all data at the leaf node. To achieve this, we create a new internal node x with a key to be the larger one of $v.key$ and e (Line 8). Then, it adds $v.key$ and e as its left/right child according to which one is larger.

Algorithm 5: Dynamic-WIRS-Sample($\mathcal{T}, t, [l, r]$)

```
1  C ← ∅, R ← ∅;
2  FindCanonicalNodes(root(T), l, r, [S.min, S.max]);
3  Create alias structure T_C for the set C of canonical nodes;
4  for i from 1 to t do
5  |  Use T_C to sample a canonical node u;
6  |  Use W(u) to sample a data point x and add x to R;
7  end
8  return R;
9  procedure FindCanonicalNodes(u, l, r, range_u):
10 |  if range_u ⊆ [l, r] then C.append(u) and return;
11 |  if range_u ∩ [l, r] = ∅ then return;
12 |  FindCanonicalNodes(u.left, l, r, [range_u.left, u.key));
13 |  FindCanonicalNodes(u.right, l, r, [u.key, range_u.right]);
```

Algorithm 6: Dynamic-WIRS-Insertion($\mathcal{T}, e$)

```
1  v ← root(T), L ← ∅;
2  while v.left ≠ ∅ do
3  |  Insert e in the WSS structure W(v) at internal node v;
4  |  L.append(v);
5  |  if e.key < v.key then v = v.left;
6  |  else v = v.right;
7  end
8  Replace v with an internal node with v.key and e as children;
9  for u in order of the list L do
10 |  if u is unbalanced then
11 |  |  Use Algorithm 4 to rebuild the subtree rooted at u;
12 |  |  break;
13 |  end
14 end
15 return root(T);
```

After completing the insertion of e, we traverse L to find the unbalanced node u closest to the root (Lines 9-14). If no such u exists, the insertion is done. Otherwise, we first get the ordered set S_u corresponding to elements in the subtree rooted at u in $O(size(u))$ time. Then we use Algo. 4 to rebuild the subtree rooted at u in $O(size(u))$ time (Line 11). After the reconstruction, there are no unbalanced nodes, and the insertion is finished. We have the following lemma for the amortized cost of insertion.

Lemma 4.6. *$\mathcal{T}$ handles each insertion in $O(\log^2 n)$ amortized time.*

Proof. We find the position to insert the element e in $O(\log n)$ time. Next, it takes $O(\log n)$ time to update $W(v)$, either OPT or BUS scheme, for all nodes on the path from the root to the new internal node that is the parent of newly inserted e. Next, consider the impact of tree reconstruction. When a node u is unbalanced, it takes Algo. 4 to reconstruct the subtree rooted at u in $O(size(u) \cdot \log(size(u)))$ time. If the subtree rooted at u is reconstructed, it can

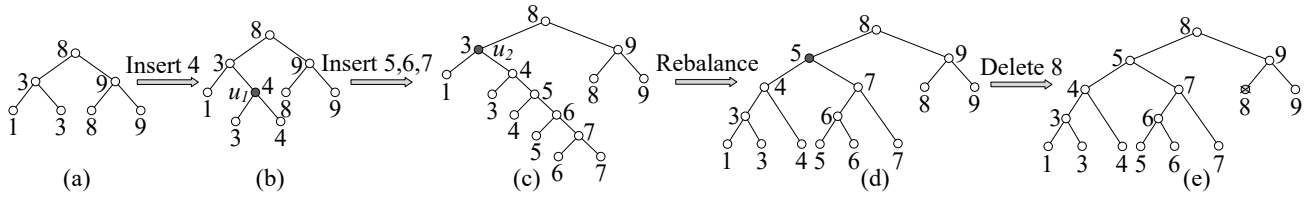


Figure 4: An example of the insertion/deletion with WSS-augmented BST ($\alpha = 0.8$).

Algorithm 7: Dynamic-WIRS-Deletion($\mathcal{T}, e$)

```
1 u ← root(T);
2 while u.left ≠ ∅ do
3 |   Delete e from the WSS structure W(u) at node u;
4 |   if e.key < u.key then u = u.left;
5 |   else u = u.right;
6 end
7 Mark u as invalid, c_i += 1;
8 if 2 · c_i > size(root(T)) then Use Algo. 4 to reconstruct T;
9 return root(T);
```

afford the insertion of $O(size(u))$ nodes. Besides, each insertion affects $O(\log n)$ internal nodes that need to be charged. It then bounds the amortized insertion cost by $O(\log^2 n)$. □

Note that the insertion cost of this basic solution is $O(\log^2 n)$. We will see later in Section 4.2 that by including the chunk idea, we can reduce the insertion cost to $O(\log n)$.

Deletion with dynamic WIRS index. Algo. 7 shows the pseudo-code of dealing with deletions. When deleting an element e from S, we first find the leaf u that holds e (Lines 1-6). Then we mark u as invalid but do not delete u (Line 7). At the same time, we record the number c_i of invalid nodes in the tree. This delete operation has no effect on the process of insertion because it does not affect the true size of leaf nodes of the left subtree and right subtree of each node. If an existing node containing e is marked as invalid while inserting e, we only need to mark it as valid. Obviously, c_i can be maintained in $O(1)$ time. When $2 \cdot c_i > size(root(\mathcal{T}))$, we directly reconstruct the entire tree (Line 8). Note that $size(u)$ counts the number of both invalid and valid leaf nodes. Meanwhile, the number of valid leaf nodes is at least half of the total number of elements in the tree, so the time complexity of insertion is not affected.

Lemma 4.7. *$\mathcal{T}$ handles each deletion in $O(\log n)$ amortized time.*

Proof. For deletion, it takes $O(\log n)$ time to update the WSS structure of affected internal nodes. Besides, as we reconstruct the tree with $O(n \log n)$ time only after deleting $O(n)$ data points, the amortized update cost per deletion is $O(\log n)$. □

The cost that each deletion charges is lower than that of insertion. To explain, we only rebuild the tree after $O(n)$ deletions while each insertion may cause some subtrees to rebuild. This reduces the amortized update cost for deletion to $O(\log n)$. Next, we give an example of how to update the index. For brevity, we omit the update of the WSS index scheme at each internal node in the example.

Example 4.8. Given a weighted set $S = \{1, 3, 8, 9\}$, assume that $\alpha = 0.8$ and the initial WSS-augmented BST $\mathcal{T}$ is as shown in Figure 4(a). Firstly, 4 is inserted into S. We find the place where 4 should be inserted via a search on $\mathcal{T}$. Following the search, it should be inserted as the right child of node 3, but note that we will keep all data at leaf nodes. Thus, we replace the leaf node 3 as an internal node u_1 (shown in Fig. 4(b)) and set the key as 4, i.e., the larger of these two. Then, we set 3 and 4 as the children of u_1. As no node is unbalanced, the insertion is done. 5 and 6 are then added to S in order. There are still no unbalanced nodes. Next, 7 is inserted. Fig. 4(c) shows the BST after inserting 7 into the right position. It also records the visited internal nodes with keys 8, 3, 4, 5, 6, and 7 in order. Then, we examine if these nodes are unbalanced following the order. Internal node 3, i.e., u_2 in Figure 4(c), is the first unbalanced since $\max\{size(u_2.left), size(u_2.right)\} = 5 > size(u_2) \cdot \alpha = 4.8$. Thus, we reconstruct the subtree rooted at u_2 by invoking Algorithm 4. The updated tree is shown in Figure 4(d), and the new key of u_2 is 5. Next, 8 is removed from S. Note that we only mark the leaf node corresponding to 8 (crossed in Figure 4(e)) as invalid after deletion, and it does not affect the size of other nodes. Hence, the nodes in the tree are still balanced. Then, the deletion is finished.

4.2 Chunk-based Optimization

Finally, we use the chunk solution [42] to reduce the space cost from $O(n \log n)$ to $O(n)$. In [42], they do not deal with updates. Here we further show how to handle updates with the chunk solution and will see that it can help reduce the amortized update time.

Index Structure. The main idea of the chunk solution is to divide the data elements into $g = \Theta(n/\log n)$ different partitions of size $c_{size} = \Theta(\log n)$, called chunks, so that each chunk $\mathcal{C}_i$ corresponds to the set of data points falling into an interval $[\ell_i, r_i)$. The chunks are divided so that the intervals of all chunks are disjoint and the union of all chunks is exactly set S. The weight $w(\mathcal{C})$ of a chunk $\mathcal{C}$ is the sum of the weights of all elements in this chunk.

Then, we apply the solution in Sec. 4.1 on the derived chunks. Notice that the key of a chunk is the smallest key of the elements in this chunk to support range queries and handle splits/merges of chunks caused by insertion/deletion efficiently. We build a $\mathcal{T}_{\mathcal{C}}$ for the set $\{\mathcal{C}_1, \ldots, \mathcal{C}_g\}$ of chunks, where the weight of $\mathcal{C}_i$ is $w(\mathcal{C}_i)$. Next, we construct an alias structure for each chunk to guarantee that a sample within this chunk can be drawn with $O(1)$ time.

Lemma 4.9. *The space cost of the WIRS structures proposed in Section 4.1 can be bounded in $O(n)$ if they are built on chunks.*

It is easy to derive the above result as for the WIRS structure, the space of $\mathcal{T}_{\mathcal{C}}$ is $O(g \log g)$ given g chunks. As we build the index on chunks, the number of chunks is $\Theta(n/\log n)$. Replacing g with $\Theta(n/\log n)$, the space is bounded by $O(n)$. Besides, the space of the sampling structure to be maintained for each chunk is linear, so the total space cost is bounded by $O(n)$.

Sampling for WIRS. To draw a sample, we first invoke the sampling method in Sec. 4.1 to sample a chunk. Assume that a chunk $\mathcal{C}$ is sampled. Within chunk $\mathcal{C}$, we then use the maintained alias structure to draw a sample. It should be noted that the chunks containing the left and right endpoints of the range Q may not be fully contained in Q. In this case, we need to handle this part of the elements separately. Since at most two chunks are not all included in the query interval and each chunk has only $\Theta(\log n)$ elements, an alias structure for these two chunks can be created in time $\Theta(\log n)$, thus not affecting the final sample complexity. Before each sampling, it can easily determine if the sample is drawn from these two chunks or not by generating a random number. Thus, the total sampling time is unaffected, and we have the same sample complexity shown in Theorem 4.5.

To further reduce the space cost, we observe that the chunks are loaded into consecutive cache lines, and thus the sampling time will not degrade even if we directly scan the chunk to do the sampling. In particular, we first draw a random number $x \in [0, w(\mathcal{C})]$. Next, we get the smallest index i such that $\sum_{i=1}^{|\mathcal{C}|} w(a_i) > x$ and return a_i in chunk $\mathcal{C}$. With this strategy, we can avoid the alias structures for such chunks. Note that we only do this when the size is smaller than a constant τ_{chunk}. Thus the sampling cost at each chunk is still $O(1)$. With such a strategy, the space cost is reduced by more than half without affecting the sampling performance, showing a good trade-off between the space cost and sampling efficiency.

Update. To insert an element e, we first locate the chunk $\mathcal{C}_i$ where e is to be inserted. We append it to chunk $\mathcal{C}_i$ with $O(\log n)$ cost. When $|\mathcal{C}_i| < 2c_{size}$ after insertion, where c_{size} is the chunk size, $\mathcal{C}_i$ does not need to be split. We can modify the corresponding chunk to a new weight on $\mathcal{T}_{\mathcal{C}}$. When $|\mathcal{C}_i| \geq 2c_{size}$, we split $\mathcal{C}_i$ into two new chunks $\mathcal{C}'_1, \mathcal{C}'_2$ and build the alias structure for them if their size is no smaller than τ_{chunk}. Then, we delete the original $\mathcal{C}_i$ in $\mathcal{T}_{\mathcal{C}}$ and add two new chunks $\mathcal{C}'_1, \mathcal{C}'_2$ to $\mathcal{T}_{\mathcal{C}}$ using the insertion and deletion algorithm presented in Sec. 4.1.

To delete an element e, we first locate the chunk $\mathcal{C}_i$ that contains e and delete e. If $|\mathcal{C}_i| \times 2 > c_{size}$ after deletion, we only update the new weight on $\mathcal{T}_{\mathcal{C}}$. If $|\mathcal{C}_i| \times 2 \leq c_{size}$, we insert the entire chunk $\mathcal{C}_i$ into the predecessor chunk or successor chunk, denoted as $\mathcal{C}_{merge}$. If this operation causes the merged chunk $\mathcal{C}_{merge}$ to be split into two chunks, the same operation is performed for the split as we have done in the insertion. We must also delete the original chunk $\mathcal{C}_i$ in $\mathcal{T}_{\mathcal{C}}$. Finally, the alias structure is constructed for the modified chunks if the chunk size is no smaller than τ_{chunk}.

Moreover, we observe that most insertions/deletions of elements will not create a new chunk. Instead, it only inserts (resp. deletes) the element e into (resp. from) an existing chunk C and increments (resp. decrements) the weight $w(C)$ by $w(e)$. Then, for the whole path from the root to the updated chunk, the sampling structure at each node along the path only has a weight update and no insertions or deletions. In previous BUS scheme, we only provide insertion and deletion operations, which will result in unnecessary update costs if we apply a deletion followed by an insertion to do the update. Since the increment of the weight usually does not significantly change the weight of the updated chunk C, it will not change the bucket that chunk C falls into. Thus, we only need to update the weight of the chunk inside the bucket, improving the practical performance. The optimization reduces the update time by almost half.

Theorem 4.10. *Combining the chunk structure with WIRS-OPT (resp. WIRS-BUS), it handles an insertion/deletion in $O(\log n)$ amortized time. It draws t independent samples with $O(\log n + t)$ (resp. $O(\log^2 n + t)$) time. The space cost of both methods is linear.*

Proof. When the sampling scheme of $\mathcal{T}_{\mathcal{C}}$ is OPT or BUS, inserting an element needs to find the corresponding chunk $\mathcal{C}^*$ to be inserted with $O(\log g)$ time, where $g = \Theta(n/\log n)$. First, consider the case where $\mathcal{C}^*$ does not need to be split. It takes $O(1)$ time to modify an affected sample structure for a node in $\mathcal{T}_{\mathcal{C}}$. Since there are $O(\log g) = O(\log n)$ affected nodes, the update cost is $O(\log n)$. Then we must rebuild the alias structure of $\mathcal{C}^*$, which takes $O(c_{size}) = O(\log n)$ time. As it does not affect the size of any node in the tree, no reconstruction is triggered. Next, consider the case when $\mathcal{C}^*$ needs to be split. We spend $O(c_{size})$ time to split C^*. It will insert a new node in $\mathcal{T}_{\mathcal{C}}$. The amortized insertion time is $O(\log^2 g)$ for this chunk. However, notice that for a chunk that gets split, at least c_{size} elements are inserted into C^*. So this $O(\log^2 g)$ update cost for the chunk can be further amortized by these c_{size} elements. Then, each insertion only incurs $O(\log n)$ amortized update time with OPT or BUS as the sampling scheme. For deletion, it is already $O(\log n)$ and cannot be further reduced since finding the element to be deleted already takes $O(\log n)$ time. To ensure the size of a chunk always stays in $\Theta(\log n)$, whenever n doubles or reduces to half, we may rebuild the structure and reset c_{chunk}. This still charges $O(\log n)$ amortized update time. We can apply Theorem 4.5 and Lemma 4.9 for the sampling time and space cost. □

5 EXPERIMENTS

We experimentally evaluate our proposed solutions against alternative dynamic solutions for WSS and WIRS problems. All experiments are conducted on a Linux machine with an Intel Xeon(R) CPU with 256GB memory. All methods are implemented in C++ and compiled with full optimization. Our code is available at [5].

Datasets. We used the following three real-world datasets tested in existing studies of weighted independent range sampling [45]: *(i)* USA [2], which contains around 24 million road junctions of the whole USA road network and each road junction is assigned a weight that summarizes the length of its connected road segment; *(ii)* Deli [3], which contains 38 million items collected by Delicious and the weight of each item is the number of tags on this item; *(iii)* Twitter [1], which includes 41 million users and the weight of each user is the number of his/her followers. In addition, we generate two synthetic datasets with two different distributions: *(iv)* Uniform, which contains 10^8 elements where the weight of each element is uniformly sampled from the range $[0, 10^7]$; *(v)* Exponential, which includes 10^8 elements where the weight of each element is sampled from an exponential distribution with rate parameter $\lambda = 1/1000$.

Main competitors. We compare our solution against existing dynamic solutions on WSS and WIRS. For WSS, we compare our BUS against the BST solution for WSS (Ref. to Sec. 2.2), dubbed as BST. For sampling performance, we also test the state-of-the-art static method alias method [43], dubbed as Alias, for reference. The OPT method is only for theoretical interest and cannot work on

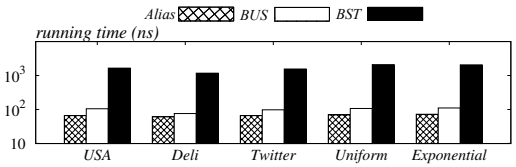


Figure 5: WSS: Query Performance

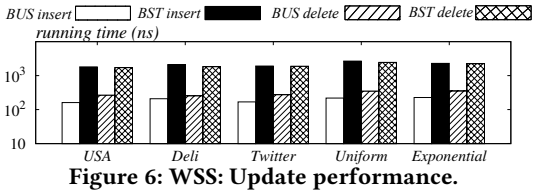


Figure 6: WSS: Update performance.

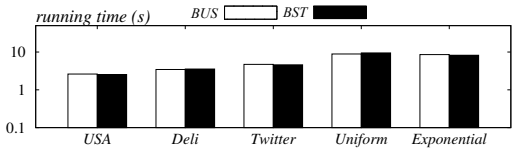


Figure 7: WSS: Index construction time.

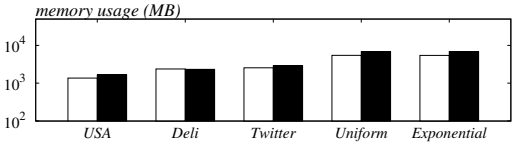
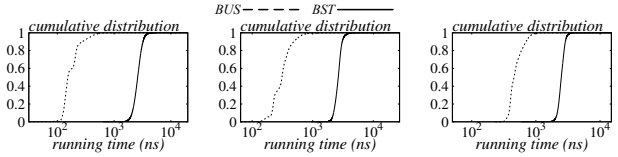


Figure 8: WSS: Memory usage.



(a) Sampling (b) Insertion (c) Deletion

Figure 9: WSS: Time cost distribution on Exponential dataset.

large datasets, as we explained in Section 2.2, and we thus omit it in the experiments. For the WIRS problem, our solution, dubbed as WIRS-BUS, combines *(i)* our proposed weight-balancing BST via reconstruction, *(ii)* our BUS as the WSS structure at each internal node, and *(iii)* the chunk idea to put $\Theta(\log n)$ elements in each leaf node with $c_{size} = 240$ and $\tau_{chunk} = 480$ (Ref. to Sec. 4.2) by default. We omit the basic solution presented in Sec. 4.1 without the chunk idea as it takes too much space (100x more space) and is impractical for large datasets. We also omit WIRS-OPT as it is only for theoretical interest. The main competitor is a dynamic extension of the static solution proposed by Xie et al. [45] as we mentioned in Section 2.3. We denote this method as WIRS-BST. We also compare our solution against the state-of-the-art static method mentioned in Sec. 2.3, dubbed as WIRS-Alias, on sampling performance, as a reference. The chunk settings of WIRS-Alias are the same as ours.

5.1 Weighted Set Sampling

Exp 1: Sampling efficiency on WSS. In the first set of experiments, we examine the sampling performance of our BUS scheme against BST and Alias. Figure 5 shows the results of the average sampling time to draw 10^7 samples on five datasets for the WSS problem. Note that the y-axis is log-scale. As we can see, BUS achieves up to an order of magnitude speedup over the BST method on all datasets,

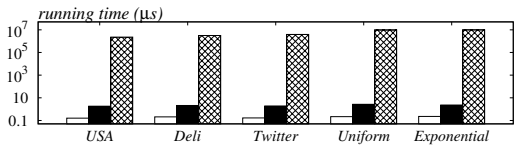


Figure 10: WSS: Insertion performance.

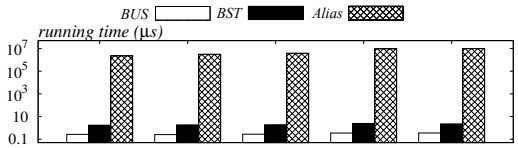


Figure 11: WSS: Deletion performance.

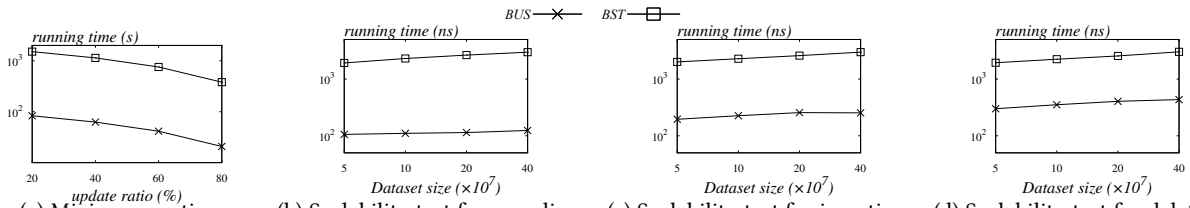
as it reduces the sampling cost from $O(t \log n)$ time to $O(\log n + t)$ expected time. Remarkably, our BUS achieves comparable sampling performance as Alias when Alias is a static structure that must be rebuilt when updates occur while our method supports super efficient updates as we will see shortly. This shows that our BUS gains better a trade-off in sampling efficiency and update efficiency, and is the preferred choice when we have updates.

Exp 2: Update efficiency on WSS. Next, we examine the update performance of BUS and BST. We first randomly delete 10^6 elements and then insert these 10^6 elements back. Figure 6 reports the average time to insert and delete these 10^6 random elements. Note that the y-axis is log-scale. Our BUS is far more efficient than BST for both insertion and deletion, where BUS is up to an order of magnitude faster (resp. 7x faster) than BST on insertion (resp. on deletion). For the static Alias, it needs to rebuild the structure after every update, which is too expensive. The experimental results are shown in Figure 10 and 11, where BUS (resp. BST) is 7 (resp. 6) orders of magnitude faster than Alias (by reconstruction). This shows the high efficiency of our BUS when dealing with updates.

Exp 3: Indexing cost on WSS. Next, we compare the indexing cost of BUS against BST. Fig. 7 reports the indexing time of both methods. The indexing time for BUS and BST are similar. Note that here BST can be built with $O(n)$ cost after sorting the elements thus the index construction can be very efficient. Fig. 8 further reports the memory consumption of both methods. Our BUS takes a similar space cost to that of BST as both methods take linear space.

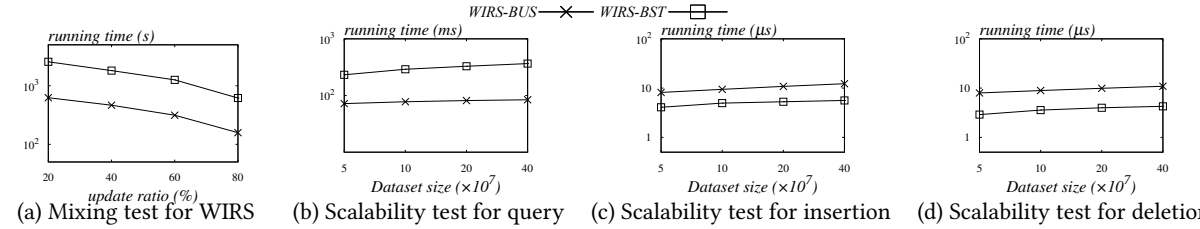
Exp 4: Distribution of sampling and update time on WSS. Fig. 9(a) shows the distribution of sampling time for BUS and BST on the Exponential dataset for drawing 10^6 samples. Fig. 9(b) (resp. Fig. 9(c)) reports the distribution of insertion time (resp. deletion time) for BUS and BST on the Exponential dataset for inserting 10^6 samples (deleting 10^6 elements). As we can see, even though our BUS is based on amortized/expected cost, our solution is still far more efficient than BST for sampling and updates in extreme cases on the Exponential dataset. To conclude, our BUS shows high efficiency for both the average case and tail cases.

Exp 5: Mixing operation and scalability test on WSS. Next, we examine the performance of BUS and BST when we mix sampling queries and updates. Fig. 12(a) shows the performance of both methods under a mixing workload of insertions, deletions, and queries. We vary the update ratio from 20% to 80%, where the update ratio is the fraction of update operations. For example, if the update ratio is 50%, then we have 50% of updates. Insertion and deletion in updates are set as 1:1. Each WSS sampling query draws



(a) Mixing operation (b) Scalability test for sampling (c) Scalability test for insertion (d) Scalability test for deletion

Figure 12: WSS: Changing update ratio and data size



(a) Mixing test for WIRS (b) Scalability test for query (c) Scalability test for insertion (d) Scalability test for deletion

Figure 13: WIRS: Changing update ratio and data size

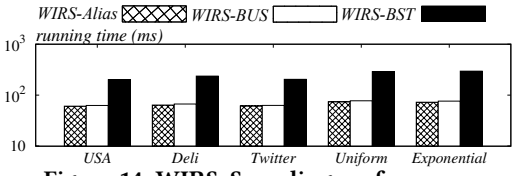


Figure 14: WIRS: Sampling performance.

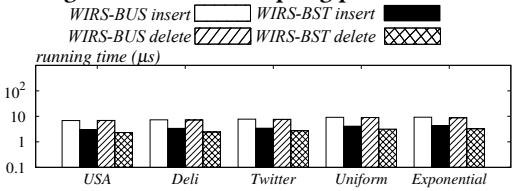


Figure 15: WIRS: Update performance.

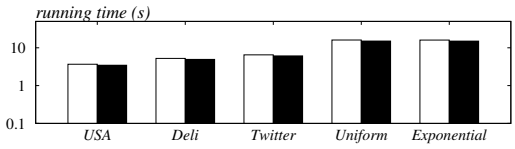


Figure 16: WIRS: Index construction time.

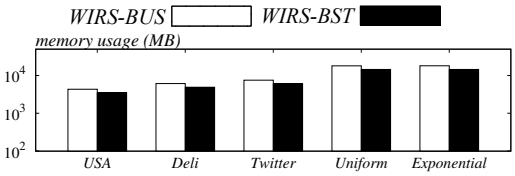
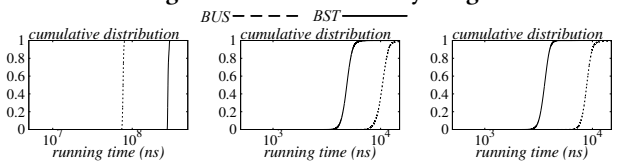


Figure 17: WIRS: Memory usage.



(a) Sampling (b) Insertion (c) Deletion

Figure 18: WIRS: Time cost distribution on Exponential dataset.

10^5 samples. This is due to the fact that in order to answer some statistical queries, we generally need to draw multiple samples to get reliable results. As we can observe from Fig. 12, BUS shows a stable superiority against BST under all tested update ratios.

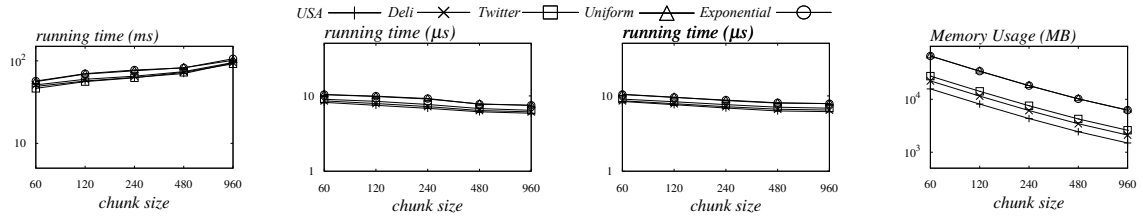
We then test the scalability of both methods. We choose the synthetic *Exponential* dataset and vary the set size from 5×10^7 to 4×10^8. The results are shown in Figs. 12 (b)-(d). We can see that BUS significantly outperforms BST under all tested scales. With the increase of the data scale, the advantage of BUS becomes more significant compared to BST for both sampling and updates. This shows that our BUS gains better scalability than the BST method.

5.2 Weighted Independent Range Sampling

Exp 6: Sampling efficiency on WIRS. In this set of experiments, we examine the sampling performance of WIRS-BUS, WIRS-BST, and WIRS-Alias on the WIRS problem. We generate 10^3 WIRS queries where each range covers 50% of elements in the input set S. We will examine the performance with varying coverage later. For each WIRS query, we draw 10^5 independent samples. Figure 14 reports the average sampling time for these 10^3 WIRS queries where each query generates 10^5 samples. As we can observe, WIRS-BUS is up to 5x faster than BST to handle WIRS queries since WIRS-BUS requires $O(\log^2 n+t)$ time while WIRS-BST requires $O(t \log n)$ time to draw t samples. Remarkably, our WIRS-BUS achieves almost identical performance as WIRS-Alias, while WIRS-Alias is a static index and needs to be rebuilt when any update occurs. That means our WIRS-BUS gains high update efficiency (as to be shown later) without compromising the sampling efficiency.

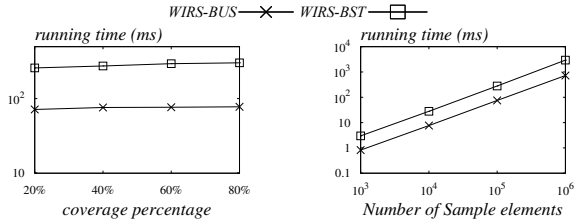
Exp 7: Update efficiency on WIRS. Figure 15 reports the insertion and deletion times of WIRS-BUS and WIRS-BST. Recap that the WIRS-BST has $O(\log n)$ update time and only needs to update the BST maintained. Our WIRS-BUS has the same amortized update time and has more complicated WSS sample structures maintained at each internal node of the BST. Thus, WIRS-BUS is expected to incur a higher practical update cost. This is verified in our experiments where WIRS-BUS is slightly slower than WIRS-BST. However, in most scenarios, queries are more frequent than updates. Our WIRS-BUS still achieves a good trade-off between the query and update efficiency and is the preferred choice when sampling queries are more frequent than updates.

Exp 8: Indexing cost on WIRS. We also examine the indexing time and memory cost of WIRS-BUS and WIRS-BST. Figure 16 and



(a) Query time (b) Insertion time (c) Deletion time (d) Memory Usage

Figure 19: Impact of chunk size on all datasets



(a) Impact of query range (b) Impact of number of samples

Figure 20: Impact of sample parameters to query time on Exponential dataset.

17 show the indexing time and memory usage of both methods. The observation is that both methods have a similar indexing time and memory cost since both methods take $O(n \cdot \log n)$ preprocessing time and $O(n)$ space. To get the best trade-off between memory usage and sampling time, we further conduct experiments on the influence of parameter c_{size} on performance in Exp 11.

Exp 9: Distribution of sampling and update time on WIRS. Figure 18 shows the time cost distribution of both methods of sampling and updates on the Exponential dataset. The parameters of the query are the same as Exp 6. In terms of sampling, we can acquire similar conclusions to the WSS sampling distribution test of Exp 4. In terms of updating, WIRS-BUS achieves a comparable update cost as WIRS-BST in both the average and tail cases.

Exp 10: Mixing operation and scalability test on WIRS. Next, we examine the performance of WIRS-BUS and WIRS-BST when we mix sampling queries and updates. We still vary the update ratio (Ref. to Exp 5 for the definition) from 20% to 80% and set insertion and deletion to 1:1. Figure 13(a) shows the performance of both methods under a mixing workload of updates and queries. The parameters of each query are the same as those in Exp 6. We can find that WIRS-BUS performs significantly better than WIRS-BST for any update ratio. To explain, the sampling query is in the order of *ms*, while updates are in the order of μs. Thus, our WIRS-BUS outperforms WIRS-BST in all tested update ratios.

We further test the scalability of WIRS-BUS and WIRS-BST. We still use the Exponential dataset and change the size from 5×10^7 to 4×10^8. The results are shown in Figs. 13 (b)-(d). For WIRS query time, WIRS-BUS outperforms WIRS-BST in all scales. As the size of the dataset increases, the advantage of WIRS-BUS over WIRS-BST becomes more significant. For update cost, WIRS-BUS and WIRS-BST exhibit similar performance as that in Exp 7 on all scales. The results show the excellent scalability of our WIRS-BUS.

Exp 11: Impact of parameters. We further do experiments to examine the impact of the chunk size c_{size} (Ref. to Section 4.2) on our WIRS-BUS and set τ_{chunk} accordingly. Figure 19 shows the performance of various aspects of WIRS-BUS for all data sets with different chunk size parameters. Here, we intend not to construct the alias structure for each chunk to find the proper choice of τ_{chunk}, the threshold to control when we should build the alias structure. We can see that when we increase the size of the chunk, the query time rises slightly because the chunk is loaded into the main memory with high locality, and the main cost is the memory access to the chunk. However, the space cost decreases significantly as the chunk size increases because a larger chunk size represents a smaller number of nodes maintained by the BST in the WIRS index, so the space cost of maintaining the WIRS index is smaller. In terms of update time, it also decreases with the size of chunks, which is expected since the larger the chunk, the fewer nodes are maintained by the BST of the WIRS index. Therefore, for the chunk size c_{size}, we set it to 240 in our experiments, which has a small impact on query performance but can save a lot of space. Accordingly, we set τ_{chunk} to twice the c_{size}, which is 480, and thus we can directly scan all data in a chunk with high locality.

We also have experiments to examine the impact of the range and the impact of sample number to the WIRS query processing on Exponential dataset. Figure 20(a) shows the experimental result when we vary the coverage percentage of the query range. A coverage percentage of 20% indicates that the query covers 20% of elements in the input set S. Both methods generally perform consistently when we vary the range from 20% to 80%. We further examine the impact of the number of samples for each WIRS query. Figure 20(b) shows the performance of all methods when we increase the sample number t from 10^3 to 10^6. All methods incur a higher running cost when the number of samples increases, which is expected as all methods linearly depend on t. Based on the above results, we can find that WIRS-BUS is 5x faster than WIRS-BST for all tested query parameters, which is the same as the result of the Exp 6 query performance comparison under the default parameters.

6 CONCLUSIONS

This paper studies the WSS and WIRS problems. For WSS, we propose BUS, which supports $O(1)$ amortized update time and can draw t samples with $O(\log n + t)$ expected sampling time. The idea is a simplified single-level bucket structure with a carefully designed sampling algorithm. Then, we further present WIRS-BUS, by extending BUS to the WIRS problem. WIRS-BUS can draw t samples in $O(\log^2 n + t)$ time and handle an update in $O(\log n)$ time. We also present WIRS-OPT that can improve the sampling time to $O(\log n + t)$, which is mainly for theoretical interest. Experiments show the effectiveness of the proposed BUS and WIRS-BUS. For future work, we plan to study the dynamic WSS and WIRS problems in external memory setting. Designing more efficient dynamic WIRS schemes in spatial databases is also an interesting direction.

REFERENCES

[1] 2010. Twitter. https://anlab-kaist.github.io/traces/.

[2] 2010. USA Road Networks. http://users.diag.uniroma1.it/challenge9/download.shtml.

[3] 2013. Delicious. http://delicious.com/.

[4] 2019. Parallel Weighted Random Sampling. In *ESA*, Michael A. Bender, Ola Svensson, and Grzegorz Herman (Eds.), Vol. 144. 59:1–59:24.

[5] 2023. Experiment code and technical report. https://github.com/zzzzzfy/DynamicWeightedSetSampling.

[6] Peyman Afshani and Jeff M. Phillips. 2019. Independent Range Sampling, Revisited Again. In *SoCG*, Vol. 129. 4:1–4:13.

[7] Peyman Afshani and Zhewei Wei. 2017. Independent Range Sampling, Revisited. In *ESA*, Vol. 87. 3:1–3:14.

[8] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*. 29–42.

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

[10] 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 Trans. Database Syst.* 47, 1 (2022), 4:1–4:40.

[11] Martin Aumüller, Rasmus Pagh, and Francesco Silvestri. 2020. Fair Near Neighbor Search: Independent Range Sampling in High Dimensions. In *PODS*. 191–204.

[12] Maham Anwar Beg, Muhammad Ahmad, Arif Zaman, and Imdadullah Khan. 2018. Scalable Approximation Algorithm for Graph Summarization. In *PAKDD*, Vol. 10939. 502–514.

[13] Pawel Brach, Alessandro Epasto, Alessandro Panconesi, and Piotr Sankowski. 2014. Spreading rumours without the network. In *COSN*. 107–118.

[14] Vladimir Braverman, Rafail Ostrovsky, and Gregory Vorsanger. 2015. Weighted sampling without replacement from data streams. *Inf. Process. Lett.* 115, 12 (2015), 923–926.

[15] Karl Bringmann and Kasper Green Larsen. 2013. Succinct sampling from discrete distributions. In *STOC*. 775–782.

[16] Surajit Chaudhuri, Gautam Das, Mayur Datar, Rajeev Motwani, and Vivek R. Narasayya. 2001. Overcoming Limitations of Sampling for Aggregation Queries. In *ICDE*. 534–542.

[17] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. 1998. Random Sampling for Histogram Construction: How much is enough?. In *SIGMOD*. 436–447.

[18] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. 1999. On Random Sampling over Joins. In *SIGMOD*. 263–274.

[19] Joshua Colvin, Michael I Monine, Ryan N Gutenkunst, William S Hlavacek, Daniel D Von Hoff, and Richard G Posner. 2010. RuleMonkey: software for stochastic simulation of rule-based models. *BMC bioinformatics* 11, 1 (2010), 1–14.

[20] Teresa Maria Creanza, Giuseppe Lamanna, Pietro Delre, Marialessandra Contino, Nicola Corriero, Michele Saviano, Giuseppe Felice Mangiatordi, and Nicola Ancona. 2022. DeLA-Drug: A Deep Learning Algorithm for Automated Design of Druglike Analogues. *Journal of Chemical Information and Modeling* 62, 6 (2022), 1411–1424.

[21] Pavlos S. Efraimidis. 2015. Weighted Random Sampling over Data Streams. In *Algorithms, Probability, Networks, and Games*. 183–195.

[22] Igal Galperin and Ronald L. Rivest. 1993. Scapegoat Trees. In *SODA*, Vijaya Ramachandran (Ed.). 165–174.

[23] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. WTF: the who to follow service at Twitter. In *WWW*. 505–514.

[24] Torben Hagerup, Kurt Mehlhorn, and J. Ian Munro. 1993. Maintaining Discrete Probability Distributions Optimally. In *ICALP*, Vol. 700. 253–264.

[25] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. 2018. Distributed Prioritized Experience Replay. In *ICLR*.

[26] Xiaocheng Hu, Miao Qiao, and Yufei Tao. 2014. Independent range sampling. In *PODS*. 246–255.

[27] Lorenz Hübschle-Schneider and Peter Sanders. 2020. Communication-Efficient Weighted Reservoir Sampling from Fully Distributed Data Streams. In *SPAA*. ACM, 543–545.

[28] Rajesh Jayaram, Gokarna Sharma, Srikanta Tirthapura, and David P. Woodruff. 2019. Weighted Reservoir Sampling from Distributed Streams. In *PODS*. 218–235.

[29] Marc Langheinrich, Atsuyoshi Nakamura, Naoki Abe, Tomonari Kamba, and Yoshiyuki Koseki. 1999. Unintrusive Customization Techniques for Web Advertising. *Comput. Networks* 31, 11-16 (1999), 1259–1272.

[30] Long Ji Lin. 1992. Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching. *Mach. Learn.* 8 (1992), 293–321.

[31] Wenqing Lin. 2019. Distributed Algorithms for Fully Personalized PageRank on Large Graphs. In *WWW*. 1084–1094.

[32] Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. 2003. Dynamic Generation of Discrete Random Variates. *Theory Comput. Syst.* 36, 4 (2003), 329–358.

[33] Mohammad Najafi, Sarah Taghavi Namin, Mathieu Salzmann, and Lars Petersson. 2016. Sample and Filter: Nonparametric Scene Parsing via Efficient Filtering. In *CVPR*. IEEE Computer Society, 607–615.

[34] Jürg Nievergelt and Edward M. Reingold. 1973. Binary Search Trees of Bounded Balance. *SIAM J. Comput.* 2, 1 (1973), 33–43.

[35] Frank Olken and Doron Rotem. 1986. Simple Random Sampling from Relational Databases. In *VLDB*. 160–169.

[36] Frank Olken and Doron Rotem. 1993. Sampling from Spatial Databases. In *ICDE*. 199–208.

[37] Frank Olken and Doron Rotem. 1995. Random sampling from databases: a survey. *Statistics and Computing* 5, 1 (1995), 25–42.

[38] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[39] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algorithms* 51, 2 (2004), 122–144.

[40] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized Experience Replay. In *ICLR*.

[41] Jürgen Schmidt, Rihab Laarousi, Wolfgang Stolzmann, and Katja Karrer-Gauß. 2018. Eye blink detection for different driver states in conditionally automated driving and manual driving using EOG and a driver camera. *Behavior research methods* 50 (2018), 1088–1101.

[42] Yufei Tao. 2022. Algorithmic Techniques for Independent Query Sampling. In *PODS*. 129–138.

[43] Alastair J. Walker. 1977. An Efficient Method for Generating Discrete Random Variables with General Distributions. *ACM Trans. Math. Softw.* 3, 3 (1977), 253–256.

[44] Lu Wang, Robert Christensen, Feifei Li, and Ke Yi. 2015. Spatial Online Sampling and Aggregation. *Proc. VLDB Endow.* 9, 3 (2015), 84–95.

[45] Dong Xie, Jeff M. Phillips, Michael Matheny, and Feifei Li. 2021. Spatial Independent Range Sampling. In *SIGMOD*. 2023–2035.

[46] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *SIGMOD*. ACM, 1525–1539.